Live Mesh: An Interactive 3D Image Segmentation Tool

by

John Edwards

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

December 2004

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

John Edwards

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

_____	_____
Date	Parris Egbert, Chair
_____	_____
Date	Bryan Morse
_____	_____
Date	Dan Olsen

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of John Edwards in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

_____ _____

Date

Parris Egbert
Chair, Graduate Committee

Accepted for the Department

_____

David Embley
Graduate Coordinator

Accepted for the College

_____

G. Rex Bryce
Associate Dean, College of Physical and Mathematical Sciences

ACKNOWLEDGMENTS

Acknowledgments here.

Table of Contents

List of Figures

List of Algorithms

Chapter 1 Introduction

Three-dimensional object segmentation refers to the task of detecting and describing objects in volume data. Applications such as medical imaging, geological surveying, and computational fluid dynamics can all benefit greatly from 3D image segmentation. Approaches that are currently available to accomplish this task are typically non-interactive and difficult to use. In the 2D world, interactive tools are well developed. However, attempts to extend these techniques to 3D have not enjoyed the successes of their 2D counterparts. This thesis outlines Live Mesh, a novel interactive 3D image segmentation algorithm that provides an easy-to-use interface for extracting 3D geometry from volumetric data.

One field in which Live Mesh can be extremely useful is medical imaging. A common task for a doctor is to determine the size and shape of a tumor using CT scan image slices. Using Live Mesh, the doctor can interactively segment the tumor data from the surrounding bone and tissue data in a 3D-rendered environment and obtain a precise model of the tumor.

Few 3D object segmentation strategies proposed earlier allow the user to work directly in three dimensions. Many rely on 2D segmentation techniques such as Snakes or Live Wire (described in the next chapter). After segmenting in 2D the descriptions are combined and interpolated to produce a 3D object. While simple to use, these approaches don't allow the user to work on the data as a whole. Rather, they present the user with one slice at a time. The tools and user may make errors because of lack of context of the slice data. The Live Mesh algorithm works on the volume data, and, in addition, allows the user to interact in a 3D environment.

The Live Mesh method is a 3D extension of Live Wire, an algorithm that uses Dijkstra's shortest-cost algorithm and a user-interactive interface to find object edges. Live Wire has proven powerful and is widely used because of its high degree of user-interactivity. This is because automatic segmentation is still an unsolved problem and many datasets still require expert knowledge from users. Live Mesh retains the user-interactivity of Live Wire while extending the working space to three dimensions.

Novel ideas proposed in this thesis include a 3D extension of the Live Wire cost function (Chapter 3), a spatial restriction for the 3D graph search (Section 3.3.2) and the Live Mesh algorithm (Section 4.1) with proposed interpolation techniques (Section 4.3). A simple implementation of the Live Mesh method, the *SimpleSeg* application, is also discussed (Chapter 5).

Chapter 2 Previous Work

Image segmentation techniques have been used for many years in a variety of different applications to detect and describe objects in images. The applications range from simple to complex, and no one segmentation technique is adequate for all uses. Current image segmentation methods can be split into three [11] general categories based on the origin of the data used to perform the segmentation: global image information, region-based data, or edge-based data.

2.1 Global

The primary segmentation technique that depends on global knowledge of the image is thresholding [15,16]. This is accomplished for a gray-scale image by simply setting all pixels with intensity levels above a threshold T to intensity one, and setting all pixels below T to zero. This produces a binary image, with all "on" pixels identified as part of the object. Thresholding can utilize any image characteristic, including intensity, color or gradient.

Thresholding is simple both to understand and implement, and it provides an effective segmentation solution for a wide variety of applications. For simple problems, thresholding can be used exclusively, but often more is required. More commonly, it is used in conjunction with other, more sophisticated segmentation algorithms.

Thresholding uses global knowledge by obtaining the threshold T by analysis of the image's histogram. T is usually then applied to the entire image.

The remaining two general methodologies, region-based and edge-based, contain numerous specific segmentation techniques which are more focused and sophisticated than thresholding.

2.2 Region-based segmentation

Consider a picture of a red glove sitting on a brown table. If the lighting is good, most pixels comprising the glove will be some shade of red or another. Exploiting this feature, an algorithm could pick a pixel in the middle and mark it as being part of the glove. The algorithm could then mark all neighboring pixels as part of the glove. This

could continue recursively until all contiguous pixels of some shade of red are found. This is the basic principle that region-based segmentation techniques [14,18] are based on.

Region-based segmentation relies on homogeneity of the target object's pixels or some characteristic of the pixels. If two regions in an image are sufficiently similar to each other are merged into one region. This can be done recursively, starting at the pixel level, until all remaining, adjacent regions in the image are dissimilar enough that no more merging can be done. The region resulting from merging is considered to be the object desired.

Region-based methods are generally considered to be fairly tolerant to noise but they are hindered by the requirement that the object's pixels be similar to each other in order to be considered in the same region. If the object does contain disparate regions then these regions must be manually merged, and if the object has many such regions, manual merging can be prohibitively time-consuming, such as would be the case if the user were segmenting a patchwork quilt from a scene. In such a case either the method's similarity metric must be customized to the image (or class of images) or a different segmentation technique must be used.

2.3 Edge-based segmentation

Consider the above example of a patchwork quilt. The pixels within the quilt are disparate, but the edge of the quilt against the background is quite clear. Edge-based segmentation techniques [11,17] take advantage of this by finding edges of the object. Contiguous pixels exhibiting similar edge characteristics are grouped together and considered to define an object boundary (or edge), ideally circumnavigating the object.

Edge-based techniques work on the output from edge detector operators. Edge detectors, such as the Laplace, Prewitt and Sobel operators, find pixels in the image which exhibit characteristics of an edge. Pixels with high likelihood of being an object edge are ones which are highly dissimilar from some (but not all) of their neighbors. If the image were considered as a function, these pixels would have a very high first derivative or gradient. Additionally, the pixel's second derivative would likely be close

to zero, indicating that it is a peak in the gradient. The output from these derivative-finding operators provides a starting place for edge-based segmentation techniques.

Most image segmentation techniques use little user interaction. However, as complex scenes may require expert *a priori* knowledge, a degree of user interaction may be desirable. Two successful edge-based, highly user-interactive segmentation techniques are Snakes [2] and Live Wire [1].

2.3.1 Snakes

Snakes [2], or Active Contour Models, are splines which fit themselves to object boundaries based on internal energy, external energy and user-defined constraints. We define internal energy as the parameters describing the flexibility of the snake, or how much force is required to bend the snake. The external energy is obtained from the image itself: areas with strong edge characteristics attract the snake, and areas that do not appear to be edges repel the snake. If an image is thought of as a mountain range, with peaks defined as areas of low gradient, then the snakes algorithm can be thought of as placing a tube of flexible rubber along the mountain range. The tube will have the tendency to settle in valleys (external energy), but only insomuch as required bending doesn't exceed its degree of stiffness (internal energy).

Users have some control over the behavior of the snake as it settles into a low-energy state. User-defined constraints such as springs and repulsors allow the user to manipulate the behavior of the snake at runtime.

The snake is first initialized by the user, either by being drawn (Figure 1a) or initialized using Live Wire[6] (explained below). The snake then settles into a state of equilibrium between the internal and external energy factors (Figure 1b). The snake's settled state is often off the mark because of lack of precision of the initial snake or misleading external forces. Constraints can then be used to push and pull the snake into position for a final, precise segmentation (Figure 1c).

Snakes have proven to be very effective segmentation tools. While constraining forces provide tools for the user to manipulate the position and shape of the snake, if the snake goes far amiss, the task of repairing can be difficult and time-consuming.

(a) *(b)* *(c)*

Figure 1 Snakes Example
(a) User-initialized snake. (b) Snake after reaching equilibrium. (c) Snake at equilibrium after manipulating with constraints.

2.3.2 Live Wire

Live Wire is a very powerful edge-based segmentation method. Its strength lies in its very high degree of user-interactivity. The user clicks on an edge of the object with the mouse, defining a "seed point." He then moves the cursor to some other portion of the object's edge. The pixel that lies under the cursor is called the "free point." As the free point moves, a wire connecting the seed point and the free point automatically snaps to the edge (Figure 2a). While the algorithm does do significant automatic segmentation (by snapping to the edge), the user maintains complete control in that he can adjust placement of the free pixel if the algorithm fails to find the correct boundary (Figure 2b and 2c).

Live Wire uses Dijkstra's shortest cost path algorithm. Dijkstra's algorithm finds the lowest cost path between two points. The cost function in Live Wire is defined such that the paths found by the algorithm lie along edges. The wire that snaps to the edge is simply the path from the free point to the seed point that has been determined to be of lowest cost. Dijkstra's algorithm not only finds the path between these two points, but also the shortest paths from every point in the search space back to the seed point. The Live Wire algorithm is treated in more detail in Chapter 4.

(a) *(b)* *(c)*

Figure 2 Live Wire Example
(a) User dragging the wire around the dolphin. (b) Getting a little carried away. (c) After finding the maximum drag distance while still finding the optimal boundary.

2.4 3D Segmentation

Both Snakes and Live Wire have 3D extensions. Snakes' 3D counterparts are known as Balloons [13]. They work directly in 3 dimensions, expanding, contracting and warping to the object. Balloons suffer from the same difficulty as Snakes – loss of user control after initialization. Because of this, users prefer to drop back into the 2D world by running the snake algorithm on the initial slice, and then using the resultant snake as the initial snake for the successive slices in the volume.

Live Wire can also be used to segment 3D volumes. Schenk *et al.* [7] have optimized this process by only computing the cost for successive slices on regions that fall within a given distance from a reference slice's completed boundary. A distance map, showing distances from the reference slice's boundary, is generated using a chamfering process. This distance map is used to determine which pixels in the current slice will be used for the local cost computation. In addition to speeding cost computation, local cost feature optimization can be performed.

Schenk *et al.* also introduce an automatic segmentation step using shape-based interpolation followed by a Live Wire-driven optimization process [7,12]. The user segments selected slices, then intermediate slices are automatically interpolated. The interpolation is done by first building distance maps for each segmented slice. A conventional gray-level interpolation algorithm then builds a new, interpolated distance map. Finding the zero-crossings of the interpolated distance map then generates a binary contour. New seed points are interpolated from the seed points used in the segmented

images, and these seed points are used to run Live Wire in automatic mode to optimize the interpolated boundary. This offers a degree of automation, though the user is still required to inspect each slice and correct any errors.

Falcao and Udupa [8] developed a method that determines object boundary information from orthogonal slices of a volume segmented by a user. This technique is fairly complicated, but it does allow the user to segment a minimal number of slices, reducing the total segmentation time. The 3D scene is first viewed as a montage of axial slices, from which the user can determine a general idea of the topology of the object of interest. The user then breaks the object up into *slabs* of constant object topology, or sets of contiguous axial slices such that the connectedness of the object boundary pixels does not change from slice to slice. For a simple object, such as a sphere, only one slab needs to be defined. But for a more complicated surface, multiple slabs may be required as the boundary of the surface in an axial slice may need to be represented by several 2D boundary contours.

Once the slabs are defined, the user selects a point c on one of the slices in a slab. Each point on a line l parallel to the z-axis and passing through c must lie within the object of interest. A set S of slices intersecting with l is chosen. Each slice in S is segmented with Live Wire, forming a set P of points. For each axial slice, all points in P that lie on the axial slice are used effectively as seed points for automatic segmentation of the axial slice using Live Wire. This approach reduces the time required for segmentation of the entire 3D boundary, given that the number of orthogonal slices necessary is relatively low. This segmentation technique requires a fair degree of user training, however.

The above 3D segmentation methods use Live Wire to a degree, but suffer from loss of user control, much like balloons.

Chapter 3 Graph Search

To segment objects in true 3D space, a 3D graph search is required. As Live Wire uses Dijkstra's traditional 2D algorithm, we have developed a 3D version of the same algorithm for use in Live Mesh. Both of these versions of the algorithm find shortest cost paths between pixels such that the paths lie along object boundaries. Extending Dijkstra's algorithm to three dimensions requires a 3D extension to the cost function (Section 3.2). In addition, since Live Mesh is a user-interactive tool, we have developed optimizations (Section 3.3) so that the algorithm is fast enough to keep the software responsive to the user.

3.1 Dijkstra's Algorithm

Dijkstra's algorithm is the basis for the dynamic nature of Live Wire and Live Mesh. Live Mesh uses the same basic algorithm as Live Wire. We have modified it slightly for optimization (Section 3.3). Dijkstra's algorithm finds the shortest cost path from any vertex (pixel) *v* to a seed vertex *s*. The algorithm is as follows:

Algorithm 1 Dijkstra's algorithm

```
Description: Finds shortest cost path from every Vertex v back
to a seed Vertex s.
Structure: Vertex containing three properties:
     Vertex prev
     Number cost
     Boolean isVisited
Input: seed Vertex s, Graph G
Output: prev pointer for each Vertex v in G

(1)  SortedList q
(2)  Vertex v
(3)  Number c
(4)  for every Vertex v in G
(5)     set v.cost = INFINITY
(6)     set v.isVisited = false
(7)  end for
(8)  set s.cost = 0
(9)  insert s into q
(10) while q is not empty
(11)    set v = least cost Vertex in q
(12)    remove v from q
(13)    set v.isVisited = true
(14)    for each neighbor Vertex v_i near v
(15)       if not v_i.isVisited
(16)          set c = cost(v, v_i)
```

```
(17)         if v.cost + c < v_i.cost
(18)            set v_i.cost = v.cost + c
(19)            set v_i.prev = v
(20)            insert v_i into q
(21)         end if
(22)       end if
(23)    end for
(24) end while
```

The seed vertex is initialized with a cost of zero (line 8), as the cost of traveling from itself to itself is zero, and all other vertices are initialized with a cost of infinity (line 5). The seed is then placed in a cost-sorted list q (line 9), called the "active list." As the seed vertex is the only element in the active list q, it is the first to be removed. Then, for each neighbor v_i the cost c from the neighbor back to the seed is determined. For performance, the cost c should simply be a lookup or a simple calculation. If the seed cost v.cost plus the calculated cost c is less than the neighbor's cost v_i.cost then v_i.cost is set to be v.cost + c (line 18), and the neighbor is given a pointer to the seed ($v_i.prev$). The neighbor is then added to the active list (line 20).

Once all the seed's neighbors' costs have been updated, the seed is discarded and the neighbor with the lowest cost to the seed is selected as the next vertex to visit (line 11). The same process is iteratively followed for each vertex.

Once a vertex has been visited, it contains the optimal path back to the seed vertex following each vertex's *prev* property:

Algorithm 2 Print shortest-cost path

```
Description: Prints all vertices v_i in shortest-cost path from a
vertex v to a seed vertex s.
Structure: vertex containing one property:
     vertex prev
Input: vertex v, s

(1) vertex cur
(2) set cur = v
(3) while cur ≠ s
(4)    print cur
(5)    set cur = cur.prev
(6) end while
```

Once the algorithm has visited all connected vertices in the graph, each of those vertices has an optimal path back to the seed.

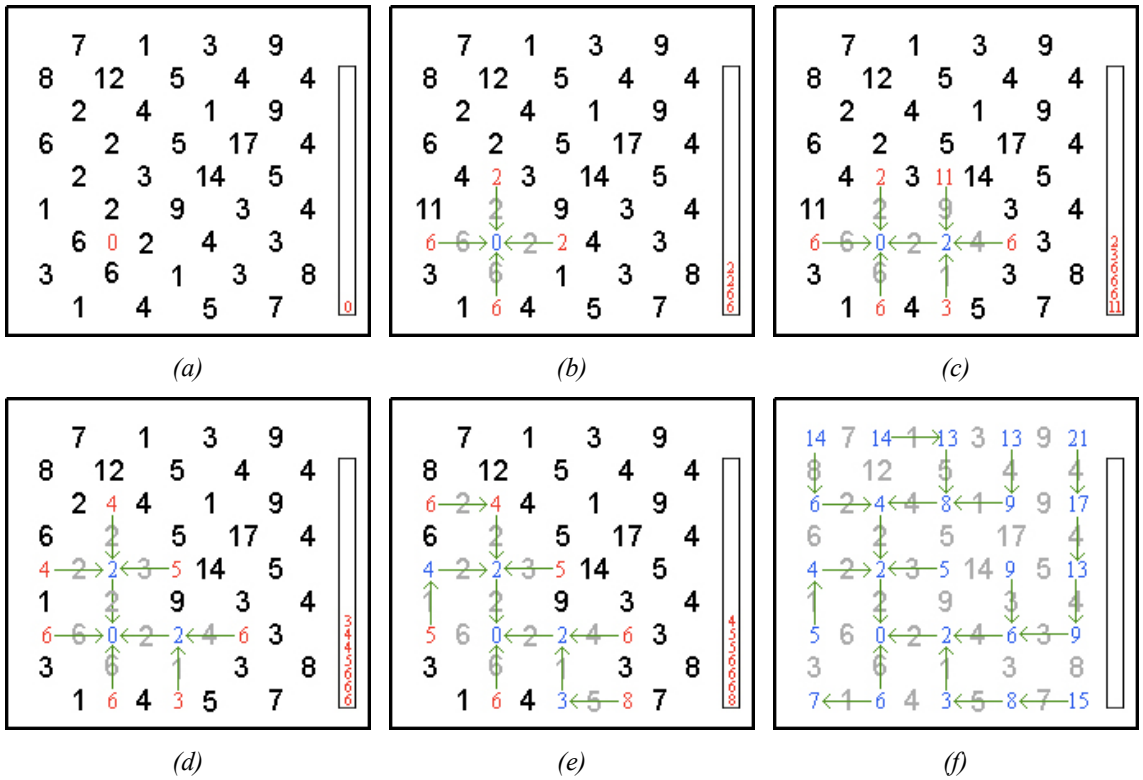


Figure 3 Dijkstra's algorithm
(a) Initial image. Each black number is the cost of traveling between two pixels. The seed pixel (in red) is initially the only pixel in the sorted list (right-hand box). (b) After visiting the seed pixel. Each red number is the cost of traveling to the seed pixel. (c-e) Continuing the search. (f) The final result.

3.2 Cost function

The cost function used in the algorithm described above is simply a function which calculates the cost of traveling between two adjacent voxels. The function is formulated such that the cost is lowest along edges of an object, ensuring that the lowest cost path will lie along a boundary. Three primary image features are used: Laplacian zero-crossing (f_Z), gradient magnitude (f_G) and gradient direction (f_D). Each feature is weighted with constants (ω_Z, ω_G and ω_D). The cost function is formulated as

$$\text{cost}(u,v) = \omega_Z * f_Z(v) + \omega_G * f_G(v) + \omega_D * f_D(u,v)\,. \qquad (1)$$

The following discussion of the cost function refers to 2D pixels rather than 3D voxels for simplification. Extending the cost function to three dimensions is fairly straightforward and intuitive. Rather than looking at the eight neighbor pixels, as in the

2D case, the 27 voxel neighbors are considered in the 3D case. In addition to this change, there are a few other minor modifications that will be noted as they are described.

3.2.1 Laplacian zero-crossing

The Laplacian zero-crossing feature gives an indication of whether a pixel is on an edge or not. The algorithm uses information about the image and returns a binary value specifying whether the pixel is an edge pixel or not.

The Laplacian operator finds the second derivative of the image intensity, or the derivative of the gradient. By definition of the derivative, every pixel in the Laplacian image that is equal to zero is a maximum of the first derivative (image gradient). Additionally, any place where adjacent pixels have different signs denotes a gradient maximum (Figure 4). The Laplacian image is binarized such that all pixels on zero crossings are given a pixel value of zero and all other pixels are given a value of one by the following algorithm:

Algorithm 3 In-place binarizing of Laplacian image

```
Input: Laplacian Image lap
Output: Binarized Laplacian Image lap

(7)  for every pixel p_ij in lap
(8)    if p_ij < 0
(9)      for every 8-connected neighbor n_ij of p_ij
(10)       if n_ij > 0
(11)         p_ij = 0
(12)       end if
(13)     end for
(14)   end if
(15)   if p_ij != 0
(16)     p_ij = 1
(17)   end if
(18) end for
```

The choice to find crossings from negative pixels to positive pixels is arbitrary. The opposite would work equally as well.

$f_Z(v)$ is defined as the value of pixel v in the image output from Algorithm 3. Since this value will be zero or one, the contribution the Laplacian makes to the cost function (Equation 1) will be either zero or ω_Z.

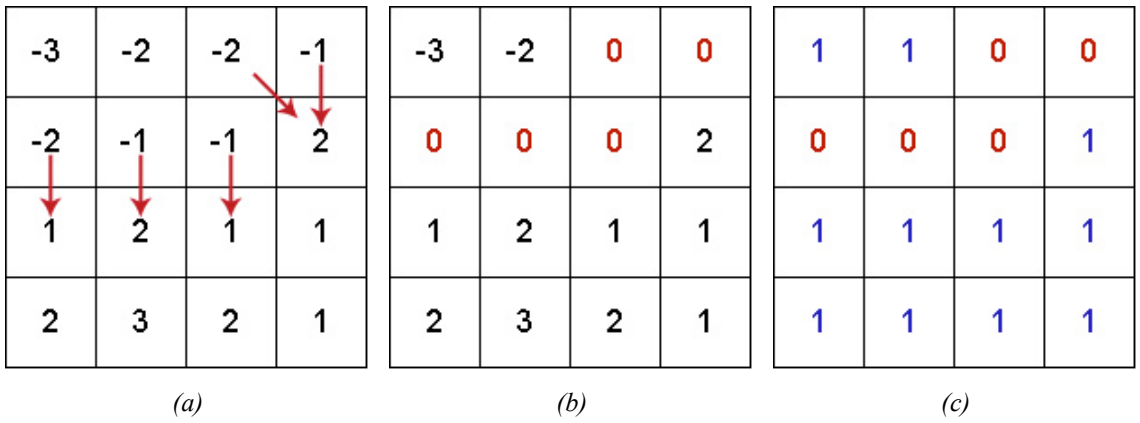


Figure 4 Laplacian zero crossings
(a) Laplacian image with zero crossing indicators. (b) Pixels that are now effectively zero. (c) The binarized image.

3.2.2 Gradient magnitude

The gradient magnitude feature of the cost function gives us an idea of how confident we are that this pixel is an edge (as opposed as "whether the pixel is an edge or not," which is given us by the Laplacian zero-crossing calculation). The gradient magnitude is the first derivative of the image. In 3D, it can be visualized using the mountain analogy. A cliff has a very high gradient magnitude, whereas a grassy meadow has a very low gradient magnitude.

The gradient magnitude can be computed using the partial derivatives of a volume. The Prewitt and Sobel operators (among others) find the partial derivative in one direction. We can derive the gradient magnitude at a pixel p as

$$G(p) = \sqrt{I_x^{\,2} + I_y^{\,2}} \approx |I_x| + |I_y| \tag{2}$$

or at a voxel v as

$$G(v) = \sqrt{I_x^{\,2} + I_y^{\,2} + I_z^{\,2}} \approx |I_x| + |I_y| + |I_z| \tag{3}$$

where I_x, I_y and I_z are the partial derivatives along the x, y and z axes, respectively.

Pixels with a high gradient magnitude have a high likelihood of being edges. The final cost of a pixel near an edge should be low, so $f_G(p)$ is computed using the inverse of the gradient:

$$f_G(p) = \frac{\max(G) - G(p)}{\max(G)} = 1 - \frac{G(p)}{\max(G)} \quad (4)$$

where $\max(G)$ is the maximum gradient magnitude value of any pixel in the image. Therefore f_G is the inverse of the normalized gradient magnitude at pixel p.

3.2.3 Gradient direction

Mount Shasta in California is shaped very much like a cone. A person standing halfway up the north side and wishing to hike to the point on the south side at the same altitude has a few choices. An adventurous person would hike up and over the top. A starving, tired person wishing to expend as little energy as possible would travel around the mountain, taking care to neither gain nor lose altitude, traveling along a contour of constant altitude.

Object boundaries very often lie along "contour" lines of the image. In the original Intelligent Scissors implementation, Mortensen and Barrett [3] added a gradient direction constraint such that lowest cost paths would tend to lie along these contour lines. In doing so, they noted that the wire was smoothed out. Live Mesh also uses the gradient direction constraint. This is the only constraint that doesn't give any indication of whether the pixel is on an edge or not. It deals only with whether two pixels lie along the same contour.

3.2.3.1 Live Wire (2D)

Consider an image that is white on the left side and gradually moves to black on the right side (Figure 5a). All pixels have identical gradient magnitudes. Each edge between two pixels p and q would yield very similar costs if we used only the f_Z and f_G constraints. But the cost of traveling to the right or left should be much higher than up or down, since moving up or down would be going along a contour of constant color. A gradient direction vector $D(p)$ is defined by the partial derivatives I_x and I_y (and I_z for a voxel) and by definition points up the steepest slope at that point (Figure 5b). Rotating $D(p)$ by 90 degrees defines $D'(p)$ which points along the gradient contour (Figure 5c). Figures 4d and 4e show what $D(p)$ and $D'(p)$ look like for various pixels in a less trivial image.

We then define a variable L as the direction between two pixels p and q such that the difference between L and $D(p)$ is minimized:

$$L(p,q) = \frac{1}{\|p-q\|}\begin{cases} q-p; & \text{if } D'(p)\cdot(q-p) \geq 0 \\ p-q; & \text{if } D'(p)\cdot(q-p) < 0 \end{cases} \tag{5}$$

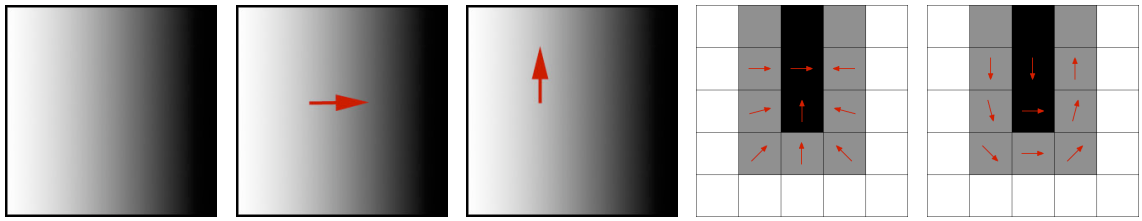
We can express the difference between $D'(p)$ and L as

$$\begin{aligned} d_p(p,q) &= \text{acos}[D'(p)\cdot L(p,q)] \\ d_q(p,q) &= \text{acos}[L(p,q)\cdot D'(q)] \end{aligned}. \tag{6}$$

f_D is then defined as

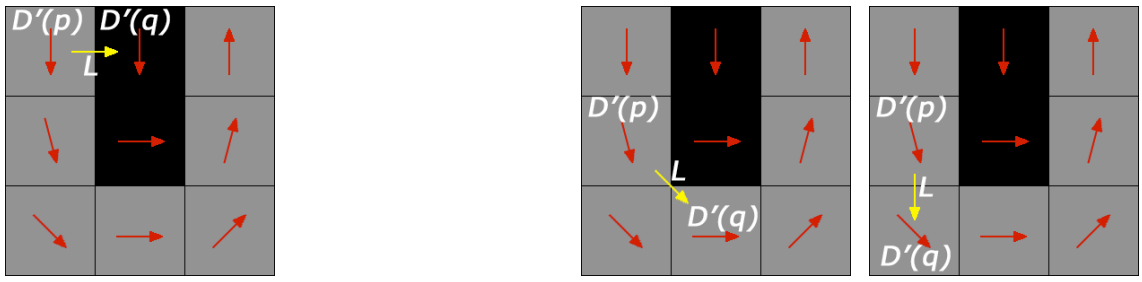
$$f_D(p,q) = \frac{2}{3\pi}\{d_p(p,q) + d_q(p,q)\}. \tag{7}$$

Figure 6 shows examples of values of f_D for different pixel combinations. Figures (a) and (c) yield very different f_D values since the vector between pixels L in (a) is pointing very much contrary to the contour of the image, whereas in (b), L is in very close alignment with the contour. Note that if $D'(p)$ or $D'(q)$ were rotated by 180 degrees for any p or q the value for f_D would not change (5a and 5b). This is because $d_q(p,q)$, by reason of the inverse cosine, is invariant to 180-degree rotations in either contributor to the dot product (Equation 6).



(a) *(b)* *(c)* *(d)* *(e)*

Figure 5 Gradient Direction of Two Different Images
(a) Image with constant gradient magnitude. (b) Gradient direction $D(p)$ for a pixel. (c) Rotated gradient direction $D'(p)$ for the same pixel. (d) $D(p)$ for each pixel (edge pixels' gradient directions are not defined). (e) Rotated gradient directions $D'(p)$.



(a)	(b)	(c)	(d)
$\boldsymbol{D}'(\boldsymbol{p}) = [0,-1]$	$\boldsymbol{D}'(\boldsymbol{p}) = [0,1]$	$\boldsymbol{D}'(\boldsymbol{p}) = [0.253,-0.967]$	$\boldsymbol{D}'(\boldsymbol{p}) = [0.253,-0.967]$
$\boldsymbol{D}'(\boldsymbol{q}) = [0,-1]$	$\boldsymbol{D}'(\boldsymbol{q}) = [0,-1]$	$\boldsymbol{D}'(\boldsymbol{q}) = [1,0]$	$\boldsymbol{D}'(\boldsymbol{q}) = [0.707,-0.707]$
$\boldsymbol{L}(\boldsymbol{p},\boldsymbol{q}) = [1,0]$	$\boldsymbol{L}(\boldsymbol{p},\boldsymbol{q}) = [-1,0]$	$\boldsymbol{L}(\boldsymbol{p},\boldsymbol{q}) = [0.707,-0.707]$	$\boldsymbol{L}(\boldsymbol{p},\boldsymbol{q}) = [0,-1]$
$f_D(\boldsymbol{p},\boldsymbol{q}) = 0.667$	$f_D(\boldsymbol{p},\boldsymbol{q}) = 0.667$	$f_D(\boldsymbol{p},\boldsymbol{q}) = 0.278$	$f_D(\boldsymbol{p},\boldsymbol{q}) = 0.220$

Figure 6 Calculation of f_D

Calculation of L and f_D for pixels varying in gradient direction and spatial relationship.

3.2.3.2 Live Mesh (3D)

In 3D, f_D is calculated very much like it is in 2D, except for the complication of not knowing what axis around which to rotate the gradient direction vector $D(p)$ by 90 degrees to obtain $D'(p)$. To get around this, we first define L' as the vector orthogonal to $(q-p)$ such that the difference between L' and $D(p)$ is minimized:

$$L'(p,q) = \frac{T(p,q)\times(q-p)}{\left\|T(p,q)\times(q-p)\right\|} \tag{8}$$

where

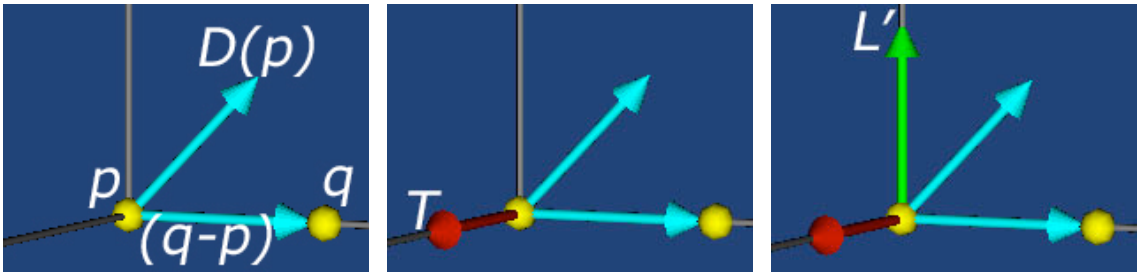
$$T(p,q) = D(p)\times(q-p). \tag{9}$$

$T(p,q)$ is the vector orthogonal to both $D(p)$ and $(q-p)$ (Figure 7b). By taking the cross product of $T(p,q)$ and $(q-p)$ we find L', which is the vector orthogonal to $(q-p)$ that is closest to $D(p)$ (Figure 7c). We can also think of L' as being a vector orthogonal to $(q-p)$ that is in the plane containing both $(q-p)$ and $D(p)$.

At this point we redefine Equation (6) to be:

$$\begin{aligned} d_p(p,q) &= \operatorname{acos}[D(p)\cdot L'(p,q)] \\ d_q(p,q) &= \operatorname{acos}[L'(p,q)\cdot D(q)] \end{aligned}, \tag{10}$$

the difference being that we're now using $D(p)$, $D(q)$ and L' rather than $D'(p)$, $D'(q)$ and L. f_D is then calculated using (7), as is done in 2D.



(a) *(b)* *(c)*

Figure 7 Calculation of L'
(a) Starting vectors $(q-p)$ and $D(p)$. (b) $T(p,q)$ is a vector orthogonal to both $(q-p)$ and $D(p)$. (c) L' is orthogonal to $(q-p)$ and $T(p,q)$.

3.3 Search optimization

One challenge in extending Live Wire to three dimensions is the actual computation time of the graph search. The computational complexity of the search is increased by an order of magnitude when adding another dimension, which renders a traditional Dijkstra search very expensive. Further speedup is accomplished by pre-processing the cost matrix and restricting the search.

3.3.1 Cost Pre-processing

The computation for the cost between two voxels in three dimensions is very expensive. Assume that at search time the `acos` function has been pre-computed into a lookup table and that the only values available are the laplacian result and the gradient magnitude. If this is the case, then nineteen multiplies and a divide are required to compute the cost between a voxel pair. To speed up the graph search a cost calculation pre-processing step is employed. This computes the cost for every voxel pair and stores each cost in a matrix for quick array access. The array is large:

$$arraysize = \#voxels * \frac{\#neighbors}{2} = \#voxels * \frac{26}{2} \qquad (11)$$

For a 256 x 256 x 256 volume this would require $256^3 * 13 = 218{,}103{,}808 = 208\text{Mb}$ of storage if the costs are stored as unsigned bytes. The size jumps to ~1.6 Gb in the case of a 512 x 512 x 512 image. Tested on a Pentium IV 1.7 with 512 Mb of RAM this pre-processing step can take upwards of 3 hours.

Lest the time required for pre-processing seem prohibitive, it should be kept in mind that the entire cost pre-processing can be done offline (without user interaction). And the benefits are substantial. Pre-computing every cost reduces the cost computation at search time to a single array lookup, resulting in a speedup of more than a factor of 5.

3.3.2 Restricted Search

By default, the graph search expands throughout the entire image, but the user is usually interested in only a portion of the image. The time required for the graph search to find the shortest cost path from the free point to the seed point can be decreased by restricting the graph search to voxels in the neighborhood of the free and seed points (Figure 8a).

If a restriction is used, the path is optimal only within the search space. This is fine if the edge contour is relatively linear, but if the image edges are poorly defined or jagged then the locally optimum path may in fact be very far from the globally optimum path. An additional problem is that once the free point is moved outside of the restricted search space the search must be restarted. The following two sections describe an algorithm by which these difficulties can be worked around with minimal impact while retaining the benefits of a restricted search.

3.3.2.1 Expanded Search Space

Suppose a restricted search has run to completion, such that each point in the search space has a locally optimal path back to the seed point. If both the seed and free points are still in place, then a new, expanded search can be started to find a path that is optimal within a larger search area. A naive implementation might be to simply re-search all points in the new, larger search area. However, many of the points in the original search already have globally optimal paths. For example, every neighbor of the seed point is guaranteed to have a globally optimal path even in a restricted search.

This novel algorithm takes advantage of the fact that we already have some globally optimal information. The original search area is re-searched from the outside-in only as far as needed. We use Dijkstra's algorithm, with a few modifications, as follows (additions are underlined and red, deletions are ~~strikethrough~~):

Algorithm 4 Expanded Restricted Search

```
Description: Finds shortest cost path from every Vertex v back
to a seed Vertex s.
Structure: Vertex containing three properties:
     Vertex prev
     Number cost
     Boolean isVisited
Input: ~~seed vertex s,~~ graph G, SortedList q, Set restriction
Output: prev pointer for each vertex v in G, list r suitable for
expanded active list

(1)  SortedList ~~q~~ r
(2)  Vertex v
(3)  Number c
(4)  for every Vertex v in G
(5)     set v.cost = INFINITY
(6)     set v.isVisited = false
(7)  end for
(8)  set s.cost = 0
(9)  insert s into q
(10) while q is not empty
(11)    set v = least cost Vertex in q
(12)    remove v from q
(13)    set v.isVisited = true
(14)    for each neighbor Vertex v_i near v
(15)       if not v_i.isVisited
(16)          if restriction does not contain v_i
(17)             insert v into r
(18)          else
(19)             set c = cost(v, v_i)
(20)             if v.cost + c < v_i.cost
(21)                set v_i.cost = v.cost + c
(22)                set v_i.prev = v
(23)                insert v_i into q
(24)             end if
(25)          end if
(26)       end if
(27)    end for
(28) end while
```

The active list q contains all voxels on deck to be visited. q is passed into the algorithm as input rather than being created inside the algorithm, as it is in the original Dijkstra's algorithm. For the first search, which originates from the seed, q would be

initialized to contain *s* and every vertex would be given a value of infinity before the algorithm is called. For each subsequent call, however, this is not done.

In the above algorithm, any vertex in the search space that has a neighbor vertex that is outside of the search space is added to a cost-sorted list *r* (line 17). At the conclusion of the algorithm, *r* contains all vertices along the boundary of the restriction. *r* can then be used as the active list in a subsequent, expanded search (Figure 8).

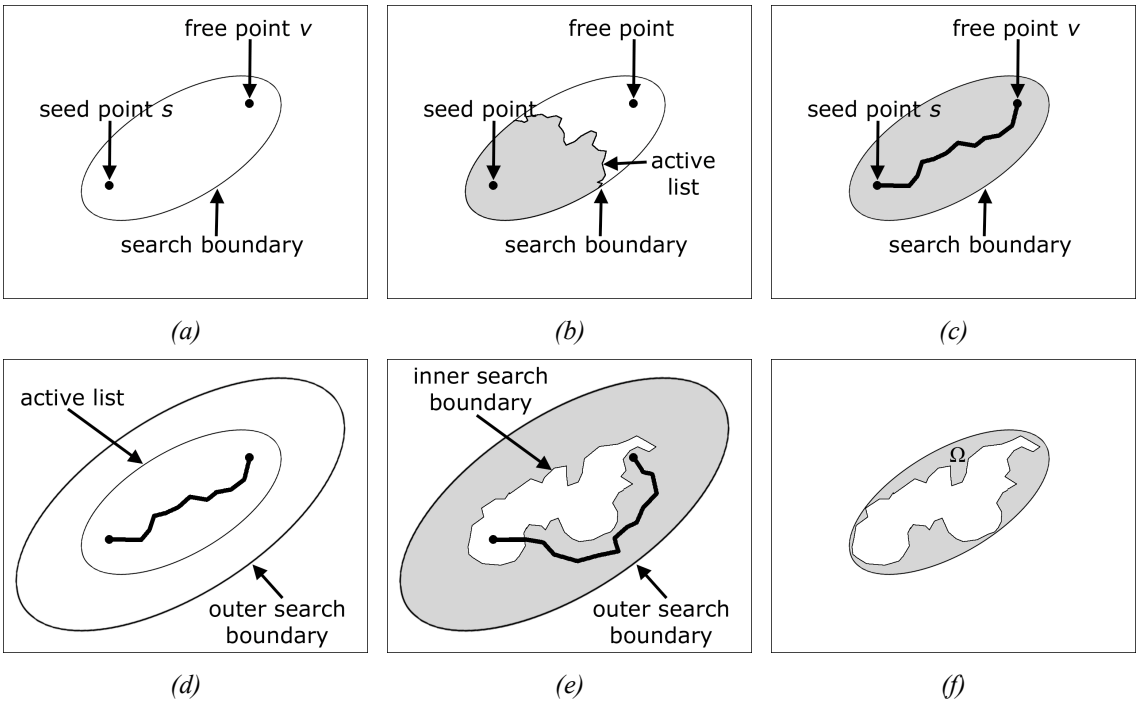


Figure 8 Expanded search
(a) Setting up the initial graph search. (b) Search in progress. Searched area is shaded. (c) Completion of initial graph search. (d) Setting up the expanded graph search. (e) Completion of expanded graph search. (f) _ is the area re-searched.

Each vertex in the graph must be reset to "not-visited" (line 6) in case vertices inside the old restriction need to be re-searched. Any vertex inside the old boundary that does not have an optimal path within the new boundary will be visited again in the new search. The area inside the previous search that is re-searched is called _, and ideally it should be very small. In some cases, however, it is large, even as large as the previous search space itself (Figure 10). In these cases the restricted search is extremely inefficient, as the previous search space is entirely re-searched the second time.

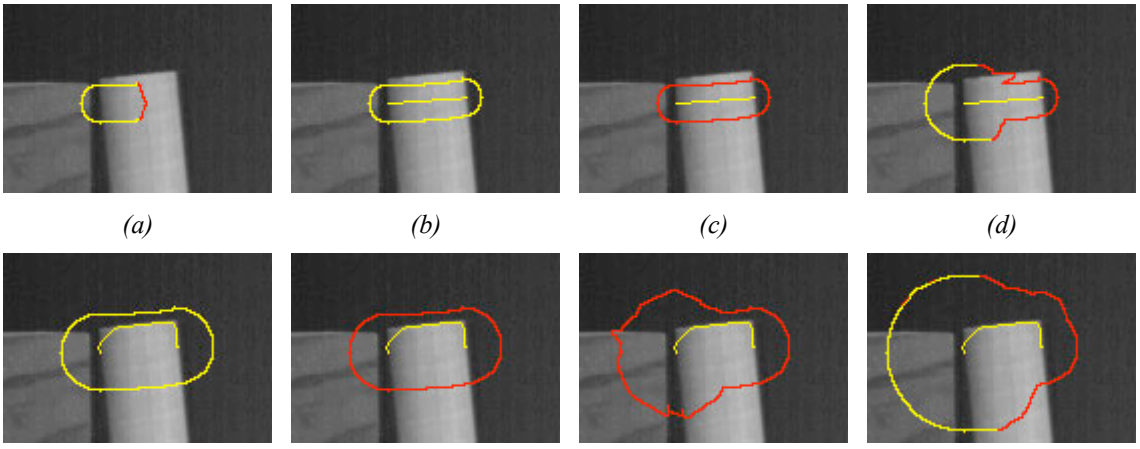


Figure 9 Expanded Search Example 1
The active list *q* is in red while the sorted list *r* is in yellow. (a) Beginning the initial search. (b) After the initial search has completed. (c) The new search with active list initialized to be *r* from the previous search. (d-h) Continuing on through the first and second expanded searches.

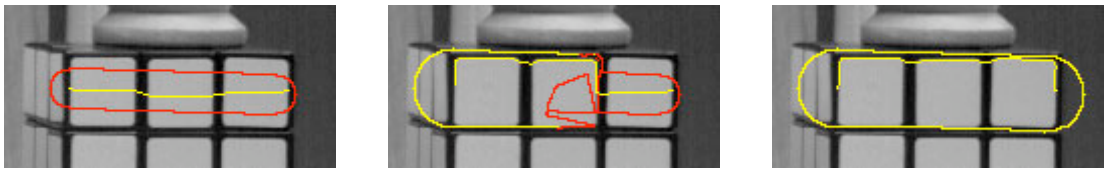


Figure 10 Expanded Search Example 2
(a) Completed initial search. (b) Almost the entire initial region is re-searched. (c) After completion of the expanded search.

3.3.2.2 Changed Search Space

Suppose a restricted search is running and the free point is suddenly moved outside of the search space. A new search must be started, ideally using information obtained in the aborted search. This can be accomplished using exactly the same algorithm as the expanded search (Algorithm 4). The only difference is that, after the free point is moved and the current search is aborted, *r* and the active list *q* must be combined to form the new active list before restarting the algorithm. In fact, *r* and *q* can also be combined in the case of an expanded search. An expanded search is never begun until the previous search is complete. *q* is empty upon completion of a search, so combining *q* and *r* would effectively yield *r*.

The area _ tends to be somewhat larger for a changed search compared to an expanded search. However, in practice, users tend not to move the free point (e.g. mouse cursor) much at all until at least an initial wire appears. At that time, if the free point is moved at all, it is usually moved either closer or further from the seed point, not in a different direction from it. If it is simply moved closer, the free point usually remains within the search space and an expanded search can take place. If it is moved further away, _ tends to be small as the new search space is close to, and often is a superset of, the old search space.

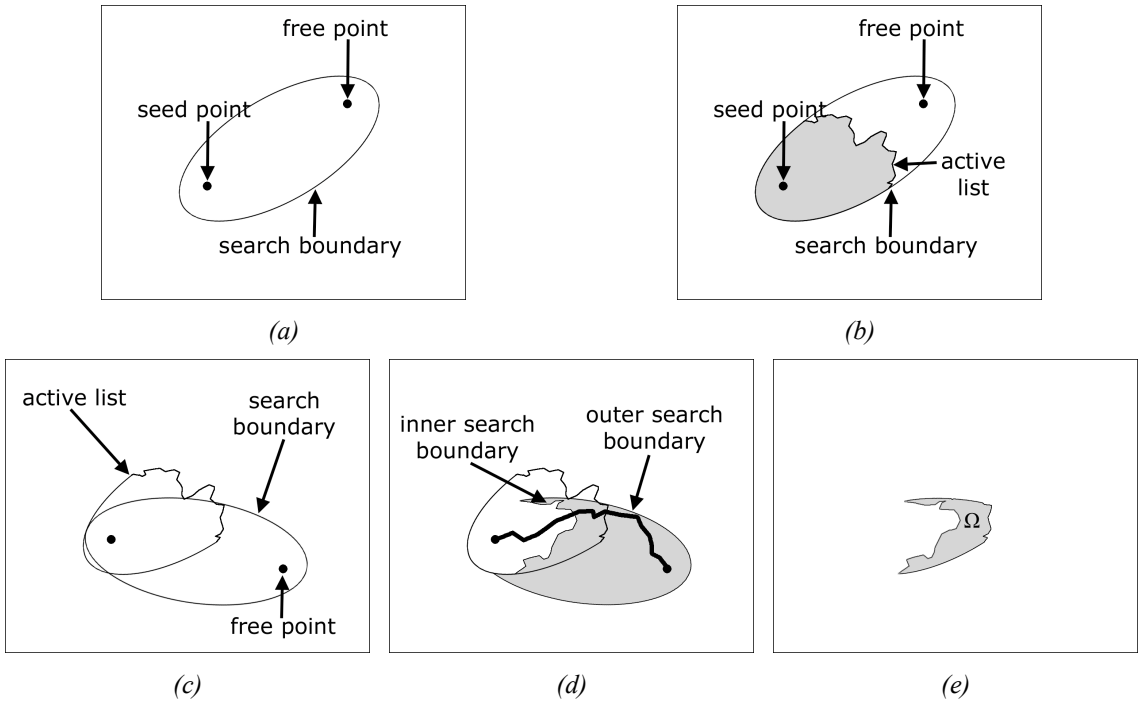


Figure 11 Changed search
(a) Setting up the initial graph search. (b) Search in progress. Searched area is shaded. (c) After the free point is moved and changed search is set up. The new active list is the union of *r* and the active list *q*. (d) Completed changed search. (e) Area _ of initial search that had to be re-searched.

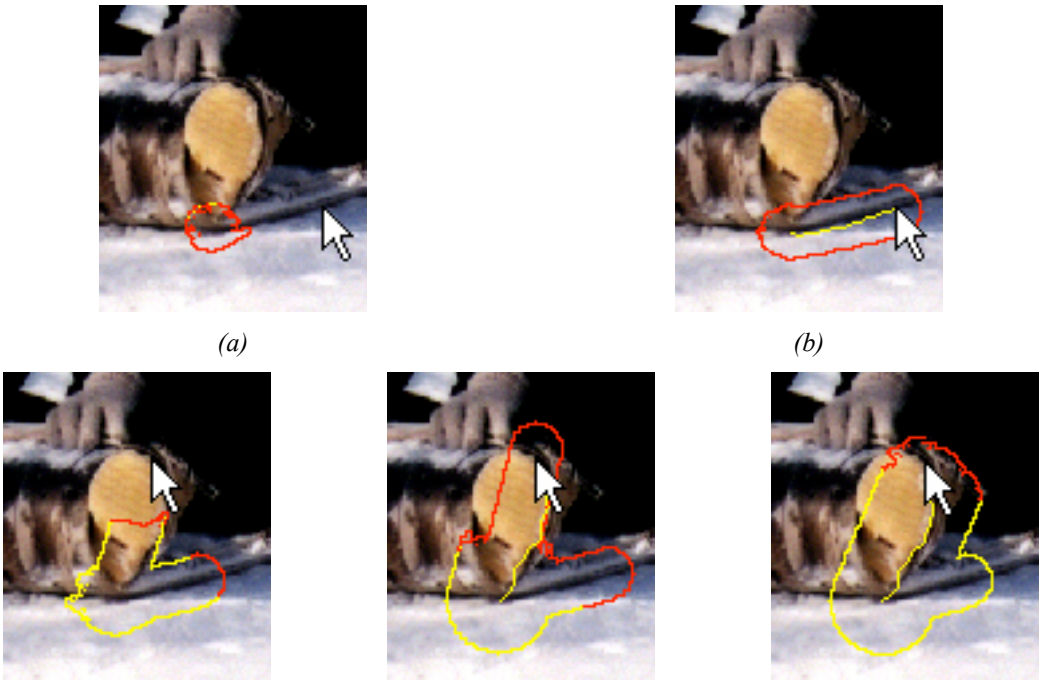


Figure 12 Changed search example
The active list *q* is in red while the sorted list *r* is in yellow. (a) At the very beginning of the initial search. (b) At the beginning of an expanded search. (c) A changed search in progress. (d) At the beginning of an expanded search after the changed search completes. (e) Expanded search in progress.

Chapter 4 Live Mesh

As Live Wire allows users to pull out wires that snap to 2D object boundaries, Live Mesh provides a way for users to stretch mesh patches (Figure 13c) over a 3D boundary. These patches fill in between three user-defined points, looking something like a rough triangulation, except that the triangle patches snap to the boundary's contour.

4.1 Method

The Live Mesh method utilizes the power of Dijkstra's algorithm to form a mesh from multiple wires. First, the user picks a point s_0 in the volume and then moves the cursor with trailing live wire w_{10} until a contour of arbitrary length is defined by picking the ending point s_1 (Figure 13a). While defining this wire, the graph search is not only finding the shortest-cost path from s_1 to the seed point s_0, but also finds the shortest-cost path from every voxel within the search space (ideally defined by the entire image, but possibly bounded by a restriction). That is, after the graph search completes we are left with shortest-cost paths reaching back from each pixel in the search space to s_0.

After picking s_1, the user moves the cursor to define the next wire w_{21}. A new graph search is started with s_1 as the seed pixel and wire w_{21} is displayed in real-time (Figure 13b). As mentioned above, Dijkstra's algorithm has already found shortest-cost paths from pixels surrounding the original seed point s_0, and so, with no additional path searching we can display shortest-cost paths leading to s_0 from every pixel defining wire w_{21} (Figure 13c).

Once point s_2 has been chosen, thereby fixing the mesh s_0 s_1 s_2, the user moves the cursor and a new mesh drags out, using graph searches that have already been completed or are well in progress. If the free point s_3 is close to s_1 then the new mesh is defined as shortest-cost paths from every point in w_{32} back to s_1 (Figure 13e). If s_3 is closer to s_0 than s_1, the new mesh is made of paths from w_{32} back to s_0 (Figure 13h). s_0 and s_1 both work equally as well as seed points for a wire mesh as they both have expanded their wavefronts during the definitions of w_{10} and w_{21}, respectively.

The user continues defining meshes in this way until the surface is covered. The fact that little or no extra graph searching needs to occur to define the extra wires forming

the mesh makes this algorithm very powerful. Once the graph search out from s_0 has been completed, any two points s_1 and s_2 in the entire image can compute a wire w_{21}, and every point in w_{21} will immediately have a shortest-cost back to s_0.

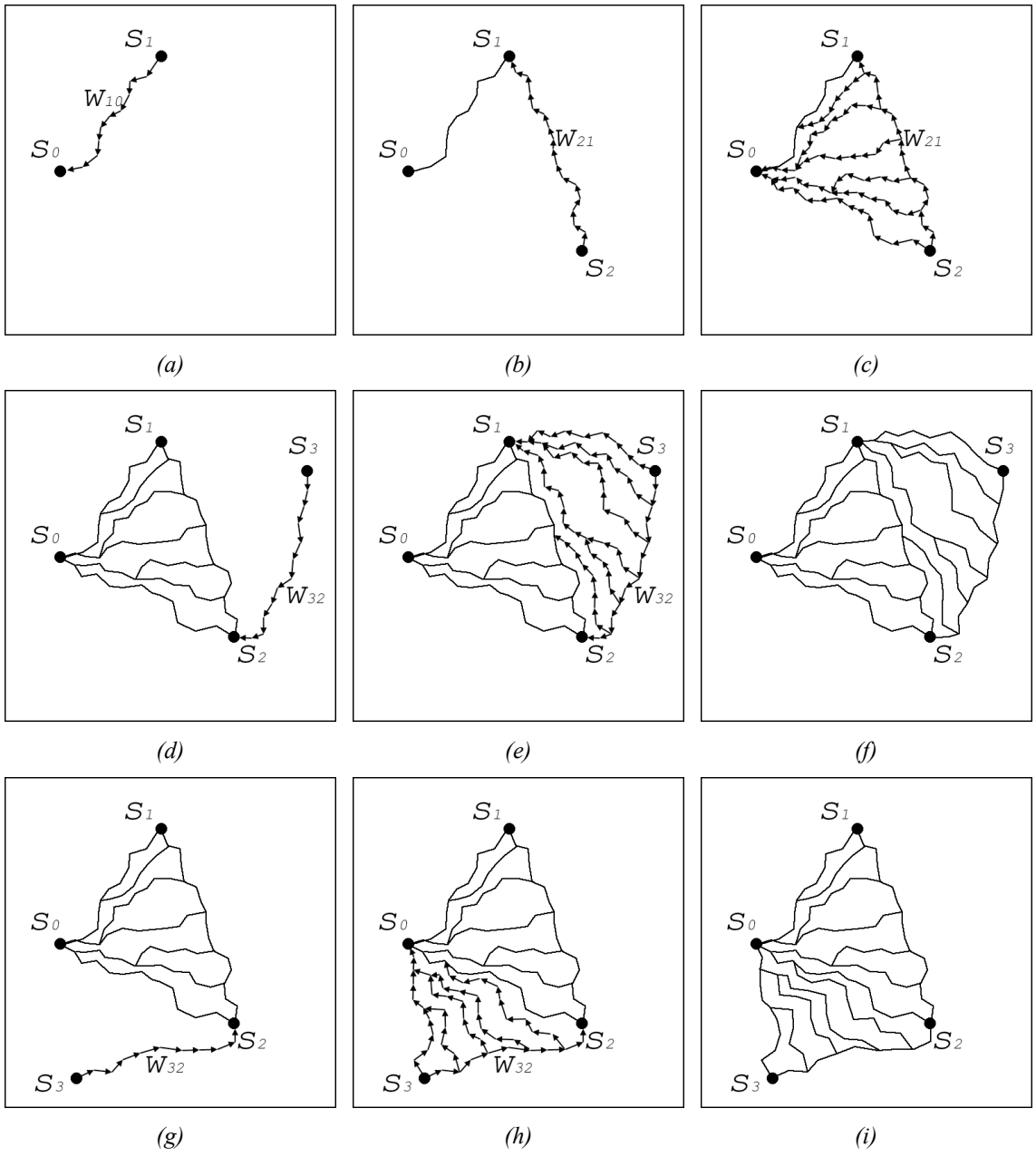


Figure 13 Live Mesh Diagram

(a) Initial Live Wire with s_o as the seed and s_1 as the free point. (b) s_2 is the new free point. (c) Shortest-cost paths to s_0 have already been computed so a mesh is formed from w_{21} to s_0 with only overhead. (d-f) A new mesh from w_{32} to s_1. (g-i) Free point s_3 was moved from its previous position and new mesh goes back to s_0 rather than s_1 since the new free point is now closer to s_0.

The end result, once the user has covered the object in meshes, is a set of points that lie on the object boundary. Exactly how an object is segmented using these points is highly application-specific. The application may simply use the points as a set, or it may use topological information found in the individual wires (e.g. (102,58,10) is linked to (101,58,10) which is linked to (100,57,11)...).

4.2 Parallelism

As the user moves the cursor to position s_2 there are two graph searches required: one with s_0 as the seed and one with s_1 as the seed (Figure 13c). Ideally the s_0 graph search will already have completed sufficiently to include any points needed during movement of s_2 to display the mesh. However, in practice this is often not the case. The s_0 graph search often has catching up to do, thus requiring two graph searches to be running at the same time.

Any application implementing Live Mesh should be multi-threaded at least sufficiently to allow two simultaneous graph searches. w_{21} is not displayed until the s_1 graph search has reached s_2 (Figure 13b). Wires in the mesh from w_{21} to s_0 are displayed as the s_0 graph search reaches each point in w_{21} (Figure 13c).

This affords an opportunity for parallelism. A computer with two processors can be running the two current graph searches in parallel. A computer with N available processors can run graph searches expanding from the last N seed points in case they are needed later. Depending on the Live Mesh implementation, virtually any previously-selected point could act as a seed point for a later mesh. In the current Live Mesh implementation, seed points are restricted to the three points comprising the last mesh. In Figure 13f, only s_1 and s_2 are candidates for subsequent meshes. In Figure 13h, s_0 and s_2 are candidates. The current implementation would make use of only two processors, but implementations without this restriction could make use of many processors.

4.3 Interpolation

While Live Mesh can form a dense mesh, some pixels are still missed. Each wire in the mesh attempts to find the shortest path to the seed pixel, and pixel boundaries of high cost are avoided, even if these are within the bounds of the mesh. When this occurs,

the mesh has visible gaps. It is expected that the user be trained to back up and attempt smaller, more dense meshes if large holes appear in the mesh. The ability to do this is an inherent characteristic of the Live Mesh tool. But it is almost inevitable that meshes will have some few pixels lost during segmentation.

(a) *(b)*

Figure 14 Iso-surface Generation from Interpolated Meshes
(a) Wire display. (b) Iso-surface display.

Depending on the application, these gaps can either be ignored or handled in some other way. The current Live Mesh implementation fills these gaps for display purposes by using a triangulation isosurface generation technique. An isosurface is found by triangulating with points found in each wire comprising the mesh:

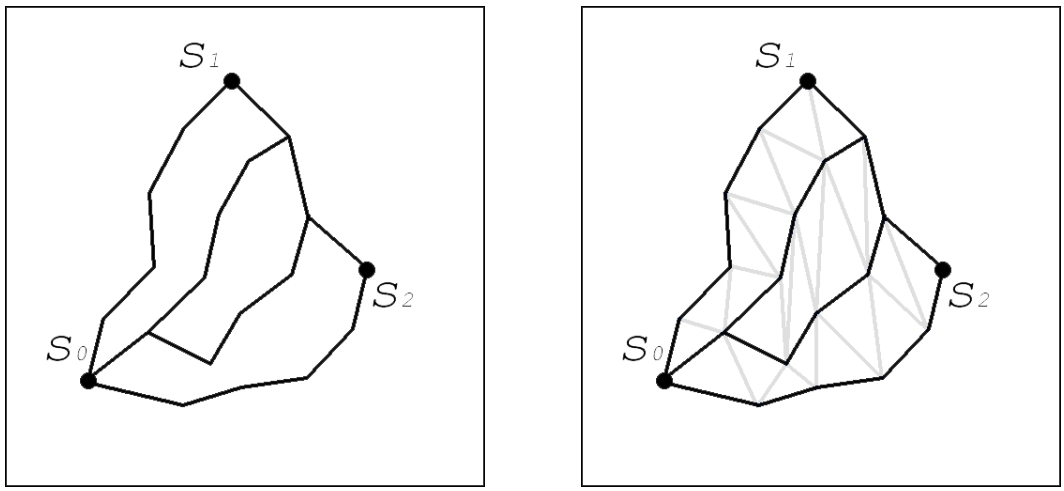
Algorithm 5 Triangulation

```
Description: Triangulates two wires to form an isosurface
between them.
Input: List wire1, wire2
Output: List trianglePts

(1) List l, reverse2, trianglePts
(2) Point p, q
(3) for every Point p in wire1
(4)    add p to l
(5) end for
(6) reverse2 = wire2 points in reverse order
(7) for every Point p in reverse2
(8)    add p to l
(9) end for
(10) p = first Point in l
(11) q = last Point in l
(12) while p is not equal to q
(13)    add p to trianglePts
(14)    add q to trianglePts
(15)    p = Point after p in l
(16)    q = Point previous to q in l
(17) end while
```

The algorithm takes two wires, which are each simply a list of points, and forms one list of points out of the wires placed back-to-back (lines 3-9). It then steps along the list from the first point forward (*p*) and from the last point backward (*q*), adding first *p* and then *q* to the list of triangle points in each iteration. The strip of triangles is defined by this list of points (Figure 15). The first three points in the list define the first triangle. The 2nd, 3rd and 4th points define the second triangle, and so on.

Our Live Mesh implementation, the *SimpleSeg* application, uses this method only for display. The triangles found by the algorithm are excluded from the final set of points defining the object boundary. However, depending on the application, this method of filling gaps could be useful in the segmentation of the object. If this method is used for final interpolation, then care should be taken to handle degenerate triangles. *SimpleSeg* ignores this possibility, as the adjacent wires have very close to the same number of points, so the few degenerate triangles that existed had little effect on the display.



(a) *(b)*

Figure 15 Mesh Triangulation
(a) A simple mesh. (b) Triangulation of the mesh shown in gray.

Chapter 5 The *SimpleSeg* Application

The previous chapters discuss the Live Mesh algorithm and its powerful features. This chapter describes *SimpleSeg*, an application implementing the Live Mesh algorithm. As the name implies, it is a simple, first-cut at showing the abilities of the Live Mesh method. This chapter describes visualization and picking techniques followed by a report of the results of a small user study conducted using *SimpleSeg*.

5.1 Data Visualization

As Live Mesh deals with the data in three dimensions, a way for the user to interact with the data in three dimensions was needed. Volume rendering is a technique for visualizing a 3-dimensional set of data in a 2-dimensional environment (e.g. a computer screen). *SimpleSeg* uses volume rendering to display a 3D view of the dataset.

For a given volume V, a ray is passed through the volume for each pixel on the computer screen. As the ray passes through the volume it accumulates the intensities of each voxel it intersects. Once the ray has passed through the entire volume the accumulated intensity is scaled and the pixel is displayed. If a ray for a particular pixel passes through a very dense portion of a volume, the pixel will be clipped to a maximum intensity, denoting an opaque portion of the image.

Volume rendering has the benefit of allowing users to interact with the volume in a quasi-3D environment. The algorithm is real time so the user can zoom, pan and rotate the volume.

Volume rendering is well adapted for visualizing the object of interest. The entire outer boundary of the object can be seen. However, data deep within the object may not be seen as the rays passing through the volume may reach complete opacity before reaching the inner data. *SimpleSeg* provides, in addition to the volume-rendered window, three orthogonal 2D slices of the volume are displayed (Figure 16). The upper-left window in the display is the “front view.” The slice is defined with the z-axis as its normal and is parallel to the view plane. The lower-left window is the “left side view.” That slice is defined as the plane with the negative x-axis as its normal. It is the view as if the user were looking at the volume from the left-hand side. The upper-right window

is the "top slice," or the view from above. Its plane is defined with the y-axis as its normal.

While these slices can be thought of as views of the data from different angles, they are not simply rotated views of the volume rendering. They are 2D images comprised of every voxel intensity value lying in a plane passing through the volume. With this interface the user can rotate the complete volume rendered image while seeing the 2D slices of obscured data (where exactly the slice cuts through the volume is treated in Section 5.3).

These 2D slices are present to allow the user to visualize and pick obscured data. This has no effect on the underlying Live Mesh algorithm, which still searches in 3D, regardless of which views are used to visualize the data.

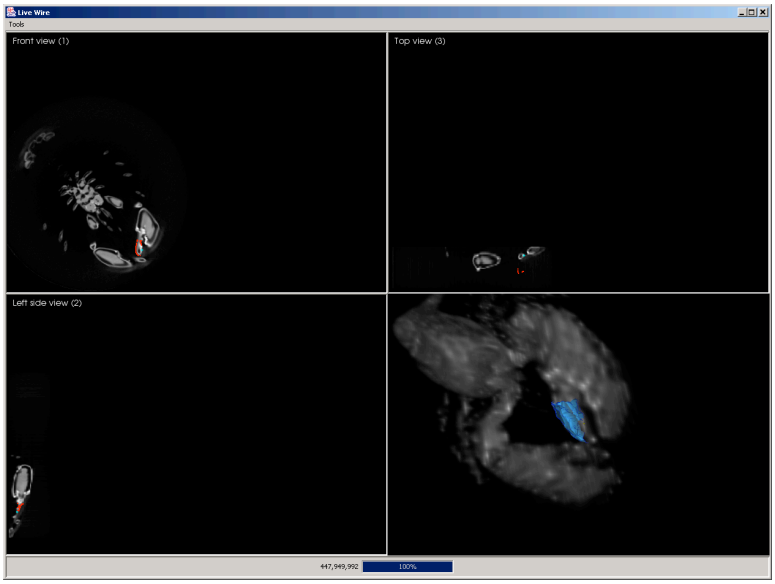


Figure 16 Live Mesh Screen
The dataset is a CT scan of a lobster. The bottom right-hand section is the volume-rendered image. The other three windows are orthogonal slices passing through the volume.

5.2 Picking

Picking is the process by which the user communicates which points to use as the seed and free points in the graph search. When working on a 3D dataset, a voxel can't be picked simply by clicking on the image. Suppose the user is working on a 32x32x32 image in a volume rendered environment. If he clicks on the view at display coordinates

(16,16) there are 32 possible voxels that could be interpreted as being picked. Determining which voxel the user intends to be chosen as a seed or free point is accomplished in two ways in Live Mesh: via the volume-rendered window or through one of the 2D slices.

5.2.1 Picking in the Volume Rendered Window

In the above example, the user clicks on pixel (16,16) of the display. There are 32 voxels lying along the ray parallel to the z-axis and passing through display pixel (16,16). For simplicity, the display x and y coordinate system is assumed to be identical to the data x and y coordinate axes. These voxels lying along the ray originating at display coordinate (16,16) are $(16,16,0),(16,16,1),\ldots,(16,16,31)$. The Live Mesh tool uses a very simple algorithm of following voxels along the ray until a voxel of a certain threshold intensity is reached:

Algorithm 6 3D Picking

```
Description: Finds the first voxel in a ray that exceeds a
threshold.  x and y are the picked values in 2D display
coordinates.  zmax is the maximum z value of the image.
Input: Number x, y, zmax, threshold
Output: Number picked_z

(1) for every z from 0 to zmax
(2)    if intensity{x, y, z} > threshold
(3)       picked_z = z
(4)    end if
(5) end for
```

Obviously the *threshold* value is highly data-dependent. *SimpleSeg* chooses a default intensity of 100, but it can be adjusted manually.

This picking method is useful in Live Mesh only on datasets in which the boundary to be segmented is visible or is defined by voxels whose intensities are higher than those of neighboring voxels. More sophisticated 3D picking techniques using image properties other than intensity (e.g. gradient, neighborhood gradient, neighborhood cost) could be used. However, they have disadvantages. With the simple intensity algorithm, the user has but one parameter to control, and a very simple one at that. As the picking technique gets more sophisticated, the number and complexity of parameters grows and by nature becomes more difficult for the user to control. This may not be undesirable in a

more automated system, but Live Mesh's effectiveness depends on a very high degree of user-interactivity, and so the user must have as much understanding of and control over what is happening as possible.

5.2.2 Picking with the 2D Slices

In addition to 3D picking, Live Mesh has a powerful 2D picking method which gives the user complete control over which voxel is picked.

As explained in section 5.1, the Live Mesh user interface has 2D views of the data. These views can be used for picking. For example, suppose the "front view" slice is positioned at z-value 10. Thus it is the place parallel to the x and y axes passing through the point $(0,0,10)$. As the user moves the cursor on that image, each (x, y) point on the image is picked and thus the 3D point picked is $(x, y,10)$. This is done similarly with the other two slices.

5.3 Positioning the Slices

Each 2D slice occupies a position in the volume. This position is determined by the current pick point. Points are picked as the user moves the mouse cursor along the 3D volume rendered window. For example, if the user picks point $(56,33,112)$ using 3D picking, then the front slice would slice through $z = 112$, the left slice would be positioned at $x = 56$ and the top slice would move to $y = 33$.

The net effect of this is that as the user moves the mouse cursor around in the 3D view all three slice views change. As the user moves the cursor around in one of the slice views, only the other two slices change. If the user moves the cursor in the front view, the (x, y) pick position is changing, so the left and top views change. But the *z* position remains the same, so the front view does not change. If the cursor moves in the left slice view, the (y, z) position changes, but *x* remains the same. The behavior is similar with the top view.

5.4 Mesh Visualization

As the user creates a new mesh, each wire in the mesh is displayed in red. Once the user finishes a mesh, the wires are displayed in blue and the new, current mesh is displayed in red.

Using the isosurface generation technique described in Section 4.3 above, an isosurface can be generated from finished meshes dynamically. It has been observed that turning on the isosurface display has been very helpful to users in visualizing the mesh.

The mesh is displayed in the volume rendered window. This way the user can observe the entire mesh being constructed while the user works. Even while the user works on the 2D slices the mesh can be seen in 3D (Figure 18). If the object is obscured, then the rendering of the volume itself can be turned off and the mesh displayed alone. This is assistive especially if the user has some *a priori* idea of what the object looks like.

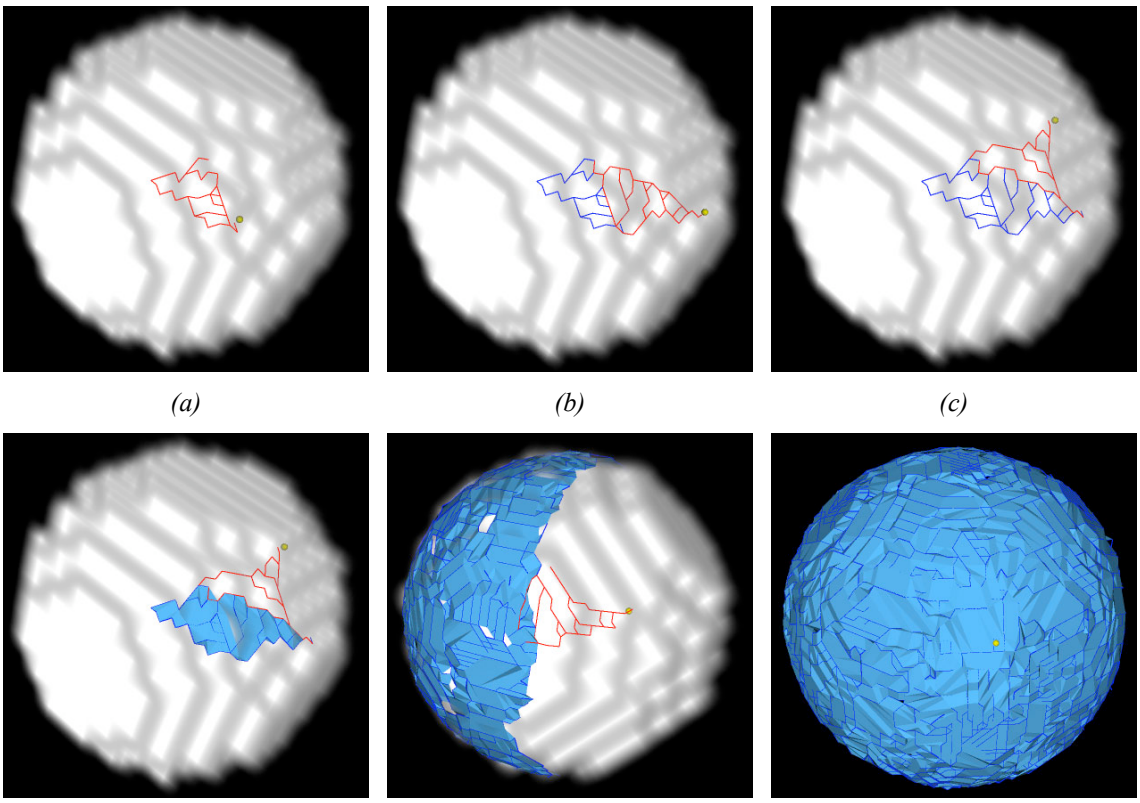


Figure 17 Mesh Visualization
(a) Dragging out an initial mesh. (b-c) Finished meshes displayed in blue; active mesh in red. (d-f) With isosurface display turned on.

(a) *(b)* *(c)*

Figure 18 Mesh visualization while working in 2D
(a) View of entire application. (b) Close-up view of the upper-left window in which the user is currently working. (c) Close-up view of the lower-left window (volume rendered) where the user observes the mesh being constructed.

5.5 User-Centered Study

We conducted a study to find out how accurate, reproducible and fast the Live Mesh algorithm is in the *SimpleSeg* implementation. We obtained accuracy results on a very simple dataset and reproducibility results from two more complicated datasets.

5.5.1 Methods

Four users participated in the study. They were given a set of instructions to tutor them through usage of both Live Wire and *SimpleSeg*. After gaining familiarity with both tools they were asked to segment three datasets. Dataset #1 was a sphere (~28 pixel radius). Dataset #2 was an MRI of a human brain. The users were asked to segment the caudate nucleus, a walnut-sized organ in the center of the brain. Dataset #3 was a CT scan of a lobster. The users were asked to segment the left "thumb."

5.5.2 Accuracy

Accuracy is how well the *SimpleSeg* segmentation describes the true boundary of the object. It is a difficult measure to obtain, because with most datasets the true boundary is not known. In fact, the true boundary is often subjective, especially if the edges are not clearly delineated. That is, one user may choose one set of voxels as the boundary, while another user may decide on a slightly different set. As such, only the

sphere dataset was used for an accuracy measure, as the dataset is artificial and the boundary is very clearly delineated.

Each user was asked to segment the sphere once slice-by-slice using Live Wire and once using *SimpleSeg*. Results are reported in Table 1. The "Num Points" column shows the total number of points that comprise all Live Wires in the segmentation. That includes not only seed and free points, but also every point in each wire. "Avg Distance" is the average Manhattan distance from each point in the segmentation to the nearest point in the actual boundary.

Table 2 shows a summarized comparison of the results of Live Wire and *SimpleSeg*. As can be seen, segmentation using *SimpleSeg* was significantly more accurate (the T-test yielded a T value of 0.000216%). Obviously the Live Mesh graph search is no more accurate than Live Wire's, as they are essentially the same. Part of the difference is because Live Mesh appears to require fewer seed points and thereby fewer opportunities for user error. As reported by Mortensen and Barrett[3], Live Wire yields significantly better results than manual segmentation, so ideally the Live Wire algorithm should be made to do as much of the work as possible, while allowing the user control to guide the algorithm. Each seed point is essentially a manual segmentation, and so limiting them can be beneficial.

Most of the rest of the difference is likely due to picking. Because of the very clear object boundary, 3D picking in *SimpleSeg* hit the actual boundary very often compared to the user's attempts to click exactly on the boundary in 2D Live Wire.

Table 2 also reveals that Live Mesh has far fewer points comprising the segmentation than Live Wire. Ideally the mesh of wires would be very dense. However, the user study revealed that users pull the mesh out as far as they possibly can before the wires lose the boundary, paying little attention to the density of the wires. And rightfully so. The user should be concerned only with how well the mesh "sticks" to the boundary and not with how many wires comprise the mesh, unless there is a large gap with an obvious missed feature. The larger the patch, the more each wire tends to run together with other wires (this is treated in Section 4.3) and the larger the gaps.

	Live Wire			*SimpleSeg*		
	Time (s)	Num points	Avg distance	Time (s)	Num points	Avg distance
User 1	557	6785	0.1892	1439	3586	0.0033
User 2	1043	6702	0.1892	2110	3871	0.0085
User 3	325	6921	0.1935	1011	3586	0.0070
User 4	295	6912	0.1934	217	2222	0.0014

Table 1 Results of sphere dataset
Time, number of points in all wires and average Manhattan distance from each point in the segmentation to the nearest point on the actual boundary.

	Live Wire	*SimpleSeg*
Time (s)	555	1194
Num points	6830	3316
Avg distance	0.191	0.005
% exact	81%	99.50%

Table 2 Sphere results summary
Direct comparison of averaged times, number of points, average distance and percent of segmentation points lying on the boundary.

5.5.3 Repeatability

Datasets #2 and #3 (MRI of a human brain and lobster, respectively) were used to obtain repeatability statistics of Live Wire compared to *SimpleSeg*. The MRI dataset was chosen because the caudate nucleus is in the center of the brain and is completely obscured when in a volume-rendered environment. This forced the users to work exclusively from the 2D slices. The lobster dataset was chosen because the users were able to work in the volume rendered window, but the object had a more interesting and rough surface than the sphere. The users were asked to segment each dataset twice using Live Wire and then twice using *SimpleSeg*. The average distance, which finds the average distance from each point in a segmentation to the nearest point in a second segmentation and vice versa, is obtained using Algorithm 7 (Appendix A) and was chosen to be the metric of repeatability.

SimpleSeg yielded a high average distance value (~1.14 voxels) compared to that of Live Wire (~0.33 voxels). This is at least partially because the meshes were not nearly as dense as expected, and as the user rarely placed the meshes in exactly the same place on the second segmentation, the wires of the meshes rarely coincided. Additionally, the user interface of *SimpleSeg* is somewhat primitive and doesn't exploit the features of the

Live Mesh algorithm as well as it could. Ideas for enhancing the user interface are presented in Chapter 6 .

5.5.4 Segmentation time

Segmentation time, on the average, increased dramatically (by about 200%) with *SimpleSeg*. *SimpleSeg* has a high learning curve, while Live Wire is natural and easy to use. However, note in Table 1 that User 4 required less time for the *SimpleSeg* segmentation of the sphere than with Live Wire with no loss in accuracy. Judging from this result, it is possible that the large time difference is due to the high learning curve for *SimpleSeg*, and that once users become experienced at *SimpleSeg* that it could actually be used to segment images faster than Live Wire. A user study involving experienced *SimpleSeg* users would be required to know if this is true.

Observation during the user study showed that working on the 3D dataset from the 2D slices was very difficult for users because they had to work with multiple, orthogonal slices, thereby making the 3D transformations in their head to re-orient themselves in different slices. Additionally, the users watched the mesh being built in 3D (Figure 18 above). Future work on the *SimpleSeg* user interface (or other implementations of Live Mesh) should concentrate on providing a way for users to work exclusively in the 3D view.

Chapter 6 Conclusions and Future Work

Live Mesh is a powerful 3D image segmentation tool, and has great possibilities for the future. The 3D cost function extension is effective for finding object edges. The restricted search and pre-processing of the cost function make the algorithm fast enough to be responsive to the user. The Live Mesh algorithm, including the idea of pulling patchwork meshes over 3D object boundaries, is an intuitive extension to the Live Wire method.

Objective results from a user study using *SimpleSeg* show that points comprising the meshes tend to be on or very close to the desired object boundary for simple datasets, though the repeatability of such needs further study. The Live Mesh algorithm provides a solid foundation for highly repeatable segmentations, and that an intuitive user interface makes this a very powerful tool.

SimpleSeg should be enhanced to allow the user to bypass the 2D slices and work solely in a 3D view. Many volume rendering techniques, including clipping and transfer functions, which allow the user to highlight and work on obscured data, are well researched and accepted in the community. Color and weighting can also be used.

Additionally, the user could be given control over a cut plane with arbitrary axes which is shown directly in the 3D window. This plane could control clipping and provide the user easier access to obscured regions for visualization and picking.

To get a more meaningful repeatability measure the segmentation needs to be more complete, with some sort of interpolating post-processing step. The interpolation scheme is data-dependent. For example, for some datasets, the convex hull may be best way to describe the final object boundary. Others may use the points of the mesh to initialize a Balloon[13]. Still others may use some sort of recursive Live Wire algorithm to fill in all of the gaps. Because it is dependent on the dataset, and a survey of final interpolation methods is beyond the scope of this work, Live Mesh does not make any attempt at determining a final interpolated segmentation.

As Live Mesh does not define any sort of interpolating post-processing for a final segmentation, a method for accomplishing this should be researched. Ideas for doing this are using the convex hull, using the mesh points to initialize a balloon or recursively

defining wires to make the mesh denser. The latter could be done as follows: choose a wire in a mesh. Step along each point in the wire looking for adjacent points that also belong to a wire in the mesh. If there only two such points (the previous and next points in this wire) then start a graph search looking for the first point belonging to any wire in the mesh. Once that point is found, define a wire from that point to the seed point. Done recursively, this could fill in holes left by the original mesh definition.

While responsive, Live Mesh could use even further speedup, and parallelism is certainly an option to be explored. Multiple graph searches could run simultaneously on separate processors, so that when shortest-cost paths are needed from any particular seed, it is likely that many or all of those paths will have already been computed.

Live Mesh is a solid 3D image segmentation tool. Its usefulness in allowing the user to work directly in 3D, while maintaining a high degree of user-interactivity has been shown.

Appendix A

Algorithm 7 Average distance

Description: Finds the average distance from each point in a segmentation to the nearest point in a second segmentation and vice versa. Sets *A* and *B* contain the points in the two segmentations.
Input: Set *A*, *B*
Output: Number *d*

```
(1) Number cnt, total, d
(2) Point q
(3) cnt = 0
(4) total = 0
(5) for every Point p in A
(6)    q = nearest point in B to p
(7)    total = total + distance from p to q
(8)    cnt = cnt + 1
(9) end for
(10) for every Point p in B
(11)    q = nearest point in A to p
(12)    total = total + distance from p to q
(13)    cnt = cnt + 1
(14) end for
(15) d = total / cnt
```

References

[1] E. N. Mortensen and W. A. Barrett, "Intelligent scissors for image composition," in *SIGGRAPH 95 Conference Proceedings*, R. Cook, ed., Annual Conference Series, pp. 191-198, ACM SIGGRAPH, Addison Wesley, Aug. 1995. held in Los Angeles, California, 06-11 August 1995.

[2] M. Kass, A. Witkin, and D. Terzopoulos, "Snakes: Active contour models," *International Journal of Computer Vision* **1**(4), pp. 321-331, 1988.

[3] E. N. Mortensen and W. A. Barrett, "Interactive Segmentation with Intelligent Scissors," *Graphical Models and Image Processing* **60**, pp. 349-384, 1998.

[4] A.X. Falcao, J.K. Udupa, S. Samarasekera, S. Sharma, B.E. Hirsch, and R.A. Lotufo, "User-steered image segmentation paradigms: live-wire and live-lane," *Graphical Models and Image Processing*. vol. 60. no. 4. pp. 233-260, Jul, 1998.

[5] A.X. Falcao, J.K. Udupa, S. Samarasekera, and F.K. Miyazawa, "An ultra-fast user-steered image segmentation paradigm: live-wire-on-the-fly," *Proceedings of SPIE on Medical Imaging*. San Diego, CA, vol. 3661, pp. 184-191, Feb, 1999.

[6] J. Liang, T. McInerney, and D. Terzopoulos, "United Snakes," *Proceedings of the IEEE International Conference on Computer Vision*. Kerkyra, Greece, vol. 2, pp. 933-940, Sep, 1999.

[7] A. Schenk, G. Prause, and H.O. Peitgen, "Local Cost Computation for Efficient Segmentation of 3D Objects with Live Wire," *Proceedings of SPIE on Medical Imaging*. vol. 4322, pp. 1357-1363, 2001.

[8] A.X. Falcao, and J.K. Udupa, "Segmentation of 3D Objects using Live Wire," *Proceedings of SPIE on Medical Imaging*. vol. 3034, pp. 228-235, 1997.

[9] P.A. Heng, K.C. Wong, T.T. Wong, and T.Y. Law, "A Freehand Volume Cutting Technique for Medical Visualization," *Proceedings of SPIE on Medical Imaging*. vol. 3658, pp. 520-527, 1999.

[10] K.D. Toennies, and C. Derz, "Volume rendering for interactive 3-d segmentation," *Proceedings of SPIE on Medical Imaging*. vol. 3031, pp. 602-609, 1997.

[11] M. Sonka, V. Hlavac, and R. Boyle, "Image Processing, Analysis, and Machine Vision," Brooks/Cole Publishing Company, Pacific Grove, CA, 1999.

[12] A. Schenk, G. Prause, H.-O. Peitgen, "Efficient Semiautomatic Segmentation of 3D Objects in Medical Images," *Medical Image Computing and Computer-Assisted Intervention, MICCAI*. pp. 186-195, Oct, 2000.

[13] D. Terzopoulos, A. Witkin, and M. Kass, "Constraints on deformable models: Recovering 3D shape and nonrigid motion," *Artificial Intelligence* 36(1), pp. 91-123, 1988.

[14] Cheevasuvit, H. Maitre, and D. Vidal-Madjar, "A Robust Method for Picture Segmentation Based on a Split-and-Merge Procedure," *Computer Vision, Graphics, and Image Processing*. vol. 34, pp. 268-281, 1986.

[15] N. Ramesh, J. H. Yoo, and I. K. Sethi, "Thresholding Based on Histogram Approximations," *IEE Proceedings Vision, Image and Signal Processing*. vol. 142, pp. 271-279, 1995.

[16] C. A. Glasbey, "An Analysis of Histogram-Based Thresholding Algorithms," *CVGIP – Graphical Models and Image Processing*. vol. 55, pp. 532-537, 1993.

[17] H. K. Liu, "Two- and Three-Dimensional Boundary Detection," *Computer Graphics and Image Processing*. vol. 6, pp. 123-134, 1977.

[18] D. L. Milgram, "Region Extraction Using Convergent Evidence," *Computer Graphics and Image Processing*. vol. 11, pp. 1-12, 1979.