

California State University, San Bernardino

CSUSB ScholarWorks

Theses Digitization Project

John M. Pfau Library

2004

An image delta compression tool: IDelta

Kevin Michael Sullivan

Follow this and additional works at: <https://scholarworks.lib.csusb.edu/etd-project>



Part of the [Software Engineering Commons](#)

Recommended Citation

Sullivan, Kevin Michael, "An image delta compression tool: IDelta" (2004). *Theses Digitization Project*. 2543.

<https://scholarworks.lib.csusb.edu/etd-project/2543>

This Thesis is brought to you for free and open access by the John M. Pfau Library at CSUSB ScholarWorks. It has been accepted for inclusion in Theses Digitization Project by an authorized administrator of CSUSB ScholarWorks. For more information, please contact scholarworks@csusb.edu.

AN IMAGE DELTA COMPRESSION TOOL:

IDELTA

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

by

Kevin Michael Sullivan

June 2004

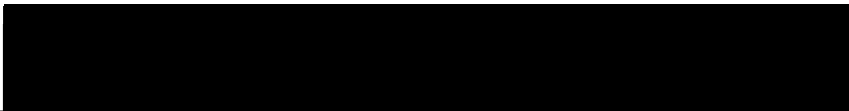
AN IMAGE DELTA COMPRESSION TOOL:

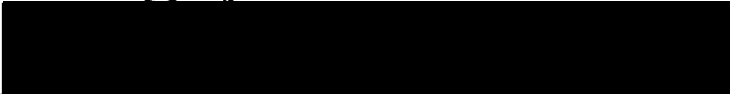
IDELTA

A Thesis
Presented to the
Faculty of
California State University,
San Bernardino

by
Kevin Michael Sullivan
June 2004

Approved by:


Dr. Kay Zernowdeh, Chair, Computer Science


Dr. Richard Botting


Dr. George Georgiou

6/4/04
Date

© 2004 Kevin Michael Sullivan

ABSTRACT

Versioning techniques which save only the changes, or 'deltas', made in a file during an edit are important text editing tools. Saving and managing deltas rather than the entire file for each version iteration uses less disk space and assists in maintaining version integrity. Tools that provide this capability have become a critical part of fields such as software engineering.

The currently available equivalents for binary files perform poorly on multidimensional data such as images, despite significant progress in compression algorithms. When using commercial image editing tools, one must store each edited image as an independent file, making versioning difficult.

The lack of versioning tools for images is a result of two problems. First, creating an efficient representation for the changes for binary data is more difficult than for text data. Seemingly small edits by the user can potentially lead to large changes in much of the data in the byte file. This is especially true for multi-dimensional files such as image. Second, modern image files are quite complex, frequently containing multiple layers with varying degrees of similarity between these layers.

This paper examines the possibility that modern delta compression combined with a detailed understanding of an image formats composition could yield a useful image versioning tool that can work with commercial editors. This paper specifically examines optimizing the current binary delta compression tool zdelta for use with the Photoshop file format (PSD). This optimized PSD format differencing tool is integrated into Photoshop as a file format module to save and open differences directly from the Photoshop interface.

ACKNOWLEDGMENTS

I would like to acknowledge the following people for assistance with my thesis: my family, my co-workers, my professor Dr Zemoudeh, and Dimitre Trendafilov, the author of zdelta.

TABLE OF CONTENTS

ABSTRACT	iii
ACKNOWLEDGMENTS	v
LIST OF TABLES	viii
LIST OF FIGURES	ix
CHAPTER ONE: BACKGROUND	
1.1 Introduction	1
1.2 Purpose of the Thesis	1
1.3 Context of the Problem	2
1.4 Significance of the Thesis	8
1.5 Definition of Terms	8
1.6 Organization of the Thesis	9
CHAPTER TWO: LITERATURE REVIEW	
2.1 Introduction	10
2.2 Tool and Algorithm Review	10
2.3 History of Differencing	11
2.4 Summary	14
CHAPTER THREE: METHODOLOGY	
3.1 Introduction	15
3.2 Discussion of Solutions	15
3.2.1 Solution One	16
3.2.2 Solution Two	17
3.2.3 Solution Three	17
3.2.4 Solution Four	17
3.2.5 Solution Five	18

3.3 Hypothesis Statement	19
3.4 How Differences are Compressed and Decompressed With IDelta	20
3.4.1 Compression	20
3.4.2 Decompression	22
3.4.3 Algorithm Walk Through	22
3.5 Photoshop Implementation	24
3.6 IDelta's Relationship to Zdelta	24
3.7 Summary	26
CHAPTER FOUR: RESULTS	
4.1 Introduction	27
4.2 Comparison Tests	27
4.3 Result Tables	29
4.4 Discussion of Results	40
4.5 Summary	43
CHAPTER FIVE: CONCLUSIONS AND FUTURE WORK	
5.1 Introduction	44
5.2 Conclusions	44
5.3 Future Work	44
5.4 Summary	47
APPENDIX A: EXAMPLE PHOTOS	48
APPENDIX B: ABRIDGED PHOTOSHOP FILE FORMAT WHITE PAPER	50
APPENDIX C: DELTA FILE LAYOUT	57
APPENDIX D: PHOTOSHOP IMPLEMENTATION SCREENSHOTS	60
REFERENCES	64

LIST OF TABLES

Table 1.	Test Results with TestImage.psd	29
Table 2.	Test Results with Buterfly3.psd, #1	30
Table 3.	Test Results with Buterfly3.psd, #2	31
Table 4.	Test Results with Globe.psd	32
Table 5.	Test Results with Cactus.psd, #1	33
Table 6.	Test Results with Cactus.psd, #2	34
Table 7.	Test Results with Cave wave.psd	35
Table 8.	Test Results with Pfeifer.psd	36
Table 9.	Test Results with Lighting.psd	37
Table 10.	Test Results with Gods Eye.psd	38
Table 11.	Aggregate Comparison of IDelta to Zdelta	39

LIST OF FIGURES

Figure 1. Byte Separation Example	5
Figure 2. Color Image Comparison	7
Figure 3. Algorithm Walk Through	23

CHAPTER ONE

BACKGROUND

1.1 Introduction

Versioning systems can utilize 'Differencing' or 'Delta' algorithms to save space and maintain version integrity. These algorithms accomplish lossless compression by finding the data that has not changed between two versions of a file and only save the changed data. The data that has not changed is referenced with markers in the original file. Thus, only the delta is saved rather than re-saving the entire file for each version iteration. Optimally, the algorithm should store only the data that actually changed in the delta file, and this should occur without high costs in time complexity.

1.2 Purpose of the Thesis

This thesis will present a modified version of the algorithm used in the open source differencing tool zdelta, entitled 'iDelta'. This algorithm will manage file data and will be built specifically to difference images in the Photoshop file format. Hence, it will be a 'file type aware' differencing algorithm. The Photoshop file format is currently the most popular and robust 2-d image



file format, therefore it is the best candidate for this thesis.

1.3 Context of the Problem

The analysis of image file formats revealed the following observations. The optimal way to represent an image is via byte combinations, which when read together can represent a set range of colors. 24-bit Red Green Blue (RGB) color can represent; $(1 \text{ byte} * 1 \text{ byte} * 1 \text{ byte}) = (255 * 255 * 255) = 16,581,375$ potential colors. Each pixel is thus represented by 3 bytes, containing a relative color value for a specific color range, commonly referred to as a channel (One byte red, one byte green, one byte blue.) When these are read together, they make up the exact color for any one pixel. Each pixel also has a coordinate (based on the format used) which determines its position on screen. The placement and order of the bytes in a file are subject to the method employed by the format used.

This is a fundamentally different organization than is used by non-binary files. By examining a Microsoft Word or a Word Perfect document, it is evident that even complex word processing text files save data sequentially in a file. Image files are not saved to disk in the same

way they are presented on screen. The data is broken up into different logical sections (vis-à-vis 24 bit RGB byte combinations), so it is not written sequentially in the file. This becomes one of the primary problems for differencing image files. When changes occur, even when these changes are small, these changes can be spread throughout the file.

Images are not text. Although this sounds self evident, the concept of change is different when applied to images. The type of change that images usually undergo should be considered. Images represent a completely different medium of communication. When one changes an image over time, it is truly a different process than when one changes a text document, be it code, art or scholastic. Although it is difficult to make empirical statements to this effect, it is important to consider as the investigation proceeds. In trying to capture change, a typical change for an image involves different actions, and results in a different outcome.

The following example illustrates how the non-sequential storage of bytes in an image file can make binary image data more difficult to difference than text data. If a paragraph is cut or copied into a different location in a text document, the paragraph itself is still

intact; it is just in a different location. If one adds a new paragraph in the middle of already existing text, the previous and following paragraphs are still the same. When committed to disk, the bytes that are written into the file are in the same sequence as in the original file, save the addition of the new paragraph. Alterations that the user affects while changing the file do not disrupt the binary file in a drastic fashion. The bytes following the new paragraph are shifted down by the length of the addition, but the byte order is maintained over all.

The byte order of image files, however, is easily disrupted. Simple changes at the user level may cause large byte reordering in the binary file. For instance, a still life picture of a basket of vegetables was altered by copying a vegetable to the upper right side of the basket (see appendix A for photos.) While this type of change is simple and rather ineffectual at the user level, it could have the repercussions discussed below.

Moving or copying a portion of an image from one location to another does not have a one to one effect as does moving a block of text. A 'one to one' effect here refers to the grouping of bytes and where they reside in the file. Textual data is stored one ASCII character after another. Image data is stored in a PSD file as horizontal

scan lines in the image. Any particular object in an image, such as the red pepper in the example photo, is not written to disk in the grouping of the red pepper, but as horizontal one pixel wide lines of bytes. Thus the bytes that make up the copied red pepper are separated by the bytes that precede them and follow them in the particular scan lines on which they reside. See table 2.1 for a visual representation of this. In the table, the bytes proceeding and following the pepper are represented by 'x's and the byte of the pepper are represented by 'o's. The PSD format then encodes these byte vectors via a

Scanline:	Bytes:
1	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxooxxxxxx
2	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxoooooxxxxx
3	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxoooooxxxxx
4	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxoooooxxxxx
5	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxoooooxxxxx
6	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxoooooxxxxx
7	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxoooooxxxxx
8	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxooxxxxxx

Figure 1. Byte Separation Example

loss-less compression such as Run Length Encoding (RLE) to save space in the disk file [1]. In the example, the new vegetable added changes approximately 10% of the scan lines in the image. This will change the compression ratio that the RLE had achieved in the first iteration of the file. In this case, the ratio was not as efficient as in

the first file. Instead of a flat background, the pepper introduces a new color. All affected scan lines will now take up more space in the flat file. Any bytes that followed that portion of the image (from scan lines 9 to the end of the image) will now be shifted lower in the byte file. The overall file size will increase as well, even though we did not actually increase the dimensions of the file.

It is also apparent from this example that the bytes that comprise the pepper are surrounded by the bytes that precede and follow them on each scan line in the image. This phenomenon is why image data is described as multi-dimensional [2]. If we were to look for this change, we would traverse linearly down the file, scan line 1 to 2 to 3...etc, and continue this way. The changes on scan line 1 from table 2.1 would be separated by a large number of bytes from the changes on scan line 2, etc. So not only did the addition cause a change in the compression ratio, but the changes to the byte file are interwoven between the unchanged bytes that precede and follow them.

To summarize, two example files were altered, a text document and an image file. In the text document, text was moved and added, and the resulting file was slightly larger than its original, and resulted in a small shift in

the byte file. Similar user level changes in the image file analogous to the text file were made, and the file grew by a noticeable amount. Further, the change was intermingled between many other unchanged portions of the image.

The base assumption in creating a delta rather than resaving the entire file is that as versions are created through the life of the file, each incremental change is not drastically different from its original. This assumption can simply be false for image files. Figure 2.1 is an example of a hex view of the image data section of two, very small 24 bit single color image files. These are the sets of byte scan lines of the image; the set on the left represents a blue square, and the set on the right

Bytes: Image One	Bytes: Image Two
E807E807E807E807E807E807E807E807	E8E9E8E9E8E9E8E9E8E9E8E9E8E9E8E9
E807E807E807E807E807E807E807E807	E8E9E8E9E8E9E8E9E8E9E8E9E8E9E8E9
E807E807E807E807E807E807E807E807	E8E9E8E9E8E9E8E9E8E9E8E9E8E9E8E9
E807E8D7E8D7E8D7E8D7E8D7E8D7E8D7	E8E9E813E813E813E813E813E813E813
E8D7E8D7E8D7E8D7E8D7E8D7E8D7E8D7	E813E813E813E813E813E813E813E813
E8D7E8D7E8D7E8D7E8D7E8D7E8D7E8D7	E813E813E813E813E813E813E813E813
E8D7E8D7E802E802E802E802E802E802	E813E813E81DE81DE81DE81DE81DE81D
E802E802E802E802E802E802E802E802	E81DE81DE81DE81DE81DE81DE81DE81D
E802E802E802E802E802E802E802E802	E81DE81DE81DE81DE81DE81DE81DE81D
E802E802E802	E81DE81DE81D

Figure 2. Color Image Comparison

represents a blue-green square. The PSD format writes the data one channel at a time, in the pattern RRR, GGG, BBB,

the bytes are presented here to be read from left to right, top to bottom.

The color was changed to illustrate how the byte file is affected. Note how different the byte set on the right is. When differencing these files, regardless of differencing methods employed, a relatively large delta will be produced. Though the encoding of the ultimate difference can mitigate this problem to a degree, there is clearly significant change, so a large delta is unavoidable. This illustration demonstrates that simple changes can alter the byte layout for an entire section of an image file, and invalidate the versioning delta assumption stated earlier.

1.4 Significance of the Thesis

The current delta algorithms perform poorly when used with image data. The significance of the thesis was to address whether the poor performance of the current differencing tools when used on multi-dimensional data could be noticeably improved.

1.5 Definition of Terms

The following terms are defined as they apply to the thesis. RGB stands for the Red, green, blue color model. This was the primary color model used for test cases run

with iDelta. PSD stands for Photoshop Document, and is the registered file type for this type of document.

1.6 Organization of the Thesis

The thesis was divided into five chapters. Chapter One provides an introduction to the context of the problem, purpose of the thesis, significance of the thesis, and definitions of terms. Chapter Two consists of a review of relevant literature and a history of delta compression. Chapter Three documents the Methodology used in this thesis. Chapter Four presents the results from the thesis. Chapter Five presents the conclusions from the thesis, and makes suggestions for future work. The Appendices for the Thesis follows Chapter Five. Finally, the references for the Thesis are presented.

CHAPTER TWO

LITERATURE REVIEW

2.1 Introduction

Chapter Two presents the current tools that perform the task of delta compression, and the history of delta compression. Currently, most image differencing engines, such as MPEG-2, DIVX, and MPEG-4 are built for motion image formats. As part of the initial literature review, the following sources were exhaustively researched for pertinent documentation on tools that could potentially be used for static images; IEEE transactions, the online ACM Library, The California State University Library, and the Internet.

2.2 Tool and Algorithm Review

The current open source binary differencing tools are the following: diff, vdelta (currently implemented as VCDiff), and zdelta. Diff is currently part of standard Unix distributions, VCDiff is written by Vo Phong at AT&T [3], and zdelta was designed by Dimitre Trendalov of New York Polytechnic. All are implemented on UNIX and have source code available. zdelta was chosen as the foundation for this thesis, based on a number of considerations. The primary reason is that zdelta is the most current

iteration of these tools, and performed better than the other tools in most tests. The other reasons included compact code base, and portability to the WIN 32 platform.

The UNIX tool 'diff' performs well for text files such as source code; however, it performs poorly for binary files [3],[4]. In response to the need to difference such files, a number of methods have been developed. Vdelta, xDelta, and zDelta have all been created to work on a wider range of files [3],[5]. Zdelta is the most recent tool, and showed the best results overall in the tests performed.

All of these tools were investigated and reviewed to determine what the best method to difference images would be. While achieving good compression over all, none of these tools perform well on multi dimensional files, such as image and sound files.

2.3 History of Differencing

Lossless compression revolves around representing portions of a file with like segments of the file itself. By replacing a string of characters that already exists in a file with a pointer back to that string, considerable space can be saved during compression. Ziv and Lempel devised an algorithm known by their initials and the year

it was created (LZ77) [6], which uses this copy based method of compressing individual files. The Unix compression library zlib was based on their algorithm. This style of compression inspired the algorithms that will be discussed here.

Delta compression takes this compression technique one step further by using a reference file to make pointers to, so that not only data that is similar in the file itself can be refereed, but also data in a reference file. This can lead to significant increase in efficiency in the compression ratio. The caveat is that the reference file will always be needed to decompress the delta file.

Early efforts in differencing centered on finding the longest common subsequence (LCS) of matching bytes, and represented them with edit commands in the delta file (Tichy's String to String correction problem [7]). The non-similar bytes are placed in a delta file with reference to where they belong in the target file. To reconstruct, the edits are enacted on the delta to reproduce the original file. This is both costly in time complexity and assumes that the data will be in a similar order in both files. It further lacks the optimization of accounting for repeated substrings in the reference file.

Tichy then improved on the string to string correction method by adding what are called 'Block moves'. Block moves add the optimization of accounting for repeated strings within a file. A block move is defined as a common substring found in the reference and target files, represented by a starting point, ending point, and a length. The block move method insures that unlike LCS, once this common substring is included, it does not need to be re-included in the delta file. If this substring is found again, only a reference to it will be placed in the delta file. With the assumption that all operations have unit cost, the best solution under the block-move approach proved at least as good as that of the longest common subsequence approach [3].

The next optimization made to further compress the difference was to match substrings that are only present in the target file that do not appear in the reference file. This is how compression formats such as zlib gain efficiency and was utilized by zdelta to make the differencing library that iDelta is built on.

The final advance was to take the difference file and encode it using standard encoding techniques. Zdelta utilizes Huffman encoding to compress the chunks of the

already differenced file as it compares the reference file and the target file.

2.4 Summary

A discussion of the literature important to the thesis was presented in Chapter Two. All current differencing tools were reviewed to find a good foundation for an image compression tool for use with sequences of static images. Zdelta was chosen because of its good performance and good portability. The history of differencing was discussed to explain the previous work that has been done in this area.

CHAPTER THREE

METHODOLOGY

3.1 Introduction

The methodology section reviews the solutions explored to achieve a more efficient image delta. The implementation of the solutions is then discussed in detail. iDelta was then implemented in C++ and run from within the Photoshop host application to create deltas from the test images. The following chapter discusses this process.

3.2 Discussion of Solutions

The following are the five methods that were initially evaluated to assist in creating an efficient image delta. As a result of the initial evaluation, three were chosen for implementation, and two were ruled out based on lack of potential or provable merit.

For the analysis process, the PSD format file structure was analyzed, and code was implemented to catalogue the anatomy of a Photoshop file. All section and sub section offsets were set so as to make decisions on the file make up possible. For an abridged description of the PSD file format, please review the attached white paper "Abridged PSD file format" in the appendix B. [8].

3.2.1 Solution One

Difference the file based on the different sections of the PSD file format.

The first aspect of the structure that appeared to be useful for differencing is that the format is laid out in logical sections, some containing basic file information, others containing detailed metadata about certain areas of the file, and others containing the actual pixel data. Given the nature of image files reviewed above, differencing the file based on these logical sections appeared to be the best place to begin. In the processing for this method, each section of reference file and the target file is copied to separate buffers, differenced, encoded, and ultimately stored in a total delta buffer along with the deltas of the other sections. This stands to answer one of the two issues cited previously, namely the byte schisms that develop between iterations of a versioned file. By synchronizing the reference file and target files' metadata and pixel sections, the difference will not be incorrect due to their location in the file. This solution was implemented.

3.2.2 Solution Two

Difference the metadata and pixel data of the layers section.

A layer has two logical portions, the layer metadata, and the channel pixel data. Although the layer meta section is generally not large, it can change enough to shift the bytes which follow it. This was done as a precursor to solution four. This solution was implemented.

3.2.3 Solution Three

Enlarge the byte comparison window of zdelta.

The current implementation of zlib, and consequently zdelta (a modification of zlib) use a 64 kilobyte window for byte comparison iterations. The window size is built into the foundation of the code base, and changing it for this thesis was not possible. This change was explored regarding concerns that matching strings could be offset by more than this amount.

3.2.4 Solution Four

Difference each individual layer's channel data.

This requires some supposition regarding the relationship of reference and target file layers. Layers are not given unique identifiers, so matching them can lead to erroneous comparisons. Layers have names, but these are easily changed. Layers have dimensions, but if

these change, the value of differencing them separately is suspect. Despite the potential for incorrect comparisons, the difference will be no worse than if all of the data is out of alignment. Test indicated that even if one layer was matched, this method produces a smaller delta. This solution was implemented.

3.2.5 Solution Five

Establish the bounding box of individual layer dimension data to extract what part of the underlying image data will be affected. Then break up the image data section via these coordinates and difference the portions separately based on the coordinates of the effecting layers.

Each layer contains the coordinates of its pixel data. The layer sections' purpose is often to add effects to the underlying image data. If changes to an image reveal that only the layer data has changed, and the layer only affects the upper third of an image, the lower two thirds of the image will likely have little or no change. So, if handed separately to the differencing function, it was considered possible that a more efficient delta would be produced by doing so.

Analysis showed that one of two following cases is most common. The first case is when only the layer data changes. The same efficiency will be achieved whether

these sections is separated or not. This is due to that since the layer sections add effects to the underlying image data, the underlying image data is unchanged in the flat file. It is not till the image is opened for viewing that the effects are applied.

The second case is when the image data itself changes, and changes to the image data do not necessarily correspond to the layer data above. Not enough information is available to make an informed decision in this case. This would lead the algorithm to portion the image data when differencing it in mass would have been preferable. Performing the layer arithmetic and differencing the image data based on this could cause disparate portions of the image to be compared, and even lead to larger, rather than a smaller delta. This option was ruled out due to these concerns.

3.3 Hypothesis Statement

Based on this analysis, the following hypothesis was developed. By differencing corresponding portions of an image file, the effects of change that usually disrupts the differencing process can be mitigated, and reasonable sized deltas can be produced in most cases.

3.4 How Differences are Compressed and Decompressed With iDelta

iDelta uses the copy-based method developed by Tichy, as well as the hashing and indexing scheme for string matching, as utilized by VCDiff [3] and zDelta [4]. The difference is then encoded using Huffman encoding to achieve a more compact difference. This is primarily based on the fact that iDelta is built on top of zdelta.

3.4.1 Compression

iDelta begins by setting all the important locations of the reference file and the target file into objects to make file pointer offsets. iDelta then copies each applicable section in each file into separate buffers for comparison, to be compared one after another until the files have been completely compared.

The two file portion buffers are compared in up to 64 kilobyte chunks. The differencing process produces two basic commands; 'copy' or 'insert' (described in detail below). A hash table is constructed; file pointer offsets are created in the reference and target files and indexed in the hash table for reference. Each index in the hash table is searched to find a match. The hash table is keyed with three byte triples from the reference file for comparison purposes.

Processing at each iteration is as follows, matches are searched for, and one of the two actions occurs: If there is no match, an index is marked for the current position of the target file into the hash table, the current position pointer is incremented, and an 'insert' command is produced for output into the delta file.

If there is a match, the algorithm attempts to extend that match as far as possible. Once the current set of matched bytes is exhausted, the current position pointer is incremented by the length of the match, and a 'copy' command is generated for output into the delta file.

After a 64k buffer has been differenced, the resulting delta is then Huffman encoded to achieve a more efficient compression ratio. The Huffman encoding implemented in zdelta is reused for this task.

This is repeated until the each file section has been differenced and all section deltas are computed. Each of the four sections of the PSD file is differenced in this fashion, and the final delta is constructed. It is built in a sequential fashion. Each portion is prefaced with the delta length, followed by its actual uncompressed length. For a detailed description of the delta file make up, see appendix C.

3.4.2 Decompression

As stated above, there are 2 commands that the delta file is built with. They are the 'copy' command and the 'insert' command. Copy commands copy data from the reference to the target file during decompression. Insert commands put the changed data back into the target file in the locations where copying was not possible.

To decompress, the delta file is opened, and the same object code that parses the PSD file parses the delta and extracts all applicable lengths and offsets. The reference file is opened and parsed for lengths and offsets, and the two files are then portioned based on their applicable sections. Each section delta is extracted from the delta file, and it is Huffman decoded. Following that, some data is copied directly from the delta file into the target file when 'insert' commands are found or copied from the reference file when a 'copy' command is found. Copy commands make a marker pointing to a location in the source file, with a starting point and a length, that is how the command knows where to copy the bytes from the how many to copy.

3.4.3 Algorithm Walk Through

The algorithm compares the bytes in the reference and target buffers, comparing the sections for matches as it

proceeds. For portions of the reference and target file that match, the algorithm outputs a copy command. For changed portions it outputs an insert command along with the new data. Thus for files that have been minimally altered, the delta will consist mostly of copy commands, which are simple markers (pointers) into the original file. Figure 3 demonstrates what the algorithm will output given the two example byte strings:

Reference file: x y z x y z x y z x x y z
 Target file: x y z a b c z x y z a b c

Order of operation	command	# bytes to copy or add	iterations +/- Bytes to add
1	copy	3	1
2	insert	3	"abc" x 1
3	copy	1 from index 5	1
4	copy	3 from index 3	1
5	copy	3 from index 3 of target file	1

Figure 3. Algorithm Walk Through

Although this is a simplified example, it elucidates how the commands are generated. Notice that the copy command chooses the 'abc' string from the target file, not the reference file. This behavior enables the algorithm to further compress the delta. This behavior was discussed in the "Tool and Algorithm Review" section in Chapter 2, and is covered in the discussion about the block move method.

3.5 Photoshop Implementation

Photoshop supports a plug-in architecture. Developers can extend the application to perform tasks the original application was not designed for. As part of this investigation, approval was received from Adobe Corp. to utilize the advanced software development kit (SDK), which includes the file format information. The SDK allowed for development of a plug-in to add the differencing code into Photoshop.

iDelta was added into Photoshop via the plug-in architecture as a file format module. This allows for saving and opening the difference files directly into the application. See appendix D for screen shots of the format module being used from within the application.

3.6 iDelta's Relationship to Zdelta

iDelta uses zdelta to perform its work. Zdelta itself is a modification of the Unix zlib compression library. Zlib is based on LZ77, using the copy approach to represent portions of a file with pointers to already compressed portions of the same file. Zdelta extended this functionality to include a reference file, which made delta compression possible. Zdelta modified the zlib compression to enable it to set pointers to positions both

before and after the current file pointer, and added an extra Huffman tree for encoding these new copy pointers. Zdelta then enabled the compression routine to accept multiple reference files, so that more copy commands (references to existing data) rather than insert commands (saving the data in the delta itself) would be used, and hence a more compact delta could be created. More reference files have the potential for greater compression because they increase the chance that existing data will be available for matches.

iDelta utilizes most aspects of zdelta. iDelta is designed to be a version tool only, so only one reference file is used. Despite the fact that a greater compression can be achieved with more reference files, the problem of having more than one reference file can confuse the versioning process and was ruled out for this cause. It was necessary to port the code to be compatible in a Win32 environment, and some alterations were made to compile it with C++ rather than in its native C implementation. After these changes, iDelta reused both the compression stage and the Huffman encoding stage of zdelta. The research and analysis done for iDelta indicated that focusing on aligning the data rather than trying to improve the

already high functioning differencer / encoder, would produce a more efficient algorithm.

3.7 Summary

The methodology section reviewed the solutions explored to achieve a more efficient image delta. The implementation of the solutions was discussed, and the algorithm was presented. The implementation in the host application was presented, and screen shots for this implementation are in the appendix.

CHAPTER FOUR

RESULTS

4.1 Introduction

Included in Chapter Four is a presentation of the results of the thesis. Below is a discussion of the test procedures and results produced from the tests.

4.2 Comparison Tests

For the comparison tests, zdelta was run as a command line application, and iDelta was run directly from Photoshop. Images for tests were chosen based on size and image type, a selection of photographers and art images were used. The image sizes fit into the categories of 2-3 megabyte images, 10-20 megabyte images, 40-60 megabyte images, 100 megabyte and a 250 megabyte image. The metric's tested for were run time and ultimate size of the delta file. Results are presented in graph form with information provided about the compression ratio based on target file compressed, and execution time to perform the difference.

The following changes were made to the images to test and compare the ability of iDelta and zdelta to efficiently represent the changed image file. Note that not all images were subjected to all tests. Layers were

moved to different positions, actual layer data was altered, existing image data was copied from an exiting layer and added as a new layer, color and hue were altered, and the Unsharp Mask effect was applied.

The following tests were not performed, for reasons explained below. Large resolution changes, such as resampling an image from a print resolution of 300 dpi to a screen resolution of 72 dpi, were not tested. This process changes all pixels in the image. Color mode changes, such as switching from RGB to CMYK, were not performed. Changing the color mode requires that a layered image be flattened, so all useful versioning information will be lost if this is done. These tests were considered too invasive to the nature of the file, and will produce deltas near the size of the original file. For practical purposes, these changes would warrant starting new a complete new version.

The specifications for the machine used to test are as follows: Windows XP operating system (Service pack 1), Athlon 1.8 Gigahertz processor, 1 gigabyte of RAM, SCSI raid 5 disk array with more than 5 gigabytes of free space available. Tests were performed on a fresh restart, with all non essential applications turned off. Each test presented here was run three times, and a baseline NULL

case (differencing an unchanged image) was performed to verify accuracy.

Below are tables 1 through 10, and contain all pertinent information regarding the tests performed and the performance of the two algorithms. The tables are laid out with the information about the image being tested first, followed by the results of the specific test being run. Two of the tests were performed on the same reference image; tests 2 and 3, and tests 5 and 6. All other tests were performed on separate images.

4.3 Result Tables

Table 1. Test Results with TestImage.psd.

Document name:	TestImage.psd		
DPI:	333		
Number of Layers:	1		
Description:	The Vegetable basket from Appendix		
Size (Reference Document):	1015 kilobytes		
Changes Made for Test 1	2 layers added, the pepper from example, and the bottom background		
New size (Target Document)	1549 kilobytes		
Results for Test One	Size of Delta (kb)	Time Min:Sec. millisecond	Compression Ratio
Uncompressed	1549	NA	NA
iDelta	95	0:00.219	0.0613
zdelta	212	0:00.640	0.1328

Table 2. Test Results with Butterfly3.psd, #1

Document name	Butterfly3.psd		
DPI	72		
Number of Layers	6		
Description	Art picture		
Size (Reference Document)	1.5 Megabytes		
Changes made for test 2	Layers reordered		
New size (Target Document)	1.5 Megabytes		
Results for Test Two	Size of Delta: (kb)	Time Min:Sec. millisecond	Compression Ratio
uncompressed	1,500	NA	NA
iDelta	7	0:00.312	0.0047
zdelta	1,192	0:00.617	0.7947

Table 3. Test Results with Buterfly3.psd, #2

Document name	Butterfly3.psd		
DPI	72		
Number of Layers	6		
Description	Art picture		
Size (Reference Document)	1.5 Megabytes		
Changes made for test 3	Layer data was changed.		
New size (Target Document)	2.1 Megabytes		
Results for Test Three	Size of Delta: (kb)	Time Min:Sec. millisecond	Compression Ratio
Uncompressed	1,800	NA	NA
iDelta	481	00:01.020	0.2672
zdelta	632	00:01.050	0.3511

Table 4. Test Results with Globe.psd

Document name	Globe.psd		
DPI	233		
Number of Layers	0 (Background image only)		
Description	Art picture		
Size (Reference Document)	6.6 Megabytes		
Changes made for test 4	Piece of Background is cut and copied into a new layer		
New size (Target Document)	13.8 Megabytes		
Results for Test Four	Size of Delta: (kb)	Time Min:Sec. millisecond	Compression Ratio
Uncompressed	13,800	NA	NA
iDelta	4,400	0:06.703	0.3188
zdelta	4,800	0:10.310	0.3478

Table 5. Test Results with Cactus.psd, #1

Document name	Cactus.psd		
DPI	233		
Number of Layers	3 Layers; 1 pixel layer 2 adjustment layers		
Description	Photograph		
Size (Reference Document)	20.4 Megabytes		
Changes made for test 5	Unsharp mask is applied to pixel layer		
New size (Target Document)	21.6 Megabytes		
Results for Test Five	Size of Delta: (kb)	Time Min:Sec. millisecond	Compression Ratio
Uncompressed	20,250	NA	NA
iDelta	19,100	0:20.734	0.9432
zdelta	19,500	0:28.099	0.9630

Table 7. Test Results with Cave wave.psd

Document name	Cave wave.psd		
DPI	550		
Number of Layers	4 layers; 1 pixel layer, 3 adjustment layers		
Description	Photograph		
Size (Reference Document)	34.4 Megabytes		
Changes made for test 7	2" x 2" portion of rack face on the pixel layer is removed and made its own layer. Other layers are reordered.		
New size (Target Document)	34.5 Megabytes		
Results for Test Seven	Size of Delta: (kb)	Time Min:Sec. millisecond	Compression Ratio
Uncompressed	34,600	NA	
iDelta	2,100	0:09.330	0.0607
zdelta	16,500	0:50.688	0.4769

Table 8. Test Results with Pfeifer.psd

Document name	Pfeifer.psd		
DPI	325		
Number of Layers	1		
Description	Photograph		
Size (Reference Document)	41 Megabytes		
Changes made for test 8	Duplicate entire base image, add 2 effect layers, Hue and saturation.		
New size (Target Document)	61.7 Megabytes		
Results for Test Eight	Size of Delta: (kb)	Time Min:Sec: millisecond	Compression Ratio
Uncompressed	61,600	NA	
iDelta	31,300	0:51.740	0.5081
zdelta	47,800	1:51.030	0.7760

Table 9. Test Results with Lighting.psd

Document name	Lighting.psd		
DPI	325		
Number of Layers	15		
Description	Art Image		
Size (Reference Document)	105.6 Megabytes		
Changes made for test 9	Removed 3 pixel layers, unhide 2 previously hidden layers, reordered 4 layers to interact with newly unhidden layers		
New size (Target Document)	79 Megabytes		
Results for Test Nine	Size of Delta: (kb)	Time Min:Sec: millisecond	Compression Ratio
Uncompressed	80,000	NA	
iDelta	2,300	00:32.090	0.0288
zdelta	51,900	04:85.670	0.6488

Table 10. Test Results with Gods Eye.psd

Document name	Gods Eye.psd		
DPI	300		
Number of Layers	6 pixel layers, 2 adjustment		
Description	Art Image		
Size (Reference Document)	243 Megabytes		
Changes made for test 10	Layers are reorganized		
New size (Target Document)	243 Megabytes		
Results for Test Ten	Size of Delta: (kb)	Time Min:Sec: millisecond	Compression Ratio
Uncompressed	244,000	NA	
iDelta	5,300	11:13.010	0.0217
zdelta	200,800	33:15.970	0.8230

Table 11 below displays the results along with the aggregate ratios comparing how iDelta compared with zdelta. The results show that iDelta was able to outperform zdelta with a 2500% better compression ratio, based on an average of all tests performed. Even without the result of test 2, which could be considered an outlier, iDelta showed a 900% better compression ratio over all.

Table 11. Aggregate Comparison of IDelta to Zdelta

Name	Size (kb)	Time (Secs)	Compression Ratio	Time Ratio	Size ratio
Uncompressed	1596	NA	NA		
iDelta	92	0.219	0.0613		
zdelta	212	0.640	0.1328		
zd/id				2.922	2.700
uncompressed	1,500				
iDelta	7	0.312	0.0047		
zdelta	1,192	0.617	0.7947		
zd/id				1.977	170.286
Uncompressed	1,800	NA			
iDelta	481	1.02	0.2672		
zdelta	632	1.05	0.3511		
zd/id				1.029	1.314
Uncompressed	13,800	NA			
iDelta	4,400	6.703	0.3188		
zdelta	4,800	10.31	0.3478		
zd/id				1.538	1.091
Uncompressed	20,250	NA			
iDelta	19,100	20.734	0.9432		
zdelta	19,500	28.099	0.9630		
zd/id				1.355	1.021
Uncompressed	20,500	NA			
iDelta	1,900	5.609	0.0927		
zdelta	10,500	21.834	0.5122		
zd/id				3.893	5.526
Uncompressed	34,600	NA			
iDelta	2,100	9.33	0.0607		
zdelta	16,500	50.688	0.4769		
zd/id				5.433	7.857
Uncompressed	61,600	NA			
iDelta	31,300	51.74	0.5081		
zdelta	47,800	107.37	0.7760		
zd/id				2.075	1.527
Uncompressed	80,000	NA			
iDelta	2,300	23.9	0.0288		
zdelta	51,900	280.06	0.6488		
zd/id				9.0625	22.565
Uncompressed	244,000	NA			
iDelta	5,300	578.3	0.0217		
zdelta	200,800	1995	0.8230		
zd/id				11.718	37.887
Totals average				3.595	25.657

iDelta out performed zdelta's execution time by a factor of 3. This was an unexpected performance enhancement, as no specific speed optimization was added to iDelta. It is important to mention that the algorithm runs more quickly when there is more similar data between files rather than difference. In cases when iDelta is very successful in aligning the data, hence presenting more similarly rather than dissimilarity to the differencer, iDelta will run faster based on this fact alone. However, test 5 shows that even when similar sized deltas are produced, iDelta performed faster by a factor of 2.

4.4 Discussion of Results

Based on the results, iDelta performed at least as well as zdelta in all tests. In some trials, iDelta performed significantly better. iDelta consistently out performed zdelta in execution time. The following is detailed analysis of the results.

Test one used a very simple image, and was used as a baseline comparison to ensure that iDelta's basic functionally was intact.

The result of test 2, in which the only change was the reordering of layers, iDelta produced a delta with a compression ratio of .004. This was a better than expected

ratio, although the change made only shifts pixels, and causes minimal byte reordering.

Test 3 shows the effect of actually manipulating the data on one of the layers. In this test, pixels on the third of five layers were rotated 90 degrees. This has the effect of re-ordering the pixels on that layer, and the resulting compression ratio is considerably lower.

Test 4 removes a large portion of the background layer, and places it on a new layer. This results in some byte reordering, as well as copying pixels to the new layer. In this case, zdelta performs nearly as well as iDelta, creating a delta on 4k larger.

In test 5, both iDelta and zdelta performed poorly with .9 compression ratios. This is the case in which an effect filter (Unsharp mask) was run. Sharpening an image actually affects all pixels, running arithmetic algorithms based on the values of the existing pixels to achieve the effect. This will physically change the values of the pixels present, so there is 100% change. As discussed in the section entitled "The problem", and illustrated by figure 2.1, there are cases in which there is enough change to result in a very large delta.

Test 6 runs the same filter but on a new layer in which only a portion of the image is copied to. This is a

more common use case, and a much better compression ratio is achieved (.09.)

Test 7 shows iDelta achieving a .06 compression ratio, while zdelta achieves a .4 ratio. In cases where data was moved from one layer to another, and when layers are shuffled, iDelta performed significantly better.

Test 8 explored the case in which large amounts of data are added to the new file; in this case the new file is 50% larger than it's original. IDelta managed to achieve a .5 compression ratio when compressing the target file, while zdelta achieved a .7 compression ratio. Although .5 could be considered an average performance, it is actually quite good. The original file size grew by 50%, so 20 of the 30 megs in delta, about 66% of its size, can be explained by that alone.

Test 9 demonstrates the case in which data is eliminated, not added to the image. Although this decreases the file size, significant reordering occurs. iDelta was able to create a significantly better compression ratio than zdelta in this case.

The last test uses a large image of approximately 250 megabytes in size. A simple test is performed, the layers are reordered. IDelta should perform better than zdelta on this test, as the change made is similar to test 2 where

iDelta achieved an excellent compression ratio. This speculation was correct, as iDelta does create a significantly smaller delta than zdelta. However, iDelta takes approximately 10 minutes to perform the difference. This could be considered too long to be functionally useful.

4.5 Summary

Chapter Four covered the results of the performance tests run between iDelta and zdelta. Result tables were presented, and iDelta's over all performance was significantly better than that of zdelta.

CHAPTER FIVE

CONCLUSIONS AND FUTURE WORK

5.1 Introduction

The following chapter discusses conclusions that can be drawn from the performance of iDelta. Ideas for future work are then suggested.

5.2 Conclusions

The primary conclusion that can be drawn from this study is that data alignment can significantly assist in the differencing process. Although image files can experience large byte shifts in the process of editing, the results indicate that this issue can be mitigated with mixed success. By using information about the file type to keep sections with a higher probability of similarity matched when differencing, reasonably sized deltas can be produced. In the tests run with iDelta, 60% of the images had compression ratios greater than 80%. Despite this success, one test yielded a ratio of .9 (10% compression).

5.3 Future Work

The worst ratio achieved by iDelta was for test 5, which involved using pixel based effects. Pixel based effects change all pixel values to achieve the particular

result they were designed to create. It is unrealistic to expect any type of lossless compression, be it delta compression or standard compression, to achieve a good ratio in these cases. As a post test, the Cactus.psd file from test 5 was compressed using zip compression, and the resulting file was 4k larger than the file produced by iDelta. It would appear that once this type of change is made to a file, lossless compression is not going to yield desirable results.

One idea to avoid this would be to create a logging process, similar to transaction logging in SQL databases. Instead of saving an image that had been affected by an invasive filter, only the log event of the application of the filter could be saved. Saving only the log would not take very much space, as only the command would be written to the delta. This would require support from the host application, and could add considerable time to opening the document.

Lossless compression depends on referencing data that already exists in the file to be compressed. If the data is organized in such a way that string matching is prevented, any type of lossless compression is likely to yield a poor ratio. iDelta had success in reorganizing

data and differencing file sections that had a higher probability of being similar.

Another potential method could be to remove the dimensional aspect of the file in order to set the bytes in a more logical sequence. The example of the vegetable basket used in chapter one's section "The context of the Problem" discusses how the RLE compression ratio can change after the image has been edited. The PSD file format divides the main pixel section (Section 5 of the file format) as 2 byte long length fields for each of the channels represented, followed by each channel's scan line pixel data. If this was reordered so that each scan line length was paired with all the channels' bytes for the current scan line, then the shift in data showed in the vegetable basket example would not be problematic.

The PSD file's data section has the all of the scan line lengths grouped together. Consider a 50 x 50 pixel image; the first scan line length will be the RLE compressed length for the first Red channel scan line (The channels are stored in planar order; Red, Green, Blue.) If the red, green and blue channels scan line data was moved directly next to their respective scan line lengths, then the section would be organized by scan line, not sectionalized as lengths and planar pixel data.

This organization would be similar to the way text is organized, where the byte file is can be read left to right, top to bottom. In the current example, if only scan lines 9, 10, 11 and 12 had changed, this reorganization could lead to the creation of a smaller difference.

The downside of this method would be the time it takes to reorganize the file, and the time taken to reassemble on decompression. If the data section had undergone major changes, as seen in test 5 of the results, then this method would do little to improve the results.

5.4 Summary

This chapter discussed conclusions of the results achieved by iDelta. The primary conclusion was that by focusing on data alignment, better delta compression ratios are achievable. Recommendations for future work on the topic were then presented.

APPENDIX A
EXAMPLE PHOTOS

Example photo before change:



Example photo after change: (Red Pepper was copied to top right of the vegetable basket.)



APPENDIX B

ABRIDGED PHOTOSHOP FILE FORMAT WHITE PAPER

Photoshop file format abridged

This is a summary of the Photoshop file formats document used as reference 7. It was written to give the readers of this document an over of the file format, without need to review the manual.

The Photoshop file format is the 8BPS format on the Macintosh platform, and as the PSD format on MS windows.

Photoshop file types	
OS	Filetype/extension
Mac OS	8BPS
Windows	.PSD

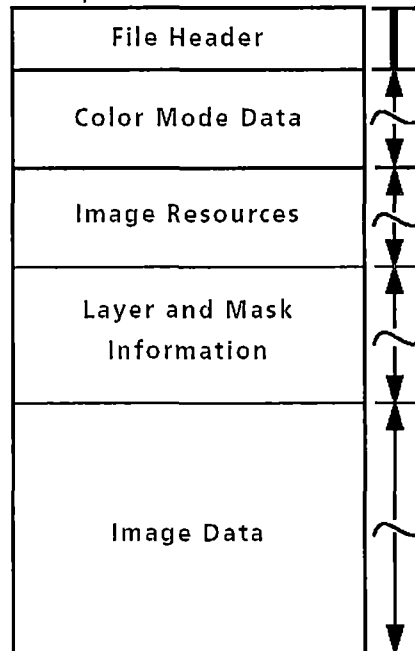
Important notes:

Photoshop is a Mac native app. It uses big endian byte ordering, and for simplicity's sake, it maintains this in other OS's. If working in a windows OS, you must byte-swap when reading or writing files.

For cross platform compatibility, all needed data is stored in the data fork of the file. For Mac's, some data is then duplicated in the recourse fork.

The format is made up of 5 distinct sections, please see Figure 1.1 below:

FIGURE 1.1 Photoshop file structure



I. **File header** - is as it sounds, this is the section in which information about the file is stored. It is a fixed length of 26 bytes, and is the only fixed length section of the format.

II. **Color mode data** - Only indexed color and duotone have color mode data, the length of which is stored in the first four bytes of the section. For all other modes, this section is just the 4-byte length field, which is set to zero. If the image is of either of these sources, the color mode specific data goes here.

III. Image Resources Section -

The image resource section derives its name from Macintosh's concept of splitting files, in which metadata was stored in a 'resource fork', and the file data was stored in the 'data fork'. This motif is mirrored here in which 'info about' the image, commonly referred to as metadata, is stored here.

Format information. It begins with a 4 byte length field.

Some obsolete data fields are stored here at specific locations, left in for backwards compatibility. The following info is stored here (truncated for Brevity)

- Resolution
- Display units (pixels, inches, etc)
- Caption
- Border info
- Default Background color
- Print flags
- Duotone / Grayscale data
- Masking info
- Path info
- Jpeg quality setting
- Copyright flag
- ICC profile
- Watermark
- Slice data
- Print scale

The more complex of these are expanded upon in the text.

By seeking the end of the section, you arrive at the beginning of the Layer section.

VI. **Layer and Mask Information Section** -

The layer section is really where Photoshop breaks away from other image editors and becomes the premiere image editing tool. By supporting layers, which are akin to having acetate layers each appearing one on top of the other, Photoshop allows you to add effects or montage images very easily and intuitively.

Layers can either affect the pixels below its logical placement in the file, or can it be actual pixel image data which resides on top of the background image data. By adjusting the transparency, you can achieve both image and affect if you need.

Format Information: As the other sections, it begins with a length field, and is then following by the layers and masks sub sections respectively. The layers specific section is comprised of a 4 byte fixed length field beginning, giving the length of the layers section. The next portion is variable in length and gives the layer info; the next is variable in length and gives the mask info. It is important to explain that although the flat image data is stored in the last section (see section 5) the layer channel data is stored in this section. By

following the lengths at each layer, you arrive at the channel data for all the layers. It is stored in planar fashion just as the data is, and is compressed with RLE (Run length encoding) as well. The layer section gives the bounding box that contains any particular layer data.

The second of the 2 sections is the mask section, which manages many minor effects fields. They are small, but there a lot of them. (In the doc, the file format goes from page 7 to 47. Pages 24 to 46 comprise the layer-mask info.) By seeking the end of the layers/masks section, you arrive at the data section.

V. Image data section -

This is where the actual image data resides. This is what is actually drawn on screen, save any affects that may be applied to it via the layer section.

Format Information: The image data section begins with a 2 byte compression method indicator.

0 = Raw

1 = RLE compression

2 = ZIP w/o predication

3 = ZIP with predication

Following this is the flat image data (by flat, it is what you actually see, it does not store hidden data) and is

stored in planar fashion, Red data first, Green data second, and Blue data third. It uses RLE encoding for compression, and is stored as the scan line coordinates, then the color data.

APPENDIX C
DELTA FILE LAYOUT

```

30 bytes - Target file header and color mode length
4 bytes  - The reference file length
Variable - The reference file name

4 bytes  - Image resource delta section length
4 bytes  - this section's uncompressed length
variable - this section's delta buffer

4 bytes  - layer metadata delta section length
4 bytes  - this section's uncompressed length
variable - this section's delta buffer

If(simple diff case)
{
    4 bytes  - channel section delta section length
    4 bytes  - this section's uncompressed length
    Variable - this section's delta buffer
}
else // complex diff, i.e. reordered layers
{
    while(layers)
    {
        4 bytes - channel sec delta section length
        4 bytes - ref layer
        4 bytes - target layer
        4 bytes - this section's true length
        Variable - section's channel delta buffer
    }
}

4 bytes  - image section delta section length
4 bytes  - this section's uncompressed length
variable - this section's delta buffer

```

The target file header is a fixed 26 bytes, so it is not worth the over head to difference it alone. Adding it to another section is an option, however, it provides a good header for iDelta, since it retains simple macro data about the target file that it is a difference of. It is simply copied into the main delta buffer for this reason.

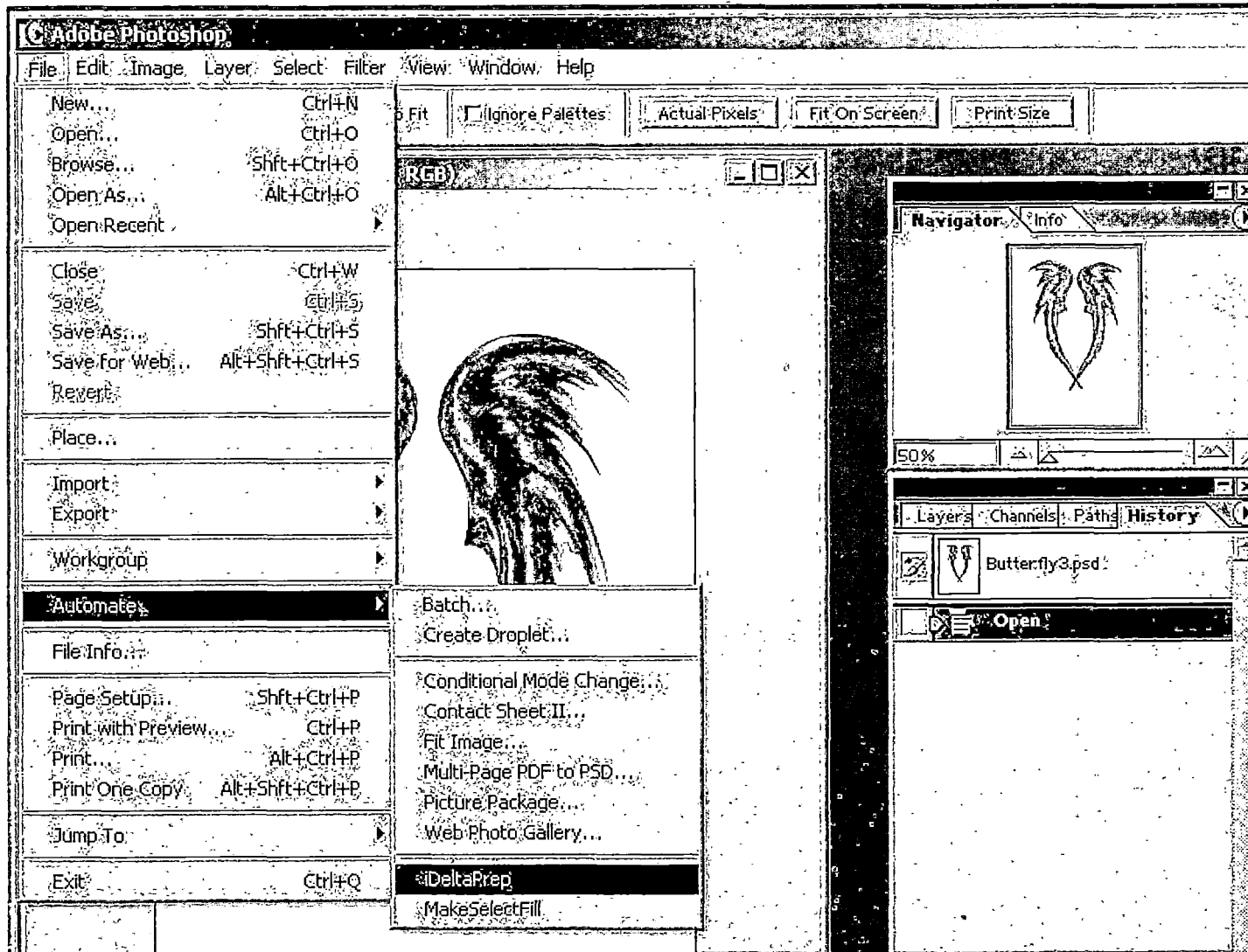
The color mode data section is '0' except for unusual cases, which iDelta is currently not supporting. It will be trivial to add later if the need arises. This is followed by the length of the reference file name, and the reference file name itself. The Next field is the length of the image resource delta section length, referring to the actual differenced data buffer for this section, followed by the true length of uncompressed data. This length is useful for both memory allocation for decompression, as well as a data check to confirm the decompression was successful. This is followed by the actual difference buffer which contains the copy and insert commands and bytes to insert for the extraction process. This pattern follows true for the layer, channel, and image sections of the file.

The only aberration of this is when layers have been reorder. In this case, iDelta does a best guess match of the layers, then sections out the channel data to difference against the most likely similar channel data. This process adds extra length fields, as more data is needed for extraction.

iDelta adds no footer, as all need info is embedded in the contents of the delta file itself.

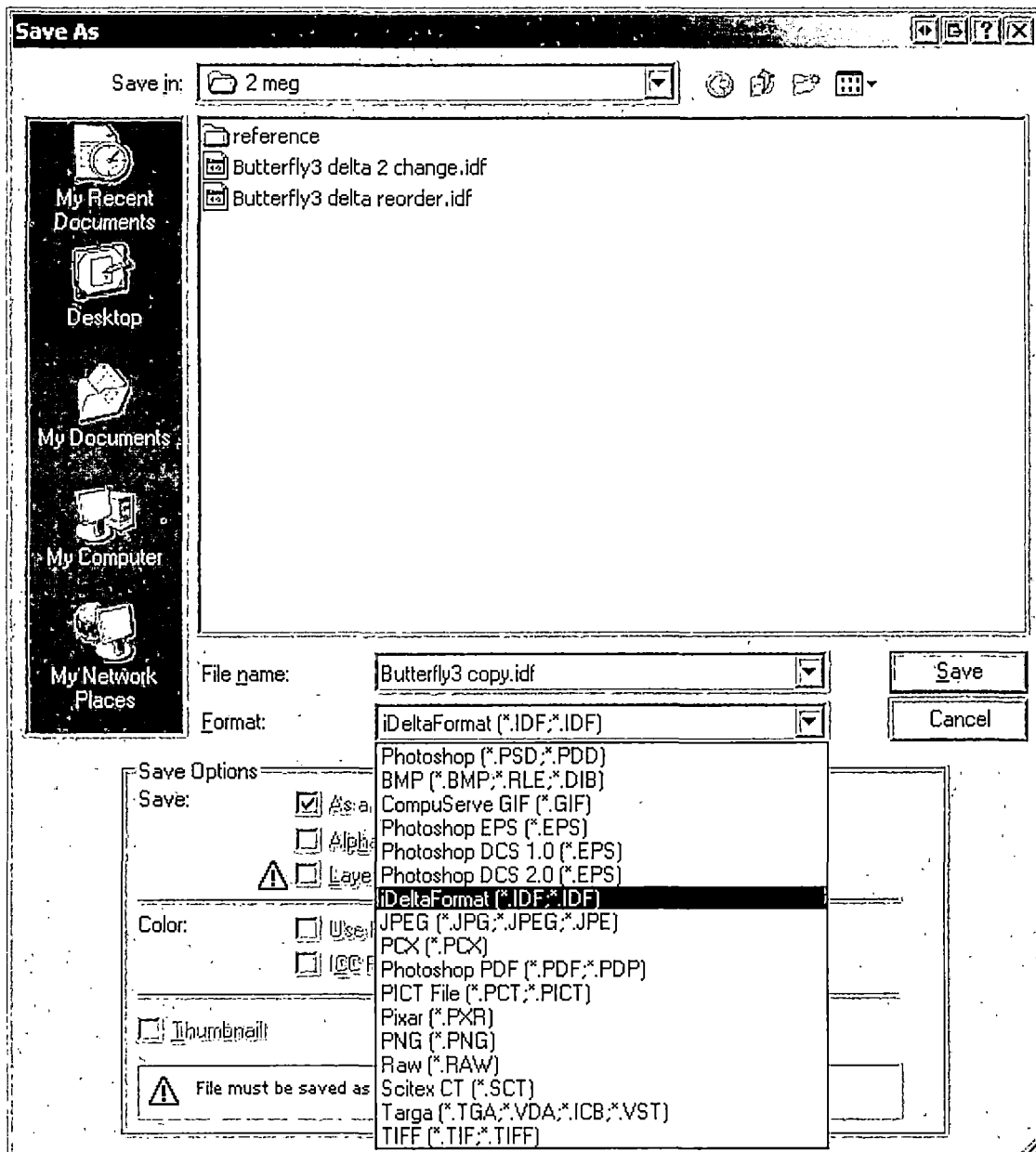
APPENDIX D

PHOTOSHOP IMPLEMENTATION SCREENSHOTS



iDelta Preparation Routine:

The iDelta save function:



The iDelta Open Dialogue:

