CONTOUR ENCODED COMPRESSION AND TRANSMISSION

by

Christopher Nelson

A thesis submitted to the faculty of

Brigham Young University

In partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

December 2006

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Christopher B. Nelson

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

_____   _____

Date   William Barrett, Chair

_____   _____

Date   Thomas Sederberg

_____   _____

Date   Eric Mercer

BRIGHAM YOUNG UNIVERSITY

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

_____
Date

_____
William A. Barrett
Committee Chairman

Accepted for the Department

_____
Parris K. Egbert
Graduate Coordinator

Accepted for the College

_____
Thomas W. Sederberg
Associate Dean,
College of Physical and Mathematical Sciences

ABSTRACT

CONTOUR ENCODED COMPRESSION AND TRANSMISSION

Christopher B. Nelson

Department of Computer Science

Master of Science

As the need for digital libraries, especially genealogical libraries, continues to rise, the need for efficient document image compression is becoming more and more apparent. In addition, because many digital library users access them from dial-up Internet connections, efficient strategies for compression and progressive transmission become essential to facilitate browsing operations. To meet this need, we developed a novel method for representing document images in a parametric form. Like other "hybrid" image compression operations, the Contour Encoded Compression and Transmission (CECAT) system first divides images into foreground and background layers. The emphasis of this Thesis revolves around improving the compression of the bitonal foreground layer. The parametric vectorization approach put forth by the CECAT system compares favorably to current approaches to document image compression.

Because many documents, specifically handwritten genealogical documents, contain a wide variety of shapes, fitting Bezier curves to connected component contours

can provide better compression than current glyph library or other codebook compression methods. In addition to better compression, the CECAT system divides the image into layers and tiles that can be used as a progressive transmission strategy to support browsing operations.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. William A. Barrett and the other members of my committee who have been patient with me throughout the past few years as this Thesis was drafted and provided aid when needed. I would also like to thank Michael Smith for allowing me the use of his code and getting this research topic started. I would also like to thank my wife Lydia for all her support and encouragement throughout this process.

Contents

List of Tables

x

List of Figures

Chapter 1

Introduction

1.1 Motivation

Ten years ago, when someone wanted to learn about a particular subject, they would typically travel to the local library or archive to find an appropriate book or periodical. With the advent of the Internet, however, this process has changed dramatically. In addition to the wealth of information that is growing daily on websites throughout the expanse of cyberspace, many books, newspapers, and periodicals have been scanned, indexed, and placed online to create "digital libraries" [1]. Now people can just log onto the Internet and go to one of these libraries to read through texts stored thousands of miles away. Even rare "special collection" documents become more valuable as the number of people with access to them increases [2].

To build up their collections, most current digital libraries scan documents, use Optical Character Recognition (OCR) to extract text, build transcripts, and publish these manuscripts on the web [3, 4, 5, 6]. This strategy works quite well when only textual information is involved. Unfortunately, this does not work for documents containing handwriting and important "non-textual" information. Document properties such as ink color, paper texture, drawings, and font information can be as important as text, especially for those of historical significance [7] as demonstrated in Figure 1.1. To publish these documents on the Internet, digital libraries must use images instead of simple text-based transcripts [8, 9, 10, 11].

Genealogical documents often fall into the category described above. These documents cannot be stored as simple text transcripts without losing some of their value and recognizing handwriting is outside the scope of current OCR engines. In many cases,
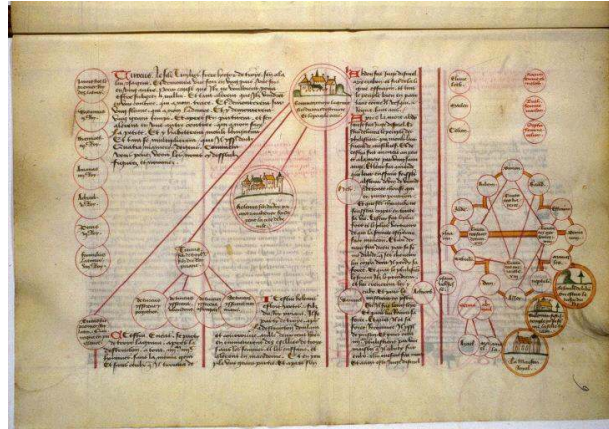
Figure 1.1 Sample Document Images. (a) Illuminated French Text (b) Illustrated French Renaissance Document

these are old, historical documents containing large amounts of handwritten text. Although most of the content is intended to be bitonal (i.e. black and white), grayscale information does provide clues to help viewers understand the document. In addition, these document images should be stored at high resolutions (200-400 dpi) to allow the scanned image to be a faithful representation of the original and improve readability.

Finding a needed genealogical document can be a challenge, especially if it is located inside a collection that has not been indexed. In this case, finding a particular document requires the researcher to scan through a collection of documents as quickly as possible, looking for specific names or dates. This process is commonly referred to as "browsing" [12]. Unfortunately, many Internet users still use 56K modem (dial-up) connections [13]. Even with higher bandwidth, waiting for large images to download can be a very exasperating exercise, especially when the image being downloaded does not contain the information needed. Trying to "browse" through numerous genealogical documents using low-bandwidth network connections is unacceptable.

To alleviate this problem, strategies such as image compression and progressive transmission can be used. Improving image compression is the most obvious optimization: smaller image files sizes result in shorter download times. The challenge facing the many image compression strategies lies in the fact that increased compression ratios often result in the loss of some important image data.

The second strategy, progressive transmission, is the process of taking a large

image and sending it over the Internet in small pieces. In some cases, coarse images are sent first, giving the researcher a general idea about the contents of the image. If progressive transmission sends image pieces at full resolution, the researcher can begin to read through the pieces of the image that have already been sent while waiting for more to arrive. This makes "browsing" through numerous documents much quicker, especially if the first few image pieces contain the names or dates needed by the researcher [12].

1.2 Solution: Contour Encoded Compression and Transmission

This thesis presents a system called "Contour Encoded Compression and Transmission" (CECAT) which uses image compression and progressive transmission to improve browsing operations for document images. Although any grayscale document image can be used, the algorithms created for the CECAT system were specifically designed to efficiently compress and transfer images containing handwriting.

CECAT breaks an image into three layers: foreground (bitonal text), residual (grayscale text), and background. The emphasis of this thesis and the CECAT system lies in developing an efficient compression of the bitonal foreground layer. This is done by detecting contours for the text and handwriting, replacing these contours with parametric curves, and storing these contours in tiles that can be transmitted progressively. This approach has the following advantages:

- Good, scalable image compression with the potential for lossless compression as the final step in the progressive transmission
- Progressive transmission of full resolution tiles with readable resolution in the handwriting
- High level curve data that can be used for subsequent pattern recognition

Chapter 2

Background

The CECAT system combines two technologies – document image compression and progressive transmission – to facilitate document image "browsing" suitable for even slow network connection speeds. This chapter will review these two technologies.

2.1 Document Image Compression

Image compression, a very active field of research, is the process of taking image data and converting it into a more compact form. This process, known as "encoding" reduces the size of an image by storing the data more efficiently. "Decoding" is the process of taking this compact form and changing it back to a viewable format.

Image compression saves storage space and thus reduces the required download time for retrieving images across the web. For example, a document image showing a single page from the 1870 U.S. Census stored at a resolution of 200 dots-per-inch (dpi) is about 15 megabytes in size in its raw, uncompressed form. Given dial-up network speeds, it would require over a half hour (15 MB at 56 Kbps = 36.6 minutes) to download this image. In addition, only 45 such images could be stored on a standard CD-ROM.

On the other hand, after applying a standard JPEG compression with a quality rating of 75 to this 15 megabyte image, the file size drops to about 800 kilobytes. As a result, download time and storage space shrink to about 5% of that required for the uncompressed image. This corresponds to a download time, over a standard dial-up connection, to a little less than two minutes and over 890 images can be stored on a CD-ROM.

Compression does not come without a cost. First, time is required time to encode and decode compressed images. For example, using a 2.39 GHz Pentium, performing
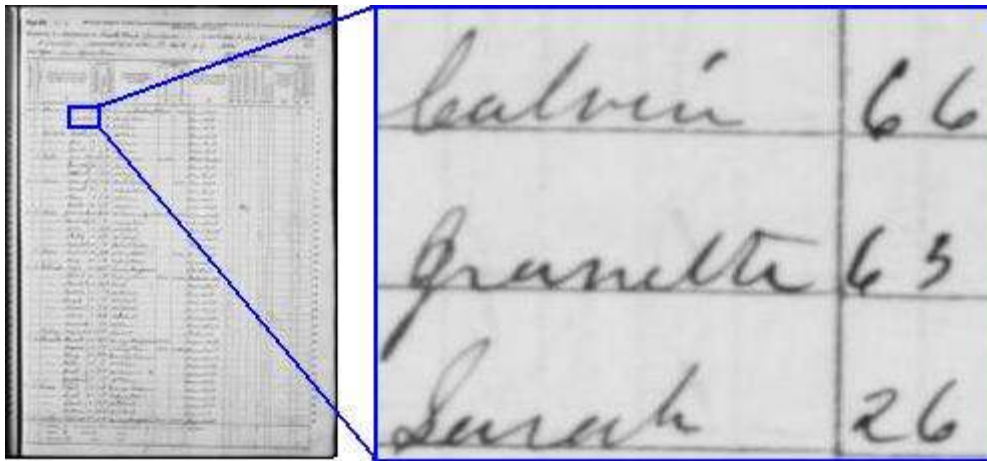
Figure 2.1 200 DPI Image from the 1870 U.S. Census

JPEG compression and storing a large image of 3400 x 4600 pixels takes about 2.62 seconds. This example image is shown in Figure 2.1. Second, compression can degrade the image. The degree to which this occurs depends on the image compression strategy used. Compression algorithms that do not alter the image are called "lossless" and those that alter the image are called "lossy". Lossy compression strategies often throw away pieces of data that are deemed unimportant or that may not be noticeable to the human eye, such as in the JPEG example mentioned earlier. Lossy compression strategies generally reduce the image file size to a fraction of the size of their lossless counterparts. Most image compression operations follow one of four encoding strategies: transform, context, dictionary, and hybrid. These are reviewed in the following subsections.

2.1.1 Transform Encoding

Transform encoding techniques work by converting raw image data (an array of three 8-bit color values for each pixel) into another format such as those created by applying a Discrete Cosine Transform, Fourier Transform, Wavelet Transform or similar transforms. This transformed data is an accurate representation of the image, except color values are replaced by points or waves in an alternate spectrum. Some of this transformed data has very little (if any) effect on the image after it is transformed back, and can be removed, making the image smaller without changing much of the original image. When decoded, the image data is transformed back for display purposes.

Figure 2.2 JPEG Compression Artifacting

The JPEG standard used to deliver images across the Internet uses a DCT encoding to transform image data into the frequency domain. Sharp changes in color (such as black letters touching white paper) require more transform coefficients to represent the image in the frequency domain. As a result, JPEG compression works very well for continuous-tone images like pictures and photographs but creates artifacts in document images [7]. Sharp-edges contain "ringing" after images are decoded from JPEG format, making JPEG encoding a conspicuous example of "lossy" image compression as shown in Figure 2.2.

In addition to the DCT, other transformation strategies have emerged during the past few years. By transforming image data into a Wavelet spectrum, the new JPEG2000 standard can create higher-quality images than the JPEG standard [24]. More Wavelet-based transforms are emerging, including the proprietary IW44 [16] strategy used in the popular DjVu compression standard.

2.1.2 Context Encoding

Context encoding encompasses a range of compression strategies that use redundant information from groups of the pixels to reduce the size of the image. These strategies represent a "neighborhood" of pixel data with a single piece of data. Run-length encoding uses a "neighborhood" along one row of pixels to compress an image. In its simplest form, run-length encoding strategies replace a series of similar bits (or pixels) (11111111111111111111) with a count of how many are on (20 1s), and their values. In this case, the "neighborhood" represents a pixel and all nineteen preceding it.

Although context encoding strategies do not compress images as well as other encoding strategies, they are very fast to encode and decode. For this reason, one popular
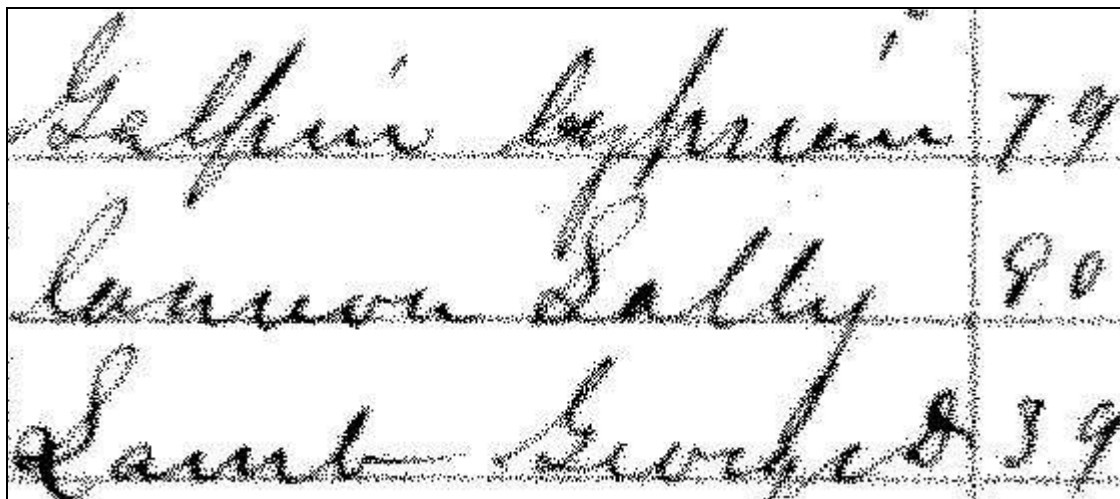
Figure 2.3 JBIG Encoded Image Slice

run-length encoding strategy is the CCITT standard, which is used for sending and receiving faxes [14]. The CCITT standard operates in two-dimensional mode using differential run-length encoding of the difference between the current and previously sent lines. By taking advantage of the similarities inherent between adjacent lines, CCITT can achieve fast, reliable compression without processing the entire image. In this example, the context of the last line is used to improve the encoding of the line following it.

Another well known context encoding strategy is JBIG, an older standard for compressing bitonal images [14]. This context compression strategy uses lower resolution copies and an approach similar to the CCITT standard to compress each image. The lowest resolution layer, known as the base layer, is encoded using one of many resolution reduction algorithms that, for example, reduce an image from 200 dpi to 100 dpi. JBIG also implements progressive transmission by sending a low resolution copy first, then sending higher resolution layers, called differential layers. A small section of a JBIG encoded image is shown in Figure 2.3. The dot patterns in the JBIG image are used to represent various levels of gray using only a bitonal image.

2.1.3 Dictionary Encoding

Dictionary compression strategies collect sequences of pixels (or symbols) from an image and store them into an indexed dictionary. These sequences can range in size from a couple of pixel values to complicated connected components like typed letters. In

8

some cases, even full-color image tiles could be using as symbols in a dictionary. Once this dictionary has been built, symbols found on the image are converted from raw pixel data to indices referencing the dictionary. If the same symbol shows up many times in an image, good compression can be achieved as the actual pixel representation for that particular symbol need only be stored once (inside the dictionary) [14].

A good example of a general-purpose dictionary compression strategy an "entropy" encoding strategy known as Huffman encoding. This compression strategy takes ordered data and replaces frequently occurring sequences of data with indices to a dictionary organized in a binary tree. By giving the most common sequence of pixels the smallest index, this strategy can compress any kind of data. Although Huffman encoding works best when compressing series of actual symbols such as text files, good compression can be obtained in image data as well.

JBIG2 and JB2 [15] standards are examples of dictionary-based compression designed specifically for compressing bitonal images. The dictionaries created by these compression strategies contain connected black components. These compression strategies perform well, especially for documents containing machine printed characters. By replacing each letter with a small index number pointing to one in the library of glyphs, compression levels up to 100:1 or more can be achieved. These bitonal image encoding strategies are discussed in Section 2.2.1.

The limitations of dictionary encoding strategies depend on two things: the size of the dictionary and the size of the indices to the dictionary. When an image is encoded and stored, the dictionary must be kept along with the actual image data, making the dictionary part of the total file size. When too many symbols are stored, the dictionary and its indices can become large. In some cases, it is possible for the index to a shape to become larger than the actual shape itself. In extreme examples, images can become larger after compression, thereby defeating the purpose.

2.1.4 Hybrid Encoding

Hybrid image compression strategies have sparked considerable interest during the past few years. By splitting images into layers and using different compression operations for each layer, high compression can be achieved. For most hybrid strategies,

(a)

(b)
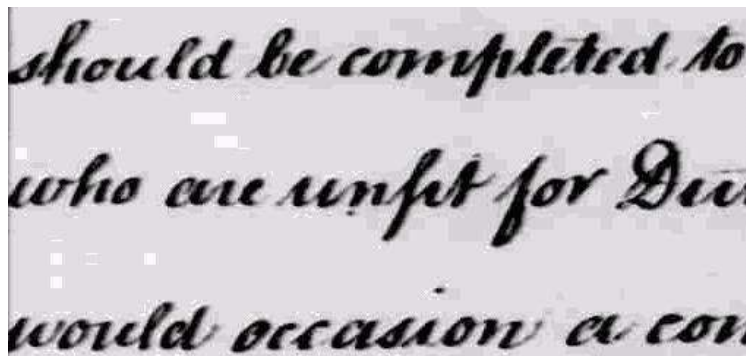
(c)

Figure 2.4 DjVu Image Layers created from JPEG Source Image. (a) Bitonal Foreground Mask. (b) Foreground Mask combined with Color Map. (c) All DjVu Layers Combined

images are divided into a foreground and a background layer. The foreground layer is a bitonal image containing all the printed and handwritten text and simple drawings. The background layer is a continuous tone grayscale/color layer containing pictures and textured surfaces.

The compression operations applied to each layer are chosen to take advantage of

10

the nature of the layer. The foreground layer is often compressed with a dictionary-based bitonal compression strategy. A transform compression strategy is usually used on the background layer. By applying different compression strategies specialized for each layer of the image, higher compression can be achieved than by applying the same compression strategy to the whole image.

The popular DjVu hybrid strategy converts an image into a high resolution (300 dpi) bi-tonal foreground mask, a small color map referenced by the foreground mask, and a lower resolution (100dpi) continuous-tone color background image [16] as shown in Figure 2.4. The foreground mask is compressed with JB2, a dictionary encoding scheme implementing the JBIG2 standard. The background image is compressed with IW44, a wavelet-based transform encoding algorithm similar to JPEG2000. Other examples of hybrid image compression are Microsoft's SLIm [17], DigiPaper [18], and DEBORA [7].

2.2 Bitonal Image Compression Strategies

Because they do not contain extraneous shade of color, bitonal images can be compressed at much higher rates than grayscale or color images. Pixels require eight bits for an accurate representation in a grayscale image and twenty four bits for a color image. Bitonal images use a single bit per pixel, which provides a large reduction in image size without any extra compression. In addition to taking advantage of the "one bit" nature of each pixel, bitonal compression strategies use techniques such as pattern matching or vectorization to further compress images.

2.2.1 Pattern Matching

Pattern matching is a form of dictionary-based image compression using connected components as symbols. For example, the JBIG2 standard uses pattern matching. When a pattern matching strategy is used, the compressor analyzes the image and creates a dictionary of commonly repeated patterns (pixel-by-pixel symbols). As a result, the data stored in the image file are simply indices to entries in this dictionary. If the entry does not match the current pattern exactly, the residual difference in encoded using common bitonal image compression techniques [19].

Patten matching algorithms come in two flavors: soft pattern matching and

pattern matching and substitution [20]. In pattern matching and substitution, if a symbol is similar to one already stored in the library but not quite close enough for a match, a new symbol must be added to the library. In soft pattern matching, the difference (delta) between the symbol in the library and the one on the image is preserved instead [21]. Pattern matching works best when a document consisting of many images can be referenced by only one dictionary. In some cases, the dictionary can be larger than the actual image data, thus, using the same codebook for a collection of images is a way to leverage greater compression efficiency [19]. Unfortunately, because of the variability in handwritten document images, this technique can not be employed effectively.

2.2.2 Vectorization

Vectorization is the process of converting an image from pixel data (often called "raster" format) into a vector-based file format. In its simplest case, a vector image is a collection of line segments. For example, take an image containing one black line from the upper-right corner of the image to the lower left. Instead of using one bit for each pixel in the image, a vectorized copy of this image only stores the two endpoints and lets the decoder plot the actual line.

In addition to image compression, vectorization has other advantages that make it attractive. First, by converting raw pixel data to "higher order" data like lines, curves, and shapes, it is much more feasible to perform pattern recognition or other computer vision operations on the data. For handwritten text, vectorized letters provide a good feature set for handwriting recognition. Second, vectors are represented by a sparse collection of points, which can be used to perform various affine transformations (rotation, scaling, and translation) on the image. Instead of manipulating the whole image, these transformations can be limited to the points defining the vectors.

Basic vectorization techniques are divided into two categories: thinning and nonthinning [22]. A thinning operation finds the midpoints of raster-based lines and shapes and converts them into vectors. Because the shapes of varying thickness are replaced by single pixel lines, this operation is referred to as creating a "skeleton" of the image [23, 37]. Each vector has a specific width assigned to it, allowing lines of various widths to be rendered accurately. Nonthinning operations use contours or the pixels

detected along the edge of each shape to represent the raster image.

Vectorization is used to convert engineering or architectural diagrams from scanned images into a clean, elegant form composed of line vectors, giving engineers the ability to manipulate the images easily using the aforementioned affine transformations. Unfortunately, absolute pixel-by-pixel vectorization is quite expensive (although preferable for engineering diagrams mentioned earlier). For document image compression, vectorization is usually a lossy operation. Fortunately, vectorization tends to smooth letter and shapes, including the curves associated with handwriting. This can improve the readability of a document image.

2.3 Progressive Image Transmission

Progressive image transmission is the process of transmitting images piece-by-piece across a network, so users with slow network connections can browse the image without having to wait for the whole image. By sending the image in small chunks, it is even possible for a user to finish reviewing the image or extract the needed information before the whole image has been downloaded.

Current Internet browsers rarely perform progressive transmission by default. In most cases, images are replaced by "alternate" text or an icon of a broken image until the entire image is downloaded. At this point, the image suddenly appears in the browser. Even if a "progressive" transmission strategy is activated for JPEG images, a raster-based image is rendered row by row from top to bottom [12]. Although this is a progressive transmission strategy, it only supports browsing if the data the researcher wants is at the top of the image.

There are two approaches or issues to progressive image transmission: quality and content. In "quality progressive transmission", images are initially sent to the user at a low resolution, with the resolution increasing as more data arrives. The JPEG standard supports this using a "Progressive DCT-based Mode" which streams coarser images to view first, improving the image by sending subsequent data [24]. For bitonal images, the JBIG standard also supports a low-to-high resolution image transmission strategy using base and differential layers [14]. The Just-In-Time-Browsing (JITB) uses the JBIG standard by sending multiple bit-planes to the browser with each one adding different

colors values to the image. As more bit-planes arrive, the image is further refined [12].

Unfortunately, this coarse-to-fine strategy does not always work well for document images. To be useful for a researcher, a document must be readable. Low resolution images tend to leave fuzzy or blocky edges on handwritten and printed text. In many cases, although sections of a coarse image can be quickly identified as text, separate letters may be impossible to distinguish.

"Content progressive transmission", on the other hand, sends full resolution image pieces to the researcher one-by-one. In some cases, these pieces are layers such as the background and foreground layers used by hybrid compression strategies (Section 2.1.4). Other content progressive strategies involve chopping images into tiles and sending these one at a time.

"Content progressive transmission" is the approach used by DjVu for its transmission strategy. DjVu separates images into multiple layers [25]. The foreground layer, consisting of text and darker sections of the document image, is sent to the user first. Only after the foreground layer has been sent does the background layer start to be sent to the user [16].

2.4 The CECAT Approach

The CECAT system provides a novel approach to the problem of document image compression as well as a progressive transmission strategy. The CECAT compression strategy is a "hybrid" compression strategy optimized for the bitonal foreground layer. By converting this bitonal layer into a collection of contours represented by parametric curves, the CECAT system uses vectorization for compression. As mentioned in Section 2.2.2, this vectorization prepares the image for future "higher-order" data manipulations. The progressive transmission strategy provided by the CECAT system is a mixture of two "content progressive" approaches. Like other "hybrid" approaches, the foreground layer is sent first, followed by a residual and a background layer. In addition, the CECAT system divides each layer into tiles that can be sent to the use one-by-one.

14

Chapter 3

Contour Encoded Compression

The main emphasis of this thesis and the CECAT compression strategy is creating an effective method for compressing the foreground bitonal layer of a document image. This section will cover the vectorization process used to convert image data from pixel values to parametric curves, while Section 4 will discuss the encoding format of this and the other grayscale image layers. Using parametric curves to represent contours surrounding the black shapes in the image reduces the image size considerably. The value this has for facilitating browsing is obvious: smaller file size equals shorter download time.

To accomplish this, the image is first converted from color to grayscale, followed by a binarization operation (Section 3.1). This creates a bitonal, or black-and-white, image. The pixels surrounding each of the shapes in the image are then detected and labeled as contours. This detection operation and its associated contour filling operations are presented in Section 3.2. Next, parametric curves (curves defined by two or more "control points") are fitted to each of the contours using a process discussed in Section 3.3. Lastly, these parametric curves are saved for later compression operations discussed in Chapter 4.

3.1 Binarization of Document Images

Like any other foreground/background or "hybrid" compression strategy, document images must be converted from color or grayscale to black and white (or bitonal) images. This process, also known as "binarization", is one of the more difficult challenges in the field of document image processing. Because image quality varies

among document images, no one strategy works best. Also, poor binarization can cause important portions of a document image to be lost. The effectiveness of the CECAT compression strategy hinges on selecting a good binarization strategy.

3.1.1 Color to Grayscale

The initial step, before binarization can take place, is the simple and well-documented process for converting color images into their appropriate grayscale representations. In the most common color representation, colored pixels are represented by three 8-bit intensity values for the colors red, green, and blue. Every grayscale value (from pure black to pure white) can be represented by a single 8-bit intensity value. As a result, converting a document from color to grayscale reduces an image size by about 66%. By applying Equation 3.1 to each pixel in the image, color pixels are easily converted into their grayscale equivalents [26].

$$Gray = 0.3 * Red + 0.59 * Green + 0.11 * Blue \qquad (3.1)$$

3.1.2 Grayscale to Bitonal

Now that we have a grayscale image, the binarization process can begin. The goal of this operation is to separate the black text from the rest of the document image. For the CECAT system, binarization is accomplished using a local thresholding algorithm. Although the development of an "optimal" binarization algorithm remains an area of active research, the algorithm proposed by Niblack in 1985 remains very competitive with current approaches [27]. For the CECAT system, a modified version of the Niblack thresholding algorithm is used. This modification was proposed by Zhang and Tan [35] and adds two constants to reduce the algorithm's sensitivity to noise. This approach was implemented by Mike Smith for a class project at BYU in 2004 and performs reasonably well for testing the CECAT system [28].

This binarization algorithm is a "local" thresholding operation because it creates a threshold that can be different for each pixel in the image. If a pixel is greater than the threshold value, it is changed to white; otherwise, the pixel is changed to black. Niblack thresholding takes the mean (μ) and the standard deviation (σ) of the area around each

pixel and factors in two empirical constants (R and κ) to create a threshold T(x, y) as described in Equation 3.2 [36].

$$T(x, y) = \mu \left[1 + \kappa \left(1 - \frac{\sigma}{R} \right) \right] \qquad (3.2)$$

For the CECAT system, the area used to create this threshold is a 19x19 square region around each pixel. The value for κ, which adjusts the amount of boundary that should be added to each black shape in the image [36], is set to -1. This removes extra "padding" around the detected shapes. The other constant, R, is set to 100.

Even with this algorithm, the binarization doesn't always perform well, especially on some of the difficult documents analyzed. As an added measure, we added a simple global minimum to the thresholding logic. If any pixel falls below this minimum value, the system designates it as a white pixel, independent of *T(x, y)*. This allowed us to test the CECAT compression system on poor quality documents by tuning the thresholding algorithm globally for each set of documents. This value is set to different values ranging from 128 to 170, depending on the quality of the collection.

3.2 Contour Detection and Rendering

A contour is an ordered list of pixels making up the outside edge of a shape. In Figure 3.1, the yellow line marks the pixels that make up a contour. Because the contour lies on the shape it represents, it is called an internal contour. If we have the contour, we can recreate the shape that it represents. Using contours instead of actual space-filling shapes is how CECAT images are compressed and rendered.

To use contours in image compression, two issues must be addressed. First, we must have a process that identifies the contours. Second, to transform contours from simple lines into human readable shapes, a contour-filling operation is needed. Although many algorithms can be used to accomplish these operations, it is important, for our purposes, to select two strategies that complement each other.

3.2.1 Layered Contour Detection

Using a bitonal image, it is possible to detect and mark the contours for each
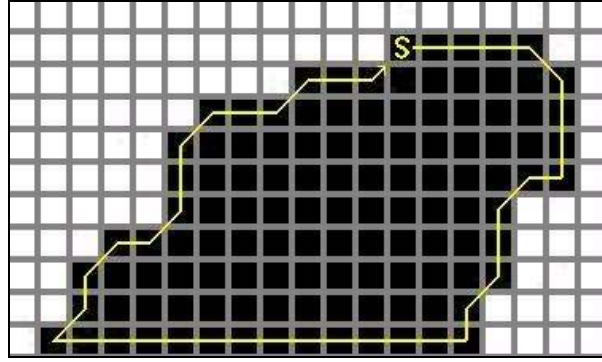
Figure 3.1 Contour Example.

shape, referred to as a connected component. For effective compression, we need to mark the pixels along the inside edge of each shape, creating an "internal" contour. When the shape is decompressed, the pixels that make up the contours become part of the shape. Following this rule is especially important for recreating shapes that are one or two pixels wide. Figure 3.1 shows an example of the "inside" edge our contour detection algorithm is trying to find.

For our purposes, a simple counter-clockwise turn recursive contour detection algorithm is used. Because we want to represent the contours with the smallest number of pixels possible, the contour detection strategy looks for eight-connected components (the contours can go diagonally as well as horizontally and vertically). The basic algorithm used for tracing an eight-connected component contour is shown on the next page.

Although this strategy will find the edges of the black connected components in an image, it fails to identify any of the white "holes" inside these black components. To capture all the necessary contour information, a strategy to detect these white "holes" is also needed. In addition, these contours must be sorted in such a way as to preserve their nested relationship, so that encompassing components are not rendered after any of their internal connected components, overwriting them in the process.

To achieve these goals, the contour detection operation works on one "layer" of the image at a time. First, an image, as shown in Figure 3.2a, is analyzed and the contour detection algorithm is used to find the outside of each contour. Figure 3.2b shows the contours detected using this operation. Once a contour has been detected, every pixel on,

20
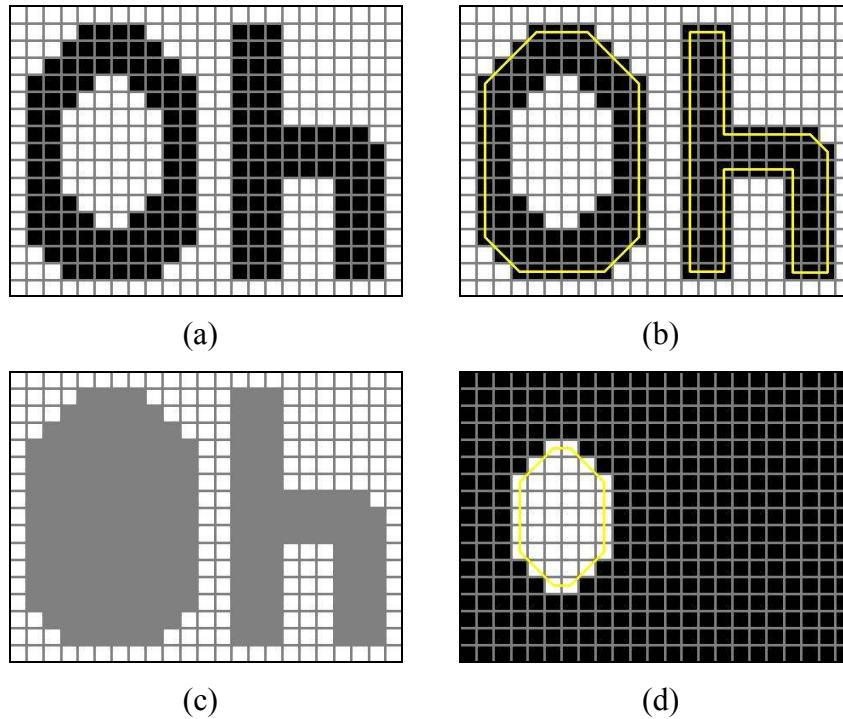
Figure 3.2 Contour Detection Example. (a) Original Image (b) Detected Contours in the First Layer (c) Filled Image Mask (d) Second Layer after Rendering Mask

procedure TRACECONNECTEDCOMPONENTS

1: **Inputs:**
2:    Point s*tart_point* {black pixel found to right of a white pixel}
3: **Outputs:**
4:    Array of points[] *contour* {sequence of points making up the contour}
5: **Variables:**
6:    Point *curr_point* {marker for the current position on the contour}
7:    Enum *direction* {"north", "northwest", "west", "southwest", "south"…}
8:    Integer *num_turns* {number of 8-compass point turns made from *curr_point*}
9: **Begin**
10: *curr_point* = *start_point*
11: *direction* = "northwest"
12: **do**
13:   *num_turns* = 0
14:   **while** Pixel in *direction* from *curr_point* is "white" AND *num_turns* < *8* **do**
15:     *direction* = next clockwise 8-point compass direction
16:     *number_turns* = *number_turns* + *1*
17:   Add *curr_point* to *contour*
18:   *curr_point* = next Pixel in the *direction* from *curr_point*
19:   **if** *num_turns* = 8 **then**
20:     *curr_point* = *start_point*
21:   *direction* = 3 steps counterclockwise on 8-point compass direction
22: **while** *curr_point* != *start_point*
23: **End**

and inside the contour is changed to gray using the contour filling algorithm described in Section 3.2.2. After detecting and filling all these contours, we have an image like the one Figure 3.2c.

Detection of the first contour layer is now complete. Next, the second contour layer makes up the "holes" in these first contours, appearing as white shapes on a black background. To prepare this layer for the contour detection operation, we first create a blank image of the same size as our original image with all the pixels set to black. Then, using the gray image created earlier as a "mask" on the original image (Figure 3.2a), we add all contents of the previously detected contour layer. This includes the white contours that make up this second contour layer. Once all this is done, we have an image like the one in Figure 3.2d.

By simply reversing the foreground and background colors in the contour detection operation, finding white contours on the black background of this new image is straight forward. This creates counterclockwise contours that make up the second layer. By filling these contours and repeating the process (simply swapping the background and foreground colors each time), we can find all the contours, no matter how many nested shapes there are. As an added bonus, these contours are sorted in the order we need to render them.

Figure 3.3, on the next page, shows a portion of a census image divided up in these layers. Because of all the nested shapes, four different contour layers are required (shown as Figures 3.3b – 3.3e). When the image is displayed, the first layer (Figure 3.3b) is rendered first. By adding each additional layer one-by-one, the internal contours are drawn last, preventing one contour from overwriting another.

3.2.2 Contour Filling Algorithm

Contour filling is the well-documented image processing problem of changing a contour into its associated shape by setting the color of all the pixels inside the contour to the color of the contour itself. Accurately performing this operation is essential for the layered contour detection strategy mentioned earlier, as well as acting as the final step in the process that converts encoded contours into a readable image.

Every contour-filling algorithm makes some assumptions, many of which do not

(a)

(b)

(c)

(d)

(e)

Figure 3.3 Contour Layers for Sample Image. (a) Original Image (b) First "Black" Layer (c) Second "White" Layer (d) Third "Black" Layer (e) Fourth "White" Layer

work for encoded contours that may contain errors (as created by CECAT encoding). For example, because the "inside" edge of each shape is used as a contour, the area inside the contour is sometimes disconnected white as shown in Figure 3.4a. A "flood fill" strategy, which changes one white pixel to black and recursively applies the same operation to all the white pixels adjacent to that pixel, will only fill half the shape.

Another popular contour filling method follows the contour around the outside edge in a clockwise direction. The left edge of the contour can be identified as locations where the contour is moving up. By filling the contour in a "scan-line" from these points to other contour edges on the right, the contour can be filled very quickly. Unfortunately, the process of mapping parametric curves to contours sometimes introduces slight errors
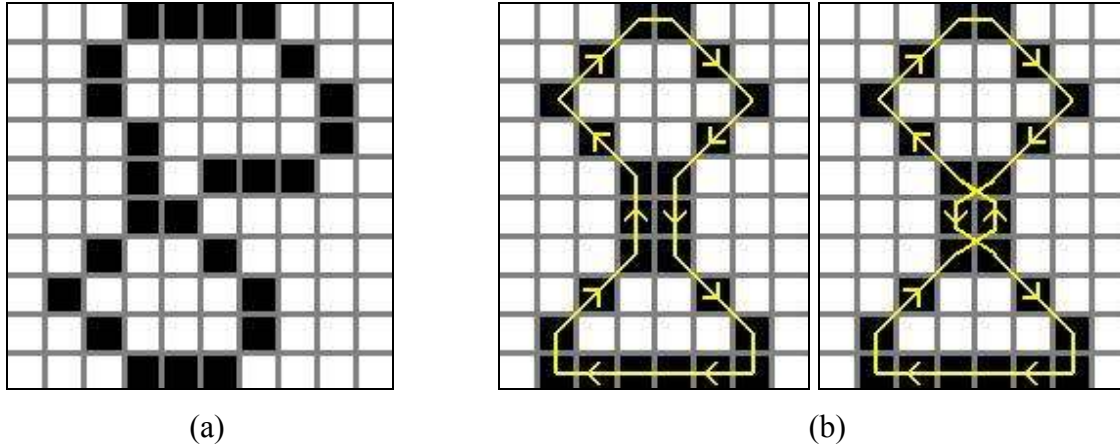
Figure 3.4 Challenges for Contour Filling. (a) Unconnected Contour Area (b) Transposed Edges

procedure FILLCONTOUR

1: **Inputs:**
2:    Array of points[] *contour* {sequence of points making up the contour}
3: **Outputs:**
4:    Array of bits[][] *canvas* {pixel values as they appear after contour fill}
5: **Variables:**
6:    Array of points[] *spans* {leftmost edges of each scan-line made by *contour*}
7:    Array of bytes[][] *grid* {plots points in *spans* and marks filled pixels}
8:    Integer *total* {running sum of labels on a given row}
9: **Begin**
10: **for** $i \leftarrow 0$ to *contour.length* **do**
11:   **if** *contour[i]* is left-most pixel of a horizontal row of pixels **then**
12:     add *contour[i]* to *spans*
13: *grid* = byte[*contour.width*][*contour.height*]
14: **for** $j \leftarrow 0$ *spans.length* **do**
15:   **if** *spans[j]* is a local minima or maxima **then**
16:     // Do Nothing
17:   **else**
18:     *grid[span[j].x][span[j].y]++*
19: **for** $k \leftarrow$ *grid.minY* to *grid.maxY* **do**
20:   *total* = 0
21:   **for** $l \leftarrow$ *grid.minX* to *grid.maxX* **do**
22:     **if** *total* is odd **then**
23:       *canvas[l][k]* = 1 {fill in the pixel point on the resulting canvas}
24:     *total* = *total* + *grid[l][k]*
25: **for** $m \leftarrow 0$ to *contour.length* **then** {fill the points along the contour}
26:   *canvas[contour[m].x][contour[m].y]* = 1
27: **End**

24

resulting in the edges being swapped as shown in Figure 3.4b, causing this contour filling strategy to fail as well.

The contour filling algorithm used by the CECAT system is similar to the scan-line parity based fill method mentioned earlier. Using contours stored as an array of x and y coordinate pairs and a small byte array to map which pixels need to be filled, this algorithm accurately fills each contour. The details for this algorithm are outlined in the pseudocode on the previous page.

As a first step, each horizontal row of black pixels in the contour is changed into a single point corresponding to the leftmost pixel of the row. These collection of points will be used later to mark the pixels that need to be filled. In the code below, these points are called *spans* and have been marked blue in an example shown in Figure 3.5a. The code shows this operation on lines 10-12.

Once these *spans* have been identified and marked, the algorithm steps through the contour again, counting how many times these spans are crossed. This information is stored on a byte array called a *grid*. Once the count has been made, any span found to be a local minimum or maximum (i.e. spans before or after are both above or below) are removed from the grid. This analysis occurs on lines 13-18 in the code and Figure 3.5b displays the count inside each marked pixel with the local minimum/maximum crossed out.

Now that the grid has been created, the actual "contour filling" process takes place. This operation, outlined on lines 19-24 of the code, is a simple scan-line parity fill. While moving from left to right along each row in the grid, each time a number value is crossed, a *total* variable is incremented by that amount. Whenever this *total* is odd, any pixel passed over is "filled" in the image (called *canvas* in the code). Figure 3.5c shows the results of applying this operation. At this point, all the pixels found inside the contour have been marked as filled. As a final step, the algorithm goes through the contour a third time and fills every point on the contour (lines 26-27). This concludes the contour filling operation. Figure 3.5d shows the final filled contour.

We saw minor performance enhancements by using the list of horizontal objects to represent *spans* instead of plotting everything onto the *grid* in the first place. This algorithm requires the whole image section to be stored in memory, but this is not much

(a)   (b)

(c)   (d)
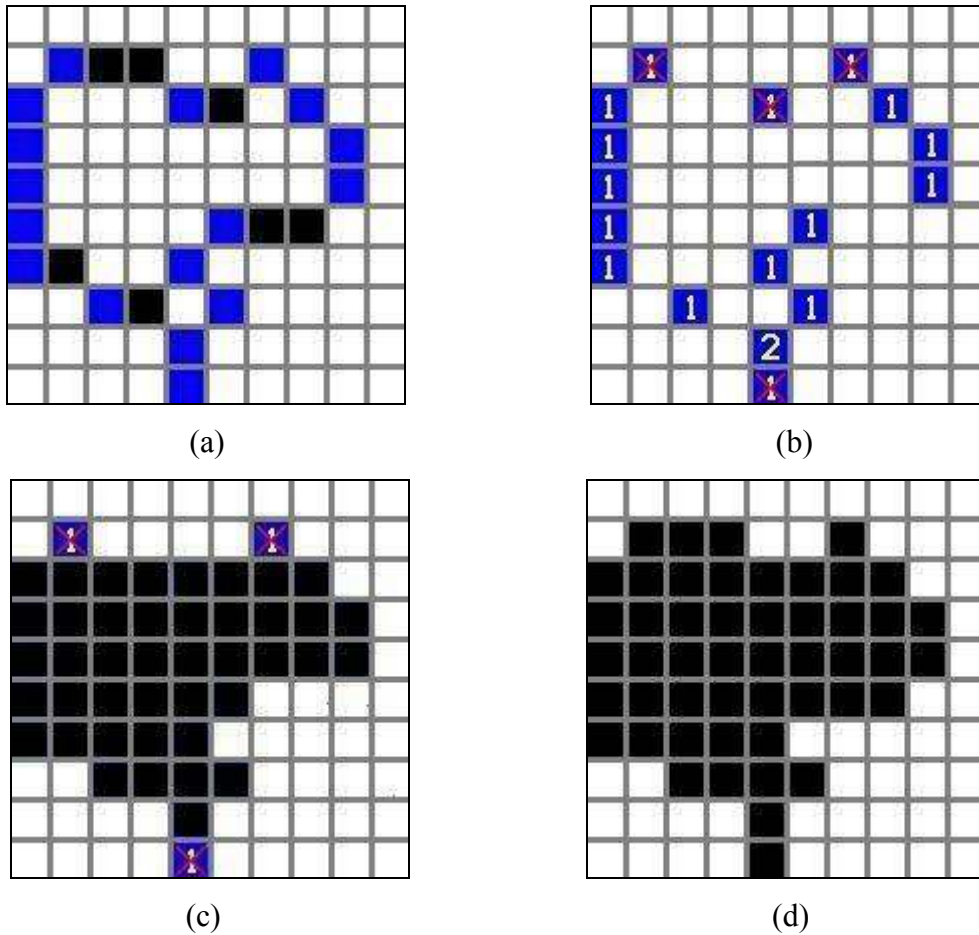
Figure 3.5 Contour Filling Algorithm Example. (a) Horizontal Spans Marked Blue
(b) Span/Contour Crossing Points Counted and Local Minima/Maxima Removed
(c) Filled Contour using Scan-line Parity (d) Completely Filled Contour

of an issue due to the CECAT localization tactic of chopping up images and the
connected components associated with them into 512 x 512 tiles (see Section 4.1 for
more details). Because of this, the connected components associated with these contours
do not grow too large for memory to be an issue.

3.3 Fitting Parametric Curves to Contours

One of the major contributions of this thesis is the process of converting contours
from an ordered list of pixel points into a collection of parametric curves. This
conversion has three major benefits: improved compression, componentization, and a
higher-order representation than raw contours.

Compression is the most obvious reason for changing the image format into a piecewise parametric representation. Instead of storing a list of points making up a straight line, it is much more efficient to simply store the two endpoints and note that they represent a line. By combining one or two more control points and a mapping equation, a few points can represent a curve which can be used to represent a section of contour even more efficiently than a collection of line segments. With parametric curves, otherwise complex shapes can be represented with a few control points instead of a list of points labeling each pixel on the contour one-by-one.

Unlike compression strategies that transform an image into another representation before compressing them (such as the discrete cosine transform used by JPEG), the control points used to represent each contour remain in XY-coordinate space. Because of this, contours can be sorted or chopped into smaller pieces using a process known as componentization. This allows the CECAT format to be used in a variety of progressive transfer strategies. In addition to componentization, changing contours into lists of control points allows for much faster scaling, rotation, and translation. Instead of applying an image-wide operation, only the control points need to be changed when performing these affine transformations.

Lastly, parametric curves provide a higher-order representation of the original contours. As such, these curves can be used in subsequent pattern recognition algorithms or further compressed by using a library of common curves. CECAT image provide a new, higher order feature space for solving computer vision and image processing problems.

3.3.1 Bezier Curves

For the initial implementation of the CECAT system, Bezier spline curves are used as the parametric form for representing contour segments. Named for the French Mathematician Pierre Bezier who discovered them in the 1960s, this parametric representation provides a simple way to define n-degree curves using $n + 1$ control points [29]. Although one of the least sophisticated of the parametric curves, Bezier curves provide an accurate and elegant way to compress curve data. They have been used by many different drawing programs throughout the past few decades.

Line: $$p(u) = (1-u)p_0 + up_1 \qquad (3.3)$$

Quadratic: $$p(u) = (1-u)^2 p_0 + 2u(1-u)p_1 + u^2 p_2 \qquad (3.4)$$

Cubic: $$p(u) = (1-u)^3 p_0 + 3u(1-u)^2 p_1 + 3u^2(1-u)p_2 + u^3 p_3 \qquad (3.5)$$

$p(u)$ = points on the curve p_n = Bezier control points $u \in [0, 1]$

Bezier curves are defined mathematically by the Bernstein polynomials (see Equations 3.3 – 3.5). The value u ranges from 0.0 to 1.0, defining along with it the length and location of each pixel on the curve. Bezier curves possess two useful properties, the first of which being endpoint interpolation [30]. This means the first and last control points lie upon the curve, simplifying the process of fitting parametric curves to contours. Because of this property, important sections of the contour can be "fixed" and connected, enclosing the contour completely. The second property is affine invariance which means simple transformations (scaling, rotation, and translation) can be applied to the control points, changing the resulting Bezier curves appropriately [30].

The simplicity of Bezier curves made them prime candidates for use in the CECAT system. Unfortunately, Bezier curves do not enforce any degree of continuity that more sophisticated spline forms require. Because high compression is more important than continuous transitions between splines, the CECAT system only enforces C^0 continuity. If the contour being mapped makes a sharp point, the extra cost of preserving continuity does nothing to improve the later rendering of the contour.

3.3.2 Using First Degree Curves (Lines)

The simplest example of CECAT compression uses only first degree parametric curves (i.e. straight line segments). Although this can lead to suboptimal results (no smooth curves and extra required segments), the algorithm is almost identical to the one used to fit higher order parametric curves to contour. Because line segments are easier to visualize than quadratic splines, we will start by compressing an image with them.

Outline of the Contour Mapping Process

The algorithm used by the CECAT system to map parametric curves onto
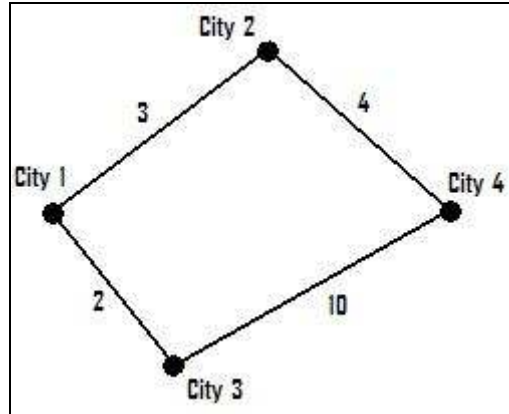
Figure 3.6 Suboptimal "Greedy" Algorithm Example.

contours is sometimes referred to as a "greedy" algorithm because it maps the longest curve available from its current point without regard to the consequences further down the road. This "locally optimal" strategy operates quite well with significantly fewer computations than a "globally optimal" strategy, because it need only deal with the current piece of contour. Unfortunately, the results can be suboptimal. For example, Figure 3.6 shows a map of four cities with the distance between each city labeled. If someone was trying to get from City 1 to City 4 and used a "greedy" algorithm for each leg of the journey, they would choose the shortest route first. Thus, they would travel to City 3, but then pay for it later as the distance between City 3 to City 4 is extremely high.

Similar to the example above, the CECAT contour mapping process starts at a given point and looks for the longest possible curve it can map and yet remain close enough to the original contour. "Close enough" is defined by something called Error Tolerance, a measurement defining how far (in pixels) from the contour any associated curve is allowed to go. Once the longest acceptable line segment is found, it is stored away and the algorithm repeats itself until it reaches the end of the contour. The implementation of each step in this process is described later in this section. The procedure *MapNextLineToContour* on lines 15 and 22 of the following pseudocode handle the process of selecting the next line segment, and lines 14-20 and 21-26 describe how these line segments are actually mapped.

Of course, there are a few exceptions to this "greedy" approach. First, contours that touch the image or tile border are assigned "fixed" line segments where they meet.

29
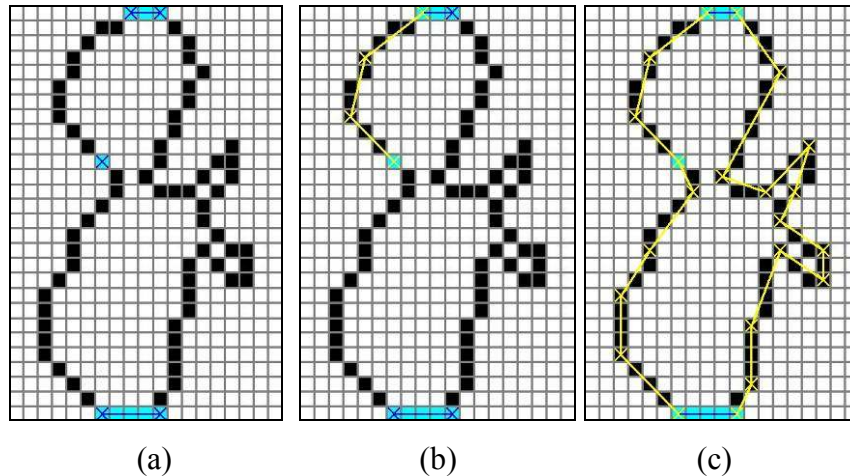
Figure 3.7 Line Segment Contour Mapping Example. (a) Initial Contour with Fixed Border Edges (b) Contour Mapping Complete for First Section (d) Contour Mapping Complete

By allowing a small amount of error in the mapping process, small "gaps" can appear inside an otherwise connected components when tiles are reassembled if these edges are not mapped perfectly. Figure 3.7a shows these fixed contour edges, and the procedure *FindBorderEdges* on line 12 is where this mapping takes place in the process.

After applying the "greedy" process outlined above to the area between the starting point and the first "fixed" line segment, a number of line segments can be mapped as shown in Figure 3.7b. At that point, the "fixed" line segment is added to the contour mapping and the process repeats itself until the contour is completely covered by line segments. The result of this mapping process is shown in Figure 3.7c. The pseudocode on the following page outlines this basic process, using upper and lower indices to specify where on the contour each line segment lays. The methods *FindBorderEdges* and *MapNextLine* will be discussed in the next few sections.

Marking the Outside Edges

Having every point on the parametric curves map perfectly to the pixels along the contour is not always desirable. Such a mapping requires too many parametric curves, reducing the efficiency of the compression strategy and removing the desirable "smoothing" effect the contour mapping provides. With that said, there are a few places on a contour where an exact pixel-by-pixel mapping is needed. The most important of

30

procedure MAPLINESTOCONTOUR
1: **Inputs:**
2:      Array of points[] *contour* {sequence of points making up the contour}
3:      Integer *start* {index for *contour* - first point of current section of contour}
4:      Integer *end* {index for *contour* - last point of current section of contour}
5:      Real *error* {max allowable distance between mapped line segment and contour}
6: **Outputs:**
7:      Array of lines[] *lines* {Line segments that have been mapped to the contour}
8: **Variables:**
9:      Array of lines[] *edges* {contour segments that touch tile borders}
10:     Bezier *next* {next mapped line; contains indices indicating endpoints}
11: **Begin**
12: *edges* = FINDBORDEREDGES(*contour*)
13: **for** *i* ← 0 to *edge.length* **do**
14:     **while** true **do**
15:         *next* = MAPNEXTLINE(*contour*, *start*, *contour[edge[i].lowerIndex]*, *error*)
16:         add *next* to *lines*
17:         *start* = *next.upperIndex*
18:         **if** *next.lowerIndex* > *edge[i].lowerIndex* **then break**
19:     add *edge* to *lines*
20: **while** true **do**
21:     *next* = MAPNEXTLINE(*contour*, *start*, *end*, *error*)
22:     add *next* to *lines*
23:     *start* = *next.upperIndex*
24:     **if** lower index of *next* > *end* **then break**
25: **End**

these exist where the contour touches the border of the image or the edge of a tile.

Absolute precision is needed when encoding these edges for two reasons. First, slight deviations in mapping these edges have the potential to push the contour outside the dimensions of the image. If this were to occur, the decoder would clip the tile when trying to reconstruct the image. Second, because the progressive transfer strategy uses tiles to localize and transmit images in a piecewise manner, imprecise edges can create gaps when two tiles are pieced back together.

To prevent both of these conditions, each contour is first analyzed and these border segments are detected and saved (as shown by the blue segments in Figure 3.8). By storing a list of segments along with indices indicating their starting and stopping points, these line segments can be "fixed". In this way, we are guaranteed to have precise fitting edges between each tile, and no contour moves beyond the edge of the image.

Adding these "fixed" border edges is not without cost. The smoothing effect
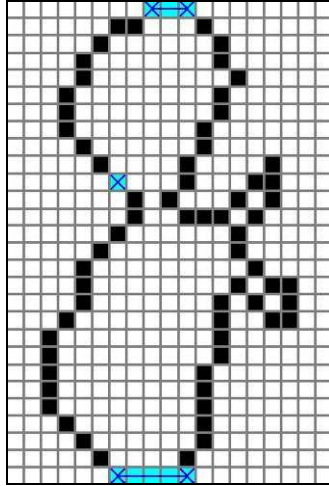
Figure 3.8 Detected Border Edges.

Error Tolerance (pixels)	Image DPI	File Size without Fixed Borders (bytes)	File Size with Fixed Borders (bytes)	Difference (bytes)	Difference (percent)
1.5	200	86,690	88,504	1,814	2.1%
1.0	200	107,000	108,028	1,028	1.0%
0.75	200	131,694	132,447	753	0.6%
1.5	300	123,708	126,300	2,592	2.1%

Table 3.1 File Size Price for Fixed Borders.

provided by allowing a small amount of error in contour mapping is lost for these particular edges. Table 3.1 shows the image size for various CECAT files compressed with and without fixed border edges. As expected, the file size difference is less pronounced when a more restrictive error tolerance value is used. At any rate, the small cost of 0.5% – 3.0% is minor when compared to the "artifacts" this process prevents.

The process that marks these outside edges is straightforward. Because each point on the contour is stored in an ordered list, finding which segments lie along a particular edge is a matter of detecting spans where the contour touches and later leaves the edge. The algorithm steps through the list of contour points until the contour touches the edge of the tile (line 15 of the following code resolves to true). This edge is followed until the contour leaves the tile's edge or reverses direction (as in lines 30-32 and 26-29 respectively). Once this happens, a line from the *start_point* to the *end_point* is stored as an edge line segment and "fixed" for the contour mapping process. The implementation details behind this operation are shown in the pseudocode on the next page. This

function is run four times to discover and set edge line segments on all four border edges (top, bottom, left, and right).

procedure FINDBORDEREDGES
1:   **Inputs:**
2:      Array of points[] *contour* {sequence of points making up the contour}
3:   **Outputs:**
4:      Array of beziers[] *edge_list* {*contour* segments that touch bottom tile border}
5:   **Variables:**
6:      Integer *start_point* {index for *contour* - first point of current contour edge}
7:      Integer *end_point* {index for *contour* - last point of current contour edge}
8:      Integer *start_x* {first x coordinate for the current contour edge}
9:      Integer *end_x* {last x coordinate for the current contour edge}
10:     Enum *direction* {"unknown", "left", "right"; direction of current contour edge}
11:     Boolean *following_edge* {indicates that an edge is currently being followed}
12: **Begin**
13: *following_edge* = False
14: **for** *i* ← 0 to *contour.length* **do**
15:     **if** *contour[i]* is on the current edge of current tile **then**
16:        *following_edge* = True
17:        **if** *contour[i-1]* was not on the current edge of current tile **then**
18:           *direction* = "unknown"
19:           *start_x* = *end_x* = *contour[i].x*
20:           *start_point* = *end_point* = *i*
21:        **else if** *direction* = "unknown" **then**
22:           **if** *start_x* > *contour[i].x*
23:              *direction* = "left"
24:           **else**
25:              *direction* = "right"
26:        **else if** (*direction* = "left" AND *end_x* < *contour[i].x*) OR
                 (*direction* = "right" AND *end_x* > *contour[i].x*) **then**
27:           add line from *start_x* to *end_x* to *edge_list*
28:           *start_x* = *end_x*
29:           *start_point* = *end_point*
30:           reverse *direction* {"left" becomes "right" and vice versa}
31:        *end_point* = *i*
32:        *end_x* = x coordinate of *contour[i]*
30:     **else if** *following_edge* = True
31:        add a line from *start_x* to *end_x* to *edge_list*
32:        *following_edge* = False
33: **End**

Fitting a Line to a Contour Segment

To ensure good curve-to-contour mapping, CECAT encoding requires the curve
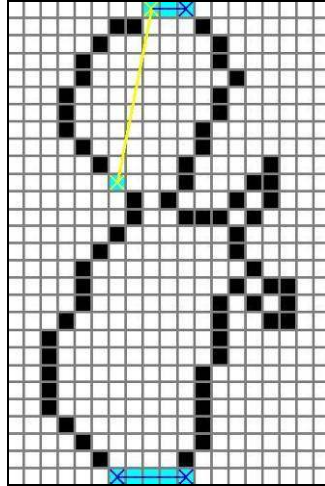
Figure 3.9 First Candidate Line Segment.

segments it uses to begin and end on pixels found on the contour. By forcing the
endpoints of each segment onto the contour, "fixing" line segments to the edges
described above is much simpler. This rule also ensures an exact curve-to-contour
mapping at least two times per curve segment and prevents the mapped curve segments
from oscillating from one side of the contour to the other and simplifies the algorithm
that chooses how to map a line segment to a section of contour as shown below:

procedure MAPLINE

  1: **Inputs:**
  2:     Array of points[] *contour* {sequence of points making up the contour}
  3:     Point *start* {starting point for the contour segment being examined}
  4:     Point *end* {ending point for the contour segment being examined}
  5: **Outputs:**
  6:     Line *next* {line segment that have been mapped to a section of contour}
  7: **Begin**
  8:    *next* = line with endpoints *start* and *end*
  9:    **return** *next*

For the "first attempt" at a contour mapping, the system tries to map a single line
from the starting point to the beginning of the first "border edge" segment detected
earlier. An example of this is shown in Figure 3.9. If there are no "border edges" on the
contour, the algorithm's initial attempt is a single point line at the starting point for the
contour.

34

Determining the How Close a Line Fits to the Contour

Once we have our first candidate line, it must be analyzed to determine the accuracy of the curve mapping it provides. To do this, we must determine which points on the contour map to which points on the mapped line. Fortunately, this operation turns out to be simple thanks to the parametric equation for a first degree Bezier curve:

$$\boldsymbol{p(u) = (1-u)p_0 + up_1} \hspace{3cm} (3.6)$$

The value for u can be found by calculating the percentage distance between each point on a contour segment and the contour segment's starting point. After calculating the distance between the points used to calculate u and its associated computed value for $p(u)$ in the equation above, the maximum distance between a point on the contour and its associated point on the parametric curve can be determined (dx and dy calculated on lines 18 and 19 in the pseudocode below). This maximum distance, as shown in Figure 3.10, is called the "error value" for the parametric curve.

procedure GETERRORFORLINE
1: **Inputs:**
2:     Array of points[] *contour* {sequence of points making up the contour}
3:     Point *start* {starting point for the candidate line being examined}
4:     Point *end* {ending point for the candidate line being examined}
5:     Array of integer[] *distances* {measured distances from each indexed point in contour to starting point}
6: **Outputs:**
7:     Integer *error* {highest measured error between points on line and contour}
8: **Variables:**
9:     Integer *total_distance* {distance measured following contour from start to end}
10:     Real *U* {relative distances along both contour and candidate line}
11:     Real *dx* {horizontal distance between points on the candidate line and contour}
12:     Real *dy* {vertical distance between points on the candidate line and contour}
13: **Begin**
14: *total_distance* = last value in *distances*
15: *error* = 0
16: **for** $i \leftarrow$ index of *start* to index of *end* on *contour* **do**
17:     U = *distances[i]* * 1 / *total_distance*
18:     dx = (1-U) * *start_point.x* + U * *end.x* – *contour[i].x*
19:     dy = (1-U) * *start_point.y* + U * *end_point.y* – *contour[i].y*
20:     **if** *maxError* < squareRoot($dx^2 + dy^2$)
21:         *maxError* = squareRoot($dx^2 + dy^2$)
22: **End**
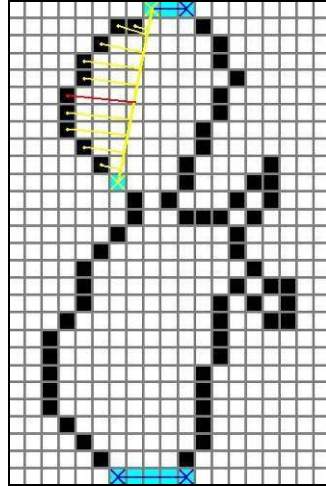
Figure 3.10 Comparison of Deltas between Candidate Line and its Associated Contour.

Throughout the process of mapping curves to contours, significant improvements in both image quality and compression can be achieved by allowing the mapped Bezier curves to depart from the contour by a small margin. This small amount of discrepancy has the benefit of both smoothing the compressed contours and reducing the size of the final image [32]. The smoothing effect can simplify the form of many handwritten characters and improve the readability of the handwriting, especially important if curve-mapped contours were ever used for tasks like automated handwriting recognition.

The maximum acceptable distance between a contour segment and its mapped Bezier curve is known as "error tolerance".   If a curve is more than this number of pixels away from its associated contour at any point, that curve is labeled as a bad match. Reducing error tolerance naturally improves the accuracy of the match, but the smoothing effect of contour mapping is reduced. In addition, more curves are needed to represent contours when the error tolerance is low. The effect of error tolerance on the image is discussed in Section 5.1.1.

Performing a Search of the Best Line Mapping

Now that we have a way to map line segments to contours and test the accuracy of these line segments, we are ready to search for the longest line segment that follows the contour from a start point. This operation is fundamentally a recursive binary tree similar to a "divide and conquer" strategy search as shown in the following pseudocode:

36

procedure MAPNEXTLINE
1: **Inputs:**
2:    Array of points[] *contour* {sequence of points making up the contour}
3:    Integer s*tart* {index of initial point where the next Bezier is mapped from}
4:    Integer *end* {index of last point where the next Bezier can be mapped to}
5:    Real *error_tolerance* {max allowable distance between *current_curve* & *contour*}
6: **Outputs:**
7:    Bezier *current_curve* {next candidate Bezier and ultimately the optimal Bezier}
8: **Variables:**
9:    Array of integer[] *distances* {measured distances from each indexed point in
      contour to starting point}
10:   Integer *min_index* {index of last point of the shortest Bezier failing to pass}
11:   Integer *max_index* {index of last point of the longest Bezier that passed error test}
12:   Real *error* {maximum distance from a point on *current_curve* and *contour*}
13: **Begin**
14: *max_index = end_point*
15: *min_index = start_point*
16: calculate and fill values for *distances*
17: FINDNEXTBEZIER(*start_point, end_point, min_index, max_index*)
18:    *currentCurve* = MAPLINE(*contour, start_point, end_point*)
19:    *error* = GETERRORFORLINE(*contour, contour[start_point], contour[end_point],*
      distances)
20:   **if** (*error > error_tolerance*) AND ((*min_index + end_point*) / 2 > 2)
21:     **return** FINDNEXTBEZIER(*start_point*, (*min_index + end_point*) / 2,
      min_index, endPoint)
22:   **if** (*error < error_tolerance*) AND ((*end_point < max_index*) > 2)
23:     **return** FINDNEXTBEZIER(*start_point*, (*max_index + end_point*) / 2, *end_point*,
      maxPoint)
24:   **else**
25:     **return** *current_curve*
26: **End**

First, a line segment going from the start point to the last available point on the contour is selected (this point is either at the end of the contour or the beginning of the next "fixed" line segment). In the code above, the process of selecting the last available point and getting the error is done on lines 14-19. Figure 3.11a illustrates this step with the selected line and the largest error labeled red. If the error for this first line is more than the error tolerance, this line fails the test and a line segment going to the "halfway" point is tested instead as shown in lines 20-21 above and illustrated in Figure 3.11b.

From this point, the binary search continues. The results of each test determine the next candidate line to be tested. If a candidate line segment fails, the next candidate
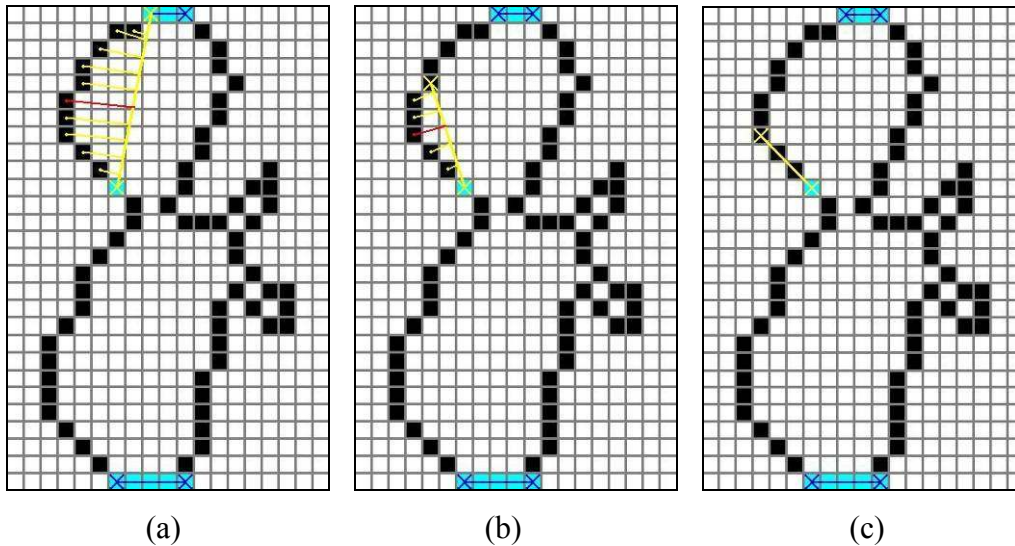
Figure 3.11 Determining the Best Line Mapping. (a) Initial Candidate Line and Associated Error Values (b) Second Candidate Line and Associated Error Values (c) Selected Line Mapping

line is set as halfway between that point and a *min_index* value which is initialized to the start point (line 15). If the candidate line segment passes the test, the next test takes place halfway between that point and a *max_index* value which is initialized to the last available point (line 14). Throughout the process, *min_index* and *max_index* values are tracked and adjusted. Every time a test fails, the endpoint of the line is saved as the current maximum distance for the optimal line segment. Each time a test succeeds, the endpoint is saved as the current "minimum" distance. Eventually, the *min_index* and *max_index* come together at which point the line segment that successfully maps to these *indeices* is chosen as the optimal line mapping.

This binary search operation was implemented to speed up the mapping process. Admittedly, testing for the longest available candidate line and stepping back one pixel each time a mapping fails does not take extremely long (it is a O(n) operation). Unfortunately, to create a CECAT image, tens of thousands of these line segments must be mapped. Using this binary search changed the operation to O(log n), which reduces the time it takes to compress an image considerably.

3.3.3 Using Second Degree Curves (Quadratics)

Mapping quadratic Bezier curves instead of line segments to contours is similar to

38

the process described for line segments. The algorithm is the same with only two small changes. First, the algorithm for fitting a curve to a contour is different. The endpoints for each perspective quadratic curve appear on the contour, but the middle control point needs to be calculated. This is done using a least squares fit algorithm. Second, the algorithm for calculating the error between a candidate curve and a contour uses the Bernstein polynomial for quadratics Beziers instead of line segments.

Fitting a Quadratic to a Contour Segment

       Because the endpoints of each quadratic Bezier curve are determined by the curve-fitting process outlined for line segments in the previous section, the only question that remains is where to place the middle control point. To do this, a computationally expensive linear algebra operation known as "least squares fit" is used [30, 31]. This operation takes the basis functions of the polynomial equation for the Bezier curve and evaluates them for a selection of sample points (in this case, every point on the contour from one end to another). These basis functions are plotted on a matrix and the eigenvector for the matrix is calculated. This eigenvector is the "least squares fit" to the data, which is actually the coordinates of the control point we are looking for. Because the x and y coordinates can be determined independently, this operation is actually much simpler than it sounds. The matrix can be reduced to a 1xN matrix, which greatly speeds up the process of calculating eigenvectors. This operation, which was originally implemented by Michael Smith [28], is shown in detail in the pseudocode on the next page.

Determining how close a Quadratic fits the Contour

       Similar to the procedure outlined for mapping line segments to contours in Section 3.3.2, the algorithm for determining how accurately a quadratic Bezier curve maps to a contour is a matter of applying the parametric equation for the second degree Bezier curve and comparing it with the points on the contour.

$$p(u) = (1-u)^2 p_0 + 2u(1-u)p_1 + u^2 p_2 \qquad\qquad (3.7)$$

       As outlined in the procedure for measuring the distance between a line segment

procedure MapNextQuadratic

1: **Inputs:**
2:   Array of points[] *contour* {sequence of points making up the contour}
3:   Integer *start* {index for *contour* - first point of current section of contour}
4:   Integer *end* {index for *contour* - last point of current section of contour}
5:   Array of integer[] *distances* {measured distances from each indexed point in
        contour to starting point}
6: **Outputs:**
7:   Line *next_bezier* {second degree Bezier that has been mapped to section of
        contour}
8: **Variables:**
9:   Integer *total_distance* {distance measured following contour from start to end}
10:   Real U {relative distances along both contour and candidate Bezier}
11:   Real T {simple holding variable used to store results of equation 2*(1-U) * U}
12:   Real *vx, vy, M* {variables used to create and evaluate Least Squares Fit matrix}
13: **Begin**
14: *total_distance* = last value in *distances*
15: $vx = vy = M = 0$
16: **for** $i \leftarrow$ index of *start* to index of *end* on *contour* **do**
17:   $U = contour[i] * 1 / totalDistance$
18:     $T = 2 * (1 - U) * U$
19:     $vx = vx + (T * (contour[i].x - (1 - U)^2 * start\_point.x - U^2 * end\_point.x))$
20:     $vy = vy + (T * (contour[i].y - (1 - U)^2 * start\_point.y - U^2 * end\_point.y))$
21:     $M = M + T^2$
22:   $controlPoint = (vx / M, vy / M)$
23: *next_bezier* = quadratic with control points: *startPoint, controlPoint, & endPoint*
24: **return** *next_bezier*
25: **End**

and contours, the value for *u* is found by calculating the percentage distance between each point on a contour segment and the contour segment's starting point. This allows us to compare the distance between *u* and the associated computed value for *p(u)*. The following pseudocode shows how this is done. Everything in this procedure aside from degree of the Bernstein equation is the same as for generating errors from line segments.

3.3.4 Combining First and Second Degree Curves

Because the steps required for mapping first and second degree Bezier curves are so similar, another "greedy" approach is used by the CECAT system to determine whether a line segment or a quadratic curve is the best choice for each piece of contour. This algorithm is controlled by a simple cost function: the cost of encoding two line

procedure GETERRORFORQUADRATIC
1:   **Inputs:**
2:     Array of points[] *contour* {sequence of points making up the contour}
3:     Point *start* {first control point for candidate Bezier being examined}
4:     Point *control* {middle control point for candidate Bezier being examined}
5:     Point *end* {last control point for the candidate Bezier being examined}
6:     Array of integer[] *distances* {measured distances from each indexed point in
          contour to starting point}
7:   **Outputs:**
8:     Integer *error* {highest measured error between points on line and contour}
9:   **Variables:**
10:     Integer *total_distance* {distance measured following contour from start to end}
11:     Real *U* {relative distances along both contour and candidate Bezier}
12:     Real *dx* {horizontal distance between points on the candidate Bezier & contour}
13:     Real *dy* {vertical distance between points on the candidate Bezier & contour}
14:   **Begin**
15:   *total_distance* = last value in *distances*
16:   *error* = 0
17:   **for** *i* ← index of *start* to index of *end* on *contour* **do**
18:     $U = distances[i] * 1 / total\_distance$
19:     $dx = (1\text{-}U)^2 * start.x + 2\ (1\text{-}U)U\ * control.x + U^2 * end.x - contour[i].x$
20:     $dy = (1\text{-}U)^2 * start.x + 2\ (1\text{-}U)U\ * control.y + U^2 * end.y - contour[i].y$
21:     **if** *max_error* < squareRoot($dx^2 + dy^2$)
22:       *max_error* = squareRoot($dx^2 + dy^2$)
23:   **return** *max_error*
24:   **End**

segments is equal to the cost of encoding one quadratic curve.

Using this simple rule, the algorithm makes two measurements from each start point. First, the longest quadratic curve is determined using the technique outlined in Section 3.3.3. Second, the next two line segments are mapped to the contour using the strategy shown in Section 3.3.2. If the quadratic curve reaches farther than the two line segments, the quadratic curve is saved as the best choice. On the other hand, if the two line segments reach farther the first of these two line segments is saved as the next mapped curve.

There are a few benefits to this strategy. First, the algorithm is simple and easy to implement. Second, the whole operation requires much less time to run than more complicated and sophisticated algorithms such as "backtrack" or "branch-and-bound". Third, it provides contour compression with locally optimal results. Fourth, and most importantly, this strategy is easily extensible. This means that adding B-splines, higher

200 DPI Census Image

Compression Type	Beziers Used	File Size (bytes)
Line Segments	29,749 Lines	71,870
Quadratics	19,782 Quadratics	86,693
Mixed Compression	24,033 Lines & 2,446 Quadratics	71,588

300 DPI Census Image

Compression Type	Beziers Used	File Size (bytes)
Line Segments	42,438 Lines	108,087
Quadratics	27,632 Quadratics	128,556
Mixed Compression	32,691 Lines & 4115 Quadratics	107,184

Table 3.2 Amount of Beziers Used During CECAT Compression

degree Bezier curves, or another parametric representation to the contour mapping process is simple. The only things needed are the following: a method for mapping a curve to a contour, a method for determining the error between the mapped curve and the contour, and a cost factor.

On the negative side, this strategy suffers from the same limitations that all "greedy" algorithms face: the consequence of locally optimal vs. globally optimal results. The final point for a particular quadratic may fall much shorter than the final point for the similar two line segments, but it might provide a much better starting point for the next step in the contour mapping.

Table 3.2 compares the CECAT file sizes using only lines, only quadratics, and a mix of the two. Although they help a little, quadratic Bezier curves do not provide much in the way of improved compression rates, as line segments are clearly superior compression-wise. On the other hand, the "smoothing" quality of the CECAT compression strategy can be reduced when only line segments are used. Changing the algorithm to allow for more quadratics and enhanced curve quality comes at the cost of file size, which is a dilemma faced by most image compression operations. Despite the

scarce use of quadratics in the current algorithm, the improved "smoothing" effect and the small improvement in compression makes mixing lines and quadratics still the best course of action

Chapter 4

Encoding and Transmission of CECAT Images

The CECAT system combines two technologies to facilitate document image browsing: image compression and progressive transmission. Chapter 3 discussed the process of encoding the document image "foreground mask" as a collection of first and second degree Bezier curves. Chapter 4 will discuss the file format for these compressed contours as well as the progressive transmission strategy used by the CECAT system. Section 4.1 will introduce the tiling strategy used to separate images into manageable chunks. Details about the file formats used to store the different layers of a CECAT-encoded image are given in Section 4.2. Section 4.3 discusses the "Curve Segment Library", a tool used to improve compression of the "foreground mask" by creating a lookup table of common line segments. Lastly, Section 4.4 will describe the progressive transmission strategy used to send the encoded images to low-bandwidth users.

4.1 Localization of Contours

The strategies employed by the CECAT system for localizing contours are quite simple. First, contours in each tile must be sorted into different layers to prevent larger contours from overwriting smaller ones. Additionally, each image is divided into tiles. To improve the compression, a consistent tile size of 512 x 512 pixels is used. These tiles can be transferred as a block and all their contents rendered in the same step. In this way, an image viewer can easily display pieces of the image to the user without forcing them to wait for the whole image to be transmitted.

4.1.1 Storing Contours as Layers

The concept of storing sets of contours as distinct layers was mentioned in Section 3.2.1. These layers are the first, and simplest, form of contour localization employed by the CECAT system. Because some contours can be completely contained inside others, it is imperative that outer contours are rendered before any contours contained inside. If this is not enforced, the larger contour will simply write over the top of the other "contained" contours. These contours represent the holes inside larger black shapes or shapes inside these holes.

Fortunately, because contour detection presorts these contours according to layer, simply storing and rendering them in the default order keeps these contours from overwriting each other. This is the strategy currently used by the CECAT system. It is simple and requires no additional computation. If an advanced strategy for sorting contours by priority inside each tile is developed, attention must be paid to prevent rendering these layers out of order.

4.1.2 Tiling the Images

The CECAT system uses a very simple tiling strategy: each tile is a 512 x 512 pixel block. The only exceptions to this are the tiles along the right and bottom edges of the image, where they are simply cropped to fit the image. Figure 4.1 shows a sample image and its associated tiles.

Fixing each tile to a maximum of 512 x 512 pixels provides several important benefits. First, the average file size for a tile of this size is usually less than three kilobytes. Table 4.1 shows the average tile size for CECAT images compressed at various error tolerance levels. This size is appropriate for a single packet passed over a dial-up internet connection. Second, fixing the size of the tile allows for some minor improvements to the encoding of each tile. One piece of data, which is essential for each contour, is a starting point in (x, y) coordinates. Because each starting point is relative to the upper-left corner of its respective tile, the slot for each of these contours can be limited to nine bits (representing coordinates ranging from 0-511) instead of the previously used two bytes. This reduces the file size of a CECAT image by about
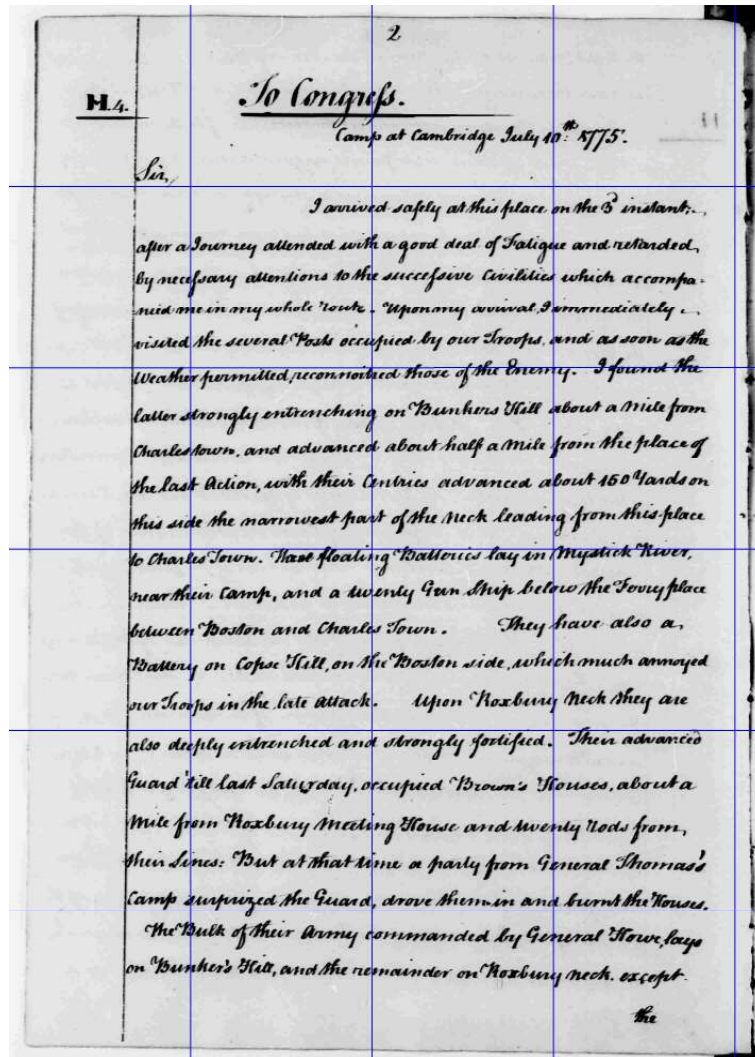
Figure 4.1 Tiled Document Image (using 512 x 512 pixel tiles).

Error Tolerance (pixels)	Tile Size (KB)
1.5	1135
1.0	1409
0.75	1895
0.5	2558

Table 4.1 Average CECAT Tile Size

two bytes per contour. Given the number of potential contours in each image, this can add up fast.

4.2 CECAT File Format

The most significant aspect of the CECAT file system is the encoding strategy for the "foreground" contour-encoded layer. This encoding strategy is a major contribution of the CECAT system as well as the result of all the work described in Chapter 3. This is described in detail in Section 4.2.1. The encoding strategy for the second two layers, the residual and background layers were added to demonstrate the progressive transmission strategy. These strategies are discussed in Sections 4.2.2 and 4.2.3 respectively. Because grayscale image compression was not the emphasis of the CECAT system, these layers have not been optimized for compression efficiency. As a result, a brief discussion about optimizations added to the compression of the foreground, contour encoded, layer continue in Section 4.3.

4.2.1 Encoded Contour Layer

One of the most important contributions of the CECAT system is the method by which the control points for the various parametric curves used to represent contours are compressed and represented in a data file. This data file format has a direct effect on the compression ratio as well as the image data availability.

There are a few principles used by the CECAT file compression system that may be useful to review before getting into the file structure. First and foremost, everything in the CECAT file format is bit-oriented. For some pieces of data like the starting points and number of Beziers per contours, the encoding strategy assigns them a particular number of bits that may or may not be divided along the standard 8, 32, or 64 bit partitions. Although this imposes a maximum value to each data slot, the amount of unused space required for the image data is significantly reduced.

The second principle used by the encoding strategy system to reduce file size is the use of "deltas" instead of fixed control points coordinates. Instead of storing absolute X and Y coordinates for each control point, the relative distance from the previous control point on the contour is stored instead. This significantly reduces file size and makes it possible to improve compression by using techniques such as the curve segment library discussed in Section 4.3.

The third principle involves the use of variable-length data elements. For example, to represent the "deltas" mentioned above, four bits are used to tell the system how many bits are needed to represent the required distances. By allowing a variation in the number of bits required for these values, there can be a much higher maximum value without the need for an excessive amount of unused "filler bits". In addition, these four bits actually represent the first '1' bit for the "delta" they reference, removing the need for repeating it in the next collection of bits. This does puts a limit to the size of the deltas that can be represented. Because the tile size is restricted to 512 x 512 bits, this does not pose a problem.

As shown in Figure 4.2, the each contour-encoded foreground layer starts with a basic image header. The image header gives basic information about the height and width of the original image. 16-bit values limit the maximum dimensions of the image of to little more than 65536 pixels and could be extended to allow for larger images, but that did not seem necessary for the initial implementation of the CECAT system. The number of bits needed to represent the height and width of each tile follow the image height and width in the header. To reduce the file size by a few bits per contour, the tile width and height are required to be factors of 2. As a result, the four bits can specify a tile edge ranging from 2 to 65536 pixels in length. Using this information, the decoder can set the correct tile boundaries.

After the image header, each contour has header that is 33 bits long. This header contains data used to render its corresponding contour. The first bit marks the contour as a black or white shape. Following this is the "Last Contour Flag" which, when set to true, tells the decoder stops looking for more contours and moves onto the next tile. 13 bits are then used to store the number of Beziers required to render the contour. The length of 13 bits was selected arbitrarily, setting the maximum number of curves used to represent for a single shape to 8192.

For each Bezier, the degree is the only piece of data required. After that, the "curve segment" data is represented by a 10-bit index to the curve segment library described in Section 4.3 or data defining the delta from one control point to another.

Data Element	Bits Used
Image Header	
Total Image Width	16
Total Image Height	16
Tile Pixel Width (bits needed to represent)	4
Tile Pixel Height (bits needed to represent)	4
Contour Header	
Internal Flag (is shape black or white?)	1
Last Contour Flag (is this the last contour?)	1
Number of Beziers	13
X Coordinate for Contour Starting Point	Tile Width
Y Coordinate for Contour Starting Point	Tile Height
Bezier Data	
Degree of Bezier Curve	2
Segment Data	
Stored Segment Flag (are deltas in library?)	1
If Stored Segment	
Curve Segment Index	10
If Not Stored Segment	
Delta Width (bits needed to represent)	4
Positive Flag (is delta X positive or negative?)	1
Delta X	Delta Width - 1
Delta Height (bits needed to represent)	4
Positive Flag (is delta Y positive or negative?)	1
Delta Y	Delta Height - 1

Figure 4.2 CECAT File Structure.

4.2.2 Residual Image Data Layer

The encoding strategy for the residual image data layer is extremely simple, and could benefit from more work (grayscale compression was not an emphasis of this thesis). The residual layer contains grayscale data for every pixel that is rendered black in the "foreground" contour layer as well as all the white pixels adjacent to these black pixels. By supplementing these extra pixels, the residual layer adds a tremendous amount of detail to an otherwise bitonal image, outlining and enhancing the handwritten content with valuable grayscale data. This is a simple antialiasing operation.

<div align="center">(a)   (b)</div>
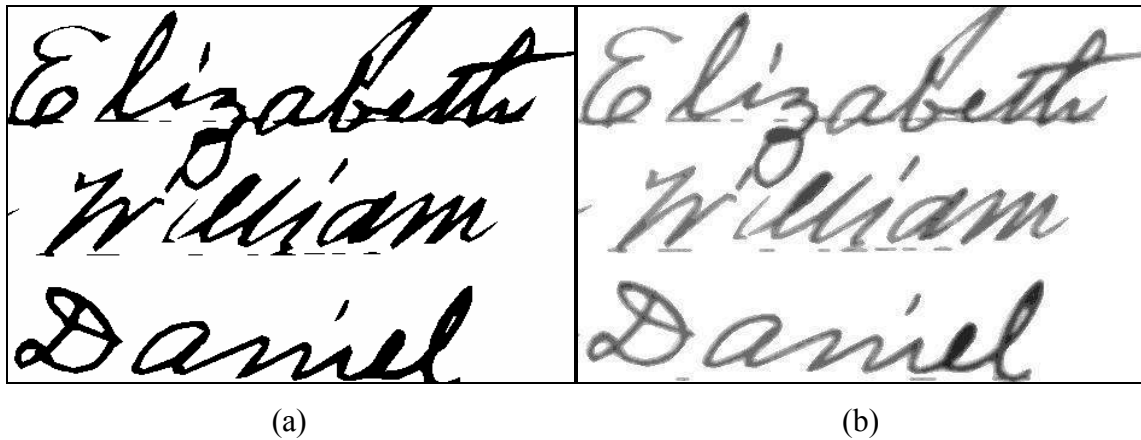
Figure 4.3 CECAT Tiles. (a) Foreground Mask (b) Residual Layer

To improve compression for this image data, the grayscale values are converted into one of the following eight levels of gray: 1, 36, 72, 108, 144, 180, 216, and 254. Because only eight different levels of gray are used, each pixel can be represented by three bits instead of the requisite eight bits required for a full grayscale pixel. This image data is further compressed using a common run-length encoding strategy known as gzip.

Like the contour-encoded "foreground" layer, this pixel information is stored in tiles so it can be later transmitted after its associated contour layer. By requiring the contour encoded layer to be transferred first, the data for the residual layer can be used to "fill in" the grayscale information onto the "foreground" layer. As a result, location references are not needed in the residual layer. As shown in Figure 4.3, the data in the residual layer is organized by using the contour encoded layer as a mask and adding the residual grayscale data sorted from upper left to lower right in regular scan-line order.

4.2.3 Background Image Data Layer

The CECAT system uses the same compression strategy for the background layer as it does for compressing the residual layer. In summary, each pixel not found in the residual image layer is converted into one of the eight different grayscale values mentioned in Section 4.3.2. These grayscale values are then stored as three-bit data values, ordered in a standard scan line order from the top of the image to the bottom. As a final touch, this data is compressed with a simple gzip compression algorithm. In short, the background layer pixels are treated just like the residual layer pixels.

Index Size	Max Length (pixels)	Compression	% of Contours in Library
10	15	5.07%	65%
11	22	4.57%	79%
12	31	1.45%	89%
13	44	-3.72%	94%

Table 4.2. Curve Segment Library Compression Enhancements

4.3 Curve Segment Library

One of the more useful optimizations discovered while developing the CECAT compression strategy was a curve segment library. As mentioned in section 4.2.1, aside from the absolute starting point, contours are stored as a chain of deltas from one control point to another. After analyzing various compressed contours, it was discovered that up to 65% of these deltas were less that 16 pixels in size and the number of bits required to represent them ranged from 10 to 18 bits.

To take advantage of this redundancy, a curve segment library was created, containing deltas ranging from (-15,-15) to (+15, +15), indexed by a 10-bit integer value. The 10-bit index was chosen following a number of experiments with different index sizes and compression improvements. Each index can only represent a range of deltas, and Table 4.2 shows the maximum delta the each index can represent, the percent of contours on the test images that fell within that range, and the overall compression improvement each library provides.

One of the big advantages of this library is that it can be created in the viewer without having to send it from the server. The library consists of an exhaustive list of all the deltas ranging from (-15, -15) to (15, 15). The CECAT image viewer is quite capable of creating this library and storing it in RAM, where it can be referenced as needed. The contents of the library are simple as demonstrated in Figure 4.4. This shows the first few deltas stored in the curve segment library along with their associated indices.

Two different types of curve segment libraries were implemented: one for decoding images and the other for encoding images. The encoder library has a constant time lookup of indices given two deltas. This greatly speeds up using this library while encoding a contour. The decoder library, on the other hand, uses a constant time lookup

Index	Delta X	Delta Y
0	0	0
1	0	1
2	0	-1
3	1	0
4	-1	0
5	0	2
6	0	-2
7	1	1
8	-1	1
9	1	-1
10	-1	-1

Figure 4.4 First 11 Entries in Curve Segment Library.

For deltas given an index. Although both libraries can be used to look up deltas and indexes, using them in the opposite direction takes much longer. The curve segment library is built right into the CECAT file format detailed in Section 4.2, using a single bit flag to tell the decoder if the contour is in the library or not.

4.4 Progressive Transmission

By compressing the various image layers in a tiled format, it is possible to send the image to a viewer a piece at a time. This process, known as progressive transmission, is the second, albeit smaller, contribution made by this thesis. Because the images have been "tiled" and broken into layers, it is possible to create a server and a viewer capable of displaying these images as if they were in the process of being downloaded to a viewer. Section 4.4.1 describes how the sample server was created to simulate transmitting tiles from a CECAT file. In Section 4.4.2, the process of receiving contour encoded tiles from a server and displaying them on a viewer is discussed. Finally, the process for transmitting and adding the residual and background layers to an image is discussed in Section 4.4.3.

4.4.1 Sample Server Implementation

To demonstrate the potential of the progressive transmission of CECAT images, a simple client-server system was set up to open compressed files. This server simulates

sending images tile-by-tile to a simple viewer. Although there is much work that can be done to improve this operation, it does a reasonable job demonstrating the potential of the CECAT file format. What follows is a brief description of the user experience associated with this sample server as well as a few implementation details on how the server operates.

User Experience with Sample Server

There are currently two ways of "downloading" a CECAT image using the sample server. One method is what might be considered the "manual" approach. The server will send one tile each time the operator presses a button. As soon as the last tile for the foreground layer image is sent, the first residual layer tile is added, followed by the rest of that layer. The same thing happens with the background layer. This approach shows how a CECAT image may appear during download, as well as what happens if the download freezes or is cancelled.

The second method for downloading a CECAT image involves something called a "floating window viewer". For this strategy, the viewer sends requests to the server for tiles in the area of the image where the viewer is currently displaying. As a result, scrolling around the image the first time sends a lot of tile requests to the server. Fortunately, the tile information is saved in the viewer, so tiles do not need to be sent a second time. This makes scrolling through the image a little jerky at first, however subsequent scrolling operations are quite fluid. To request another image layer using this viewing method, the user simply presses a button on the keyboard. This sets the viewable layer to "residual" and then to "background" if the button is pressed again. If the layer is set to one of these levels and the user scrolls into a section that has not had any layers sent yet, the server sends all the necessary layers one-by-one.

Server Implementation Details

As of the time of this implementation, CECAT images are composed of three different files, one for each layer of the image. When a viewer requests an image, the server first opens the data file containing the contour encoded layer, parses out the image data and stores information for each tile into a large array. When the viewer makes

subsequent requests for specific tiles, a copy of the tile data is sent directly from the array.

To preserve memory, the other layers (residual and background) are not stored in the server memory. After the array of contour-encoded tiles has been created, the server then goes through both the residual and background layers creating an index to each tile. Because each tile begins with a 32 bit number describing how many pixels of data it contains, creating a list of indices for these tiles only requires a single pass through the appropriate files. In response to a request for a particular tile containing one of these layers, the server opens the appropriate file at the index location, reads the requested image data, compresses it with a simple Gzip compression operation, and sends it to the viewer.

4.4.2 Rendering the Contour Encoded Tiles

Most of the steps required for the receiving and rendering of contour-encoded tiles have been described in Section 3.2.2. The basic procedure for rendering a tile is simple. The Image Server sends a CECAT tile to the CECAT Viewer, which then converts the tile a list of contours. These contours are then filled using the algorithm outlined in Section 3.2.2.

The only part of the progressive transmission strategy that has not been covered elsewhere is the 'canvas' upon which the image is painted. When the CECAT Viewer requests a compressed image, the Image Server responds with a brief "header" file telling the Viewer the height and width of the requested tile. The Viewer uses this information to create a 'canvas' (a buffer of memory that stores the image data as byte-length pixel values). The CECAT Viewer can only see this canvas, which gets updated each time a tile is received. In addition, a simple map is used to keep track of which tiles have already been received, preventing the CECAT Viewer from needlessly requesting image data a second time.

4.4.3 Adding Residual and Background Layers

Because the second two layers use the first as a mask, it is imperative that the contour-encoded foreground layer be received and rendered first. This requirement

prompts the need for the map mentioned in Section 4.4.2. Once a contour-encoded tile is rendered, the procedure for adding the other layers on top of it is simply a matter of decompressing the gzipped pixel data, changing each pixel from 3-bit to its 8-bit grayscale values, and filling over the appropriate portion of the contour-encoded mask with pixel data in a scan-line order from top to bottom. These changes are made to the 'canvas' mentioned in Section 4.4.2 and are quickly reflected in the CECAT Viewer after the image data has been received.

Chapter 5

Compression Efficiency and Results

The CECAT compression system compares favorably with other document image compression algorithms, especially the compression of the bitonal foreground mask. Although very little work was done on the grayscale compression (the residual and background layers), the compression was competitive with other more sophisticated compression algorithms once the image was reduced to eight levels of gray. In addition to a study of compression efficiency, CECAT encoded images also provide a simple tiled structure that allows for progressive transmission of portions of each image at full resolution.

This chapter shows results of the compression and usability tests, comparing the CECAT system to other freely available document image compression systems. These compression systems include the JBIG and JPEG2000 standards as implemented by the GraphicsMagick open-source imaging package [33]. In addition, the DjVuLibre package (an open-source distribution of the DjVu encoding standard) was used to compress images in DjVuBitonal, DjVuPhoto, and full DjVu files [34]. Section 5.1 presents the results of the compression tests. Image quality and usability are discussed in Section 5.2. Section 5.3 describes some of the inefficiencies and weaknesses in the CECAT system.

5.1 Analysis of CECAT Bitonal Compression

To analyze the effectiveness of the CECAT compression system, a few common compression formats were applied to four small sets of document images. Two of these sets, the George Washington Papers and the James Madison Papers consist of handwritten correspondence captured at 100 dpi resolution. The other two datasets contained US Census pages that were extracted from microfilm at resolutions of 200 and

Error Tolerance (pixels)	Image DPI	File Size (bytes)	Error Tolerance (pixels)	Image DPI	File Size (bytes)
0.5	200	193,920	0.5	300	295,844
0.75	200	143,166	0.75	300	213,102
1.0	200	107,764	1.0	300	161,440
1.25	200	91,786	1.25	300	136,732
1.5	200	84,112	1.5	300	126,300
1.75	200	79,225	1.75	300	119,451
2.0	200	75,174	2.0	300	113,725
3.0	200	66,004	3.0	300	100,396

Table 5.1 Relative CECAT File Size at Different Error Tolerance Settings

300 dpi. For more details on each of these sets of images as well as thumbnails of each image, consult Appendix A.

Because the compression enhancements were focused around the bitonal foreground mask, most of the improvements in compressions were seen at that level as shown in Section 5.1.1. By combining the cost of all the layers of the CECAT image, further tests were made against common color image compression standards in Section 5.1.2. Lastly, compression effectiveness between "hybrid" compression strategies is discussed in Section 5.1.3.

5.1.1 Getting the Settings for the CECAT System

The CECAT system has two parameters that control the amount of lossy data: error tolerance (which was discussed in Section 3.3.2) and "despeckling" which removes the small contours such as single pixel points and stray dots. By using these two settings, the CECAT bitonal image file size can be reduced considerably. Care must be taken, however, when choosing the appropriate settings, because they remove data from the image.

Error Tolerance

As discussed in Section 3.3.2, error tolerance is the maximum distance allowed between a contour and the Beziers mapped to it. Table 5.1 shows the comparative file size for a 200 and a 300 dpi image compressed using different error tolerances values

CECAT File Size vs Error Tolerance
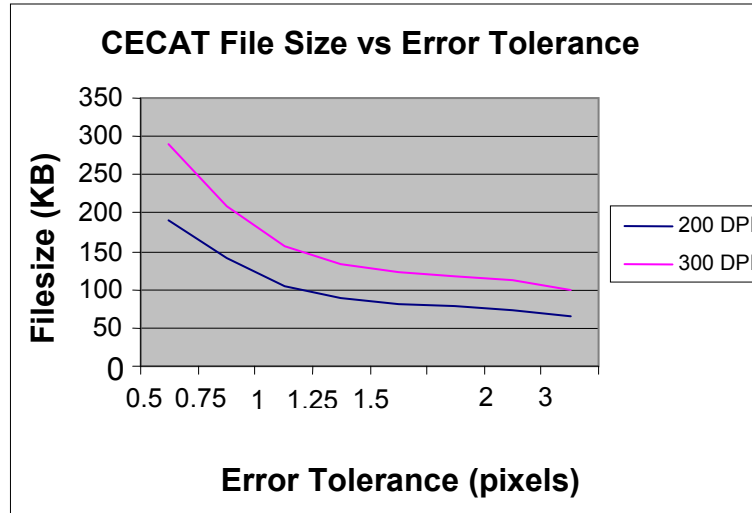
Figure 5.1 CECAT File Size verses Error Tolerance.

ranging from 0.5 to 3.0 pixels. In conjunction with Table 5.1, Figure 5.1 shows a plot comparing file size and different error tolerance values. One of the goals of this thesis is to identify the point of diminishing return (also called the "knee of the curve") with regard to error tolerance. According to this data, an error tolerance setting ranging from 0.75 – 1.25 appears to produce the best results.

Although, some substantial gains in compression efficiency can be obtained by using a large error tolerance value, this is not without cost. If the compression routine is set to allow too much error, serious artifacts can occur. The "smoothing" effect created by nicely matched quadratic Bezier curves can end up being replaced by block-like line segments. Figure 5.2 shows a few examples of the same name from a 200 dpi image compressed with different error tolerance settings. Obviously, an error tolerance setting above 2.0 appears to create some blocky hard-to-read text when applied to 200 dpi images.

Using this as a guide, the compression tests were run using the following error tolerances: 0.0, 0.5, 0.75, and 1.0. This gives a good accounting of file size vs. image quality as controlled by error tolerance settings.
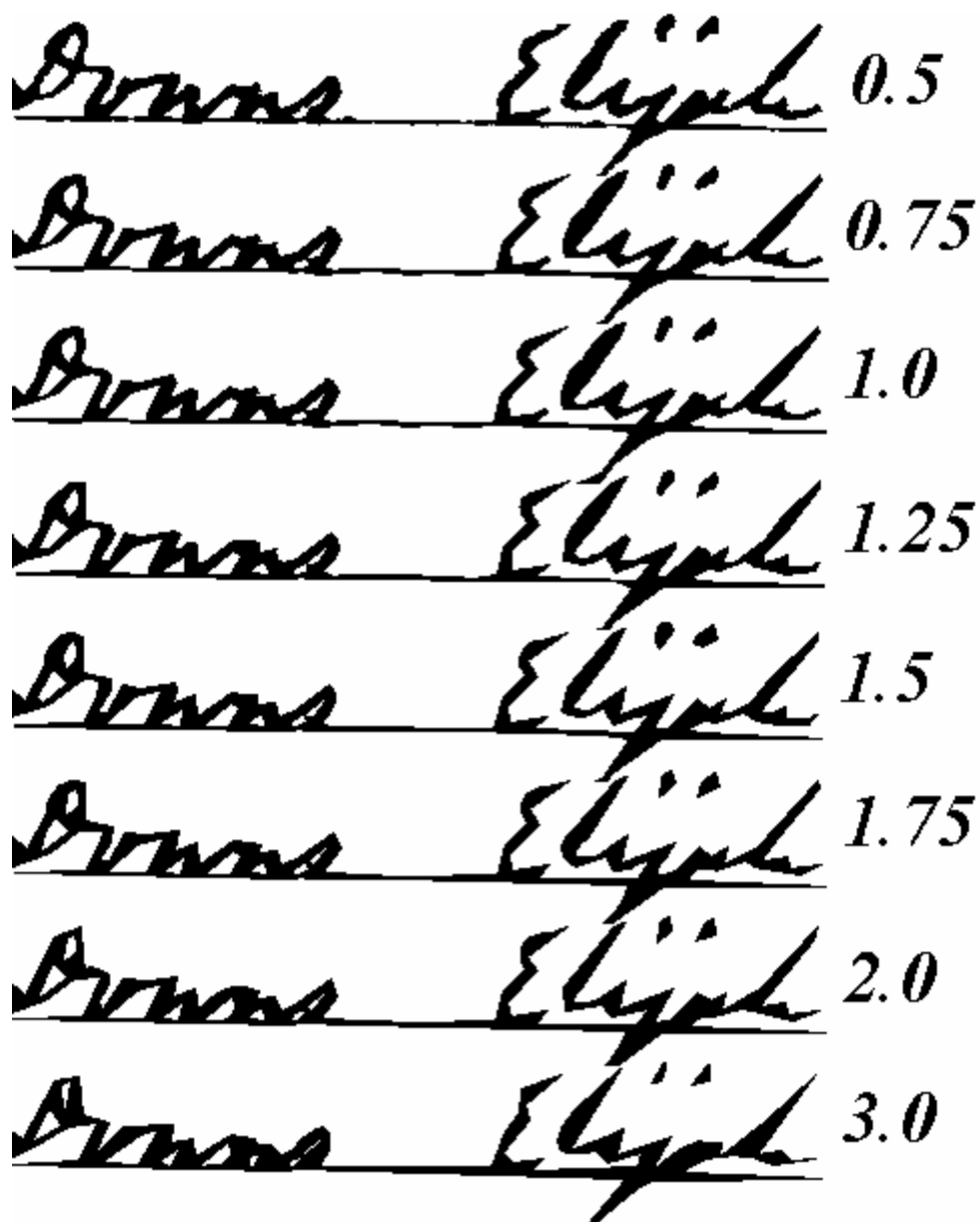
Figure 5.2 Compressed Image Quality verses Amount of Error Tolerance.

"Despeckling" Operation

To reduce the overhead of using contours to compress small (1 – 4 pixels long) shapes, a "despeckling" operation is used to remove any contours that are less than a fixed length. To determine a good value for this fixed number, a series of compression tests were run on sample images from each of the datasets. Interestingly enough, changing this value didn't affect the image quality as much as expected, although the file size definitely took a hit.

(a)   (b)   (c)
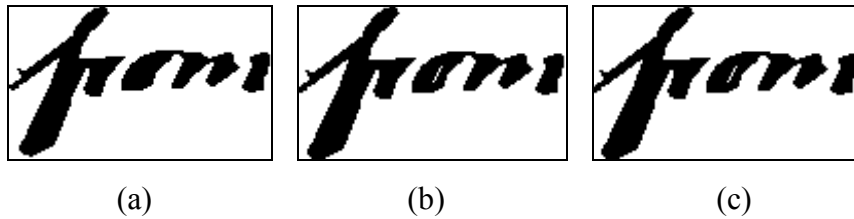
Figure 5.3 CECAT Compression from 100 dpi George Washington Papers. (a) 16 Pixel Length Despeckling (b) 12 Pixel Length Despeckling (c) No Despeckling

(a)   (b)   (c)
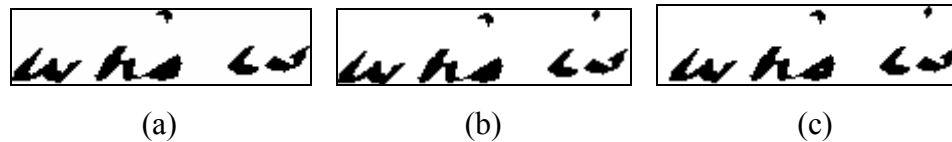
Figure 5.4 CECAT Compression from 100 dpi James Madison Papers. (a) 16 Pixel Length Despeckling (b) 12 Pixel Length Despeckling (c) No Despeckling

(a)   (b)   (c)
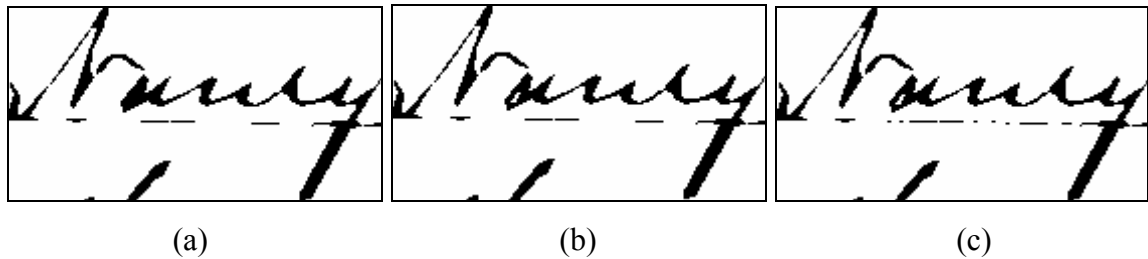
Figure 5.5 CECAT Compression from 200 dpi U.S 1870 Census. (a) 16 Pixel Length Despeckling (b) 12 Pixel Length Despeckling (c) No Despeckling

(a)   (b)   (c)
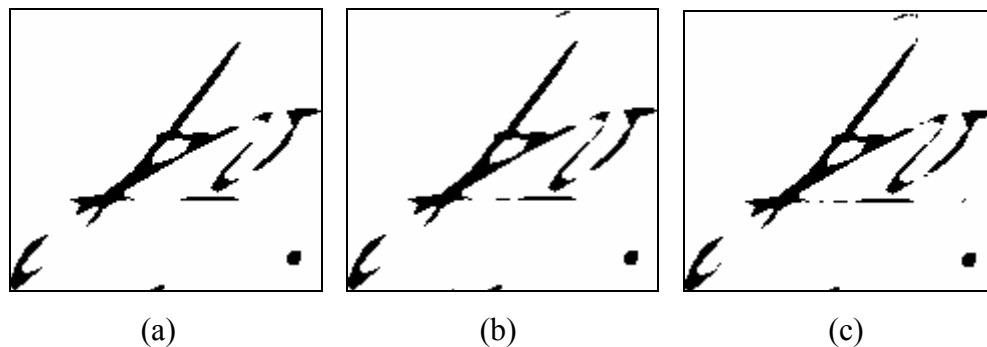
Figure 5.6 CECAT Compression from 300 dpi U.S 1870 Census. (a) 16 Pixel Length Despeckling (b) 12 Pixel Length Despeckling (c) No Despeckling

Figures 5.3 – 5.6 shows the result of despeckling the images by removing contours with less than 16 and 12 pixels in length. Further tests were done using a "despeckling" operation with 8 and 4 as the minimum pixel length, but the resulting

Despeckling Settings	GW Papers	JM Papers	200 DPI Census	300 DPI Census
None	33	42	143	168
4 Pixels	32	42	137	162
8 Pixels	31	41	126	153
12 Pixels	30	40	115	145
16 Pixels	29	40	107	139

Table 5.2: CECAT Compression file sizes (using 0.5 error tolerance) for sample images with various "despeckling" settings. The file sizes are given in Kilobytes.

images were very close those using 12 pixel despeckling. The file sizes for these CECAT images (which were compressed with a 0.5 error tolerance) are shown on Table 5.2. Given the file sizes and the overall quality improvement, a default setting of 12 pixels was selected for the compression tests.

5.1.2 Bitonal Image Compression Results

The foreground mask layer for a CECAT image is a bitonal representation of the document image. As such, the compression effectiveness can be compared to other bitonal image compression algorithms. As mentioned in Section 5.1.1, for the purposes of these tests, the CECAT compression was done using four error tolerance settings: 0.0, 0.5, 0.75 and 1.0 and the "minimum contour length" controlling the "despeckling" operation was set to remove contours containing less than 12 pixels.

Two common document image compression standards were used for these bitonal image compression tests: JBIG and DjVuBitonal. The JBIG images were compressed using default settings in the GraphicsMagick [33] software package. Although not a commercial image compression package, GraphicsMagick accurately implements the JBIG standard. The DjVu bitonal images were created using the DjVuLibre open source package [34]. The CECAT foreground masks generally ranged in size from one-third to one-half the size of both JBIG and DjVuBitonal compression. All in all, very favorable file size and quality comparisons were made despite some binarization problems. The results of a few of these tests along with sample images taken from each data set follow.

Figure 5.7 Bitonal image compression for a portion of the George Washington Papers (reduced in size). (a) JBIG (b) DjVu Bitonal (c) CECAT [1.0 error]   (d) CECAT [0.75 error]   (e) CECAT [0.5 error] (f) CECAT [no error]   (g) Original JPEG copy

Dataset 1: George Washington Papers

The first dataset tested was taken from the George Washington Papers, an online collection of George Washington's handwriting stored as digital JPEG images. These 100 dpi resolution images had good contrast, allowing the binarization algorithm to

Page	Contours No Error	Contours 0.5	Contours 0.75	Contours 1.0	DjVuBitonal	JBIG	Raw
2	161	65	45	38	111	111	782
5	190	79	53	45	125	124	764
10	171	71	48	41	135	135	797

Table 5.3: Bitonal compression comparisons for 100 dpi images from the George Washington Papers. The file sizes are given in Kilobytes.

operate effectively. Unfortunately, the fact that the original images were low quality JPEG images introduces artifacts in the images that would not be present if clean copies were used. Figure 5.7 shows the results of applying JBIG, DjVu Bitonal, and the CECAT compression at error tolerances of 0.0, 0.5 0.75 and 1.0. The relative file sizes for these four different compressed images are shown on Table 5.3.

Although the letters in the CECAT-encoded images were not as "thinned out" as the JBIG and DjVu Bitonal images (which appear to be very similar to each other), all four images are quite readable. The thickness of the letters is a result of poor binarization, likely the result of using low quality JPEG images as a source. In this case, the binarization algorithm padded each letter with the darker sections of the document surrounding it.

On the other hand, the CECAT images are also free from the dithering effect that JBIG and DjVu Bitonal compression algorithms add to darker sections of the image. This dithering effect is removed by the "despeckling" operation performed on the CECAT images before encoding begins. This operation reduces the background noise considerably. This does not come without some cost, however. With the "despeckling" operation set to remove shapes with less than 12 total pixels in the contour, a few small holes tend to be lost as well (such as in the A's or O's in the CECAT images).

As far as file size is concerned, the CECAT images ranged from about a fifty percent increase in size (for no error) to less than one-third of the size of the other image files for an error tolerance of one pixel. Although the images shown in Figure 5.7 were somewhat reduced in size, the differences between the CECAT image without error and the 0.5 pixel error CECAT image appears quite miniscule. All in all, this was a very favorable compression comparison, demonstrating the power as well as some limitations of the CECAT system.

Figure 5.8 Bitonal image compression for a portion of the James Madison Papers.
(reduced in size). (a) JBIG (b) DjVu Bitonal (c) CECAT [1.0 error]   (d) CECAT
[0.75 error]   (e) CECAT [0.5 error] (f) CECAT [no error]   (g) Original JPEG copy

Dataset 2: James Madison Papers

The second dataset contains images from the James Madison Papers, another

online collection of 100 dpi low-quality JPEG encoded images of handwriting. This

Page	Contours No Error	Contours 0.5	Contours 0.75	Contours 1.0	DjVuBitonal	JBIG	Raw
11	149	71	53	45	171	169	515
16	93	41	30	26	107	108	501
20	118	52	38	32	108	107	488

Table 5.4: Bitonal compression comparisons for images from the James Madison Papers. The file sizes are given in Kilobytes.

collection contains poorer quality images than the George Washington Papers, especially considering the contrast and readability of the images. The limitations of the binarization algorithm as well as the results of the "despeckling" operation on the bitonal image are more pronounced in these images. Despite this, the CECAT image file sizes were less than a third of the file sizes for DjVuBitonal and JBIG encoded images. Table 5.4 shows the relative file sizes of each of these images.

Figure 5.8 shows the compressed images from this dataset. The poor image quality of the original images in the James Madison Papers has an effect on the readability of the bitonal representations of this image. The JBIG and DjVuBitonal images represent some portions of letters with small collections of dots while the CECAT images fail to capture those pieces of the image. This demonstrates the danger associated with the "despeckling" operation. Like the inside of the A's and O's in the George Washington Papers, pieces of the letters found throughout this document may have been lost because the connected components were all too small. This shows the need for a more intelligent (or at least human-adjustable) "despeckling" operation.

After performing a couple more tests with the "despeckling" on the image above, it appears that the root cause of this problem is the binarization algorithm, not the "despeckling" operation. The pieces of the letters missing from the CECAT images were removed when the image was converted to a binary image before any contour compression took place. The words, which were converted correctly into foreground / background layers, are quite readable even on the CECAT images shown in Figure 5.8 (such as the words "to confer on"). On the other hand, poorly segmented words (like "army") are much more difficult to read. Improving the binarization algorithm would help this dataset considerably.
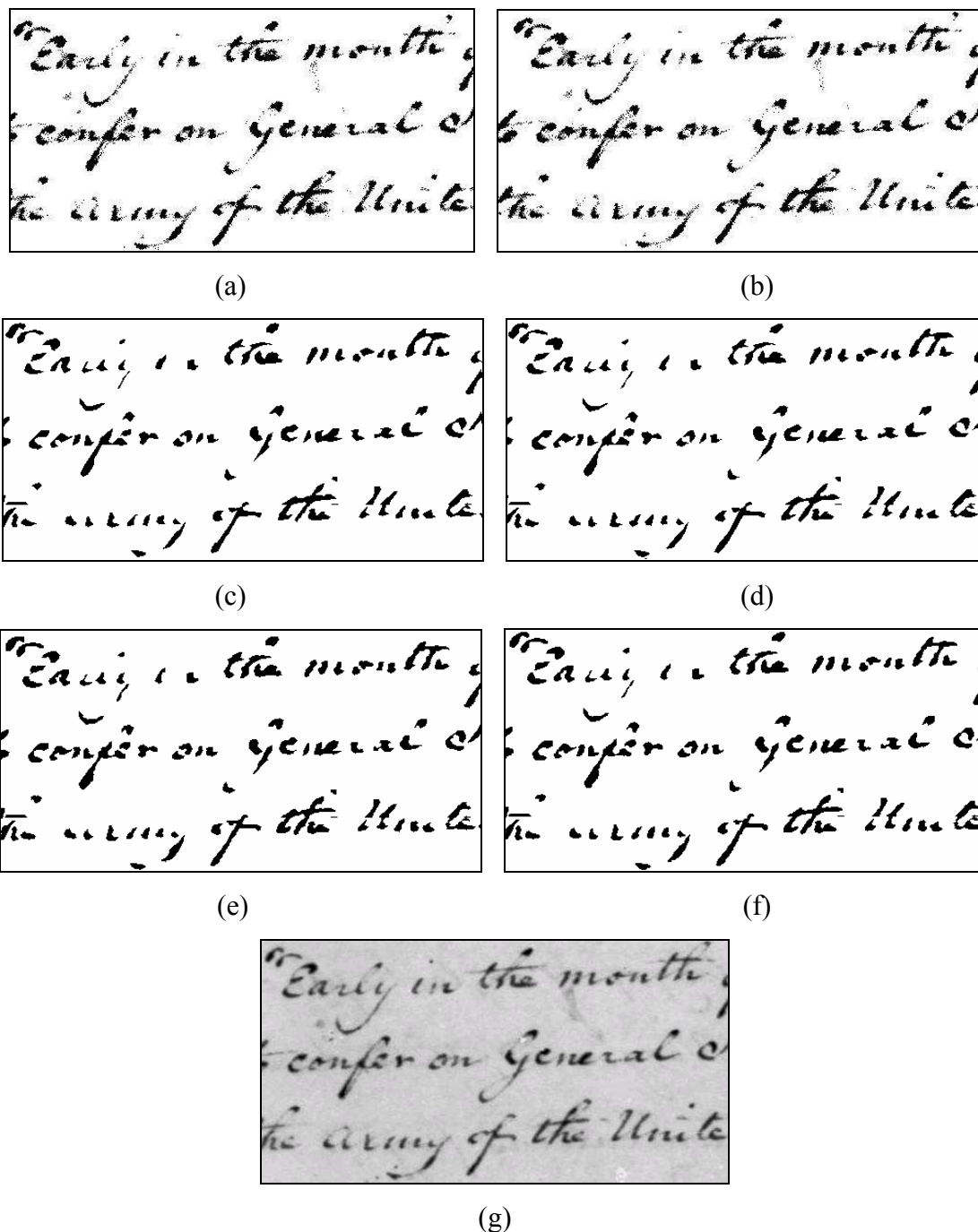
Figure 5.9 Bitonal image compression for a portion of the 1870 US Census 200 dpi. (reduced in size). (a) JBIG (b) DjVu Bitonal (c) CECAT [1.0 error]   (d) CECAT [0.75 error]   (e) CECAT [0.5 error] (f) CECAT [no error]   (g) Original JPEG copy

Dataset 3: US 1870 Census (200 DPI Resolution)

As the resolution of the images increase, the quality and readability of CECAT images improves. The next dataset used consists of images from the 1870 U.S. Census. These images were taken directly from microfilm and were scanned as 200 dpi images. Due to a limitation in the scanning operation at the time these images were taken, the

Page	Contours No Error	Contours 0.5	Contours 0.75	Contours 1.0	DjVuBitonal	JBIG	Raw
3	370	163	126	106	392	375	1904
8	403	170	131	108	362	344	1889
9	401	175	135	111	410	399	1985

Table 5.5: Bitonal compression comparisons for 200 dpi images from the US 1870 Census. The file sizes are given in Kilobytes.

contrast for these images was poor. This gave the binarization algorithm some difficulty with these with these images, but the results shown in Figure 5.9 display some promise. Although a few pieces of letters were lost (such as pieces of the letter 'l' and 'S' on the second line) due to poor binarization, overall image quality looks good. The dithering effect of the DjVuBitonal and JBIG images was replaced by smooth, solid strokes in the CECAT images, enhancing the readability and overall "crispness" of the image.

In addition to the enhanced image quality, the CECAT compression distanced itself even farther in the lead for image file size. Table 5.5 shows these compression differences for US Census images saved at a 200 dpi resolution. Since the resolution doubled, the CECAT image file size allowing 1.0 error tolerance images was about one fourth of the file size for DjVu and JBIG compressed images. As the error tolerance shrank, the CECAT image file sizes remained competitive with 0.5 error tolerance CECAT images having less than half the size of the next compression algorithm. Even more exciting than that, at this resolution the "no error tolerance" CECAT images finally come to about the same file sizes as the DjVu and JBIG images.

Dataset 4: US 1870 Census (300 DPI Resolution)

The fourth and last dataset also contains images from the US Census, only these images were captured at 300 dpi. Unfortunately, the contrast problem inherent in the previous dataset was more severe in these 300 dpi images, resulting in poor binarization. As shown in Figure 5.10, small pieces of handwritten strokes were lost: the connecting stroke between the 'a' and 'r' in the word "Farmer", the 'm' in the word "Farmer", and the connecting stroke between the 'e' and 'p' in the work "Keeper". Because of the size of the pieces missing, problems with the "despeckling" operation can be ruled out, leaving the binarization algorithm as the culprit. Aside from the inefficiencies with the
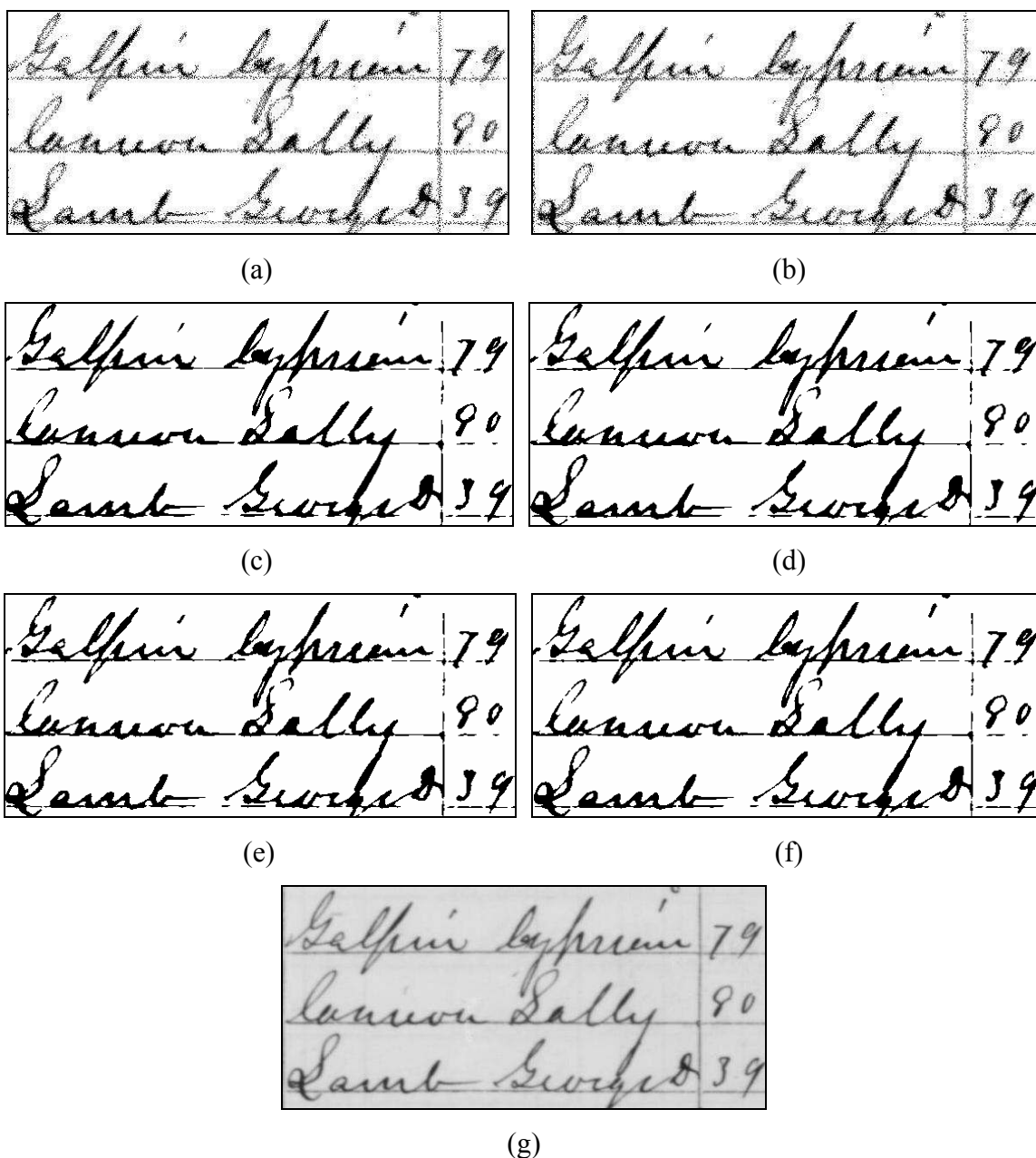
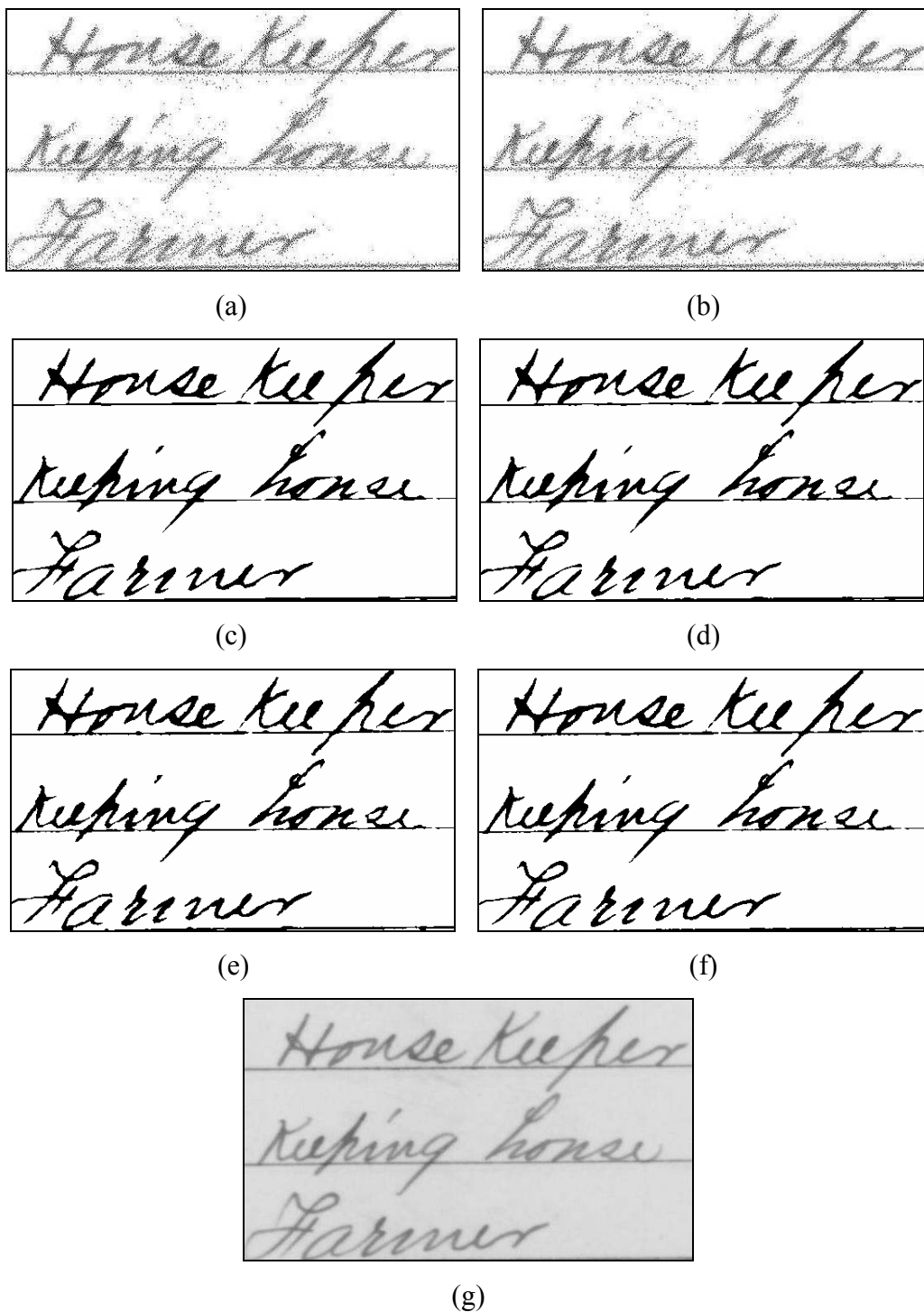Figure 5.10 Bitonal image compression for a portion of the 1870 US Census 300 dpi. (reduced in size). (a) JBIG (b) DjVu Bitonal (c) CECAT [1.0 error]   (d) CECAT [0.75 error]   (e) CECAT [0.5 error] (f) CECAT [no error]   (g) Original JPEG copy

binarization algorithm, the CECAT images contain sharp, fluid letters when compared to the "dithering" effect that blurs the handwriting in the JBIG and DjVuBitonal images.

| Page	Contours No Error	Contours 0.5	Contours 0.75	Contours 1.0	DjVuBitonal	JBIG	Raw
8	496	221	165	135	730	693	3791
12	678	277	204	163	825	769	3706
15	696	275	199	156	777	734	3681

Table 5.6: Bitonal compression comparisons for 300 dpi images from the US 1870 Census. The file sizes are given in Kilobytes.

In addition to nice contrast and overall image quality, the CECAT images continued to outperform the other compression strategies in terms of image file size. As shown in Table 5.6, the file size of the CECAT images with a 1.0 pixel error tolerance was less than one-fifth of the size of the other file formats and the 0.5 pixel error tolerance images was less than one-third the size for these higher resolution images. The most exciting result, however, is the fact that the "no error" CECAT images were actually smaller than the DjVuBitonal and JBIG images. It is important to note, however, that if the binarization algorithm was more accurate, the size of the CECAT files might be higher as more shapes appear in the image.

5.2 Analysis of CECAT Grayscale Compression

The focus of this Thesis has been the encoding of a bitonal foreground mask using contours and tiles. This is fine if only a bitonal representation of the image is needed. As explained in Section 4.2.2 and 4.2.3, the CECAT image consists of three layers: the bitonal foreground mask, the grayscale residual layer, and the grayscale background layer. This section discusses the effectiveness of the grayscale compression (all three layers of the CECAT image added together) against the following standards: JPEG, JPEG2000, DjVuPhoto, DjVu, and the raw pixel data.

The compression used for the residual and background layer was not fully developed during the course of this Thesis. Despite this, the basic strategy used is somewhat competitive with the other compression standards. The biggest limitation of the residual and background layers lies in the fact that the CECAT system reduces the 8-bit grayscale to 3-bit grayscale. Of course, this is the primary reason for good compression rates (the compression starts at 3/8 of the original image size without any extra treatment). The only other compression strategy used is the standard Gzip
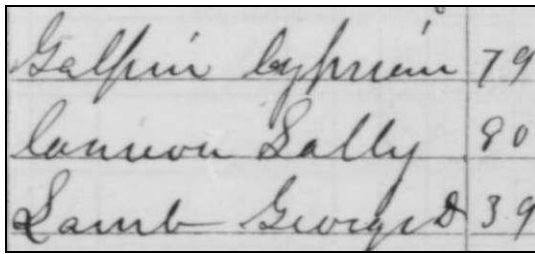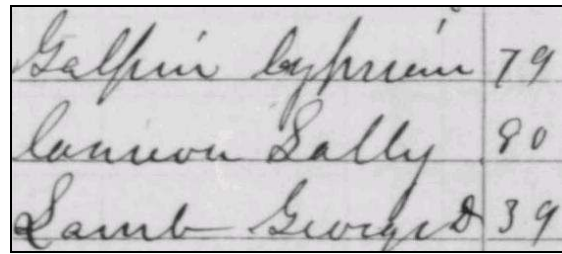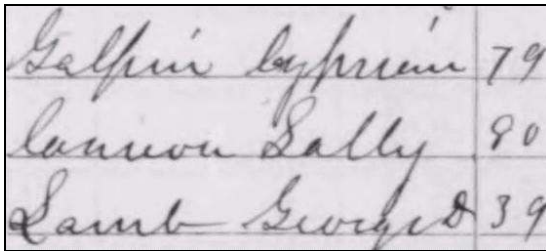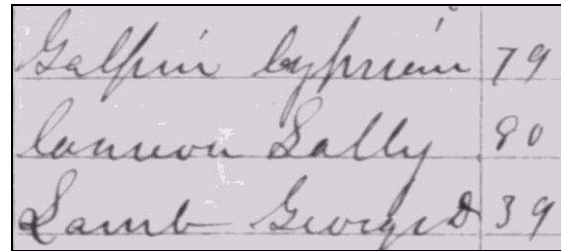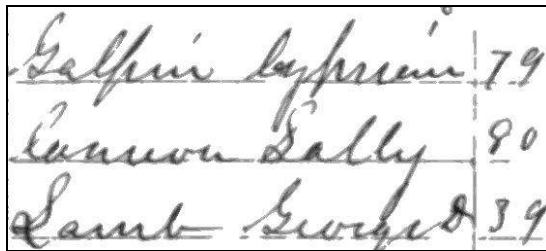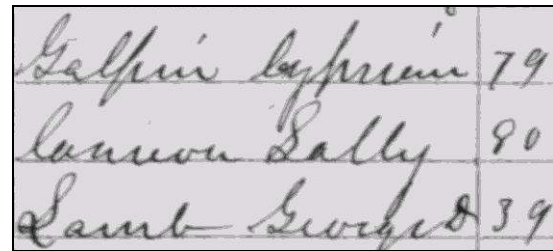
Figure 5.11 Grayscale image compression for a portion of the 1870 US Census captured at 200 dpi. (a) JPEG (b) JPEG2000 (c) DjVuPhoto (d) DjVu (e) CECAT residual layer with an error tolerance of 0.75 (f) CECAT full image

encoder. As a slight bonus to the compression, chopping the image into the residual and background layers tends to group similar shades of gray together (this improves the Gzip operation). As for the foreground mask, after the entire image has been transferred the visible pixels only come from the residual and background layers. In many respects, the contour-encoded foreground mask only adds to the final file size as it is overwritten by these other two layers in the end.

With that in mind, the CECAT grayscale images compared favorably to the other compression standards. As Figure 5.11 shows, reducing the color number of shades of gray from 256 to 8 does not impact the readability of the images very much. In some ways, the residual layer, with its white background and grayscale foreground is more readable than the other, more sophisticated, approaches. Of course, the strength of the

Page	JPEG	JPEG2000	CECAT 0.75	CECAT 1.0	DjVuPhoto	DjVu	Raw
2	217	391	359	355	398	601	6240
5	237	381	429	425	448	736	6086
10	229	399	419	414	419	657	6360

Table 5.7: Compression comparisons for 100 dpi images from the George Washington Papers. The file sizes are given in Kilobytes.

Page	JPEG	JPEG2000	CECAT 0.75	CECAT 1.0	DjVuPhoto	DjVu	Raw
11	352	258	439	431	199	607	4117
16	300	251	276	272	179	600	4000
20	336	244	376	370	217	683	3884

Table 5.8: Compression comparisons for 100 dpi images from the James Madison Papers. The file sizes are given in Kilobytes.

Page	JPEG	JPEG2000	CECAT 0.75	CECAT 1.0	DjVuPhoto	DjVu	Raw
3	1070	953	806	793	530	1350	15230
8	1106	945	832	813	579	1547	15106
9	1114	993	849	832	562	1223	15878

Table 5.9: Compression comparisons for 200 dpi images from the US 1870 Census. The file sizes are given in Kilobytes.

Page	JPEG	JPEG2000	CECAT 0.75	CECAT 1.0	DjVuPhoto	DjVu	Raw
8	1619	1896	1232	1204	637	1548	30326
12	1745	1854	1398	1362	699	1845	29648
15	1790	1841	1433	1393	734	2307	29447

Table 5.10: Compression comparisons for 300 dpi images from the US 1870 Census. The file sizes are given in Kilobytes.

other compression standards is the fact that they are representing the image with all 8 bits, providing the potential for finer detail.

Tables 5.7 – 5.10 show the differences in file size between the CECAT grayscale images and the other various file formats, with Table 5.9 showing the file sizes of the images shown in Figure 5.11. Quantitatively speaking, the CECAT grayscale compression performed consistently better than DjVu with 50% – 60% less file size. At resolutions of 200 dpi and higher, the CECAT grayscale images also outperformed JPEG

and JPEG2000 images. DjVuPhoto turned out to perform much better on all but the George Washington Papers dataset.

5.3 "Hybrid" Image Layer Comparison

Using the "out of the box" DjVu compression routine found in the open-source DjVuLibre project [34], hybrid DjVu files containing multiple layers similar to the CECAT encoded images were created. Both formats, DjVu and CECAT, consist of a bitonal foreground mask, a grayscale layer containing color information, and an encoded background color layer. These layered images facilitate a content progressive transmission by sending one or more layers at a time, allowing the user to view to contents of these earlier layers without having to wait for the whole image to be transmitted.

One advantage that the CECAT system has over the DjVu progressive transmission strategy lies in the fact that each layer of the image is further subdivided into tiles that can be transmitted one by one. For a simple comparison of progressive transmission strategies, the DjVuLibre encoder and viewer was used to show the three layers of the DjVu file. Figure 5.12 shows the different layers of a CECAT encoded image and DjVu images side-by-side using samples from the George Washington papers dataset.

Apparently, the DjVu foreground mask suffers from poor binarization just like the CECAT system, although from the look of Figure 10b, the results of the foreground mask is too "blocky" to read. In its defense, the DjVu was not specifically designed for handling grayscale images, having more of a focus on color images. Even so, the CECAT foreground bitonal mask is superior to the DjVu image in terms of readability and size. Of course, some of the distortion in the DjVu foreground mask could spring from the fact that this dataset contains low-quality JPEG images as its source.

Once the residual grayscale layer has been transmitted, the DjVu image is just as readable as the CECAT residual image (see Figures 10c and 10d), especially since the DjVu residual layer contains the background pixels covered by the "blocky" foreground mask. The CECAT image, however, does contain a much higher contrast as the background remains mostly white. This sharp contrast can make it easier to follow the
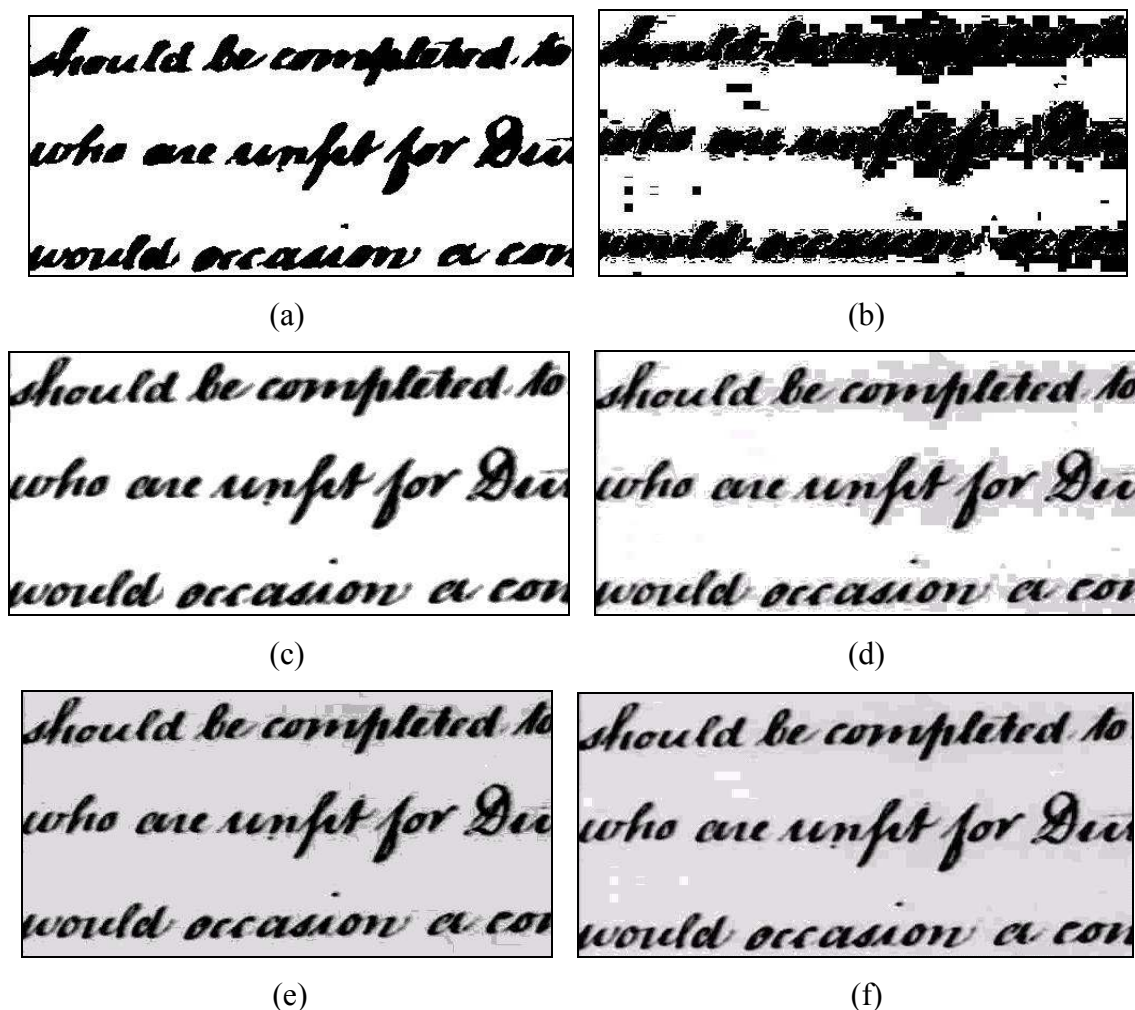
Figure 5.12 "Hybrid" image compression comparison for a portion of the George Washington Papers. (a) CECAT Foreground Layer (b) DjVu Bitonal Foreground Layer (c) CECAT Residual Layer (d) DjVu Grayscale Foreground Layer (e) CECAT Background Layer (f) DjVu Background Layer

strokes of the letters with the human eye.

For a second example, the foreground image masks from the 300 dpi resolution copy of the 1870 U.S. dataset are shown in Figure 5.13. Obviously, the binarization algorithm failed, leaving the foreground image mask as an opaque black square. In these cases, the foreground mask and the residual color layer are needed before any image details can be made out.

In addition to comparing these various layers qualitatively, the tools found in the DjVuLibre package can provide the file sizes for each of the three DjVu layers. By analyzing these images and the file size of each layer, some interesting trends were seen.

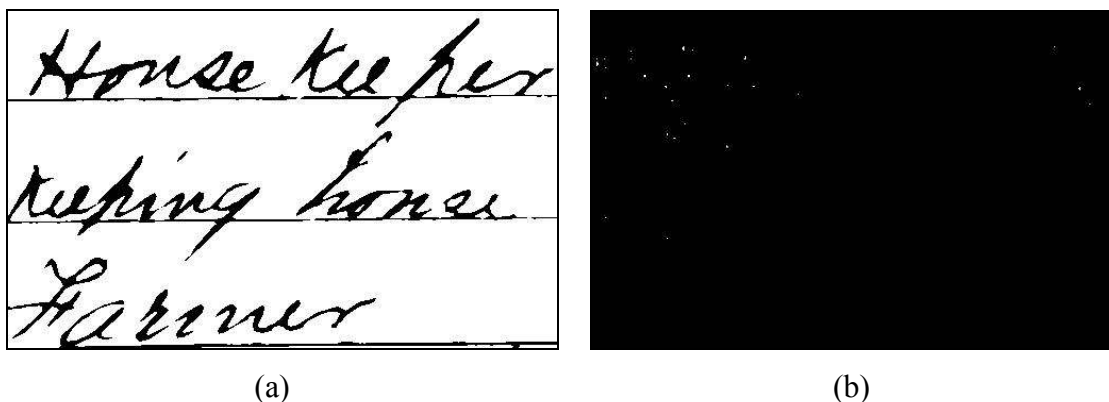|	
(a)	(b)

Figure 5.13 "Hybrid" image compression comparison for a portion of the 1870 US Census scanned at 300 dpi. (a) CECAT Foreground Layer (b) DjVu Bitonal Foreground Layer

First of all, the DjVu background and foreground color layers were extremely well encoded. The foreground mask, on the other hand, made up for most of the total file size (around 95%) and was larger than the whole CECAT image.

Looking at these results layer-by-layer, the CECAT system outperformed the DjVu encoding for the bitonal layer, resulting in contour-encoded image files which were less than 10% of the DjVu foreground layer (called the JB2 Bilevel layer). The DjVu encoding, however, outperformed the CECAT system in the residual/JB2 color layers. It is possible that DjVu uses context information from the first layer to render the next layers. Quantitatively, the DjVu JB2 color layer was less that 20% of the CECAT residual layer. Of course, the biggest gain in the DjVu encoding was seen in the IW4 background layer which never exceeded 1 KB in size. Tables 5.11 – 5.14 show how the DjVu and CECAT images compare in size, layer-by-layer.

5.4 Limitations of the CECAT System

Despite the compression efficiency of the CECAT system, these tests revealed a few of its limitations as well. The most glaring of these is the dependency on an underdeveloped binarization algorithm for detecting the foreground mask. As mentioned in Section 3.1.2, the bitonal conversion process was limited to a basic localized binarization algorithm with a tunable threshold. In the case of the US Census images, the threshold had to be changed from 64 to 128 to achieve reasonable binarization.

	CECAT (Error 0.75 / Error 1.0)			DjVu		
Page	Contours	Residual	Background	JB2 Bilevel	JB2 Colors	IW4
2	43 / 36	214 / 212	102 / 107	571	32	1
5	50 / 42	264 / 263	115 / 120	698	39	1
10	45 / 37	242 / 241	132 / 136	626	31	1

Table 5.11: Comparison of "Hybrid" image layers for 100 dpi images from the George Washington Papers. The file sizes are given in Kilobytes.

	CECAT (Error 0.75 / Error 1.0)			DjVu		
Page	Contours	Residual	Background	JB2 Bilevel	JB2 Colors	IW4
11	56 / 47	188 / 188	195 / 196	579	29	1
16	30 / 26	111 / 110	135 / 136	573	28	1
20	40 / 33	140 / 140	196 / 197	651	32	1

Table 5.12: Comparison of "Hybrid" image layers for 100 dpi images from the James Madison Papers. The file sizes are given in Kilobytes.

	CECAT (Error 0.75 / Error 1.0)			DjVu		
Page	Contours	Residual	Background	JB2 Bilevel	JB2 Colors	IW4
3	117 / 98	319 / 320	370 / 375	1292	57	1
8	124 / 101	352 / 353	356 / 359	1487	61	1
9	126 / 103	345 / 347	378 / 382	1171	52	1

Table 5.13: Comparison of "Hybrid" image layers for 200 dpi images from the US 1870 Census. The file sizes are given in Kilobytes.

	CECAT (Error 0.75 / Error 1.0)			DjVu		
Page	Contours	Residual	Background	JB2 Bilevel	JB2 Colors	IW4
8	156 / 126	380 / 382	696 / 696	1509	40	1
12	196 / 156	530 / 532	672 / 674	1772	72	1
15	194 / 151	631 / 635	608 / 607	2218	89	1

Table 5.14: Comparison of "Hybrid" image layers for 300 dpi images from the US 1870 Census. The file sizes are given in Kilobytes.

Although grayscale-to-bitonal conversions were not the emphasis of this thesis, poor binarization severely affects the usefulness of the CECAT contour layer. Letters can be chopped into disconnected pieces and sometimes entire words can be missing from the bitonal representation of the image. Admittedly, these missing pieces do reappear when the background layer is added to the image, but if the user is required to

wait for the final layer to transmit in order to read the document the progressive transmission strategy is marginalized.

The other limitation of the CECAT system is the 8-bit to 3-bit grayscale conversion. Because of this operation, fully downloaded CECAT images are lossy images, at least until further work is done to improve the compression of the residual and background layers.

Lastly, the CECAT images can take up to three minutes to compress. This may limit the usability of this compression strategy, especially for large collections that could take years to convert. Hopefully further improvements can speed up this process, especially since a large portion of the time is spent reading tiles from the original "uncompressed" image.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The Curve-Encoded Compression and Transmission (CECAT) system provides significant compression improvements to the bitonal foreground image layer, especially those containing large amounts of handwriting. The bitonal foreground layer of CECAT images were only 20% - 30% of the size of the JBIG and DjVuBitonal and yet still quite readable. This shows significant improvement. In addition, when binarization was good, this image layer has more fluid, continuous lettering with background noise removed by a "despeckling" operation.

To add readability and demonstrate the usefulness of the encoded images, the residual and background layers were encoded as 3-bit grayscale image data. As a result, a fully transmitted CECAT image shows the image data as it appears on the document (after this 8-to-3 bit quantization) without distortions or artifacts that appear on other lossy compression algorithms.

In addition, the layers created by the CECAT system facilitate progressive transmission functionality. Compared with the open source implementation of the popular DjVu standard, the bitonal foreground layer is much more readable and appropriate for browsing through multiple documents quickly. As an extra level of functionality, the CECAT image layers are segmented into 512 x 512 bit tiles which can be streamed to a viewer one piece at a time, providing another form of progressive transmission.

6.2 Future Work

The CECAT system introduces a novel method for compressing and transmitting document images. As is often the case with new approaches to old problems, new areas for study as well as further enhancements are made available.

One very important enhancement revolves around the binarization algorithm used to separate the foreground from the background. Because the intent of this thesis revolved around parametric compression and progressive transmission, the operation of converting grayscale image into good bitonal images was only lightly touched. However, the usefulness of the first "contour compressed" layer of CECAT images is determined by the effectiveness of the binarization algorithm. Many such operations have been developed throughout the past few years and this problem remains an active area of research. On a positive note, the CECAT system has been architected so that a new binarization operation can easily be swapped in, with the only change being a simple method call. One such operation is using an approach known as graph cut for segmenting text from background, rather than applying a thresholding algorithm. By seeding the foreground and background, good binarization can be achieved.

Another obvious enhancement involves the residual and background layers. Although eight color grayscale images are quite readable, there are better algorithms available for reducing the size of these two layers without reducing the color palette. These layers can easily be further compressed using sophisticated one-dimensional signal compression techniques such as an arithmetic encoder. Because some locality information is preserved in those layers, some two-dimensional encoding strategies might be useful as well. Future tests may even discover that only two layers of an image are needed, allowing the residual and the background layer to merge in some tightly compressed lossless format. At the very least, the simple gzip encoding done as a last step could be changed to a more effective arithmetic encoder. There are many possibilities enhancing the compression efficiency of these other layers, including a combination of CECAT foreground layer with the tightly compressed DjVu background layer.

As mentioned in *Section 3.3*, the currently implemented CECAT system only uses quadratic and linear Bezier curves. More experimentation could be done to determine if

there exists a better choice for this purpose. Although the gain between linear and quadratic curve representations turned out to be small, further gains might be possible if cubic or even higher-order Bezier curves are used. Another set of experiments could be performed to determine the value of using B-Splines, NURBS, or another parametric form. Because compression efficiency was more important than parametric curve connectivity, Bezier curves were chosen. The advantages of good curve connectivity may outweigh a slight increase in file size as these experiments may show.

Another enhancement, which was pursued lightly during the course of this thesis, was something akin to a shape library. The CECAT system combined vectorization (mapping lines and curves to contours) with codebook (segment library) compression strategies quite effectively. Another challenge faced by the CECAT system is the need for a good method for encoding small contours. Since the segment library successfully reduced the overall CECAT file size by about 5%, a good shape library may compress these images even farther.

Enhancements to the CECAT system are not the only avenues for future work. Having readable copies of document images stored as parametric curves makes new options available in the field of image manipulation. Because Bezier curves are affine invariant, scaling, translations, and rotation operations can be safely performed on the CECAT control points. Building a viewer to take advantage of this would be beneficial as a first step. Rotation and zooming operations would not require very intensive calculations in this case.

Image manipulation is not the only field of research than can benefit from using the CECAT system. Because shapes have been converted to parametric curves, it is possible to use those curves as a feature set to identify content in the image. Pattern recognition is always a difficult problem. At the extreme end, handwriting recognition may benefit from the sequence of encoded curve information the compressed contours can supply. In the short term, form recognition or other such operations could benefit from the additional features provided by the CECAT-encoded contours.

The CECAT server can also be developed further. The implementation of the server was only meant for demonstration purposes. The challenges associated with making a connection, streaming data, and adding image data into the viewer as it is

transferred have not been addressed during the course of this thesis. Third party software may provide a great fit here, such as the server used in the JITB system.

The CECAT viewer is also in its infancy. Only simple operations like 90 degree rotations and mirroring can be performed on the image while it is being displayed at the viewer. Tools such as a progress meter, pan window, and interactive zoom could go a long way to improve the overall browsing experience. In the best case, a browser plug-in could be developed to viewer CECAT images transmitted over http.

Resolving the issues mentioned above could advance the CECAT system, making it a much more powerful method for encoding and delivering document images across potentially low bandwidth connections for browsing operations.

Appendix A
Image Datasets

To test the effectiveness of the CECAT compression system, images from four different sources were taken, compressed, and compared. What follows are thumbnails and a brief description of each of these sets of images.

A.1 George Washington Papers

This first dataset was published by the Library of Congress and contains the collected writings of George Washington. This dataset provided a number of documents consisting of mostly handwriting. As such, these documents lay squarely in the "target" as it were of the CECAT compression system. Unfortunately, these images were JPEG images before performing the various compression tests, creating at least two generations of image degradation. Full details for the images in this dataset are as follows:

George Washington Papers at the Library of Congress, 1741-1799: Series 3a Varick Transcripts; George Washington to Continental Congress, July 10, 1775; http://memory.loc.gov/ammem/gwhtml/gwseries3.html (Subseries A Continental Congress LetterBook 1)
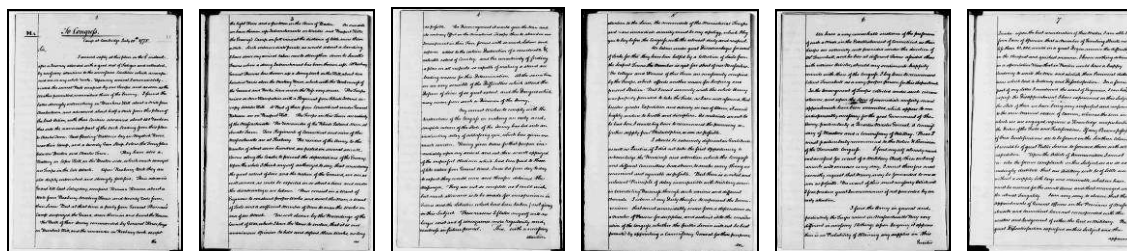
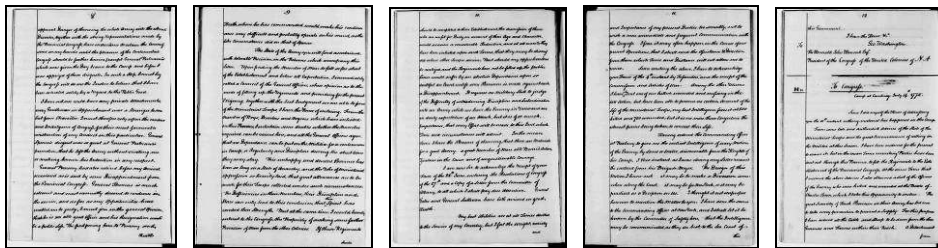Page 02   Page 03   Page 04   Page 05   Page 06   Page 07
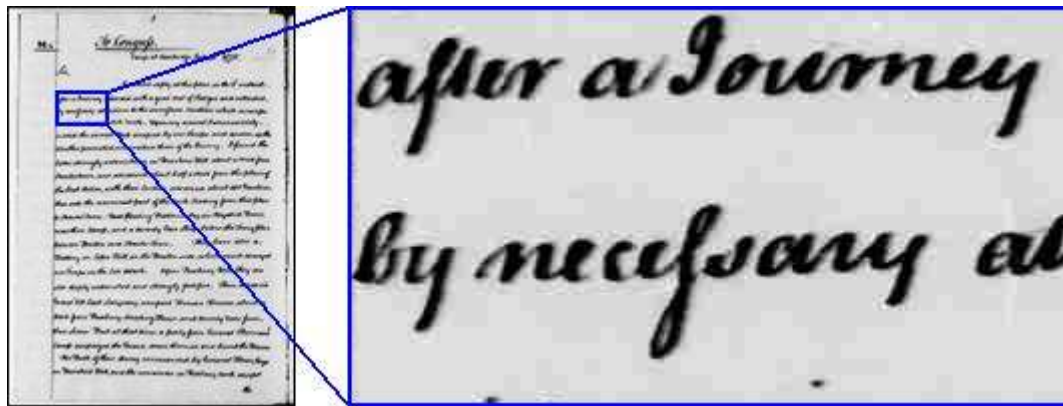
| Page 08 | Page 09 | Page 10 | Page 11 | Page 12 |

Full Resolution Snapshot of Page 02

A.2 James Madison Papers

The second dataset was also published by the Library of Congress, consisting of number of James Madison's writings. Like the George Washington Papers, these images consisting of mostly handwriting as well as the JPEG image degradation. Also, like the George Washington Papers, these documents lay squarely in the "target" area for the CECAT compression system. Full details for this collection are as follows:

The James Madison Papers; Series 3: Madison-Armstrong

Correspondence, 1813-1836; James Madison. Review 1824;

http://memory.loc.gov/ammem/collections/madison_papers/mjmser3.html;

Credit Line: Library of Congress, Manuscript Division.

| Page 11 | Page 12 | Page 13 | Page 14 | Page 15 |
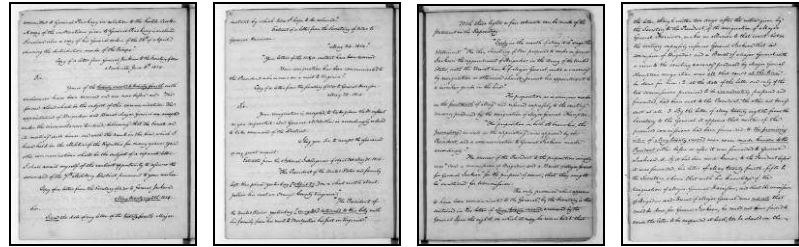
Page 16   Page 18   Page 19   Page 20
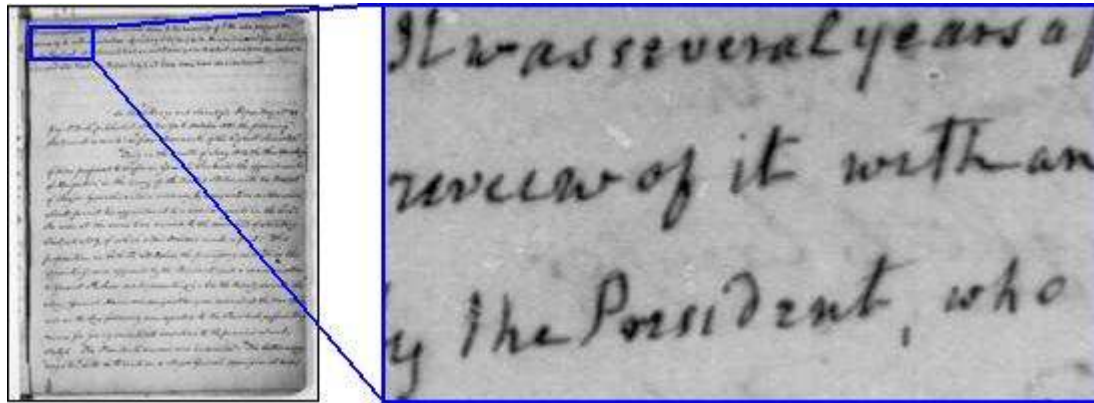
Full Resolution Snapshot of Page 11

A.3 US 1870 Census (200 dpi)

The third dataset consists of records from the 1870 United States Census. These images were scanned directly off microfilm and saved off as uncompressed images, reducing the amount of image degradation. In addition, these census images were saved at 200 dpi resolution. Although the Census form is not handwriting, it still compresses fairly well.

Population Schedules of the Ninth Census of the United States 1870;

National Archive Microfilm Publications; Roll 110, Connecticut Vol. 7,

New Haven County, New Haven City, Wards 4-8
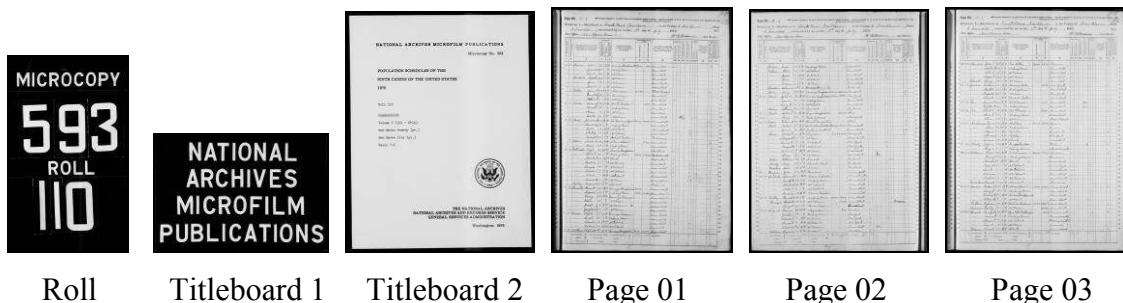
Roll   Titleboard 1   Titleboard 2   Page 01   Page 02   Page 03
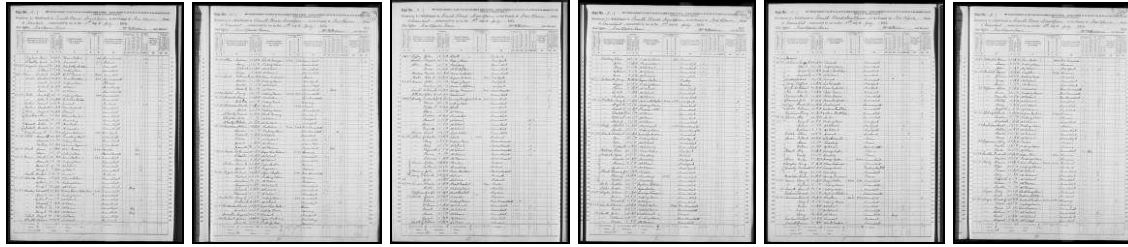
Page 04   Page 05   Page 06   Page 07   Page 08   Page 09
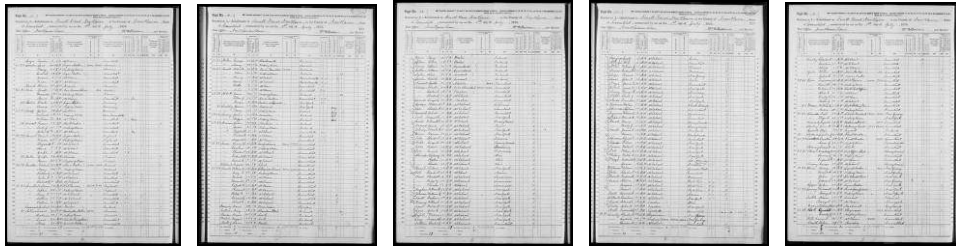
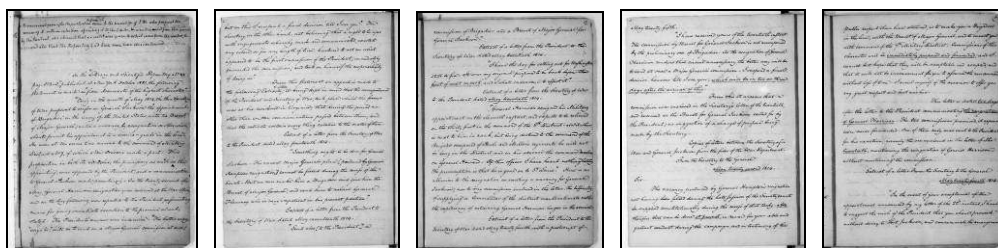Page 10   Page 11   Page 12   Page 13   Page 14
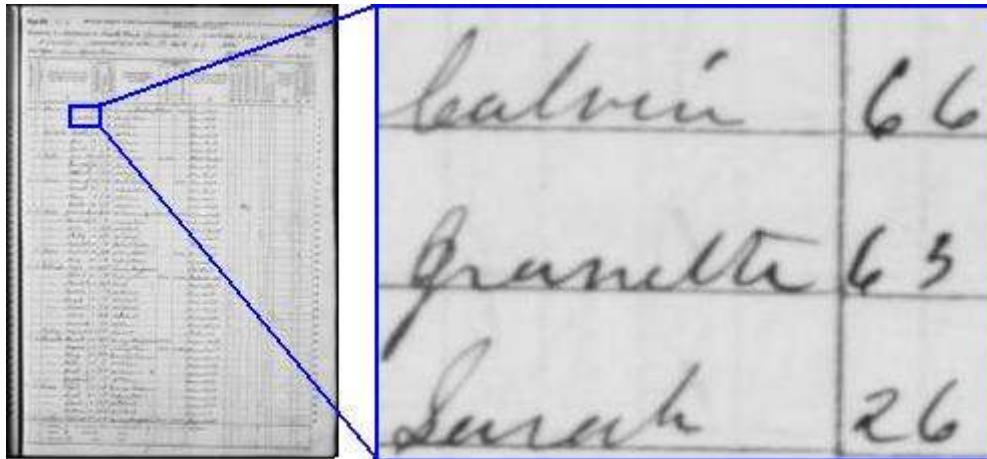
Full Resolution Snapshot of Page 01

A.4 US 1870 Census (300 dpi)

The fourth and final dataset consists of a few more pages from the 1870 United States Census, also scanned directly from microfilm. These images were saved at a resolution of 300 dpi.

Population Schedules of the Ninth Census of the United States 1870;

National Archive Microfilm Publications; Alabama, Jackson County

Page 08 Page 09 Page 10 Page 11 Page 12

Page 13 Page 14 Page 15
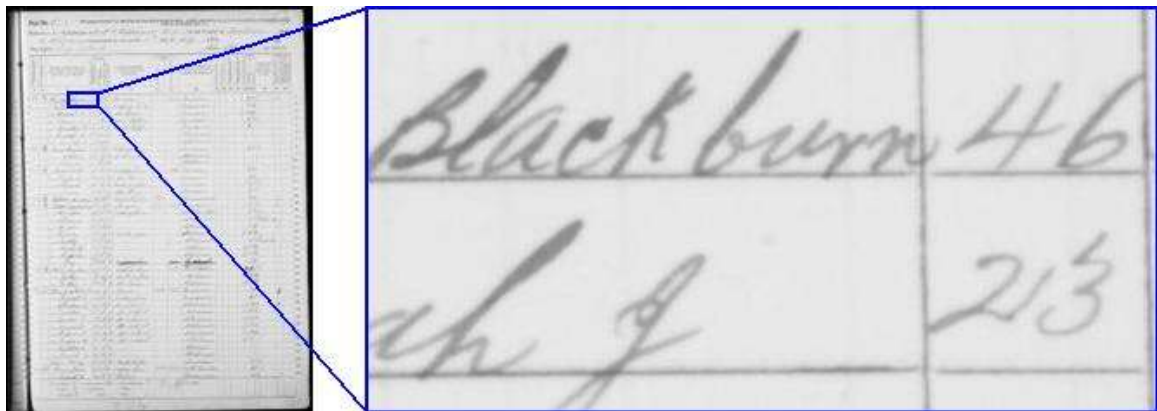
Full Resolution Snapshot of Page 08

Appendix B

User's Guide

Two graphical user interfaces where created to support the CECAT system, one to compress images and the other to view them. What follows is a summary of each of these interfaces as well as how to use them to perform their appropriate function.

B.1 Compression Interface

The primary purpose of this interface is to perform the actual CECAT compression on a tiled image. In addition, some methods have been added to allow the user to view contours as well as each layer of an image tile. The interface is simple consisting of an image viewer, a dropdown menu and a couple simple widgets.

File Menu

This menu offers basic options for opening and saving image files. The initial implementation supports the following image formats: jpeg, gif, png, ppm, pgm, and pbm.

Open First Tile:

This option allows the user to open and view the 512 x 512 pixel tile located in the upper-left corner of the image. This tile can then be compressed using different options from the *compression menu* and viewed at different levels using the *display* menu.

Open and Compress Tile Image:

This option runs the entire CECAT compression algorithm on an image file, creating all three layers using the parameters specified on the interface controls (error tolerance and minimum contour length). Simply put, to compress an entire image, use this option. The three different layers of the CECAT Image will be saved under the same name, in the same directory as the original file except that the extensions will be *cec*, *res*, and *bkg* for the CECAT layer, residual layer, and background layer respectively.

Encode Entire Image:

After selecting this option, the user is prompted to select an image. Once an image is selected, the CECAT compression will compress the entire image as one large tile (instead of segmenting them out into smaller tiles). Only the CECAT layer is created in this manner, and the *cec* file is saved in the same directory as the original file.

Quit:

This exits the compression interface.

Compression Menu

Once a single tile has been opened, this menu allows the user the opportunity to see the results of applying different types of CECAT compression approaches.

File	Compression	Display
	Line Compress	Alt-L
	Quad Compress	Alt-U
	Mixed Compress	Alt-M
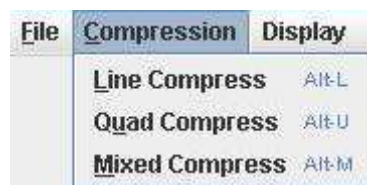
Line Compress:

This displays the contours that result from applying CECAT compression but restricting the curve mapping to line segments only.

Quad Compress:

This displays the contours that are rendered after applying CECAT compression with only quadratic Bezier curves (no line segments allowed).

Mixed Compress:

> This option allows the user to view the results of applying a normal CECAT operation to an open tile.

Display Menu

After a tile has been opened using the *File menu* and compressed using one of the options found in the *Compression menu*, this menu will give the user the opportunity to view different layers of the CECAT image.
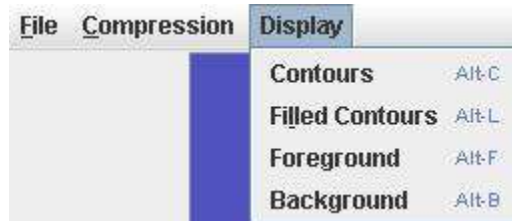
Contours:

> This option forces the display to show only the currently active contours. If a compression algorithm has been run, these contours are the result of the CECAT compression operation; otherwise, the results of the contour detection algorithm are displayed.

Filled Contours:

> By selecting this option, the display shows the results of applying the contour fill operation to the list of current contours (either CECAT compressed contours or the currently detected, uncompressed, contours). When applied to CECAT compressed contours, this option displays the foreground mask.

Foreground:
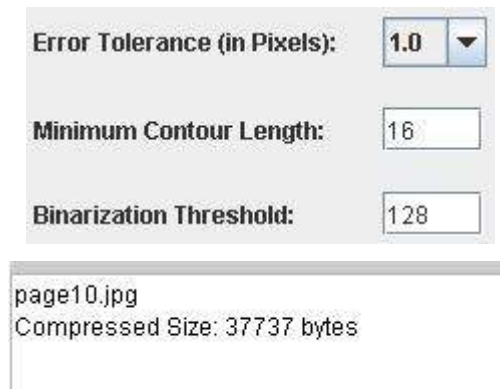
> Choosing this option displays the residual layer created during CECAT compression (assuming that a CECAT operation has already be performed).

Background:

> Choosing this option displays the background layer created during CECAT compression is displayed, assuming that a CECAT operation has already been performed on the current image.

CECAT Compression Controls

Only three parameters are currently exposed for changing the quality and size of CECAT compressed files: error tolerance, minimum contour length, and a global binarization minimum threshold. Three controls are present on the Compression Interface to allow the user to change these settings. Once the compression operation is complete, the name of the original file and the final size of the CECAT compressed layer are displayed in a text area.

B.2 CECAT Image Viewer

For the most part, the options offered by the CECAT viewer are self explanatory. Basic file open/save and rotation/mirroring operations make up most of the viewer's exposed functionality. The only unusual controls allow the user to request a different layer of the image from the server.

To view a CECAT image, simply open the image using the File menu and use the view window to scroll around the image. Each time the user looks at a new part of the image in the viewer, the appropriate tile is downloaded (of it does not already exist in memory). The viewer starts out displaying the CECAT-encoded foreground layer. If another image layer is requested, those tiles are downloaded to the viewer.

File Menu

This is another standard file menu with the standard open, save, and quit options available.

Open Image:

>This allows the user to specify a CECAT image to view. Note that JPEG, GIF, and PNG file formats are also supported in this viewer.

Save Current Image:

>By selecting this option, the user can save a JPEG, GIF, or PNG copy of the image currently displayed in the viewer.

Quit:

>This closes the CECAT viewer window and exits the system.

Edit Menu

This menu allows the user some basic control over the ninety degree rotation and the mirroring of the current image.
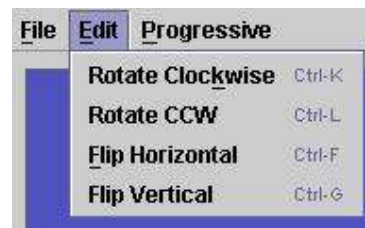
Rotate Clockwise:

>Selecting this option rotates the image currently displayed in the viewer ninety degrees clockwise.

Rotate CCW:

>Selecting this option rotates the image currently displayed in the viewer ninety degrees counter-clockwise.

Flip Horizontal:

>Selecting this option mirrors the image currently displayed in the viewer from left to right.

Flip Vertical:

Selecting this option mirrors the image currently displayed in the viewer from top to bottom.

Progressive Menu

This menu allows the user to simulate some of the progressive transmission features available through the CECAT compression format. By default, when a CECAT image is opened, the only layer currently viewable is the CECAT-encoded foreground layer. By using this menu, other layers (residual and background) can be viewed as well as single tiles can be requested from the server.

Get Next Layer:

This option sets the viewer to download and display the next layer of a CECAT encoded image. If the current layer is the foreground layer, the residual layer is downloaded after selecting this option. If the residual layer is currently being viewed, the background layer is downloaded. If tiles from the previous layers have not been downloaded from the server yet, they will be downloaded as needed before the residual or background layers tiles.

Get Next Tile:

Instead of scrolling around the image viewer, additional image tiles can be downloaded from the server by selecting this option. As a rule, tiles from the current layer will be downloaded first.

Appendix C

CECAT Code Base

C.1 CECAT Package

This is the root package for the CECAT system contains the GUI interfaces used to compress or view CECAT images.

CECATViewer

Description:

This is the first implementation of a CECAT image viewer. Details behind using this interface are given in Appendix B.

CompressionInterface

Description:

This provides a graphical interface for compressing jpg or pgm/ppm/pbm images into CECAT form. Appendix B contains details on how to use this application.

C.2 CECAT.compression Package

This package is primarily responsible for running the operations associated with detecting contours and mapping Bezier curves using the strategies outlined in Sections 3.2 and 3.3. Most of these classes possess static methods and might more easily be thought of as simple C style procedures. The only data object in this package is the ContourDetails class, which was a recent addition used to track the number of Beziers being mapped to each contour as part of an experiment.

BorderMarker

Description:

The methods contained inside this class have only one function, detecting where contours meet the edge of the tile/image. Essentially a static method, this operation uses a few helper functions to detect and return a list of spans indicating these locations. This class is primarily a "holding location" for all the code used in this operation (as opposed to a true object-oriented data class).

***Public Methods*:**

findEdges (static)

This is the primary (and only) operation performed by this class, which gets contour information and image boundaries as input. Edges are detected, marked, and stored off as an ArrayList of BezierMappings. Using these mappings as "fixed" parametric curves, seamless connection from one tile to another is achieved.

Inputs:

contour (Contour) – the contour to be tested

maxX (int) – coordinate for the right edge of the tile/image

maxY (int) – coordinate for the left edge of the tile/image

Output:

ArrayList of cecat.contour.BezierMappings representing first degree Bezier curves lying on the edge of the tile or image

ContourDetails

Description:

This is a simple data object stores the count of how many first, second, and third degree Bezier curves are needed to represent a contour. This is returned as output from a contour compression operation.

***Public Variables*:**

lineCount (int) – number of first degree Bezier curves used

quadCount (int) – number of second degree Bezier curves used

cubicCount (int) – number of third degree Bezier curves used

Public Methods:

Constructors

The constructors available for this object allow you to initialize all the public variables or allow them to default to zero.

ContourDetector

Description:

All contour detection operations run from inside this class. Unlike other operators in this package, current implementation requires a constructor followed by a call to the method "getContours". Details behind the strategy for breaking down the image into "layers" and detecting them are outlined in Section 3.2.

*Public Methods***:**

Constructor

This constructor takes an image (in the form of a byte array) and computes the image height, width, and initialized the detected contour array.

Input:

image (byte[][]) – image from where the contours are to be detected

getContours

Contours are detected using the image data entered in the constructor detecting a contour, filling it in, and creating a map. This map represents the first contour "layer". After all the contours on that layer are detected, the map is used to create an image of the next contour "layer" and the process repeats. This operation is described in detail in Section 3.2.

Output:

ArrayList of Contour objects representing the contours detected for this operation.

ContourLineFitter

Description:

This is the outward face for a CECAT compression strategy described in detail in Section 3.3.2. Mapping only first degree Bezier curves (line segments) to the contour, this class is somewhat limited in its capabilities but provides useful

experiments. The Contour objects inputted into this class are assumed to have their *points* array set, as this class maps Beziers to these points and stores the resulting Beziers in the Contour's *curves* array.

***Public Methods*:**

fitBorderContours (static)

One of multiple strategies for running this first degree Bezier mapping strategy, this one goes through the process of detecting and setting the border segments before going through the process of mapping line segments to the rest of the Contour. The results of this operation are stored in the Contours' *curves* arrays, thus no actual output exists.

Input:

contours (ArrayList) – list of contours to be processed

maxX (int) – coordinate for right edge of the tile/image

maxY (int) – coordinate for bottom edge of the tile/image

errorTolerance (double) – error (in pixels) allow by the CECAT mapping system

minimumContourSize (int) – contours consisting of fewer pixels than this are not processed

fitContours (static)

Another strategy for mapping first degree Beziers to contours, this strategy ignores border cases and simply goes through the process of mapping line segments each piece of the Contour. The results of this operation are stored in the Contours' *curves* arrays, thus no actual output exists.

Inputs:

contours (ArrayList) – list of contours to be processed

errorTolerance (double) – error (in pixels) allow by the CECAT mapping system

minimumContourSize (int) – contours consisting of fewer pixels than this are not processed

fitEntireContour (static)

Although used by the previous two methods, this method can be called by itself to map line segments to a single contour using the CECAT mapping strategy outlined in *Section 3.3.2*.

Inputs:

contours (Contour) – contour to be processed

errorTolerance (double) – error (in pixels) allow by the CECAT mapping system

Output:

CompressionDetails describing how many line segments were needed to represent the contour.

fitContourSections (static)

This third contour compression method performs the CECAT compression on one contour using first degree Bezier curves; however, it allows an outside operation to determine and "fix" particular Beziers to the contour. The **fitBorderContours** operation uses this method, although other strategies could be developed which use other "fixed" Bezier curves.

Input:

contour (Contour) – contour to be processed

errorTolerance (double) – error (in pixels) allow by the CECAT mapping system

fixedCurves (ArrayList) – a list of BezierMappings describing which sections of contour are to be replaced by which Bezier curve

ContourMixtureFitter

Description:

This is the outward face for a CECAT compression strategy described in detail in Section 3.3.4. This class maps both first and second degree Bezier curves to the contour, creating a CECAT compressed image (or tile). The Contour objects inputted into this class are assumed to have their *points* array set,

as this class maps Beziers to these points and stores the resulting Beziers in the Contour's *curves* array.

Public Methods:

fitBorderContours (static)

One of multiple strategies for running this Bezier mapping strategy, this one goes through the process of detecting and setting the border segments before going through the process of mapping line segments or quadratic Bezier curves to the rest of the Contour. The results of this operation are stored in the Contours' *curves* arrays, thus no actual output exists.

Input:

contours (ArrayList) – list of contours to be processed

maxX (int) – coordinate for right edge of the tile/image

maxY (int) – coordinate for bottom edge of the tile/image

errorTolerance (double) – error (in pixels) allow by the CECAT mapping system

minimumContourSize (int) – contours consisting of fewer pixels than this are not processed

fitContours (static)

Another strategy for mapping first and second degree Beziers to contours, this strategy ignores border cases and simply goes through the process of mapping line segments and quadratic Bezier curves to each piece of the Contour. The results of this operation are stored in the Contours' *curves* arrays, thus no actual output exists.

Inputs:

contours (ArrayList) – list of contours to be processed

errorTolerance (double) – error (in pixels) allow by the CECAT mapping system

minimumContourSize (int) – contours consisting of fewer pixels than this are not processed

fitEntireContour (static)

Although used by the previous two methods, this method can be called by itself to map Bezier curves to a single contour using the CECAT mapping strategy outlined in Section 3.3.4.

> *Inputs*:
>
>> contours (Contour) – contour to be processed
>>
>> errorTolerance (double) – error (in pixels) allow by the CECAT mapping system
>
> *Output*:
>
>> CompressionDetails describing how many line segments were needed to represent the contour.

ContourQuadraticFitter

Description:

This is the outward face for a CECAT compression strategy described in detail in Section 3.3.3. Mapping only second degree Bezier curves (quadratics) to the contour, this class is somewhat limited in its capabilities but provides useful experiments. The Contour objects inputted into this class are assumed to have their *points* array set, as this class maps Beziers to these points and stores the resulting Beziers in the Contour's *curves* array.

***Public Methods*:**

fitBorderContours (static)

One of multiple strategies for running this second degree Bezier mapping strategy, this one goes through the process of detecting and setting the border segments before going through the process of mapping quadratic Bezier curves to the rest of the Contour. The results of this operation are stored in the Contours' *curves* arrays, thus no actual output exists.

> *Input*:
>
>> contours (ArrayList) – list of contours to be processed
>>
>> maxX (int) – coordinate for right edge of the tile/image
>>
>> maxY (int) – coordinate for bottom edge of the tile/image

errorTolerance (double) – error (in pixels) allow by the
CECAT mapping system

minimumContourSize (int) – contours consisting of fewer
pixels than this are not processed

fitContours (static)

Another strategy for mapping second degree Beziers to contours,
this strategy ignores border cases and simply goes through the process of
mapping quadratic Bezier curves to each piece of the Contour. The results
of this operation are stored in the Contours' *curves* arrays, thus no actual
output exists.

Inputs:

contours (ArrayList) – list of contours to be processed

errorTolerance (double) – error (in pixels) allow by the
CECAT mapping system

minimumContourSize (int) – contours consisting of fewer
pixels than this are not processed

fitEntireContour (static)

Although used by the previous two methods, this method can be
called by itself to map quadratic Bezier curves to a single contour using
the CECAT mapping strategy outlined in Section 3.3.3.

Inputs:

contours (Contour) – contour to be processed

errorTolerance (double) – error (in pixels) allow by the
CECAT mapping system

Output:

CompressionDetails describing how many line segments
were needed to represent the contour.

fitContourSections (static)

This third contour compression method performs the CECAT
compression on one contour using second degree Bezier curves; however,
it allows an outside operation to determine and "fix" particular Beziers to

the contour. The **fitBorderContours** operation uses this method, although other strategies could be developed which use other "fixed" Bezier curves.

> *Input*:
>
>> contour (Contour) – contour to be processed
>>
>> errorTolerance (double) – error (in pixels) allow by the CECAT mapping system
>>
>> fixedCurves (ArrayList) – a list of BezierMappings describing which sections of contour are to be replaced by which Bezier curve

C.3 CECAT.contour

The contour package contains a number of data objects used to represent contours, Beziers, and points. These classes are used throughout the CECAT code base to store and transfer information used to represent these fundamental units of a contour-compressed file.

Bezier

Description:

This is the "in memory" representation of a line, quadratic, or cubic Bezier curve. In this data object are the degree of the Bezier curve represented and the coordinates of the control points used.

*Public Variables***:**

degree (int) – degree of the Bezier curves represented

x0, x1, x2, x3 (double) – x-coordinates for the Bezier curves represented

y0, y1, y2, y3 (double) – x-coordinates for the Bezier curves represented

Public Methods:

Constructors

The four constructors available for this object allow you to initialize an empty data object or set up a first, second, or third degree Bezier by initializing 2, 3, or 4 control points.

BezierComparator

Description:

This is an extremely simple Java comparator used to sort lists of BezierMapping objects according to their position in the contour.

Public Methods:

compare

This method is used indirectly when a collection of BezierMapping objects are sorted. It takes two BezierMappings compares their *lowerPointIndex* variables, returning a sort priority accordingly.

BezierMapping

Description:

In most cases, this is a wrapper for a Bezier object used to map a Bezier curve to the points contained on a contour. By using upper and lower indices to the list of points in a contour, this mapping is accomplished. This allows Bezier curves to be sorted in order of the contour and compared during the CECAT mapping process.

*Public Variables***:**

upperPointIndex (int) – point on the contour mapping to the last control point of the Bezier curve

lowerPointIndex (int) – point on the contour mapping to the first control point of the Bezier curve

curve (Bezier) – Bezier curve that has been mapped to the contour

Public Methods:

Constructor

Only one constructor is available for this data object, which is used to initialize the Bezier curve and both indices.

CECATImage

Description:

The CECATImage object can holds all the data contained in the first layer of a CECAT compressed file. To be more specific, general information about the image and tile dimensions coupled with the contours contained on those tiles is stored in this data object.

*Public Variables***:**

> **contours** (ArrayList[][]) – lists of contours for each tile whose coordinates
> are identified by the 2D array indices
>
> **width** (int) – width of the entire image in pixels
>
> **height** (int) – height of the entire image in pixels
>
> **tileWidth** (int) – width, in pixels, of each tile in the CECAT image
>
> **tileHeight** (int) – height, in pixels, of each tile in the CECAT image

Public Methods:

> **Constructor**
>
>> Only one constructor is available for this data object, which is used
>> to initialize the contour lists and the CECAT image/tiles dimensions.

Contour

Description:

> Entire contours, whether they are represented by a list of points or a list of
> Bezier curves, are encapsulated in this data object. In addition to acting as a
> simple data object, there are a couple methods used to generate statistics about the
> contour or convert Bezier curves to a list of points.

*Public Variables***:**

> **internal** (boolean) – flag marking the contour as surrounding a black or
> white connected component
>
> **points** (ArrayList) – list of PixelPoints marking each point on the contour
>
> **curves** (ArrayList) – list of Beziers used to represent the contour
>
> **maxY** (int) – y-coordinate the bottommost pixel on the contour
>
> **maxX** (int) – x-coordinate of the rightmost pixel on the contour
>
> **minX** (int) – x-coordinate the leftmost pixel on the contour
>
> **minY** (int) – y-coordinate the topmost pixel on the contour

Public Methods:

> **Constructor**
>
>> Only one constructor is available for this data object, which
>> initializes a contour as internal or not and takes a list of PixelPoints or
>> Beziers, initializing the appropriate list in the contour. If a list of

PixelPoints is used to initialize the contour, the min/max variables are determined and set.

calculatePointsFromCurves

If the contour is represented by Bezier curves, this operation converts them into PixelPoints, filling the appropriate list and setting the min/max variables in the process.

countCurves

Although used by the previous two methods, this method can be called by itself to map line segments to a single contour using the CECAT mapping strategy outlined in *Section 3.3.2*.

Inputs:

degree (int) – degree of Bezier to be used in the count

Output:

Int identifying the number of Bezier curves of the specified degree used to represent this contour.

HorizontalSpan

Description:

The process of filling contours accurately requires small data objects that represent horizontal spans of pixels encompassed by these contours. Each "span is represented by this data object.

*Public Variables***:**

start (int) – relative x-coordinate for the start of this span

end (int) – relative x-coordinate for the end of this span

yValue (int) – absolute y-coordinate for this span

xValue (int) – absolute x-coordinate for the start of this span

Public Methods:

Constructor

Only one constructor is available for this data object, which is used to initialize the absolute x and y coordinates for the start of this span.

PixelPoint

Description:

To prevent the use of the large java Point class, this simple data object was created to hold a pair of x and y coordinates representing a pixel on an image.

***Public Variables*:**

x (int) – x-coordinate for this pixel

y (int) – y-coordinate for this pixel

Public Methods:

Constructor

Only one constructor is available for this data object, which is used to initialize the x and y coordinates for this pixel.

C.4 CECAT.decoder

All the algorithms used to decode the different layers of a CECAT file are included in this package. In addition to this, the initial implementation of the CECAT server (which essentially decodes, indexes, and sends out portions of images to an image viewer). On top of this, a utility used to read data in bit-sized portions from a data source appears in this package.

BackgroundDecoder

Description:

As mentioned in Section 4.2.3, the background layer of a CECAT image consists of three bit values stored for each pixel not contained or adjacent to pixels in the foreground mask. This class contains methods used to decode this background layer using the foreground mask as a guide to determine the coordinates of these pixels.

Public Methods:

decodeBackground (static)

This method is used to decode a background tile that has been completely transferred.

Inputs:

mask (byte[][]) – foreground mask for the CECAT image

encodedImage (byte[]) – background layer for CECAT image

decodeBackground (static)

This method is just like the previous method except that the image data currently resides in a file instead of in memory.

Inputs:

mask (byte[][]) – foreground mask for the CECAT image

decoder (FileBitDecoder) – reader which reads background CECAT data from a file

BitDecoder (interface)

Description:

Simply put, this in an interface for a file input stream, designed to allow access to individual bits. These bits can be read one-by-one or many at a time. If more than one is read at once, the results are converted into integer value.

Public Methods:

close

This closes the file input stream and cleans up the connection.

startMeasurement

This method sets a flag that starts measuring the number of bits that have been read from the input stream.

stopMeasurement

This method stops the process of measuring the number of bits that have been read from the input stream and reports.

Output:

Long describing how many bits have been read from this input stream since measuring has begun.

getBufferData

When a large amount of raw data is needed from a data file, this method is used.

Inputs:

size (long) – number of bits to read from the data file

Output:

Byte[] with the requested data from the input stream.

getData

Unlike the previous method, this method reads a smaller number of bits and translates it into a single number before sending it back. This is used for reading a single piece of data from the input stream.

Inputs:

numBits (int) – number of bits to read for the data file

Output:

Int translated from the data from the input stream.

CECATDecoder

Description:

When a CECAT-encoded foreground layer needs to be decoded, this class must be used. It contains all the methods associated with transforming Bezier curves into their representative contours and eventually shapes.

Public Methods:

decodeEntireImage (static)

Taking an input file, this method is used for decoding an entire CECAT-encoded foreground layer (without regard to tiles).

Inputs:

in (BitDecoder) – data stream from a CECAT file

Output:

CECATImage that holds image data and statistics used for displaying a grayscale representation of the CECAT image.

decodeNextTile (static)

This method is used to decode the next tile from a CECAT file.

Inputs:

in (BitDecoder) – data stream from a CECAT file

tileSizeX (int) – width (in pixels) of the tiles used

tileSizeY (int) – height (in pixels) of the tiles used

Output:

ArrayList of contours found on the CECAT tile.

measureEntireImage (static)

This method returns the file size of a CECAT-encoded foreground layer.

Inputs:

in (BitDecoder) – data stream from a CECAT file

tileSizeX (int) – width (in pixels) of the tiles used

tileSizeY (int) – height (in pixels) of the tiles used

Output:

Long showing the file size of the CECAT foreground layer.

CECATImageServer

Description:

To demonstrate the CECAT progressive transmission strategy, this simple "server" was implemented. While not a server in any sense of the word, this class pretends by reading and indexing a CECAT file and sending requested tile data to image viewers. This class was described in Section 4.4.1.

*Public Variables***:**

width (int) – width (in pixels) of the CECAT image

height (int) – height (in pixels) of the CECAT image

tileWidth (int) – width (in pixels) of the tiles used by the CECAT image

tileHeight (int) – height (in pixels) of the tiles used by the CECAT image

tileSizeX (int) – width (in pixels) of the tiles used by the CECAT image

tileSizeY (int) – height (in pixels) of the tiles used by the CECAT image

tilesX (int) – number of tiles

tilesY (int) – number of tiles

Public Methods:

Constructor

This class has a simple constructor: a one parameter method that tells the "server" where the CECAT file is by passing in a filename.

Inputs:

fileName (String) – name of the CECAT file to open

getCECATTile

Using simple coordinates, this method sends a tile from the CECAT-encoded foreground layer to the attached viewer.

Inputs:

col (int) – index to the column where the CECAT-encoded tile can be found

row (int) – index to the row where the CECAT-encoded tile can be found

Output:

Byte[] holding the CECAT-encoded representation of the desired tile. A decoding operation must be performed on this data to recreate the foreground mask.

getResidualTile

Using simple coordinates, this method sends a tile from the encoded residual layer to the attached viewer.

Inputs:

col (int) – index to the column where the residual tile can be found

row (int) – index to the row where the residual tile can be found

Output:

Byte[] holding the residual of the desired tile. A decoding operation must be performed on this data to recreate the residual layer completely (pixels are still stored in three bits).

getBackgroundTile

Using simple coordinates, this method sends a tile from the encoded background layer to the attached viewer.

Inputs:

col (int) – index to the column where the background tile can be found

be found

Output:

Byte[] holding the background of the desired tile. A

decoding operation must be performed on this data to

recreate the background layer completely (pixels are still

stored in three bits).

ContourFiller

Description:

The contour filling algorithm described in Section 3.2.2 is implemented by

static methods contained in this class. Each method requires image data and

contours as input, returning image data for a "filled contour".

Public Methods:

fillSingleContour (static)

Although this method is used by the "fillAllContours()", it can be

used separately to simply fill a single contour.

Inputs:

image (byte[][]) – image data where the contour is to be

filled

filledContour (Contour) – contour to be filled

color (int) – grayscale value the contour is to be filled with

Output:

Byte[][] representing the image data with the filled contour.

fillAllContours (static)

This method fills a list of contours and returns image data with the

results of this operation.

Inputs:

contours (ArrayList) – collection of contours to be filled

image (byte[][]) – image data where the results of the filled

contours is stored

Output:

Byte[][] representing the image data with all the contours filled.

ResidualDecoder

Description:

As mentioned in Section 4.2.2, the residual layer of a CECAT image consists of three bit values stored for each pixel contained or adjacent to pixels in the foreground mask. This class contains methods used to decode this "residual" layer using the foreground mask as a guide to determine the coordinates of these pixels.

Public Methods:

decodeResidual (static)

This method is used to decode a residual tile that has been completely transferred.

Inputs:

mask (byte[][]) – foreground mask for the CECAT image

encodedImage (byte[]) – residual layer for CECAT image

decodeResidualFromFile (static)

This method is just like the previous method except that the image data currently resides in a file instead of in memory.

Inputs:

mask (byte[][]) – foreground mask for the CECAT image

decoder (FileBitDecoder) – reader which reads residual CECAT data from a file

C.5 CECAT.decoder.io

This package contains different implementations of the BitDecoder class, each implementing readers for a different source of data.

ArrayBitDecoder (implements *CECAT.decoder.BitDecoder*)

Description:

This implementation of BitDecoder allows bits to be extracted from an array of bytes.

FileBitDecoder (implements *CECAT.decoder.BitDecoder*)

Description:

This implementation of BitDecoder opens a stream to a file and extracted bits from there.

C.6 CECAT.encoder

All the algorithms used to encode the different layers of a CECAT file are included in this package. This includes those processes described in Section 3.3, some simple utilities used to facilitate these operations, and methods to encode the file formats described in Section 4.2. On top of this, a utility used stream data into bit-sized portions and place them in a data file appears in this package. Internally, this encoder uses a gzip compression algorithm as a final step when saving off a background file.

BackgroundEncoder

Description:

This class contains a number of static methods used to encode the background layer of an image and create a 'background layer' image data file. Of course, this requires access to the foreground mask encoded as a CECAT file first, but once that is in place, the methods provided in this class can do the rest.

Public Methods:

encodeEntireImageAsTiles (static)

The primary operation of this class is brought into effect using this method, which performs the background layer encoding operation for an entire image.

Inputs:

fileName (String) – name of the file where the background layer is to be saved

cecatImageStream (ImageReader) – image data stream from a CECAT-encoded foreground image layer

120

originalImageStream (ImageReader) – image data stream from the original grayscale image

Output:

Int denoting the size-encoded background image layer (in bytes).

encodeTile (static)

Although primarily used internally, this method encodes a single tile from an image and returns the encoded results as a data array.

Inputs:

image (byte[][]) – grayscale representation for the tile

mask (byte[][]) – foreground mask associated with this tile

Output:

Byte[] used to hold the encoded background layer for the image tile in question.

CECATEncoder

Description:

As described in Section 3.3, this class performs the CECAT encoding operation, creating a CEC file containing the compressed foreground layer of the encoded image. It only contains two static methods, one for encoding the foreground layer using tiles and one without tiles.

Public Methods:

encodeEntireImageAsTiles (static)

This method performs a CECAT encoding operation using 512x512 pixel tiles.

Inputs:

fileName (String) – name of the file to which the CECAT data will be saved

imageStream (ImageReader) – image data input stream

errorTolerance (double) – number of pixels a mapped Bezier can be off from the absolute contour

minimumContourSize (int) – only contours with a size equal to or greater than this will be compressed

Output:

Int describing the size of the CECAT encoded foreground layer.

encodeEntireImage (static)

This method performs the CECAT compression operation treating the entire image as one large tile. Unlike the previous method, this requires the contours to be detected prior to the encoding operation and passed to it.

Inputs:

fileName (String) – name of the file to which the CECAT data will be saved

imageWidth (int) – width of the image in pixels

imageHeight (int) – height of the image in pixels

contours (ArrayList) – contours found on the image

Output:

Int describing the size of the CECAT encoded foreground layer.

EncodingUtilities

Description:

These utilities are used by both the BackgroundEncoder and the ResidualEncoder to distinguish between the two layers and convert the 8-bit grayscale values into smaller 3-bit values.

*Public Variables***:**

eightGrayValues (int[]) – (static) u{1, 36, 72, 108, 144, 180, 216, 254};

Public Methods:

findClosestGrayLevel8 (static)

This method takes a pixel intensity value and returns the closest intensity stored in the "eightGrayValues" array.

Inputs:

pixelValue (byte) – pixel intensity value

Output:

Byte representing the index to which of the
"eightGrayValues" the input pixel intensity value is closest.

isMasked (static)

This utility uses the mask and coordinates to determine if a
particular pixel is part of the foreground mask (or adjacent to it).

Inputs:

mask (byte[][]) – foreground mask applied to the image

xPosition (int) – x-coordinate for the pixel in question

yPosition (int) – y-coordinate for the pixel in question

Output:

Boolean value indicating if the pixel is part of or adjacent
to the foreground mask.

FileBitEncoder

Description:

This utility class is used to write data to a file bit-by-bit. This means that
integers, booleans, and strings of 1's and 0's can be written to a data file using a
minimum number of bits. In addition, integer data can be "padded" with zeros,
which cab force a particular encoding size.

*Public Variables***:**

size (int) – number of bits written to the data file (can be reset at any time)

currentLocation (byte) – current location inside the current byte (1-8
bits)

currentFilePosition (long) – total number of bits written to the data file

Public Methods:

Constructor

This constructor initializes the data stream (applying a gzip output
stream where necessary) and attaches it to a file, the name of which is
passed in as its only parameter.

save

This method finishes the data file by padding the last byte with zeros and closes out the data stream.

    Output:

        <u>Long</u> describing the total number of bits written to the data file.

addBit

    This method writes a single bit (0 or 1) to the data file.

    Inputs:

        <u>bit</u> (boolean) – bit to write out to the file

addByte

    This method writes an entire byte to the data file.

    Inputs:

        <u>newData</u> (byte) – byte to write out to the file

addInt

    This method writes an integer to the data file using the minimum number of bits to represent it.

    Inputs:

        <u>newData</u> (int) – integer value to write out to the file

addInt

    This method writes an integer to the data file; however, it forces the number of written bits to be a particular size, by padding the number with leading zeros. This is the primary method used by the CECAT encoder to store data into a file.

    Inputs:

        <u>newData</u> (int) – integer value to write out to the file

        <u>absoluteSize</u> (int) – number of bits to use to write the integer

        <u>DEBUG</u> (boolean) – flag which prints out the data to the screen (for debugging purposes)

addString

    This method writes a string of 1's and 0's as bits to the data file.

> newData (String) – string to write out to the file

ResidualEncoder

Description:

This class contains a number of static methods used to encode the residual layer of an image and create a 'residual layer' image data file. Of course, this requires access to the foreground mask encoded as a CECAT file first, but once that is in place, the methods provided in this class can do the rest. Internally, this encoder uses a gzip compression algorithm as a final step when saving off a residual file.

Public Methods:

encodeEntireImageAsTiles (static)

The primary operation of this class is brought into effect using this method, which performs the residual layer encoding operation for an entire image.

> *Inputs*:
>
> > fileName (String) – name of the file where the residual layer is to be saved
> >
> > cecatImageStream (ImageReader) – image data stream from a CECAT-encoded foreground image layer
> >
> > originalImageStream (ImageReader) – image data stream from the original grayscale image
>
> *Output*:
>
> > Int denoting the size-encoded residual image layer (in bytes).

encodeTile (static)

Although primarily used internally, this method encodes a single tile from an image and returns the encoded results as a data array.

> *Inputs*:
>
> > image (byte[][]) – grayscale representation for the tile
> >
> > mask (byte[][]) – foreground mask associated with this tile

Output:

> Byte[] used to hold the encoded residual layer for the image tile in question.

C.7 CECAT.images

This package contains most of the utility methods used to read image data from an image file. In addition to input streams, the data structures used inside the CECAT code base to represent images and tiles are included in this package as well.

ImageReader (interface)

Description:

> This interface provides outside access to the contents of an image file using a variety of different methods. Because different image formats gather this information using a variety of encoders, this interface was made as generic as possible.

Public Methods:

getHeight

> This 'getter' returns the total height of the image.
>
> *Output*:
>
> > Int representing the height of the image.

getWidth

> This 'getter' returns the total width of the image.
>
> *Output*:
>
> > Int representing the width of the image.

getTileHeight

> This 'getter' returns the height of the tiles used by this image.
>
> *Output*:
>
> > Int representing the height of the image tiles.

getTileWidth

> This 'getter' returns the width of the tiles used by this image.
>
> *Output*:
>
> > Int representing the width of the image tiles.

getImageContours

This 'getter' returns a list of contours found in the image.

Output:

ArrayList containing Contour objects corresponding to each contour found in the image.

getEntireImage

This 'getter' returns the image as a large array of grayscale pixel intensity values.

Output:

Byte[][] containing all the grayscale pixel data values found on the image.

getEntireImage

This 'getter' returns the image as a large array of grayscale pixel intensity values rotated and/or mirrored using a particular orientation code.

Inputs:

orientation (String) – two-character orientation code which describe which corners of the image are found at the top corners of the viewer

Output:

Byte[][] containing all the grayscale pixel data values found on the image.

getImageRegion

Used primarily for extracting tiles from an image, this method returns an array of grayscale pixel intensities for a specified region of the image.

Inputs:

x (int) – x-coordinate of the upper left corner of the region of interest

y (int) – x-coordinate of the upper left corner of the region of interest

width (int) – width of the region of interest

127

height (int) – height of the region of interest

Output:

Byte[][] representing the grayscale pixel data values found in the region of interest on the image.

getImageRegion

Used primarily for extracting tiles from an image, this method returns an array of grayscale pixel intensities for a specified region of the image, which has been rotated and/or mirrored according to a particular orientation code.

Inputs:

x (int) – x-coordinate of the upper left corner of the region of interest

y (int) – x-coordinate of the upper left corner of the region of interest

width (int) – width of the region of interest

height (int) – height of the region of interest

orientation (String) – two-character orientation code which describe which corners of the image are found at the top corners of the viewer

Output:

Byte[][] representing the grayscale pixel data values found in the region of interest on the image.

setTileSize

Tile sizes (although they default to 512x512) are adjustable using this method.

Inputs:

width (int) – new width for each tile in pixels

height (int) – new height for each tile in pixels

getNextLayer

By calling this method, the ImageReader begins to transfer the next CECAT layer (residual or background). If the image is not a CECAT encoded image, this does nothing.

getNextTile

This is the front-end to a simple tile iterator used to get tiles in order from left to right, top to bottom.

Output:

ImageTile object containing the next tile and its associated graphic.

getTile

This method allows outside access to any particular tile given its row and column.

Inputs:

row (int) – row where the requested tile is found

col (int) – column where the requested tile is found

Output:

ImageTile object containing the requested tile and its associated graphic.

getNextRawTile

Same as getNextTile, this is the front-end to a simple tile iterator. The only difference is that the Java-viewable graphic is not generated.

Output:

RawImageTile object containing the next tile.

getRawTile

Same as getTile, this allows outside access to a specific tile. The only difference is that the Java-viewable graphic is not generated.

Inputs:

row (int) – row where the requested tile is found

col (int) – column where the requested tile is found

Output:

ImageTile object containing the requested tile and its associated graphic.

resetTileCounters

This restarts the tile counter used by getNextTile and getNextRawTile to run the tile iterator.

close

This closes the data input streams and frees up the resources.

ImageReaderFactory

Description:

The purpose of this class is to provide a central location from which to find the appropriate ImageReader for any image file type (even if that ImageReader is the UnsupportedImageReader).

Public Methods:

getImageReader

Using the file extension, this method determines the appropriate ImageReader, instantiates one, and passes it back to the calling method.

Input:

fileName (String) – name of the file where the image is stored

Output:

ImageReader that can be used to access the image.

ImageTile

Description:

This data object stores coordinate, pixel data, and a Java-viewable image object used to render an image tile. This information is used primarily by Java-based image viewers to display the tile to a user.

*Public Variables***:**

xPosition (int) – absolute x-coordinate of the upper left corner of the tile with respect to the original image

yPosition (int) – absolute y-coordinate of the upper left corner of the tile with respect to the original image

image (byte[][]) – image grayscale pixel data

graphic (Image) – java-based image object used to display the image

gridX (int) – column where the tile is found on the original image

gridY (int) – row where the tile is found on the original image

width (int) – width of the tile

height (int) – height of the tile

Public Methods:

Constructor

This constructor uses a RawImageTile data object to initialize all the variables and create the java-based image object for viewing purposes.

RawImageTile

Description:

This data object stores coordinate and raw pixel data about an image tile. This information is used by encoders, image processing operations, and viewers.

*Public Variables***:**

xPosition (int) – absolute x-coordinate of the upper left corner of the tile with respect to the original image

yPosition (int) – absolute y-coordinate of the upper left corner of the tile with respect to the original image

image (byte[][]) – image grayscale pixel data

gridX (int) – column where the tile is found on the original image

gridY (int) – row where the tile is found on the original image

width (int) – width of the tile

height (int) – height of the tile

Public Methods:

Constructors

Two constructors exist for filling the data inside this class, both of which initialize the image data array, coordinates and size. The second construction, however, also allows the initialization of the gridX and gridY values.

C.8 CECAT.images.readers

In the *CECAT.images* package, there is an interface called *ImageReader*. The goal behind this interface is to provide a general API for all image input streams. In addition to traditional "read the whole image into memory" approaches, this interface provides the functionality required to chop the image into tiles (of a customizable size) and read them in, piece-by-piece. Each implementation deals with a different file format or approach as described below.

CECATImageReader (implements *CECAT.images.ImageReader*)

Description:

This ImageReader implements methods for decoding CECAT compressed files (.cec) and performs contour filling operation to present the completely decoded CECAT foreground.

CECATImageReceiver (implements *CECAT.images.ImageReader*)

Description:

Although similar to the *CECATImageReader*, this ImageReader does not use files as its input. Instead, this reader is receives blocks of array data (usually from the prototype CECAT server) and organizes the image for consumption by other classes.

StandardImageReader (implements *CECAT.images.ImageReader*)

Description:

Java contains built-in functionality to decode JPEG, GIF, and PNG file formats. This ImageReader makes use of this functionality to allow tiling of these "common" file formats.

UncompressedImageReader (implements *CECAT.images.ImageReader*)

Description:

Adapted from the JIGL library created at BYU, this is a "bare-bones" decoder for the following "raw" image formats: PGM, PPM, PBM, and PRGM.

UnsupportedImageReader (implements *CECAT.images.ImageReader*)

Description:

This default interface returns empty data sets and is used as a placeholder for file formats currently not implemented.

C.9 CECAT.images.viewers

This package contains the Java Swing components used to create scrollable windows that can be used to view an image. These images are represented fed into the viewer using the ImageReader interface, making these viewers independent of file format.

ColorModels

Description:

This class contains static color models used by Java to render grayscale images. They are stored here to prevent declaring them in multiple places.

FloatingImageViewer

Description:

This viewer, used by CompressionViewer, is a complex grayscale image viewer used to render a grayscale image as a collection of tiles inside a scroll pane. Simply put, this viewer creates a virtual canvas for the image and adds tiles and layers of image data as it receives them. This canvas is displayable at any stage in the process.

Public Methods:

Constructor

Only one constructor is available for this data object, which is used to set the height and width of the viewer window as well as initialize the viewer.

displayVisibleTiles

By calling this method, the viewer is forced to repaint its contents for the user.

setNewImage

This operation sets the viewer to receive a new image from an ImageReader, initializing the virtual canvas size and connecting to the image source.

Inputs:

imageStream (ImageReader) – input stream for the image pixel data

rotateLeft

This rotates the image contained in the viewer 90 degrees counter-clockwise.

rotateRight

This rotates the image contained in the viewer 90 degrees clockwise.

flipHorizontal

This mirrors the image contained in the viewer from left to right.

flipVertical

This mirrors the image contained in the viewer from top to bottom.

invert

This operation essentially creates a 'negative' copy of the currently displayed image (white pixels become black).

changeOrientation

This method can be used to change the 90 degree rotation and/or mirroring of the currently displayed image using internal orientation codes.

Inputs:

orientation (String) – two-character orientation code which describe which corners of the image are found at the top corners of the viewer

requestNextLayer

By calling this method, the viewer starts requesting the next layer of the CECAT encoded image (residual or background). If tiles from the

134

previous layer have not been sent yet, they will be transmitted when needed.

requestNextTile

This method sends a request to the image source for the next available tile.

GrayscaleImageViewer

Description:

This viewer, used by CompressionInterface, is a very simple grayscale mage viewer used to render a grayscale image inside a scroll pane. Essentially a merging of an array of pixel data and a display pane, this viewer can be used to convert grayscale data into a more viewable form.

Public Methods:

Constructor

Only one constructor is available for this data object, which is used to set the height and width of the viewer window as well as initialize the viewer.

setImage

This operation displays a grayscale image in the viewer.

Inputs:

image (byte[][]) – grayscale image data

C.10 CECAT.preprocess

Most image manipulation operations, many of whom are applied to images before the CECAT encoding occurs, are stored in this package. Most of these operations are implemented as static methods for performance reasons.

BitonalThresholding

Description:

The thresholding operation described in Section 3.1 is implemented in this package. Although designed to be a holding place for a variety of binarization algorithms, the modified Niblack operation is the only one currently implemented.

Public Methods:

NiblackThreshold (static)

This method actually performs the binarization operation described in detail in Section 3.1, a modified Niblack thresholding operation with a fixed global threshold.

Inputs:

image (byte[][]) – a grayscale image map (each byte corresponding to a pixel)

ImageProcessing

Description:

Most of the different image processing operations used by the CECAT system are consolidated into this one package. These operations range from the simple "open" operation to the application of image masks to create residual and background layers. Any further "stand-alone" image processing operations should be added to this class.

Public Methods:

MaskImage (static)

This operation applies a mask to the image and "whites out" the grayscale pixel values not covered by it.

Inputs:

image (byte[][]) – original (pre-encoded) image data

mask (byte[][]) – bitonal foreground mask after it has been decoded from its CECAT form

GenerateBackground (static)

The inverse operation of the "MaskImage" method, this operation applies a mask to the image and "whites out" the grayscale pixel values covered by it.

Inputs:

image (byte[][]) – original (pre-encoded) image data

mask (byte[][]) – bitonal foreground mask after it has been decoded from its CECAT form

ApplyMorphOpen (static)

This is a basic image processing "open" operation used on bitonal image to remove small pixel noise.

Inputs:

image (byte[][]) – bitonal image data

C.11 CECAT.segments

This package contains all the code required to implement the Curve Segment Library described in Section 4.3.

CurveSegmentLibrary (interface)

Description:

This interface exposes the main functionality of the curve segment library, namely looking up curve segments, indices, or identifying the size of the curve segment library.

Public Methods:

lookupCurveSegment

This method converts a library index into a curve segment.

Inputs:

index (int) – index for the curve segment desired

Outputs:

Segment object representing the curve segment that has been found using the inputted index.

lookupIndex

This method finds the library index for a particular curve segment.

Inputs:

deltaX (int) – change in the x direction for curve segment

deltaY (int) – change in the y direction for curve segment

Outputs:

Int which provides an index to the library where the curve segment with the inputted deltas is found.

getSize

Used most encoding operations, this method returns the maximum bit size a curve segment can possess and still be indexed in this library. Its primary purpose is to determine if a given curve segment should be represented by an index to this library or not.

Outputs:

<u>Int</u> showing the maximum bit size of curve segments indexed in this instance of the curve segment library.

DecoderSegmentLibrary (implements *CECAT.segments.CurveSegmentLibrary*)

Description:

This is one of two implementations of the CurveSegmentLibrary and is optimized for the process of decoding curve segments given an index. Simply put, this library creates a large array of segment objects, all of which can be retrieved in constant time given a particular index. Finding an index given a curve segment, on the other hand, requires a full search once through the table.

EncoderSegmentLibrary (implements *CECAT.segments.CurveSegmentLibrary*)

Description:

The second implementation of the CurveSegmentLibrary, this is optimized for encoding curve segments by providing a constant time lookup of an index given a curve segment. Going the opposite direction (looking up a curve segment given an index) requires a search through the whole library.

Segment

Description:

Curve segments are, simply put, the x and y coordinate deltas from one control point to another. This data object represents this curve segment.

*Public Variables***:**

deltaX (int) – change in the x direction from one control point to another

deltaY (int) – change in the y direction from one control point to another

Public Methods:

Constructor

The constructor for this object allows you to initialize the deltas when creates an instance of this object.

Bibliography

[1]   Simone Marinai, Emmanuele Marino, Francesca Cesarini, and Giovanni Soda. A General System for the Retrieval of Document Images from Digital Libraries. In *Proceedings of the IEEE First International Workshop of Document Image Analysis for Libraries (DIAL)*, Palo Alto, CA, pages 150-173, January 2004.

[2]   Charles Cullen. Special Collections Libraries in the Digital Age: A Scholarly Perspective. *Impact of Digital Technology on Library Collections and Resource Sharing*. Haworth Information Press, New York, NY, 2001.

[3]   Project Gutenberg. Found at URL http://www.promo.net/pg/, visited on 22 September 2005.

[4]   Carnegie Mellon Million Books Project. Project summary and proposal found at URL http://www.ulib.org, visited on 22 September 2005.

[5]   The Newton Project. Found at URL http://www.newtonproject.ic.ac.uk/index.html, visited on 21 September 2005.

[6]   The Internet Text Archive. Found at URL http://www.archive.org/details/texts, visited on 21 September 2005.

[7]   F. Le Bourgeois, E. Trinh, B. Allier, V. Eglin, H. Emptoz. Document Image Analysis solutions for Digital Libraries. In *Proceedings of the IEEE First International Workshop of Document Image Analysis for Libraries (DIAL)*, Palo Alto, CA, pages 2-24, January 2004.

[8]   Gutenberg Bible, The British Library. Found at URL
        http://www.bl.uk/treasures/gutenberg/homepage.html, visited on 21 September
        2005.

[9]   International Children's Digital Library. University of Maryland. Found at URL
        http://www.icdlbooks.org/, visited on 23 September 2005.

[10]   Mountain West Digital Library, Utah Academic Library Consortium. Found at
        http://www.lib.utah.edu/digital/mwdl/, visited on 4 October 2005.

[11]   The 1901 Census of England and Wales.   Found at URL
        http://www.1901census.nationalarchives.gov.uk/, visited on 23 September 2005.

[12]   Douglas J. Kennard. Just-In-Time Browsing for Digital Images. Thesis Presented
        to BYU: February 2003.

[13]   National Telecommunications and Information Administration. A nation online:
        Entering the Broadband Age. September 2004. Found at URL
        http://www.ntia.doc.gov/ntiahome/dn/, visited on 10 October 2005.

[14]   Ian H. Witten, Alistair Moffatt, Timorthy C. Bell. *Managing Gigabytes*. Van
        Nostrand Reinhold: New York. 1994

[15]   Yan Ye, Pamela Cosman. Dictionary Design for Text Image Compression with
        JBIG2. URL: http://code.ucsd.edu/~yye/TransIP2001.pdf, visited on 6 March
        2004.

[16]   Leon Bottou, Patrick Haffner, Paul G. Howard, Partice Simard, Yoshua Bengio,
        Yann Le Cun. High Quality Document Image Compression with DjVu. *Journal of
        Electronic Imaging*, Vol 7, No 3, pp 410-425, SPIE, 1998.

[17]   MicroSoft SLIm (Segmented Layered Image) project summary found at URL
        http://research.microsoft.com/dpu/, visited 17 October 2005.

[18] Dan Huttenlocher, Angela Moll. On DigiPaper and the Dissemination of Electronic Documents. *D-Lib Magazine*, Vol 6, No 1. Found at URL http://www.dlib.org/dlib/january00/moll/01moll.html, visited 17 October 2005.

[19] Qin Zhang, John M, Danskin. A Pattern-Based Lossy Compression Scheme for Document Image. Dartmouuth College Department of Computer Science. Electronic Publishing-Origination, Dissemination and Design. June 24, 1996. Found at URL: http://citeseer.ist.psu.edu/43442.html

[20] Overview of JBIG2. A Presentation by PlanetDjVu, June 10, 2003. Originially presented by Xerox Parc a couple years previous. Found at http://www.planetdjvu.com/overview_of_jbig2.htm

[21] Paul G. Howard. Text Image Compression Using Soft Pattern Matching. *The Computer Journal*. Vol. 40, No. 2/3, 1997

[22] Dov Dori, Wenyin Liu. Sparse Pixel Vectorization: An Algorithm and Its Performance Evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 21, No. 3. March 1999.

[23] Karl Tombre, Christian Ah-Soon, Philippe Dosch, Gerald Masini, Salvatore Tabbone. Stable and Robust Vectorization: How to Make the Right Choices. Found at URL: http://citeseer.ist.psu.edu/tombre99stable.html.

[24] Tinku Acharya, Ping-Sing Tsai. *JPEG2000 Standard for Image Compression Concepts, Algorithms and VLSI Architectures*. Wiley-Interscience: New Jersey. 2005.

[25] P. Haffner, L. Bottou, Y. LeCun, L. Vincent. A General Segmentation Scheme for DjVu Document Compression. In *Proceedings of the International Symposium on Memory Management (ISMM)*, Berlin, Germany, June 2002.

[26] Carey Bunks. *Grokking the Gimp*. New Riders Publishing, 2000. Found at URL: http://gimp-savvy.com/BOOK/index.html, visited 6 March 2004.

[27] Wayne Niblack. *An Introduction to Digital Image Processing*. Prentice-Hall International, 1985.

[28] Michael D. Smith. Handwriting Compression Using Quadratic Curves. Brigham Young University Computer Science 750 Projct Write-Up. November 29, 2003.

[29] The Mathematics of String Art: A Tribute to Pierre Bezier (1910-1999). The Glossary of Mathematical Mistakes Archive (5/2000). Found at URL: http://members.cox.net/mathmistakes/bezier.htm

[30] Tim Andrew Pastva. Bezier Curve Fitting. Thesis submitted to Naval Postgraduate School, Monterey, California. September 1998.

[31] Carlos F. Borges, Tim Pastva. Total Least Squares Fitting of Bezier and B-spline Curves to Ordered Data. *Computer Aided Geometric Design* 19. 2002.

[32] Lauralea Otis. Project 2 My Handwriting Library. Brigham Young University Computer Science 750 Project Write-Up. November 2003.

[33] GraphicsMagick Image Processing System. Found at URL http://www.graphicsmagick.org, visited on 1 March 2006.

[34] DjVuLibre: Open Source DjVu Library and Viewer. Found at URL http://djvulibre.djvuzone.org, visited on 1 March 2006.

[35] Z. Zhang and C. L. Tan. Restoration of images scanned from thick bound documents". In *Proceedings of International Conference of Image Processing*, Vol. 1, 2001, pages 1074-1077.

[36] Graham Leedham, Chen Yan, Kalyan Takru, Joie Hadi Nata Tan, Li Mian. Comparison of Some Thresholding Algorithms for Text/Background Segmentation in Difficult Document Images. In *Proceedings of 7th International Conference on Document Analysis and Recognition*, 2003.

[37] David Tam, William Barrett, Bryan Morse and Eric Mortensen. Breakpoint Skeletal Representation and Compression of Document Images. In *IEEE Data Compression Conference (DCC '98)*, page 75. Snowbird, Utah, March 1998.