Intelligent Rotoscoping: A Semi-Automated Interactive Boundary Tracking Approach to Video Segmentation

Seth R. Holladay
Brigham Young University - Provo
INTELLIGENT Rotoscopy: A Semiautomated Interactive Boundary Tracking Approach to Video Segmentation

by

Seth Richins Holladay

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

August 2007
BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Seth Richins Holladay

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

__________________________________________  William A. Barrett, Chair
Date

__________________________________________  Bryan S. Morse
Date

__________________________________________  Phillip J. Windley
Date
As chair of the candidate’s graduate committee, I have read the thesis of Seth Richins Holladay 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

William A. Barrett
Chair, Graduate Committee

Accepted for the Department

Parris K. Egbert
Graduate Coordinator

Accepted for the College

Thomas W. Sederberg
Associate Dean, College of Engineering and Technology
ABSTRACT

INTELLIGENT ROTOSCOPING: A SEMI-AUTOMATED INTERACTIVE BOUNDARY TRACKING APPROACH TO VIDEO SEGMENTATION

Seth Richins Holladay
Department of Computer Science
Master of Science

Video segmentation is an application of computer vision aimed at automating the extraction of an object from a series of video frames. However, it is a difficult problem, especially to compute at real-time, interactive rates. Although general application to video is difficult because of the wide range of image scenarios, user interaction can help to reduce the problem space and speed up the computation.

This thesis presents a fast object-tracking tool that selects an object from a series of frames based on minimal user input. Our Intelligent Rotoscopying tool aims for increased speed and accuracy over other video segmentation tools, while maintaining reproducibility of results. For speed, the tool stays ahead of the user in selecting frames and responding to feedback. For accuracy, it interprets user input such that the user does not have to edit in every frame. For reproducibility, it maintains results for multiple iterations.

Realization of these goals comes from the following process. After selecting a single frame, the user watches a speedy propagation of the initial selection with
minor nudges where the selection misses its mark. This allows the user to “mold” the selection in certain frames while the tool is propagating the fixes to neighboring frames. It has a simple interface, minimal preprocessing, and minimal user input. It takes in any sort of film and exploits the spatial-temporal coherence of the object to be segmented. The tool allows artistic freedom without demanding intensive sequential processing. This thesis includes three specific extensions to Intelligent Scissors for application to video:

1. Leapfrogging, a robust method to propagate a user’s single-frame selection over multiple frames by snapping each selection to its neighboring frame.

2. Histogram snapping, a method for training each frame’s cost map based on previous user selections by measuring proximity to pixels in a training set and snapping to the most similar pixel’s cost.

3. A real-time feedback and correction loop that provides an intuitive interface for a user to watch and control the selection propagation, with which input the algorithm updates the training data.
ACKNOWLEDGMENTS

A special thanks goes to my family, especially my wife, Kira, who always encouraged and supported me through the long amounts of work. To Dr. Barrett, for his willingness to always be of help, for his excellent counsel as my advisor, and for always having helpful answers. To my entire committee for their wise advise and their flexibility. To Brent Adams, for convincing me to start along this path. To Brian Price, my student peer, who many times was willing to discuss and help with technical aspects of my thesis. To my parents, for instilling in me the importance of education and hard work.
Contents

Acknowledgments vii

List of Tables xiii

List of Figures xvii

1 Introduction 1
   1.1 Video Segmentation .................................................. 1
   1.2 Problems With Video Segmentation ................................. 2

2 Previous Work 9
   2.1 Image Segmentation .................................................. 9
   2.2 Intelligent Scissors .................................................. 9
      2.2.1 Analysis .......................................................... 18
   2.3 Video Segmentation Techniques ................................. 19
      2.3.1 Edge-based Tracking ........................................... 19
      2.3.2 Region-based Tracking ........................................ 24
   2.4 Fast, Moldable Tool ............................................... 28

3 Intelligent Scissors for Video 31
   3.1 Considerations for Extending Intelligent Scissors to Video .... 31
   3.2 Overview of the Intelligent Rotoscoping Algorithm ............ 33
   3.3 Extensions to Intelligent Scissors’ Data Structure ............ 36
      3.3.1 Cost Map Calculation .......................................... 36
      3.3.2 Stored Data Structures ....................................... 42
      3.3.3 User Interface .................................................. 43
3.4 Leapfrogging .................................................. 45
3.5 Training .......................................................... 53
  3.5.1 Data Structure Setup for Training ...................... 56
  3.5.2 PixelID Creation and Feature Calculation ............. 57
  3.5.3 Train on (x,y) Position .............................. 63
  3.5.4 Algorithm for Setting Up Training Histogram ...... 64
  3.5.5 TrainingMap Data Structure ....................... 65
  3.5.6 Training Algorithm ................................. 70
  3.5.7 Histogram Snapping .............................. 71
3.6 Live-wire Manipulation ..................................... 76
  3.6.1 Inserting a Seed Point .......................... 77
  3.6.2 Moving a Seed Point ............................ 79
  3.6.3 Deleting a Seed Point .......................... 79
3.7 Sculptable Interface ....................................... 79
  3.7.1 Interactive Time ................................ 80
  3.7.2 Propagating Fixes ............................... 81
  3.7.3 Updating the Training Set ..................... 83

4 Results ................................................................ 87
  4.1 Example Video Sequences .................................. 87
    4.1.1 Sequence 1: Cookie Monster ..................... 88
    4.1.2 Sequence 2: Cinderella ......................... 89
    4.1.3 Sequence 3: Red Statue ....................... 96
    4.1.4 Sequence 4: Waving Girl ...................... 99
    4.1.5 Sequence 5: Ballerina ......................... 102
  4.2 Discussion and Comparisons ............................. 104
  4.3 Cost Map Comparison .................................... 106
    4.3.1 Masking Gradient/Laplacian vs. Adding Gradient/Laplacian . 106
    4.3.2 Computational Cost ............................ 108
    4.3.3 Other Implications for Video .................. 109
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Cookie Monster Sequence Results</td>
<td>88</td>
</tr>
<tr>
<td>4.2</td>
<td>Cinderella Sequence Results</td>
<td>90</td>
</tr>
<tr>
<td>4.3</td>
<td>Red Statue Sequence</td>
<td>96</td>
</tr>
<tr>
<td>4.4</td>
<td>Waving Girl Sequence Results</td>
<td>99</td>
</tr>
<tr>
<td>4.5</td>
<td>Ballet Sequence Results</td>
<td>103</td>
</tr>
<tr>
<td>4.6</td>
<td>Data Comparison With Keyframe-based Rotoscopying</td>
<td>105</td>
</tr>
<tr>
<td>4.7</td>
<td>Data Comparison With Video Cutout</td>
<td>106</td>
</tr>
<tr>
<td>4.8</td>
<td>Cost Map Addition vs. Masking</td>
<td>107</td>
</tr>
<tr>
<td>4.9</td>
<td>Preprocessing Times</td>
<td>109</td>
</tr>
<tr>
<td>4.10</td>
<td>Leapfrogging Accuracy</td>
<td>112</td>
</tr>
<tr>
<td>4.11</td>
<td>Leapfrogging Speed</td>
<td>114</td>
</tr>
<tr>
<td>4.12</td>
<td>Leapfrogging Speed</td>
<td>125</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Foreground/background distinction ........................................... 2
1.2 Image Segmentation and Compositing Example .............................. 3
2.1 Intelligent Scissors Example .................................................. 10
2.2 Gradient Magnitude Image .................................................... 12
2.3 Laplacian Image ........................................................................ 13
2.4 Intelligence of Intelligent Scissors in the Boundary Snapping .......... 17
2.5 Intelligent Scissors Results Graphs ........................................... 18
2.6 Keyframe Rotoscoping Example ................................................ 21
2.7 Keyframe-based Rotoscoping Results Chart ................................ 22
2.8 Video Cutout Interface ........................................................... 28
3.1 Explanation of term “Inside Edge.” ...................................... 34
3.2 Example of Intelligent Scissors Corner Cutting. ...................... 35
3.3 Intelligent Rotoscoping Algorithm Summary. .............................. 37
3.4 Numerical example of a cost map. ........................................ 40
3.5 Higher-level view of costmap calculation. .............................. 41
3.6 User interface for Intelligent Rotoscoping ............................... 44
3.7 Automated Selection Without Leapfrogging ............................. 46
3.8 Leapfrogging implementation .............................................. 48
3.9 How Leapfrogging Cuts Through Objects ............................... 50
3.10 Polygonal Approximation of Cinderella’s Head ....................... 51
3.11 Leapfrogging With Seeds Placed by Polygonal Approximation ..... 52
3.12 Cost map of a silhouette ...................................................... 54
3.13 The PixelID Struct and Corresponding Histogram ................... 59
3.14 Features used for setting up a PixelID. ................................ 60
3.15 Data structure used for the training histogram. ......................... 66
3.16 PixelID scaling process ................................................. 68
3.17 Example of scaling down PixelID values ................................ 69
3.18 Histogram Snapping ...................................................... 73
3.19 Live-wire editing ......................................................... 78
3.20 Propagating user fixes ................................................... 84
3.21 Conceptualization of updating the training set .......................... 85
4.1 Selected Frames of Cookie Monster’s Silhouette ....................... 88
4.2 Single Selection of Cookie Monster ...................................... 89
4.3 Selected Frames of Cinderella ............................................ 90
4.4 Single Selection of Cinderella ............................................ 91
4.5 Recoverability of Leapfrogging on Cinderella ........................ 92
4.6 Cinderella Selection With Bad Training Data .......................... 94
4.7 Cinderella Selection With and Without Training ...................... 95
4.8 Selected Frames of the Red Statue ...................................... 97
4.9 Single Selection of Red Statue ......................................... 97
4.10 Example of Training the Red Statue .................................... 98
4.11 Selected Frames of the Waving Girl ................................... 100
4.12 Single Selection of Waving Girl ....................................... 100
4.13 Single Selection of Ballerina .......................................... 103
4.14 Ambiguous Cost Map Data ............................................. 108
4.15 Motion Blur and Cots Maps ............................................ 110
4.16 High Costs in the Cost Map ........................................... 116
4.17 Polygonal Approximation Seed Points ................................ 117
4.18 Intelligent Rotoscoping Can Miss Appendages ....................... 119
4.19 Selection Training on Gradient ........................................ 120
4.20 Selection Training on Color ............................................ 121
4.21 Selection Training on Neighbor Color .................................. 121
4.22 Selection Training on the Inside Color ................................ 122
4.23 Selection Training on Gradient, Color, and Neighboring Colors ... 123
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.24 Selection Training on Gradient, Color, Neighbor Colors, and Position</td>
<td>124</td>
</tr>
<tr>
<td>4.25 Histogram Profiles for Training Set</td>
<td>127</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

A common problem in computer vision is that of image segmentation, the extraction of a foreground object or objects from the background in a photo image (Figure 1.1) [1, 2, 3, 4, 5, 6, 7, 8]. However, video segmentation, the extraction of an object from a series of frames (image extraction over time), is not a simple extension of image segmentation applied to more than one frame. Video segmentation is a much more difficult problem due to the added dimension of time. Nevertheless, the characteristics of the added constraint of time and the spatial-temporal coherence of video can be exploited to address the problem.

1.1 Video Segmentation

Imagine an assignment to film a movie charting the voyage of an enormous steamship across a sandy desert. One might film a real ship sailing through water, then cut the ship out of that footage and add it to footage of a sandy desert (Fig. 1.2). Being able to segment the ship out of the water is a non-trivial problem. This segmentation of an object over multiple frames is referred to as “video segmentation”.

Video segmentation is the process of applying image segmentation to the same object throughout a series of video frames. With the many applications of visual media to technology and communication today (entertainment, education, national security, etc.), there are various circumstances where a user desires to emphasize or isolate an object or subobject in temporally adjacent images. As in the example above, it is useful in the film industry for taking elements of one clip and compositing them into another clip to achieve shots that might not otherwise be possible. It can also
Figure 1.1: Foreground vs. background distinction. In this paper, foreground refers to the object the user desires to segment from the rest of the image or “background”. Note: all other (undesired) objects are considered background, even if they look similar to the foreground object.

be very useful for the fields of medicine (e.g. tracking a scan of a heartbeat or adding emphasis, colorization, etc. to the object), national security, traffic control, scientific data extraction, and so forth. There are many cases when something recorded on a video needs to be tracked or extracted but where the complexity or similarity of surrounding elements make this challenging. Due to the huge variety of video segments, this is an extremely broad problem.

1.2 Problems With Video Segmentation

The problems with image segmentation are compounded with video segmentation for several reasons. First, objects move and deform in a non-linear fashion, so that knowledge of movement between previous frames does not always predict the next series of frames [3, 9]. Thus, temporal adjacency/coherence from one frame to the next can provide better predictive accuracy than morphing between keyframes [10].

Another problem is that not only is the background surrounding the object changing across the object’s boundary within a single frame, but that background
Figure 1.2: Example of image segmentation and compositing. We want a ship sailing over the desert. a) The desert background image is not taken with b) the desired foreground (the ship). c) The ship only is extracted, and the green region will be replaced with our background in a). d) The final composite tells a novel story not found in any of the original photographs.
can also be constantly changing from one frame to the next. Thus an algorithm that relies on consistent border attributes between frames may be useless for object localization in frames where the local object position has changed. To effectively segment objects moving across a varying background, “training” on consistent object or background properties may be necessary. Training, or teaching the segmentation which boundaries are best to follow, will therefore likely have to depend on such attributes as those on the inside of the object edge. Even more troublesome is when background objects pass in front of the target object, occluding part or all of the target for several frames, or cutting the object in half.

Even with these problems resolved, selection time still remains a big issue, not only in the time the selection itself takes, but also in the time needed for pre- and post-processing of the images. Some techniques, such as Video Cutout [10] can select an object for 150 frames in as little as 10 to 20 seconds. However, this is assuming the object is selected in a single iteration, but typically manual corrections are needed in current video segmentation methods. Based on the results recorded in Video Cutout [10], the artist takes an average of 20 minutes to select an object to satisfaction, with a worst case of close to an hour. In all cases, it also requires about 30 minutes of preprocessing, or between 5 to 10 seconds per frame. 150 or more frames is not unusual for a sequence, but even with the most advanced tools, where each frame is requiring a few seconds on average, the user can easily lose interest in the tool.

The interface for video segmentation tools must be smarter than the interface for image segmentation tools, in terms of exploiting user input, since techniques that require manual intervention with each and every frame can become unacceptably laborious. The user will want an intuitive interface requiring minimal selection, allowing quick and simple corrections where the selection goes wrong, and ultimately giving the user the final say in what is selected. This means that the selection algorithm can allow the minimal input to have maximum benefit. Spatio-temporal cubes do well with resolving interpolation and keyframing problems [11], but the interface may not always allow for intuitive selection.
Motion blur and filming with an unstable camera can also create undesired artifacts [12, 10]. More specifically, motion blur smears the object with the background, distorting or losing data pertinent to separating foreground from background. Unstable camera moves break up the predictable motion or temporal adjacency of the object. Therefore, even if the challenging problem of tracking well-defined, temporally cohesive objects is completely solved, the video sequence is not certain to be free of other pitfalls.

Ultimately, the dilemma is that most fully automated single-frame selection tools prove too inaccurate when applied to segmenting multiple frames, requiring too much manual correction after the fact. More recent tools that show promising results are still not to a stage of application beyond the research lab because of the complexity of the algorithm, making it difficult to apply them in a practical way. Also, problematic may be the use of too many “magic numbers”, the long wait period, either in pre-processing, post-processing, or the automation of the selection itself. In actual practice in the industry, those needing to do image or video segmentation often prefer to segment out the foreground by hand, frame by frame, over having to fix incorrect selections by slow tools [13], in the spirit of getting things right the first time. If selections are always incorrect in the majority of the frames during automatic propagation, they are often not easy to fix or to steer back onto the right course. Thus, finding and correcting the erroneous portions of the automatic selection can end up being as tedious as manual selection.

Many video tracking techniques, especially the more recent ones, are robust and even fast enough for many applications, if the job is to closely approximate the boundaries and region of a desired object over time. So why do they still take undesirable amounts of effort for practical applications? Why are these techniques not flooding the markets when the implementation techniques are out there? Certainly they are being used some, but much time still gets spent on refining the selection and hand-fixing small errors in each frame, even to the point that implementation may not be worth the trouble. So why so picky with the small problems that have a good chance of being acceptable for single-image editing?
The answer is that our eyes and brain are extremely sensitive to temporal coherence, and they easily pick up inconsistencies in edges from frame to frame. This is because the tiny errors in the boundary change positions from frame to frame, resulting in pops and flickers. Our eyes miss large discrepancies when there is a break in what we are seeing (such as a camera cut in film), but on something that is a continual focus, as with object edges, they pick up very strongly on discrepancies [14]. Our brains recognize this as very wrong (not true to life), and it is distracting and annoying. For this purpose, a video segmentation tool must not just be good enough to capture the desired object, but to do so such that boundary transitions from frame to frame are smooth and consistent. In other words, consistency could be more important than accuracy. We definitely want accuracy, which should be attainable by training every frame’s selection to be consistent with the user-defined selection.

So, once the desired object is identified and propagated for video segmentation, the user needs a simple and quick interface for fixing and “sculpting” inaccuracies in the automated selection, with the corrections themselves getting propagated to neighboring frames. This pulls the user tight into the loop, as with a video game but without wearing out the user. The user should only need to perform high-level interaction and suggestions, while the software articulates and localizes the boundary.

Therefore, a tool is needed that can speed up video segmentation techniques considerably without compromising accuracy. In other words, a tool is needed that makes the best use of minimal user input while allowing the algorithm to quickly, accurately, and consistently localize the boundary. It would be desirable to have the user make high-level suggestions while the algorithm snaps to, and localizes the boundary.

The snapping and training features of Intelligent Scissors can be expanded to the temporal dimension to create just such a fast, intuitive, and flexible object-tracking/selection tool, requiring minimal user input while allowing for technical and artistic guidance where desired. The user will be able to sculpt or guide the selection in a minimal number of frames, acting and reacting with high-level mouse gestures. Propagation and localization of the boundary in temporally adjacent frames proceeds
using a snapping technique called “leapfrogging.” Progressive improvement of boundary localization in succeeding frames makes use of histogram snapping to train the boundary to adhere to the object edge selected in previous frames. These features, implemented as “Intelligent Rotoscoping,” allows video segmentation to be sped up by leveraging off of minimal user input to automatically select an object over multiple frames. By keeping the user in the selection loop, the tool allows it to stay on track through the high-level corrections that are propagated and used to adapt the selection in future frames.
Chapter 2

Previous Work

Current advancements in computer vision provide many techniques for segmenting photographic images and video, facilitating work in medicine, desktop publishing, special effects, etc. Current tools are working toward the quick, accurate, and interactive requirements needed for speed and efficiency.

2.1 Image Segmentation

Some of the more recent advancements in image segmentation include boundary editing techniques such as boundary vertex editing [1], snakes [2, 7], and Intelligent Scissors [15, 5]; region growing techniques such as Intelligent Paint [6] and graph cut [1, 16, 17]; and statistical methods such as Bayesian and Poisson [13, 8]. This thesis will refer to segmentation of a single image as “image segmentation” (Fig. 1.2).

2.2 Intelligent Scissors

Intelligent Scissors [5, 15] is an image segmentation tool that is the basis for Intelligent Rotoscoping. It works by first setting up a cost map for an image and assigning a cost to each pixel, the lowest costs belonging to pixels along object edges. During the interactive selection, the user places seed points incrementally along the desired object’s boundary and the cost map is used to create a path along the edge between those seed points. When a pixel in the image is marked by a user as a seed point and the cursor then moved to another location in the image, Dijkstra’s algorithm is used to create the least cost path from the seed point to the cursor position at interactive rates. When the user places a new seed point, the least cost
Figure 2.1: Demonstration of Intelligent Scissors technique on a single frame [5]. Cursor path (white) only need roughly approximate the object edge, an easy, quick task for user. Intelligent Scissors boundary (yellow) snaps from current cursor position to edge. Boundary sections that stick long enough “cool” to edge, becoming permanent along edge.

path from the previous seed point to the new one is finalized, and a new least cost path is interactively updated from the new seed point to the user’s cursor. This interactive rate where the computer keeps up with the user’s every move is called “real-time” interaction. The live path between the most recent seed and the user’s cursor position is called the live wire. Any least cost path between seed points is called the selection boundary.

Given the terminology, selection boundaries are calculated in real time between adjacent seed points as the user places them, giving the user real-time feedback. As the user moves the cursor, the live wire automatically snaps to edges, doing most of the selection work. Therefore, the user only needs to swing the mouse in an approximate cursor path around an object in an image, laying seed points as the selection boundary snaps to a portion of the desired edge (Fig. 2.1). The better the cost map function, the longer the selection path between two seed points, and the less effort the user has to expend.

Costs in the cost map are inversely proportional to the edge gradient. The edge gradient of an image is calculated by convolving a Sobel filter over the original
image, \( I \), in both the \( x \) and \( y \) directions. The Sobel filter computes the image's partial derivative in \( x \), \( I_x \), and in \( y \), \( I_y \), respectively, which two images can be combined by computing the following for each pixel \((x, y)\):

\[
I_G = \sqrt{I_x^2 + I_y^2}
\]  

Or in other words, the square root of the sum of squares of the partial derivative is combined to obtain the gradient magnitude, \( I_G \), of the image, assigning higher values of \( I_G \) in regions with higher gradient changes, such as edges. The equation can also be written for any vector-valued pixel, \( \mathbf{p} = \begin{pmatrix} x \\ y \end{pmatrix} \), in the image, as follows:

\[
f_G(\mathbf{p}) = \sqrt{I_x(\mathbf{p})^2 + I_y(\mathbf{p})^2}
\]  

This image exposes edges, or areas of high contrast (high gradients), in the original image. Edges are marked with a high intensity value, the highest intensity displayed as white, and the rest of the gradient magnitude image has low intensity values, much closer to zero, the lowest displayed as black (Fig. 2.2). The gradient magnitude is very helpful at identifying the strongest edges without a lot of extraneous noise. It also is useful for distinguishing one edge from another, since it retains the color and intensity information of the edges. Stronger edges have a higher intensity. The non-edge portions of the original image are assigned much lower values. The resulting lines in the Gaussian image also have some thickness greater than one pixel, depending on the size of the convolution filter and the rate of change of the gradients in the image. This sometimes can make it difficult to determine the exact pixels that lie on the object’s edge (though it approximates the local position very nicely).

Another important piece of information obtained from the Sobel images, \((I_x, I_y)\), is the gradient direction, or the direction of the edge for any given pixel. The direction of the edge is perpendicular to the tangent direction at that point. The gradient direction, for pixel \( p \), is computed as follows:

\[
f_D(p) = \tan^{-1} \left( \frac{I_y(p)}{I_x(p)} \right)
\]  

11
Figure 2.2: A gradient magnitude filter, $I_G$, (right) emphasizes the edges of an image (left). Note that the thicker edges of the letter K has a thicker gradient magnitude line than the finer edges occurring in the background texture.

A second method to calculate edges in the image is to use the zero-crossing map of the image. The zero crossing image, $I_Z$, is a binary image indicating pixels where zero crossings occur in the second partial derivative $I_{xx}, I_{yy}$ of $I$, the Laplacian image, $I_L$. $I_L$ is computed by convolving $I$ with a Laplacian filter. $I_Z$ is computed by solving the following for each pixel, $p$, in $I_L$:

$$f_Z(p) = \begin{cases} 
1; & \text{if } f_L(p) \approx 0 \\
0; & \text{otherwise}
\end{cases}$$

(2.4)

In the discrete world of digital images, selecting the zero crossing means that when two neighboring pixels with opposite signs are found, the neighbor closest to zero is turned on (i.e. change the pixel with the minimum absolute value from 0 to 1). A pixel’s neighbor is any directly adjacent, 4-connected pixel. The resulting Laplacian image identifies the exact positions of the edges in the image, but it can be misleading since it is very sensitive to noise in the image and may identify undesired or nonexistent edges. The Laplacian can be based off of a single, grayscale channel of an image, or
Figure 2.3: A Laplacian filter (right) emphasizes the edges of an image (left). It does this by marking the pixels where the second derivative of the image crosses zero. This is done separately for each color channel. For each pixel, each color channel is marked with either a one (white) or a zero (black).

Intelligent Scissors creates a cost map based on a weighted combination of the Gradient magnitude (Eq. 2.2) and the Laplacian (Eq. 2.4) as given in Equation 2.5 [15], where $w_Z$ and $w_G$ are the predetermined weighting functions. Here is the equation, represented for each pixel $p$ in the image, with each term assigned its own pre-determined weight, $w$:

$$C(p) = w_Z.f_Z(p) + w_G.f_G(p)$$  \hspace{1cm} (2.5)

In the resulting image, $I_C$, the edges have the highest values and the uniform regions have the lowest. For the purposes of Intelligent Scissors, the lowest costs should belong to the edges so $I_C$ is inverted, and zero values are reassigned a positive value close to zero. Thus, the cost map is inversely proportional to the gradient image.

This takes advantage of the strengths of each edge operation: $I_G$ identifies the
best edges with inconsequential noise and $I_Z$ identifies the precise edge pixels (no thicker than one pixel wide). $I_G$ tends to receive a stronger weight than $I_Z$, since the extraneous edges from noise in the zero crossings can cause the user selection to follow spurious paths and cut corners through objects. Nevertheless, the preciseness of $I_Z$ is needed to localize the boundary. Different methods of combining the gradient magnitude and Laplacian images can be used to minimize the noise, such as the use of multiplication instead of addition. The use of other image properties, in addition to gradient magnitude and zero-crossing, can also improve the accuracy of the Intelligent Scissors selection, especially when it comes to training. Some of those additional properties include gradient direction, $I_D$, edge pixel value, $I_P$, neighboring pixel value in the foreground (inside), $I_I$, and neighboring pixel value in the background (outside), $I_O$.

$$C(p) = w_Z.f_Z(p) + w_G.f_G(p) + w_D.f_D(p) + w_P.f_P(p) + w_I.f_I(p) + w_O.f_O(p)$$  \[(2.6)\]

The 1995 Siggraph paper on Intelligent Scissors [5] narrows down this equation to the following, for any given pixel $p$ and neighboring pixel $q$:

$$\ell(p, q) = \omega_Z \cdot f_Z(q) + \omega_G \cdot f_G(q) + \omega_D \cdot f_D(p, q)$$  \[(2.7)\]

Note that even though additional attributes are used to increase the accuracy of the cost map, the data is still flattened down into a single cost map. For training purposes, this can lead to training errors when two completely different pixels in the image add up to the same value in the cost map.

The foregoing yields a static cost map and occurs in a preprocessing step before the interactive stage. Since the cost must be calculated for each pixel in the image, the convolution can be expensive for computing $I_G$ and $I_Z$. The simplest convolution involves a 3x3 filter for both images, which means that a video-sized image (720x486 pixels, or approximately 350,000 pixels) can require nine multiplications, eight additions, and a division for each pixel. With a 2GHz computer, the time to preprocess such a video image averages between one to two seconds. This is very acceptable for a single image, but it still means a minute of preprocessing for just
a couple of seconds of video. It would be preferrable if the preprocessing required little or no wait, but some amount of preprocessing is also forgivable if it means the selection process allows immediate response and feedback to the user.

The preprocessing for Intelligent Scissors takes into account the possibility of various edge widths. In other words, the gradient width or "blurriness" of an edge can vary significantly from image to image, or even within an image. For sharp edges, a small-sized convolution filter (i.e. 3x3 or 5x5) produces an accurate edge image. Blurrier edges extend across multiple pixels, for which larger convolution filters result in the cleanest gradient magnitude image. The method Intelligent Scissors uses to account for varying edge width within an image is to compute the Gaussian and Laplacian images with multiple convolution filter sizes. For each pixel, the value from the filter producing the best result is stored in the final edge image. While this method obtains the most desirable edges, it compounds the problem of lengthy preprocessing times. With current computing speeds, this compounded time is undesirable for preprocessing tens or hundreds of frames.

The precomputed cost map is used to form those selection boundaries by expanding a wavefront from the manually placed seed points to calculate the minimum cumulating cost path. In other words, the lowest cost pixel in the expanding wavefront is popped off and used to compute its neighbors' cumulated cost, adding each neighbor's cost to the popped pixel's cumulated cost. Then the neighbors are pushed onto the wavefront.

As the wavefront expands, the boundary snaps to the lowest cost edge between the last placed seed point and the user's cursor. Therefore, as long as the cost map correctly assigns low costs to the desireable edges, the live-wire is more likely to follow the correct edges. Thus, Intelligent Scissors forms a globally optimal path, like a river following the low altitudes of a ravine, in a way that is locally robust for finding edges, a property that is especially critical when expanding Intelligent Scissors into the time domain.

Therefore, the primary intelligence behind Intelligent Scissors is the snapping of boundaries between the seeds, not in the placement of the seeds themselves, which
is done by the user. A cost map with accurate edges and with little or no noise guarantees that this snapping occurs desirably. This means the user needs to only place a few well-positioned seeds instead of many seeds to assure the rest of the boundary snaps properly (Fig. 2.4).

The natural effect of the gradient image on the cost map is to give strong edges the lowest, or best, cost. However, the user may want the live-wire to snap to weaker edges and avoid strong edges in the vicinity. Intelligent Scissors has a powerful, simple solution to train the live wire to snap to edges similar to those already selected by the user. It remaps the cost mapping function to give the lowest costs to values in the cost map similar to pixels already selected. The remapping function is the inverse of the histogram formed by the pixels in the cost map that line up with the selection. Therefore, the training is done by means of an order-one lookup. This concept of training can also be helpful over time, to be able to train the best costs consistently to a given object.

A speed-up for the live wire and training is to calculate costs only along watershed region boundaries [17, 12]. A watershed is a small region of an image that is formed by joining neighboring pixels based on shared proximity to the locally minimum-cost pixel, like water dumped over a topologically rough terrain would slide downhill and pool in the closest watershed. From each pixel, a path is followed through the lowest cost neighbors until reaching a minimum, the pixel whose neighbors all have a higher cost than itself. The boundaries between watershed regions in the image correspond to edges in the image. Since boundaries are what is needed for object selection with Intelligent Scissors, watersheds can greatly reduce the search space by only using pixels along watershed boundaries. This can reduce the costmap size to less than half of the number of pixels in the image. However, the disadvantage is the possibly missed edges that fall in the middle of a watershed as well as the increased preprocessing time and memory requirement for the watersheds. For application to video, this could lose too much valuable information and take too much preprocessing time.
Figure 2.4: The intelligence of the Intelligent Scissors algorithm lies in the boundary snapping algorithm, which depends on a strong cost map. a) A noisy costmap may cause the boundary to snap to non-existent or undesired edges unless many more seeds are placed, whereas b) a well-calculated cost map leads to a correct selection of the K with fewer seed points placed by the user.
2.2.1 Analysis

Thus, Intelligent Scissors is a big time saver over manual selection, since the user need only place a few seed points instead of selecting every pixel on the boundary. Speed comparison tests show that Intelligent Scissors is multiple times faster, with significantly better accuracy (Fig. 2.5). The manual selection could be as accurate as the Intelligent Scissors, but with a significant time loss.

The advantage of Intelligent Scissors is that it immediately displays the selection as seeds are placed. In object selection algorithms requiring little or no human intervention, it generally requires a long wait to make the selection. In the end, these algorithms may do a better job in one way or another, such as with extracting intricate details, but they require a much larger time cost. The worst part of it is that if the automated algorithm does a bad job, parameters must be reset or approximated and the user must wait patiently again before receiving more feedback. On the other
hand, with a little seed placement here and a small nudge there, Intelligent Scissors
does the job satisfactorily within a minute’s time. Also, due to its snapping prop-
erties, Intelligent Scissors lends itself nicely to non-linear movement since it is not
biased toward, translation, rotation, or scale, but only to proximity.

2.3 Video Segmentation Techniques

For video segmentation and compositing in the movie industry, the desired
object is often filmed in front of a bluescreen for easy isolation and composition in
post processing \[13, 18\]. However, bluescreening is often not an option, such as in
preexisting footage or with ocean-sized backdrops, so other solutions for foreground
extraction have been developed.

2.3.1 Edge-based Tracking

A general technique for object selection through time, being of special interest
for this thesis, is rotoscoping. Rotoscoping is the process of tracking an object frame-
by-frame based on its boundary’s attributes (as opposed to tracking an object based
on the attributes of its inner region).

Manual Selection

There are many methods for rotoscoping, the most simple and common process
being to hand-trace a selection around the boundary one frame at a time \[13, 19\].

Splines

Manual tracing is often facilitated with geometric tools, such as bezier or spline
(mathematically defined) curves \[20\]. Splines allow the user to do vertex-based editing
of the selection boundary instead of having to hand pick each and every boundary
pixel \[21\]. This is an extremely important notion, that the user does not necessarily
need to touch every pixel to have absolute control over what pixels are selected. In
other words, pulling a single control point of a bezier curve automatically pulls many
pixels on either side to where the user wants them. Still, manual selection in each and
every frame can be long and tedious, even with splines. For example, it might require a user 60 seconds per frame to select an object, which is generously fast work, and footage generally runs at 24 frames per second. If the user had to select an object out of 10 seconds of footage, that would add up to 240 frames, or 240 minutes of work. Four hours selecting out 10 seconds of footage, with 20 minutes still to process, is not a quality use of time. Nevertheless, since the human eye is currently the most accurate judge of where the object’s boundary lies, more advanced algorithms have been developed to reduce the amount of user time required without losing the accuracy.

**Morphing**

A simple speedup is to use morphing, or selecting the object in two end frames and interpolating between those selections [19, 22, 23, 20, 24]. This provides a good initial approximation in each frame, but using this technique often leaves the user still having to adjust the selection in every frame, which is less desirable when dealing with hundreds if not thousands of frames. The advantage, nevertheless, over complete manual selection is that, while the user is still in control of the selection pixels, the user has the benefit of automated interpolation to make an approximation so that each adjustment to the actual boundaries takes much less effort than starting from scratch.

Attempts have also been made to apply the edge-snapping and real-time selection properties of Intelligent Scissors to rotoscoping [15, 25, 26]. However, the original Intelligent Scissors implementation would need to be extended for rotoscoping purposes [26, 12], since the original implementation on an image-by-image basis required user input to determine the optimal selection for every boundary segment [25, 27]. Nevertheless, the “snapping” capabilities of the tool show promise for applications of the tool to video segmentation.

Energy minimization techniques used by snakes and active contour models have also been applied to video segmentation [28, 24, 29], requiring an initial selection by the user, followed by an automatic selection over subsequent frames [29].
selection in the subsequent frames does not have to be exact based on the previous frame, but can be an approximate polygonal initial contour, because when it “cools,” it will wiggle (like a snake) to converge to the optimal boundary. However, one drawback is the reliance on other tracking algorithms, such as point tracking [28], to get a good approximation from one frame to the next. Another drawback is the necessary user-defined number of iterations [29]. It also requires adjustment of snake parameters for different image types.

Keyframe-based Rotoscoping

Keyframe-based rotoscoping is a high-end automated rotoscoping tool at the forefront of video segmentation tools. It expands on the idea of adding intelligence to the interpolation of two selection boundaries (Fig. 2.6) [19]. The intelligence stems from a combination of minimizing an energy function based on the user-defined selection, utilizing advanced shape-blending techniques [21], and retaining similar boundary attributes from frame to frame.

In keyframe-based rotoscoping, the user selects an object in the first frame by placing seed points to form a piecewise cubic Bézier curve. The same object is selected
Figure 2.7: Results data from keyframe-based rotoscoping [19]. Ratio of percentage of user-edited points to total points in boundaries in all frames reflects success of technique. Every frame needed some editing, but a worst case of 11.4% is very good compared to previous rotoscoping techniques.

<table>
<thead>
<tr>
<th>Sequence</th>
<th>user-edited points</th>
<th>total points</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 5</td>
<td>483</td>
<td>4230</td>
<td>11.4%</td>
</tr>
<tr>
<td>Figure 6</td>
<td>338</td>
<td>5204</td>
<td>6.5%</td>
</tr>
<tr>
<td>Figure 7</td>
<td>494</td>
<td>8606</td>
<td>5.7%</td>
</tr>
</tbody>
</table>

Figure 2.7: Results data from keyframe-based rotoscoping [19]. Ratio of percentage of user-edited points to total points in boundaries in all frames reflects success of technique. Every frame needed some editing, but a worst case of 11.4% is very good compared to previous rotoscoping techniques.

in the last frame in like manner. The computer then selects the in-betweens. The energy function that the computer minimizes over all the frames contains an image term, which favors image contours similar to the selection, and a shape term, which penalizes fast-changing curves and shapes from frame to frame. After the control points are placed in positions where the “energy” is minimal, the user corrects any discrepancies in one or more of the in-betweens. The corrected frames are treated as keyframes for the next iteration in which the computer adjusts the inbetween selections based on the new energy function for each pair of keyframes [19].

The keyframe-based rotoscoping paper claims that it is difficult to determine the success of their results, and thus does not provide selection times for any of their test cases. However, they do record the amount of user interaction by computing the percentage of user-edited points from the total number of control points in the entire selection (Fig. 2.7). In the simplest test case (a 5 second sequence or approximately 150 frames), 5.7% of the 8606 total control points were user-edited. This means that the user had to manipulate an average of \( \sim 3 \) control points per frame. In a 3-second sequence (approximately 90 frames) with more complex images, 483 points were user-edited, for an average of 5.4 control points manually adjusted per frame.

The results of the keyframe-based rotoscoping are a strong improvement to previous rotoscoping options. It is a simple, intuitive, interactive process that takes away an average of 90% of the workload from the user, according to the paper’s data.
This algorithm shows that control points are useful to simplify the interface to the rotoscoping tool, minimizing the amount of work for the user. It builds intelligence into morphing through energy minimization. On top of that, only approximately 10 percent of those control points are touched by the user. This is a key concept that Keyframe-based Rotoscoping proves, the idea that the data gathered from a little bit of user input can go a long way. An algorithm should leverage minimal user input as much as possible. Keyframe-based rotoscoping is also good because it has decently short iteration times (few seconds to minutes) and allows quick user fixes between each iteration. In other words, “intelligent” does not necessarily mean “slow” or “long waits.” The small amount of preprocessing required is also advantageous.

Despite the advantages of keyframe-based rotoscoping, there is still much left to be desired. For one thing, even though the computer’s feedback of each iteration comes at reasonable rates, the user cannot do anything while the computer works and must wait for the computer to make each selection, due to the nature of energy minimization. This is as opposed to making a correction and immediately moving on to other corrections even while the first correction propagates. Also, 90% of the work is taken care of by the computer, yet the user still has to correct an average of several control points per frame. This means that the user still must make hundreds of corrections, and the task is still time consuming. It can also become difficult or complex as the number of keyframes build up. It is our goal to have an algorithm that does even more of the work while still letting the user ultimately guide the selection. Another problem is that the splines used by keyframe-based rotoscoping work best for objects with smooth edges. The energy minimization process is not a true snapping to the pixels (as with snakes or Intelligent Scissors) since it only deals with the control points, even though it claims to be similar to Intelligent Scissors. If there are complex, jagged boundaries, they are difficult to handle and require many control points. Another drawback is that, since objects in film generally do not move in a linear fashion, non-linearity is not inherent in morphing. That is, morphing is a linear process that is the basis of keyframe-based rotoscoping, so it has to “force” non-linearity with the energy minimization. We want to base our tool off of a more
inherently non-linear algorithm from the start, one of the reasons we choose Intelligent Scissors for our implementation. The difficulty in measure times and success of the Keyframe-based Rotoscoping tool can also be seen as something of a problem.

However, in the end, it is the intuition, speed, and ease of use that matters most, since the general feel and robustness of the tool is what makes those needing it for practical purposes (e.g. in film editing) want to use it. In this case, Keyframe-based rotoscoping sets a great precedent of such qualities, as one of the first intelligent rotoscoping tools that is a help, not a hindrance. So we will build on some of these good qualities to create an even more efficient and more robust tool.

2.3.2 Region-based Tracking

Another general method for object selection through time is based on “region” attributes of an object, or patterns, colors, and relations occurring inside the object’s boundaries.

Optical Flow

One method computes the optical flow occurring from one frame to the next to track the movement of the object [30, 11, 31]. The optical flow is a velocity vector assigned to each pixel (or possibly small region) to where it flows in the next frame. The velocity vectors point toward the pixels that are probabilistically most likely to be the same pixel in the current frame. This method is good on a macro level for calculating the general motion of an entire object. However, due to high sensitivity to noise, optical flow is not nearly precise enough on a pixel level to track exact boundary movement.

Mean-shift

Mean-shift algorithms are used for creating segmentation regions [11, 22]. The algorithm does this by clustering pixels within a sphere whose center point represents the mean of the points within the sphere. This is a useful method for categorizing regions of an image, even on a hierarchical level [11], but calculations are long and the
regions formed by mean-shift algorithms generally are not specific enough for defining exact boundaries without becoming too long-running.

Spatio-temporal Cubes

Spatio-temporal cubes have proven valuable for tracking objects through time, making use of region-based attributes [10, 11, 32, 9, 33, 34], as used in Video Cutout (Fig. 2.8). A spatio-temporal cube is a specific interface for viewing and interacting with 3-D data or with a video sequence, time being the third dimension to the two dimensions of the image. Any area inside the cube can be viewed by rotating the cube and slicing the cube with a plane parallel to the screen, zoomed to any depth into the cube. For video, this can be a great way to visualize a continuous block of an object through time, as a uniform smear through the cube. This is good for marking a selection through a region of an object through multiple frames all at once. Nevertheless, this is not the most natural or intuitive method to view video, since we are used to seeing life “one frame at a time”, as in a video. So for rotoscoping, it does not help for selecting out boundaries. We are better off letting the user watch the boundary selection through a loop, where the eye can quickly pick out bad frames due to temporal inconsistencies, as discussed earlier.

Min Graph-Cut

Min graph-cut is another powerful method useful for region-based tracking [16, 10, 35, 36]. In computer vision, graph-cut started out as a method to segment a single image. It intelligently segments part of an image based on a small training sample provided by the user. The user simply swipes a brush over a small area of the desired object, and min graph-cut determines the minimum cost boundary separating foreground (based on the users foreground sample) from the background (based on a separate, background sample also marked by the user).
Live Surface

Live Surface [37, 34] makes use of a hierarchical speed-up on 3d volume segmentation applied to video segmentation. Its results are developed and tested primarily for 3d volumes, but it also applies well to video. It has the best selection times of current video segmentation tools, as it runs selections in real-time as opposed to the several second wait of each iteration in tools such as Video Cutout.

Video Cutout

Video Cutout [10] is a recent, top-of-the-line tool, published in 2005 Siggraph, that combines spatio-temporal cubes and min graph-cut to create a powerful region-based video segmentation tool. In Video Cutout, a preprocessing pass breaks apart each image, grouping pixels into hierarchical regions of similar pixels based on a mean-shift clustering algorithm. As discussed earlier, mean-shift algorithms currently do not run at interactive times, and the preprocess can require 10 to 30 minutes for 100 to 200 frames of video, or about 6 to 10 seconds per frame. However, once the preprocessing is done, the user interacts with a spatio-temporal cube of the video frames at close to real-time rates. The user marks a sample of either foreground or background through a uniform region in time, and a min graph-cut segmentation is quickly computed over all the frames. The 100 to 200 frames are segmented in 10 to 15 seconds, for about a quarter of a second per frame. This is as good or better than most published segmentation tools in the forefront (Live Surface [37, 34] being a strong exception). Still, each segmentation is not perfect, and the user has to continue marking foreground and background regions, waiting 10 to 15 seconds for results until making any more needed fixes. For 100 to 200 frames, this interactive pass, or “artist time” as referred to in the paper, takes anywhere from a couple of minutes for very simple cases to 40 minutes or longer for more complex cases. Adding on the preprocessing, the total time can add up to over an hour or so of work. Once the selection is complete, a post-process of 30 minutes or more is run to refine the matte edge.

From examination, we see that the user has to do much less work than with
previous segmentation tools. This is due to some strong ingenuity and effort in creating an interface that minimizes user effort and maximizes efficiency of the user input. In other words, there is a high amount of control from little effort. Another possible strength is that there is no dependence on keyframes, due to the volume painting. It is good idea to keep rethinking the way we interface with time and video. In this case, Video Cutout does not look at the added dimension of time as a crutch but as an asset to add intelligence to the selection, leveraging off of properties of time that are unique to spatial properties. Another powerful feature is that it uses a hierarchical system to speed up the selection.

Again, despite the good there is still much to be desired. Video Cutout is relatively fast compared to other approaches, but we know that video segmentation tools can be even faster. Where Video Cutout falls short is that the user is waiting so much on the computer. Though the user is finally doing much less work, the computer still has a long preprocessing pass. There is also still a wait of several seconds between each user stroke (i.e. it is not real-time interactive). As seen in the high amount of artist time, the 10 to 15 seconds between each stroke adds up fast. Luckily, the time spent by the computer now is less than the time spent by users in more primitive tools, so things are moving in the right direction. We claim that the user should have to do even less work still. In a sense, the user is still having to fidget with every frame. Each paint stroke is quick, but sometimes it takes a lot of paint strokes to correct all the small mistakes. Edits may even occur on a per frame basis, since a fix in a single frame does not necessarily affect other frames. Another problem with Video Cutout is that, though inventive and ingenious, the spatio-temporal cube is not necessarily an intuitive way of looking at time and video, so there can be a lot of back and forth between interfaces to make sure the correct and consistent details are selected. As a last argument for need of a better tool, the properties of spatio-temporal cubes inherently demands that video stabilization, a possible additional preprocess, is necessary to have temporal adjacency of uniform regions of an object throughout the video range.

So the Video Cutout tool has many attributes to be desired, just like the
Keyframe-based Rotoscopy tool. Intelligent Rotoscopy, as described in this thesis, is a tool to take advantage of several of these strengths while putting aside some of the weaknesses, in order to increase speed and accuracy.

Among all the developed techniques, the most desired ones are those requiring the user to interact with the least possible frames, while exploiting the user’s input in those frames. To avoid the necessity of a long lunch break each time the automated selection tool is run, the selection in each frame must be fast, the ultimate goal being a real-time selection where the user is never waiting on the computer.

2.4 Fast, Moldable Tool

What is needed is a fast object-tracking tool that will select an object from a series of frames based on user input from only a single or a few frames. The automated selection should not select based on generic image data but should train on specific data from the user’s input, extracting as much training information as possible that results in a more accurate selection. From there, the user should be able to watch a speedy propagation of the initial selection, and nudge it now and again where the selection becomes unruly. This, in a sense, would allow the user to
“mold” the selection in certain frames and wait for the tool to propagate the fixes to neighboring frames. It should require a simple interface, minimal preprocessing, and minimal user input. It should be able to be applied to any sort of film with some temporal coherence, but not necessarily require a tripod to stabilize the footage. Artistic freedom and intuitive flexibility is necessary for the selection.

Three desirable elements of our Intelligent Rotoscopying tool include speed, accuracy, and reproducibility of the selection, while minimizing user input. The speed goal is for the processing to always be ahead of the user. The accuracy goal is to avoid the user needing to touch every frame (or even every other). The reproducibility goal is for multiple users to be able to meet the speed and accuracy goals multiple times and on multiple image sequences with little or no variation in the resulting boundaries.
Chapter 3

Intelligent Scissors for Video

This chapter presents the key methods for Intelligent Rotoscoping. In Section 3.1 we review the contributions and desirable characteristics of an Intelligent Rotoscoping algorithm. Section 3.2 outlines the general Intelligent Rotoscoping algorithm. Section 3.3 covers the extensions to the Intelligent Scissors data structures needed for rotoscoping. Section 3.4 introduces an automatic method devised to propagate a selection boundary from frame to frame, where we apply a robust, real-time snapping technique to snap one frame’s selection boundary to the object in the next frame. We call this process we term “leapfrogging.” In Section 3.5 we develop “Histogram Snapping” to allow cost maps to be trained as accurately as possible to represent the object’s changing boundaries with costs much lower than surrounding extraneous edges. Knowing that our current methods will not select every frame perfectly, Section 3.6 discusses how we make the Intelligent Scissors boundaries editable for corrections. Then Section 3.7 presents an interface for the user to watch the boundary propagate from frame to frame and intercept any incorrect selections. User corrections are propagated through adjacent frames, so that a single edit is leveraged to correct multiple adjacent frames.

3.1 Considerations for Extending Intelligent Scissors to Video

The implementation of Intelligent Rotoscoping works by harnessing various features of Intelligent Scissors for video. Intelligent Scissors has valuable properties that empower Intelligent Rotoscoping to consistently make robust and quick selections through the dimension of time. It allows a user to intuitively and interactively guide a
selection to desired boundaries in an image, while the boundary extraction algorithm does most of the work, snapping entire wire segments to the exact pixel-level boundary. There is no computer lag time between user adjustments, so corrections happen in real-time, meaning the automatic selection is fast enough to exceed user feedback time. In other words, the annoyance of waiting is eliminated, since the user is the time-bottleneck, not the computer. This is also possible for Intelligent Rotoscopy by capitalizing on the following elements of Intelligent Scissors: precomputed/cached cost maps, live-wire snapping, an interactive user interface, and training through simple histogram remapping techniques.

While the time required by Intelligent Scissors is fast and reasonable, what is reasonable for a single frame compounds into too much work when having to place each and every seed over numerous frames. In our algorithm, we consider how to take a single image’s selection and extrapolate the information so that the user does not have to place every seed and even avoid touching some frames at all. The primary intelligence behind Intelligent Scissors is the snapping of boundaries between the seeds, not in the placement of the seeds themselves. We can take advantage of that by placing as few seeds as possible and letting the live wires in between them do the snapping. However, if there is too much distance between seed points, Intelligent Scissors tends to cut through the desired object (cut corners), cutting needed pieces of the object out of the selection (Fig. 3.1). This natural tendancy to cut corners (cut off object appendages/branches/skinny portions/elongations/etc.) is either the fault of noise in the cost map or of “inside edges”, the boundaries and detail passing through the inside of the target object (Fig. 3.2a). This is generally not a huge problem in a single frame, since the user can strategically place seed points to prevent corner cutting. This is more of a hinderance when Intelligent Rotoscopy automatically lays down seed points, since identifying exact locations for seed points that absolutely prevent corner cutting is a tough problem. It is best that fewer seeds are placed and that the live-wire is able to follow around elongations and corners between two far-apart seeds, than worrying about whether or not seed points were place correctly to avoid missing the correct edge. The better method is to train the costs of the object’s
boundary to have a much more preferrable cost than any inside edges, consequently giving the automatic seed placement some flexibility in where the seeds can land. For example, in Figure 3.2, rather than worry that a seed lands exactly on the extrusion of the arm to avoid cutting off the arm (Fig. 3.2b), our algorithm just needs to hit the edge in the vicinity of the arm, and the arm is selected (Fig. 3.2c). If the cost maps are assigning low costs to incorrect edges, the user cannot completely obtain the desired edge since it snaps to improper locations. Even worse, when the algorithm trains on that bad data, there is no way it will automatically capture edges of the correct object during boundary propagation. Our algorithm is only as good as the fidelity and consistency of the cost maps. In other words, our methods must train on the user input correctly so as not to lose pieces of the desired object as much as possible. The idea is to squeeze as much as possible out of the live-wire boundary snapping (to account for seed points missing the boundary) to guarantee robustness in the intelligence of automated seed placement.

### 3.2 Overview of the Intelligent Rotoscoping Algorithm

The algorithm for Intelligent Rotoscoping, with the goal of interactive speeds and minimized user guidance, is as follows (see Fig. 3.3):

1. The user selects a boundary (closed loop of pixels along an edge) around the desired object in the first frame of a video sequence only (Fig. 3.3.1).

2. The user-defined path’s pixel data is stored as training information for future frames’ cost maps.

3. The rotoscoping tool “stamps” the input boundary from the current frame onto the temporally adjacent frame, treating it as the approximate cursor path around the object which has moved to a slightly new position in the new frame.

4. A process called “leapfrogging” (§3.4) samples points along the approximate cursor path and uses them as seed points to “snap” an Intelligent Scissors boundary to the desired object, based on the training data. This is called the automated selection of an object in a frame (Fig. 3.3.2).
Figure 3.1: “Inside edges,” details or edges passing through the target object, are sometimes problematic since they may provide a shorter, lower cost path than the desired object edge (yellow), causing boundary to snap to them instead of the desired edge. Example inside edges (pink) are a) the edge of Lucy’s sleeve, b) her foot, or c) even the shadow across her neck. It is possible that an inside edge is as strong or stronger than the object’s edges unless the cost function is trained otherwise.
Figure 3.2:  a) Selection of Lucy has seed points (blue) and snapped selection boundaries (yellow). Selection boundary incorrectly cuts off Lucy’s arm due to shorter path of inside edge. b) Two seed points on either side of arm are not enough, so c) additional seed point corrects it. d) More intelligently computed cost map better distinguishes inside and outer edges so that additional seed points are not necessary. For automated selection, fewer seed points means less room for error.
5. The Intelligent Rotoscoping tool continues to propagate each frame’s selection to its neighbor (Fig. 3.3.2-4).

6. As propagation continues, the user watches a continuous loop of the current selection and intercepts frames where the selection goes astray (Fig. 3.3.3a). Any manual corrections are quick, high-level mouse gestures, which the software uses to adjust the boundary and fix the error.

7. Any boundary segments manually placed or corrected by the user are used to augment the training set.

8. The manual corrections are propagated forward through already selected frames, then the entire selection continues to propagate through non-selected frames in temporally adjacent order (Fig. 3.3.3a-4a).

9. The propagation stops when the last frame is segmented, although the user can still go back and make any necessary corrections.

Again, Intelligent Rotoscoping takes advantage of Intelligent Scissors and extends its best properties, the automated snapping to boundaries and the user interactivity. The user is in the loop to guide the process while the computer does the “heavy lifting” of the selection work. Exploiting, while minimizing user interaction is an important aspect of our work.

3.3 Extensions to Intelligent Scissors’ Data Structure

3.3.1 Cost Map Calculation

As with the original Intelligent Scissors, every image has a corresponding cost map, $C_i$, the same size as the image, with a cost assigned for each pixel. We apply no shortcuts to the cost map, such as calculating costs only along watershed boundaries, even if they would speed up the algorithm, since the cost information is used not only to find correct boundaries but is also preserved for subsequent training (see §3.5). We would not want to introduce an additional layer of possibility of losing the object’s
Figure 3.3: Intelligent Rotoscoping summary. 1) Intelligent Scissors selection of object in first frame. 2-4) Selection propagated through neighboring frames. There is a midstep between each frame where the first frame’s selection is “stamped” into the second frame, then snapped to the second frame’s boundary. Due to extreme change of target’s hand, propagation of selection misses hand in frame 3. 3a) User catches mistake, “molding” boundary (white segments) to correct selection. 3a-4a) Correction is propagated back through frames where automated selection missed.

boundary from frame to frame. We also only use a 3x3 filter to do the edge detection, instead of computing with multiple filters, since the preprocessing time would add up too high given seconds of multiple frames.

To compute the image’s respective cost map, the original Intelligent Scissors algorithm adds the weighted gradient magnitude image to the weighted zero crossing image (Eq. 3.1). The Laplacian helps to localize boundaries, but also introduces a lot of false low-cost paths into the selection image (Fig. 2.4). This is fine for single-image selection because the user can move the live-wire around until the optimal boundary is selected. However, since video segmentation is automated, we cannot afford to probe many different possible paths interactively, so we minimize costs of noise and false paths as much as possible. Yet, the Laplacian is still helpful, so we will now discuss how to minimize the noise and erroneous low-cost edges introduced by the Laplacian while still making use of its good qualities.

Our approach to avoiding many extraneous edges cutting through the object
is to mask the gradient magnitude images, $I_G$, with their corresponding zero crossing images, $I_Z$, instead of adding them. Since pixels in the zero crossing are either a value of zero or one, it can be used as a binary mask on $I_G$ by multiplying the two images (Eq. 3.2). Each resulting pixel either retains the value of $I_G$ (where $f_Z(p) = 1$) or is masked out to zero ((where $f_Z(p) = 0$)) For each pixel, each color channel is masked separately, then the results are added together to get a single cost value for that pixel. The resulting cost image, $\zeta_I$, is a grayscale image.

When $I_G$ and $I_Z$ are multiplied, the highest valued pixels in the resulting image indicate some object’s edge in the image (we refer to these as “edge pixels”), and low or zero valued pixels indicate uniform-colored image regions (the opposite will be true once it is time to invert the cost map). Pixels values in the gradient image will either be masked out by the zero crossing (multiplied by zero) or else will be kept around by the zero crossing (multiplied by one). If the gradient value is not masked out but is low, the pixel retains a low value. Therefore, only pixels where both the zero crossing and the gradient magnitude images have high values will retain a high value. These are the edge pixels. After the masking operation, the resulting cost image is inverted, resulting in low costs where there are edge pixels. For any pixels with a cost of zero, a small amount, $\epsilon$, is added back in to prevent loops in the selection boundary. The cost map is now correct, since the Intelligent Scissors boundary should follow the path of the lowest cost.

This masking process does tend to leave some small high-cost “holes” where there are object edges, due to the binary (0 or 1) nature of the Laplacian (Figures 3.4b, 3.5c). To handle the holes, the inverted, grayscale gradient magnitude image is added back into the cost map, but only for pixels with costs close to the maximum cost in the cost map. This raises non-edge costs even higher but does not boost costs much where there are edges, though those costs are still not as low as the high-certainty edges (Eq. 3.2). Since the cost of non-edge pixels is much higher than edge pixels in the cost map, manual selections and corrections do not cut through objects as much and instead follow around corners as they should (Fig. 3.4).

Adding the gradient back in helps smooth out rough parts in the selection,
but it is not given too much emphasis or else it introduces extraneous edges back in too strongly. A more important help to avoid corner cutting is to scale up the pixels costs that are close to the maximum cost by some amount, \( \omega \) (Eq. 3.2.2). We use a value of 5 for \( \omega \). This value does not have to be specific, just a value high enough to discourage the selection from cutting through those high cost areas. This is good enough for all cases.

Cost map \( \zeta_I \) is the result of these calculations over each pixel in the image (Fig. 3.5). For a given pixel, \( p \), and an adjacent pixel, \( q \):

**Intelligent Scissors:**

\[
\zeta_I(p, q) = \omega_Z \cdot f_Z(p) + \omega_G \cdot f_G(p) + \omega_D \cdot f_D(p, q)
\]  

**(3.1)**

**Intelligent Rotoscoping:**

1. \( \zeta_I(p, q) = f_{Zr}(p) \cdot \omega_G \cdot f_{Gr}(p) + f_{Zg}(p) \cdot \omega_G \cdot f_{Gg}(p) + f_{Zb}(p) \cdot \omega_G \cdot f_{Gb}(p) + \epsilon \) where \( f_Z(p) \in \{0, 1\} \)
2. if \( \zeta_I(p, q) \approx \max(\zeta_I(p, p)) \Rightarrow \zeta_I(p, q) = \zeta_I(p, q) \cdot \omega + f_G(p) \)

**(3.2)**

where \( f_Z \) represents the Laplacian image, \( f_G \) the Gaussian, and \( f_D \) the gradient direction, for any given pixel \( p \) and neighbor (adjacent pixel) \( q \), as shown in the previous chapter (2.2). Note that we do not use gradient direction in our implementation.

Identifying good edges in the cost map is more difficult than simply combining edge-filtered images, though that is a good start. This is because not every edge in an image is a desirable edge, and what is considered a good edge to one user may be undesirable to another user wanting to select something completely different. The cost map calculation therefore needs to be able to expand or train on user-provided data to update cost maps, as addressed in Intelligent Scissors [5].

Thus, we make two separate cost maps for each image, one for user selection and one for automated selection. User or interactive selection is where the user manually places seed points. The cost map \( \zeta_I \) is used for computing boundaries for interactive selection. Automated selection is where the user selection is automatically propagated from frame to frame. A cost map based on training data, \( \zeta_T \) is used for the
Figure 3.4: Simplified representation of a cost map, based on top image. a) Pixel costs represented as both gray scale and numerical values. Low cost pixels in cost map match edge in image. Yellow path follows lowest cost path between seed points (blue). Current lowest cost path adds to 7. b) If a “hole” exists in cost map’s edge (pixel with cost of 7), lowest cost (yellow) path now cuts through a “non-edge” pixel. Lowest cost = 11, true edge path cost = 13. c) Multiplying gradient image against high-cost pixels raises costs of non-edge pixels the most while retaining sharp, crisp edges from the Laplacian in the lowest costs. The lowest cost path once again follows the true edge.
Figure 3.5: Example costmap calculation of a) Lucy image. b) Gradient image, $I_G$. c) Gradient masked by Laplacian, for sharper edges. There are small holes introduced in the edges by the masking (marked by green circles). d) Cost map inverted (so lowest costs are along edges), then inverted gradient added to pixels with cost closest to highest cost, lightly fills in the holes.
automated selection (§3.5). $\zeta_I$ actually does come into play in creating the training cost map, so it is not completely ignored for the automated selection, but it plays a minor role (§3.5.7).

Though the interactive cost maps, $\zeta_I$, we have discussed are not the core of our Intelligent Rotoscopy algorithm (they are extensions of the already developed Intelligent Scissors), we have focused on the cost map for a couple of reasons. First, it ties into concepts helpful for understanding the creation of cost maps through training discussed later (§3.5). Second, it is important for the user to get long, accurate boundary segments for quicker editing and better training information.

### 3.3.2 Stored Data Structures

All the images and their corresponding cost maps need to be stored in memory for the duration of the program’s run time. Following is a discussion of what we store and the space required.

The original image for each frame must be stored and displayed for the user interface. The cost map is stored as a 2D array, with dimensions of width and height. We also store, in arrays, pixel properties useful in the training algorithm, such as color, gradient magnitude, and gradient direction, in arrays for quick accessibility to the data as training happens in real-time.

During automatic selection of the Intelligent Rotoscopying process, several seeds must be placed and the lowest cost path for each adjoining boundary segment must be calculated very quickly, totalling within a second per frame, to keep it interactive. For the graph search expansion, we use a linked list data structure for the stack containing active pixels on the wavefront of the graph expansion. To make pushing and popping operations on the stack as fast as possible, we omit any method calls and inline the “push” and “pop” linked list operations (this is the main reason we use a linked list instead of a vector implementation). A node in the linked list contains its pixel’s (x,y) position, the direction it is pointing (N, NE, E, SE, S, SW, W, or NW), and a pointer to each of its neighbors in the linked list. To avoid the overhead of creating and deleting unneeded nodes, we create and store a node for every pixel in
the image as part of the preprocessing pass. The nodes are stored in a 2D array the size of the image, so pushing and popping is a simple operation of adding or removing a pointer to the proper node in that array.

### 3.3.3 User Interface

The user interface provides for the manual selection of an initial boundary and for adjustments of errors in the automated selection (Fig. 3.6). It consists of a main window displaying the current frame (Fig. 3.6a). The user can choose which frame it displays by scrubbing through a provided timing bar or by using the arrow keys or interface buttons to step one frame backward (left) or forward (right) in time (Fig. 3.6d). When any frame is displayed that has no selection, the user can manually provide a selection by placing seed points with the mouse. When any frame is displayed that has an Intelligent Scissors selection, it will be displayed with its boundary segments and seed points. If the user or the computer is currently selecting on the viewed frame, the selection will be displayed immediately for the viewer to see; however, if the user is viewing another frame, the user will not see the active selection until s/he steps to the frame of the selection front. The automated selection is started with a simple click of the “Select Inbetweens” button (Fig. 3.6e).

While stepping through frames examining the selections, if the user sees a problem, a click in the problematic frame will halt the automated selection so that the user can make a correction. We first allowed the automated selection to continue through unselected frames as the user makes selections, but this method does not work as well since the problems often continue to propagate and become more difficult to fix. Instead, the user makes the correction, the correction propagates through the already selected frames, then the selection continues on to unselected frames. This works well since the computer is waiting on the user and not vice-versa. The user continues to check for mistakes until all needed corrections have been made.
Figure 3.6: Demonstration of the user interface. a) Main editing window, showing current frame. This window is where user makes selections, watches selection propagations, makes fixes, etc. b) Currently viewed frame number. c) User interfaces with mouse. d) Multiple options for user to switch frames (arrow buttons, slider bars, as well as right and left keys on keyboard). e) Button to start automated selection. The selection uses current frame’s selection boundary and starts in next frame. f) Display only selected foreground pixels (background assigned a constant color, such as green). g) Select which pixel attributes to train on. h) Select dimension of the n-degree training histogram.
3.4 Leapfrogging

The power of intelligent scissors lies in its ability to snap a boundary to an object edge going between two “seed points.” In video segmentation, this snapping is extremely important. To select an object in a frame, seed points taken from the neighboring frame’s selection boundary are laid on the current frame as an approximate selection. An Intelligent Scissors segment is calculated between each seedpoint, and those segments naturally snap to edges.

An initial thought we had was that the snapping capabilities of Intelligent Scissors alone should be enough to snap the approximate boundary to an object’s boundary. However, since the object will have moved, the seedpoints will not land exactly on the object’s edge (Fig. 3.7a), and the portions of each segment closest to the seedpoints will fall off of the desired object’s edge. Even worse, two or more neighboring seeds with enough error will force boundaries to snap to an erroneous path between them instead of to the real path (Fig. 3.7b-d). When this new, incorrect selection is used to approximate the next frame, it will be even further off than the approximate selection placed in the current frame. That segment will continue to propagate to erroneous edges and, pull even more seed points and boundary segments into error with them. With our tests, especially in more complex images, the object was always quickly lost and not coming back, so we needed a clean way to snap the seeds as well as the boundary segments to the exact edge.

Thus we created a process, deemed *leapfrogging*, that will make sure automatically deposited seed points will snap to the right edge. It does so by determining the pixels most likely to lie on the object edge, as described here.

Leapfrogging Procedure:

1. Seed points $s_1..s_x$ are taken at equal pixel intervals along the boundary from the previous frame, seed points (Fig. 3.8a).

2. Temporary boundary segment $b_{1,2}$ (pink) calculated from $s_1$ to $s_2$ (Fig. 3.8b).

3. A second temporary boundary segment, $b_{1,3}$ (pink), is also calculated from $s_1$, but goes to $s_3$ (Fig. 3.8c).
Figure 3.7: Propagating the selection without leapfrogging can quickly lose the desired boundary. a) Seeds from previous frame’s star boundary miss actual star. Without a way to snap seeds to actual edge, the selection boundary detaches from the boundary (even when implementing some training). b-d) In more complex example, based on real data, selection from frame (b) dropped onto frame following leapfrogging (c). Even adjacent frames can have bad misses. In (d), more distant frame, selection quickly loses hair and hem of skirt due to multiple seeds falling on top of another edge inside the object. In general, the problem is that seeds inconsistently falling off the boundary from frame to frame.
4. The intersection of $b_{1:2}$ and $b_{1:3}$ captures the overlapping segment that actually “snapped” to the object boundary, since both segments “agree” that that portion is most likely to be on the object’s edge. It is the consensus between two overlapping segments, catapulting the boundary forward that prompted the name “leapfrogging.” Finalized segment $B_1$ (yellow) is created from that intersection (Fig. 3.8d).

5. The final point in $B_1$ is set as the first finalized seedpoint, $S_1$, in the new boundary, $B$ (Fig. 3.8e). Note that the first finalized seed falls exactly on the boundary.

6. A segment already exists from $S_1$ to $s_3$ (the back end of $b_{1:3}$), but a new segment from $S_1$ to $s_4$ is created, the intersection of which completes the next boundary segment $B_2$. The final point in $B_2$ becomes seed point $S_2$ (Fig. 3.8f-g).

7. A new segment from $s_2$ to $s_4$ is created. The intersection is taken as before (Fig. 3.8f).

8. The process is repeated through all the seed points, up to $s_x$ and back to $s_1$ for closure on the boundary. We wrap around to $s_0$ and $s_1$ due to the necessity of overlapping segments to calculate the final closing segment.

Since all seed points for an image are laid down before any boundary snapping occurs, the graph search for creating the cost between two seeds need only expand from one seed until reaching the neighboring seed point. Therefore, unlike in Intelligent Scissors, we do not expand the graph search over the entire image. We expand it from the source seed point until the destination seed gets picked up in the active wavefront. We stop the expansion since we know the lowest cost path between the two seed points is contained within the currently searched pixel. From there we go on to start the graph expansion of the next seed point. Since the entire image does not need to be searched for each boundary segment, we keep the speed of leapfrogging within interactive times.
Figure 3.8: a) Approximate placement of seed points. b) Temp boundary segment $b_{1:2}$ formed by Intelligent Scissors boundary from temp seed $s_1$ to temp seed $s_2$. c) Segment $b_{1:3}$ formed from $s_1$ to $s_3$, overlapping $b_{1:2}$. d) Permanent segment $B_1$ is intersection of $b_{1:2}$ and $b_{1:3}$. e) $S_1$ is last point in $B_1$ and is a permanent seed point. Each new permanent seed point becomes starting point for leapfrogging next segment. $b_{1:3}$ is now segment from $S_1$ to $s_3$. f) $b_{1:4}$ is segment from $S_1$ to $s_4$. g) $B_2$ is intersection of $b_{1:3}$ and $b_{1:4}$. The process will continue until all temp seed points exhausted. For example, $B_3$ will be the intersection of $b_{2:4}$ and $b_{2:5}$.
Distance Between Leapfrog Seeds

To reiterate, seed points used for leapfrogging are obtained by placing an approximate boundary (from the previous frame) in the current frame, then choosing select pixels from that boundary to be seeds. Our initial system for selecting those seeds was to step along the boundary at equal increments. The increment size was determined by trial and error.

We learned that if the increment size is too low and the space between seeds too small, the leapfrogging is less likely to snap to the desired edge. This is because it becomes easier for the boundary segment to jump from one seed to the next than to snap all the way to the boundary and come back up. On the other hand, if the increment size is too large, then the leapfrogging tends to cut off skinnier appendages of the object. This is because the path from \( s_1 \) to \( s_2 \) and from \( s_1 \) to \( s_3 \) may not intersect at all, in which case segment \( s_1s_2 \) has to be ignored and \( s_1s_3 \) must be intersected with \( s_1s_4 \). The ignored segment from \( s_1 \) to \( s_2 \) may have been necessary if it was the part wrapping around the appendage (Fig. 3.9).

We did find something of a sweet spot for the increment amount, yet it still is not robust for all cases, such as smaller objects or areas of high curvature in the object’s boundary. For this reason, we use a quick yet more adaptable method for determining the position of the seed points selected from the approximate boundary. We apply the polygon approximation algorithm to the estimated boundary, which recursively chooses pixels along the boundary that are farthest from the polygon formed by lines adjoining already chosen boundary points (Fig. 3.10). This is a similar idea to selecting seed points based on the curvature of the boundary, since it places more seeds where the curvature is high without the expense of actually calculating the curvature. This means that more seeds are placed where there is more detail in the boundary (such as a head or hand) and less elsewhere. This is exactly what we need to prevent corner cutting without placing too many seeds over the entire boundary (Fig. 3.11).
Figure 3.9: Description of why leapfrogging has a tendency to cut corners.  
a) Seeds are evenly placed around Cinderella’s head and the selection process started. Placement of seed points does not reflect the geometry, meaning there are not more seeds where there is more detail in boundary.  
b) Lack of extra needed seeds around the head means leapfrogging segment $s_1s_2$ hardly intersects $s_1s_3$.  
c) The next leapfrogging segments intersect, but across the necklace.  
d-e) Selection continues but has missed the head.
Figure 3.10: Steps of polygonal approximation of the boundary around Cinderella’s head. a) Vertices placed at extremal points of the boundary. b) Farthest boundary point on either side of the intersection line calculated. c) Polygonal edges join current polygon vertices. Farthest boundary point from each polygonal edge found. d-e) New polygon formed and process repeats recursively until all polygonal edges are within $\epsilon$ of the boundary. The polygon approximates the boundary. f) Polygon vertices used as seed points for leapfrogging. Note the seeds generally end up at points of higher curvature and where there is more detail in the boundary.
Figure 3.11: Description of how leapfrogging works when enough seeds are placed around areas of detail through polygonal approximation of boundary. Polygonal approximation puts enough seeds around the head that it selects the correct boundary instead of cutting across inside edge.
Silhouettes

The leapfrogging technique proves to be especially robust in video clips where the selection object is a silhouette (all one color), even when the background is fairly detailed, as in Figure 3.12. This is because there are no inside edges cutting through the silhouetted object (Fig. 3.1). Since it has a uniform value throughout, the whole object is filled with high costs. An example where an inside edge would cause problems is the line of a t-shirt sleeve cutting across the arm of a person, assuming we want to select the entire arm along with the rest of the body. If seed points are at the armpit and shoulder of the person, the selection boundary between them would most likely cut across the t-shirt sleeve instead of capturing the entire arm, which is a longer, higher cost path (see Fig. 3.2a). However, if the object is represented as a silhouette, the cost is much cheaper around the edge of the entire arm than to cut through where the sleeve edge would have been (see Fig. 3.2c).

Of course, always having a silhouette for an object is unrealistic. However, the cost maps can be trained such that all the inside edges and areas of the object receive high costs while its outer edges receive the lowest costs, thus making the object a silhouette in the cost map. We achieve this to a degree by training the cost map based on the pixel values underneath the user-selected boundary. Even if this does not create a perfect silhouette in the cost map, meaning that there are still lower cost “inside edges” passing through the desire object, the boundary edge will be trained to have still much lower costs. We dub this training of the inside edges to have higher costs, the “silhouette effect.”

3.5 Training

Training is a key element to make Intelligent Rotoscopying work robustly. In regular Intelligent Scissors, each time the user places a seed point and a new boundary segment is formed, the cost remapping histogram is updated on the fly using the pixel information from the previous segment. That way, upcoming segments snap to similar-gradient edges instead of only the strongest gradients. We extended a similar training idea to Intelligent Rotoscopying, where the costs under the pixels of the user
Figure 3.12: We show the cost map of a silhouette in an image, demonstrating that it would be ideal for all cost maps to maximize costs inside edges. a) Desired object is a silhouette (no strong foreground edges). b) The cost map of a silhouette, where lowest costs follow the silhouette’s edge, makes for an easier, cleaner selection than c) if there are inside edges in the image, where the selection can take short cuts. Ideally, training would omit the foreground edges (and, even better, background edges) in the cost map, as if it were a silhouette like in b). The silhouette and edge have been emphasized in this diagram for demonstration purposes.
selected boundary are used to remap the costs of pixels in subsequent frames. Training data is only extracted from interactively placed selection boundaries, not from automatically propagated selections, since those could possibly be in error, introducing bad training. We call the boundary used for training the “Training Boundary” (TB). A histogram of pixel values in TB is used to remap each pixel’s cost in the next frame. Pixels which have very similar or the same attributes as pixels in TB are mapped to the lower costs and pixels that are not so similar are mapped to higher costs.

*If the training works correctly, “inside edges” receive higher costs than boundary edges, as well as edges of undesired objects in the background which also receive higher costs, thus creating something close to a silhouette of the object in the cost map!* The leapfrogging technique thus is more likely to snap to the object boundary than to foreground or background edges.

To get a successful silhouette effect in Intelligent Rotoscoping cost maps, more than one feature for TB pixels needs to be used, not just gradient information. Additional features include pixel color, neighboring pixel colors, neighboring pixel colors of only pixels inside the object, and the \((x,y)\) coordinate position. Another key to training is that any selections or corrections performed by the user are considered valid training information and be used to update the training set.

An important note is that when the user edits/manipulates propagated boundaries, the cost map for interactive selection, \(\zeta_I\), is used to calculate the user’s boundary, not the trained cost map \(\zeta_T\) (§2.2). The training data is only used when making automated selections. This way, the user’s input is not dependant on previous training information, which could be irrelevant if the object boundary has changed significantly over time. It is possible that the automated selection the user is correcting failed because of bad training data, so the user needs to make the corrections based on original gradient information.

When the user does make a correction, the pixel data of the corrected boundary pixels are added to the training set, since anything finalized by the user can be trusted as accurate training data. From there, automatic propagations are less likely to
make the same mistake again. Adding pixel information from automatically selected boundaries to the training set could have been done, but this ends up being more problematic than helpful. This is because some boundary segments created in the automatic propagation do not always capture the correct edge or else cut through the object. The algorithm has no way of knowing for sure which boundary segments are correct and which are not (or else it would not make the mistakes in the first place). So adding pixel data from every boundary introduces bad data into the training set.

This section is a crucial idea for this thesis. The idea is that since we have so much information to train on, we should not be using a generic cost map every frame. Our method does not train on every possible feature, but it demonstrates that the selection process can keep improving by intelligently making use of available features, without a great time expense.

3.5.1 Data Structure Setup for Training

A multi-dimensional histogram composed of the pixels under the user selected boundary is created for remapping the trained costs. We call this the remapping histogram, \( H_r \). Each dimension represents a single-valued feature of the pixels (i.e. characteristic trait or attribute). For example, the red channel is a pixel feature. Possible traits are color channels, color channels of neighboring pixel, gradient magnitude and direction, and so forth. Because a pixel can be defined by multiple features, the histogram has multiple dimensions. Each feature in the histogram has a defined range, from zero to some positive integer, \( a_i \). Therefore,

\[
H_r = H[0..a_1][0..a_2]...[0..a_n]
\] (3.3)

for a histogram with \( n \) dimensions, where \( a_i \) is the range of dimension \( i \). For example, the red channel goes from 0 to 255, so \( a = 255 \) for the red dimension. The ranges do not need to be the same for each dimension.

It would seem most efficient to only train on as few features as possible, which may be true timewise. However, in our implementation process, it became apparent that the addition of several useful features improved the results, so we implemented
the training to be able to handle a variable number of features, despite the added complexity.

Let the set of chosen attributes and their corresponding values for a given pixel, be called the “PixelID” for that pixel. We implement the PixelID as an array of values, \( \text{PixelID}[0\ldots n] \), where \( n \) is the number of features.

In our implementation, a PixelID is a combination of up to 12 attributes: the three channels of the pixel’s color, the three channels of the gradient, the three channels of the average neighboring (adjacent) pixel colors, and the colors of neighboring pixels inside the boundary (Fig. 3.13). We can use any combination of the above attributes, though we default to gradient magnitude, color, neighbor color, and position.

### 3.5.2 PixelID Creation and Feature Calculation

A PixelID has two uses, one for filling in the training histogram and the other for looking up a pixel’s cost in the histogram.

1. To update the training histogram, a PixelID is formed for each pixel in the training set (obtained from the user selected boundary). Each PixelID increments the cost of the slot it indexes to.

2. To create the cost map for a given image, a PixelID is created for each pixel in the image. The PixelID does a lookup to its corresponding slot in the histogram and uses that slot’s value as the pixel’s cost.

To calculate the PixelIDs, we do the following for a given pixel:

1. Create a PixelID object with desired pixel attributes.

2. For each attribute in the PixelID definition, calculate that feature for the current pixel and insert it in the PixelID.

Before discussing in detail the algorithm for using and training from the cost-remapping histogram, we explain in detail the training features we use to create the PixelIDs and define the histogram. A PixelID does not need all those attributes,
nor is it bound to only those; however, the attributes used for PixelIDs must remain constant through the entire selection process. Following are descriptions of each of the training features in detail. Each feature is a dimension in the histogram as well as a slot in the PixelID feature array.

**Train on Edge Color**

This is a simple one. The red, green, and blue (RGB) values for each pixel in the path are a useful feature. The hue, saturation, and brightness (HSB) values can also be used in place of RGB. Figure 3.14a shows an example pixel in the training boundary, TB, used to fill in the color attributes of that pixel’s PixelID. It can be represented either as three features in the PixelID, one for each color channel, or as a single grayscale feature \(0.3 \times Red + 0.59 \times Green + 0.11 \times Blue\). In our implementation, if the image is a color image, we represent the color as three separate channels (thus three separate features), since the more beneficial information we have for distinguishing between boundary edges and unwanted edges, the better (Fig. 3.14a).

**Train on Gradient Magnitude**

Values from the gradient image are used as one of the training features in the PixelID. Each pixel in the image has a corresponding gradient magnitude value with an equal number of color channels, as shown in Figure 3.14b. The gradient is beneficial to the training set because it is a robust way of detecting similar edges based on the strength of the edge. Even when exact matching gradients are not found from one frame to the next, similar gradients can be found. As with color, the gradient has a range of values from 0 to 255. The number of color channels in the gradient map matches the number of channels in the image.

Though the Laplacian image is associated with the gradient magnitude for creating the initial cost maps, the information from the Laplacian is not used at all in the training, due to its binary nature and tendency toward noise.
Figure 3.13:  a) A PixelID keeps track of the \( n \) feature values for a given pixel. It also can store the matching cost that is looked up in the histogram.  b) For a PixelID with six features, there is a dimension for each feature. Each feature (and thus each dimension) has a set range.  c) The PixelID for some pixel contains a single value for each feature. This example’s combination of values maps to a single cost in the histogram. Cost is the number of matching PixelIDs in the training set. So our example PixelID \( \{1, 8, 3, 4, 13, 9\} \) maps to cost 18, meaning there are 18 other matching PixelIDs in the training set.
Figure 3.14: Given the red and green image above, with selected green region (close up shows boundary pixels in yellow), the following are features we use in the PixelID data structure. 

- **a)** For a given boundary pixel (bordered black), (red, green, blue) (RGB) values are each stored as attribute. 
- **b)** For same boundary pixel, RGB values of the gradient magnitude image are each stored as attribute. 
- **c)** For average neighbor attribute, adjacent pixels in direction perpendicular to selection boundary are used, for a distance of x pixels in each direction (in this case, x = 2). For each RGB channel, the values from all neighbor pixels are averaged. 
- **d)** For inner neighbor attribute, we step inside perpendicular to selection until hitting a uniform region, where the edge gradient stops.
Train on Neighboring Pixels

For each pixel in the training boundary, neighboring pixels perpendicular to the boundary and within a small distance to the pixel are averaged together, as in Figure 3.14c. In our example, the pixel values of the two neighbors on either side are (183, 63, 15), (183, 63, 15), (14, 203, 67), and (14, 203, 67). So the attributes are the average red channel, 98, average green, 134, and average blue, 40. Note this is different from the color of the pixel on the boundary.

The direction of the neighboring pixels can be acquired from either the geometry or the gradient direction (§2.2). If we know the object’s boundary, the direction perpendicular to the edge can be acquired by computing the line between neighboring points along the boundary. For the \(n^{th}\) pixel in the boundary, the approximate direction angle, \(\theta\), is computed as follows:

\[
\theta = \tan^{-1}\left(\frac{y_{n+1} - y_{n-1}}{x_{n+1} - x_{n-1}}\right)
\]  

(3.4)

In the frames where automated selection happens, no boundary exists from which to determine the direction based on geometry, so the gradient direction is used. Since the gradient direction can be noisy, we take the average of the gradient direction in all three color channels.

To test the cost of a given pixel, look up the gradient direction for that pixel, \(\theta_p\). The gradient direction is in radians from \(-\pi/2\) to \(\pi/2\), so that can be used to determine the rise and run for the normalized direction vector.

\[
\text{rise} = \sin(\theta_p)
\]  

(3.5)

\[
\text{run} = \cos(\theta_p)
\]  

(3.6)

Repeat for the neighboring pixels by stepping the rise and run amount from the pixel, adding in the color of the neighbor, stepping \(n\) times, where \(n\) is close to 1. For \(n\) steps:

\[
\text{sumColorPositiveDirection} = \frac{1}{n} \sum_{i=1}^{n} \text{color}_{(x+i*\text{run}, y+y+i*\text{rise})}
\]  

(3.7)
That will obtain either the foreground or background neighbor color. Then step \( n \) steps in the other direction:

\[
\text{sumColorNegativeDirection} = \frac{1}{n} \sum_{i=1}^{n} \text{color}_{[x+i(-\text{run}),y+i(-\text{rise})]}
\]  

Now the colors can be added and averaged, by dividing by the number of sampled neighbors:

\[
\text{avg color} = \frac{1}{n \cdot 2} (\text{avgfgcolor} + \text{avgbgcolor})
\]  

See Figure 3.14c for an illustration of the equations. This is similar to the color training, but training on color and gradient alone is not as robust as when adding neighboring pixel information into the histogram.

**Train on Inner Color**

Training on the inner color is similar to training on the average neighboring pixels, except that a neighboring pixel is pulled only from the inner side of the selection, as in Figure 3.14d. As with computing neighboring pixels, the inner color is extracted from a pixel perpendicular to the boundary or in the gradient direction, away from the current pixel. For this feature, we want to capture a pixel that is in a uniform region of the object, not a pixel caught in the gradient of the edge. The gradient of the edge can be any width, so we keep stepping until the gradient from one pixel to the next levels off (until hitting a uniform region). The color information from the pixel we stop on is used in the PixelID.

Next, the side of the pixel that is the inside of the selection must be determined. This is easy for PixelIDs calculated during the training histogram setup, since they are derived from an existing selection boundary. With a simple, quick floodfill, we determine which pixels are inside and outside of the training boundary, then we sample perpendicularly from the boundary on both sides to find which side is the inside. However, this takes more calculation when calculating the cost map of an image, since no boundary yet exists for that image and the inside of the object is not yet located. What would be considered the inside of this pixel can only be approximated, temporarily assuming it is on a boundary. We can sample on either
side of the pixel in the direction parallel to the gradient direction (or perpendicular
to our assumed boundary). The gradient direction, though noisy, is the quickest
estimate of the boundary direction. To calculate which side is most likely the “inside,”
we compare the pixel colors in both directions with the “inner color” values in the
training set. The side with colors most similar to the training set are assumed to be
inside the object and are used for the PixelID.

3.5.3 Train on (x,y) Position

For Intelligent Rotoscoping, training is essentially the process of assigning a
cost to each pixel in an image based on the pixel’s PixelID, then using the resulting
cost map for the leapfrogging. However, it’s not quite so simple. There is an important
pixel feature that is extremely valuable to training the selection but that cannot be
treated like other features in a PixelID: the pixel’s (x, y) position. The position is
guaranteed to be unique from pixel to pixel, even in pixels that are exactly the same
for all other attributes. However, position is useful for training based on proximity.

This is a key element to making the training work on both a global and local
level. During the automated propagation, we want to be able to take into considera-
tion all training data for every remapped pixel. However, in a realistic scenario the
tracked object will have differing feature values across its boundary. One side of the
object may train on one type of color/gradient while the other side has a completely
different type. For this reason, training data that works well on one side should be
ignored on the other side, which contradicts the desire to train globally on the data.
The compromise is to compare the pixel data of the current pixel with every combina-
tion in the training histogram, but if it maps to a low cost, check if the current pixel
is within a certain distance of the pixels from which the current training data was
obtained. If it is within the distance, then the low cost can be assigned. Otherwise,
it is assigned a high cost unless another similar training combination matches the
pixel’s data and is within a reasonable distance.

Using (x,y) positional data in the training process is discussed further in Sec-
tion §3.5.7.
3.5.4 Algorithm for Setting Up Training Histogram

Here we discuss the method for setting up the training histogram. Pixels in the user-selected boundary are used as the training set.

The histogram keeps count of the number of pixels in the boundary that match a given PixelID (a given combination of values). To create the training histogram, we visit each pixel in the training boundary, use the pixel’s PixelID to index the corresponding histogram slot (each PixelID value matching with a dimension in the histogram), then increment the slot. The slot is an integer value, representing the number of same PixelIDs in the training set and the cost for that specific PixelID.

We store the maximum cost in the histogram, \( c_{\text{max}} \). Once the histogram is completed, it is inverted by subtracting each value in the histogram from \( c_{\text{max}} \), or for any given slot in the histogram,

\[
H[x_1][x_2]...[x_n]_{\text{inv}} = c_{\text{max}} - H[x_1][x_2]...[x_n]
\] (3.10)

or more generally,

\[
H_{\text{inv}} = c_{\text{max}} - H
\] (3.11)

Thus, as in the original Intelligent Scissors [5], \( H_{\text{inv}} \) assigns the lowest costs to PixelIDs that are most common in the training set. If new training data must be added to the histogram, \( H_{\text{inv}} \) is inverted back to \( H \) (i.e. \( H = c_{\text{max}} - H_{\text{inv}} \)), each new PixelID in the training data increments the corresponding slot in \( H \), \( c_{\text{max}} \) is updated if necessary, and \( H \) is once again inverted for cost map calculation.

In summary, do the following to create the histogram:

1. Create an n-dimensional histogram, where \( n \) is the number of PixelID attributes.
2. Initialize each slot in the histogram to 0.
3. Set \( c_{\text{max}} \) to 0.
4. For each pixel in the training set, calculate a PixelID.
5. For each PixelID, index the matching histogram slot (each PixelID value indexes into its corresponding dimension).
6. Increment the slot’s value, thus keeping a count of the number of pixels with that exact PixelID.

7. If the slot’s value is greater than $c_{\text{max}}$, $c_{\text{max}}$ equals the slot’s value.

Since the range of the histogram contains all possible PixelID combinations, this can lead to large data structures if the arrays were dimensioned to the full range. Specifically, if we set up a 9-dimensional array of integers, with each dimension being 256 units long, that is an array with $256^9$, or $4,722,366,482,869,645,213,696$ (close to 5 sextillion) integers. That is $10^{22}$ bytes, which, of course, is impractical for any real implementation and look-up.

**Performing a Lookup in the Training Histogram**

When the cost of any pixel, $c_p$, needs to be determined in computing the trained cost map, a lookup is done in the histogram, based on the PixelID that defines that pixel. It indexes the histogram with each corresponding value in the PixelID.

$$c_p = H[\text{PixelID}[0]]H[\text{PixelID}[1]]...H[\text{PixelID}[n-1]]$$  \hspace{1cm} (3.12)

See Fig. 3.13c.

**3.5.5 TrainingMap Data Structure**

Even though the number of possible PixelIDs (attribute combinations) that any pixel can have is huge, the number of actual slots in the histogram that will be non-zero is miniscule (i.e. there are only as many PixelIDs to increment histogram slots as there are pixels in the boundary segments used for training). Thus, in our algorithm we use a TrainingMap data structure, M, that only keeps track of slots in the histogram that are non-zero. The TrainingMap data structure is called such because it maps existing PixelIDs to their respective costs during training.

M is a doubly-linked list of each unique PixelID in the training set, with each PixelID counting the number of that PixelID in the training set (i.e. the cost of that PixelID). See Fig. 3.15. In other words, it is just a sparse matrix representation of
Figure 3.15: TrainingMap data structure only stores PixelIDs existing in the training set, as a doubly-linked list. The cost variable keeps a count of the number of matching PixelIDs. This is the cost in the histogram used for creating the cost maps.

the histogram. Therefore, all the many zero-value slots left in the histogram do not need to be stored in memory. For some pixel $p$, the PixelID $PixelID_p$ is used to index the TrainingMap data structure (which, again, is the histogram made manageable in memory). Mathematically, given a PixelID in the training set:

$$M[PixelID_p] = M[PixelID_p] + 1 \quad (3.13)$$

Given a pixel needing a cost lookup:

$$C_p = M[PixelID_p] \quad (3.14)$$

Even with the histogram now taking up little memory, the immense range of histogram slots is still a problem. The problem lies in the fact that when comparing 9 attributes of one pixel to 9 pixel attributes of another, it is likely that no two pixels in the entire training set, much less an entire image, will match exactly. In other words, the TrainingMap data structure would be a long list of PixelIDs each with a cost of one. A major explanation for this is that due to the complexity of most images, foreground and background colors do not remain constant from pixel to pixel due to natural gradients in the image, subtle as they may be. Also, noise in the image can cause two neighboring pixels on a uniform edge or region to be slightly different. This puts two almost matching pixels into completely different slots in the histogram. Thus, unwanted pixels that slip into the training selection that are very different from
edge pixels have as much weight as a group of many wanted edge pixels. The edge pixels should have the same PixelID but instead each take up a different slot in the histogram, which means all the edge pixels get a cost of one, equal to the bad pixel. Very similar pixels (not just exact matches) should be inserted in the same histogram slot, giving that PixelID a much lower cost.

The way to match similar pixels in the histogram is to scale down the ranges of the histogram’s dimensions to a more controllable range. In other words, the 256 possible values for a given pixel attribute can be scaled down to a range of 32 or 16 and floored (rounded down to the nearest integer). For example, if the 256 possible values of the red color channel of a pixel are scaled down to values between 0 and 31 ([0 − 31]), any value between 0-7 is scaled to the same value of 0, any value between 8-15 is scaled down to 1, and so forth (Fig. 3.16). Figure 3.17 demonstrates that with the scaling, two similar PixelIDs become the same PixelID, boosting its cost. This does not perfectly categorize all pixels that should be matches, but there will be a lot more correct matches. This is the equivalent effect of blurring the histogram.

To scale a PixelID from range [0-v] to range [0-w], the following calculation is done to each value, \( val_i \), in the PixelID:

\[
val_i = \frac{val_i (v)}{(w)}
\]  

(3.15)

In summary, the process for the setting up the TrainingMap data structure is as follows:

1. Initialize the maximum cost, \( c_{max} \), to 0.

2. For each pixel in the training set, do the following:
   
   (a) Calculate the pixel’s PixelID, then scale its values down to their designated ranges.

   (b) If the PixelID is not in the TrainingMap, \( M \), add a node for that PixelID and increment it to 1.

   (c) Otherwise, increment the value of the node in \( M \) that matches that PixelID.
Figure 3.16: Range of PixelID attribute, R, is scaled from range [0-15] to [0-3] for six PixelIDs. By scaling down, PixelIDs with close values become grouped together under same values. For the corresponding cost histogram, range of R-dimension also drops from [0-15] to [0-3].
Figure 3.17: Attribute values of PixelID (a) are compared to those of three PixelIDs, (b), (c), and (d). Left column shows actual values for each PixelID, where all attribute combinations are different, but it is not clear how different. Scaling them down in the right hand column shows that (b) is an exact match, (c) is a very close match, and (d) is not a match at all, even though for (d) the CB attribute is an exact match by itself in both columns. Though (b), (c), and (d) have the same number of “dissimilar” pixels before scaling, what appears dissimilar before scaling might really not be so dissimilar.
(d) If the PixelID’s cost is greater than $c_{\text{max}}$, update $c_{\text{max}}$ to the new cost.

When computing a trained cost map, $\zeta_T$, the cost of each pixel can be computed with the TrainingMap data structure, $M$, as follows. The PixelID is computed for the current pixel. With that PixelID, a lookup is performed into $M$ to get the remapped cost (the integer value in the mapping). The lowest costs should go to the most common PixelIDs in the training set, so the cost is inverted, getting subtracted from the maximum cost value, $c_{\text{max}}$. If the inverted cost is zero, it is incremented a small non-zero value, since the final cost map cannot have any zero values, so that there are not multiple minimum cost paths between seed points [5]. Therefore, for some pixel, $p$, in the image, with PixelID $\text{PixelID}_p$, the corresponding cost in the cost map is:

$$c_p = (c_{\text{max}} - M[\text{PixelID}_p]) + \epsilon$$

where $\epsilon$ is some small positive integer close to zero. If the PixelID does not exist in the TrainingMap data structure, it is assumed to be zero in the histogram and thus the maximum cost when inverted (low probability of being an edge).

### 3.5.6 Training Algorithm

The following is a summary of how training is done when a boundary is being calculated between two seed points (e.g. during the leapfrogging process).

In Intelligent Scissors, when graph expansion happens, the cost is looked up from a precomputed cost map. We don’t precompute the trained maps, to avoid doing cost lookup for unused pixels in the boundary graph expansion. When a cost for a given pixel is needed, we compute it on the fly if it is not already stored in the cost map. Since the cost of a pixel is cached, boundary segments calculated during leapfrogging can reuse pixel costs already calculated during previous boundary segment calculations for the same image.

Given the techniques we have discussed up to this point, an algorithm for calculating the training map is to run the Intelligent Scissors expansion algorithm and do the following for each pixel hit by the expansion wavefront.
1. If a cost has not been calculated for the current pixel, calculate it with these steps:

   (a) Calculate and scale the PixelID for the pixel, called $\text{PixelID}_p$

   (b) Look up the PixelID in the TrainingMap (see if there’s a matching PixelID)

   (c) If it exists, subtract the PixelID’s cost from $C_{max}$ and assign the resulting cost to the current Pixel. If cost is zero, assign $\epsilon$. Else, assign the cost to be $C_{max}$.

   (d) Use the resulting cost for the expansion algorithm and cache it out.

2. Else, use the cached out cost for the current pixel.

This is a simple look at the process, but this is still not quite the full algorithm. After running the training algorithm as laid out above, the training may get a perfect selection for one frame, but can fall apart quickly in succeeding frames. This is because even temporally adjacent boundaries of the desired object will not be exactly the same due to spatial shifting, rotations of the object, changing background, and noise from one frame to the next. Thus, one frame away from the interactive selection can already have numerous pixels on the object’s edge with PixelIDs not contained in the training set, even after scaling them down. Even if the selection in one automatically selected frame does well, the continual change in position, background, and so forth, from frame to frame means pixels are less and less likely to exactly match the training data. In other words, if we only rely on the process outlined above, the farther we get from the one frame with reliable training data, the less likely the selection will capture some if any of the correct edge. However, this can be overcome with “histogram snapping.”

### 3.5.7 Histogram Snapping

Histogram snapping relaxes the requirement for an exact match between a given PixelID and the PixelIDs in the training set, by using, instead, the closest Euclidean distance to that pixel’s PixelID.
The Euclidean distance is defined as follows: Given two PixelIDs with \(n\) features, \(\mathbf{a} = \{a_1, a_2, ..., a_n\}\) and \(\mathbf{b} = \{b_1, b_2, ..., b_n\}\), the Euclidean distance is

\[
\text{Dist} \left| \mathbf{a}, \mathbf{b} \right| = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + ... + (a_n - b_n)^2} \quad (3.17)
\]

For some pixel \(p\), after the distance is computed from its PixelID, \(\text{PixelID}_p\), to each PixelID in the training set, the \(\text{PixelID}_p\) is “snapped” to this closest PixelID, \(\text{PixelID}_{\text{closest}}\). In other words, the \(\text{PixelID}_p\) is overwritten with the values of the matching \(\text{PixelID}_{\text{closest}}\) in the training set. If an exact match is found, the cost for \(p\) is the cost in the histogram, otherwise the cost for \(p\) is the distance between \(\text{PixelID}_{\text{closest}}\) and \(\text{PixelID}_p\). This is the essence of histogram snapping. See Fig. 3.18.

One problem with the histogram snapping is that, since we use the distance between the two PixelIDs as the cost, just one mistake in the training set can propagate and grow very quickly during the automated selection, like a cancer. There are two main reasons for this. First, even if there is just a single background PixelID in the training set, it is as likely to be snapped to as all the edge PixelIDs since many pixels in the cost map could possibly be closer to it than anything else. Secondly, there are many more background pixels than edge pixels, so tens or hundreds of background pixels in the vicinity that do not have an exact PixelID match could be most similar to a bad training pixel and thus get assigned a very low cost. The propagated selection will easily grab onto these low cost background pixels.

To solve this, first we run a small median filter over \(M\), which gets rid of outliers. This helps when the median filter is small, but if it is too big, the training begins to get worse, due to loss of some training information.

Since the median filter does not always get rid of the issue, we also add the scaled down cost from \(\zeta_I\), the cost map for interactive selection, to the trained cost:

\[
\zeta_I = \zeta_I + \zeta_T \cdot \omega, \text{ where } 0 < \omega \ll 1 \quad (3.18)
\]

The trained cost still has the most weight, but this cripples the cost of pixels in uniform regions to a small degree, enough that all edges get better costs and the trained
Figure 3.18: Calculating cost of pixel $\text{Pixel}_p$ by comparing PixelID distance, $d_{ID}$, and positional distance, $d_{xy}$, between $\text{Pixel}_p$ and each pixel in the training set (from a previous frame). We define the proximity threshold to equal 2. The cost assigned to $\text{Pixel}_p$ is the closest PixelID distance (2) within the positional proximity. Note that in the training set there is a PixelID with a closer distance and another PixelID that is an exact match to $\text{Pixel}_p$, but both are too far away.
edges still get the best costs. Additional training based on positional proximity also reduces this problem.

**Applying Positional Data to the Training**

To fully implement the definition of “snapping”, snapping based on positional (x, y) proximity must be included. Snapping occurs only when \( p \) is positionally close enough to its target, like a magnet to metal. A match can be found from the entire set of pixels in the training set, but it is actually better to find the best match within a certain positional distance on the image plane. This helps ensure locality in the training. Since the edge can change significantly as it goes around the selected object, it is ideal to train against the closer pixels. For example, a pixel that is background or part of an inside edge may have a PixelID that is very different from the nearby pixels on the boundary, yet it may be very similar to a boundary pixel on the opposite side of the selection. Without taking position into account, that background pixel may train as a low-cost pixel because it matches a part of the boundary that is in a completely different context of the image. Thus, histogram snapping not only snaps based on proximity between PixelIDs but between pixel positions as well.

To quickly be able to find the positional distance between \( \mathbf{p} \) and a pixel in the training set, as we set up \( \mathbf{M} \), we also keep a linked list in each PixelID of Points (pixel (x, y) positions) where that PixelID resides. When \( \text{PixelID}_p \) snaps to its closest PixelID in \( \mathbf{M} \), \( \text{PixelID}_i \), the (x, y) position of \( \text{PixelID}_p \) is compared with each Point attached to \( \text{PixelID}_i \) to test for the closest distance between the two pixels. If the closest distance is less than some computed threshold, then \( \text{PixelID}_i \) is considered a match, otherwise the next closest PixelID is tested for proximity.

To compute the position threshold, we take an average of the scanline width of the object in the x and y directions. Then on top of that, we add the average offset of the entire object. This gives a good approximation of how far away a matching pixel could be while excluding too many inside edges or details or opposite sides of the object.
Updating the Training Set

During automatic propagation, we update the training histogram with each new selection boundary, but we do not add new PixelIDs from that selection boundary, since it would only work on the assumption that it acquired good training (edge) data. This assumption does not hold. There is no guarantee that the automated selection is completely correct (or else we would not need the user in the loop). When the computer snaps a boundary to even a small incorrect boundary and trains on it, it can ruin the selection, as discussed earlier.

To update the training histogram, new PixelIDs from an automatically selected boundary, $TB_{new}$, are not added to the training set. However, we increment the histogram value of an already existing PixelID in the training data that each pixel in $TB_{new}$ snaps to. That way, none of the PixelIDs from the initial training set are lost and no erroneous PixelIDs are added, but the already existing PixelIDs are updated. For each new frame’s selection, the positional data is wiped out and the positions of pixels in $TB_{new}$ are used. Whatever PixelID in $M$ that a pixel in $TB_{new}$ snaps to gets the positional Point of that pixel added to its position linked list.

Eventually, the training data will still inevitably become invalid, but this gets more mileage out of the initial training. To make sure the training set stays updated, the user can still sculpt, or correct, the propagation, and data from user corrections is added to the training set as good, fresh training information.

However, if the user decides to restart the automated selection from a different frame, the training set is reset and training on the new frame’s selection is initiated.

The Final Training Algorithm

Our final training algorithm to compute the cost for some pixel $p$ is as follows:

1. Calculate and scale $PixelID_p$

2. Create variables ClosestPixelID and DistanceToClosestPixelID to keep track of the closest PixelID in $M$ to $PixelID_p$

3. For each $PixelID_i$ in the TrainingMap $M$, do the following:
(a) If $PixelID_i$ is an exact match to $PixelID_p$, and it is within our positional threshold distance, the cost of $p$ is $(c_{max} - M[PixelID_p]) + \epsilon$, and we exit the loop

(b) If $PixelID_i$ is not an exact match, compute the Euclidean distance between $PixelID_p$ and $PixelID_i$, $d_{ID}$

(c) If $d_{ID}$ is less than DistanceToClosestPixelID, then test the positional distance, $d_{xy}$, between $PixelID_p$ and $PixelID_i$

   i. If $d_{ID}$ is less than DistanceToClosestPixelID and $d_{xy}$ is less than our threshold distance, then $DistanceToClosestPixelID = d_{ID}$ and $ClosestPixelID = PixelID_i$

4. If there were no matching PixelIDs within our positional threshold, then assign the cost to be $c_{max}$ Else assign the cost to DistanceToClosestPixelID

That training based on attribute proximities is histogram snapping.

### 3.6 Live-wire Manipulation

In single-frame Intelligent Scissors, it is easy for the user to correct mistakes to the live-wire during selection, so there is no need to correct it after the fact. Once the selection is closed off, ideally no more editing need take place. There is not much need for after-the-fact editing of the seed points, since the user molds the live-wire during the creation of the wire. On the other hand, in Intelligent Rotoscoping, the user watches the automatic creation and makes edits after the fact. Nevertheless, in tradition with Intelligent Scissors, “molding” still happens during the selection process. In multi-frame Intelligent Rotoscoping, “molding” refers to the interception of bad sections of boundaries by grabbing already-completed seed points and adjusting them. For live-wire editing, the cost map with the gradient magnitude masked by the Laplacian is used, as discussed in §3.3.

Live-wire manipulation consists of three parts:

1. Inserting new seed points along the wire
2. Deleting unnecessary or faulty-placed seeds from the wire

3. Moving the positions of one or more seedpoints along the completed wire

As automated selection runs, the viewing window normally displays the current frame where the active selection is happening, stepping to the next frame when the current frame’s selection completes (closes the boundary). However, to examine the correctness of already selected frames, the user can step through the frames to view them and the automatic selection will continue “behind the scenes.” If the user returns to the frame at the beginning of the automated selection, the propagation will continue to be displayed. As the user watches the propagation, if errors occur, the automated selection propagation can be stopped by the user with a click. The edit is made by the user, after which training data is updated and fixes propagated forward through additional frames that may have also been selected before the interception. As a background process, the fix propagates until reaching an interactively selected frame or an unselected frame, then the propagation continues on unselected frames.

### 3.6.1 Inserting a Seed Point

Seed insertion is the simplest of the three processes (Fig. 3.19a). The user indicates where the seed point should be inserted into the completed boundary with a mouse click. The seed is inserted at the point on the boundary closest to the mouse click. Consider two seed points, A and B, and any boundary pixel in between A and B, \( p_b \). Since Intelligent Scissors follows the principle of optimality in path search and boundary extension, the optimal boundary from seed point A to seed point B contains the same points as the boundaries from A to \( p_b \) unioned with the boundary from \( p_b \) to B. Therefore, when a seed point is inserted between two other seed points, the two resulting boundaries need not change at all from their parent boundary. Though the insertion itself does not initially change the boundary, insertion is necessary to facilitate editing of the boundary by moving the inserted seed.
Figure 3.19: Live wire editing with seed insertion. a) Boundary pixel, $p_b$, is converted to seed point (seed insertion). The union of the two new optimal (lowest cost) paths $A$ to $p_b$ and $p_b$ to $B$ is same as optimal boundary between $A$ and $B$. b) A moved seed point, $p_b$ means recomputation of paths from $A$ to $p_b$ and $p_b$ to $B$. New paths are optimal for new seed positions. c) Deletion of $p_b$ means recomputation of path from $A$ to $B$, since union of paths $A$ to $p_b$ and $p_b$ to $B$ most likely is not optimal path from $A$ to $B$ (unless $p_b$ lies on $AB$’s optimal path).
3.6.2 Moving a Seed Point

Moving a seed point requires least-cost path calculations between the moved seed and its neighbors (see 3.19b). There is more than one way to handle seed point manipulation, going from very simple methods to the more complex. The simplest is that when the user presses the mouse button and drags the mouse, only the closest seed point on the boundary follows the cursor, point for point. The seed point can, but is not required to snap to the cursor on the mouse click. As the user drags seed point $p_b$, between neighboring seeds A and B on either side of $p_b$, the boundaries from A to $p_b$ and from $p_b$ to B are updated in real-time by computing the least-cost path on the fly, then finalized upon releasing the mouse. No other boundary segments need be touched with this method.

3.6.3 Deleting a Seed Point

Deleting a seed point $p_b$ from between two points, A and B, is a step more complex than inserting a point (see 3.19c). In contrast from the rule of optimality allowing simple insertion, the union of the optimal boundary from point A to $p_b$ with the optimal boundary from $p_b$ to B is not necessarily the same as the optimal boundary from A to B. This is because point $p_b$ most likely does not lie on the optimal boundary from A to B (in contrast with insertion which is guaranteed to be on the optimal path). Still, deletion of a seed point is not complex. When $p_b$ is deleted, its adjacent boundaries are also deleted and the boundary segment from A to B is simply recomputed. The cost map expansion need only occur from A to B in order to compute this and not over the entire image, so the computational time is still low and thus the deletion interactive.

3.7 Sculptable Interface

A major importance of the Intelligent Rotoscopying approach to video segmentation is its interactivity with the user. The first key to interactivity is the real-time propagation speed of the selection through the video. The second key is the ability of the user to guide and “sculpt” the propagation, or be able to fix problems in the
propagation without disrupting the flow of the automatic selection. The definition of sculpting is that corrections occur during the selection so that the user does not have to wait for the entire selection to occur before making fixes. The process is lengthened and tedious if the user has to wait for the entire selection to occur before receiving feedback on the accuracy of the segmentation.

An essential element behind this research is the interface between the user and the segmentation tool. A tool can be ingenious and yet useless because the interface is difficult, unintuitive, or slow. The slowness and difficulty can be, in part, caused by the fact that though the selection in all the frames happens lightning fast, the selection ends up not being exactly what is wanted and the user ends up spending more time than necessary trying to correct areas where the selection got off onto a wrong path. Video cutout [10] resolves this problem by providing a spatio-temporal interface that visualizes every frame all at once, allowing the user to drag a graph cut selection through time. This works well with solid temporal adjacency for the object in question, meaning that there cannot be large jumps from frame to frame and that the shot must be taken from a camera on a steady tripod. Also, picking out alpha channels in motion-blurred or blurred edges of the desired object, and making certain the correct boundaries are selected, could be difficult.

### 3.7.1 Interactive Time

The simplicity of Intelligent Rotoscoping begins with selecting the desired object in one frame and one frame only. (The user does not have to slide a selection through every frame or have to select keyframes and hope that the interpolation between frames is even somewhat close to what is wanted.) Once the selection is made in a single, base frame, the tool immediately begins propagating that selection to the temporally adjacent frames, stepping from one neighboring frame to the next. We make it easy for the user to watch for mistakes in the selection by 1) giving the user control to step through the frames with the arrow keys even when the selection is occurring and by 2) displaying a running loop of the current selection, making it obvious when the propagation begins to miss a desired portion of the selected object or
hang onto portions of unwanted objects. As the automated selection jumps from one frame to the next, it happens fast enough that sometimes the user may not intercept it as soon as a problem occurs. If the selection gets off, the user can intercept the propagation, stopping it by clicking in the main window, then making the correction of the proper seed. (Fig. 3.3f,g).

When the selection is complete, the user can loop through the frames, displaying the selected pixels only. Frames with bad selections can be corrected with a simple move of a seed point, and that information again will be used for automatically adjusting neighboring frames, thereby getting the most out of user input, while dramatically shortening the overall segmentation over frame-by-frame Intelligent Scissors. Thus the user “sculpts” the selection, molding boundary segments in one frame or another so that it fixes the frames all around.

3.7.2 Propagating Fixes

The idea behind the propagation of manual edits to the automated selection is as follows. Each time the user makes a fix, the fixed portion of the boundary is propagated to neighboring frames in a similar manner as explained before with the initial selection. That is, the edited segment of the boundary is laid on top of neighboring frames and leapfrogged, integrating the new, leapfrogged portion with the untouched segment of the boundary. The edited portion of the current frame’s fixed boundary is propagated to the next neighbor and so forth. Whatever frames the user stops on and fixes will be assumed to be “set in stone” (or the boundaries are very highly weighted) since it may be assumed that the selection the user ends up with on that frame is what was desired and should not be affected by any propagations from any other frames. See Figure 3.20.

One method we have to propagate the selection is to automatically take the neighboring $x$ segments on either side of the user-manipulated seed and use that as the approximate segment to leapfrog over the next frame. However, using a set number of neighboring segments may not be enough or may be too much, depending on how many segments the algorithm is hardcoded to pick up. Also, the user may
need to more than just move a seed (such as deleting extraneous seeds) before the fix propagates.

A more flexible method we implemented is for the user to first make the needed corrections (inserting, moving, and deleting seeds as necessary) then to indicate the range of the selection boundary that should be propagated. This is indicated by marking the bounding seed points of desired segments to be propagated. Once they are marked, the fix automatically starts propagating.

The following steps detail the method for propagating the fix:

1. The user makes the correction of seed $s$ in a chosen frame, $f$, resulting in corrected neighboring boundaries $b_1$ and $b_{-1}$, which are adjacent to $s$.

2. $b_1$ and $b_{-1}$ are combined with adjacent boundary segments $b_2$ and $b_{-2}$, and the resulting boundary $b_{\text{temp}}$ is placed into adjacent frame $f + 1$. The adjacent boundaries to $b_1$ and $b_2$ are included to make a segment long enough for leapfrogging in the next step.

OR

1. The user makes corrections by manipulating the necessary seeds

2. The user marks seeds $s_m$ and $s_n$ that bound segments $b_m...b_{n-1}$, the segments to be propagated. Those segments are joined to form $b_{\text{temp}}$

Once either of those methods is done, the following steps are taken:

1. Seed points are selected through the polygonal approximation method across $b_{\text{temp}}$ and leapfrogged to get final boundary $B$.

2. Since we only want to correct a local section of the boundary, we integrate the leapfrogged segment into the rest of the (assumedly correct) boundary of frame $f + 1$. To fit the leapfrogged boundary into $f + 1$, we do the following:

   (a) Find the two seed points in the $f + 1$'s boundary, $s_i$ and $s_j$, that are closest to the end seed points in the leapfrogged boundary.
(b) Delete all boundary segments and seeds between $s_i$ and $s_j$.

(c) Compute a boundary segment from $s_i$ to one end seed of the leapfrogged segment. Compute another boundary segment from the other end seed of the leapfrogged segment to $s_j$.

3. Use the integrated boundary segment in $f + 1$ as the boundary to leapfrog in the next frame, and continue this process until hitting a frame that does not yet have a completed selection boundary or the final frame in the sequence.

### 3.7.3 Updating the Training Set

When sculpting the selection to fix errant boundary segments, the user is not only indicating what should be correct boundary pixels but is also tagging which pixels are incorrect for the boundary. This is because one can assume the user is moving the boundary from mostly incorrect pixels to correct pixels. This is perfect information for updating and honing the training set.

We discussed two methods for indicating the boundary to be propagated for corrections. We implemented both since they both work. The first is quicker (shift-drag and let it go), and the other allows more flexibility (make corrections, mark two seeds, and let it go). We will discuss updating the training set for both. The first is handled as follows.

When the user grabs a seed point, $s$, before $s$ is moved for correction purposes, PixelIDs are extracted from the pixels in the adjacent boundaries to $s$. This set of PixelIDs, $PID_I$, from the incorrect boundary are PixelIDs unwanted in the training set. Therefore, we step through each PixelID in $PID_I$, and if there is a matching PixelID in the training set, we tag it as bad training data. We do not remove tagged PixelIDs from the training set yet, since that same PixelID may end up in the corrected boundaries, meaning it is still good.

After the user moves $s$ and releases, PixelIDs are extracted from all pixels in the updated adjacent boundaries to $s$. This set of PixelIDs, $PID_C$, from the correct boundary defines desired PixelIDs for the training set. We step through each PixelID
Figure 3.20: Propagation of boundary fixes. a) In frame 1, selection of fish has b) an incorrect selection (in red). c) User inserts seed point in incorrect segment and moves it to correct edge (green segment). d) Corrected boundary segments and their neighbor segments placed in adjacent frame (frame 2). e) (White) Seed points extracted from segment based on polygonal fit for leapfrogging. f) Leapfrogged segment connected to closest seed’s in the main boundary.
Figure 3.21:  a) The selection boundary misses the object’s edge. Pixels (yellow) in adjacent boundaries to $s$ are bad training data. b) $s$ moved to correct edge. Pixels (yellow) in adjacent boundaries to $s$ are good training data. c) Any pixels in the corrected boundaries (green) are considered good training data even if they were in the original incorrect boundaries (dark green). Any pixels in the incorrect boundaries that do not end up in the correct boundaries (red) are removed from the training set if they are in it.

in $PID_C$ and add it to the training set using the method described in §3.5.5. If it matches a PixelID tagged as bad training data, the tag is removed and that PixelID’s cost is updated appropriately. Any PixelIDs still tagged as unwanted in the training are removed. See Fig. 3.21.

For the second method of marking the correction segment, the bad pixels must be indicated before any corrections occur. The user selects two seeds bounding the segments that are deemed to have bad pixels, then signals for those segments’ pixels to be added to $PID_I$. Once those are stored, the user makes any desired corrections. When the user marks the bounding seeds of the segments to be propagated, the pixels in those segments make up $PID_C$. Updating the training set continues as explained above.
Thus Intelligent Rotoscoping constantly takes advantage of user information. Not only does it train on an initial selection, but it also lets the user intercept the propagation at any time. It takes advantage of those user corrections to better the training set and make the same correction automatically in other frames. Intelligent Rotoscoping and the user are able to communicate on level that reduces the user’s work load.
Chapter 4

Results

The main three elements we wish to test on our Intelligent Rotoscoping tool are speed, accuracy, and reproducibility. In this chapter, we first show results of Intelligent Rotoscoping on a variety of video sequences, analyzing speed, accuracy, and reproducibility with supporting data for each. From this we will make observations of the general success of the tool. This will include making data comparisons against results from previous video segmentation tools.

After the presenting general results, we will report the details on how well the specific parts of our Intelligent Rotoscoping implementation worked and contributed to the success of the tool, addressing the following: First, we discuss the method we used for generating cost maps. Second, we analyze our leapfrogging technique, showing why it is vital to our tool’s success. Third, we address the success, difficulties, and drawbacks of our training algorithm, principally the histogram snapping technique we developed. Fourth, we cover the results of our interface and feedback loop. Finally, we discuss the memory requirements of our implementation.

4.1 Example Video Sequences

We chose a variety of sequences to test the range of Intelligent Rotoscoping, from an animated silhouette of Cookie Monster to a complex, real-life video of a ballerina. The sequences that fall between those are chosen for a degree of simplicity, yet they are complex enough to require the propagation and training capabilities of our tool.
4.1.1 Sequence 1: Cookie Monster

To demonstrate that leapfrogging snaps well to a silhouette, our first test shows our algorithm on a video of cookie monster’s dancing silhouette (Fig. 4.1). We did not run this with training since the silhouette already provides a strong edge and leapfrogging can take it from there. We include Table 4.1 with information on the results along with discussion of those results.

![Image of Cookie Monster silhouette](image)

Table 4.1: Results for the Cookie Monster sequence.

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Per Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Size</td>
<td>720x480</td>
<td></td>
</tr>
<tr>
<td>Frames</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Frames Untouched</td>
<td>8 (40%)</td>
<td></td>
</tr>
<tr>
<td>Preprocess Time</td>
<td>25 sec</td>
<td>0.64 sec</td>
</tr>
<tr>
<td>User Selection Time</td>
<td>2 min 31 sec</td>
<td>7.6 sec</td>
</tr>
<tr>
<td>Seeds</td>
<td>931</td>
<td>46</td>
</tr>
<tr>
<td>Inserted Seeds</td>
<td>12</td>
<td>0.6</td>
</tr>
<tr>
<td>Moved Seeds</td>
<td>11</td>
<td>0.55</td>
</tr>
<tr>
<td>Deleted Seeds</td>
<td>45</td>
<td>2.25</td>
</tr>
<tr>
<td>Total Adjustments</td>
<td>68</td>
<td>3.4</td>
</tr>
</tbody>
</table>

Figure 4.1: Some frames of Cookie Monster’s silhouette extracted with the Intelligent Rotoscoping tool.
Since in this case our target object is a silhouette with a very defined edge, it was not necessary to use the training. This made the snapping from one frame to the next very quick, even though we ran it at full resolution. Since Cookie Monster is dancing, the motions of his arms are sometimes very extreme so the selection would miss. The user edits were corrections to the hands. However, sometimes it caught back on after a couple of frames, showing that the better the boundary stands out in the cost map, the better leapfrogging handles the selection.

4.1.2 Sequence 2: Cinderella

Our next test case is a 60 frame clip from “Cinderella,” amounting to a 4-second clip compared with clips used in other video segmentation papers (Fig. 4.3). It is a fairly simple object (2d drawing) placed over a simple background. We ran tests with training enabled, training the selection on gradient, color, and neighboring color features. Table 4.2 shows the average results of three separate tests, which we now analyze and discuss.
<table>
<thead>
<tr>
<th>Cinderella</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Size</td>
<td>468x312</td>
</tr>
<tr>
<td>Frames</td>
<td>60</td>
</tr>
<tr>
<td>Frames Untouched</td>
<td>36 (60%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Per Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preprocess Time</td>
<td>32 sec</td>
<td>0.275 sec</td>
</tr>
<tr>
<td>User Selection Time</td>
<td>9 min 22 sec</td>
<td>9.4 sec</td>
</tr>
<tr>
<td>Seeds</td>
<td>1995</td>
<td>33</td>
</tr>
<tr>
<td>Inserted Seeds</td>
<td>47</td>
<td>0.79</td>
</tr>
<tr>
<td>Moved Seeds</td>
<td>89</td>
<td>1.48</td>
</tr>
<tr>
<td>Deleted Seeds</td>
<td>67</td>
<td>1.12</td>
</tr>
<tr>
<td>Total Adjustments</td>
<td>203</td>
<td>2.87</td>
</tr>
</tbody>
</table>

Table 4.2: Results for the Cinderella sequence.

Figure 4.3: Some frames of Cinderella extracted with the Intelligent Rotoscopying tool.
Figure 4.4: A single frame of Cinderella’s selection, automatically selected by Intelligent Rotoscoping, with minor user corrections.

**Speed**

The preprocessing time was small compared to the time to segment Cinderella, so it was not a significant overhead. Most of the time was spent in interactive selection, averaging to about 9–10 seconds per frame, though time was spent much more in some frames than in others, especially since 60% of the frames were not even touched by the user. We questioned if it may have been just as fast to use Intelligent Scissors to select Cinderella frame by frame. Making a strong effort to beat Intelligent Rotoscoping with a frame-by-frame selection, it surprisingly took twice as long, or 20 seconds a frame, even though this is a simple object.

When the selection propagated by itself, it averaged about 2.2 seconds per frame. This is less than desirable, since it means that we were sitting and watching for significant amounts without doing anything for part of the time. The slowest part is that at the beginning of each new frame it pauses to compute or update the training set. The calculation of PixelIDs to get each pixel’s cost is also slow. If the general
gradient/Laplacian cost map is used instead of trained costs, the propagation takes 0.37 seconds per frame. If not for the introduction of more errors, using the original cost map would be ideal.

Despite the slowness, our algorithm selected 36 of the frames without any user intervention. That amounts to completing about 60% of the frames in 1 minute 19 seconds. This is a result of good training data being able to find the consistent edge values, though those edges moved all over. In fact, if the selection got off track, it would often jump back on in a couple of frames (Fig. 4.5) So if the automated selection worked perfectly it would have completed the whole selection in 2 minutes 12 seconds, or a little over 2 seconds per frame!

The speed hit is taken in a minority of the frames. Even if a frame was edited by the user, it did not mean a lot of time was spent on that frame. Several of the edited frames were just a one-second slide of one or two seed points to catch the thin
area under Cinderella’s chin. So about 25% of the frames consumed the majority of
the selection time. The problem area in this sequence is where Cinderella does a full
turn. Due to the significant changing boundaries, the algorithm tends to miss her
head and arms on and off. Each time the selection missed the head, it was fixed and
the fix would propagate through all the selected frames as it should. The propagation
helped, though sometimes better than others. In a few frames, the propagated fix
would immediately miss the next frame, in which case the next frame was fixed and
propagated. In most cases, the fix would at least get one extra frame, cutting the
amount of user effort at least in half. Also, the fix propagation is very quick, averaging
around 0.4 seconds per frame, since it is only a fraction of the size of the selection
boundary. So once the user got in the correction loop, there was not much waiting
required.

Most of the time was spent doing the manual adjustments. Since the user is
able to constantly keep track of how the selection is going, little time is spent finding
the errors. Once an error was found, it required time to make sure the correction was
correct before propagating it. Also, in only a couple of cases, the fix propagations
made one or two frames worse. The longest successful propagation went five frames.

At the end, we had to flip through the entire selection one frame at a time
to find any errors, though that took only a minute since any errors are immediately
obvious due to the popping it causes along the boundary. However, this would have
to be added to conventional Intelligent Scissors too (or any other tool), which we did
not take that into consideration with our frame-by-frame selection.

Accuracy

In our results, accuracy is subjective based on visual inspection. A more
detailed study on the pixel-level accuracy of Intelligent Scissors is discussed in the
original paper [5]. For most of the selection, the automated selection or fix propa-
gations picked up the correct edge. As shown in the data, the average number of
interactive tweaks to the selection was 1.48 per frame (number of moved seeds per
frame). Since it was actually only 24 frames that were actually tweaked, the number
Figure 4.6: a) A user misplaces the boundary and thus gets significant bad training data from the background (circled). b) The resulting cost map assigns low costs to background pixels that match the training data. c) The resulting selection cuts across low cost pixels in the background.

The number of interactive edits per touched frame was 3.72, meaning that even the frames with mistakes had most of the boundary correct. Even better, if a frame had a mistake, that error would often correct itself, especially around the arms or dress.

Accuracy was mostly affected by the ability of the user to provide useful training data. In tests where we placed a bad user selection, the background pixels in the training data would lead to low costs in the background and selections would get off the correct boundary fairly quickly, as in Figure 4.7. However, when we had good training data, the selection would be very faithful to the correct edge. In fact, it would bump up costs of unwanted edges, even undesired edges that had low costs in the original cost map, which is the exact result we need. For example, the hem of Cinderella’s dress and her entire body gets picked up correctly in almost every frame with the training, whereas parts often get missed without it (Fig. 4.7).

Note in Figure 4.3 that we could not carve out the background where it shows
Figure 4.7: We demonstrate a series of frames from the Cinderella sequence selected with Intelligent Rotoscopy but without user intervention, both a) without training and b) with training. a) Frames selected without training quickly lose the entire object since the snapping is prone to grab strong inside edges. b) Frame selection with training is much more robust. Though not shown, it still misses with training, even sometimes where it hits without training. However, it does not miss nearly as much as without training.

between Cinderella’s arms and waist, though this could be resolved by allowing the addition of a couple more boundaries. This is a small disadvantage to using rotoscoping methods versus region-based tracking methods, since region-based methods used the connectivity of the regions to exclude background seen in the middle of the object.

Reproducibility

In repeated trials, times required to select Cinderella ranged from 8 minutes to close to 10 minutes, ranging so widely mostly due to user error, so the reproducibility of the selection time was not one hundred percent. There is a little skill in using the tool needed from the user to determine how fast the selection happens. The selection boundaries ended up the same each time, despite differences in the exact placement of the seeds, since the cost maps consistently picked up the same edges.

Two problems cause multiple selections of the sequence to not reproduce exactly. One was that the user selection was not the same each time, sometimes being
Table 4.3: Results for the Red Statue sequence.

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Per Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preprocess Time</td>
<td>24 sec</td>
<td>0.324 sec</td>
</tr>
<tr>
<td>User Selection Time</td>
<td>9 min 42 sec</td>
<td>14.6 sec</td>
</tr>
<tr>
<td>Seeds</td>
<td>1303</td>
<td>32</td>
</tr>
<tr>
<td>Inserted Seeds</td>
<td>34</td>
<td>0.85</td>
</tr>
<tr>
<td>Moved Seeds</td>
<td>74</td>
<td>1.85</td>
</tr>
<tr>
<td>Deleted Seeds</td>
<td>87</td>
<td>2.18</td>
</tr>
<tr>
<td>Total Adjustments</td>
<td>195</td>
<td>4.88</td>
</tr>
</tbody>
</table>

less accurate than others. The other was that the seeds and boundary segments adjusted to correct mistakes in the boundary were different or done in different orders each time. Consistency is due to the quality of the training data defined by the user selection and by how well the user decides propagated segments.

4.1.3 Sequence 3: Red Statue

The red statue sequence (Fig. 4.8) is a short sequence with the camera slowly rotating around an abstract statue. The background is still simple but more complex than in the Cinderella sequence. For example, there are windows behind the statue with very strong edges and the edges of the statue where the sun hits blend somewhat with the background. The results are recorded in Table 4.3.

This sequence is a great demonstration that the robustness of our algorithm is a result of a strong training set. Complexities in the background are not necessarily inhibitions to the selection boundary. Without the training, the selection boundary loves to snap to the windows behind the statue and not let go. However, with the training, the wire can be taught to avoid the windows. This is because the red edge is distinct enough from the building and the rest of the background. In other words,
Figure 4.8: Some frames of the red statue extracted with the Intelligent Rotoscopy tool.

Figure 4.9: A single frame of Red Statue’s selection, automatically selected by Intelligent Rotoscopying, with minor user corrections.
Figure 4.10: The red statue sequence is trained to not snap to the windows in the background. A correct selection from frame 1 propagates incorrectly in frames 2 and 3, catching onto background windows. When user corrects frames 2 and 3, training set is updated to not catch window, and selection propagates to correct edge in frames 4 and 5.

the statue is not blending with the background and does not have matching edges in the background. At first, even with the initial training set from the user selection, the selection did grab onto the windows. Each time it grabbed onto a bad edge, we would mark that edge segment as bad training data to get erased from the training set. Therefore, the first several frames were slow (unlike with Cinderella), but the time was made up once the correct edge was selected consistently from frame to frame. This was a great example of the algorithm taking interactive user input and using it to minimize the amount of work for the user. Therefore, the selection times were not as good as with Cinderella, but comparable due to the adaptability of this sequence.
4.1.4 Sequence 4: Waving Girl

This sequence of a waving girl (Fig. 4.11) comes from the Keyframe-based Rotoscopying paper and was also used to test Video Cutout. We ran it at full video resolution (640x480) so that we could compare some of the data with previous techniques. We also record not only the speed of Intelligent Rotoscopying with training implemented but also the time it takes to manually select the boundary with Intelligent Scissors frame by frame as well as the time it takes Intelligent Rotoscopying to select it without training. We now give a discussion of the data found in Table 4.4.

In this example, our selection algorithm with the training implemented went faster than manually selecting each frame with Intelligent Scissors. However, the time for manual selection is much closer to our Intelligent Rotoscopying selection time than in our previous examples (Intelligent Rotoscopying only saves 12%). The difficulty in this sequence lies in the similarity between the color values of the shirt and of
Figure 4.11: Some frames of the waving girl extracted with the Intelligent Rotoscoping tool.

Figure 4.12: A single frame of Waving Girl’s selection, automatically selected by Intelligent Rotoscopying, with minor user corrections.
the background. Distinguishing such ambiguous edges where the image hardly distin-
guishes an edge between foreground and background has not yet been adequately
solved in video segmentation. Despite the difficulty it introduces, the simplicity of
the background allows our tool to overcome this.

The main difficulty these indistinct edges cause for Intelligent Rotoscoping is
that it is as likely or more likely to catch onto other edges in the background or
wrinkles in the shirt as it is to catch onto the correct edge. It is just barely distinct
enough that hand selecting it can be done at a decent rate. Nevertheless, it still takes
careful user direction and close-together seed points since the costs of surrounding
pixels are so close. Though it takes several seedpoints to keep the selection in line,
stamping down seed points in close proximity is generally faster than moving seeds
of an incorrectly propagated boundary segment. In other words, seed placement is
quicker than seed adjustment.

Without training, the leapfrogging snaps any which way in the vicinity of the
ambiguous edges, especially since it likes the wrinkles and the edges that pass through
the background. It also easily cuts corners since everything is close to the same value.
Correcting it takes more time than just running the selection manually. However, it
picks up the strong edges, such as the arms, face, and hair just fine.

The more difficult edges are where training is helpful, as long as the initial
user selection captures the exact edge and trains on the exact correct values, since it
can be especially sensitive there. The training still is not as robust as in the previous
sequences, especially since parts of the edges blend right into the background. This
introduces training data that is correct but that still assigns low costs to background
pixels. During selection, we updated the training set by marking selected segments
as good or bad training information, but this did not work nearly so well as in the
red statue sequence. It turns out that continually trying to update the training set
slowed things down but did not make the propagation better. The main reason for
this is that when we mark sections to exclude from the training set, it is so similar
to the edge data that it gets rid of good data as well. When we mark what should
be good training data, it easily introduces bad data since the edge and background
are so similar. We ran a test where we spent most of the time trying to constantly update the training set until the fix propagations always got it right (which it never did all the way), but doing that made the selection take over 10 minutes for the first 10 frames and just as long after that.

Despite not updating easily, it turns out that the training still helps more than hinders. It does not always miss the correct edge. Note that we did not even have to interact with 2 of the 10 frames! We did the selection as usual, making corrections and letting those corrections propagate. The propagations did well on the hair and arms and face and was especially helpful for the extended, waving arm. A few times it helped with the shirt, but if the fix missed, we went immediately to correcting the missed edges by hand.

The nice thing about Intelligent Rotoscoping is that, even though it does not always hit the right boundary, it generally puts it in a good proximity to that edge, especially with training, and is easy to fix. Training is capable of knocking the selection way off, but in this case, it was quickest to let the tool do as well as it could on the first or second try then take it from there by hand. This is a great demonstration of the power of the interactivity of this tool.

One last observation is that when Intelligent Rotoscoping propagated the selection without training, it ran at about 0.4 seconds per frame; whereas, when it ran with training, it took up to 8 seconds per frame. However, the additional accuracy of the propagation and the fixes made up for that difference in time. This may not always be the case, however, such that a sequence could work best without the training since it propagates at the rates we would prefer.

4.1.5 Sequence 5: Ballerina

The ballerina is a sequence taken from the Video Cutout paper (Fig. 4.13). It is the most complex video segment we use, consisting of a person moving in many different paths and rotations and moving appendages in extreme motions. The skin tone is also very similar to the background in color and value. Following is a discussion on Table 4.5.
<table>
<thead>
<tr>
<th></th>
<th>Manual</th>
<th></th>
<th>No Training</th>
<th></th>
<th>Training</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Per Frame</td>
<td>Total</td>
<td>Per Frame</td>
<td>Total</td>
<td>Per Frame</td>
</tr>
<tr>
<td>Selection Time Seeds</td>
<td>12m 2s</td>
<td>36s</td>
<td>20m 19s</td>
<td>43s</td>
<td>16m 33s</td>
<td>50s</td>
</tr>
<tr>
<td>Inserted Seeds</td>
<td>644</td>
<td>32.2</td>
<td>844</td>
<td>42.2</td>
<td>924</td>
<td>46</td>
</tr>
<tr>
<td>Moved Seeds</td>
<td>646</td>
<td>32.3</td>
<td>364</td>
<td>18.2</td>
<td>308</td>
<td>15.4</td>
</tr>
<tr>
<td>Deleted Seeds</td>
<td>4</td>
<td>0.3</td>
<td>445</td>
<td>22.2</td>
<td>315</td>
<td>15.7</td>
</tr>
<tr>
<td>Total Adjustments</td>
<td>652</td>
<td>32.6</td>
<td>936</td>
<td>46.8</td>
<td>720</td>
<td>36</td>
</tr>
</tbody>
</table>

Table 4.5: Results for the ballet sequence.

Figure 4.13: A single frame of Ballerina’s selection, automatically selected by Intelligent Rotoscopyng, with minor user corrections.
This is where the algorithm finally falls apart. It still does not fail, but it takes longer to use the Intelligent Rotoscoping tool than to select the object frame by frame. The big drawback is that the arms and legs would constantly blend enough into the background. This is probably one of the biggest enemies to Intelligent Rotoscoping, what we would call “ambiguous appendages.” This is where appendages of the desired objects have extremely weak edges. Corner cutting is extremely prone to happen in these cases.

Even Video Cutout did not capture the hands and feet in several of their frames. The advantage of having Intelligent Scissors is that it can be used to manually capture the hands and feet in every frame after the propagation captures the easily followed red and black of the clothing. However, since this falls apart in practically every frame and since the fixes do not propagate, using the automated selection proves to be slower than manually selecting it correctly in the first place.

4.2 Discussion and Comparisons

Our Intelligent Rotoscoping tool shows that Intelligent Scissors’ features can be applied to video, though its success is based on how well the training works. Doing leapfrogging without the training does well enough, but we see from the results that if the tool can truly make good use of as much user input as possible instead of depending on general costs, it can go even further.

With Intelligent Rotoscoping and the accompanying, basic Intelligent Scissors tools, the accuracy (or proximity to the true edge) achieved by the computer is much higher after seconds of work than can be achieved after many minutes of manual, unaided selection. Even better is that the feedback comes close to the real-time range, so the user is not waiting significant amounts of time but using all the time correcting and adjusting where the tool falls short.

We compare the results of our tool with some of the results of more recent video segmentation tools, Keyframe-based Rotoscoping and Video Cutout. Despite Intelligent Rotoscoping not being as fast as we want, we have taken a new direction with video segmentation and have results comparable with previous techniques. Our
Comparison With Keyframe-based Rotoscoping

<table>
<thead>
<tr>
<th></th>
<th>Keyframe-based Rotoscoping</th>
<th>Intelligent Rotoscoping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Pts / Frame</td>
<td>117.0</td>
<td>55.1</td>
</tr>
<tr>
<td>Edited Points / Frame</td>
<td>13.4</td>
<td>8.8</td>
</tr>
<tr>
<td>Selection Time</td>
<td>40m</td>
<td>23m</td>
</tr>
</tbody>
</table>

Table 4.6: Comparisons of speed and number of control (seed) points with Keyframe-based Rotoscoping [19] for the waving girl sequence. Results for the Intelligent Rotoscoping was extrapolated from a 10-frame selection to the 36 frames in the entire sequence. Intelligent Rotoscoping ran the selection at 65% the full resolution.

tool falls behind with the most difficult sequence, but we still demonstrate the merit of our tool in comparing simpler sequences with Keyframe-based Rotoscoping and Video Cutout.

Table 4.6 makes some comparisons with numbers we computed from results found in the Keyframe-based Rotoscoping paper [19]. We see that on average, the Intelligent Rotoscoping tool required fewer seed points and a shorter selection time. However, the Intelligent Rotoscoping selection was done at 65% the full resolution, so we can assume it would take longer at full resolution. The automated selection would run slower due to the larger cost map expansions required for each seed point and the larger training data sets.

Table 4.7 makes some comparisons with numbers we computed from results in the Video Cutout paper [10]. Video Cutout explains that it had an easy time with the waving girl sequence, though to catch the ambiguous shirt edge they had to revert to splines and the Keyframe-based Rotoscoping. Intelligent Rotoscoping was able to catch the whole edge itself, despite the longer selection time. The ballerina was the toughest of our examples, though it still was not too far behind the Video Cutout. Manual selection with Intelligent Scissors was comparable to the Video Cutout times.
### Comparison With Video Cutout

<table>
<thead>
<tr>
<th></th>
<th>Video Cutout</th>
<th>Intelligent Rotoscoping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waving Girl</td>
<td>11.25 sec/frame</td>
<td>39 sec/frame</td>
</tr>
<tr>
<td>Ballerina</td>
<td>40 sec/frame</td>
<td>50 sec/frame</td>
</tr>
</tbody>
</table>

Table 4.7: Speed comparisons with Video Cutout for the waving girl and ballerina sequences. Results for the Intelligent Rotoscoping was extrapolated from a 10-frame selection. Intelligent Rotoscoping ran the selections at 65% the full resolution.

### 4.3 Cost Map Comparison

The method for computing cost maps is critical to achieving the accuracy and speed we need, both in user selection or in training. It affects the selection speed because the better the cost map, the fewer seeds the user needs to place or adjust to capture the desired object. Also, automated selection runs at real-time speeds depending on the speed of the on-the-fly calculation of the PixelID in the expanding wavefront. Cost map calculation also directly affects the accuracy of the tool. The better the cost map is at identifying edges, the more accurate the interactive selection. The more accurate the interactive selection, the better the training data and the resulting trained cost maps. Correct training data is essential for leapfrogging to snap to desired edges. In this section, we take consideration of how well our cost maps pick up the correct object boundaries.

#### 4.3.1 Masking Gradient/Laplacian vs. Adding Gradient/Laplacian

We decided to go the route of multiplying the gradient and Laplacian images (i.e. masking the gradient image with the binary Laplacian image) over adding them together. This improves accuracy of interactive selections because, due to the binary nature of the Laplacian image, masking off the gradient with the precise results in definite, sharp edges avoids as many extraneous edges as when the zero crossing is added into the gradient (as in Figure 2.4).
<table>
<thead>
<tr>
<th>Image</th>
<th>Addition</th>
<th>Masking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amira</td>
<td>33</td>
<td>18</td>
</tr>
<tr>
<td>Ballet</td>
<td>19</td>
<td>18</td>
</tr>
<tr>
<td>Elephant</td>
<td>45</td>
<td>33</td>
</tr>
<tr>
<td>Red Statue</td>
<td>13</td>
<td>8</td>
</tr>
<tr>
<td>Minivan</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>Average</td>
<td>25.2</td>
<td>17.8</td>
</tr>
</tbody>
</table>

Table 4.8: Number of seeds for user selection with added cost maps versus masked cost maps. Masking method does not require as many seeds - on average, 70% of addition. The main improvement is less corner cutting.

The masking method speeds up the user selection and corrections a significant amount compared to addition, due to less extraneous strong edges in the cost map. The user has to place fewer seed points to get the selection boundary (Table 4.8). This helps the user selection go faster, but more importantly, it helps user corrections go quicker, since the boundaries affected by moving a single seed cover longer lengths in general and snaps better around extrusions instead of cutting across them. Thus the method of calculating the cost helps user interaction speeds.

The calculation method of the gradient/Laplacian cost maps has an indirect though significant effect on the automated selection. If the cost map makes it difficult for the user to snap to the desired boundary, due to bias towards strong edges or ambiguity in the edge, the user may unexpectedly catch incorrect pixels or noise around the selection (Fig. 4.14). This can be a problem with adding the pixels or noise in cost maps, since the many low cost extraneous paths catch the boundary easily. On the other hand, sometimes the most pixels or noise at pixels or noise at gradient/Laplacian method emphasizes strong edges so much that when the selection needs to cut across too weak of an edge, it does not want to let the user do that and follows surrounding incorrect paths instead. The addition method is more flexible in those cases. Though the user must place several more seeds, it still is more forgiving.
Selection has trouble grabbing sleeve of sweater. Sleeve looks obviously different from wall to our eyes, but highlight on edge of sleeve blends edge with background. The gradient magnitude and zero crossing images are very noisy, and thus so is the combined cost map.

in allowing the user to select the very weak edge. This becomes important in images with ambiguous edges between the foreground and background. The addition was used for the “Red Statue” test case to get better results, though it initially required more seeds.

Despite the advantages of adding the gradient magnitude and zero crossing images, the masking tends to work best in many cases. Adding the gradient magnitude back in over the masked cost map makes it easier to capture the weaker edges without compromising the longer distance between seed points.

4.3.2 Computational Cost

The preprocessing step takes only a fraction of the entire selection time, which is a big advantage to using Intelligent Rotoscopy. Since all the training and cost calculations are done on the fly, the gradient and Laplacian images are all that need calculating. On a 3 GHz machine, this requires under a second for each frame at video
resolution. The same amount of video that requires a process like Video Cutout 20 to 30 minutes to precompute requires only 2 to 5 minutes for Intelligent Rotoscoping. The user has little wait time and is practically ready to go from the start. Also, preprocessing time per image is significantly faster than the original Intelligent Scissors, requiring a second for what would originally have required 4 to 5 seconds, due to exclusion of multi-sized filters and so forth. The important idea is that we have simplified the Intelligent Scissors in order to keep the user from waiting unacceptable amounts of time.

### 4.3.3 Other Implications for Video

The most important result we discovered from the cost maps is that a general cost map is not the best for automated selection. A trained cost map is more likely to achieve the “silhouette effect.” However the general cost map is still important because if the interactive selection is inaccurate, the rest of the selection goes bad since the cost maps are created off of bad training data.

One additional problem of the Intelligent Scissors cost maps in general is that they do not deal well with motion blur. Motion blur introduces ambiguous boundary and training data into the cost maps. To get around this in our test cases, we had to set up the training data (the interactive selection) in a frame without motion blur. In that case, the trained costs are likely to find a fairly good boundary through the
motion blur, though it will be jagged instead of smooth.

4.4 User Interaction

In this section, we examine the success of our interface, testing the amount of control the user has and whether or not there is much waiting from the user’s end.

4.4.1 Interaction/Feedback Loop

One of the aims of the Intelligent Rotoscoping tool is to have a tool fast enough that the user can constantly check for and correct errors instead of having to wait for an entire iteration of the automated selection before being able to do anything. In our results, the user is able to constantly check for errors but there is still some lag time waiting for the computer, due to the speed of the cost map training.

The waiting happens while the selection automatically propagates to new frames, as long as the propagated selection is correct. As soon as something gets off, the user is immediately in the loop to intercept the bad segments and let the selection continue on its way. Ideally the automatic selection would move quicker than it takes the user to step from frame to frame and check the selection for errors.
However, our tests on frames with width 360 require just over 2 seconds per frame (close to a minute to segment 1 second of footage), while our user analysis of the selection happens at rates under a second per frame. User analysis is so quick because we can analyze it by looking at the selected pixels only, our eye easily picking out inconsistencies and pops in the boundary. This is because the feedback to the user is clear and easy to analyze. The user can focus on one frame at a time and not be confused by an array of images. Also, while looping through the frames, the user can observe the selected pixels only, thus only needing to focus on each image within the second’s time and moving through them at interactive rates. As soon as a bad frame is found, the corrections happen immediately.

This is still different from previous video segmentation techniques since they require the user to wait for the algorithm to calculate the selection over all frames before being able to analyze and correct them. In Intelligent Rotoscoping, the user is still able to look at and correct any frame while the propagation is occurring. In fact, in our results, it often worked best to stop the selection when it started to get off and make the correction, than to let it finish an entire iteration before making corrections. This is because the selection is not allowed to get too far off track. Therefore, there are fewer user tweaks, since the user is not constantly revisiting each frame with multiple iterations. Once a frame is corrected, it is ready to go and as a result, multiple frames after that frame are also completely correct with little or no tweaks.

The propagation of fixes happens very quickly, covering over two frames per second unless the corrected segment covers most of the length of the boundary. This makes the tool especially interactive, since the user gets a quick feel for how well the correction helped. So once the corrections begin, the user stays significantly in the loop. As discussed in the Cinderella sequence results, the bottleneck in the selection time is the user, as long as the training cost maps are accurate enough for picking up the correct edges. The fewer tweaks the user can use to make a correction, the faster the process. Due to the nature of Intelligent Scissors, each tweak itself is real-time, taking only as long as it takes the user to adjust a seed to the correct position.
Leapfrogging Accuracy

<table>
<thead>
<tr>
<th>Image</th>
<th>Leapfrogging Seeds</th>
<th>Adjusted Seeds</th>
<th>% Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cookie Monster</td>
<td>919</td>
<td>56</td>
<td>94%</td>
</tr>
<tr>
<td>Cinderella</td>
<td>1948</td>
<td>156</td>
<td>92%</td>
</tr>
<tr>
<td>Red Statue</td>
<td>1269</td>
<td>161</td>
<td>80%</td>
</tr>
<tr>
<td>Waving Girl</td>
<td>459</td>
<td>168</td>
<td>73%</td>
</tr>
<tr>
<td>Ballerina</td>
<td>616</td>
<td>412</td>
<td>33%</td>
</tr>
</tbody>
</table>

Table 4.10: A look at the accuracy of leapfrogging as part of the whole selection process, recorded as a percentage. The “Leapfrogging Seeds” column is the number of seeds inserted by the leapfrogging (total seeds minus the number of seeds inserted by the user). “Adjusted Seeds” is the number of seeds that had to be moved or deleted by the user due to inaccuracies in the automated selection. “% Correct” is the percentage of seeds that were left completely alone.

Making one correction at a time happens quickly and is more intuitive than letting several fixes go and having to keep track of each and whether or not the correction helps or hurts.

The user mainly works off of the selection propagated from the first frame, only correcting small sections of the boundary when needed. Every once in a while, though, the background or object edge changes enough that the training data is no longer useful, so the user has to completely restart the selection for the remaining frames from that frame (which completely resets and builds the training data). Fortunately, this is much more the exception than the norm.

4.4.2 Percentage of Manually Edited Frames

The number of frames on average that a user needs to touch depends on the difficulty of the sequence. Table 4.10 demonstrates how much the automated selection gets correct.
4.4.3 Comparison With Keyframe-based Rotoscoping

A major difference between this interface and that of Keyframe-based Rotoscoping [19] is that Intelligent Rotoscoping only needs the user selection from a single frame. Since Keyframe-based Rotoscoping is a form of interpolation, it needs at least two frames of user input to start and there has to be some prediction of how the object will move between those frames.

However, Intelligent Rotoscoping makes no assumptions except that the boundary can move, rotate, or scale in any direction. It does well at snapping to the edge no matter which direction it has turned (for example, Cinderella's back and forth movements and spin), unless the edge makes a large move (for example, see the last two frames of Cookie Monster in Figure 4.1).

4.5 Leapfrogging

4.5.1 Speed of Leapfrogging

Leapfrogging selects frames at acceptable rates for interactive selection. Leapfrogging is slower than just calculating a single boundary segment from each seed to its neighbor. First, it is slower because it calculates two boundary segments from each seed point, to get the overlap, instead of one. Secondly, the second segment calculated from a given seed is longer because it has to stretch to its neighbor's neighbor, so the leapfrogging is more than twice as slow as the alternative. Nevertheless, it still runs fast enough to select frames at real-time rates. Table 4.11 shows the rate of the propagation for leapfrogging without any training, demonstrating that the leapfrogging alone runs at interactive rates. Also, the chart compares leapfrogging against propagation without leapfrogging, showing that the slower times are less than twice as slow and thus hardly a loss, especially taking into consideration the major gain in accuracy. We see that without the slow-down of training, the selection propagates at over two frames per second.
<table>
<thead>
<tr>
<th>Image</th>
<th>Leapfrogging</th>
<th>No Leapfrogging</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cookie Monster</td>
<td>0.515s</td>
<td>0.297s</td>
<td>0.58</td>
</tr>
<tr>
<td>Cinderella</td>
<td>0.110s</td>
<td>0.063s</td>
<td>0.57</td>
</tr>
<tr>
<td>Amira</td>
<td>0.235s</td>
<td>0.140s</td>
<td>0.60</td>
</tr>
<tr>
<td>Average</td>
<td>0.287</td>
<td>0.167</td>
<td>0.58</td>
</tr>
</tbody>
</table>

Table 4.11: Speed comparison of leapfrogging a boundary over a single frame versus just calculating a single boundary segment between each neighboring pair of seeds. Time is calculated in seconds. Each boundary laid down 40 seed points. We computed this over a single frame multiple times instead of letting it propagate over multiple frames because the latter biased the times based on whether or not the boundary grew or shrunk over time in one method or the other (smaller boundary means faster leapfrogging per frame).

### 4.5.2 Accuracy

Leapfrogging, in contrast with just calculating a single boundary segment between each seed, is a vital and effective element of Intelligent Rotoscoping’s selection accuracy. Our results show that leapfrogging is vital because it forces seed points to the object’s edge instead of leaving them floating away from the edge. With leapfrogging, the only thing that causes a seed point to float away from the edge would be because a seed point is attracted to a false boundary due to the false edge having feature similarities to the training set. Since leapfrogging snaps one segment at a time to the new frame’s object boundary, it assures that each seed is in place before moving on to the next (instead of dropping all seeds in approximate locations and leaving them there). Note that we are making on average 2 to 3 seed adjustments per frame where there are a total average of 30 to 40 seeds per frame. This is extremely useful since we do not have to adjust most seed points to the object’s boundary, whereas if there was no leapfrogging (Fig. 3.7), we would have to adjust almost every seed. Also, the user is not guaranteed to drop the seed exactly on the edge, whereas leapfrogging will.
Another powerful feature of leapfrogging is robust enough that if it misses part of the selection in one frame, it tends to catch back on the desired object edge in the next frame, even without any user guidance. It is much less likely to do so if propagating without leapfrogging, even if tracking a silhouette. Leapfrogging was used to track the Cookie Monster sequence without the use of training, since it is a silhouette and already has a well-defined edge. Without leapfrogging, however, it falls apart. Therefore, leapfrogging avoids the case of more than one erroneous seed in a row throwing off the entire selection.

Leapfrogging is also helpful in that it can correctly place a high percentage of the seeds, though much more difficult sequences can result in less than 33% accuracy (see Fig. 4.10). Table 4.10 demonstrates the number of user-edited seeds versus the total number of seeds in our test sequence. On the other hand, even when a user is interactively selecting every single frame with Intelligent Scissors, the seeds are not guaranteed to be right on the edge. The percent of seeds not adjusted is largely due to leapfrogging correctly placing seed points.

The problem with leapfrogging is that it tends to cut corners (cut off object extremities) more readily than without leapfrogging. Cinderella’s head tended to get missed in the section where she spins around, due to the skinniness of her neck, though it was picked up much more often than not. The training helps prevent this since it does well at forcing a very high cost inside the object, even inside the skinny regions (Fig. 4.16).

Polygonal approximation of the boundary successfully prevents most, though not all, corner cutting. In a sense, it works as another training term based on shape, since it places more seeds where there is higher curvature. Those areas of higher curvature are at the end of appendages and extrusions of the object, and when there are enough seeds at the end of the extrusion, the leapfrogging picks up edges at the end of the extrusion instead of skipping over them (Fig. 4.17).

The polygonal approximation has to find a balanced range though in how far the approximation goes. If it does not subdivide enough, it won’t define those high curvature areas with multiple seeds and the leapfrogging will skip right by it as usual.
Figure 4.16: Properly trained high costs for pixels inside the object prevents selection from cutting through object. For example, in this image from real original and trained cost maps, white pixels represent very high costs and dark pixels very low costs. The automated selection will be less likely to cut across the neck in the trained cost map, since it does not have low cost edges or canyons crossing it as in the original cost map.
Figure 4.17: The polygonal approximation method of placing seed points prevents corner cutting better than placing seed points at even intervals since it places additional seeds in sections of the boundary with higher curvature. The yellow circles are seed points extracted from the previous frame’s boundary evenly spaced (right) or polygonally spaced (left), and the blue circles are the leapfrogged seed points.

If it subdivides too much, there are so many seeds at high curvature points that the small distance between seeds prevents the connecting boundaries from snapping to any nearby edges. Also, high subdivision causes small bumps or texture in the boundary to pick up seed points where it is not necessary. In our implementation, we tend to get at least two to three seeds at the end of each major extrusion, resulting in fewer corners being cut than when seed points are placed at equal intervals.

Polygonal approximation does cause some problems when an object rotates. When an extrusion is occluded behind a rotating object from one frame to the next, the close-together seed points resulting from the extrusion try to pick up a false appendage and can grab bad boundaries. This is not a problem though when the temporal coherency is high enough and the extrusion gradually falls away.
4.5.3 Capturing Non-linear Movement

The leapfrogging works well with non-linear movement of objects in video. Unlike morphing techniques for rotoscoping, it does not have to guess where or how the object will move next. It waits for the object to move then snaps to it whichever direction it has gone. In fact, when we would try to guess the next motion based on previous motions, the tool would do worse due to the constant unpredictability of the movement. It works best without those assumptions, since most objects do not move in linear patterns or at linear speeds. This is also an advantage because it does not have to deal with the computational complexity of predicting the motion.

4.5.4 Temporal Coherency

Leapfrogging does have a need for significant temporal coherency, however. If an object moves too far in one frame such that one side of the object passes the other, the selection will grab just to the closest edge. For example, the arm of a walking person will swing fast enough that there is no overlap from one frame to the next, and the tool will miss that (Fig. 4.18).

4.5.5 Reproducibility

Leapfrogging robustly reproduces its results based on consistent input by the user and a consistent cost map. It does not randomly snap to one edge in one iteration then a separate edge between test runs if the user’s seed placement remains the same.

4.6 Training

During leapfrogging, the first segment calculated for a given image will generally take longer to calculate since it has to calculate a cost for every pixel during expansion (other boundaries will be able to reuse pixel costs already calculated).

4.6.1 Training Features

The algorithm is only as good as the cost map and thus the accuracy of the training data is important. Here we discuss each training feature and its effectiveness
Figure 4.18: This figure demonstrates that if an object’s appendage moves too far between frames, the boundary can snap to one side of it. Since the boy’s arm is so skinny and it moves farther than it’s own width from frames 1 to 3, the approximate selection ends up on one side of the arm and snaps only to that side.

first separately, then together.

Train on Gradient

The gradient makes a significant contribution to the training, similar to Intelligent Scissors. It helps distinguish between hard edges and soft edges, giving preference to edges with similar strength of the interactively selected edge. See Fig. 4.19.

Train on Color

The color of the pixel is also a powerful feature to train on. Gradient alone does not give enough information to distinguish many edges, so checking direct color similarities helps classify pixels further. The pixel’s color on an object’s edge is generally a mix of the foreground and background colors, which actually helps find similar edges because the color is different from both the foreground and the background. If the background is changing quickly, the edge colors will also change partly with the changing background. However, the color coming from the inside of the object, or
the foreground color, is much more likely to remain the same or very similar. Since the foreground colors remain fairly, if not completely, consistent, the color tends to retain enough similarity from frame to frame to be beneficial to training. PixelIDs do not have to be exactly the same to be a match, meaning that if pixels are similar enough they are considered part of the same edge, as was explained in the section on histogram snapping (§3.5.7). It turns out that this makes edge color beneficial to the training set (Fig. 4.20). Nevertheless, the foreground colors just inside the edge can also be obtained and used for training, as will be explained next.

**Train on Neighbors’ Color**

The average neighboring color on either side of a pixel did not seem like it would be too significant since it just reemphasized the color feature. However, adding this feature on top of the others is what made the training algorithm really start working well like it should. See Fig. 4.21.
Figure 4.20: Selection results training on edge color only.

Figure 4.21: Selection results training on average neighboring colors only.
Train on Inner Neighbor’s Color

Training on the inner neighbor (§3.5.2) seemed like it would be an important feature to differentiate a pixel from the background. However, it does not help much and can even hurt the selection more than help. It is also the most expensive feature to calculate, since it has to step away from the pixel in two directions and calculate how much the gradient has changed with each step. As a result of that, we generally leave the inner neighbor feature out of the training (Fig. 4.22).

Combining the Training Data

The combination of the different training features into a single training set makes for a much more powerful training algorithm than using any feature alone. It is an important synergy to make the training work. The best combination for PixelIDs in our test cases were to train on color, gradient, and neighbor color together. See Fig. 4.23.

Each new used feature means a slightly higher calculation time for each PixelID.
and thus a longer time to run the cost map expansion. However, the computational cost from gradient and color are small since they are three array lookups each. The computational cost from computing the neighboring color is slightly higher than the other features, but it is still insignificant, being a few adds then a divide, additional to the value lookups.

Finally, training on position is a significant help to the accuracy of the automated selection. It is a helpful feature to localize the costs to PixelIDs close by in the training set. For one thing, it prevents background on one side of the foreground object getting counted as a foreground edge because it has similarities to the edge on the other side of the object. It only helps and does not hurt, unless the desired edge begins to fall outside of the positional range. This was not a problem with our tests, since the positional distance is based on the size of the object and distance between its edges. See Fig. 4.24.

The time it takes to train on individual features separately is hardly faster

Figure 4.23: Selection results training on gradient magnitude, color, and neighboring colors.
Figure 4.24: Selection results training on gradient magnitude, color, neighboring colors, and position. This is close to the desired “silhouette” of Cinderella in the cost map. The gray half-circle around the foot is the result of some pixels from the foot put in the training set, which PixelIDs are close to pixels in the dress.
## Training Feature Speeds

<table>
<thead>
<tr>
<th>Trained Feature</th>
<th>Single Frame</th>
<th>Per Frame for 5 Frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gradient (G)</td>
<td>0.5s</td>
<td>1.80s</td>
</tr>
<tr>
<td>Color (C)</td>
<td>0.718s</td>
<td>2.33s</td>
</tr>
<tr>
<td>Neighbor Colors (N)</td>
<td>0.672s</td>
<td>2.08s</td>
</tr>
<tr>
<td>Inside Color (I)</td>
<td>14.03s</td>
<td>11.55s</td>
</tr>
<tr>
<td>G, C, N</td>
<td>1.06s</td>
<td>3.12s</td>
</tr>
<tr>
<td>G, C, N, Position</td>
<td>2.73s</td>
<td>1.81s</td>
</tr>
<tr>
<td>G, C, N, I, Position</td>
<td>8.55s</td>
<td>6.94s</td>
</tr>
</tbody>
</table>

Table 4.12: Speed comparison of training with various features separately then together. All training is not done taking position into account unless otherwise specified. We did tests over a single frame and over five frames, recording per-frame computation times. Each frame does not take the same time to compute, due to updates to the training set and the length of the selection boundary is each frame, so the selection can speed up or slow down over time.

than the time it takes to train on all the features together. This is because the main computational cost in the algorithm happens after the PixelID is formed, when comparing it with the PixelIDs in the training set. The one exception is training on the inside color, since it takes considerable more time to find which side of the pixel has a more similar color to inside color information in the training set. Table 4.12 shows the general times required for training with each feature separately and all together.

### 4.6.2 Histogram Snapping

Histogram snapping is a quick way to fill in a multi-dimensional histogram when the number of filled buckets in the histogram is sparse, so in that sense it is very effective. For computing costs, the idea makes a lot of sense. In fact, in images with significant contrast between the foreground and background, it works well. However, in general practice, assigning a pixel a cost by a lookup to its most similar PixelID is not the most effective method of computing a cost, at least according to our results.
Getting a histogram with nice representative peaks does not happen as easily as we thought it should. If the pixel features are not scaled down, as explained in our methods, every PixelID will more likely get its own bucket since the probability of there being an exact match in a 9-dimensional histogram is very small. This results in a flat histogram where background PixelIDs slipping into the training set have as good a cost as edge PixelIDs (Fig. 4.25). When a histogram is scaled down too far, even though it has better peaks, it can generalize the buckets too much such that background pixels fall in the same cost bucket as edge pixels. Also, even if the histogram is scaled down, the histogram can end up pretty flat due to the variety and distribution of the pixels throughout the edge. In that case, a lot of inner edges end up finding a match in the histogram and receiving equal costs to the desired edges. We cannot necessarily weight the costs by distance from the training set to solve this because, again, we have no idea which way the desired object boundary is going to turn. An inner edge could end up positionally closer to the training set pixels than the correct edge.

Doing the lookup in the histogram is used for the PixelIDs that find an exact match in the training set. Otherwise, we set the Euclidean distance between the features of two PixelID as the cost. It turns out that the distance between PixelIDs is assigned as the cost for at least 99% of the pixels in the image, since not many PixelIDs find exact matches. This is much more robust than assigning the histogram value. When we just assign the histogram value, it is not highly guaranteed that the edge pixels in a new frame are close enough to a PixelID in the training set, meaning there is a good chance it does not receive a low enough cost. Due to this uncertainty, the training is not very strong. Even if the background and edge values are changing from frame to frame, as long as the boundary pixels are still the most similar PixelIDs to those in the training set, they get the lowest costs.

With this cost assignment method, our results work much more accurately when the feature scaling range is higher (such as 128) instead of lower (such as 16). This is because with higher feature ranges there is more variety and range in the distances between PixelIDs, distinguishing edge costs with higher sensitivity.
Figure 4.25: This figure demonstrates the varying profile of the training set histogram based on how much each feature is scaled down. We output the 2-dimensional versions of the training set histograms for scaled feature ranges of 0 to 16 and of 0 to 128. For each, we also output the histogram after training on a single frame and after training on four frames. Since PixelIDs are more likely to fall into different buckets for the 128 range than for the 16 range, the 128 range is flatter and more PixelIDs have equivalent weights. After a few frames of training, the histogram profile obtains more peaks, though this still does not guarantee the success of training with histogram snapping.
When basing the costs off of the distance to the closest PixelID, the user’s selection MUST capture only accurate edge pixels with no unwanted background pixels to make Intelligent Rotoscoping completely robust, as in the Cinderella example. However, if even a single background pixel slips into the training set, background pixels will match it as a very similar PixelID in the training set and get assigned a low cost since the Euclidean distance is small. This can be devastating to the selection since it will tend to snap to background pixels as easily as to the correct edge if it is close enough to the training data.

At least a few background pixels inevitably slip into the training set, due to noise in the cost map or else the user not placing a seed point exactly on the edge. The median filter gets rid of these outliers as it should, except where the object’s boundary is fairly ambiguous to the background and lots of noise is introduced into the cost map between the foreground and background. More background pixels get picked up in the training set than can be filtered out. Adding the cost map back in helps the selection capture the correct edge better, since the background pixels do not quite have as high a cost. However, it also reintroduces the dilemma that the inner edges end up with equivalent costs to the object’s edge and corner cutting may occur.

Training on position actually helps alleviate this a bit, since the background pixels getting low costs from the training set are local to where the bad data was introduced. At least background pixels far from the bad training pixels will not get assigned low costs.

4.6.3 Updating the Training Set

For each new frame in the propagation, the training set is set up or updated. This causes a significant pause before the first seed in each frame shows up, making up half of the time taken for that frame to be selected. Automated selection would reduce this to half the time it currently takes if we did not update the training set each frame. However, omitting that step is not an option since doing so means the proximity information in the training set is not updated. If the proximity information
of the PixelIDs in the training set is not updated, then the edge often falls out of range of the training data and thus is assigned high costs.

Updating the training from user correction segments also helps strengthen the training data, especially when we mark incorrect segments as bad training data. This helps clean out bad data that was introduced into the training set, though sometimes it can clean out too much from the training set so that ambiguous edges are no better off than before since they have nothing (instead of too much) to train off of. When a user marks a corrected portion of the boundary to be propagated, updating the training set with the corrected positional data is effective in helping the propagation snap to the correct proximity.

4.6.4 Stored Data Structures

All the images and their corresponding cost maps need to be stored in memory for the duration of the program’s run time. Following is a discussion of what we store in memory and the space required.

The original image for each frame must be displayed for user selection, so we store it out as a QPixmap, an object compatible with displaying an image to the screen. The cost map is stored out as a 2D array, with dimensions of width and height. The width, height, and depth are equivalent to that of the original image. Each color channel is four bytes. Therefore, a 720x486x3 (RGB) video image requires approximately 4 MB of data memory per image. Since the user is allowed to edit any image at any time, none of the cost maps can be thrown away.

We also store pixel information that is useful in the training algorithm. The training algorithm is used for updating cost maps for automated selection, based on the properties of pixels in the user selection. We store the R, G, and B values for each image for quick accessibility to the color data, which is not readily available from the display image object. We also keep the gradient magnitude and gradient direction information for each image, each having an array the size and depth of the image. For a video image, this totals to about 9.33 MB of information stored per image in the video sequence.
During automatic selection in the Intelligent Rotoscopying process, several seeds must be placed and the lowest cost path for each adjoining boundary segment must be calculated very quickly, totalling within a second per frame, to keep it interactive. For the graph search expansion, we use a linked list data structure for the stack containing active pixels on the wavefront of the graph expansion. A node in the linked list contains its pixel’s (x,y), the direction it’s pointing (N, NE, E, SE, S, SW, W, and NW), and a pointer to each of its neighbors in the linked list. To make pushing and popping operations on the stack as fast as possible, we omit any method calls and inline the “push” and “pop” linked list operations. Also, to avoid the overhead of creating and deleting unneeded nodes, we create and store a node for every pixel in the image as part of the preprocessing pass. The nodes are stored in a 2D array the size of the image, so pushing and popping is a simple operation of adding or removing a pointer to the proper node in the node array. Each node is about 12 bytes, so a 720x486 image stores approximately 4 MB of data per image.

Thus, for each image, we have to store up to 18 MB of data. This is a drawback from other video segmentation tools since it restricts Intelligent Rotoscopying to only handling a few seconds of footage at a time. However, as the amount of RAM increases on computers, this becomes less of an issue, unless the amount of footage needing fixing continues to increase.
Chapter 5

Limitations, Future Work, and Conclusions

In the introduction, we discussed a variety of roadblocks that must be overcome for robust video segmentation. Due to the complexity and generality of the problem, we were able to address only some of the issues satisfactorily, others to a lesser degree, and some none at all. We were able to successfully snap the selection boundary from frame to frame without the need to predict the object’s direction prior to snapping. The training also worked successfully, applying user input and corrections over multiple frames without user intervention. We were able to demonstrate that using multiple features for a training set made the training more robust. The current characteristics of Intelligent Rotoscoping, such as boundary snapping, training on user input, and selecting at interactive rates, are promising enough to merit future work and expansion of its concepts. In this chapter, we discuss some of the limitations of Intelligent Rotoscoping and suggest ideas for future work that can further improve the tool.

5.1 Limitations and Future Work

Some general problems inherent in rotoscoping issues were not addressed by our Intelligent Rotoscoping. Rotoscoping does not handle holes in an object. Our tool does not handle multiple boundaries on a single frame or propagation of multiple boundaries, but implementing multiple boundaries per frame would be a simple extension of our current data structures. When an object passes in front of our target object, occluding significant portions of the edge, the selection boundary can get lost, especially if a background object such as a lamp pole cuts the desired object in
two. Our tool does not handle cases where large parts of the target object are lost or separated then come together, though minor occlusions can be handled by updating the training set to handle the new background edge between the two objects.

Another significant problem with video is motion blur, since a sharp boundary no longer exists when the foreground and background smear together. We discussed in results that minor motion blur is handled by Intelligent Rotoscoping, but if the blur is too wide, the training information will not distinguish clearly between foreground and background. Similarly, it will not track regions that are defined by a gradual gradient instead of a sharp edge. For these cases, a more complex, statistical algorithm would most likely be needed.

5.1.1 User Interface

Currently, our interface has a single window for feedback, so it can still be slow for the user to find problems in the boundary. It would be helpful to have a separate feedback window that is constantly looping the current selection, updated in real-time, so that the user can watch it and quickly see where pops and other mistakes occur in the boundary. An additional level of interfacing between the user and the selection loop as well as the main window and the selection loop would need to be implemented.

Our method of boundary editing could also be improved to leverage more of the user’s input. Currently, the user can only edit one seed at a time. However, if the user moves a seed, Intelligent Rotoscoping should be able to decide whether or not to automatically adjust neighboring seeds with it, possibly moving them a weighted distance from the user’s seed. This way, if multiple adjacent boundary segments need editing, the user can move or get rid of multiple seeds in a single stroke. Since user edits constitute a significant slow down to the process, this could speed up the more complex selections and the training set updates from the user.
5.1.2 Cost Maps

We simplified and adjusted the Intelligent Scissors cost maps to speed up user adjustments and help selections wrap further around corners, but these general cost maps can be problematic on very weak edges (such as the waving girl’s shirt blending with the back wall). If there was a way to get extra precision on the selection in those areas without slowing the selection time, it would create a stronger training set for those areas. A simple method we could have used to help with the precision would be a zooming option to see the selection closer.

5.1.3 Training

The most important future work could be done in the area of the training. For one thing, training should not be so sensitive! If minimal amounts of bad training data enters the training set, it wreaks havoc on the selection boundary. However, the training method is robust with correct training data, so research could be done in how to keep the training robust while allowing a buffer for the inevitable background pixels that enter the training set. The histogram snapping provides such a buffer, keeping the small amount of background pixels to much lower peaks than the more prevalent good pixels. Our histograms did not produce ideal peaks, so more work can be done in figuring out a way to make those peaks more representative of the percentage of good and bad pixels in the training boundary, as seen by the user, possibly through subpixel edge estimates [15]. Then computing the costs could be a mix of basing the costs on distance between PixelIDs and on the value in the histogram.

Future work could also be done with other machine learning techniques that may be fast enough to deal with our training data, especially as the speed of computers increases. Also, the training data is handled on a per frame basis, so maybe the data could be interpolated over frames or the automated selection could learn something from the change of training data over time.

We did not do an exhaustive implementation on all features that can be trained on, so more features could be discovered. Also, our technique for training on the inner neighbor could be better, since it did not produce the results we had hoped for. We
could possibly train on the inner color, not only based on its color but also based on
its contrast from the outer color on the other side of the pixel.

Our method of positional training depended on a dynamic threshold. This
was done so that if the closest PixelID was far away, it would not even be considered.
Another method of handling this would be to find the closest pixel then weight it based
on its distance. However, it is best to completely ignore that pixel and consider a
slightly less similar PixelID that is positionally much closer.

Some improvement could also be done to train the selection to be able to follow
appendages better even when they move too far for the selection to follow from frame
to frame, such as the a swinging arm of a walking person. This would probably fall
back into the realm of morphing, though, which would have to be implemented in
a way that does not impede the interactive nature of Intelligent Rotoscoping, as we
found it to do.

We updated the training set at the beginning of each frame, causing a pause
for each frame. If there were a way to update the training set on the fly as soon as a
segment is placed, the process could probably be sped up.

5.1.4 Data Structures

Our training was slow partially due to the fact that we implemented it with the
standard template libraries, meaning there were millions of method calls happening
for each frame. Linked list data structures that require only manipulation of pointers
could be used to speed up the training.

5.2 Conclusion

In conclusion, Intelligent Rotoscoping proves to be a video selection algorithm
that leverages off of user input to propagate selections over multiple frames. Even
though user intervention is allowed at any time, it can be minimal enough that the
computer does the significant majority of the work at a faster rate than the user could
accomplish. Long-running selection algorithms with frustrating interfaces and input
settings would probably get the exact boundary long after the user could complete
it manually, pixel by pixel. However, we have shown that Intelligent Scissors can be
used to create a moldable, trainable video segmentation tool that does not require
long waits, either in pre-processing or during the selection process.
Appendix A

The algorithm for creating the histogram using a Map data structure is as follows:

Given a set of pixels in a selection boundary to be trained on:

Method CalcPixelID:

Vector CurPixelID

Do:

Extract the 3 color channels
Scale the color channels from range 0-255 to range 0-30
PixelID.push channels 1, 2, and 3
Repeat for extracting gradient and neighbor information

Return CurPixelID

Map TrainingMap // maps each PixelID to its corresponding cost
Map PositionMap // maps each PixelID to all (x,y) positions where that PixelID lies in the Training Selection Path.

For each pixel:

Vector CurPixelID = CalcPixelID (see CalcPixelID method above for details)
If CurPixelID is not in the TrainingMap and...
...the current frame is not the training frame:

nearestPixelID = NULL
nearestDistance = INFINITE

For each HistPixelID in the TrainingMap:
curDistance = distance from CurPixelID to HistPixelID
If HistPix is closer than nearestDistance
   nearestPixelID = HistPixelID
   nearestDistance = curDistance
CurPixelID = nearestPixelID
If CurPixelID is not in the PositionMap:
   PositionMap.insert(Pair<CurPixelID, CurPixelID_xy_coordinate>)
Else
   PositionMap.find(CurPixelID).pushback(CurPixelID_xy_coordinate)
If CurPixelID is not in the TrainingMap:
   Add CurPixelID to the TrainingMap
   Map CurPixelID to an integer value of one
Else
   Increment the integer that CurPixelID maps to by one
Invert the histogram so that the histogram maps lowest costs to the best PixelIDs
Bibliography


