Intelligent Scissors for Image Composition

Eric N. Mortensen[1] William A. Barrett[2]
Brigham Young University

Abstract

We present a new, interactive tool called *Intelligent Scissors* which we use for image segmentation and composition. Fully automated segmentation is an unsolved problem, while manual tracing is inaccurate and laboriously unacceptable. However, Intelligent Scissors allow objects within digital images to be extracted quickly and accurately using simple gesture motions with a mouse. When the gestured mouse position comes in proximity to an object edge, a *live-wire boundary* "snaps" to, and wraps around the object of interest.

Live-wire boundary detection formulates discrete dynamic programming (DP) as a two-dimensional graph searching problem. DP provides mathematically optimal boundaries while greatly reducing sensitivity to local noise or other intervening structures. Robustness is further enhanced with *on-the-fly training* which causes the boundary to adhere to the specific type of edge currently being followed, rather than simply the strongest edge in the neighborhood. *Boundary cooling* automatically freezes unchanging segments and automates input of additional seed points. Cooling also allows the user to be much more free with the gesture path, thereby increasing the efficiency and finesse with which boundaries can be extracted.

Extracted objects can be scaled, rotated, and composited using live-wire masks and *spatial frequency equivalencing*. Frequency equivalencing is performed by applying a Butterworth filter which matches the lowest frequency spectra to all other image components. Intelligent Scissors allow creation of convincing compositions from existing images while dramatically increasing the speed and precision with which objects can be extracted.

1. Introduction

Digital image composition has recently received much attention for special effects in movies and in a variety of desktop applications. In movies, image composition, combined with other digital manipulation techniques, has also been used to realistically blend old film into a new script. The goal of image composition is to combine objects or regions from various still photographs or movie frames to create a seamless, believable, image or image sequence which appears convincing and real. Fig. 9(d) shows a believable composition created by combining objects extracted from three images, Fig. 9(a-c). These objects were digitally extracted and combined in a few minutes using a new, interactive tool called *Intelligent Scissors*.

When using existing images, objects of interest must be extracted and segmented from a surrounding background of unpredictable complexity. Manual segmentation is tedious and time consuming, lacking in precision, and impractical when applied to long image sequences. Further, due to the wide variety of image types and content, most current computer based segmentation techniques are slow, inaccurate, and require significant user input to initialize or control the segmentation process.

This paper describes a new, interactive, digital image segmentation tool called "Intelligent Scissors" which allows rapid object extraction from arbitrarily complex backgrounds. Intelligent Scissors boundary detection formulates discrete dynamic programming (DP) as a two-dimensional graph searching problem. Presented as part of this tool are *boundary cooling* and *on-the-fly training*, which reduce user input and dynamically adapt the tool to specific types of edges. Finally, we present *live-wire masking* and *spatial frequency equivalencing* for convincing image compositions.

2. Background

Digital image segmentation techniques are used to extract image components from their surrounding natural background. However, currently available computer based segmentation tools are typically primitive and often offer little more advantage than manual tracing.

Region based magic wands, provided in many desktop applications, use an interactively selected seed point to "grow" a region by adding adjacent neighboring pixels. Since this type of region growing does not provide interactive visual feedback, resulting region boundaries must usually be edited or modified.

Other popular boundary definition methods use active contours or snakes[1, 5, 8, 15] to improve a manually entered rough approximation. After being initialized with a rough boundary approximation, snakes iteratively adjust the boundary points in parallel in an attempt to minimize an energy functional and achieve an optimal boundary. The energy functional is a combination of internal forces, such as boundary curvature, and external forces, like image gradient magnitude. Snakes can track frame-to-frame boundary motion provided the boundary hasn't moved drastically. However, active contours follow a pattern of initialization followed by energy minimization; as a result, the user does not know what the final boundary will look like when the rough approximation is input. If the resulting boundary is not satisfactory, the process must be repeated or the boundary must be manually edited. We provide a detailed comparison of snakes and Intelligent Scissors in section 3.6.

Another class of image segmentation techniques use a graph searching formulation of DP (or similar concepts) to find globally optimal boundaries [2, 4, 10, 11, 14]. These techniques differ from snakes in that boundary points are generated in a stage-wise optimal cost fashion whereas snakes iteratively minimize an energy functional for all points on a contour in parallel (giving the appearance of wiggling). However, like snakes, these graph searching techniques typically require a boundary template--in the form of a manually entered rough approximation, a figure of merit, etc.--which is used to impose directional sampling and/or searching constraints. This limits these techniques to a boundary search with one degree of freedom within a window about the two-dimensional boundary template. Thus, boundary extraction using previous graph searching techniques is non-interactive (beyond template specification), losing the benefits of further human guidance and expertise.

[1] *enm@cs.byu.edu, Dept. of Comp. Sci., BYU, Provo, UT 84602 (801)378-7605*
[2] *barrett@cs.byu.edu, Dept. of Comp. Sci., BYU, Provo, UT 84602 (801)378-7430*

The most important difference between previous boundary finding techniques and Intelligent Scissors presented here lies not in the boundary defining criteria per se´, but in the *method* of interaction. Namely, previous methods exhibit a pattern of boundary approximation followed by boundary refinement, whereas Intelligent Scissors allow the user to *interactively select* the most suitable boundary from a set of ***all*** optimal boundaries emanating from a seed point. In addition, previous approaches do not incorporate on-the-fly training or cooling, and are not as computationally efficient. Finally, it appears that the problem of automated matching of spatial frequencies for digital image composition has not been addressed previously.

3. Intelligent Scissors

Boundary definition via dynamic programming can be formulated as a graph searching problem [10] where the goal is to find the optimal path between a start node and a set of goal nodes. As applied to image boundary finding, the graph search consists of finding the globally optimal path from a start pixel to a goal pixel--in particular, pixels represent nodes and edges are created between each pixel and its 8 neighbors. For this paper, optimality is defined as the minimum cumulative cost path from a start pixel to a goal pixel where the cumulative cost of a path is the sum of the local edge (or link) costs on the path.

3.1. Local Costs

Since a minimum cost path should correspond to an image component boundary, pixels (or more accurately, links between neighboring pixels) that exhibit strong edge features should have low local costs and vice-versa. Thus, local component costs are created from the various edge features:

Image Feature	Formulation
Laplacian Zero-Crossing	f_Z
Gradient Magnitude	f_G
Gradient Direction	f_D

The local costs are computed as a weighted sum of these component functionals. Letting $l(\boldsymbol{p},\boldsymbol{q})$ represents the local cost on the directed link from pixel $\boldsymbol{p}$ to a neighboring pixel $\boldsymbol{q}$, the local cost function is

$$l(\boldsymbol{p},\boldsymbol{q}) = \omega_Z \cdot f_Z(\boldsymbol{q}) + \omega_G \cdot f_G(\boldsymbol{q}) + \omega_D \cdot f_D(\boldsymbol{p},\boldsymbol{q}) \qquad (1)$$

where each ω is the weight of the corresponding feature function. (Empirically, weights of $\omega_Z = 0.43$, $\omega_G = 0.43$, and $\omega_D = 0.14$ seem to work well in a wide range of images.)

The laplacian zero-crossing is a binary edge feature used for edge localization [7, 9]. Convolution of an image with a laplacian kernel approximates the 2^{nd} partial derivative of the image. The laplacian image zero-crossing corresponds to points of maximal (or minimal) gradient magnitude. Thus, laplacian zero-crossings represent "good" edge properties and should therefore have a low local cost. If $I_L(\boldsymbol{q})$ is the laplacian of an image I at pixel $\boldsymbol{q}$, then

$$f_Z(\boldsymbol{q}) = \begin{cases} 0; & \text{if } I_L(\boldsymbol{q}) = 0 \\ 1; & \text{if } I_L(\boldsymbol{q}) \neq 0 \end{cases} \qquad (2)$$

However, application of a discrete laplacian kernel to a digital image produces very few zero-valued pixels. Rather, a zero-crossing is represented by two neighboring pixels that change from positive to negative. Of the two pixels, the one closest to zero is used to represent the zero-crossing. The resulting feature cost contains single-pixel wide cost "canyons" used for boundary localization.

Since the laplacian zero-crossing creates a binary feature, $f_Z(\boldsymbol{q})$ does not distinguish between strong, high gradient edges and weak, low gradient edges. However, gradient magnitude provides a direct correlation between edge strength and local cost. If I_x and I_y represent the partials of an image I in x and y respectively, then the gradient magnitude G is approximated with

$$G = \sqrt{I_x^2 + I_y^2}. \qquad (3)$$

The gradient is scaled and inverted so high gradients produce low costs and vice-versa. Thus, the gradient component function is

$$f_G = \frac{\max(G) - G}{\max(G)} = 1 - \frac{G}{\max(G)} \qquad (3)$$

giving an inverse linear ramp function. Finally, gradient magnitude costs are scaled by Euclidean distance. To keep the resulting maximum gradient at unity, $f_G(\boldsymbol{q})$ is scaled by 1 if $\boldsymbol{q}$ is a diagonal neighbor to $\boldsymbol{p}$ and by $1/\sqrt{2}$ if $\boldsymbol{q}$ is a horizontal or vertical neighbor.

The gradient direction adds a smoothness constraint to the boundary by associating a high cost for sharp changes in boundary direction. The gradient direction is the unit vector defined by I_x and I_y. Letting $\boldsymbol{D}(\boldsymbol{p})$ be the unit vector perpendicular (rotated 90 degrees clockwise) to the gradient direction at point $\boldsymbol{p}$ (i.e., for $\boldsymbol{D}(\boldsymbol{p}) = (I_y(\boldsymbol{p}), -I_x(\boldsymbol{p}))$), the formulation of the gradient direction feature cost is

$$f_D(\boldsymbol{p},\boldsymbol{q}) = \frac{2}{3\pi}\{\operatorname{acos}[d_{\boldsymbol{p}}(p,q)] + \operatorname{acos}[d_{\boldsymbol{q}}(p,q)]\} \qquad (4)$$

where

$$\begin{aligned} d_{\boldsymbol{p}}(p,q) &= \boldsymbol{D}(\boldsymbol{p}) \cdot \boldsymbol{L}(\boldsymbol{p},\boldsymbol{q}) \\ d_{\boldsymbol{q}}(p,q) &= \boldsymbol{L}(\boldsymbol{p},\boldsymbol{q}) \cdot \boldsymbol{D}(\boldsymbol{q}) \end{aligned} \qquad (3)$$

are vector dot products and

$$\boldsymbol{L}(\boldsymbol{p},\boldsymbol{q}) = \begin{cases} \boldsymbol{q} - \boldsymbol{p}; & \text{if } \boldsymbol{D}(\boldsymbol{p}) \cdot (\boldsymbol{q} - \boldsymbol{p}) \geq 0 \\ \boldsymbol{p} - \boldsymbol{q}; & \text{if } \boldsymbol{D}(\boldsymbol{p}) \cdot (\boldsymbol{q} - \boldsymbol{p}) < 0 \end{cases} \qquad (5)$$

is the bidirectional link or edge vector between pixels $\boldsymbol{p}$ and $\boldsymbol{q}$. Links are either horizontal, vertical, or diagonal (relative to the position of $\boldsymbol{q}$ in $\boldsymbol{p}$'s neighborhood) and point such that the dot product of $\boldsymbol{D}(\boldsymbol{p})$ and $\boldsymbol{L}(\boldsymbol{p}, \boldsymbol{q})$ is positive, as noted in (5). The neighborhood link direction associates a high cost to an edge or link between two pixels that have similar gradient directions but are perpendicular, or near perpendicular, to the link between them. Therefore, the direction feature cost is low when the gradient direction of the two pixels are similar to each other and the link between them.

3.2. Two-Dimensional Dynamic Programming

As mentioned, dynamic programming can be formulated as a directed graph search for an optimal path. This paper utilizes an optimal graph search similar to that presented by Dijkstra [6] and extended by Nilsson [13]; further, this technique builds on and extends previous boundary tracking methods in 4 important ways:

1. It imposes no directional sampling or searching constraints.
2. It utilizes a new set of edge features and costs: laplacian zero-crossing, multiple gradient kernels.
3. The active list is sorted with an O(N) sort for N nodes/pixels.
4. No a priori goal nodes/pixels are specified.

First, formulation of boundary finding as a 2-D graph search eliminates the directed sampling and searching restrictions of previous implementations, thereby allowing boundaries of arbitrary complexity to be extracted. Second, the edge features used here are more robust and comprehensive than previous implementations: we maximize over different gradient kernels sizes to encompass the various edge types and scales while simultaneously attempting to balance edge detail with noise suppression [7], and we use the laplacian zero-crossing for boundary localization and fine detail live-wire "snapping". Third, the discrete, bounded nature of the local edge costs permit the use of a specialized sorting algorithm that inserts points into a sorted list (called the active list) in constant time. Fourth, the live-wire tool is free to define a goal pixel interactively, at any "free" point in the image, after minimum cost paths are computed to ***all*** pixels. The latter happens fast enough that the free point almost always falls within an expanding cost wavefront and interactivity is not impeded.

The Live-Wire 2-D dynamic programming (DP) graph search algorithm is as follows:

Algorithm: Live-Wire 2-D DP graph search.

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

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

Output:
  p                {Pointers from each pixel indicating the minimum cost path.}

Algorithm:
  g(s)←0;  L←s;            {Initialize active list with zero cost seed pixel.}
  while L≠∅ do begin        {While still points to expand:}
    q←min(L);               {Remove minimum cost pixel q from active list.}
    e(q)←TRUE;              {Mark q as expanded (i.e., processed).}
    for each r∈N(q) such that not e(r) do begin
      g_tmp←g(q)+l(q,r);             {Compute total cost to neighbor.}
      if r∈L and g_tmp<g(r) then     {Remove higher cost neighbor's }
        r←L;                         {  from list.}
      if r∉L then begin              {If neighbor not on list, }
        g(r)←g_tmp;                  {  assign neighbor's total cost, }
        p(r)←q;                      {  set (or reset) back pointer, }
        L←r;                         {  and place on (or return to) }
      end                            {  active list.}
    end
  end
```

Notice that since the active list is sorted, when a new, lower cumulative cost is computed for a pixel already on the list then that point must be removed from the list in order to be added back to the list with the new lower cost. Similar to adding a point to the sorted list, this operation is also performed in constant time.

Figure 1 demonstrates the use of the 2-D DP graph search algorithm to create a minimum cumulative cost path map (with corresponding optimal path pointers). Figure 1(a) is the initial local cost map with the seed point circled. For simplicity of demonstration the local costs in this example are pixel based rather than link based and can be thought of as representing the gradient magnitude cost feature. Figure 1(b) shows a portion of the cumulative cost and pointer map after expanding the seed point (with a cumulative cost of zero). Notice how the diagonal local costs have been scaled by Euclidean distance (consistent with the gradient magnitude cost feature described previously). Though complicating the example, weighing by Euclidean distance is necessary to demonstrate that the cumulative costs to points currently on the active list can change if even lower cumulative costs are computed from as yet unexpanded neighbors. This is demonstrated in Figure 1(c) where two points

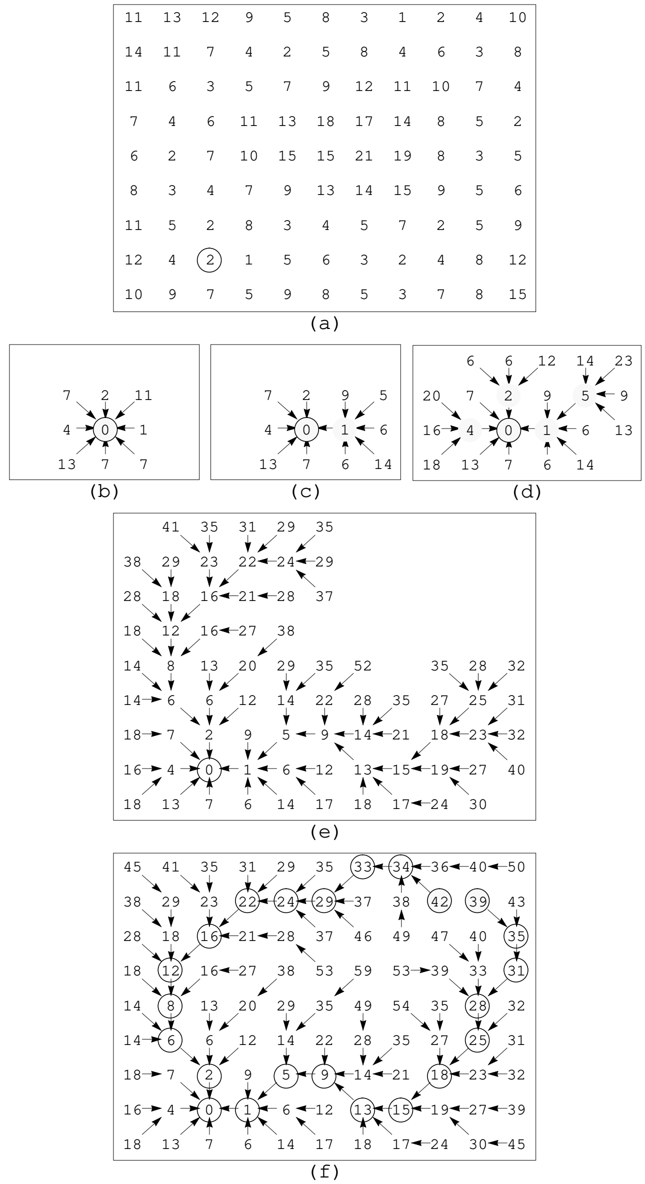


Figure 1: *(a) Initial local cost matrix. (b) Seed point (shaded) expanded. (c) 2 points (shaded) expanded. (d) 5 points (shaded) expanded. (e) 47 points expanded. (f) Finished total cost and path matrix with two of many paths (free points shaded) indicated.*

have now been expanded--the seed point and the next lowest cumulative cost point on the active list. Notice how the points diagonal to the seed point have changed cumulative cost and direction pointers. The Euclidean weighting between the seed and diagonal points makes them more costly than non-diagonal paths. Figures 1(d), 1(e), and 1(f) show the cumulative cost/direction pointer map at various stages of completion. Note how the algorithm produces a "wavefront" of active points emanating from the initial start point, called the seed point, and that the wavefront grows out faster where there are lower costs.

3.3. Interactive "Live-Wire" Segmentation Tool

Once the optimal path pointers are generated, a desired boundary segment can be chosen dynamically via a "free" point. Interactive movement of the free point by the mouse cursor causes the boundary to behave like a live-wire as it adapts to the new minimum cost path by following the optimal path pointers from the free point back

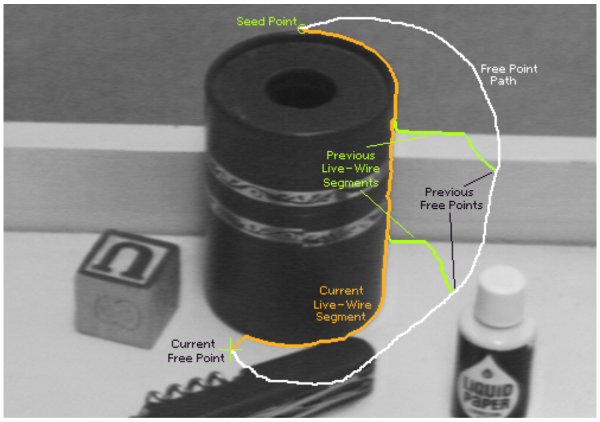


Figure 2: *Image demonstrating how the live-wire segment adapts and snaps to an object boundary as the free point moves (via cursor movement). The path of the free point is shown in white. Live-wire segments from previous free point positions (t_0, t_1, and t_2) are shown in green.*

to the seed point. By constraining the seed point and free points to lie near a given edge, the user is able to interactively "snap" and "wrap" the live-wire boundary around the object of interest. Figure 2 demonstrates how a live-wire boundary segment adapts to changes in the free point (cursor position) by latching onto more and more of an object boundary. Specifically, note the live-wire segments corresponding to user-specified free point positions at times t_0, t_1, and t_2. Although Fig. 2 only shows live-wire segments for three discrete time instances, live-wire segments are actually updated dynamically and interactively (on-the-fly) with each movement of the free point.

When movement of the free point causes the boundary to digress from the desired object edge, interactive input of a new seed point prior to the point of departure reinitiates the 2-D DP boundary detection. This causes potential paths to be recomputed from the new seed point while effectively "tying off" the boundary computed up to the new seed point.

Note again that optimal paths are computed from the seed point to ***all*** points in the image (since the 2-D DP graph search produces a minimum cost spanning tree of the image [6]). Thus, by selecting a free point with the mouse cursor, the interactive live-wire tool is simply selecting an optimal boundary segment from a large collection of optimal paths.

Since each pixel (or free point) defines only one optimal path to a seed point, a minimum of two seed points must be placed to ensure a closed object boundary. The path map from the first seed point of every object is maintained during the course of an object's boundary definition to provide a closing boundary path from the free point. The closing boundary segment from the free point to the first seed point expedites boundary closure.

Placing seed points directly on an object's edge is often difficult and tedious. If a seed point is not localized to an object edge then spikes results on the segmented boundary at those seed points (since

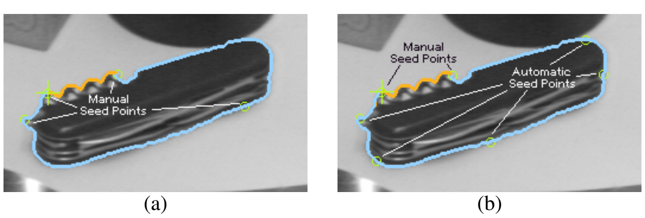


Figure 3: *Comparison of live-wire without (a) and with (b) cooling. Without cooling (a), all seed points must be placed manually on the object edge. With cooling (b), seed points are generated automatically as the live-wire segment freezes.*

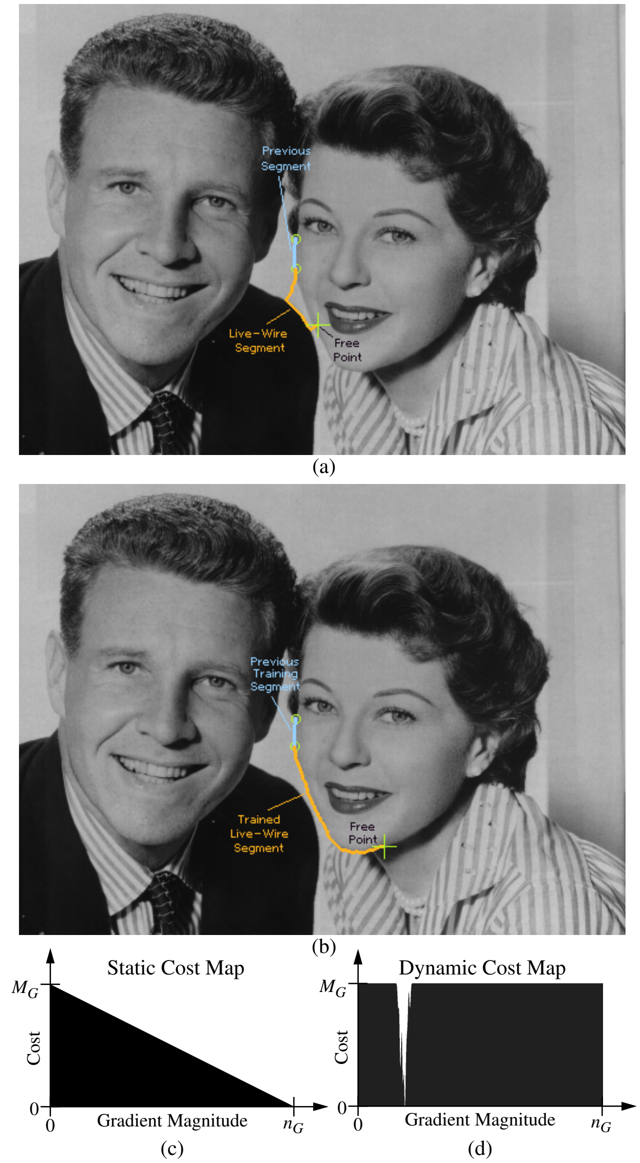


Figure 4: *Comparison of live-wire (a) without and (b) with dynamic training. (a) Without training, the live-wire segment snaps to nearby strong edges. (b) With training, it favors edges with similar characteristics as those just learned. (c) The static gradient magnitude cost map shows that without training, high gradients are favored since they map to low costs. However, with training, the dynamic cost map (d) favors gradients similar to those sampled from the previous boundary segment.*

the boundary is forced to pass through the seed points). To facilitate seed point placement, a cursor snap is available which forces the mouse pointer to the maximum gradient magnitude pixel within a user specified neighborhood. The neighborhood can be anywhere from 1×1 (resulting in no cursor snap) to 15×15 (where the cursor can snap as much as 7 pixels in both x and y). Thus, as the mouse cursor is moved by the user, it snaps or jumps to a neighborhood pixel representing a "good" static edge point.

3.4. Path Cooling

Generating closed boundaries around objects of interest can require as few as two seed points (for reasons given previously). Simple objects typically require two to five seed points but complex objects may require many more. Even with cursor snap, manual placement of seed points can be tedious and often requires a large portion of the overall boundary definition time.

Automatic seed point generation relieves the user from precise manual placement of seed points by automatically selecting a pixel on the current active boundary segment to be a new seed point. Selection is based on "path cooling" which in turn relies on path coalescence. Though a single minimum cost path exists from each pixel to a given seed point, many paths "coalesce" and share portions of their optimal path with paths from other pixels. Due to Bellman's Principle of Optimality [3], if any two optimal paths from two distinct pixels share a common point or pixel, then the two paths are identical from that pixel back to the seed point. This is particularly noticeable if the seed point is placed near an object edge and the free point is moved away from the seed point but remains in the vicinity of the object edge. Though a new optimal path is selected and displayed every time the mouse cursor moves, the paths are typically identical near the seed point and object edges and only change local to the free point. As the free point moves farther and farther away from the seed point, the portion of the active live-wire boundary segment that does not change becomes longer. New seed points are generated at the end of a stable segment (i.e., that has not changed recently). Stability is measured by time (in milliseconds) on the active boundary and path coalescence (number of times the path has been redrawn from distinct free points).

This measure of stability provides the live-wire segment with a sense of "cooling". The longer a pixel is on a stable section of the live-wire boundary, the cooler it becomes until it eventually freezes and automatically produces a new seed point.

Figure 3 illustrates the benefit of path cooling. In Fig. 3(a), the user must place each seed point manually on the object boundary. However, with cooling (Fig. 3(b)), only the first seed point (and last free point) need to be specified manually; the other seed points were generated automatically via cooling.

3.5. Interactive Dynamic Training

On occasion, a section of the desired object boundary may have a weak gradient magnitude relative to a nearby strong gradient edge. Since the nearby strong edge has a relatively lower cost, the live-wire segment snaps to the strong edge rather than the desired weaker edge. This can be seen in Fig. 4(a). The desired boundary is the woman's (Harriet's) cheek. However, since part of it is so close to the high contrast shoulder of the man (Ozzie), the live-wire snaps to the shoulder.

Training allows dynamic adaptation of the cost function based on a sample boundary segment. Training exploits an object's boundary segment that is already considered to be good and is performed dynamically as part of the boundary segmentation process. As a result, trained features are updated interactively as an object boundary is being defined. On-the-fly training eliminates the need for a separate training phase and allows the trained feature cost functions to adapt *within* the object being segmented as well as between objects in the image. Fig. 4(b) demonstrates how a trained live-wire segment latches onto the edge that is similar to the previous training segment rather that the nearby stronger edge.

To facilitate training and trained cost computation, a gradient magnitude feature map or image is precomputed by scaling the minimized gradient magnitude image, G', into an integer range of size n_G (i.e., from 0 to n_G - 1). The actual feature cost is determined by mapping these feature values through a look-up table which contains the scaled (weighted) cost for each value. Fig 4(c) illustrates edge cost based on gradient magnitude without training. Note that with training (Fig. 4(d)) edge cost plummets for gradients that are specific to the object of interest's edges.

Selection of a "good" boundary segment for training is made interactively using the live-wire tool. To allow training to adapt to slow (or smooth) changes in edge characteristics, the trained gradient magnitude cost function is based only on the most recent or closest portion of the current defined object boundary. A training length, t, specifies how many of the most recent boundary pixels are used to generate the training statistics. A monotonically decreasing weight function (either linearly or Gaussian based) determines the contribution from each of the closest t pixels. This permits adaptive training with local dependence to prevent trained feature from being too subject to old edge characteristics. The closest pixel (i.e., the current active boundary segment end point) gets a weight of 1 and the point that is t pixels away, along the boundary from the current active end point, gets a minimal weight (which can be determined by the user). The training algorithm samples the precomputed feature maps along the closest t pixels of the edge segment and increments the feature histogram element by the corresponding pixel weight to generate a histogram for each feature involved in training.

After sampling and smoothing, each feature histogram is then scaled and inverted (by subtracting the scaled histogram values from its maximum value) to create the feature cost map needed to convert feature values to trained cost functions.

Since training is based on learned edge characteristics from the most recent portion of an object's boundary, training is most effective for those objects with edge properties that are relatively consistent along the object boundary (or, if changing, at least change smoothly enough for the training algorithm to adapt). In fact, training can be counter-productive for objects with sudden and/or dramatic changes in edge features. However, training can be turned on and off interactively throughout the definition of an object boundary so that it can be used (if needed) in a section of the boundary with similar edge characteristics and then turned off before a drastic change occurs.

3.6 Comparison with Snakes

Due to the recent popularity of snakes and other active contours models and since the interactive boundary wrapping of the live-wire may seem similar to the "wiggling" of snakes, we highlight what we feel are the similarities and their corresponding differences between snakes and Intelligent Scissors.

<u>Similarities (compare with corresponding differences below):</u>

1. The gradient magnitude cost in Intelligent Scissors is similar to the edge energy functional used in snakes.
2. Both methods employ a smoothing term to minimize the effects of noise in the boundary.
3. Snakes and live-wire boundaries are both attracted towards strong edge features.
4. Both techniques attempt to find globally optimal boundaries to try to overcome the effects of noise and edge dropout.
5. Snakes and Intelligent Scissors both require interaction as part of the boundary segmentation process.

<u>Differences (compare with corresponding similarities above):</u>

1. The laplacian zero-crossing binary cost feature seems to have not been used previously in active contours models[1] (or DP boundary tracking methods for that matter).
2. The active contour smoothing term is internal (i.e., based on the contour's point positions) whereas the smoothing term for live-wire boundaries is computed from external image gradient directions[2(next page)].

---

1. Kass et al. [8] did use a squared laplacian energy functional to show the relationship of scale-space continuation to the Marr-Hildreth edge detection theory. However, the squared laplacian does not represent a binary condition, nor could it since the variational calculus minimization used in [8] required that all functionals be differentiable.

3. Snakes are typically attracted to edge features only within the gravity of an edge's gradient energy valley whereas the live-wire boundary can snap to strong edge features from arbitrary distances (since the 2-D DP's search window is the entire image).
4. Snakes are globally optimal over the entire contour whereas live-wire boundaries are piece-wise optimal (i.e., optimal between seed points). We feel this creates a desirable balance between global optimality and local control. This piece-wise optimality also allows for path cooling and intra-object on-the-fly training.
5. Finally, snakes refine (and interactively "nudge" by placing springs, etc.) a single rough boundary approximation where the live-wire tool interactively *selects* an optimal boundary segment from potentially *all* possible minimum cost paths.

Interactive optimal 2-D path selection is what makes Intelligent Scissors work and is the key difference between Intelligent Scissors and all previous techniques. Snakes are interactively initialized with an approximate boundary contour (often requiring several manually placed points); this single contour is then iteratively adjusted in an attempt to minimize an energy functional. The live-wire tool, on the other hand, is interactively initialized with just a single seed point and it then generates, at interactive speeds, all possible optimal paths from the seed point to ***every*** other point in the image, thus, allowing the user to interactively select the desired optimal boundary segment. As a result, Intelligent Scissors typically require less time and effort to segment an object than it takes to manually input an initial approximation to the object boundary.

Actually, the live-wire tool is much more similar to previous stage-wise optimal boundary tracking approaches than it is to snakes, since Intelligent Scissors were developed as an interactive 2-D extension to previous optimal edge tracking methods rather than an improvement on active contours.

4. Image Composition with Intelligent Scissors

As mentioned, composition artists need an intelligent, interactive tool to facilitate image component boundary definition. Since Intelligent Scissors can quickly segment object from an image, it serves as a tool for cut and paste operations. After object boundaries have been extracted, object can be copied from the image and placed into a buffer (i.e., clipboard) for future processing and placement into another image, or the same image if desired.

The cut object can be transformed--i.e., rotated, scaled, and translated, (RST)--before combination with the destination image. This is done using an interactive graphical tool with "handles" for RST control. The tool specifies a 2-D RST transformation matrix, $\boldsymbol{M}$. The source image is then bilinearly interpolated through the matrix to paste the cut object into the destination image.

Image composition often requires blending an object from one image into another image such that the cut-out object is not in the foreground. This requires the composition artist to "slip" the cut-out object behind some scene components while leaving it in front of other components. This operation can again be performed using the live-wire tool to create a composition mask[1]. Scene components can be cut out of an image to create a mask such that any additions or changes to the scene will not affect masked pixels.

4.1. Edge Filtering

As described, live-wire boundaries are limited by the pixel resolution of the image. This may produce jaggies along object boundaries in a resulting composition. However, subpixel accuracy can be obtained by exploiting the signed output of the laplacian operator. That is, the position of the object edge can be estimated to subpixel accuracy by using a (linearly) weighted combination of the laplacian pixel values on either side of the zero-crossings.

Since the live-wire boundary will not always correspond to a zero-crossing, jaggies can also be reduced by appropriate edge filtering, similar to anti-aliasing. Edge filtering is also desirable because real world images are acquired using finite image detectors and, as a result, pixels on or near an object boundary share information (i.e., color) from the object and the background.

One approach to edge filtering is to perform a local post-smoothing of the image around the pasted object boundary. However, this does not account for the fact that the edge pixels of the cut object very likely contain some background information from the original image. This is most noticeable when an object is composited into a scene with a different background color than the object's original background color. A more general solution would determine how much of each edge pixel corresponds to the actual object color and weight them accordingly when combining into another image.

4.2. Spatial Frequency and Contrast Matching

Once the object of interest has been segmented from the surrounding background we can scale it, rotate it, color it, or paste it onto another (destination) image. When pasting, it is desirable to perform image composition "seamlessly" in order to make it believable. That is, we should not be able to detect where the paste occurred. However, the source and the destination images will often have differing spatial frequencies or contrast due to differences in focus or lighting when the images were acquired. Thus, equivalencing of spatial frequencies and normalization of contrast is sometimes desirable in order to produce a convincing composition.

Equivalencing of spatial frequencies is performed by matching the spectral content of the cut piece and the destination image in the vicinity where it is to be pasted. Convincing composition often requires the spectra of the object and the destination image to match. This is accomplished by low-pass filtering the spectrum with the higher frequency content to match that of the other. The spectrum with the higher frequency content is determined by parameter fitting of a Butterworth low-pass filter (BLPF) to both spectra. Parameters corresponding to the spectrum with the lower frequency content are used to low-pass filter the spectrum of the other image.

The BLPF $B(u, v, d_0, n)$ is given by

$$B(u, v, d_0, n) = \frac{1}{1 + \left[\frac{\sqrt{u^2 + v^2}}{d_0}\right]^{2n}} \tag{3}$$

where d_0 is the distance of the cutoff frequency from the origin and n is the filter order.

Equivalencing of spatial frequencies is performed by first computing the Fourier transforms, $S(u, v)$ and $I(u, v)$ of the source image $S(x, y)$ and the destination image $I(x, y)$. We then compute the log power spectra $s(u, v)$ and $i(u, v)$:

$$\begin{aligned} s(u, v) &= \log[S(u, v)]^2 \\ i(u, v) &= \log[I(u, v)]^2 \end{aligned} \tag{3}$$

By varying the two parameters d_0 and n, a least squares fit can be used to create a normalized Butterworth filter $B(u, v, d_0', n')$ (where d_0' and n' are the fit parameters) matched to the spatial frequency characteristics of $i(u, v)$. If $i(u, v)$ demonstrates lower spatial fre-

2. Admittedly, the gradient direction cost used in Intelligent Scissors is more susceptible to noise in areas of low contrast (since it computes a smoothness cost based only on two points and one link). However, it is possible to extend the gradient direction term to include 3 pixels and 2 links without significant loss of computational efficiency.

1. Similar in concept to an optical mask used in motion picture special effects.

quency content than $s(u, v)$, the spatial frequencies between the two images can be equivalenced by fitting $B(u, v, d_0', n')$ to $s(u, v)$. The equivalenced result using the inverse Fourier transform

$$S'(x, y) = F^{-1}[B(u, v, d_0', n') \cdot S(u, v)] \tag{3}$$

is then pasted onto $I(x, y)$. Prior to pasting, the colors in the source image are scaled to the range of $I(x, y)$ to account for differences in contrast.

5. Results

Figures 5, 6, and 7 show the boundaries defined using Intelligent Scissors on a variety of image types. Fig. 5 is an artificial test image that exhibits gaussian edge blurring and point noise typical of some imaging hardware. Fig. 6 is the desktop scene used in Figures 2 and 3. Fig. 7 (a CT image of a lumbar spine) demonstrates the live-wire's application to medical imaging. The boundary definition times (for a trained user) for each displayed object boundary are given in the caption of each respective figure.

Figure 8 graphically compares the live-wire boundary definition times and boundary accuracy with manual tracing. These results show the average time and accuracy from a study where 8 untrained users[1] were asked to define the boundaries of five objects (the two objects in Fig. 5, the paper clip holder and pocket knife in Fig. 6, and the outer boundary of the spinal vertebrae in Fig. 7).

Figures 9(a-c) demonstrates Intelligent Scissors application to color images and show the boundaries defined using Intelligent Scissors for the image composition in Fig. 9(d). Objects were scaled, rotated, and (in the case of Fig. 9(a)) flipped to produce the final composition in Fig. 9(d). Note also that live-wire masking was performed on some of the foreground (grass).

Preprocessing requires 36 convolutions for color images (from 3×3, 5×5, 7×7, and 9×9 kernels), a gradient orientation calculation, a maximum gradient neighborhood search, and creation of a local cost map. For color images, we maximize feature values over the three color bands rather than averaging.

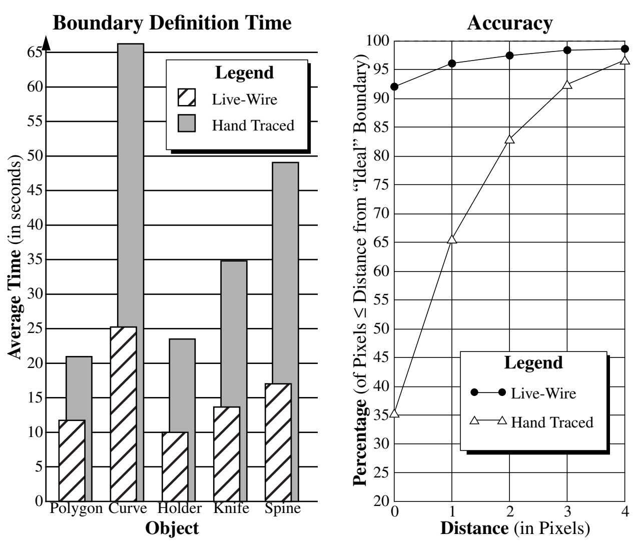


Figure 8: *Average timing and accuracy comparison between manually traced and live-wire boundaries for 8 users. Boundary times are for individual objects where accuracy measurements are over all objects.*

1. Each user spent a few minutes becoming familiar with the live-wire tool as well as a manual tracing tool and then were asked to define the boundary of 5 objects Each boundary was defined multiple times by each user with both Intelligent Scissors and manual tracing (to also measure intra- and inter-user reproducibility--not shown).

Previously, dynamic programming approaches to boundary detection were typically computationally expensive. However, by formulating DP as a graph search and restricting the local costs to integer values within a range, the 2-D DP algorithm can take advantage of an O(N) sort for N points. As mentioned, adding points to the sorted active list and removing points from it requires constant time. As a result, the algorithm's computational complexity for N image pixels is O(N). This can be seen by examining the algorithm in a worst case situation. As a pixel is removed from the active list, it is expanded by computing the cumulative cost to all of its neighbors that have not already been expanded. In the worst case, a pixel has its cumulative cost computed by all of its 8 neighbors, resulting in 8N cumulative cost computations for N pixels. Obviously, not every point can be expanded after all of its neighbors have. Except for the seed point, every point that has a cumulative cost must have at least one neighboring point that has already been expanded. Thus the cumulative cost is not recomputed for those neighbors. In short, it can be shown that at most only 4N cumulative cost computations are performed, resulting in an O(N) algorithm.

6.Conclusions and Future Work

Intelligent Scissors provide an accurate and efficient interactive tool for object extraction and image composition. In fact, and in sharp contrast to tedious manual boundary definition, object extraction using the live-wire is almost as much fun as the final result (the composition). Intelligent Scissors are intuitive to use and can be applied to existing black and white or color images of arbitrary complexity. There are many rich extensions of this work, including: (1) making use of the weighted zero-crossings in the Laplacian to perform subpixel edge filtering and anti-aliasing, (2) use of multiple layered (multiplane) masks, (3) making spatial frequency equivalencing locally adaptive, (4) varying the light source over the object using directional gradient shading (artificial or borrowed) to provide consistent lighting in the composition, and, most importantly (5) extension of the 2-D DP graph search and application of the live-wire snap and training tools to moving objects and moving, multiplane masks for composition of image sequences.

References

[1] A. A. Amini, T. E. Weymouth, and R. C. Jain, "Using Dynamic Programming for Solving Variational Problems in Vision," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 2, pp. 855-866, Sept. 1990.

[2] D. H. Ballard, and C. M. Brown, *Computer Vision*. Englewood Cliffs, NJ: Prentice Hall, 1982.

[3] R. Bellman and S. Dreyfus, *Applied Dynamic Programming*. Princeton, NJ: Princeton University Press, 1962.

[4] Y. P. Chien and K. S. Fu, "A Decision Function Method for Boundary Detection," *Computer Graphics and Image Processing*, vol. 3, no. 2, pp. 125-140, June 1974.

[5] D. Daneels, et al., "Interactive Outlining: An Improved Approach Using Active Contours," in *SPIE Proceedings of Storage and Retrieval for Image and Video Databases*, vol 1908, pp. 226-233, Feb. 1993.

[6] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, pp. 269-270, 1959.

[7] M. M. Fleck, "Multiple Widths Yield Reliable Finite Differences," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, no. 4, pp. 412-429, April 1992.

[8] M. Kass, A. Witkin, and D. Terzopoulos, "Snakes: Active Contour Models," in *Proc. of the First Int. Conf. on Computer Vision*, London, England, pp. 259-68, June 1987.

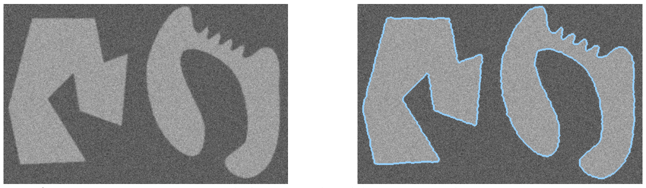


Figure 5: *Test image exhibiting edge blurring and point noise. Boundary definition times--polygon: 4.3 sec and curve: 8.3 sec.*

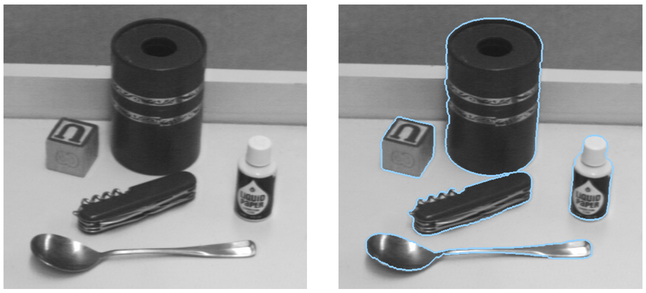


Figure 6: *Desktop scene. Boundary definition times--block: 2.4 sec; paper clip holder: 3.6 sec; pocket knife: 4.6 sec; liquid paper bottle: 5.1 sec; and spoon: 9.8 sec.*

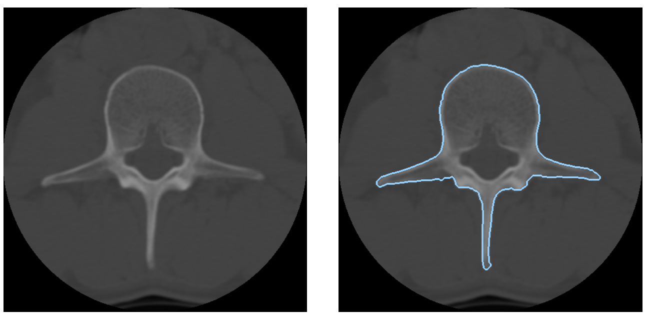


Figure 7: *Spinal vertebrae. Boundary definition time--5.9 sec.*

[9] D. Marr and E. Hildreth, “A Theory of Edge Detection,” in *Proceedings of the Royal Society of London--Series B: Biological Sciences*, vol. 207, no. 1167, pp. 187-217, Feb. 1980.

[10] A. Martelli, “An Application of Heuristic Search Methods to Edge and Contour Detection,” *Communications of the ACM*, vol. 19, no. 2, pp. 73-83, Feb. 1976.

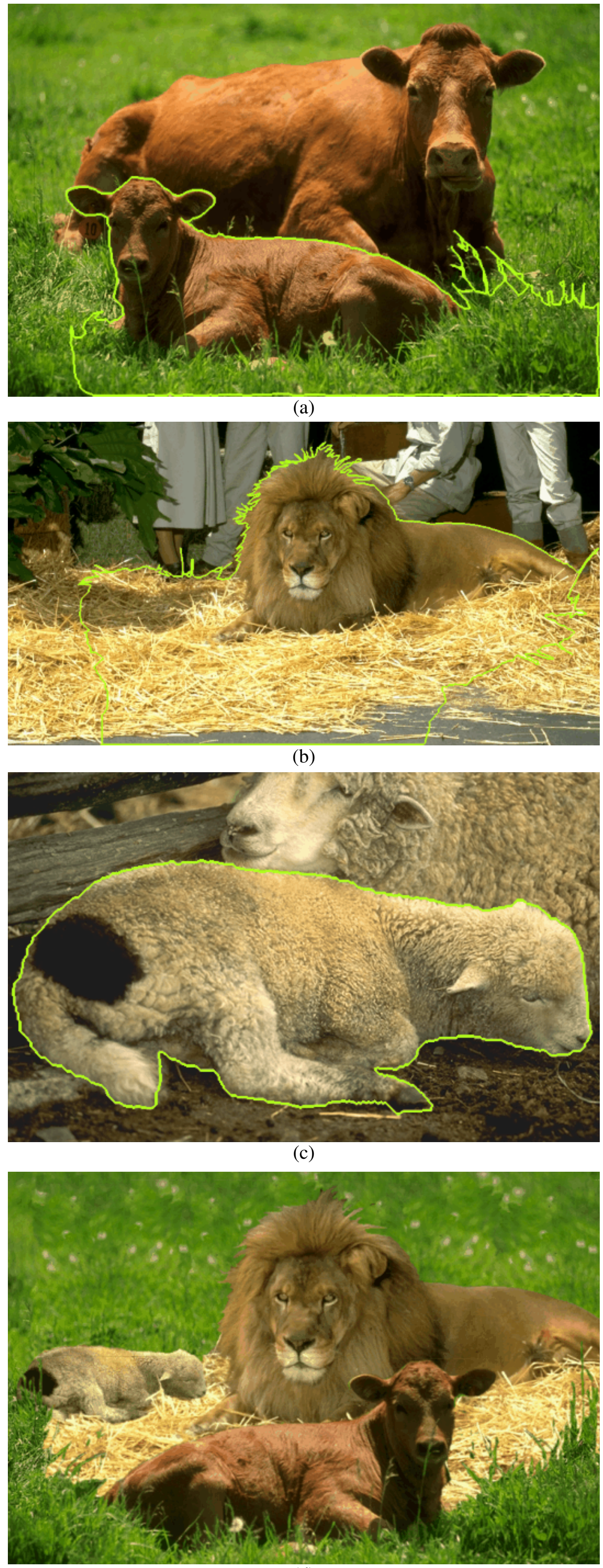
[11] U. Montanari, “On the Optimal Detection of Curves in Noisy Pictures,” *Communications of the ACM*, vol. 14, no. 5, pp. 335-45, May 1971.

[12] E. N. Mortensen, B. S. Morse, W. A. Barrett, and J. K. Udupa, “Adaptive Boundary Detection Using ‘Live-Wire’ Two-Dimensional Dynamic Programming,” in *IEEE Proceedings of Computers in Cardiology*, pp. 635-638, Oct. 1992.

[13] N. J. Nilsson, *Principles of Artificial Intelligence*. Palo Alto, CA: Tioga, 1980.

[14] D. L. Pope, D. L. Parker, D. E. Gustafson, and P. D. Clayton, “Dynamic Search Algorithms in Left Ventricular Border Recognition and Analysis of Coronary Arteries,” in *IEEE Proceedings of Computers in Cardiology*, pp. 71-75, Sept. 1984.

[15] D. J. Williams and M. Shah, “A Fast Algorithm for Active Contours and Curvature Estimation,” *CVGIP: Image Understanding*, vol. 55, no. 1, pp. 14-26, Jan. 1992.



(a)

(b)

(c)

(d)

Figure 9: *Images used in a composition. (a-c) Live-wire boundaries used for segmentation. (d) Final composition: “Millennium.”*