APPROVAL SHEET

Title of Thesis: GPU Random Walkers for Iterative Image Segmentation

Name of Candidate: Sean Peter Dukehart
M.S. in Computer Science, 2009

Thesis and Abstract Approved: ______________________________
Dr. Marc Olano
Associate Professor
Department of Computer Science and
Electrical Engineering

Date Approved: ____________________________________

Curriculum Vitae

Name: Sean Peter Dukehart.

Permanent Address: 155 E Green Street, Westminster, Maryland 21157.

Degree and date to be conferred: M.S. in Computer Science, Febrary 2009.

Date of Birth: May 16, 1981.

Place of Birth: Baltimore, Maryland, USA.

Secondary Education: North Carroll High School, Hampstead, Maryland.

Collegiate institutions attended:

University of Maryland Baltimore County, M.S. in Computer Science, 2009.

Salisbury University, B.S. in Computer Science, Summa Cum Laude, 2003.

Major: Computer Science.

Professional positions held:

Technical Lead, Social Security Administration. (July 2007 – Present).
IT Specialist, Social Security Administration. (July 2004 – July 2007).
IT Specialist, Lockheed Martin. (August 2003 – July 2004).

ABSTRACT

Title of Thesis: GPU Random Walkers for Iterative Image Segmentation

Sean Peter Dukehart, M.S. in Computer Science, 2009

Thesis directed by: Dr. Marc Olano, Associate Professor
Department of Computer Science and
Electrical Engineering

Image segmentation is the act of partitioning an image into distinct regions based on properties that the pixels in those regions share, such as luminance, texture, or color. Image segmentation finds application in fields ranging from medical imaging to computer vision, all of which require the ability to distinguish contiguous regions. Based on user-specified foreground and background “seed” pixels, the random walkers segmentation algorithm calculates probabilities for pixel-placed random walkers “walking” across additional pixels that they are connected to and arriving at one of the seeds. The probability is calculated based on the variance of the shared property.

This thesis presents an algorithm to expand the usefulness of random walkers that provides the ability for interactive image segmentation and refinement. This approach minimizes delays in visual feedback during segmentation through the use of iterative processes. Starting with lower convergence thresholds leads to lower initial probabilities with less definitive segmentations. As more seed points are added and more is known about the desired segmentation, the system maintains interactivity for further refinements through dynamic convergence thresholding. To aid in this computationally intensive process, highly parallel graphics processing units are employed. The implementation is developed as an Adobe® Photoshop® plug-in to enable comparison with other currently available image segmentation techniques.

GPU Random Walkers for Iterative Image Segmentation

by

Sean Peter Dukehart

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
M.S. in Computer Science
2009

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Marc Olano for his guidance and direction on this work. I would also like to thank my family for their love, motivation and patience with me as I focused so many of my efforts and so much of my time in the pursuit of this thesis' completion. Without them, and the grace of God, this would have never been possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

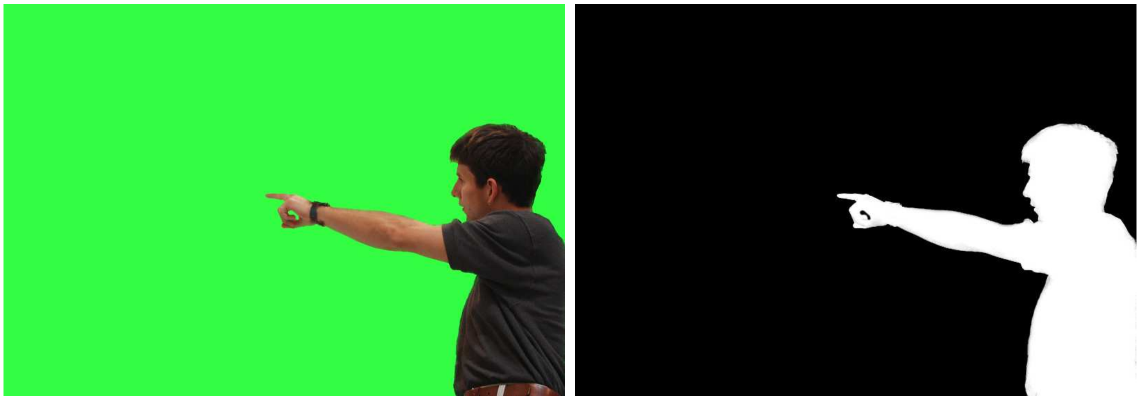
Chapter 1

INTRODUCTION

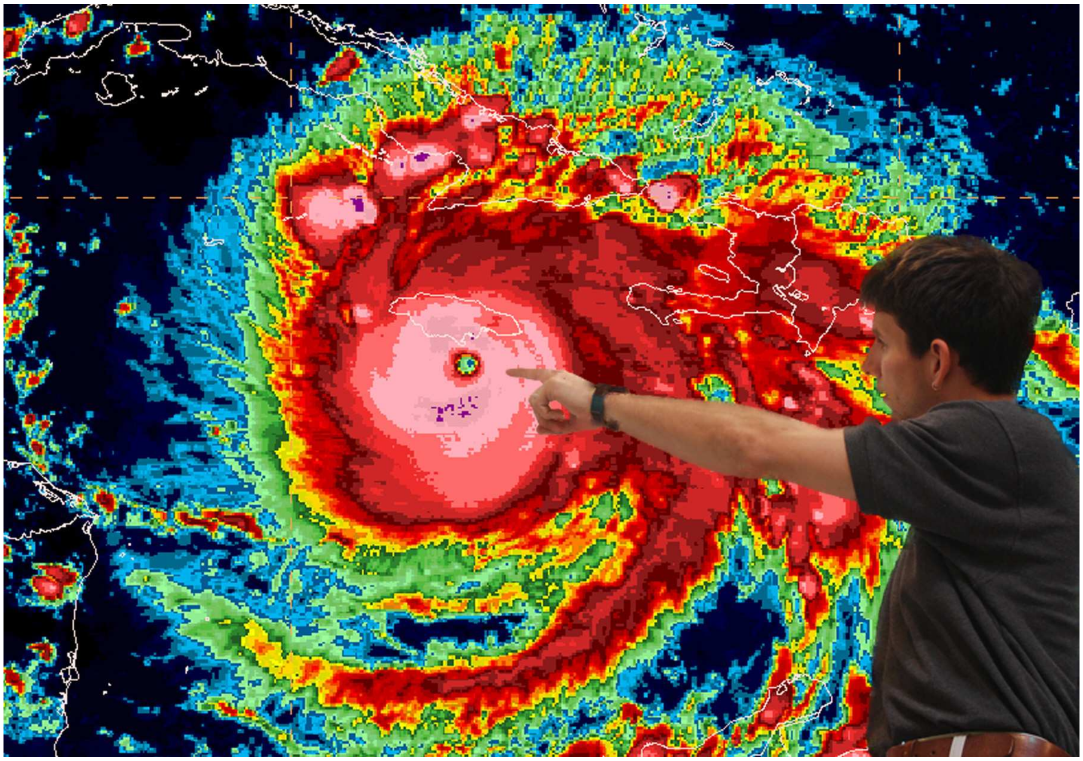
Segmentation is the process of identifying different pieces or groupings of similar information from a larger collection of information. The problem of segmentation appears in a number of contexts—for example, volume segmentation offers doctors the ability to extract a model of an individual's aorta or other internal structures from cardiac CT (computed tomography) volume data (Sherbondy, Houston, & Napel 2003). With video segmentation, it is possible to track objects through several frames and to place these objects into a new scene (Wang *et al.* 2005). Although segmentation problems arise in many fields, the focus of this paper is on the application of segmentation to imaging. With the advent of digital cameras, high-resolution scanners, and a multitude of other digital imaging input devices, the challenge has arisen of what to do with the images that they produce and the information that these images contain. It has become increasingly important to develop tools and processes for extracting this information.

1.1 Image Segmentation

In controlled situations such as video chroma keying, extracting the foreground region from a uniformly colored background is straightforward (see Figure 1.1). It is feasible to require a uniform background behind a television weather anchor, but many segmentation



(a) Green Screen (b) Segmentation



(c) Newly Composited Scene

FIG. 1.1. An example of the compositing made possible by segmentation. Weather anchors can stand in front of a green-screen background, and have the uniform colored screen manipulated in image / video space as if it were a completely separate entity. This process, known as “chroma keying,” enables weather maps to be displayed behind them as if the anchor were standing in front of what is actually a separate scene.

(a) Montage Segments (b) Final Composite

FIG. 1.2. Using Interactive Digital Photomontage segmentation and restoration to create a final composite image from multiple sub-images, where each individual color overlay indicates the sub-image that a segmented region came from (Agarwala *et al.* 2004)

situations are not so flexible. For instance, a tumor or cancerous region in a medical image is in many ways similar to the surrounding non-cancerous regions. The problem of breaking the underlying image into more meaningful parts based on what would be defined as objects and boundaries is exceedingly challenging. Difficulties include noise, similarity of texture between objects, semi-transparent foreground objects such as hair, and under-constrained problem specifications. The ultimate goal is to overcome these difficulties and to glean quantifiable information from images through the *image segmentation* process.

Image segmentation is used in fields ranging from face and fingerprint detection, to advertising, to machine vision. As can be seen in Figure 1.2, it has been used in composition, enabling a single image to be created from a montage of aligned images. This process permits desired parts of multiple images to be combined into a final composite that shows an “ideal” scene. Image segmentation and volume segmentation are more commonly used in the medical field, permitting object boundaries in x-rays or scans to be distinguished that would otherwise be extremely difficult to differentiate.

1.2 Technology

In 1965, “Moore’s Law” (Moore 1965; Moore 1975) predicted a doubling of the number of transistors in electronic components every two years, an increase that has continued through the present day. However, the maximum throughput of modern-day CPUs has met a number of bottlenecks, so that performance no longer tracks with the exponential growth in transistor count. As more transistors are added to a smaller and smaller space, interconnections and power consumption limit how quickly new processors can perform. As a result, processor manufacturers have moved to multi-core parallel computing (Intel 2005). The notion is that if tasks can be broken down into parallel parts, more can be done in the same span of time, thus increasing throughput and allowing for improved performance. This observation has resulted in dual and quad core processors from the major CPU manufacturers trying to leverage the opportunities that parallelization affords.

Although CPUs have seen a great deal of improvement from this parallelization, today’s GPUs (Graphics Processing Units) found in consumer-level video cards already have inherent parallel computing pathways built in. Since a graphics card’s purpose is to rapidly generate display information, it is designed for sending a screen’s worth of pixel information to the monitor 30 or more times per second. The approach that GPUs have taken is to employ parallel pathways for the generation of pixel information. This has resulted in a significant increase in the amount of processing that can be done concurrently. When processes utilize the available parallelism of the processor, certain classes of problem gain huge increases in performance.

Comparing the 51.2 peak Gigaflops of Intel’s® current top-of-the-line 3.2 GHz

QX9775 Core™ 2 Extreme CPU (Intel 2008) to the 230.4 peak Gigaflops of the 2-year-old NVIDIA® GeForce® 8800 GTS GPU used for this thesis (Charpentier 2007), it is clear that in terms of floating point throughput, the GPU comes out the winner. With current GPUs breaking the Teraflop barrier (AMD 2008b), as well as their pervasive use of more and more silicon (upwards of 1.4 billion transistors (NVIDIA 2008b)), the power and performance of the modern-day GPU will continue to be a major selling point for moving away from the CPU and into the much more parallel world of GPU computing.

1.3 Problem Area

As technology advances and processing power increases, there is a necessity to develop algorithms that were previously not feasible due to hardware restrictions. Today's most accurate single-image segmentation algorithms are too slow for true interactive use (Boykov & Kolmogorov 2003; Agarwala *et al.* 2004; Grady 2006). Other algorithms are designed for interactivity, but give lower-quality results (Mortensen & Barrett 1995). There is a gap between speed and accuracy-motivated solutions to the segmentation problem. With hardware continually becoming less of a restriction to what algorithms are possible, new approaches to interactive single-image segmentation that fill this gap are possible.

1.4 Contributions

The GPU-based random walkers segmentation method presented in this thesis, or *GPURW* as it will be referred to herein, extends the random walkers image segmentation algorithm (Grady & Funka-Lea 2004) (the standard algorithm is discussed in Section 2.2). GPURW is an interactive segmentation algorithm for single images that allows foreground / background objects for a given image to be distinguished. It presents a parallel solution for filling the speed-accuracy gap, providing an algorithm that is both fast and accurate. Other systems have sought to produce high-quality results after initial segmentation or at each user interaction point, but are based on minimal knowledge of a user's desire. This can lead to results that fail to match what the user actually wanted. The GPURW approach uses accumulated accuracy; as the system learns more about a user's intentions, the segmentation will more closely match their target segmentation.

GPURW is implemented using NVIDIA® CUDA™ (Compute Unified Device Architecture) technology for doing general-purpose processing on the highly parallel graphics processing unit, also known as *GPGPU* (General Purpose Graphics Processing Unit) computing. The algorithm is incorporated into an Adobe® Photoshop® plug-in to present a standard testbed for comparison with other current solutions to the image segmentation problem.

The GPURW algorithm produces segmentations that equal those of the standard random walkers algorithm; the distinctions are that GPURW does so interactively, and does not require being run to the standard algorithm's level of convergence. It provides immediate visual feedback on the current segmentation, producing segmentations in comparable time to other Photoshop segmentation plug-ins. The resultant segmentations are as good or better than those produced by other binary (object / not object) segmentation methods.

1.5 Organization

Chapter 2 examines image segmentation approaches and algorithms, in particular the random walkers image segmentation algorithm, and presents the relevance of graphics hardware to the GPURW algorithm. Chapter 3 describes how this particular approach to image segmentation was designed and implemented. Chapter 4 presents the results and considerations. Finally, Chapter 5 discusses limitations of the approach, examines areas that might be fruitful for future work, and presents conclusions.

Chapter 2

BACKGROUND AND RELATED WORK

This chapter describes technical concepts that must be understood for the remainder of the thesis. Prior work in the field of image segmentation is discussed, and the technology needed for the implementation outlined in Chapter 3 is presented.

2.1 Image Segmentation Overview

Image segmentation is the decomposition of an image into meaningful parts. Segmentation can be expressed as a selection, a definitive extraction, or a matting of some part of an image. The number of possible segmentations of an n-pixeled image into $0 \leq m \leq n$ distinct labellings is m^n. With such an immense number of possible ways to partition the image, the question becomes how to identify the “right” partitions that present the desired meaning. Since meanings are generally perceptual in nature (Is a person’s clothing part of the person? Is a license plate defined by the outline of the plate or by the letters and numbers?), it is increasingly important to recognize that there is not generally a single “correct” partition. Rather, the task is to partition the image into the best approximation of the user’s desired meaning.

Falcão et al. (1998) indicated that a segmentation must be repeatable, accurate and efficient. Grady (2006) extended this assertion by indicating that for an image segmentation algorithm to be a practical solution, the following must hold true:

1. It must be fast to compute.
2. It must allow for fast editing.
3. It must produce intuitive segmentations.
4. There must be the ability to produce arbitrary segmentations with enough interaction.

Although the definition of *fast* can be debated, the general premise was that a user of an image segmentation algorithm should not have to provide initial input for a segmentation and then return days later to see a result. Since they could have interacted with the algorithm incorrectly or unintentionally, there must be relatively immediate indications of the results of their action so that refinements can be made.

There are three generally accepted classes of segmentation: manual, automatic and semi-automatic. The manual pathway requires complete user involvement; that is, the only parts of the image to be selected for segmentation are those that the user has directly chosen. Examples of this approach include individual pixel or explicit region selection, as can be seen in the rectangular, elliptical, or other shape-based selection and free selection tools found in nearly every image-editing suite. Automatic approaches exist but are nearly always specialized to handle a very specific type of object, such as license plates (Acosta 2004) or classes of objects learned from previous user interactions (Lee & Street 2000).

In contrast, semi-automatic approaches involve varying amounts of user interaction. The first class of semi-automatic segmentation approaches, known as *seed-based* segmentation, requires points to be specified as either inside or outside the boundaries of objects (Boykov & Jolly 2001; Lombaert *et al.* 2005; Grady 2006). A second class of methods deals with explicitly indicating object boundaries, and is sometimes referred to as *boundary tracing* (Mortensen & Barrett 1995; Wang, Agrawala, & Cohen 2007). There is also a third approach, which involves indicating a bounding shape around an object (Rother, Kolmogorov, & Blake 2004). This approach inherits from both seed-based and boundary tracing methods. These semi-automatic approaches encompass nearly all recent work in image segmentation.

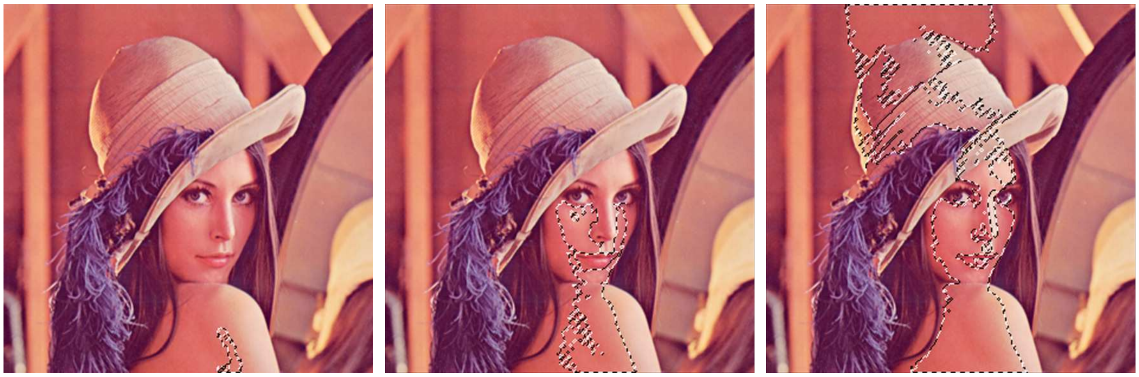
2.1.1 Magic Wand / Fuzzy Selection

Both Adobe Photoshop's *Magic Wand* (Adobe 2008)[1] and GIMP's *Fuzzy Selection* (GIMP 2008) tools can be thought of as color similarity tools. They can be likened to the region-growing methodologies of Haralick (1985), which were based on pixel intensities. A user clicks on a starting pixel in the image and nearby pixels with colors similar to the starting point are selected. This seed-based approach enables segmenting an image into regions of similar color that can be (but are not required to be) contiguous. Thus, either solid objects or spans of like color across an image can be selected.

One of the key elements to this style of segmenter is the use of a threshold or tolerance that dictates how close similar pixels must be to the specified starting pixel in order to be included in the segmentation (see Figure 2.1). As this tool most often finds use on large sections of like color, a user can select a variety of different pixels in semi-uniform sections that would result in the same segmentation. To prevent results like those seen in in Figure 2.1c, painstaking attention must be paid to the tolerance that is specified. In addition to varying the tolerance, anti-aliasing of the edge regions of the segmentation is often an option. The color similarity tool is most useful when images have distinct edges, since with tolerance adjustments, the segmentation can be restricted to include everything up to those edges.

---

[1] According to Schewe (2000), the Magic Wand tool has been included in Photoshop since the 0.87 Alpha release of the product in 1988.



(a) 10% tolerance (b) 30% tolerance (c) 60% tolerance

FIG. 2.1. How varying tolerance affects what is selected for segmentation using the Magic Wand tool, where the same initial pixel was specified for each selection.

2.1.2 Intelligent Scissors / Magnetic Lasso

Another segmentation tool found in many image editing suites is the *Intelligent Scissors* (Mortensen & Barrett 1995) or, as Photoshop has labeled it, the *Magnetic Lasso* tool (Adobe 2008)[2]. It is based on Mortensen & Barrett's (1995) *Live-Wire* boundary snapping. The idea is that a segmentation boundary "wire" or line snaps to object boundaries within a given proximity to the interest point (normally the mouse or stylus position). This approach allows a user to interactively select the most suitable boundary from a set of possible boundaries. The goal is to produce boundaries that are of the lowest local *costs*, where cost is measured by the gradient magnitude of the image. Thus, the live-wire is attracted toward strong edge features. Additionally, the path's cost is determined by the interest point's proximity to previously selected path elements and potential future elements. Section 2.1.4 gives a more detailed explanation of what the term *cost* means in the context of image segmentation.

---

[2] According to Adobe (1998), the Intelligent Scissors tool was incorporated into the Photoshop 5.0 release.

(a) Beginning Selection (b) Continued Selection (c) Final Selection

FIG. 2.2. An example selection using the Magnetic Lasso tool, where the user roughly traced the outline of the figure in the image.

While this method works well for images that have distinct edges, it has trouble when trying to segment highly textured or untextured regions due to the many low-cost paths, which can result in an increase in the amount of necessary user interactions. This can be seen at the bottom of the segmentation in Figure 2.2c, where no distinct edge existed.

2.1.3 Bayesian Matting

The “trimap” approach to image matting and segmentation was introduced by Chuang et al. (2001), based on the alpha estimation work of Ruzon and Tomasi (2000) and prior alpha channel work of Porter and Duff (1984). The concept of a trimap is used to initially distinguish between foreground, background, and unknown regions of an image. There then occurs a hard segmentation over distinctly foreground or background regions, with the unknowns resulting in varying alpha values to represent the degree to which an unknown pixel belongs to either the foreground or the background. This algorithm models color distributions probabilistically, and allows for the creation of high-quality mattes.

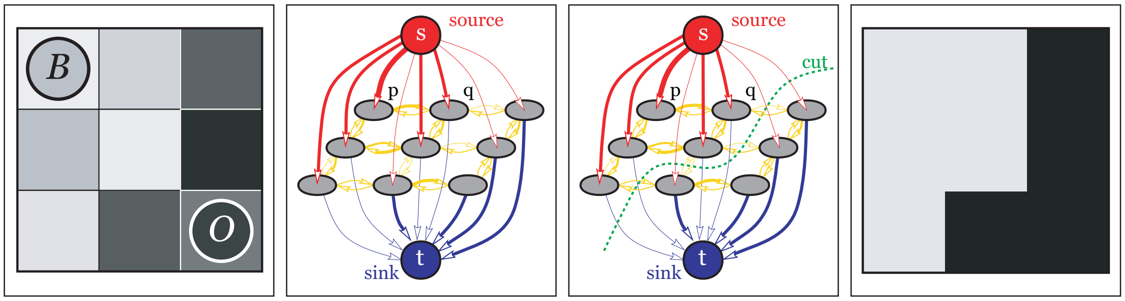
2.1.4 Graph Cuts

A number of image segmentation algorithms treat images as graphs, with pixels being equated to vertices / nodes. The following notation will be used to describe a graph:

$G = (V, E)$	G is a graph; V is a vertex set; E is an edge set
$v \in V$	v is a vertex
$e \in E \subseteq V \times V$	e is an edge
$v_i \xleftrightarrow{e_{i,j}} v_j$	$e_{i,j}$ is the edge between v_i and v_j
$w_{i,j} = w(e_{i,j})$	$w_{i,j}$ is the weight of edge $e_{i,j}$

Graph cutting, a segmentation algorithm that uses image graphs to achieve a desired segmentation, is a building block for a number of other segmentation approaches. The set of all pixels in an image are treated as vertices in V, with two or more additional special vertices called *terminals*, labeled in Figure 2.3b as *source* and *sink*. These terminals correspond to the set of *labels* that can be assigned to pixels. In terms of image segmentation, foreground and background terminals can be defined in order to distinguish a foreground element or object in the image from its surroundings. To allow a user to specify which parts of an image should be a part of the foreground or background, the user is often given the option of specifying *seed* points on the image that correspond to a specific label. These seed points influence surrounding pixels' likelihood of also being associated with the seed's label.

Graph vertices for adjacent image pixels are connected by edges, where every edge in the graph is defined to have some non-negative *weight* or *cost*. The edge weight can be derived from the color or luminance difference between two pixels, but can also take on more complex forms, such as those used by the "imaging objectives" of Agarwala et al.



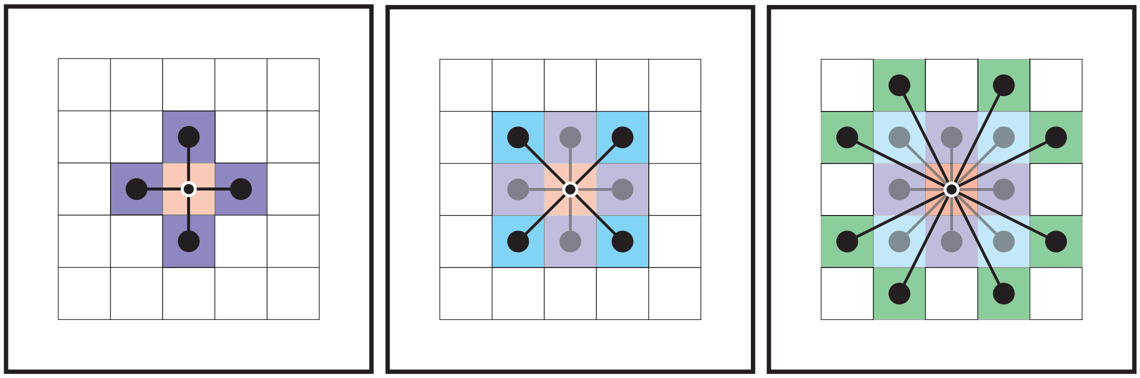
(a) Image with Seeds (b) Graph (c) Cut (d) Segmentation

FIG. 2.3. Example of a directed capacitated graph with edge costs reflected by thickness of the edges(Boykov & Kolmogorov 2003).

(2004), as discussed further in Section 2.1.6. If the graph is undirected, the cost of the edge going from v_i to v_j will equal the cost of the edge going from v_j to v_i. It is often sufficient to calculate and store undirected edge weights, in which case a connection between two pixels can be stored efficiently using a single value. However, in some cases, a directed graph is desirable. In those instances, moving from pixel to pixel in each direction requires its own weight calculation and storage. A case where directed graphs are appropriate is when cost is to be calculated based on gradients, and moving in one direction across the gradient is more desirable than the other. The edge from a dark pixel to a lighter pixel might have a lower cost than the higher cost of an edge from a light to a dark pixel.

It is not necessary for a pixel to only have edges between it and its immediate neighbors. More general cases use some measure of proximity to determine whether or not an edge exists between pixels. Figure 2.4 shows several possibilities for different neighborhood sizes and the resulting structures. Varying neighborhood sizes find use in many parts of the imaging field, from anti-aliasing and filter kernels, to texture synthesis (Efros & Leung 1999; Wei & Levoy 2000; Hertzmann *et al.* 2001).

Once a graph has been created for an image, it is decomposed into n disjoint sets,



(a) 4-Connected Neighborhood (b) 8-Connected Neighborhood (c) 16-Connected Neighborhood

FIG. 2.4. Examples of regular neighborhoods used in 2D image processing (Boykov & Kolmogorov 2003).

where each terminal appears in a different set, along with a subset of the pixels in V. The aim of graph cutting is to minimize the cost of *cutting* edges between nodes. For both directed and undirected weighting, the cost of a *cut* between two pixels is the sum of all of the edges that are severed due to the cut, or more formally:

$$\sum w_{x,y} \qquad \forall e_{x,y} \text{ where } x \neq y; x, y \in \{i, j\}. \tag{2.1}$$

This leads to the *min-cut* methodology, where edge weights are thought of as energy, and the goal is to find the minimum cut that can be made, among all possible cuts that would separate the graph into the n disjoint sets. Another name for this approach is *energy minimization*. The corollary to min-cut is *max-flow*, in which every edge's weight is thought of as having some flow potential. Max-flow methods seek to retain the connections that allow for the greatest flow for each of the disjoint sets. Min-cut and max-flow are equivalent formulations.

The idea of using graph cuts for image segmentation is a relatively new concept, first introduced by Boykov & Jolly (2001). The “Max-Flow” algorithm has found adoption and expansion, with significant improvements being made to the initial Ford & Fulkerson (1956) approach of “Augmenting Paths,” and the “Push-Relabel” implementation by Goldberg-Tarjan (1988). Max-Flow has seen application in the field of texture synthesis in the work of Kwatra et al. (2003), with an increased performance implementation developed by Boykov & Kolmogorov (2002) opening the door for more interactive opportunities in the graph-cutting of images. They were able to reduce multiple-minute computations down to just a few seconds.

Lombaert et al. (2005) presented a novel enhancement to graph cutting called *Multi-level Banded Graph Cuts*, which significantly reduces computation time while decreasing the memory footprint for image segmentations. As can be seen in Figure 2.5, their approach involves *coarsening*, *initial segmentation*, and *uncoarsening* stages. The coarsening stage simplifies the image as well as the seed points. Once a certain coarseness threshold is reached, a standard graph-cutting algorithm is run on the coarsened image to produce an initial segmentation. Uncoarsening translates the initial segmentation to the next level of coarseness (the next level closer to the actual image) and runs a constrained refinement on a band of pixels surrounding the segmentation boundary, adjusting the boundary locally. Uncoarsening is an iterative process that continues until the original image level has been reached.

2.1.5 GrabCut

Rother et al. (2004) presented a system that utilizes graph cuts in a novel segmentation framework. The user is tasked with specifying a rough bounding box around an image object that they wish to segment out of the image. In doing so, the user provides multiple background seeds (the four corners of the bounding box), which enables an initial energy minimization to produce a “hard” binary segmentation of the image. The initial seg-

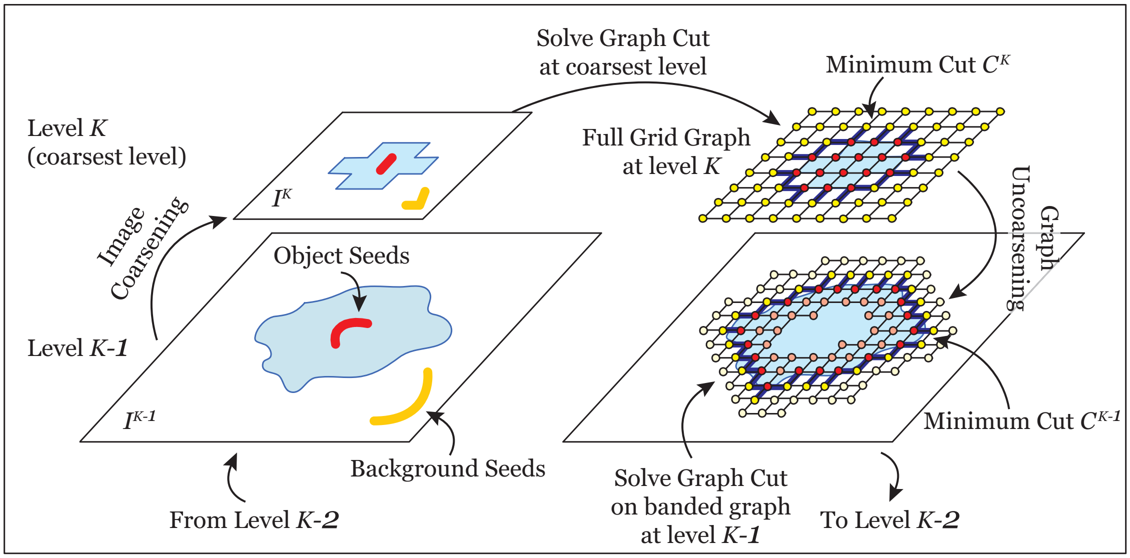


Fig. 2.5. A visual overview of the multilevel banded graph cuts algorithm presented by Lombaert et al. (2005).

mentation is followed by running the edges of the segmentation through a border matting algorithm to determine alpha values (modifying the segmentation from binary (object vs. not object) into a *matte* (object, not object, and partially object)). The partially object pixels are given a calculated color estimate and an alpha value so that when the matte is placed in a different scene, that new scene affects those partially object pixels. The user is then permitted to refine the matte by explicitly specifying foreground or background sections.

2.1.6 Interactive Digital Photomontage

Agrawala et al. (2004) presented a use of the graph-cutting method that enables a single composite image to be made from the preferred parts of other images (see Figure 1.2). They did so through “imaging objectives” such as *designated color*, *designated image*, *minimum / maximum luminance*, *minimum / maximum contrast*, *minimum / maximum likelihood*, or *minimum / maximum difference*. These high-level objectives effectively produce varying energy functions for use in the edge weight calculation. Unlike other image segmentation schemes, edges exist not only between adjacent pixels in a single image, but also between aligned pixels in a stack of images. For a stack of n images, there are potential edges between the corresponding v_i for all n images ($e_i^{0,1}$ between images 0 and 1 ... $e_i^{0,n}$ between images 0 and n), with different weights for each of those edges.

This framework allows for interesting effects such as time-lapsed photography from multiple separate still frame images, where an individual's position melds from a starting position in the first picture to where they end up in each subsequent picture. This process shows motion across all blended images. Additionally, images taken with different depths of field can be merged into a single "extended depth-of-field" image with all objects appearing in focus. Another effect is the ability to take multiple images and use only desired parts of each one (see Figure 1.2), which is useful in situations where multiple pictures of a particular scene exist, in which people or objects occlude part of the view. Using this image objective allows a final composite to be created from the parts of each picture where no occlusion occurs.

Some of these high-level objectives are extremely powerful. However, the time it takes to calculate a result across multiple images is a function of not only how many images are present, but also their sizes and which objective was chosen. Extending the algorithm with *Multilevel Banded Graph Cuts* enabled the images in Figure 1.2 to be generated approximately two to three times faster than the original implementation allows.

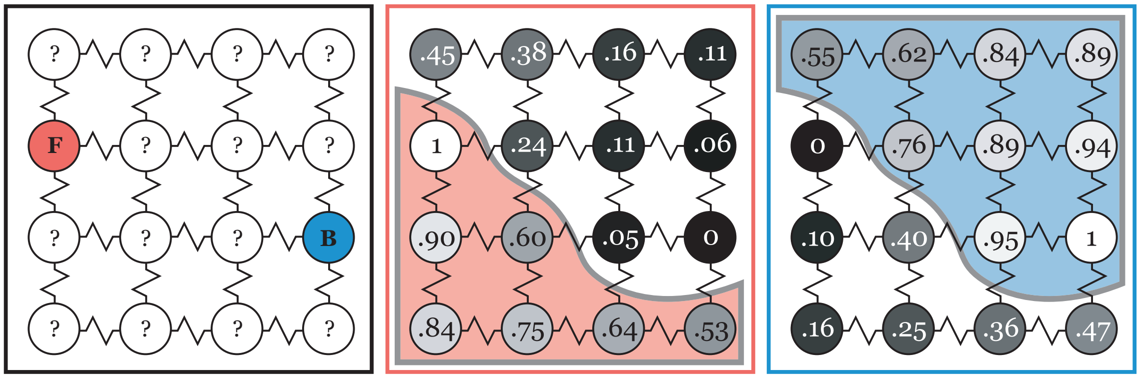
2.2 Random Walkers

The random walkers algorithm represents a different formulation than the aforementioned segmentation approaches. It calculates probabilities for pixel-placed random walkers "walking" across a series of connected pixels, creating paths of connectivity to one of the specified seeds. For the non-seed pixels of an image—pixels that have yet to be associated with a particular label—this algorithm determines the probability that a random walker starting at the given pixel first reaches each of the seed pixels. Seed pixels are defined to belong to a particular labeling. For a given labeling s, the probabilities for all seed pixels of that labeling can be calculated as a single system of equations. The resultant probabilities, x^s for labeling s, are visualized for foreground / background labellings in figures 2.6b & 2.6c respectively. As can be seen from these Figures, each pixel becomes part of the segmentation for the labeling that it has the greatest probability of reaching.

Grady & Funka-Lea (2004) first presented the random walkers algorithm as an approach to the image segmentation problem, allowing for multi-label segmentations. While Figure 2.6 shows a binary segmentation of foreground / background labellings (this thesis takes the binary segmentation approach as well), the random walkers algorithm is capable of handling an arbitrary number of labels. Grady et al. (2005) then presented the algorithm's application to alpha matting. These works, as well as Grady's (2006) outlining of some of the beneficial properties of the random walkers algorithm (the abilities to handle weak boundaries, function even when image noise is present, and deal with ambiguous unseeded regions), have resulted in unique applications of the algorithm, such as the one seen in Section 2.2.3.

2.2.1 Random Walkers Algorithm

While the concept of random walkers has been applied to the 3-dimensional segmentation of volume data (Grady 2006), the focus here is on its application to 2D images. In



(a) Seed Points (b) Foreground Probabilities (c) Background Probabilities

FIG. 2.6. A visual representation of the random walkers input, where seed points are represented by *F* & *B*, with question marks equating to unknown pixels. Possible outputs of the random walkers algorithm can be seen in the next images, which equate to the probability that a random walker starting from each node first reaches the foreground seed, and that a random walker starting from each node first reaches the background seed. The foreground and background segmentations (red and blue shaded regions) are shown for clarity's sake.

this context, each image pixel will be represented by a graph vertex:

$$v_i = pixel_i \qquad \text{i}^{th} \text{ pixel of specified image}$$

$$w_{i,j} = \exp\left(-\frac{\|g_i - g_j\|^2}{\sigma^2}\right) \qquad \text{Gaussian weighting function} \tag{2.2}$$

$$d_i = \sum w_{i,j} \qquad \text{degree; } \forall e_{i,j} \text{ incident on } v_i, \tag{2.3}$$

where g_i is some measurable property of v_i (in the case of images, g_i is generally either color, intensity, texture, or luminance). Note that the value of σ (a weighting parameter) is the only free parameter in this equation. If σ is constant, the σ part of Equation 2.2 can be simplified to a product. Specifically, the weighting calculation can be reduced to:

$$\beta = \frac{1}{\sigma^2} \tag{2.4}$$

$$w_{i,j} = \exp(-\beta \left\| g_i - g_j \right\|^2). \tag{2.5}$$

As outlined by Grady (2006), the random walker probabilities have the same solution as the combinatorial *Dirichlet problem*. He indicated that “the solution to the combinatorial Dirichlet problem on an arbitrary graph is given exactly by the distribution of electric potentials on the nodes of an electrical circuit with resistors representing the inverse of the weights (i.e., the weights represent conductance) and the ‘boundary conditions’ given by voltage sources fixing the electric potential at the ‘boundary nodes.’ ” In Figure 2.6, the jagged lines between nodes represent these resistors / weights. Grady’s use of the term “boundary nodes” (visualized as *F* & *B* in Figure 2.6a) refers to the seed pixels, where the “boundary conditions” are functions that are applied at those pixels (see Equation 2.8).

The purpose of the Dirichlet problem here is to find a function that satisfies this set of boundary conditions for a given image / weighting. In the case of the random walkers algorithm, the Dirichlet problem seeks to find the non-boundary values (unknown pixels’ random walker probabilities) for the Laplacian matrix L. L, which is indexed by vertices v_i and v_j (Dodziuk 1984), is defined as:

$$L_{i,j} = \begin{cases} d_i & \text{if } i = j, \\ -w_{i,j} & \text{if } v_i \text{ and } v_j \text{ are adjacent nodes,} \\ 0 & \text{otherwise.} \end{cases} \tag{2.6}$$

The Laplacian matrix is used as a way to represent the connectivity of a image. Negative weights give the affinity of a pixel to its neighbor. Elements on the diagonal are the accumulated weights for all neighbors connected to a particular pixel. Elements where no pixel connectivity exists are zero. A visual representation of a Laplacian matrix can be seen in Figure 2.7b for a general graph and in Figure 2.7d as it specifically applies to an image graph, where $\sum$ has been used to indicate d_i.

Unlike graph cuts, this algorithm does not employ terminals. Instead, vertices are considered either marked or unmarked, where marked vertices equate to the seed pixels. V is partitioned into V_M (the marked pixels) and V_U (the currently unmarked pixels), such that $V_M \cup V_U = V$ and $V_M \cap V_U = \emptyset$. Note that V_M contains all of the marked seed pixels, regardless of which label they have. L can be reordered to reflect this partitioning:

$$L = \begin{bmatrix} L_M & B \\ B^T & L_U \end{bmatrix}. \tag{2.7}$$

Probabilities need to be found for the unmarked portion of the Laplacian matrix, L_U, since the labellings for the marked seed pixels are already known.

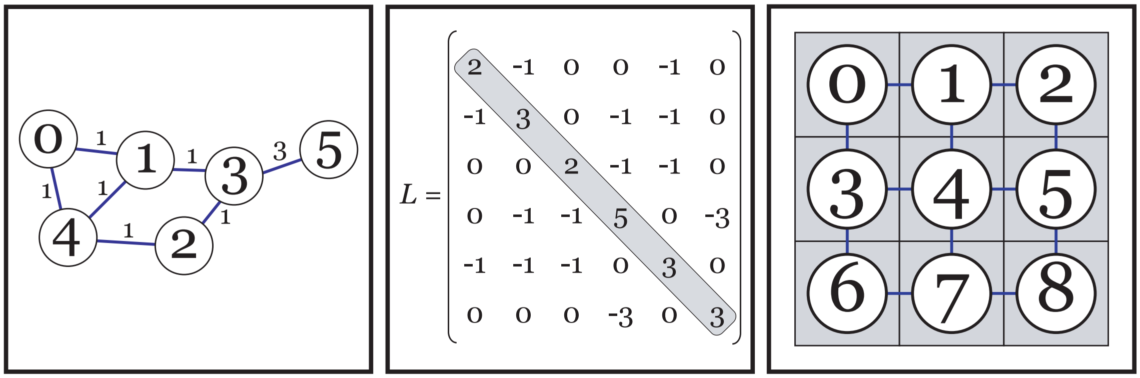
K is defined to be the number of labels, with s equating to a particular labeling. Determining the Dirichlet boundary conditions for seed point v_i can then be defined as a function $Q(v_i)$, $\forall v_i \in V_M$, where $s \in +\mathbb{Z}$, $0 < s \leq K$.

$$m_i^s = \begin{cases} 1 & \text{if } Q(v_i) = s \\ 0 & \text{if } Q(v_i) \neq s \end{cases} \tag{2.8}$$

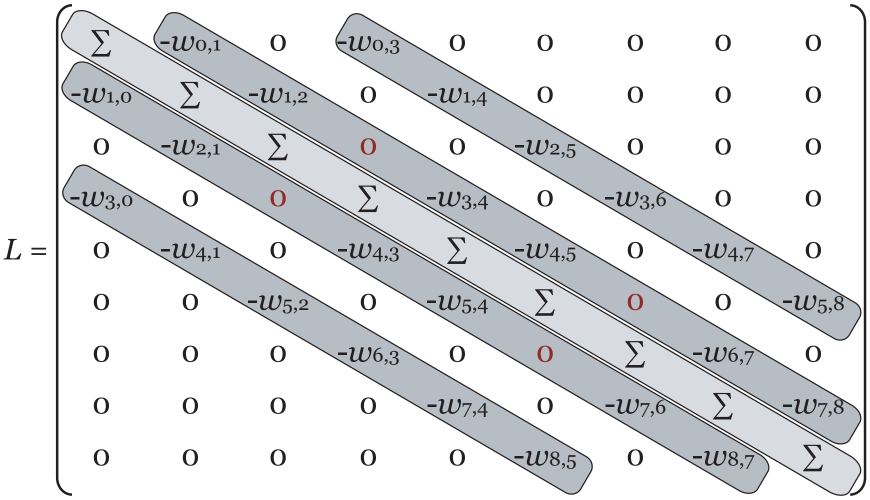
The solution to the combinatorial Dirichlet problem, which equates to the desired random walker probabilities for unknown / unseeded pixels, can be found by solving the large, sparse, symmetric, system of linear equations for each label s:

$$L_U x^s = -Bm^s. \tag{2.9}$$

For a given label s, the probability of pixel v_i being part of labeling s is x_i^s for non-seed



(a) Weighted Undirected Graph (b) Laplacian Matrix for 2.7a (c) 2D Image Graph



(d) 2D Image Laplacian Matrix

FIG. 2.7. The Laplacian matrix for a 2D image employing a 4-connected neighborhood has a very sparse structure, resulting in only 5 diagonals; the main diagonal is the negative row sums of the secondary diagonals. As an example of one of the sums on the main diagonal, $L_{0,0} = d_0 = w_{0,1} + w_{0,3} = -(-w_{0,1} + -w_{0,3}) = -(L_{0,1} + L_{0,3})$ (the negative row sum). Note that Figure 2.7c represents the graph for a 3×3 image, with each gray box representing an image pixel. The zeros shown in red fall within the secondary diagonals, yet due to the structure of an image graph and its boundaries, these indicate the absence of an edge and thus have zero edge weight.

nodes, and m_i^s for seed nodes. The random walkers algorithm as it applies to images can thus be summarized as follows:

1. Calculate the edge weights for adjacent pixels using Equation 2.5
2. Populate the Laplacian matrix with these edge weights, and sum the rows to the main diagonal
3. Determine the labeled (seed) pixels set, V_M, comprised of K labels through user specification or an automated process
4. Solve Equation 2.9 to attain the random walker probabilities x^s for each label s
5. Determine the segmentation by labeling each unknown pixel, v_i, with the label that corresponds to $max_s(x_i^s)$

Grady (2006) indicated that should interactive segmentation be desired, starting at the 3^{rd} step above, it would be possible with changes to the seed pixel set to "use the previous solution [x^s] as the starting point for an iterative matrix solver for the new system (Equation 2.9)." To allow interactivity, GPURW implements this approach.

2.2.2 Iterative Solvers

Solving linear systems of equations of the form $Ax = b$, as seen in Equation 2.9, can be accomplished directly using methods such as *LU* decomposition (Golub & Van Loan 1996). However, doing so comes with a significant memory requirement when working on large systems that can quickly exceed the computational power of today's commodity hardware. Alternative iterative methods provide a means for solving these systems that reduces the required memory while allowing for constrained computation times (at the cost of accuracy) (Barrett *et al.* 1994). As an added benefit, the operations required by the iterative methods (data parallel sparse matrix-vector multiplication, and reduction for inner products) can be readily parallelized (Bolz *et al.* 2003; Krüger & Westermann 2003).

The generalized form for iterative solvers requires the matrix A, a starting value vector x_0, the right-hand-side vector b, a maximum number of iterations i_{max}, and an error tolerance $\tau < 1$. Iterative solvers loop through a number of iterations, calculating a refined result per iteration. The intent is for each iteration's result to come closer to the actual solution than the previous iteration, with convergence (success) being achieved when the difference between the result of adjacent iterations is less than τ. Iteration results may fluctuate or diverge, depending on the system to be solved. However, the random walkers algorithm has been proven to converge.

One of the best known iterative solvers, the *Jacobi* method (Jacobi 1846), determines a result for iteration i by decomposing A into its diagonal D, its strict upper quadrant U, and its strict lower quadrant L:

$$x_i = D^{-1}(L + U)x_{i-1} + D^{-1}b. \tag{2.10}$$

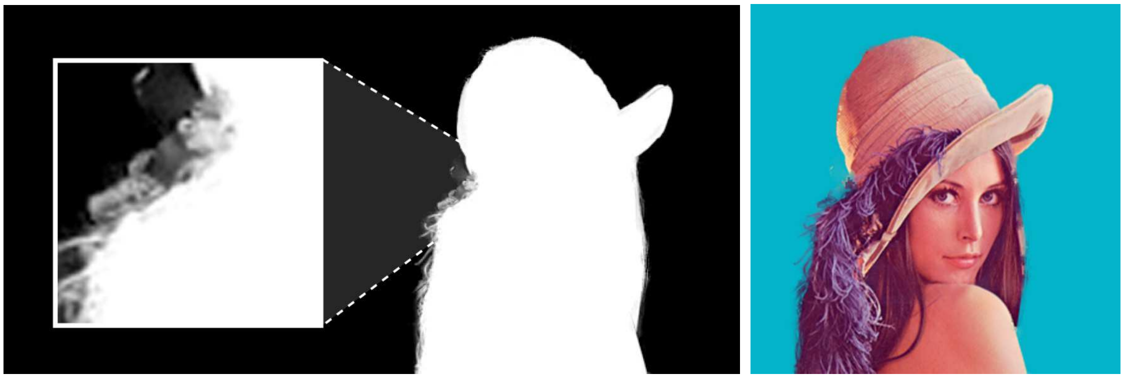
This method provides a stationary means of solving these systems that requires few calculations per iteration. As this is a fairly simplistic method, not much information is used in calculating each successive iteration other than the results of the previous iteration. Thus, convergence for many classes of problems is slow. As an alternative, the *CG* (conjugate gradient) method (Hestenes & Stiefel 1952) offers a non-stationary option with additional flexibility in terms of solving different classes of problems. Non-stationary methods present information at each iteration to be used in helping the calculations converge more quickly, with CG allowing for the inclusion of a preconditioner P (see Listing 2.1). The simplest such preconditioner is again attributed to Jacobi, in which P equates to the diagonal D.

```
$r_0 = b - Ax_0$                        //Initial Residual
$d_0 = P^{-1} r_0$                      //Initial Preconditioned Search Directions
$\delta_0 = r^T d_0$
for $(i = 1$ to $i_{max}) \{$
    $q = Ad_{i-1}$
    $\alpha = \delta_{i-1} / d_{i-1}^T q$
    $x_i = x_{i-1} + \alpha d_{i-1}$    //Iteration $i$ Result
    $r_i = r_{i-1} - \alpha q$          //Residuals
    $s = P^{-1} r_i$                    //Preconditioning
    $\delta_i = r^T s$
    if $(\delta_i > \tau^2 \delta_0) \{$
        return                          //Convergence
    $\}$
    $\beta = \delta_i / \delta_{i-1}$
    $d_i = r_i + \beta d_{i-1}$         //Search Directions
$\}$
```

Listing 2.1. Preconditioned conjugate gradient method of Buatois et al. (to appear).

2.2.3 Soft Scissors

As the name implies, *Soft Scissors* (Wang, Agrawala, & Cohen 2007) inherits from the *Intelligent Scissors* algorithm discussed in Section 2.1.2, in that it is also a *boundary tracing* segmenter. Unlike the *Live-Wire* boundary snapping of its predecessor, which results in binary segmentations of foreground and background, this algorithm produces high-quality mattes for objects that could encompass fur and hair (notoriously difficult elements for segmentation). Another unique element of Soft Scissors is that it confines computations to very narrow update regions through what the authors call “incremental matte estimation.” Instead of recomputing values for the entire image at each request for an update, as is the case in graph cut-based algorithms, only the minimum set of pixels have their color and alpha values updated. This reduces the amount of computation necessary, and allows for an interactive system that gives near immediate feedback to the user.



(a) Matte (b) Extracted Image

FIG. 2.8. An extract using Power Mask ©Digital Film Tools, which implements the Soft Scissors algorithm. Note that gray values in the matte indicate varying percentages of alpha transparency. The blue background in the extracted image is there for contrast purposes.

Soft Scissors additionally employs the random walkers (Grady 2006) algorithm for its alpha and color determinations at the matte boundaries, inheriting the benefits—such as noise robustness and the ability to handle weak boundaries—provided by the random walkers algorithm. When calculating color values, weighting calculations of neighborhood size four are used with the random walkers algorithm to determine a desired color value. However, when calculating alpha values, Wang et al. choose to use a neighborhood size of 25 (presumably a 5×5 neighborhood, with the center being the pixel itself). It would seem that this works in favor of ensuring soft edges, as the title of the algorithm implies.

2.3 Graphics Hardware

With their work in programmable graphics architectures for use with procedural shading, Olano and Lastra (1998) made other real-time approaches to the shading problem possible (Peercy *et al.* 2000; Proudfoot *et al.* 2001), but more notably showed what programmability within the GPU could mean. With fixed function units being replaced by a fully programmable pipeline, programmers can do more than merely create textured polygons. Through interactions with vertex / fragment / geometry shader hardware, and floating point support[3], it is possible to achieve interactive effects with graphics hardware that would have previously been inconceivable. Such effects include real-time lighting, dynamic soft shadows, volumetric / layer / view distance fogging, screen space ambient occlusion, subsurface scattering, motion blur and depth of field, all found in the PC video game *Crysis* (Crytek 2008).

Graphics is an “embarrassingly parallel” field, in that image pixel values can often be calculated independently of each other. To handle the workload that this property presents, today’s GPUs are highly parallel processors. With many sub-processors within the main processor core, they provide efficient communication for the processing of geometry and texture data on the graphics card. Though the data that resides on the GPU generally pertains to graphical information such as vertices, color, and lighting, ultimately the GPU uses integer or floating point data in the mathematically intensive computations that prevail in the field of graphics.

2.3.1 GPGPU Computing

In the recent past, there has been an insurgence of the GPU being used on non-graphics information; this phenomenon is known as *GPGPU* (General Purpose Graphics Processing Unit) computing. Thanks to the programmable pipeline and the ability to interact with it through standard APIs, it is now possible to run iterative solvers and other linear algebra operations on the GPU in parallel (Krüger & Westermann 2003; Bolz *et al.* 2003). NVIDIA’s *CUDA* (NVIDIA 2008a), ATI’s *CTM* (Close to Metal)—

---

[3] Beyond single-precision, double-precision IEEE floating point support has recently become available in NVIDIA’s newest GT200 series (NVIDIA 2008b).

which paved the way for AMD's *Stream* (AMD 2008a)[4]—and Apple's *OpenCL* (Open Computing Language) initiative (Munshi 2008) represent vendors' attempts to integrate GPGPU functionality with particular hardware. NVIDIA's (formerly AGEIA's) PhysX (NVIDIA 2008c) real-time physics platform is able to interact with a number of different graphics hardware platforms, from the XBox 360, to the Playstation 3's Cell processors, and now with NVIDIA's GeForce 8 or greater lines of chips using CUDA.

In GPGPU computing, problems are decomposed into a group of operations called a compute *kernel*. Operations might consist of basic math, memory / texture accesses, and function calls. The kernel and its contained operations are executed in parallel on a number of threads within the GPU, with each thread being able to process different sets of data, yet using the same operations as all other threads in the corresponding kernel. Consider the simple problem of adding two vectors of length n, a and b, together and storing them back into vector r. A compute kernel to perform this task could spawn n threads, and use each thread's id i to index into the vectors: $r[i] = a[i] + b[i]$. Thus, thread i could calculate the i^{th} element of the resultant vector. Since GPUs are highly optimized for data-parallel or SIMD (single instruction multiple data) problems such as this one, kernels need to be designed that allow for this distributed processing to take full advantage of the processing power available. Careful planning and attention are required to ensure that problems are decomposed into kernels that best utilize the GPU's power.

2.3.2 CUDA

NVIDIA's *CUDA* is a set of APIs for the C programming language. It adds a number of extensions to C for specifying where particular code should run, either on a CUDA-enabled *device* or on the *host* platform (generally the CPU). Additionally, there are exten-

---

[4] Stream uses an extended version of the Brook (Buck *et al.* 2008) compiler.

sions for specifying the number of threads that should concurrently be processing a particular compute kernel. It is the responsibility of the kernel to take advantage of the available parallelism, though doing so often only requires changing the way that a loop iterator is defined.

CUDA abstracts away the notions of running on the GPU by dealing with threads rather than pixels, and device / host memory rather than textures. Once information has been transferred from the host to device memory, outputs of one device function can be used as inputs to the next. Thus, a number of operations or kernels can be run in sequence without transferring any data back to host memory. The arbitrary addressing of memory, or *scatter* / *gather* operations, are also permitted. This allows device memory to be accessed just as if it were host memory, although the accessing can only be done by device kernels. For a host function to access device memory, that memory must first be copied back from the device to the host. Once a kernel has been developed, it is compiled into byte-code for use by a CUDA-enabled device. The compiled device code can be linked to separately compiled host code so that interoperability between the two can take place.

The extensions to C provided by CUDA have enabled a vast array of non-graphics applications to leverage the GPU's processing power, from financial calculations and Monte-Carlo option pricing, to Mersenne Twister random number generators. Yet some of the most profound contributions of CUDA come as built-in functionality from NVIDIA's provided CUBLAS (CUDA implementation of the Basic Linear Algebra Subprograms) and CUFFT (CUDA implementation of the Fast Fourier Transforms) libraries (NVIDIA 2007). Both allow for applications to be developed without requiring the developer to implement these building blocks for the GPU.

While a number of iterative solvers have been developed for the GPU (Krüger & Westermann 2003; Bolz *et al.* 2003), none to date have been developed for one of the major GPGPU frameworks such as CUDA (excepting the *Concurrent Number Cruncher* work of Buatois et al. (to appear)). While Sengupta et al. (2007) provided sparse matrix-vector multiplication for interoperability with CUDA, and the CUBLAS library provided the remaining parts needed for iterative solvers to be created, the two have not previously been combined.

Chapter 3

APPROACH

Like most graphics applications, GPURW follows an **Initialize** → **Display Loop** approach. The **Display Loop** method allows for intermittent **Input** and **Update** calls, as can be seen in the structural overview depicted in Figure 3.1.

The user loads an image to initialize the system. The image's width and height determine the allocation of host and device memory. The image's pixel color data is transferred to the GPU's texture memory. Constant index, offset, and access arrays are created to define the sparse structure of the Laplacian matrix that will later be used by the random walkers algorithm. These allow for reduced calculation at run-time in the determination of weight locations and sums in the Laplacian matrix.

Once initialized, the main **Display Loop**'s **Output** section begins processing and rendering an orthographic view of the loaded image overlaid by the current segmentation mask. **Input** methods allow the label seeding to be manipulated: seeds can be added, removed, or modified by changing their label. Additionally, display modes can be switched to enable the viewing of probabilities, seeds, or other pertinent information, as opposed to the default view of the current mask. Finally, **Input** allows for manipulating certain parameters of the system (such as tolerances or thresholds) and refining the current mask.

In the event that the seeds / parameters are changed, or mask refinement is requested,

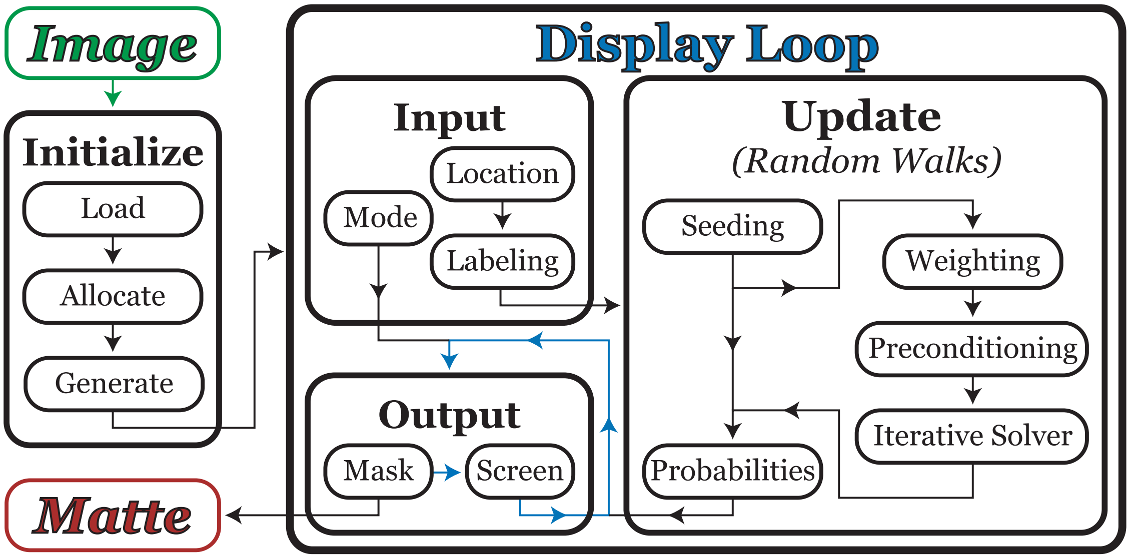


FIG. 3.1. An overview of the GPURW method.

Input signals to **Update** that the random walkers algorithm needs to re-compute the pixel probabilities based on new information. Since **Update** is the component of the algorithm that requires the most computation, and is also the bottleneck of the entire process, it is only called as necessary, to minimize latency in the user experience. **Update** starts by calculating weights for the connectivity graph of the image. These weights are propagated into the Laplacian matrix with the help of the access arrays defined during the **Initialize** stage (see the bottom five rows in Figure 3.2). Additionally, these weights are used in the summation that appears on the main diagonal of the Laplacian matrix. Once the matrix is filled, preconditioning for the random walker's iterative solver occurs. The iterative solver is executed for the Laplacian matrix, stopping once a convergence threshold reaches the seed-determined tolerance discussed in Section 3.3.1.

Since the results of the random walkers algorithm are approximate probabilities of a specific unmarked pixel having a particular labeling, the probabilities for each labeling need to be compared to determine which labeling has the greatest probability for each pixel. The segmentation mask is then determined based on this information.

Output updates the user's monitor using OpenGL routines to display the image overlaid by the current segmentation mask or other requested overlay. If a different display mode was requested, the information pertaining to that mode may instead be shown. Additionally, **Output** has the responsibility of returning the segmentation mask as an image matte once a satisfactory segmentation is achieved.

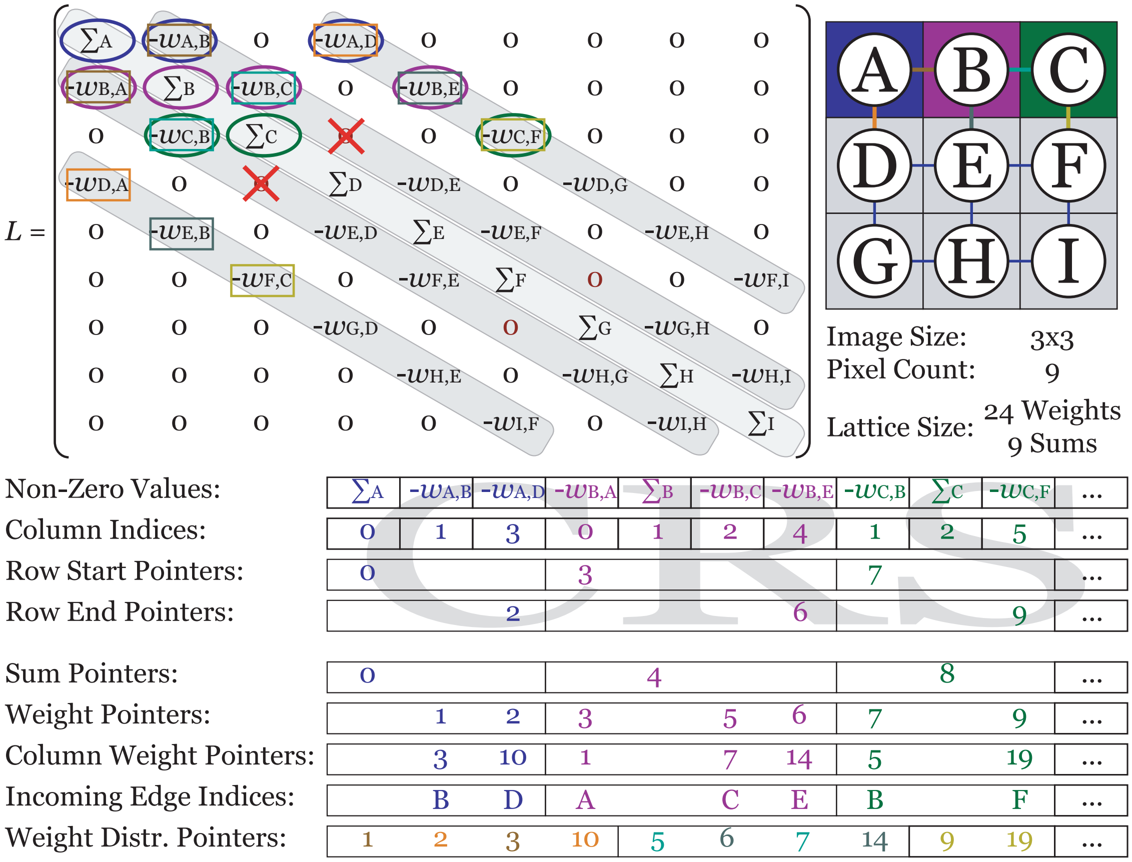
3.1 Initialization

After loading an image into GPURW, the pixel information that the image contains is transfered to device memory. OpenGL pixel and luminance buffers are allocated that will later be used to transfer data on the device from CUDA's memory space to OpenGL's memory space without leaving the GPU. Use of the pixel buffer enables the display of base color data for the image for the computation of overlay information identifying which label a given region of the segmentation belongs to. The luminance buffer allows for computed probabilities to be displayed. Since the probabilities occur in the range from 0 to 1, displaying them as luminance prevents the need for first converting them to RGB color values and then displaying three color components rather than displaying the single gray-scale luminance component. Once the buffers have been created, they are associated with corresponding textures. All image and probability information is drawn as a screen-aligned rectangle. Binding the buffers to a texture allows the rectangle to simply be textured with the calculated values and overlays in order for results to be displayed to the screen.

After OpenGL initialization, storage and structures necessary for the random walkers process are initialized. The allocation of $temporary$, x, b, and c vectors takes place. This is followed by the generation of the sparse matrix representation for A that will be needed for iteratively solving $Ax = b$ on the Dirichlet problem. Since CUDA allows for 1D, 2D and 3D device arrays, creating the vectors as 1D arrays is a direct mapping. CUDA provides a number of functions on 1D arrays, such as calculating the L2-norm and efficiently calculating $\alpha x + y$ for arrays x and y and the scalar α; these are all necessary for iterative solvers. Implementing the vector-vector multiplication that is also needed for calculating the preconditioner term, P, when preconditioning is used with CG equates to a simplistic kernel, similar to the one described in Section 2.3.1.

For the matrix representation of A, the translation to CUDA is much more complicated. With problems that have densely packed matrices, a 2D device array could have been used as the matrix representation. However, the Laplacian matrix for a 2D image is extremely sparse, with only a few non-zero elements per row. For a 4-connected neighborhood, only a pixel's immediate neighbor's weights will appear in the secondary diagonals, resulting in a maximum of five non-zero elements per row (four neighbors and a sum). Since a pixel may only have two or three neighbors (image corners and edges respectively), there are entries in the secondary diagonals that will equate to zero as well (represented by the red zeros in Figure 3.2). Excluding these from the Laplacian representation has the potential of avoiding the computation of $2w * 2h$ floating-point zero values.

Instead of representing the matrix as a standard 2D CUDA array, the *CUDPP* (Sengupta *et al.* 2007) sparse-matrix extension to CUDA is used to allow for a greatly condensed version of the matrix A. It uses a sparsity structure known as *CRS* (compressed row storage). CRS includes a vector of column indices for all non-zero entries in the Laplacian, a vector of pointers into the column indices vector that indicate where a new row begins, and a second vector of pointers to show where each row ends. In addition to those vectors needed by CUDPP for the CRS representation, GPURW creates convenience vectors for sum pointers (where sums should be stored into the CRS Laplacian once they get calculated), weight pointers (those weights that contribute to a particular sum), edge pixel



$$L = \begin{pmatrix} \sum A & -w_{A,B} & 0 & -w_{A,D} & 0 & 0 & 0 & 0 & 0 \\ -w_{B,A} & \sum B & -w_{B,C} & 0 & -w_{B,E} & 0 & 0 & 0 & 0 \\ 0 & -w_{C,B} & \sum C & \cancel{0} & 0 & -w_{C,F} & 0 & 0 & 0 \\ -w_{D,A} & 0 & \cancel{0} & \sum D & -w_{D,E} & 0 & -w_{D,G} & 0 & 0 \\ 0 & -w_{E,B} & 0 & -w_{E,D} & \sum E & -w_{E,F} & 0 & -w_{E,H} & 0 \\ 0 & 0 & -w_{F,C} & 0 & -w_{F,E} & \sum F & 0 & 0 & -w_{F,I} \\ 0 & 0 & 0 & -w_{G,D} & 0 & 0 & \sum G & -w_{G,H} & 0 \\ 0 & 0 & 0 & 0 & -w_{H,E} & 0 & -w_{H,G} & \sum H & -w_{H,I} \\ 0 & 0 & 0 & 0 & 0 & -w_{I,F} & 0 & -w_{I,H} & \sum I \end{pmatrix}$$

Non-Zero Values:	$\sum A$	$-w_{A,B}$	$-w_{A,D}$	$-w_{B,A}$	$\sum B$	$-w_{B,C}$	$-w_{B,E}$	$-w_{C,B}$	$\sum C$	$-w_{C,F}$	...
Column Indices:	0	1	3	0	1	2	4	1	2	5	...
Row Start Pointers:	0			3				7			...
Row End Pointers:			2				6			9	...
Sum Pointers:	0				4				8		...
Weight Pointers:		1	2	3		5	6	7		9	...
Column Weight Pointers:		3	10	1		7	14	5		19	...
Incoming Edge Indices:		B	D	A		C	E	B		F	...
Weight Distr. Pointers:	1	2	3	10	5	6	7	14	9	19	...

FIG. 3.2. Compressed row storage for an image-based Laplacian matrix, with additional convenience vectors for on-the-fly kernel-based calculation of the Laplacian. Note that letters are used in place of zero-based indices for clarity's sake. Since the graph is undirected, the Laplacian is symmetric, with corresponding indices represented by like-colored boxes in L and like-colored lines in the image graph.

indices (which edges are affected if a seed gets placed in this pixel), and all weight pointers (weights are calculated from upper left to lower right in the image, eliminating redundant calculations of weights). These stored values can also be seen in Figure 3.2. Although these convenience vectors entail additional storage requirements, they permit a number of serial calculations to be saved in determining the sparsity structure for CRS that would otherwise bottleneck the pipeline. Initialization does not populate the non-zero entries of the Laplacian; rather, it builds the structures for housing them, determines how to calculate them, and indicates where to place them once they have been calculated.

Without the work done by Sengupta et al. (2007), who provided a pre-release version of *CUDPP* (the implementation of their Scan Primitives, which included sparse matrix-vector multiplication functionality), the work presented in this thesis would have been much more difficult. Although other sparse matrix-vector multiplication routines existed for the GPU (Krüger & Westermann 2003; Bolz *et al.* 2003), they leveraged shader code instead of providing the interoperability with CUDA that was required for GPURW. While the CUDPP sparse matrix-vector multiplication implementation provides the ability to change the incoming vector, it required a fixed matrix. Once this fixed matrix had been specified, changing the values within the matrix was not an option. For the GPURW algorithm, the Laplacian matrix is based on the currently specified seed points and changes with every manipulation to those seeds. Therefore, extensions to CUDPP are required to get a direct reference to the underlying CUDA storage where the matrix values are stored. So that the actual sparse matrix need not change and the sparsity structure does not have to be derived at every seed change, the same structure that gets initially allocated is reused at each iteration.

3.2 Input

In order to visualize the different outputs and stages, the GPURW process allows for a number of display modes to view the image alone, labels, foreground / background probabilities, and the current segmentation. To permit these to be seen better on a variety of image types (gray scale, low contrast), multiple overlay modes enable the segmentation to be viewed as different colors or as the image and its inverse. Both the display and overlay modes are made accessible by the input stage, and can be toggled using shortcut keys. Additionally, shortcut keys have been specified for performing user-requested refinements to the random walkers output. A user-requested refinement modifies the iterative solver threshold and triggers a recalculation of the probabilities, resulting in a more accurate result at the expense of additional processing time. For comparison purposes, switching between different iterative solvers is also an option.

The main focus of the input stage is the user interactions to specify the seeds' labels and locations. The system provides a seed painting brush to facilitate adding, changing and removing seeds from V_M. This brush acts like those found in standard painting packages: painting begins when the mouse is clicked, follows the brush as the user moves the mouse, and ends when the user releases the mouse button. However, unlike those brushes, color or paths based on the brush are not applied. Instead, every pixel under a painted region is marked as a seed point with its labeling coming from keyboard modifiers (CTRL, SHIFT) that are pressed during the application of that brush stroke. The keyboard modifiers allow for switching between *foreground*, *background* and *none* so as to specify which labeling should be applied. Seed points can thus be changed from one labeling to another by simply switching the keyboard modifiers and repainting a desired region. Additionally, if a region is unknown (i.e., not foreground or background), repainting it with the *none* labeling removes any seed points that had previously been painted there. Since it may be desirable to indicate a large number of pixels as all being seeds of the same label, the seed painting brush's size can be adjusted, allowing more seeds to be specified per brush stroke.

3.3 Update

When the user changes the marked seed pixels, the results must be refined towards the user's desired segmentation. Since a modification to the seed pixels changes the arrangement of the marked and unmarked (L_M and L_U respectively) Laplacian entries of Equation 2.7, recalculation of the values for L each time an update occurs is necessary. Updating translates the strokes from the seed painting brush into a seeding. It then takes the initial image's pixel values and by applying Equation 2.2, dictates the weighting and the ultimate content of L. Preconditioning then occurs when appropriate. This is followed by the solving of the combinatorial Dirichlet problem using an iterative solver. Finally, the probability results from the iterative solver are translated, along with the seeding, into the final probability values. Further update passes are initialized with the prior random walker solution to allow for faster convergence as well as better interactive response.

3.3.1 Seeding

Iterative solvers converge if and when calculations reach a steady state where changes from one iteration to the next are below a specified tolerance τ. Additionally, iterative solver calculations can be stopped if a maximum number of iterations have taken place. This is useful when convergence is taking too long, and prevents cases where convergence will not occur from continuing to iterate indefinitely. For problems that do converge (such as the random walkers algorithm), τ acts as a flexible limiter that provides a relatively uniform result from one invocation to the next. Thus, GPURW calculates tolerances in reference to random walkers as:

$$\tau = \frac{|V| -}{|V|} \left(\tau_{max} - \tau_{min}\right) + \tau_{min} \tag{3.1}$$

where $|\cdot|$ indicates cardinality, and τ_{max} and τ_{min} are free parameters. For few seeds ($|L_M| \cong 0$), $\tau = \tau_{max}$. As the number of seeds approaches the number of pixels ($|L_M| \cong |V|$), $\tau = \tau_{min}$. For the purposes of this paper, $\tau_{max} = 0.4$ and $\tau_{min} = 0.1$ were found to allow results that were interactive and still produced high-quality segmentations as more seeds were specified.

Using this definition of τ, the random walkers algorithm generates a smooth accumulation of accuracy as seed points are added. As the number of seeds increases towards $|V|$, the tolerance decreases, producing results that are closer to the optimal solution found through *LU* decomposition. While having less accurate results to begin with is not as desirable as producing the optimal solution, the trade-off in seeking greater accuracy up front is slower convergence and, as a result, a system that is not interactive. Thus, the user must wait to refine their results. GPURW takes the approach that as more is known about a user's desired segmentation (more seed pixels equates to less unknown pixels), the more accurate it should strive to be.

3.3.2 Weighting / Laplacian Filling

The entire weighting pipeline is implemented as a set of GPU *kernels*, or parallel processing passes. The g_i values used in the calculation of weights for the Laplacian come from a CIE L*a*b* color conversion of the image, and use a β value of 90 for the free parameter in calculating Equation 2.5. While RGB values could have been used directly, a slight benefit was seen from first converting to CIE L*a*b*, since perceptual color spaces such as CIE L*a*b* provide a more direct mapping to color distinctions that humans perceive. The weights that are propagated into the sparse Laplacian matrix are recomputed at each iteration through the **Update** process. This recomputation would allow future extensions to support varying weights. As weights are calculated, their negative values are *scattered* into the Laplacian matrix L as the secondary diagonals. The color conversion, weight calculations, and prorogation all take place in a single device kernel. Note that no sums have been calculated at this point.

Solving the combinatorial Dirichlet problem ($L_U x^s = -Bm^s$) provides the means for deriving the updated random walker probabilities for labeling s. Recall that iterative solvers are designed to solve systems of the form $Ax = b$. Thus, to find the random walker probabilities, the Laplacian and seeding information must be converted into a representation that is comprised of A, x and b parts. GPURW starts with the task of finding:

$$b = -Bm^s \tag{3.2}$$

for the right-hand side of the $Ax = b$ equation. A second kernel derives this b vector based on the weights found in the B-portion of L (see Equation 2.7) that were calculated in the first kernel. The b vector is initialized to zeros, with seed points m^s (represented by the incoming edge indices vector in conjunction with V_M) and the corresponding column weight pointers vector that maps to B, being used to fill out b. Since GPURW allows for two labellings, one for foreground and one for background, two separate b vectors are produced based on the seeds, m^s, marked as such.

A third kernel is used to *gather* the weights in order to create the sums (using the weight and sum pointers respectively), and to place them on the main diagonal. At the same time, the inverse of the main diagonal is calculated as it is used by both the fixed-point Jacobi and Jacobi preconditioned CG solvers. The fixed-point Jacobi solver requires $L_U = D^{-1}(M - D)$ for some matrix M, and the diagonal of that matrix D. As the diagonal inverse D^{-1} has just been calculated, it is used to transform the Laplacian into L_U by multiplying D^{-1} by the weights. Since ultimately L_U will have zeros on the main diagonal for the fixed-point Jacobi, this is accounted for at allocation time, with the main diagonal being excluded from the CRS representation for this type of solver.

Since the structure of the matrix used by CUDPP for sparse matrix-vector multiplications does not change after the **Initialization** stage, the row and column entries that would have been removed (where seeds exist) are instead set to zeros in the Laplacian matrix. For the matrix operations required by CUDPP, this has the same affect as actually removing them from the structure. Though this is not the ideal solution (calculations will still be carried out on these zero values), it circumvents the need to continually reallocate the matrix. Thus, L_U is the result of zeroing the row, column, and sum values in L for rows / columns where the sum corresponds to a seed point.

3.3.3 Preconditioning

Preconditioning utilizes the fact that iterative solvers allow for the use of prior knowledge, or an initialized x_0 vector. Finding the optimal x_0 is difficult, yet x^s from the previous results provides an excellent source for this information. The result of one **Update** pass, or x^s, is used as x_0 in the next **Update** pass for label s. The vector x^s represents probabilities for a prior pass based on the specified seeds at that point in time. Since the seeds that are added, removed, or changed from one **Update** cycle to the next are generally in a close proximity to those that existed previously, the resultant x^s for subsequent **Update** cycles are usually fairly close to those of the previous cycle. Thus, GPURW iteratively (with each brush stroke) produces results that account for user feedback with a relatively short computation delay between refinements.

3.3.4 GPU Iterative Solver

With an initialized L_U matrix, b vectors from the weighting / Laplacian filling stage, and the preconditioned x_0 vector, the system must now be solved in order to determine the random walker solution. The fixed-point Jacobi method (see Equation 2.10) is the simplest iterative solver with storage beyond the matrix itself being limited to the inverse of the matrix diagonal. Fixed-point Jacobi provides slower convergence than alternative solvers and is rarely used in practice (Moin 2001), but in testing, it provided interactive and accurate segmentations. Recall that for the fixed-point Jacobi implementation, when calculating L_U, the diagonal is not stored as part of L, with this representation being multiplied by the inverse of the diagonal. Doing this gives P, as seen in Listing 3.1. Beyond this upfront computation of the sparse matrix, the only other calculations involved in this solver are the vector-vector multiplication required to calculate c, and the computation-intensive matrix-vector multiplication seen in calculating Px. Fortunately, CUDPP provides a GPU optimization for the non-trivial sparse matrix-vector multiplication of Px. The CUDPP CRS matrix representation and sparse matrix-vector operations, along with the CUBLAS GPU optimizations for determining an L2-norm *cublasSnrm2* and for adding two vectors together *cublasSaxpy*, provide the necessary building blocks for implementing iterative solvers on the GPU.

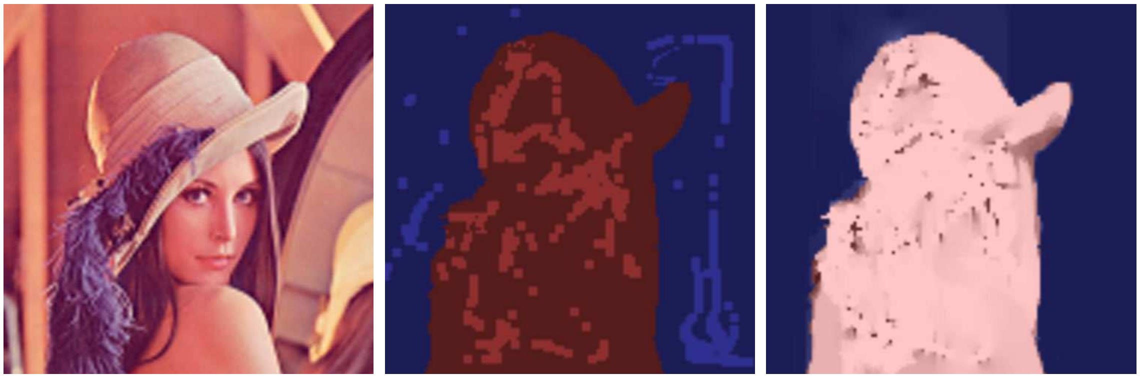
While pairing different preconditioners with the conjugate gradient method or choosing alternative storage formats (other than CRS) might yield better results, they would come at the cost of more storage and increased complexity in the solvers' iteration loop (Barrett *et al.* 1994). For comparison, a Jacobi preconditioned CG solver as seen in Listing 2.1 has been implemented, where P simply equals to the matrix diagonal. Additionally, an unpreconditioned CG solver has been included, which equates to setting $d_0 = r_0$ and $s = r_i$ in that same Listing.

```
$D$ = **diagonal**($A$)          //Sparse Diagonal Matrix
$ID$ = **inverse**($D$)
$P = -ID * (A - D)$
$c = ID * b$                     //Vector-Vector Multiplication
$\delta_0 = |b|$                 //L2-norm
**for** ($i = 1$ **to** $i_{max}$) {
    $\delta = \delta_0 / |x|$
    **if** ($\delta < \tau$) {
        **return**
    }
    $x = c + Px$                 //Matrix-Vector Multiplication
}
```

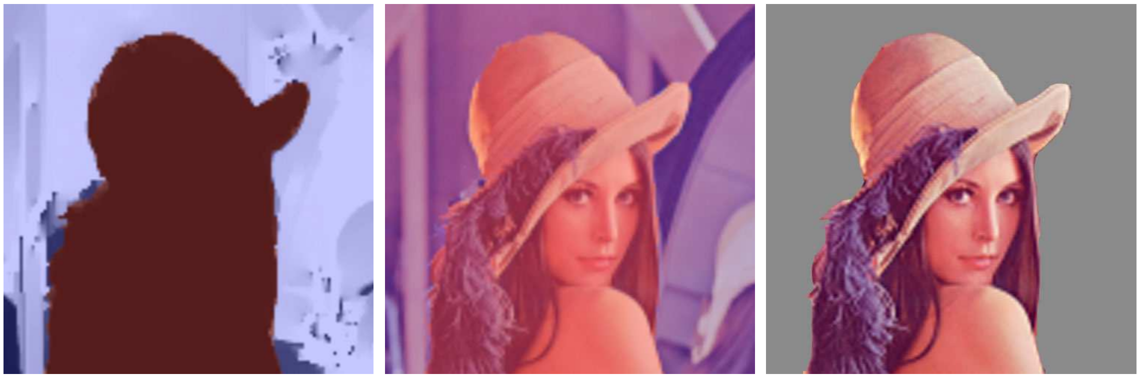
Listing 3.1. Fixed-Point Jacobi Solver pseudo-code based on Matthews (2004).

3.3.5 Probabilities

With the standard random walkers algorithm, it is sufficient to terminate computation once $K - 1$ of the x^s systems have converged (GPURW has $K = 2$ different labels for foreground and background). At that point, determining which labeling has the greatest probability at each pixel dictates which labeling that pixel belongs to. With two labellings, this could be simplified to computations in only a single system. If the pixel's probability were greater than 50% for one of the two labelings, that pixel would be associated with that higher probability labeling. However, the systems in GPURW do not generally reach the same level of convergence that would allow for this to happen. Instead, the calculations end when the seed determined tolerance of Equation 3.1 is met, so the probabilities of the two labelings do not necessarily sum to one. Thus, it becomes necessary to run calculations against both systems and determine, for each pixel, which system (foreground or background) has a greater probability in that labeling's x^s vector. Since all seed pixels equate to boundary conditions, and were thus removed from the combinatorial Dirichlet



(a) Unsegmented Image (b) Seed Points (c) Foreground Probabilities



(d) Background Probabilities (e) Segmentation (f) Extracted Image

FIG. 3.3. Different outputs of the Random Walkers process.

problem, the seed pixels are merged with their respective labeling's x^s vector as having 100% probability, and are merged with the x^s vector of the alternative labeling as having 0% probability, based on Equation 2.8. This distinction between whether a pixel belongs to the foreground or the background, based on the comparison of the probability vectors and whether that pixels was explicitly labeled as a seed, determines the segmentation.

3.4 Output

As seen in Figure 3.3, GPURW offers many different possible outputs and display modes. Using the probability outputs calculated during probability determination, a segmentation mask is created. The mask is used by the OpenGL screen output functionality to overlay the current segmentation on top of the base image. In Figure 3.3, the segmentation is displayed using a red-blue color overlay mode. The overlays can be high-contrast, such as viewing the background portion of the segmentation as the base image inverse, with the foreground shown as overly saturated. Another option is simply dulling the background and allowing the foreground to show through normally. Beyond seeing the image overlaid with the segmentation, a means for seeing the underlying calculated probabilities from the random walkers process, as well as all of the seed points that have been previously placed, is provided. In this way, if a user is attempting to determine why the segmentation is not matching their idea of what it should be, they can see where probabilities diverge from their expectations and find places that may require additional seed points.

Since GPURW follows the **Display Loop** paradigm, user inputs such as mode switches or seed manipulation are permitted as soon as screen output is written. These both act as interrupts that trigger the **Input** stage. Given display mode switches, the current input and calculated outputs are reused, with only a few OpenGL parameters requiring manipulation. This process reduces the amount of recomputation required in the **Update** stage. The **Display Loop** continues until the user either requests to exit the application, or indicates that the results are satisfactory. Once a satisfactory segmentation has been found, it is output as a grayscale alpha matte.

3.5 Application Interface

Following the direction of Fung & Murray (2008), the GPURW algorithm is implemented as a CUDA-enabled Photoshop filter. In order to interact with Photoshop, GPURW requires an RGB mode image (although the image data can encode grayscale values) to be loaded into an unlocked layer. A layer mask must then be added to the image layer. This layer mask is where the segmentation results will be written, effectively creating a matte segmentation. Once these two preliminary steps are performed, the GPURW filter must be selected, and the user will be presented with the application window. The application window is a darkened version of the image; if the mouse is within the window, a cursor indicates the location and size of the seed-painting brush.

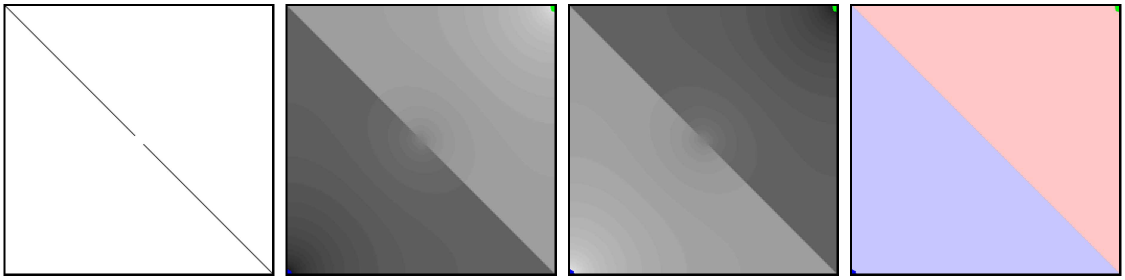
Chapter 4

RESULTS

4.1 Validity

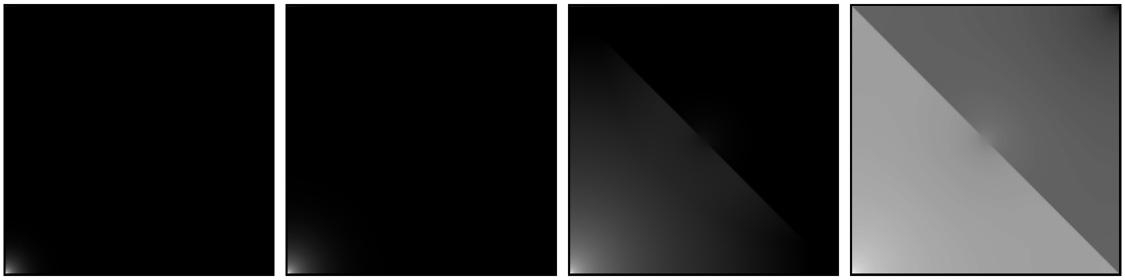
To evaluate the validity of the segmentations that GPURW produces, its results are compared to those produced by the random walkers MATLAB implementation of Grady (2006) for numeric equivalence. Grady's implementation produces a highly converged probability vector, x^s, for each labeling s. Comparing GPURW's probabilities to those produced by Grady's implementation requires that GPURW be configured to run to higher levels of convergence than it was designed for. This dictates setting τ to a fixed tolerance, and increasing i_{max} to values that no longer yield interactive results. Additionally, programmatic seeding is required that places foreground / background seeds in identical locations between the MATLAB implementation and the GPURW implementation.

Apart from a few aliasing artifacts, this setup produces nearly equal probability images between the two algorithms (see Figures 4.1b & 4.1c compared to Figures 4.1h & 4.1l, Figures 4.2b & 4.2c compared to Figures 4.2k & 4.2l, Figures 4.3a & 4.3b compared to Figures 4.3j & 4.3k, Figures 4.4a & 4.4b compared to Figures 4.4j & 4.4k, Figures 4.5b & 4.5c compared to Figures 4.5l & 4.5m, and Figures 4.6b & 4.6c compared to Figures 4.6k & 4.6l). Any variance can be attributed to the fixed tolerance τ, being either greater or less than the tolerance that the MATLAB sparse matrix operations uses. It can be seen

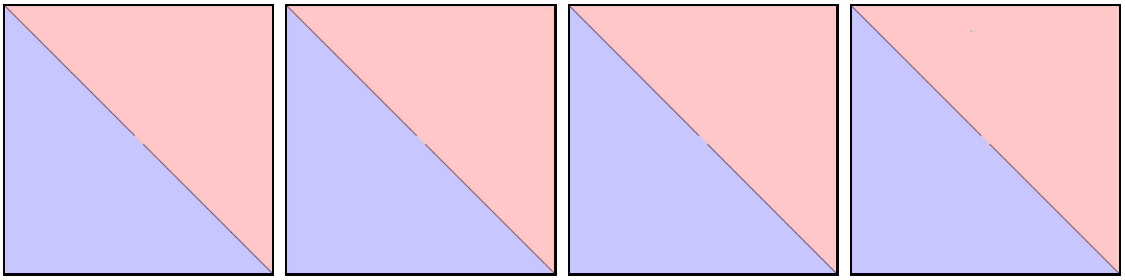


(a) Base Image (b) Foreground Grady (c) Background Grady (d) Segmentation Grady

(e) Foreground A (f) Foreground B (g) Foreground C (h) Foreground D



(i) Background A (j) Background B (k) Background C (l) Background D



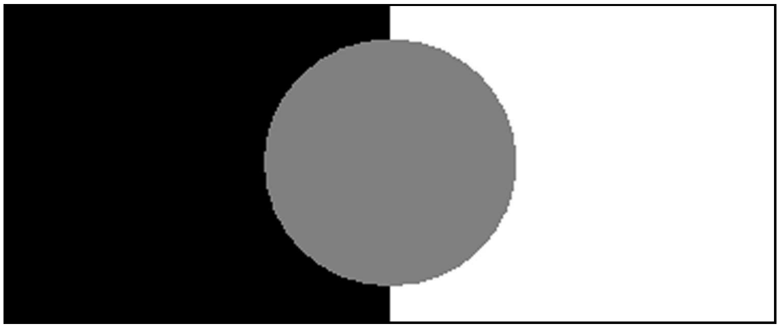
(m) Segmentation A (n) Segmentation B (o) Segmentation C (p) Segmentation D

FIG. 4.1. Comparison of GPURW outputs to outputs of Grady's (2006) MATLAB code run to convergence. The 2^{nd} & 3^{rd} rows show GPURW foreground / background probabilities for the Jacobi fixed-point solver with 800 iterations (A), the Jacobi conjugate gradient solver with 300 total iterations at 10 iterations per **Update** pass (B), the Jacobi conjugate gradient solver with 300 iterations from a single **Update** pass (C), and finally the Jacobi conjugate gradient solver with 750 iterations (D - convergence). The final row shows segmentations for the corresponding test-cases.

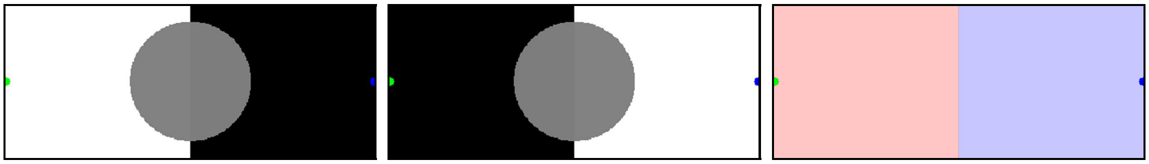
from the preceding Figures that GPURW convergence, using the iterative Jacobi conjugate gradient solver, matches that of the MATLAB random walkers implementation. The Jacobi fixed-point solver also reaches convergence, though many more iterations are required than for the Jacobi conjugate gradient method. GPURW's use of iterative solvers and its overall implementation of the random walkers algorithm are shown to produce segmentations that have the same validity as Grady's implementation of the random walkers algorithm.

However, producing interactive segmentations dictates that GPURW cannot run to this level of convergence. Instead, a relatively small value for i_{max} and calculating τ based on Equation 3.1 allows for short periods of calculation to produce intermediate results. Since GPURW does not run to full convergence, it is also necessary to show that intermediate segmentations from higher tolerances produce equivalent segmentations to those produced by full convergence (see Figures 4.1m, 4.1n & 4.1o compared to Figure 4.1d, Figures 4.2g & 4.2j compared to Figure 4.2d, Figures 4.3f & 4.3i compared to Figure 4.3c, Figures 4.4f & 4.4i compared to Figure 4.4c, Figures 4.5g & 4.5j compared to Figure 4.5d, and Figures 4.6g & 4.6j compared to Figure 4.6d). As can be seen from these Figures, GPURW using the Jacobi fixed-point solver or the Jacobi conjugate gradient solver in a variety of different seeding situations, produces identical segmentations to those produced by Grady's method run to convergence. It can also be observed that the Jacobi fixed-point solver requires significantly more iterations than the Jacobi conjugate gradient solver to produce comparable segmentations.

Since part of GPURW's preconditioning is its use of the previous **Update** pass's x^s as the x_0 values to initialize the iterative solver, showing that multiple passes through the iterative solver still produces the same segmentation is also required (see Figures 4.1n, 4.2m, & 4.3l). GPURW produces the segmentation in Figure 4.1n using the Jacobi conjugate gradient solver with 10 iterations per **Update** pass and 30 update passes for 300 total iterations. This is the same segmentation seen by running to convergence. While multiple



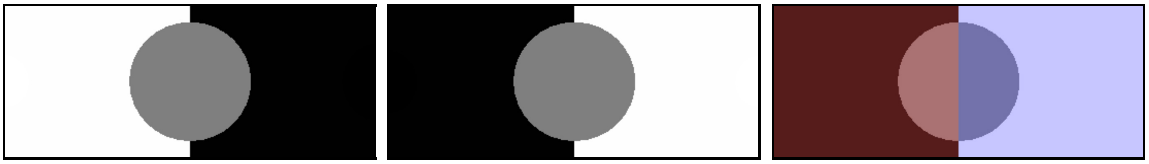
(a) Base Image



(b) Foreground Grady (c) Background Grady (d) Segmentation Grady

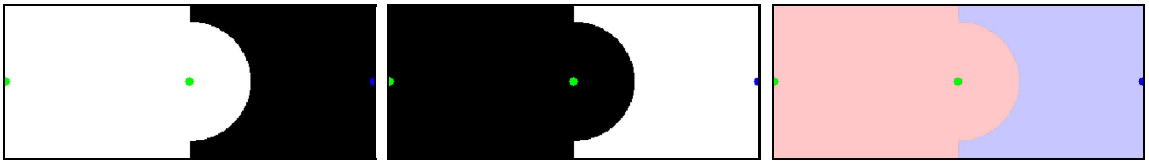
(e) Foreground A (f) Background A (g) Segmentation A

(h) Foreground B (i) Background B (j) Segmentation B



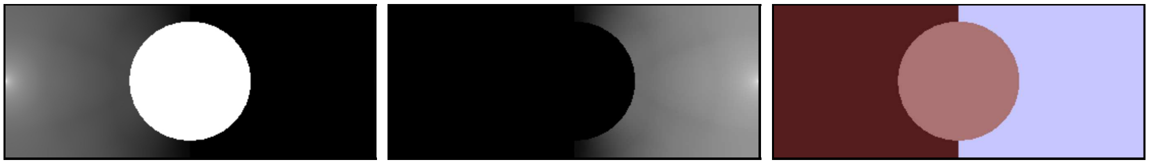
(k) Foreground C (l) Background C (m) Segmentation C

FIG. 4.2. Comparison of GPURW outputs to outputs of Grady's (2006) MATLAB code run to convergence. The center circle is 50% gray. GPURW foreground / background probabilities with segmentations are shown for the Jacobi fixed-point solver with 510 iterations (A), the Jacobi conjugate gradient solver with 260 iterations (B), and finally the Jacobi conjugate gradient solver with 3 **Update** passes at 850 iterations per pass (C - convergence).

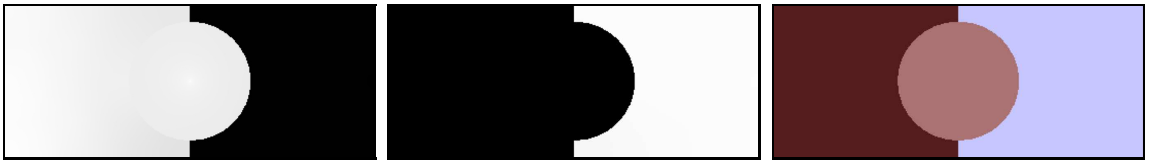


(a) Foreground Grady (b) Background Grady (c) Segmentation Grady

(d) Foreground A (e) Background A (f) Segmentation A

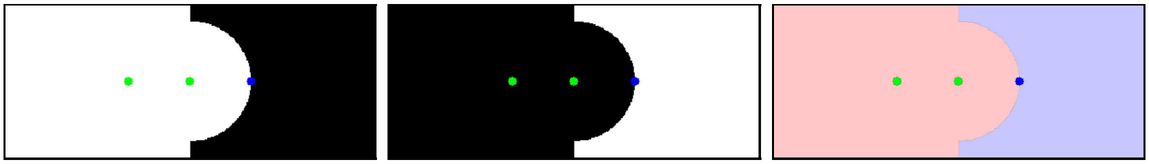


(g) Foreground B (h) Background B (i) Segmentation B

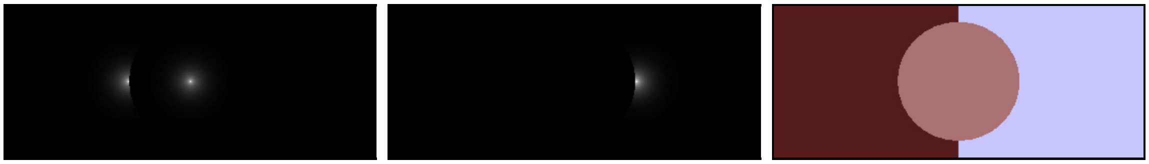


(j) Foreground C (k) Background C (l) Segmentation C

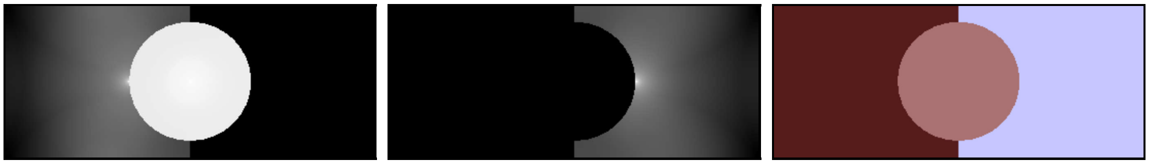
FIG. 4.3. Comparison of GPURW outputs to outputs of Grady's (2006) MATLAB code run to convergence. The center circle is 50% gray. GPURW foreground / background probabilities with segmentations having two foreground and one background seeds are shown for the Jacobi fixed-point solver with 4490 iterations (A), the Jacobi conjugate gradient solver with 270 iterations (B), and finally the Jacobi conjugate gradient solver with 2 **Update** passes at 370 iterations per pass (C - convergence).



(a) Foreground Grady (b) Background Grady (c) Segmentation Grady



(d) Foreground A (e) Background A (f) Segmentation A



(g) Foreground B (h) Background B (i) Segmentation B

(j) Foreground C (k) Background C (l) Segmentation C

FIG. 4.4. Comparison of GPURW outputs to outputs of Grady's (2006) MATLAB code run to convergence. The center circle is 50% gray. GPURW foreground / background probabilities with segmentations having two foreground and one background seeds are shown for the Jacobi fixed-point solver with 700 iterations (A), the Jacobi conjugate gradient solver with 195 iterations (B), and finally the Jacobi conjugate gradient solver with 500 iterations (C - convergence). Differently located (more closely placed) seeds allow segmentations / convergence equal to Figure 4.3 after fewer iterations.

passes through the iterative solvers that use the previous result as part of preconditioning (see Figures 4.1f & 4.1j) will yield the same segmentation as the same number of consecutive iterations (see Figures 4.1g & 4.1k), the consecutive-method approaches complete converge faster. Unfortunately, the consecutive-method approach comes at the cost of a loss of interactivity.

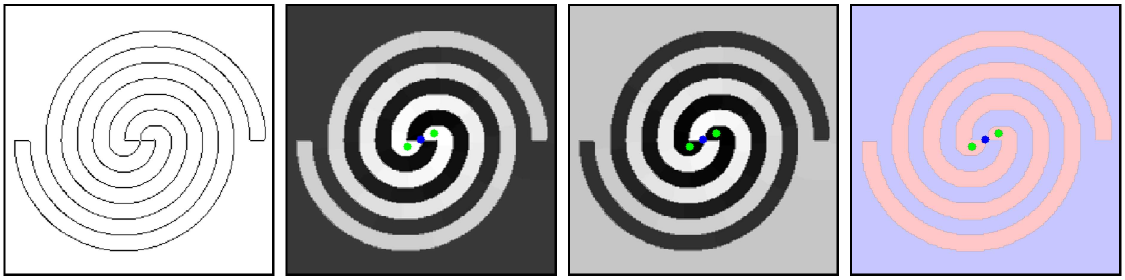
4.2 Performance Considerations

Comparing Figure 4.3 to 4.4, the latter not only converges more quickly, but also reaches the desirable segmentations more quickly. This result can be attributed to the locations of the seed points. While both Figures contain two foreground and one background seeds, the seeds in Figure 4.4 are more closely positioned, and are centrally located (further from the edges of the image). This means that the random walkers found at the edges of the image to have approximately the same distance to “walk” as those found at the center of the image. In Figure 4.3, the random walkers found at the edge of the image have to walk a short distance to reach the edge seed pixels, but random walkers towards the center of the image have much further to walk. With the seeds being more densely clustered in Figure 4.4, random walkers found between the closely located seeds more quickly show which seeds they have an affinity to. Thus, intermediate results will also reach a desirable segmentation more quickly (Figure 4.4i reaches the desired segmentation 28% faster than Figure 4.3i). Though these are properties of the random walkers algorithm that GPURW builds upon, it is important to note that the placement of seed pixels does affect convergence and segmentation speed.

In addition to the placement of seed pixels, the number of seed pixels also affects convergence speed. As can be seen by comparing Figure 4.5 to 4.6, Figure 4.6 converges significantly faster ($\approx 50\%$). Comparing the slower-to-converge Jacobi fixed-point solvers for these two examples, Figure 4.6g takes $\approx$ 24% as long as Figure 4.5g to reach the same segmentation. This difference can once again be attributed to the distance that random walkers have to “walk” to reach the seeds. With the composition of these Figures, the spiral dictates that random walkers have to walk through the spiral as if walking through a maze. The walkers at the edges of Figure 4.5 have a significant distance to travel before they reach the seeds. In contrast, the walkers with the greatest distance to travel in Figure 4.6 are those found halfway through the spiral; it takes these walkers approximately the same amount of time (number of iterations) to walk to the seeds in the center of the image as it it takes to walk to the seeds at the edges of the image. Hence there is a $\approx$ 50% speedup from Figure 4.5 to 4.6. This once again shows how seed placement and further the number of seeds can drastically affect the speed of reaching a desired segmentation. GPURW acknowledges and addresses this effect through its ability to quickly show how newly placed seeds affect the current segmentation, and its allowance for the addition of new seeds to areas that would benefit from them (areas that are not yet part of the segmentation).

Since GPURW builds upon the random walkers algorithm, it gains properties of the underlying algorithm as well. Gap-spanning (see Figure 4.1) and indeterminate-region-spanning (see Figure 4.2) are both present. These properties make the random walkers algorithm work well, yet relying solely on them and just a few seed points can result in undesirable results. As can be seen by comparing the segmentations of Figure 4.2 to Figures 4.3 & 4.4, the segmentation that is produced from just a few seeds often requires additional seeding to encompass all desired regions (in this case adding the inner circle as part of the foreground segmentation). GPURW’s approach to quickly adding seeds, and for intermediate results (not full convergence) to dictate the segmentation, makes it easy to see regions that do not match the desired segmentation and to add new seed points in those regions.

Since the number of seeds and their proximity to one another affects how quickly convergence / segmentations are reached, GPURW can be tasked with segmenting the Lena



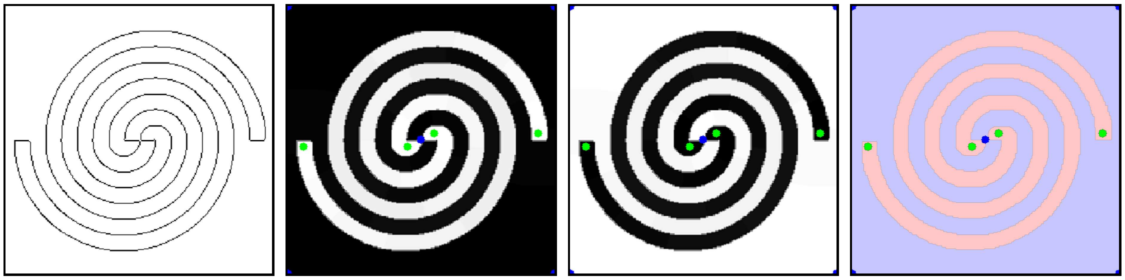
(a) Base Image (b) Foreground Grady (c) Background Grady (d) Segmentation Grady

(e) Foreground A (f) Background A (g) Segmentation A

(h) Foreground B (i) Background B (j) Segmentation B

(k) Foreground C (l) Background C (m) Segmentation C

FIG. 4.5. Comparison of GPURW outputs to outputs of Grady's (2006) MATLAB code run to convergence. GPURW foreground / background probabilities with segmentations are shown for the Jacobi fixed-point solver with 75000 iterations (A), the Jacobi conjugate gradient solver with 1530 iterations (B), and finally the Jacobi conjugate gradient solver with 2920 iterations (C - convergence).



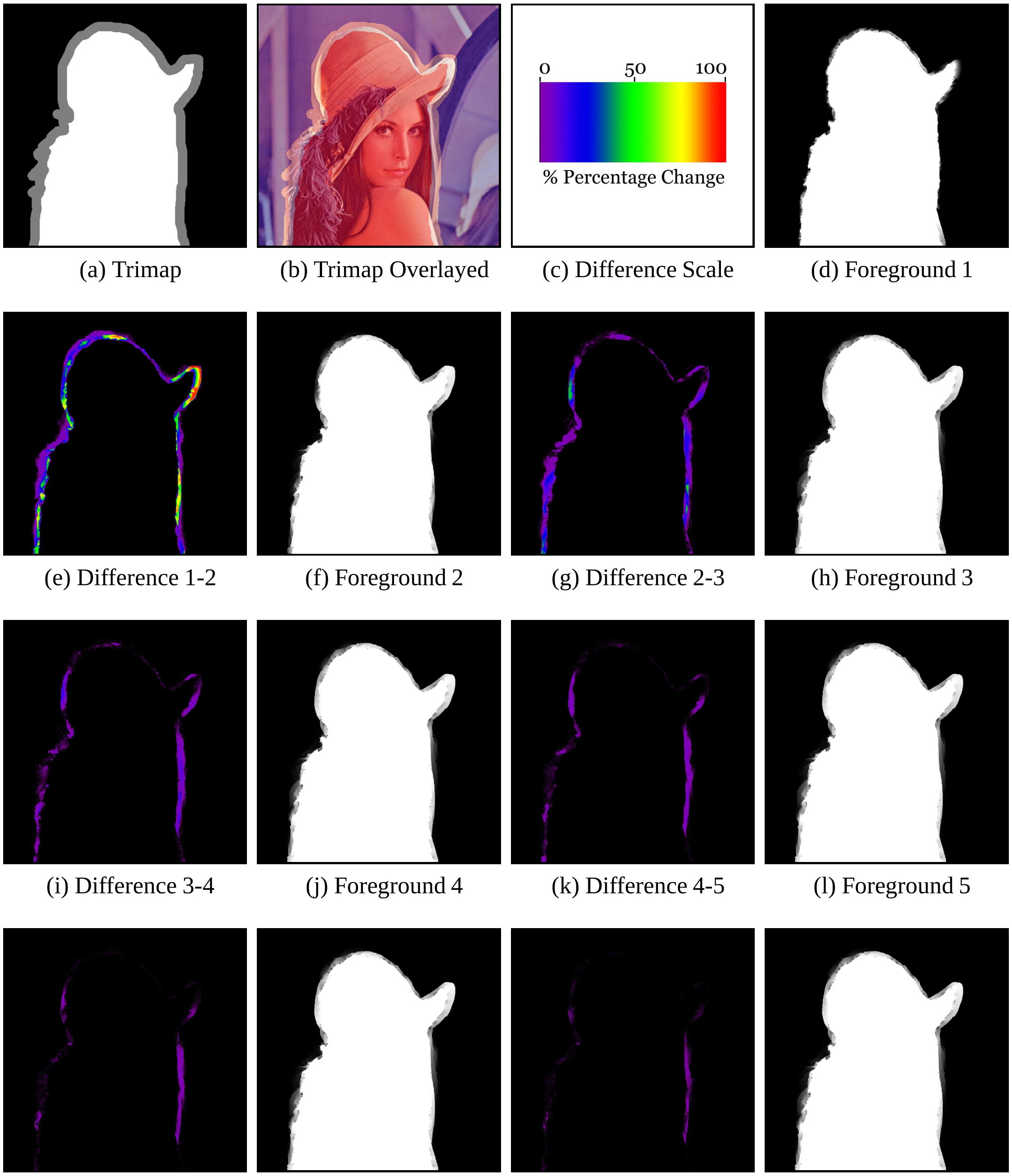
(a) Base Image (b) Foreground Grady (c) Background Grady (d) Segmentation Grady

(e) Foreground A (f) Background A (g) Segmentation A

(h) Foreground B (i) Background B (j) Segmentation B

(k) Foreground C (l) Background C (m) Segmentation C

FIG. 4.6. Comparison of GPURW outputs to outputs of Grady's (2006) MATLAB code run to convergence. GPURW foreground / background probabilities with segmentations having four foreground and five background seeds are shown for the Jacobi fixed-point solver with 17650 iterations (A), the Jacobi conjugate gradient solver with 700 iterations (B), and the Jacobi conjugate gradient solver with 1510 iterations (C - convergence). More seeds allow segmentations / convergence equal to Figure 4.5 after fewer iterations.



(a) Trimap (b) Trimap Overlayed (c) Difference Scale (d) Foreground 1

(e) Difference 1-2 (f) Foreground 2 (g) Difference 2-3 (h) Foreground 3

(i) Difference 3-4 (j) Foreground 4 (k) Difference 4-5 (l) Foreground 5

(m) Difference 5-6 (n) Foreground 6 (o) Difference 6-7 (p) Foreground 7

FIG. 4.7. Probability images (Foreground 1 - 7) and the amount of difference between one probability image and the successive probability image. All images were generated using the GPURW algorithm with a Jacobi conjugate gradient solver. The solver went through the image number (Foreground 1 = 1 ... Foreground 7 = 7) **Update** passes at 100 iterations per pass. The trimap equates to white being labeled as foreground seeds, black as background seeds, and gray being left as unknown to be solved for.

female image based on a trimap approach (see Figure 4.7). Recall that in a trimap, there are known areas of foreground and background, as well as regions of unknown pixels to be solved for. This directly equates to the random walkers algorithm with two possible labellings (foreground / background) and thus to the GPURW algorithm. The unknown regions equate to pixels that have not been given a seed and need to be solved for. Programmatically seeding GPURW with the trimap shown in Figure 4.7a, where everything in black is labeled as background seeds, everything in white as foreground, and the gray regions being left as unknown, produces a segmentation after a small number of iterations that is close to that seen in a converged result. In looking at the successive images in Figure 4.7, the greatest amount of change from **Update** pass to **Update** pass can be seen in the earliest passes. As additional passes occur, the amount of new probability information that is calculated can be seen to significantly decrease from red sections of nearly 100% change in Figure 4.7e, to the greatest change being less than 5% in Figure 4.7o.

Observing this property lends merit to GPURW's implementation of the seed-painting brush, which enables a user to paint varying-sized regions of seeds at one time based on the size of the current brush. Adding new seeds to regions of little change (regions that would otherwise take a large number of iterations to be added to the segmentation) results in the greatest amount of change in those region for the immediately proceeding **Update** passes. Additionally, the change in the effect of those new seeds is seen to level out as seeds are added elsewhere. Should new seeds be added in close proximity to preexisting seeds, there will once again be the largest amount of change surrounding these new seeds, with that changing being driven by both the new and the old seeds.

The GPURW method provides interactive image segmentations that permit continual specification of additional user intentions. Since it presents a new way of interacting with the segmentation problem, there are still a number of challenges. Through continued use of the interface, good and bad usage patterns have been identified that dictate to what extent the application remains interactive. Specifying a large number of seeds initially (i.e., starting with a large seed painting brush) results in a very low initial tolerance τ. When the algorithm begins, no or limited preconditioning information exists from prior iterations. Thus, with a low initial τ, the algorithm must run through more iterations in order to converge upon the low tolerance. The adverse affect of a lower tolerance for τ is a greater number of iterations to reach that tolerance. Thus, the system becomes less interactive. One solution to this is to reduce i_{max} to a value that permits interactivity regardless of τ.

By starting with the specification of just a few seeds, and a correspondingly higher τ, convergence is reached much more quickly then is the case with a lower τ. As more seeds are added, the application utilizes the results from each prior pass to accommodate the tolerance being gradually restricted. Though this usage pattern proves to be optimal for this application, it is not imposed as a restriction. It is up to the user to decide how they want to interact with the application. The performance statistics outlined in the following sections reflect this preferred usage pattern.

4.3 Base Performance

The testing platform used was a Windows Vista™ system comprised of an Intel® Core™ 2 Duo 6400 running at 2.13 GHz, 2 - 1GB dims of DDR2 RAM, and an NVIDIA® GeForce® 8800 GTS 640MB GPU all running at factory speeds. As can be seen in Table 4.1, the number of rows and non-zero entries in the CUDPP CRS (compressed row storage) sparse matrix structure, scale logarithmically with image size. The device and system memory also follow a logarithmic trend beyond constant memory requirements (see Figure 4.8).

Table 4.1 shows favorable results in terms of the amount of time necessary for producing a “satisfactory segmentation.” Since satisfactory segmentation is a subjective measure, producing results that approach an unrefined (without large amounts of time spent highly refining edges) result of similar quality as other segmentation algorithms is used as the success criterion. Due to the edge limitation that will be discussed in Section 5.1.1, expecting other algorithms to produce perfect results would give GPURW an unfair advantage in comparison to those other algorithms.

The test system has noticeable difficulty maintaining real-time performance with larger images, although interacting with the algorithm on images up to 1024^2 is feasible (interactivity here is defined as the system functioning at an average of two or more frames per second). As image size increases, the amount of information to be processed far exceeds the concurrent processing capabilities of the available hardware. For the NVIDIA® GeForce® 8800 GTS 640MB GPU used, there is the possibility of 9,216 concurrent threads[1] when the GPU is fully utilized. It can be seen from Table 4.1, that in the simple case of calculating all of the non-zero values for the sparse matrix even for a small 128×128 image, the number of non-zero values is much greater than the number of concurrent threads that this GPU is capable of handling. This lack of balance between threads and information to process results in all of the information not being processed at once, but instead being processed in thread *blocks*. Each block must wait until previous blocks have completed processing to gain access to the GPU’s processors. Yet the performance of the random walkers algorithm—and therefore GPURW—scales depending upon the hardware on which it is run. Fortunately, incorporating additional parallel computing power (i.e., making more threads available for concurrent processing) into systems using CUDA is feasible. NVIDIA offers the ability to link multiple GPUs together, providing more computing power as needed. Handling larger images at more interactive rates could therefore be enabled by incorporating additional

---

[1] According to NVIDIA (2008a), this GPU has 12 multiprocessors with a maximum of 768 threads per multiprocessor.

Image Size (pixels)	**128^2**	**256^2**	**512^2**	**1024^2**
CRS Row Count (pixel count)	16,384	65,536	262,144	1,048,576
CRS Non-Zero Values	65,024	261,120	1,046,528	4,190,208
Device Memory Used (MB)	59.72	69.88	110.5	273.3
System Memory Used (MB)	50.54	56.484	88.588	223.564
Satisfactory Segmentation (sec.)	14.9	19.1	34.3	93.1

Table 4.1. Statistics pertaining to GPURW for the Lena female image at different resolutions. Note that the device memory consumption indicates total memory being used by the GPU. Thus, there is constant memory in use due to the GPU additionally functioning as the primary display adapter (the operating system accounts for this constant allotment of ≈ 57.2 MB for general display purposes).

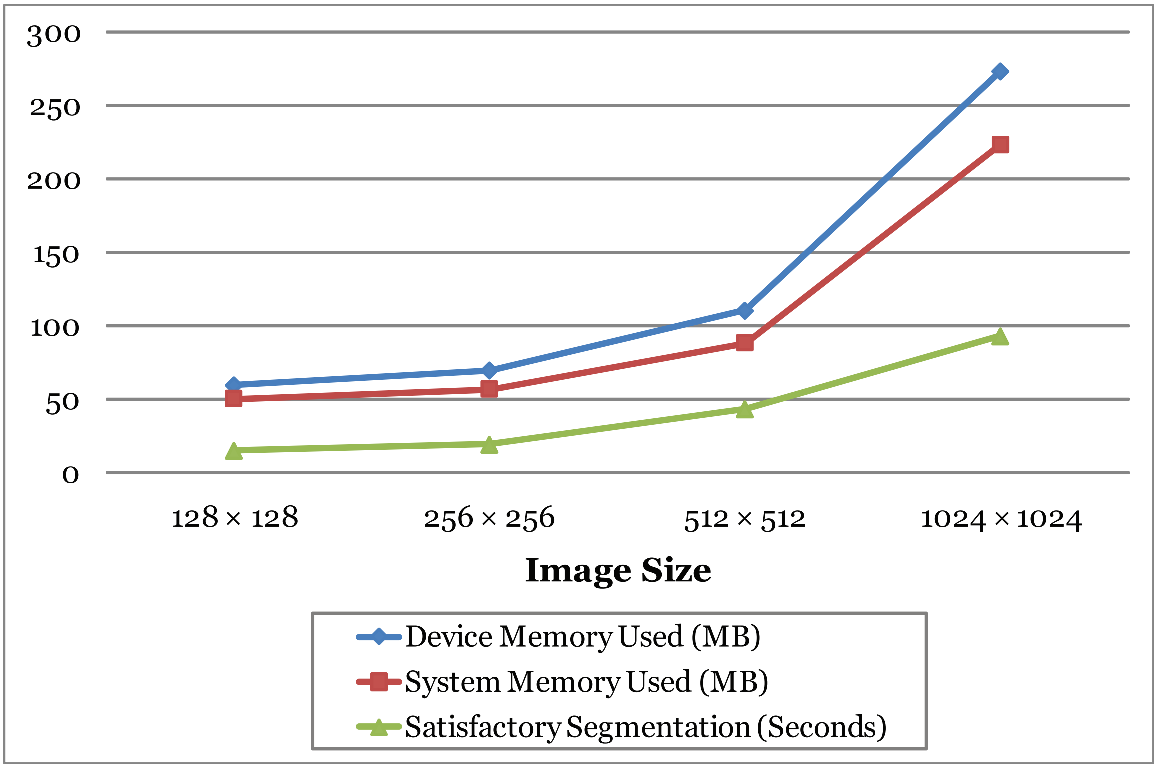


FIG. 4.8. Plotted statistics from Table 4.1, showing the correlation between the logarithmic scaling of memory use and time-to-segment with the logarithmic changes in image size.

compatible GPUs and programming CUDA to take advantage of them.

4.4 Performance Comparison

Since Photoshop is a professional digital image editing suite, some of the algorithms outlined in this paper have been implemented as Photoshop filters or selection tools, and are thus available for comparison purposes. Figure 4.9 compares the segmentation results and user interaction required by the GPURW framework to these other Photoshop plug-ins. Each is tasked with segmenting Lena (the woman in the image) from her background. The plug-ins in the comparison are:

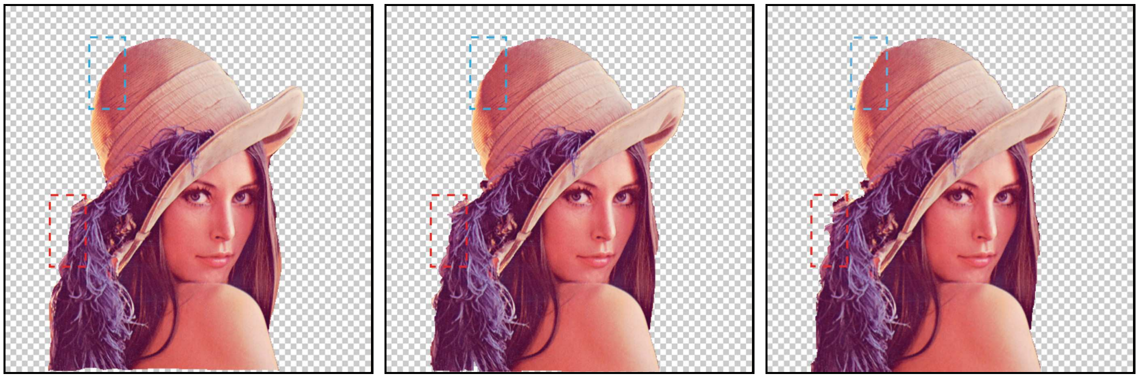
- Adobe's *Magnetic Lasso*
- *GrabCut* (implementation does not include edge-matting)
- GPURW (binary masking)
- Digital Film Tools© *Snap* v2.5.3 (graph cutting implementation)
- Digital Film Tools© *Power Mask* v1.0 (*Soft Scissors* implementation)
- GPURW Probability (probability-based masking)

The findings indicate that in all of the algorithms, edge handling poses the greatest challenge. While many algorithms offer advanced features to refine the edges, the time required for edge refinement significantly adds to the initial segmentation time.

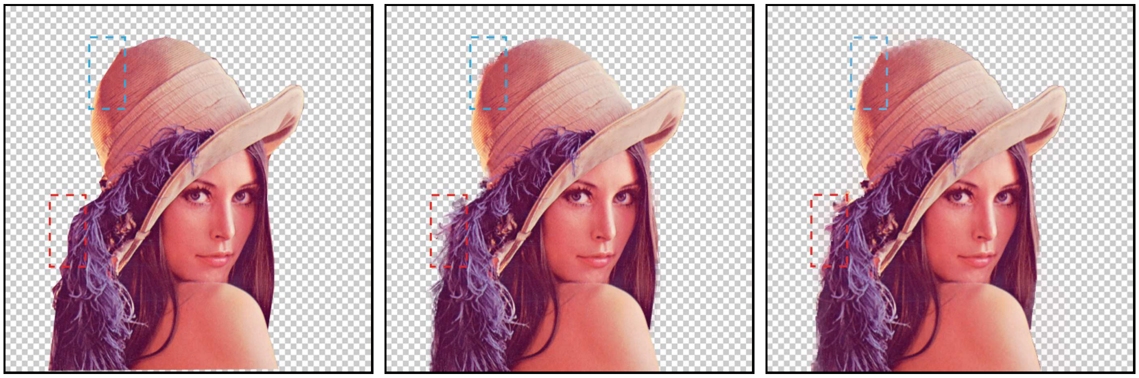
Of these segmentation algorithms, the majority provide binary (strictly foreground or background) results, except for *Power Mask* and the probability-based masking version of GPURW. Observing the binary results, all have difficulty with the purple feather and the hair on the right side of Lena's head, though *GrabCut* and GPURW retain some of the edge detail (see Figures 4.10b & 4.10c respectively). *Snap* loses parts of the edge contour due to its simplifying the graph-cutting results to produce a spline. While this results in a smooth edge, fine detail along the edge gets lost (see Figure 4.10d). *Magnetic Lasso* also suffers from a similar issue, in that the segmentation boundary "snaps" to features that sometimes encompass edge detail, and other times, to features that lie within the object or background. This "snapping" causes the boundary to over-simplify segmentation details (see Figure 4.10a). All of the binary methods handle distinct edges, like the one defined by the edge of Lena's hat, fairly uniformly.

Since there was no direct comparison for *Power Mask*, GPURW was modified to use the foreground probabilities as the matte directly (rather than determining the maximum probability between the foreground and background). By doing so, the resultant matte's values are able to indicate strictly foreground, strictly background, or some combination of foreground / background for each pixel. A comparison of the probability-based masking version of GPURW to the standard GPURW algorithm can be seen in Figure 4.11. Note that in order to achieve foreground probabilities that are able to produce a viable matte, either the probabilities need to be normalized, or GPURW needs to be run until it approaches convergence. This probability-based masking version of GPURW and *Power Mask* both handle the hair and feather more thoroughly than the binary segmentation methods. *Power Mask* does a better job than GPURW with the feather in ensuring that details remain intact (see in Figure 4.9e and more visibly in Figure 4.10e how the tendrils of the feather extend beyond the main edge). However, both of these methods have errors along the top edge of the hat, either adding to or taking away from what should be included in the segmentation (see the left side of Figures 4.10e & 4.10f).

With the *Magnetic Lasso*, *GrabCut* refinements (beyond the initial rough bounding box specification) and *Power Mask*, careful input is required. For the *Magnetic Lasso* and *Power Mask*, the main interaction with the algorithms is in explicitly tracing the edge of a desired object. For *GrabCut* refinements, specifying either additional sections of fore-



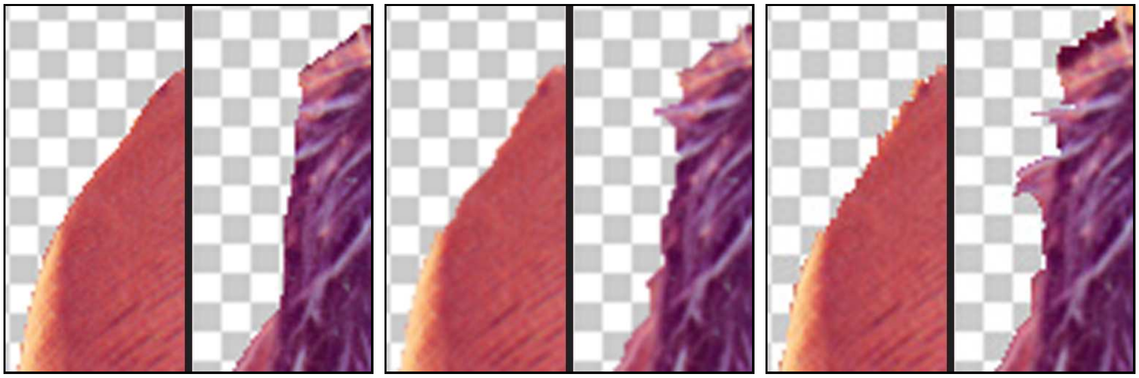
(a) Magnetic Lasso - 43s (b) GrabCut - 71s (c) GPURW - 48s



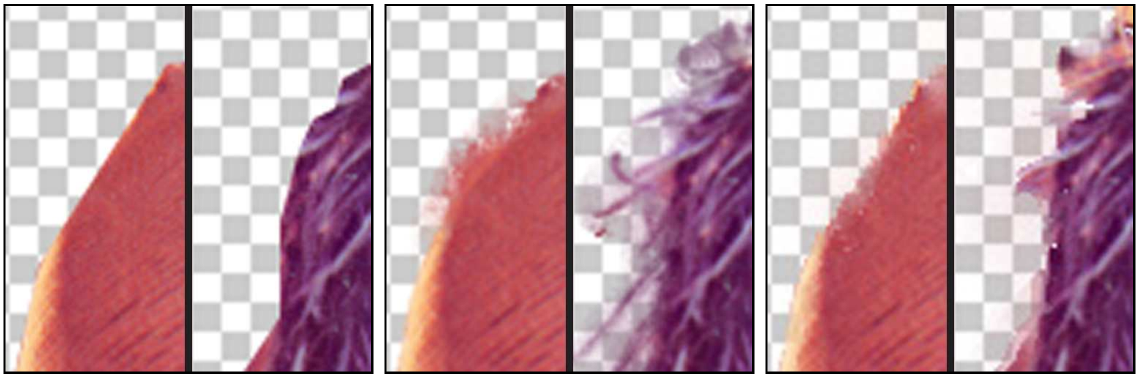
(d) DFT Snap - 40s (e) DFT Power Mask - 78s (f) GPURW Probability - 71s

Strokes (a) Strokes (b) Strokes (c) Strokes (d) Strokes (e) Strokes (f)

FIG. 4.9. Comparisons of different segmentation methods. Segmentations can be seen in the top two rows (the amount of time required for the segmentation in seconds is shown next to the name of the algorithm), while the inputs required to create the segmentations in their respective interface can be seen in the bottom row. Red strokes equate to foreground markings, blue to background, and yellow represent explicit-boundary indications. The boxes with dashed lines in the segmentation image are enlarged in Figure 4.10, with the blue seen on the left, and the red on the right.



(a) Zoom of Figure 4.9a (b) Zoom of Figure 4.9b (c) Zoom of Figure 4.9c



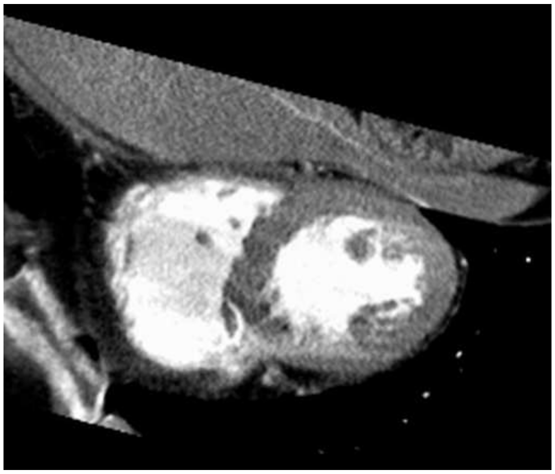
(d) Zoom of Figure 4.9d (e) Zoom of Figure 4.9e (f) Zoom of Figure 4.9f

FIG. 4.10. Comparisons of specific regions of different segmentation methods. The left side of each image is an enlarged version of the blue dashed box seen in Figure 4.9, while the right is for the red dashed box in the same Figure. The left side shows a region of the segmentation that perceptually should be a distinct hard edge running along the top of Lena’s hat. The right side shows a region comprised of the feather from Lena’s hat, which presents the challenge of semi-transparency.

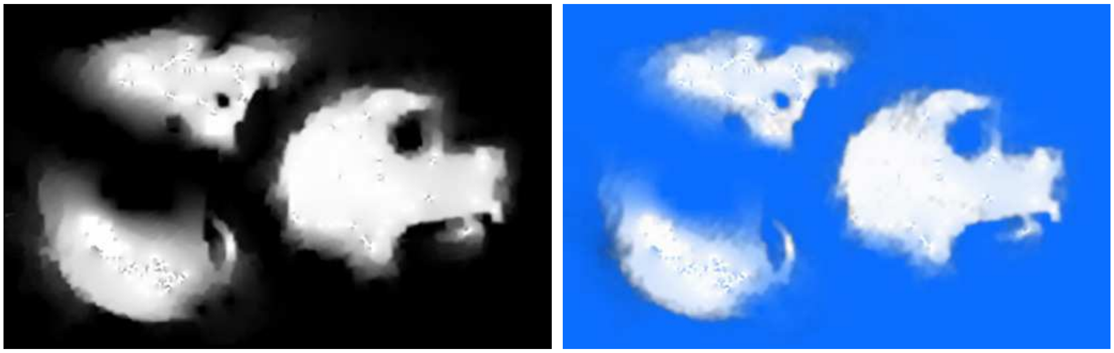
ground or background must occur very close to the actual boundary for the new specifications to have any effect on the segmentation. All three require patience and a steady hand, since accurate input equates to accurate resultant segmentations. In contrast, *Snap*, both versions of GPURW, and the initial bounding box specification of *GrabCut* permit rough indications of the user's desired segmentation. The benefit of handling these rough indications is seen in the reduced amount of time that it takes for them to be specified by a user.

Figure 4.12 shows a closely textured CT scan, in which the edges of the objects for segmentation (the three white regions) blend into their surroundings. There are no distinct edges for the three regions, but rather a tapering from white to the gray texture that surrounds them. As can be seen from Figure 4.12d, due to the lack of distinct edges, the *Magnetic Lasso* tool requires a large amount of input. This results in its taking nearly double the amount of time that *Snap* & *Power Mask* take, and more than twice the time that GPURW requires. It once again suffers from a lack of edge detail due to its over-simplification between input points. Unlike the rest of the algorithms, *Snap* is unable to handle multiple distinct objects, which results in three invocations of the plug-in to produce the three distinct objects of Figure 4.12g. Like the *Magnetic Lasso*, it suffers from over-simplification, which results in a smooth line between control points, but hard corners at the control points.

GrabCut & GPURW both handle the lack of defined edges more thoroughly than the *Magnetic Lasso*, and are able to capture pixel-level edge detail (see Figures 4.12b & 4.12c). GPURW does so in about two-thirds the time that *GrabCut* takes to attain its segmentation, and performs faster than any of the other algorithms for this image. Once again, both methods benefit from the ease of specifying rough indications of the user's desired segmentation. Both algorithms require only a few explicit strokes (in foreground regions of fine detail) beyond the rough indications to attain the segmentations seen.

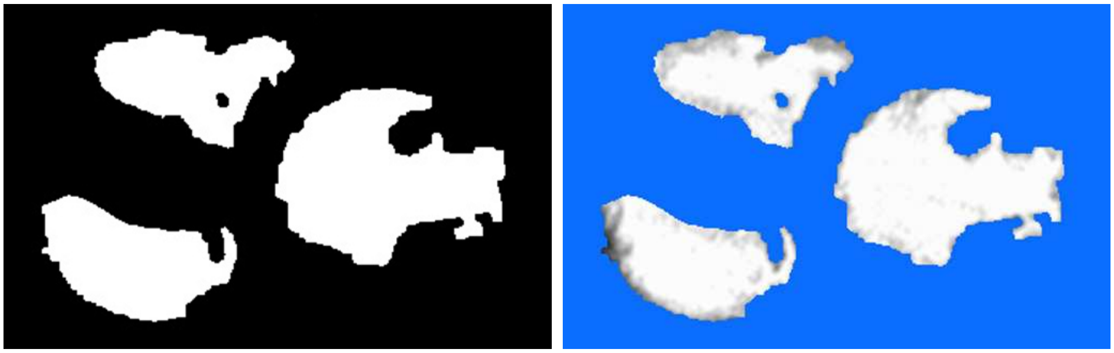


(a) CT Scan



(b) Foreground Probabilities

(c) Probability-Based Segmentation



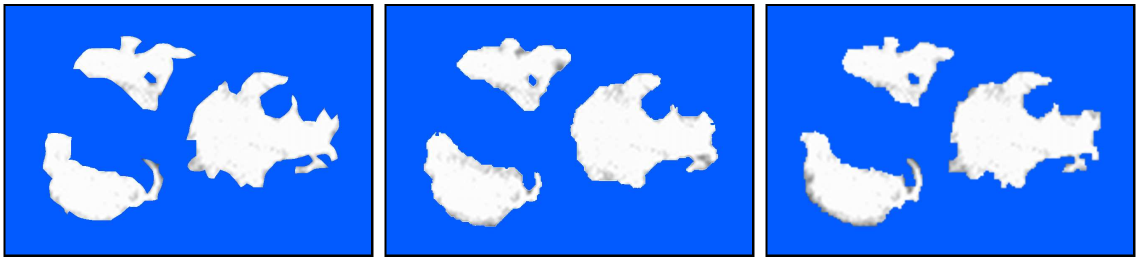
(d) Hard Mask

(e) Hard Mask-Based Segmentation

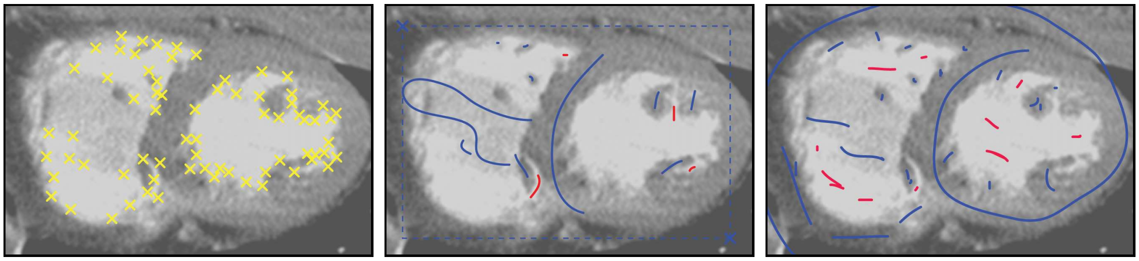
FIG. 4.11. Comparison of returning a matte based solely on the foreground probabilities, versus returning the hard segmentation matte that GPURW returns. The results have been cropped to allow for greater visible distinction. The base-image comes from Grady (2006).

For *Power Mask*, no additional refining strokes are required beyond the explicit boundary indications (see Figure 4.12h for the segmentation, and Figure 4.12k for the boundary indications). Both *Power Mask* and the probability-based masking version of GPURW capture edge variability that may be preferable for this type of image. Doing so might enable a doctor to see how confident they should be when operating on the edges of a segmentation region seen for the image. In the event that the distinction between the three white segments and their surroundings equated to healthy versus diseased tissue, much more care would be needed on the doctor’s part when removing regions where the tissue was not solely diseased (those regions seen in Figures 4.12h & more distinctly in 4.12i, where the blue background shows through the segmentation).

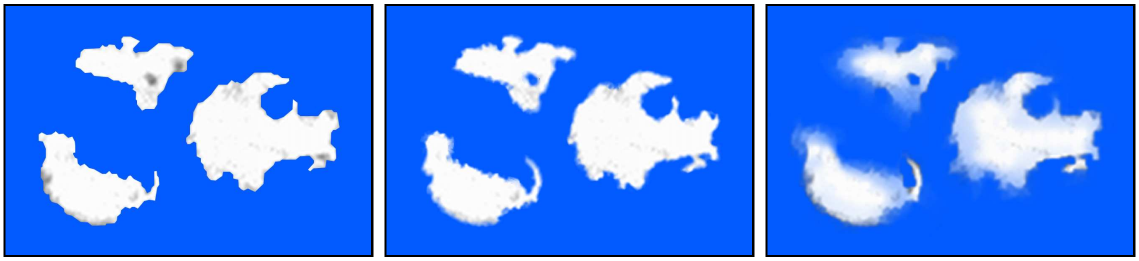
Based on the results seen in both Figures 4.9 and 4.12, the GPURW algorithm is able to perform segmentations that are comparable to or better than the other image segmentation algorithms. Not only does GPURW produce high-quality segmentations, but it does so with real-time interactive feedback, showing the user what the current segmentation looks like at all stages of the segmentation process. This enables GPURW to produce quality segmentations more quickly than competitive algorithms.



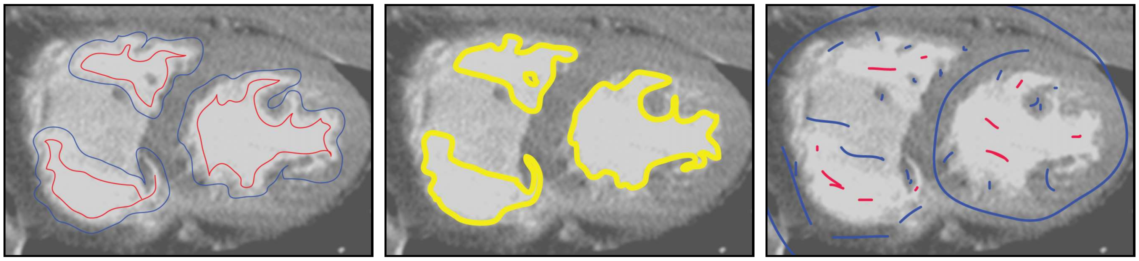
(a) Magnetic Lasso - 152s (b) GrabCut - 91s (c) GPURW - 67s



(d) Magnetic Lasso Strokes (e) GrabCut Strokes (f) GPURW Strokes



(g) DFT Snap - 83s (h) DFT Power Mask - 81s (i) GPURW Probability - 113s



(j) DFT Snap Strokes (k) DFT Power Mask Strokes (l) GPURW Probability Strokes

FIG. 4.12. Comparisons of different segmentation methods on a cropped region of a closely textured CT Scan (the amount of time required for the segmentation in seconds is shown next to the name of the algorithm). The inputs required to create the segmentations in their respective interface are shown immediately below the segmentation results. Red strokes equate to foreground markings, blue to background, and yellow represent explicit-boundary indications.

Chapter 5

CONCLUSIONS

5.1 Limitations And Future Work

Although the results presented in this thesis are promising, there are several limitations in the current system. Continued work in these areas could yield significant benefit for the GPURW algorithm. These are detailed as possibilities for future research.

5.1.1 Edge Handling

One of the primary shortcomings of the GPURW algorithm is its lack of specialized handling for transparency along segmentation edges. Since segmentation is based on approximate convergences, with the result being determined by those pixels that have greater random walkers probabilities for the foreground than for the background, GPURW produces binary segmentations (see Figure 4.11e). Figure 4.11c shows that merely using the probabilities to determine the final matte is not sufficient, since not all transparency of objects is desirable. Algorithms such as *Soft Scissors* (Wang, Agrawala, & Cohen 2007) could provide additional insight into how to extend GPURW to incorporate the necessary segmentation edge handling.

5.1.2 Flood Filling

While testing input images such as Figures 4.2 - 4.6 and Figure 4.11, in which large areas of the image have the same pixel information (all black / white / gray), it seemed that it might be possible to apply a tolerance-based flood-filling to regions of similar color in a fashion similar to the seed-filling of Heckbert (1990). One possibility is to flood-fill seed points in the region, since all pixels within the given similarity region presumably would receive the same labeling. This approach presents challenges for the calculation of τ and the preconditioning of the x_0 vector, and renders the gap-spanning ability of the random walkers algorithm useless (as can be seen in Figure 4.1).

An alternative approach would be to use flood-filling for preconditioning the x_0 vector. Doing so would also need to affect the calculation of τ in some way. Otherwise, the addition of new refinement seeds might produce no actual changes to the segmentation, since τ could result in immediate convergence with the better preconditioned x_0 vector. In either case, additional research is necessary as to whether GPURW could benefit from the flood-filling algorithm, and what changes to the calculation of τ would be needed to make this a viable option.

5.1.3 CUDPP Sparse Matrix Vector Multiplication

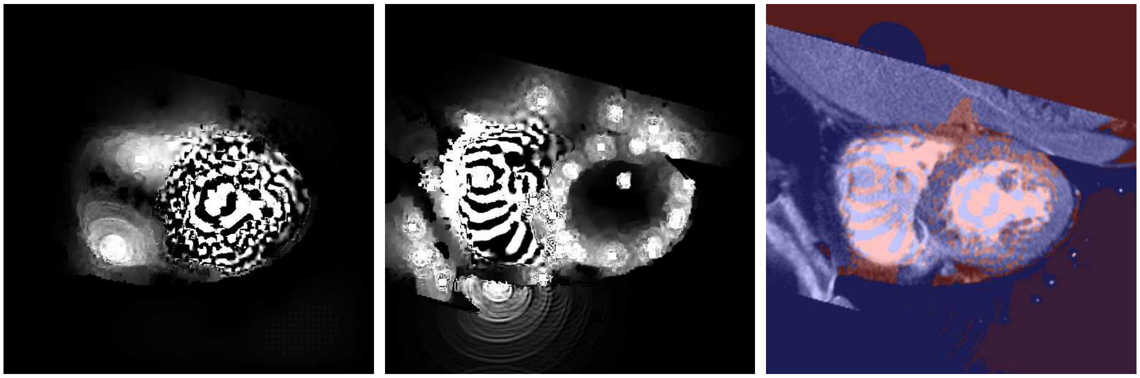
Although CUDPP provides a general solution to sparse matrix-vector multiplication on the GPU using CUDA, it is not necessarily the optimal solution for the class of problems including GPURW. Zeroing the row, column, and sum values in L for rows / columns where the sum corresponds to a seed point to produce L_U results in unnecessary calculations, which would not be incurred by an ideal sparse matrix structure and sparse matrix-vector multiplication algorithm.

Since certain properties are known about the sparse matrices for the random walker algorithm (one primary diagonal, four secondary diagonals, symmetric), a specialized algorithm such as the one presented by Grady et al. (2005) might be faster. Since the matrix is symmetric, storing all four secondary diagonals is not necessary. Depending on the relative cost of accessing memory and addition operations, calculating the diagonal at runtime from gathered secondary diagonals might also speed up performance while saving storage. Ultimately, further refinements could be made to matrix-vector multiplication handling and the structure used to store the sparse matrix. Since this is the most computationally intensive part of the GPURW algorithm, any improvements would offer significant benefits.

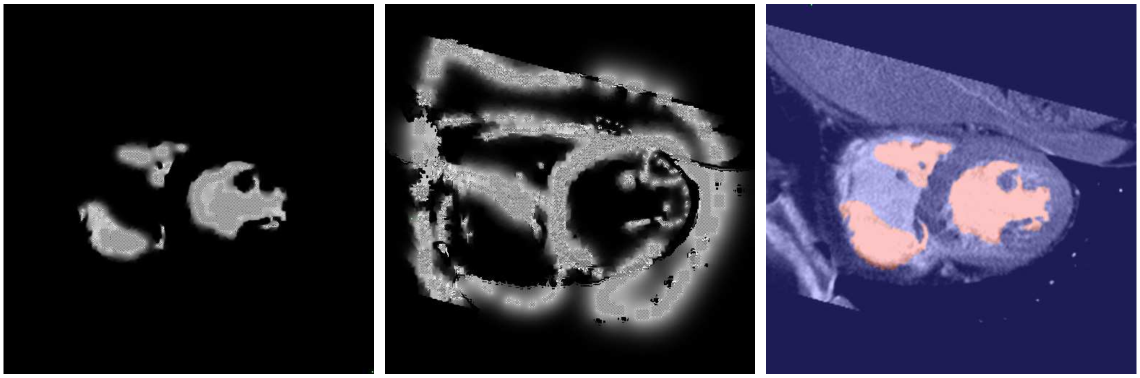
5.1.4 Different Iterative Solvers

In order to determine if the random walker problem could benefit from different iterative methods, the fixed-point Jacobi, unpreconditioned CG (conjugate gradient), and Jacobi preconditioned CG solvers were all implemented. Since the fixed-point Jacobi is the simplest form of solver, it was the first to be implemented, and produced the best results, enabling iterative interaction with the system, and the fewest “stutters” (caused by computational delays). The Jacobi preconditioned CG solver was also fairly interactive. As can be seen in Figures 4.1 - 4.6, this solver outperforms the fixed-point Jacobi solver in terms of number of iterations required for reaching a desired segmentation. With additional changes to the computation and use of τ, this solver may be the most promising.

Unlike the Jacobi fixed-point and Jacobi preconditioned CG solvers, the unpreconditioned CG solver presented unexpected results. The expectation was that regardless of the solver being used, the same results would be produced with varying computation times. However, it was surprising to find that this solver produced completely different results than the other two. As seen in Figures 5.1a and 5.1b, the calculated random walker probabilities did not match those produced by the other solvers. Instead, the probabilities had a noticeable rippling pattern, reminiscent of those seen when a pebble is dropped into a body



(a) CG Foreground (b) CG Background (c) CG Segmentation



(d) CG Checkerboard Foreground (e) CG Checkerboard Background (f) CG Checkerboard Segmentation

FIG. 5.1. Use of an unpreconditioned CG solver yielding ripple effects. Due to convergence never being reached, an iteration threshold (i_{max}) of 20 iterations per **Update** was imposed. Modifying the seed painting brush to paint seeds on a checkerboard pattern (i.e., no adjacent seeds), the unpreconditioned CG solver was rendered usable. Due to the reduced number of seeds caused by preventing seed adjacency, the calculation of τ was affected and user-initiated refinements were necessary.

of water, with the seed points being the centers of the ripples. While the reason for these results remains unclear, disallowing adjacent seeds solved the problem. By modifying the seed-painting brush to only place seeds on a checkerboard patterned grid, the rippling effect went away. Since this constraint also cut the number of possible seeds for the image in half, modifying the calculation of τ by replacing $|V|$ with $\frac{|V|}{2}$ might yield similar results to the other solvers, though it was found that merely invoking additional user-initiated refinements produced satisfactory results.

5.2 Conclusion

This thesis has presented GPURW, a fast and accurate single-image segmentation algorithm that allows for binary (object / not object) interactive segmentations through an iterative process. By painting seeds on an underlying image, the user refines the current segmentation by adding, removing, or changing seeds, which moves them towards their desired segmentation. Through this gradual indication of which sections of the image should be included in the foreground / background segmentations, accuracy is accumulated towards the desired result.

Since the random walkers algorithm that GPURW builds upon requires solving large sparse systems of equations, GPURW employs NVIDIA's CUDA-accelerated iterative solvers to aid in the calculation of the random walker probabilities. It produces real-time visual feedback for the current segmentation by utilizing this parallel acceleration, by introducing a novel tolerance calculation, and through the use of prior results to precondition the iterative solvers. Learning to use the GPURW application is simple and straightforward, and it produces segmentations that are as good as or better than those produced by other binary segmentation methods.

Ultimately, segmentation presents the challenge of finding the solution to an under-constrained problem. Through GPURW's iterative process, and with the user's aid in indicating what they expect from the segmentation, GPURW provides the ability to add constraints that yield results representative of a particular user's desired segmentation.

REFERENCES

[2004] Acosta, B. D. 2004. Experiments in Image Segmentation for Automatic US License Plate Recognition. Master's thesis, Virginia Polytechnic Institute and State University.

[1998] Adobe. 1998. Adobe Photoshop 5.0 New Feature Highlights. http://ftp.build.bg/Books_and_Help/Prepress/photoshop5.pdf.

[2008] Adobe. 2008. Photoshop CS3. http://livedocs.adobe.com/en_US/Photoshop/10.0.

[2004] Agarwala, A.; Dontcheva, M.; Agrawala, M.; Drucker, S.; Colburn, A.; Curless, B.; Salesin, D.; and Cohen, M. 2004. Interactive Digital Photomontage. In *SIGGRAPH '04*, 294–302. New York, NY, USA: ACM.

[2008a] AMD. 2008a. AMD Stream Computing - Technical Overview. http://ati.amd.com/technology/streamcomputing/AMD_Stream_Computing_Overview.pdf.

[2008b] AMD. 2008b. AMD Stream Processor First to Break 1 Teraflop Barrier. http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543~126593,00.html.

[1994] Barrett, R.; Berry, M.; Chan, T. F.; Demmel, J.; Donato, J.; Dongarra, J.; Eijkhout, V.; Pozo, R.; Romine, C.; and der Vorst, H. V. 1994. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Philadelphia, PA: SIAM.

[2003] Bolz, J.; Farmer, I.; Grinspun, E.; and Schröoder, P. 2003. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. In *SIGGRAPH '03*, 917–924. New York, NY, USA: ACM.

[2001] Boykov, Y. Y., and Jolly, M. P. 2001. Interactive Graph Cuts for Optimal Boundary

& Region Segmentation of Objects in N-D Images. In *Prococeedings of the Eighth IEEE International Conference on Computer Vision ICCV 2001*, volume 1, 105–112.

[2002] Boykov, Y., and Kolmogorov, V. 2002. An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision. In *IEEE Transactions on PAMI*. IEEE Computer Society.

[2003] Boykov, Y., and Kolmogorov, V. 2003. Computing Geodesics and Minimal Surfaces via Graph Cuts. In *ICCV '03: Proceedings of the Ninth IEEE International Conference on Computer Vision*. Washington, DC, USA: IEEE Computer Society.

[to appear] Buatois, L.; Caumon, G.; and Lévy, B. to appear. Concurrent Number Cruncher - A GPU Implementation of a General Sparse Linear Solver. *International Journal of Parallel, Emergent and Distributed Systems*.

[2008] Buck, I.; Foley, T.; Horn, D.; Sugerman, J.; Hanrahan, P.; Houston, M.; and Fatahalian, K. 2008. BrookGPU. http://graphics.stanford.edu/projects/brookgpu/. Stanford University Graphics Lab.

[2007] Charpentier, F. 2007. Specifications - Tom's Hardware - Nvidia's GeForce 8800 GTS 512 MB. http://www.tomshardware.com/reviews/geforce-8800-gts-512-mb,1743-2.html.

[2001] Chuang, Y.-Y.; Curless, B.; Salesin, D. H.; and Szeliski, R. 2001. A Bayesian Approach to Digital Matting. In *Proceedings of IEEE CVPR 2001*, volume 2, 264–271. IEEE Computer Society.

[2008] Crytek. 2008. CryENGINE2. http://www.cryengine2.com/index.php?pnr=1&conid=2.

[1984] Dodziuk, J. 1984. Difference Equations, Isoperimetric Inequality and the Transience of Certain Random Walks. In *Transactions of the American Mathematical Society*, volume 284, 787–794.

[1999] Efros, A. A., and Leung, T. K. 1999. Texture Synthesis by Non-parametric Sampling. In *IEEE International Conference on Computer Vision*, 1033–1038.

[1998] Falcão, A. X.; Udupa, J. K.; Samarasekera, S.; Sharma, S.; Hirsch, B. E.; and de A. Lotufo, R. 1998. User-Steered Image Segmentation Paradigms: Live Wire and Live Lane. *Graphical Models and Image Processing* 60(4):233–260.

[1956] Ford, L. R., and Fulkerson, D. R. 1956. Maximal Flow Through a Network. *Canadian Journal of Mathematics* 8:399–404.

[2008] Fung, J., and Murray, T. 2008. Building CUDA Photoshop Filters for the GPU. Technical report, NVIDIA.

[2008] GIMP. 2008. GNU Image Manipulation Program. http://docs.gimp.org/en.

[1988] Goldberg, A. V., and Tarjan, R. E. 1988. A New Approach to the Maximum-Flow Problem. *JACM: Journal of the ACM* 35(4):921–940.

[1996] Golub, G. H., and Van Loan, C. F. 1996. *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)*. The Johns Hopkins University Press.

[2004] Grady, L., and Funka-Lea, G. 2004. Multi-Label Image Segmentation for Medical Applications Based on Graph-Theoretic Electrical Potentials. In Šonka, M.; Kakadiaris, I. A.; and Kybic, J., eds., *Computer Vision and Mathematical Methods in Medical and Biomedical Image Analysis, ECCV 2004 Workshops CVAMIA and MMBIA*, number LNCS3117 in Lecture Notes in Computer Science, 230–245.

[2005] Grady, L.; Schiwietz, T.; Aharon, S.; and Westermann, R. 2005. Random Walks for Interactive Alpha-Matting. In Villanueva, J. J., ed., *Proceedings of the Fifth IASTED International Conference on Visualization, Imaging and Image Processing*, 423–429. Benidorm, Spain: ACTA Press.

[2006] Grady, L. 2006. Random Walks for Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28(11):1768–1783.

[1985] Haralick, R., and Shapiro, L. 1985. Survey- Image Segmentation Techniques. *Computer Vision Graphics and Image Processing* 29:100–132.

[1990] Heckbert, P. S. 1990. *A Seed Fill Algorithm*. San Diego, CA, USA: Academic Press Professional, Inc. 275–277.

[2001] Hertzmann, A.; Jacobs, C. E.; Oliver, N.; Curless, B.; and Salesin, D. H. 2001. Image Analogies. In *SIGGRAPH ’01: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, 327–340. New York, NY, USA: ACM.

[1952] Hestenes, M. R., and Stiefel, E. 1952. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards* 49(6):409–436.

[2005] Intel. 2005. Discovering Multi-Core: Extending the Benefits of Moore’s Law. http://www.intel.com/technology/magazine/computing/multi-core-0705.pdf.

[2008] Intel. 2008. Processors - Intel® microprocessor export compliance metrics. http://www.intel.com/support/processors/sb/ca-023143.htm.

[1846] Jacobi, C. G. J. 1846. Über ein leichtes Verfahren die in der Theorie der Säcularstörungen vorkommenden Gleichungen numerisch aufzulösen. *Crelle’s Journal* 30:51–94.

[2003] Krüger, J., and Westermann, R. 2003. Linear Algebra Operators for GPU Implementation of Numerical Algorithms. *SIGGRAPH '03: ACM Transactions on Graphics (TOG)* 22(3):908–916.

[2003] Kwatra, V.; Schdl, A.; Essa, I.; Turk, G.; and Bobick, A. 2003. Graphcut Textures: Image and Video Synthesis Using Graph Cuts. *SIGGRAPH '03: ACM Transactions on Graphics (TOG)* 22(3):277–286.

[2000] Lee, K.-M., and Street, W. N. 2000. Automatic Image Segmentation and Classification Using On-line Shape Learning. *WACV '00: Fifth IEEE Workshop on Applications of Computer Vision, 2000* 0:64–70.

[2005] Lombaert, H.; Sun, Y.; Grady, L.; and Xu, C. 2005. A Multilevel Banded Graph Cuts Method for Fast Image Segmentation. In *ICCV '05: Proceedings of the Tenth IEEE International Conference on Computer Vision (ICCV'05)*, volume 1, 259–265. Washington, DC, USA: IEEE.

[2004] Mathews, J. H. 2004. Jacobi and Gauss-Seidel Iteration. http://math.fullerton.edu/mathews/n2003/GaussSeidelMod.html.

[2001] Moin, P. 2001. *Fundamentals of Engineering Numerical Analysis*. Cambridge University Press.

[1965] Moore, G. E. 1965. Cramming More Components onto Integrated Circuits. *Electronics* 38(8):114–117.

[1975] Moore, G. 1975. Progress in Digital Integrated Electronics. *IEEE International Electron Devices Meeting Tech Digest* 11–13.

[1995] Mortensen, E. N., and Barrett, W. A. 1995. Intelligent Scissors for Image Composition. In *SIGGRAPH '95: Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, 191–198. New York, NY, USA: ACM.

[2008] Munshi, A. 2008. OpenCL. http://s08.idav.ucdavis.edu/munshi-opencl.pdf.

[2007] NVIDIA. 2007. CUDA CUBLAS Library. http://developer.download.nvidia.com/compute/cuda/1_1/CUBLAS_Library_1.1.pdf.

[2008a] NVIDIA. 2008a. NVIDIA CUDA Compute Unified Device Architecture - Programming Guide. http://developer.download.nvidia.com/compute/cuda/2_0/NVIDIA_CUDA_Programming_Guide_2.0.pdf. Version 2.0.

[2008b] NVIDIA. 2008b. NVIDIA GeForce GTX 200 GPU Datasheet. http://www.nvidia.com/docs/IO/55506/GeForce_GTX_GPU_Datasheet.pdf.

[2008c] NVIDIA. 2008c. PhysX FAQ. http://www.nvidia.com/object/physx_faq.html.

[1998] Olano, M., and Lastra, A. 1998. A Shading Language on Graphics Hardware: The PixelFlow Shading System. In *SIGGRAPH '98: Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, 159–168. New York, NY, USA: ACM.

[2000] Peercy, M. S.; Olano, M.; Airey, J.; and Ungar, P. J. 2000. Interactive Multi-Pass Programmable Shading. In *SIGGRAPH '00: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, 425–432. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.

[1984] Porter, T., and Duff, T. 1984. Compositing Digital Images. *SIGGRAPH Compututer Graphics* 18(3):253–259.

[2001] Proudfoot, K.; Mark, W. R.; Tzvetkov, S.; and Hanrahan, P. 2001. A Real-Time Procedural Shading System for Programmable Graphics Hardware. In *SIGGRAPH '01: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, 159–170. New York, NY, USA: ACM.

[2004] Rother, C.; Kolmogorov, V.; and Blake, A. 2004. GrabCut: Interactive Foreground Extraction using Iterated Graph Cut. In *SIGGRAPH '04*, 309–314. New York, NY, USA: ACM.

[2000] Ruzon, M. A., and Tomasi, C. 2000. Alpha Estimation in Natural Images. In *IEEE Conference on Computer Vision and Pattern Recognition 2000*, volume 1, 18–25. Hilton Head Island, SC, USA: IEEE.

[2000] Schewe, J. 2000. 10 Years of Photoshop. *Photo Electronic Imaging* 16–25.

[2007] Sengupta, S.; Harris, M.; Zhang, Y.; and Owens, J. D. 2007. Scan Primitives for GPU Computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, 97–106. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association.

[2003] Sherbondy, A.; Houston, M.; and Napel, S. 2003. Fast Volume Segmentation With Simultaneous Visualization Using Programmable Graphics Hardware. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, 23. Washington, DC, USA: IEEE Computer Society.

[2007] Wang, J.; Agrawala, M.; and Cohen, M. F. 2007. Soft Scissors: An Interactive Tool for Realtime High Quality Matting. In *SIGGRAPH '07*, 9. New York, NY, USA: ACM.

[2005] Wang, J.; Bhat, P.; Colburn, R. A.; Agrawala, M.; and Cohen, M. F. 2005. Interactive Video Cutout. In *SIGGRAPH ’05*, 585–594. New York, NY, USA: ACM.

[2000] Wei, L.-Y., and Levoy, M. 2000. Fast Texture Synthesis Using Tree-Structured Vector Quantization. In *SIGGRAPH ’00: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, 479–488. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.