Overview

- Introduction
- Demo
- Background
- Implementation
- Results
- Short Review
Introduction

Interactive tool for segmentation and composition

Uses “live-wire boundary” to find edges

Boundary detection using graph search (similar to Dijkstra) and dynamic programming (DP)

Uses innovative features to increase robustness: on-the-fly training, boundary cooling

Scale, rotate, and composite using masks and spatial frequency equivalencing.
Wish you were here?... Volunteers?
• Improvement on manual tracing
• Contemporary technologies not interactive
  ◦ Magic wands
  ◦ Snakes
  ◦ Graph search using DP to find globally optimal boundaries (using template specification)
• Most important feature compared to other technologies: interactivity.
• Previous approaches not a computationally efficient, nor do they incorporate training or cooling.
Graph search problem: goal to find optimal path between start and set of goal nodes (pixels = nodes).

For this implementation: optimality defined as minimum cumulative cost path.

For minimum cost to correlate to image boundary, pixels with strong edge features have to have low local cost.
• So cost is determined using 3 components: Laplacian Zero Crossing ($f_z$), Gradient Magnitude ($f_g$), and Gradient Direction ($f_d$). Cost is then calculated using a weighted average. (Weights where $0.43 = w_z$, $0.43 = w_d$, and $0.14 = w_g$ seem to work well on many images)

• Laplacian zero-crossing gives low values for strong and weak edges. Therefore, have to use a combination that also takes into account gradient magnitude and direction.

\[ l(p, q) = \omega_z \cdot f_z(q) + \omega_d \cdot f_d(p, q) + \omega_g \cdot f_g(q) \]
Implementation (benefits)

- Gradient direction associates high costs with sharp boundary changes, which adds a smoothing element to the formula.
- With formula for determining cost, then a 2D dynamic programming algorithm is used.
- This implementation better than past implementations for 4 reasons
  - No directional sampling or searching constraints
  - Uses laplacian zero-crossing
  - Sorted in $O(n)$ time for $N$ nodes.
  - No goal nodes/pixels set beforehand.
Algorithm: Live-Wire 2-D DP graph search.

Input:
\[ s \quad \text{(Start (or seed) pixel.)} \]
\[ l(q, r) \quad \text{(Local cost function for link between pixels q and r.)} \]

Data Structures:
\[ L \quad \text{(List of active pixels sorted by total cost (initially empty).)} \]
\[ N(q) \quad \text{(Neighborhood set of q (contains 8 neighbors of pixel).)} \]
\[ e(q) \quad \text{(Boolean function indicating if q has been expanded/processed.)} \]
\[ g(q) \quad \text{(Total cost function from seed point to q.)} \]

Output:
\[ p \quad \text{(Pointers from each pixel indicating the minimum cost path.)} \]

Algorithm:
\[ g(s) \leftarrow 0; \quad L \leftarrow s; \quad \text{Initialize active list with zero cost seed pixel.} \]
\[ \text{while } L \neq \emptyset \text{ do begin} \]
\[ q \leftarrow \min(L); \quad \text{Remove minimum cost pixel q from active list.} \]
\[ e(q) \leftarrow \text{TRUE; Mark q as expanded (i.e., processed).} \]
\[ \text{for each } r \in N(q) \text{ such that not } e(r) \text{ do begin} \]
\[ g_{\text{tmp}} \leftarrow g(q) + l(q, r); \quad \text{Compute total cost to neighbor.} \]
\[ \text{if } r \in L \text{ and } g_{\text{tmp}} < g(r) \text{ then} \]
\[ r \leftarrow L; \quad \text{Remove higher cost neighbor’s} \]
\[ \text{from list.} \]
\[ \text{if } r \notin L \text{ then begin} \]
\[ g(r) \leftarrow g_{\text{tmp}}; \quad \text{If neighbor not on list,} \]
\[ p(r) \leftarrow q; \quad \text{assign neighbor’s total cost,} \]
\[ L \leftarrow r; \quad \text{set (or reset) back pointer,} \]
\[ \text{and place on (or return to)} \]
\[ \text{active list.} \]
\[ \text{end} \]
\[ \text{end} \]
\[ \text{end} \]
• Optimal paths computed from seed point to all points in the image (creates a minimum cost spanning tree.
• The mouse curser is a “free point” that interactively chooses a path from many optimal paths.
• Lightning analogy
Implementation (seed)

- Automatic seed point placing can be customized from 1x1 pixels wide to 15x15. (providing a “snap” of up to 7 pixels.)
- Minimum of two seed points needed
- Simple objects 2 to 5 seed points
- Complex – many more
- Automatic uses “path cooling”
- Uses Bellman’s Principle of Optimality
- Dynamic training
- Uses some precomputed values
- Considers only most recent pixels within a specified time $t$ for deciding trained gradient magnitude cost function
- Place image in buffer after finding edges
- Transformed (rotate, scale, translate) using interactive tool.
- Transformation done via 2D transformation matrix (sound familiar?)
• Spatial frequency (contrast) needs to be matched when pasting into another image.
• Use low pass filter to match the spectrum of the target with the source. (Butterworth low-pass filter for those interested)
• Faster and more accurate for people to use this method than doing it manually (big surprise.)
• (next page for graphs)
Boundary Definition Time

Legend
- Live-Wire
- Hand Traced

Average Time (in seconds)

Object
- Polygon
- Curve
- Holder
- Knife
- Spine

Accuracy

Percentage of Pixels ≤ Distance from “Ideal” Boundary

Legend
- Live-Wire
- Hand Traced

Distance (in Pixels)
• Well written paper with good examples
• At least one error in their algorithm (I think)
• Interesting to see this technology in a modern product, such as Photoshop.
• And oh yeah, if you don’t think the results are “convincing...”
See... no people.
Result of Demo in class.