

COMPRESSION OF CARTOON IMAGES

by

TY TAYLOR

Submitted in partial fulfillment of the requirements

For the degree of Master of Science

Thesis Advisor: Dr. Marc Buchner

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

May, 2011

CASE WESTERN RESERVE UNIVERSITY
SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis/dissertation of

Ty Taylor

candidate for the Masters of Science in Computer Science degree *.

(signed) Marc Buchner

(chair of the committee)

M. Cenk Cavusoglu

Vincenzo Liberatore

(date) 3/22/2011

*We also certify that written approval has been obtained for any
proprietary material contained therein.

Table of Contents

Overview.....	- 1 -
Problem Statement.....	- 2 -
Motivation.....	- 3 -
Background.....	- 4 -
Bresenham's Line Algorithm.....	- 4 -
Run-Length Encoding.....	- 5 -
Fibonacci Encoding	- 7 -
Huffman Encoding.....	- 10 -
Image Compression in General.....	- 14 -
The Portable Network Graphic File Standard.....	- 15 -
The JPEG File Standard.....	- 16 -
Quad Trees and Application to Cartoon Image Compression	- 17 -
Cartoon Image Region Detection	- 19 -
Fundamental Idea.....	- 19 -
Implementation	- 19 -
Color Reduction.....	- 22 -
Problem.....	- 22 -
Modification of Region Detection Algorithm	- 23 -
Region Concatenation.....	- 26 -
Gradient Detection and Application	- 30 -
Gradient Application.....	- 31 -
Gradient Detection.....	- 38 -
Data Compression.....	- 47 -
Idea.....	- 47 -
Implementation	- 48 -
File Structure.....	- 51 -
Image Reconstruction via Low-Pass Filtering.....	- 53 -
Idea.....	- 54 -
Detecting the Edges between Regions.....	- 55 -
Low-Pass Filtering.....	- 56 -
Results.....	- 60 -
Conclusion and Future Work.....	- 63 -
Bibliography	- 68 -

List of Tables

TABLE 1. THE COMPLETE FILE STRUCTURE.	- 51 -
TABLE 2. RESULTS FOR TEST IMAGE 1.	- 60 -
TABLE 3. RESULTS FOR TEST IMAGE 2.	- 61 -
TABLE 4. RESULTS FOR TEST IMAGE 3.	- 61 -
TABLE 5. RESULTS FOR TEST IMAGE 4.	- 62 -
TABLE 6. RESULTS FOR TEST IMAGE 5.	- 62 -
TABLE 7. RESULTS FOR TEST IMAGE 6.	- 63 -

List of Figures

FIGURE 1. THE “GET LINE” FUNCTION, IMPLEMENTING BRESENHAM’S ALGORITHM.....	- 5 -
FIGURE 2. THE “FIBONACCI ENCODE” ALGORITHM.....	- 8 -
FIGURE 3. THE “FIBONACCI DECODE” ALGORITHM.	- 8 -
FIGURE 4. THE “NODE” STRUCTURE, REPRESENTING A NODE IN A HUFFMAN TREE.....	- 10 -
FIGURE 5. THE “APPLY HUFFMAN ENCODING” ALGORITHM.	- 11 -
FIGURE 6. AN ILLUSTRATED EXAMPLE OF A HUFFMAN TREE GENERATION.....	- 12 -
FIGURE 7. AN ILLUSTRATED EXAMPLE OF A QUAD-TREE DECOMPOSITION.	- 17 -
FIGURE 8. THE “FILL ALL REGIONS” ALGORITHM.	- 20 -
FIGURE 9. THE “CREATE NEW REGION” ALGORITHM.	- 20 -
FIGURE 10. THE “COLOR MATCH” ALGORITHM.	- 20 -
FIGURE 11. THE RESULT OF NOISE AND ANIT-ALIASING.	- 23 -
FIGURE 12. THE MODIFIED “COLOR MATCH” ALGORITHM.....	- 24 -
FIGURE 13. THE RESULT OF APPLIED COLOR REDUCTION.....	- 26 -
FIGURE 14. THE “CONCATENATE REGIONS” ALGORITHM.	- 28 -
FIGURE 15. THE “GET BEST PIXEL MATCH” ALGORITHM.	- 28 -
FIGURE 16. THE “DIFFERENCE METRIC” ALGORITHM.	- 29 -
FIGURE 17. A COMPARISON OF GRADIENT SHAPES.....	- 31 -
FIGURE 18. THE “APPLY GRADIENT” ALGORITHM.	- 32 -
FIGURE 19. THE “APPLY LINE GRADIENT” ALGORITHM.	- 33 -
FIGURE 20. THE “APPLY CIRCLE GRADIENT” ALGORITHM.	- 35 -
FIGURE 21. THE “INTERPOLATE COLOR” ALGORITHM.	- 36 -
FIGURE 22. THE “SNAP COLORS” ALGORITHM.....	- 37 -
FIGURE 23. THE “GET GRADIENT” ALGORITHM.....	- 39 -
FIGURE 24. THE “GET SOLID COLOR” ALGORITHM.....	- 40 -
FIGURE 25. THE “GET MIDDLE POINT” ALGORITHM.....	- 41 -
FIGURE 26. THE “GET LAST POINT” ALGORITHM.....	- 42 -
FIGURE 27. THE “GET NORMALIZED DARKNESS” ALGORITHM.	- 44 -
FIGURE 28. THE “COLOR IN RANGE” ALGORITHM.....	- 45 -
FIGURE 29. THE “GET GRADIENT SHAPE” ALGORITHM.....	- 46 -
FIGURE 30. THE “GET RLE STREAM” ALGORITHM.	- 49 -
FIGURE 31. THE “FILL REGION” ALGORITHM.....	- 50 -
FIGURE 32. THE LOSS OF ANTI-ALIASING AFTER COLOR REDUCTION IS APPLIED.	- 54 -
FIGURE 33. THE RESULTS OF EDGE-BASED OUTLINE GENERATIONS.....	- 55 -
FIGURE 34. THE WEIGHT DISTRIBUTIONS FOR THE BLUR FILTER.....	- 57 -
FIGURE 35. FILTERING RESULTS WITH VARIOUS APPROACHES.	- 59 -
FIGURE 36. POSTERIZATION IN GRADIENT REGIONS.	- 65 -

Compression of Cartoon Images

Abstract

by

TY TAYLOR

This thesis describes a new technique for compressing cartoon images by taking advantage of the distinct color regions in an image and applying a spatially-based compression algorithm. The method for storing these regions involves a combination of Binary Run-Length Encoding and Huffman Encoding. The compression of cartoon images presented here is a lossy compression scheme that removes artifacts and anti-aliasing before encoding the image, and upon decoding the image, uses an edge-restricted blur filter in an attempt to smooth the edges of the decoded image. Algorithms to support gradient detection, their application, and storage are also described. With these algorithms and the proposed file type, on average the test images were 13.75 times more compact than the corresponding PNG file and 7.45 times more compact than the corresponding JPEG file, with a best bit per pixel ratio of 0.00987 bpp.

Overview

This thesis provides the details and implementation of an original algorithm to compress cartoon-style image based on preexisting techniques. The Problem Statement section details the key concept that makes this possible, and the corresponding Motivation section details the rational for developing this procedure.

A Background section is provided, which gives a high-level overview of the preexisting technical topics, such as Huffman Encoding, Run-Length Encoding, and the PNG and JPEG file specifications, for readers who are unfamiliar with the concepts. Thereafter, technical details and algorithms are presented for implementing the compression and corresponding image processing operations. The Cartoon Image Region Detection section describes the procedures used to identify and store the regions of unique color in a cartoon image, and along with the Color Reduction section, describes procedures to reduce the number of these regions for a higher compression ratio. In addition to solid-color cartoon images, those with gradients are supported, and the methods to detect and apply gradients are presented in the Gradient Detection and Application section. Finally, the data compression techniques and proposed file format are given in the Data Compression section, and the image reconstruction algorithms used by the decoder are given in the Image Reconstruction via Low-Pass Filtering section.

After the technical details of the implementation are presented, the results of the proposed image compression algorithm are compared with preexisting compression standards in the Results section. Thereafter, discussions of the results and proposed

improvements, as well as areas in which this research can be extended, are given in the Conclusion and Future Work section.

Problem Statement

Image compression has been an active area of research in Computer Science for many years, and although advancements have been made to create image files that are quite small in size, modern methods of image compression attempt to deal with general images. A general image could be an image of anything, in any format, with any range of colors, and with any pattern of colors. However, by creating encoding algorithms and techniques to deal with general images, they will not be optimal as compared to algorithms that exploit predictable patterns to reduce the size of the image beyond which the general approaches are capable of producing.

The specific pattern that is examined by the research presented in this thesis is that of cartoon-style images. Cartoon images, as opposed to other types of images such as a photograph, have the unique property that colors are grouped into visually distinct regions, where a human or computer can easily identify the outlines of these regions by looking at color differences—which are easily noticeable. This is different from images such as photographs, in which colors rapidly disperse across the image, making it difficult to detect significant regions of a constant color. This research explores a method for encoding cartoon images in a manner that does better than the best general image compression algorithms.

Motivation

Although computer storage systems are growing in capacity at a rapid rate, the need for new and more efficient data compression is still a necessary research topic due to the bandwidth limitations of the internet, which is not increasing nearly as quickly as storage device capacities. A massive amount of data is sent and received via the internet every minute, and therefore, any data that is compressed before sending will reduce the amount of data sent over the internet and therefore make download speeds faster. Cartoon images are a widely used type of image for webpages, online Flash games, online cartoon video, and more. The most common file standard in which to save cartoon images is the Portable Network Graphic (PNG) standard, but by creating a file format and encoding algorithm that reduce the number of bits used to save a cartoon image, the total upload bandwidth used by a server that is hosting these images can be reduced significantly.

An additional motivation for studying the compression of cartoon images is that cartoon compression can be extended to general image compression. Many standard image compression formats, used for compressing general images such as photographs, use a Discrete Cosine Transform or a Fourier Transform to store the Pixel color data in an efficient manner. However, these transforms have low variability, and thus low storage requirements, in regions of the image with small changes in color, and these transforms have high variability, and thus a larger storage requirement, over the borders of these regions. The idea is to apply the techniques presented in this research to create a cartoon-like representation of the general image, and then only apply the DCT or FT on each of those regions separately, where the variability would always be low, and thus represent

the general image with several DCTs or FTs of low variability, resulting in a more compactly encoded image. Similar ideas have been presented in [8], [11], [14], and [13].

Background

This section describes various background research and algorithms related to the research presented in this thesis. Bresenham's Line Algorithm is described because it is used to detect and apply linear gradients. The Run-Length Encoding and Huffman Encoding Algorithms are detailed because these are used for storing the image data, and Fibonacci Encoding is described because it is used to store the Huffman Tree. Additionally, the basic algorithm of the PNG, JPEG, and Cartoon-Specific Quad-Tree file specifications are given, as these are used for comparison in the Results section.

Bresenham's Line Algorithm

Bresenham's Line Algorithm, first presented in [1], is used to efficiently obtain all integer Points (x-y pairs) that are on the line between two input Points. This is one of the most efficient algorithms to do this, as it deals only with integers (i.e., does not require floating point arithmetic) and no divisions. In addition, this algorithm will return the minimum number of Points required to create a gapless line between the two input Points. The algorithm is given below.

```

GetLine(Point p0, Point pf) returns Linked List of Points
1.   Create "toReturn", an initially empty Linked List of Points.
2.   dx <- Abs(p0.x - pf.x)
3.   dy <- Abs(p0.y - pf.y)
4.   sx <- -1
5.   sy <- -1
6.   if p0.x < pf.x then sx <- 1
7.   if p0.y < pf.y then sy <- 1
8.   err <- dx - dy
9.   x <- p0.x
10.  y <- p0.y
11.  loop forever
12.    toReturn.Add(x, y)
13.    if x = pf.x and y = pf.y then break out of loop
14.    e2 <- 2 * err
15.    if e2 > -dy
16.      err <- err - dy
17.      x <- x + sx
18.    end if
19.    if e2 < dx
20.      err <- err + dx
21.      y <- y + sy
22.    end if
23.  end loop
24.  return toReturn

```

Figure 1. The “Get Line” function, implementing Bresenham’s Algorithm.

This function returns all Points in a Line between, and including, the two endpoints, which are specified as the parameters to this function. The function keeps track of a “current” x and y value, which are initially set to the x and y values of the first endpoint, and the loop executes until these x and y values match the other endpoint. At each iteration of the loop, either one or both of these values is either incremented by 1 or decremented by 1 in the direction of the second endpoint, thus generating the Points on the line between the two endpoints.

Run-Length Encoding

The purpose of Run-Length Encoding, or RLE, is to encode a string of symbols into a more condensed representation by counting and recording the number of consecutive, unchanging symbols, along with the symbol that the count represents. For

example, if the character set consists of the letters of the English Alphabet (A through Z), then the string AAAABBBCCCCCDD might be encoded with RLE as 4A3B5C2D, which is considerably shorter in length. On the binary level, this string could be encoded in a more condensed manner by using a sophisticated distinction between the numerical symbols and the character symbols; one such approach is to treat the numbers as symbols and then use Huffman Encoding, discussed in the Huffman Encoding section, to encode the string.

In the context of the research presented in this paper, only Binary RLE will be used. With Binary RLE, the character set only consists of {0, 1}, which allows for the significant simplification that the character values do not need to be stored along with the run-length of that value, unlike the previous example. This is because the run-length counts are as long as possible, meaning that if there is a character to be counted that is the same as the previous character in the string, then it will be counted also. Thus, because all contiguous 0's will be counted together, it is known that the next character is a 1, and vice-versa. For example, Binary RLE would encode 000001111000011100 with the values 5, 3, 4, 3, 2. Notice that the characters that these values represent are not shown. However, it is necessary to record the first bit in the string, as otherwise, the run-lengths 5, 3, 4, 3, 2 might be decoded as 11111000111100011, which is the negation of the correct string. Therefore, along with the run-lengths, the first bit, 0 in this case, must also be specified.

Notice that Binary RLE transforms a character string of 0's and 1's into a set of values, not another character string. Although these values could be represented in a character string, there are more efficient ways of representing these values, such as

Fixed-Length Encoding, Universal Encoding, or the method that is used for the research in this paper, Huffman Encoding.

Fibonacci Encoding

Fibonacci Encoding is a type of Universal Encoding method, meaning that it is used to encode a single integer value. Fibonacci Encoding does not use a fixed length string to store a value, but rather has a termination condition which indicates when the reading of a value from a bit string should stop. The termination condition is reading “11” from the bit stream, and immediately following “11”, the first bit of the next value in the bit stream can be written, as it is known that the next bit does not belong to the value that has just been decoded.

The basic idea behind encoding a number using Fibonacci Encoding is to represent each positive integer with a stream of bits such that the sequence “11” does not appear anywhere in the stream, except at the end. Fibonacci Numbers, which follow the equation $F(n) = F(n - 1) + F(n - 2)$, where $F(0) = 1$ and $F(1) = 2$, have the unique property that they can be used to represent any positive number by adding unique Fibonacci numbers together. Because no Fibonacci number will be used more than once and no two consecutive Fibonacci Numbers will be used, the sequence “11” will not appear in the encoded bit stream if each bit, from beginning to end, is used to represent its corresponding Fibonacci number in the summation, where the first bit is $F(0)$, and so on. The last bit in the representation of the value is a 1, because otherwise it would be 0 and therefore be able to be shortened, as it is unnecessary to indicate which numbers are higher than the value that is being encoded. By merely appending an additional “1” at the end of the bit stream, this indicates that the value encoding is done.

The following algorithms, based from those presented in [4], show the encoding and decoding of a value using Fibonacci Encoding.

```
FibonacciEncode(integer "value") returns BitStream
1.   Create the BitStream, "toReturn", initially empty.
2.   value <- value + 1
3.   toReturn <- toReturn + "1"
4.   highestIndex <- 0
5.   for i = 0 to infinity
6.     if F[i] > value
7.       break for
8.     end if
9.     highestIndex <- i
10.  end for
11.  for i = highestIndex to 0 (descending)
12.    if value >= F[i]
13.      value <- value - F[i]
14.      toReturn <- "1" + toReturn (Note "1" is being added to
        the front of toReturn)
15.    else
16.      toReturn <- "0" + toReturn (Note "0" is being added to
        the front of toReturn)
17.    end if
18.  end for
19.  return toReturn
```

Figure 2. The "Fibonacci Encode" algorithm.

```
FibonacciDecode(BitStream input) returns integer
1.   Create the empty Linked List of Booleans, "newBits".
2.   do:
3.     Read one bit from input (and move the read pointer by 1).
4.     Add that bit to newBits.
5.   while the length of newBits is less than two or the 2nd-to-last
        bit of newBits is false or the last bit of newBits is false
6.   toReturn <- 0
7.   for i = newBits.Count - 2 to 0 (descending)
8.     if newBits[i] is true
9.       toReturn <- toReturn + F[i]
10.    end if
11.  end for
12.  return (toReturn - 1)
```

Figure 3. The "Fibonacci Decode" algorithm.

Note that both the encoding algorithm and decoding algorithm involve the use of an array, **F**. This is the array that holds the Fibonacci Numbers; for example, **F**[3] = 5. It is recommended that, rather than calculating the i^{th} Fibonacci Number each time this is

referenced, this array is pre-computed up to some reasonable value, say $F[50]$, which is unlikely to be used except for extremely large values, and then save these values in an array to be used whenever these algorithms are used. This will save a significant amount of processing time compared to finding these values every time the function is called. Additionally, the encoding algorithm that is given here adds 1 to the input, where the decoding algorithm subtracts 1 from the value returned. This is because the traditional Fibonacci Encoding does not work with the value 0, but by doing these two steps, it can encode and decode 0, which is necessary for the image encoding used in this paper. Finally, in these algorithms, a Bit Stream is assumed to have a “pointer” to the next index to read from or write to. The entire Bit Stream used for decoding several values can be passed to the “Fibonacci Decode” function, where it will read from this index, and upon finishing reading a value, leave in index at the beginning of the next item to be decoded.

As a simple example, the number 64 will be encoded and decoded. First, 1 is added to 64 to get 65. The largest Fibonacci number less than or equal to 65 is $F[8] = 55$, and so this is subtracted from 65 to get 10. This process is repeated to find that $F[4] = 8$ is the largest Fibonacci number less than or equal to 10, and likewise, $F[1] = 2$ is the largest Fibonacci number less than or equal to 2 ($10 - 8$), resulting in a value of 0. Because the Fibonacci numbers which add to 65 are $[8]$, $F[4]$, and $F[1]$, the encoding is “0100100011” (note the additional “1” at the end of the bit string). To decode the bit string, observe that the bits at the indices 1, 4, and 8 are set to “1” (excluding the last bit), and so the resulting value is $F[1] + F[4] + F[8] - 1 = 55 + 8 + 2 - 1 = 64$.

Huffman Encoding

The basic idea behind Huffman Encoding, originally presented in [7], is to take a finite input set of characters, calculate the relative frequencies of each unique character in the set, and use these frequencies to generate an encoding for each type of character such that the total length of the encoding is minimized. For the purposes of the research presented in this paper, numeric values will be used instead of generic “characters”, but this does not change the algorithm. The algorithm for using Huffman Encoding generates a tree structure, which can be used to encode and decode values, and is itself easy to encode and decode efficiently. The following algorithm outlines the Huffman Encoding Process.

Node structure

- “Value”, an integer representing the unique value of the node. This is the value that is encoded.
- “Frequency”, a float in (0, 1] representing the relative frequency of “value” in the input list of non-unique integer values.
- “Child A” and “Child B”, two Nodes which are the children of this Node in the tree structure. Note that either both of these can be null or neither of these, but not only one.

Figure 4. The “Node” structure, representing a Node in a Huffman Tree.


```

ApplyHuffmanEncoding(Linked List of integers, "values") returns Node
1.   Create "frequencies", which is a Linked List of Nodes.
2.   Populate "frequencies" with the unique integers in "values" and
     their respective frequencies. Each Node in "frequencies" should
     not have any children Nodes at this time.
3.   Sort "frequencies" from lowest to highest frequency.
4.   while the number of Nodes in frequencies is greater than 1
5.       Remove the first 2 Nodes of frequencies. Call these "A" and
         "B".
6.       Create a new node, whose "value" is -1 (-1 is assumed to
         never be used), whose "frequency" is A.Frequency +
         B.Frequency, and whose children are A and B.
7.       Insert this new node into "frequencies" such that the
         ascending sorted order is still maintained.
8.   end while
9.   At this point, "frequencies" contains only one Node, which is
     the root of the Huffman Tree. Return that Node.

```

Figure 5. The "Apply Huffman Encoding" algorithm.

The algorithm given here returns a tree structure in the form of the Node that is the root of the tree, whereby all other Nodes can be accessed. All intermediate Nodes (i.e., Nodes which have children) have values equal to -1, and in the context in which Huffman Encoding is used in this research, only non-negative integers will be encoded, so this will never be a conflict. In general, some "null" value or any such indicator value will work in place of -1 if necessary. All leaf Nodes (i.e., Nodes which do not have children) have values corresponding to a unique value in the input list, and no two leaf Nodes have the same value. To find the encoding of a value, simply backtrack from the leaf Node containing that value to the root of the tree, and if it is the left child of a Node, a 0 is used to encode the value and otherwise a 1 is used. The order of these bits from the root to the leaf is the unique encoding for the value. Note that for optimization purposes, a lookup table of values and corresponding encodings is stored after the tree is generated so that the tree does not have to be re-traversed to find a particular value, but rather the encoding of a value can be found in $O(1)$ time by using this table.

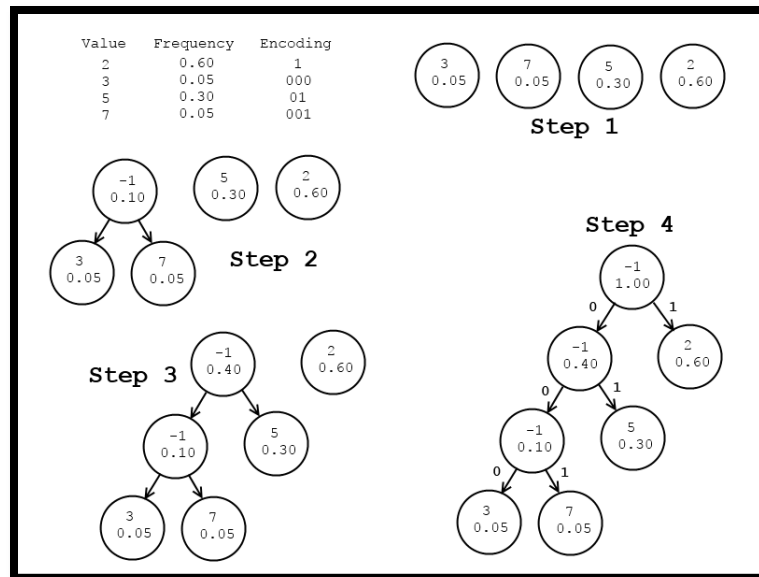


Figure 6. This shows the steps to create a Huffman Tree from a set of unique values and associated frequencies. The unique values and associated frequencies are found by scanning all input data and recording the frequencies of each unique value. The final encoding of each value using the resulting Huffman Tree is also shown.

A value is decoded by following the bits in the stream to be decoded, from the root Node, travelling down the tree until reaching a leaf Node, at which point the value is decoded. For example, if the bit stream “001” is being decoded from the example in Figure 6, starting at the root of the tree, simply follow the left child, then the next left child, then the next right child, to get to the leaf Node with value 7. A property of Huffman Encoding, as with any prefix encoding scheme, is that the encoding values need not have fixed lengths, meaning that all of the encoded bits of adjacent values can be concatenated together. Therefore, after decoding one value, the next bit in the bit stream is used to start decoding the next value, again from the root of the Huffman Tree.

The Huffman Tree itself can be encoded in an efficient manner by taking advantage of the tree structure, as well as the fact that the values of the intermediate Nodes do not matter, and, once the tree is constructed, the frequencies also do not matter. This encoding and decoding method can be used for any binary tree with non-negative

integer values only on the leaf nodes, and thus it can be used for encoding and decoding Huffman Trees.

The algorithm to encode the tree works as follows. Starting at the root, apply a Depth-First Traversal of the tree, where the recursion always chooses the left child before the right. If the current node being visited is an intermediate node, print a “1”. If it is a leaf node, print a “0” and then print the value using Fibonacci Encoding. When the DFT terminates, the tree is fully encoded.

To decode the Huffman Tree, create the tree “in-place” as the bits from the encoding process are read. Start by creating the root Node in the Huffman Tree, then read in the first bit. If it is a 1, and if the left child has not been created, then create the left child and recursively repeat (from the current position in the bit stream) on the newly created Node. Likewise, if it is a 1 and the left child has been created, then the right child must not have been created, so create a right child and recursively repeat on that newly created node. Finally, if a 0 is read from the bit stream, then the node is a leaf node, so no children are created, and the value is set by using Fibonacci Decoding to read in an integer from the bit stream. Recall that Fibonacci Decoding does not need the exact length of what it will read in, it will stop reading after the reading of the number has been completed, and it will leave the next bit to be read as the first bit directly after the number which was just read. After the recursive procedure has finished, the tree will be fully created and usable for Huffman Decoding.

Note that the length of the tree encoding (number of bits) need not be stored. When decoding the tree the decoding process will end exactly when the tree is completed and it will require no more or no less bits. Therefore the bits that encode the tree indicate

when to stop building the tree, simply due to the fact that there is nowhere left in the tree to expand, as all leaf Nodes have been identified. The length of the encoding of the tree does not need to be specified, and bits pertaining to something else (the rest of the data file) can be added directly after the encoding of the tree, and an interpreter will be able to know exactly where these bits begin after creating the tree.

Image Compression in General

The purpose of image compression is to represent an input image, which is a 2-dimensional array of color values, into a binary format that is smaller than a representation that simply lists the color values in a binary file, as the RAW file standard does. Image compression methods take advantage of patterns in the image data, such as redundancy of colors or irrelevant colors such as noise, so that by generalizing the structure of the colors in the image and eliminating or changing certain colors or pixels, the number of bits used to store an image in binary format can be reduced.

There are two types of image compression—lossless compression and lossy compression. Lossless compression is used by image encoding standards such as BMP and PNG, and these standards encode the input image such that, when decoded, each Pixel of the output image is the same as the corresponding Pixel of the input image. Lossless image compression is important for applications such as images with sharp transparent edges, webpage images, and images which are used to store data as a two-dimensional array. The other type of image compression is lossy image compression, which is used by the JPEG and GIF encoding standards. With lossy compression, the output image, when decoded from the encoding of the input image, does not necessarily equal the input image at the Pixel level. It is the attempt of lossy compression algorithms

to alter the input image in such a way as to cause the image to conform to patterns which can be encoded, thus reducing the size of the output file, while at the same time not significantly changing the visual appearance of the image when viewed by a human.

The Portable Network Graphic File Standard

The Portable Network Graphic, or PNG, is a lossless image compression standard which supports colors involving transparency, and this is often used for internet graphics and for images which require lossless color representation or transparent color details. Unlike the older file format used for web-based images, the GIF, the PNG can support any number of unique colors, whereas the GIF can result in a lossy representation of the image when it posterizes to 256 colors, the maximum number of unique colors allowed in a GIF image due to its 8-bit color palette.

The compression of an image into the PNG format involves the use of two techniques. The first is the use of a predictive filter, which makes use of the three Pixels above, left, and diagonally above-and-left of a target Pixel to predict what the color should be, based on the colors of those three Pixels. The actual color of the target Pixel is then compared to the predicted value, and the difference between these two colors is stored; the difference is a small value in regions of similar color, and thus more compact to store. Once these differences are found, the DEFLATE technique is used, which uses a combination of LZW encoding and Huffman encoding, to compress these values in a lossless manner.

The PNG is the most commonly used image format for compressing cartoon images. The file specification and implementation details are fully described in [3].

The JPEG File Standard

Another commonly used image compression standard, which can be used to compress cartoon images that do not involve transparency, is the JPEG. The JPEG image format is lossy, but the amount of loss to the input image can be reduced by adjusting a quality metric for the image which is being compressed; a higher quality means that the size of the binary file will be larger, and likewise, a lower quality mean that the size of the binary file will be smaller. With the JPEG algorithm, an encoded image can be created in which a human can perceive it as being the same image as the original, but on the Pixel level the image will be quite different, especially due to compression artifacts that form on the image.

The JPEG image encoding works by first evenly dividing the image into 8x8 Pixel blocks and then applying a Discrete Cosine Transform (DCT) to each of these blocks. The DCT attempts to fit a block to each of 64 predefined bit patterns, each of which is an 8x8 block as well, where the floating-point result of each is stored. Because the 8x8 blocks from the input image will not necessarily match all of the 64 predefined bit patterns, and because a DCT does not necessarily represent the matching to one of the bit-patterns exactly, there are often noticeable Pixel artifacts that arise in a JPEG image. Due to the fashion in which the data is stored, the 8x8 blocks that were used to encode the image are often visible at the Pixel level due to these artifacts creating an outline for the blocks when there should not be an outline. When an image is encoded with the JPEG algorithm, a quality metric in the range of [1, 100] is used, and this specifies the accuracy of the discretization of the continuous coefficients from the DCT to discrete values which

can be stored in binary. With a low quality value, these continuous values are stored in a fewer number of bits than with a higher quality value.

JPEG images are often used for photographs, because the artifacts, unless severe, are difficult to notice, given that each Pixel color will be slightly different in the lossless input image. The JPEG encoding can also be used to compress cartoon images when the small size of the image is more important than the presence of artifacts. The research presented in this paper assumes that the input image may have been previously encoded with the JPEG algorithm, and thus first eliminates these artifacts before encoding the cartoon image. The file specification and implementation details are fully described in [6].

Quad Trees and Application to Cartoon Image Compression

A currently existed approach to compressing cartoon images, as presented in [15], is through the use of Quad-Trees. A Quad-Tree, in general, is a tree structure in which every node in the tree has either four children or none. If a node has no children, it is a leaf node. Quad-Trees can be used in image analysis to divide the image evenly into four subsections, where each of these subsections can be further divided recursively. See Figure 7.

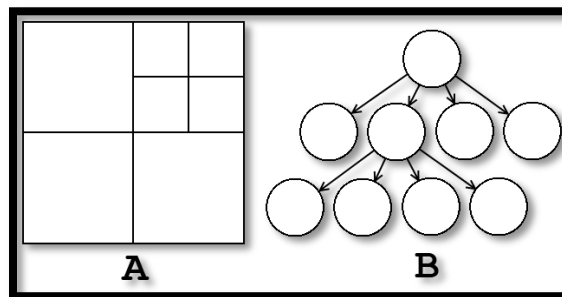


Figure 7. This shows how an image might be decomposed into sections with a Quad-Tree representation. The full rectangle of the image (A) is decomposed into sub-rectangular sections, which is represented by the Quad-Tree (B), where each node in the tree represents a section, and the node will be a leaf node if the section is not divided, and otherwise it will have the children leading to the subsection. The root node of the tree is the rectangle of the full image.

Note that the Quad-Tree decomposition can be extended to images which are not square and do not have side lengths which are powers of 2 by dividing a section into 4 sub-sections along one dimension, rather than along 2 dimensions, when one of the side lengths of the section is greater than 4 times the other side length.

For cartoon image compression, the concept that was proposed in [15] involves analyzing the decomposed sections in terms of the color representation to determine if the section should be further divided. This is a recursive procedure initiated on the image as a whole. If the section of the image contains one or two colors or is less than 4 pixels in area, then the section is not sub-divided, and otherwise it is. When a section is not further divided, the colors in the section are stored, and if the section is larger than 4 pixels in area, Binary RLE is used to store the representation of the section. Because the Quad-Tree composition is a well-defined procedure, given a Quad Tree and the full image width and height, the resulting section locations and dimensions can be determined. The Quad Tree is saved, along with the leaf node data, similar to how the Huffman Tree is saved, as described in the Huffman Encoding section, by using a DFT along with indicating when a node is a leaf node.

Quad-Trees work effectively for cartoon image compression because in large regions of the same color, the likelihood that a large section in the Quad-Tree Decomposition contains two or less unique colors is much higher than compared to images such as photographs. Because of this, the Quad-Tree will contain several sections which have large areas, thus covering a larger portion of the image with a small amount of storage space. In fact, the Quad Tree presented in [15] outperforms the PNG in 5 out

of 6 test cases. Note, however, that the Quad-Trees presented in [15] do not perform well on input images with noise, anti-aliasing, or gradients.

Cartoon Image Region Detection

Fundamental Idea

The unique property of cartoon-style images is the extensive use of large regions of the same color. By taking advantage of this property, cartoon images can often be compressed at much higher rates than standard methods of generic image compression.

Implementation

Define a Texture object as containing a Linked List of Region objects, where a Region is a hashed (for constant-time lookup) Linked List of Point objects: x-y integer pairs. These Points represent the indices into the Texture's Pixel array which belong to a Region, where no two Regions share such an index. That is to say that all Regions are completely disjoint. Further, all Pixels in the Texture's Pixel array must belong to a Region after the image processing has been completed, and accordingly, each Pixel object stores a pointer to the Region which contains it for a fast reverse-lookup. Initially, the Regions in the Texture will not be known, and so if the Pixel does not belong to a Region, the pointer will be set to null.

The algorithm used in this research to find all regions is given in Figure 8.

```

FillAllRegions(Texture texture) returns nothing
1.   for each Pixel, p, in texture
2.       if p.Region = "null"
3.           CreateNewRegion(texture, p.X, p.Y)
4.       end if
5.   end for

```

Figure 8. The “Fill All Regions” algorithm.

```

CreateNewRegion(Texture texture, int x, int y) returns nothing
1.   Create a Queue of Points, "bfsQueue", initially empty.
2.   bfsQueue.Enqueue(x, y)
3.   Create "visited", a Hash Table of points, initially empty.
4.   Create a new Region, "toAdd", initially empty.
5.   while bfsQueue is not empty
6.       (xCurrent, yCurrent) <- bfsQueue.Dequeue()
7.       if visited.Contains(xCurrent, yCurrent) or
          !texture.Contains(xCurrent, yCurrent) or
          texture[xCurrent, yCurrent].Region != null
8.           continue (Go to line 5)
9.       end if
10.      visited.Add(xCurrent, yCurrent)
11.      for each (xNew, yNew) in
          {(xCurrent - 1, yCurrent), (xCurrent + 1, yCurrent),
           (xCurrent, yCurrent - 1), (xCurrent, yCurrent + 1)}
12.          if texture.Contains(xNew, yNew) and
              ColorMatch(texture[xNew, yNew].Color,
                          texture[xCurrent, yCurrent].Color)
13.              toAdd.Add(xNew, yNew)
14.              texture[xNew, yNew].Region <- toAdd
15.              bfsQueue.Enqueue(xNew, yNew)
16.          end if
17.      end for
18.  end while
19.  texture.Regions.Add(toAdd)

```

Figure 9. The “Create New Region” algorithm.

```

ColorMatch(Color A, Color B) returns Boolean
1.   return A.RGBA = B.RGBA

```

Figure 10. The “Color Match” algorithm.

The algorithm for creating all Regions in a Texture is a straight-forward implementation of the Breadth-First Traversal Algorithm, in the context that a Vertex in the corresponding Graph is a Pixel in the Texture, and an Edge exists between two Pixels

simply if they are orthogonally adjacent (i.e., above, below, to the left, or to the right of one-another). The alteration that was made to the BFT algorithm is simply that the traversal happens between two pixels if and only if they are the exact same color. This check is performed through the use of the “Color Match” function.

This algorithm expands a Region to be as large as possible, meaning that if there is a Pixel to add to the Region, then it will be added. In other words, there cannot exist two adjacent Regions which have the same color. This can be easily shown by counter-example. Suppose there are two adjacent Regions of the same Color, then there must be at least one Pixel in each Region which are adjacent with each other, and therefore, when the BFT examines one of these Pixels, it will examine the other, due to the fact that the BFT is applied to find the Regions in no particular order. Thus, the other Pixel could not already belong to a different Region. Because the Pixels are the same Color (or, in fact, cause the “Color Match” function to return true, regardless of what the executing code is), then by the if statement at line 12 of the “Create New Region” function, the Pixels must belong to the same Region. Thus, Regions are as large as possible, given the contents of the “Color Match” function, which is altered in Figure 12 in such a way that does not change this proof. In addition, it is clear that each Pixel belongs to a distinct Region, as by line 7 in the “Create New Region” function, if a Pixel has already been assigned a Region, then it cannot be assigned another. Similarly, each Pixel has a Region assigned to it by the time that the algorithm finishes executing due to the loop in the “Find All Regions” function, which verifies that all Pixels which do not have a Region are assigned one.

This algorithm is able to assign all Regions to all Pixels in $O(n)$ time, where n is the number of Pixels in the image. Let k_r be the number of Pixels in Region r . Then populating Region r takes $O(k_r)$ time, as this is a constant cost modification to the BFT algorithm, which executes on the order of the number of Pixels in the search, k_r . Because it was shown that all regions are disjoint, the running time is thus $O(\sum_{i=0}^R k_i) = O(n)$, where R is the number of Regions.

The algorithm, as thus far described, will only work well for cartoon images made up of perfectly segmented color regions. This is because, although every distinct color region is correctly identified, expanded to the maximum Pixel area, and distinctly stored, it might be the case that the original image to be compressed contained noise, artifacts, or aliasing. The Color Reduction section provides methods for reducing these problems, one of which is through altering the “Color Match” function.

Color Reduction

Problem

The algorithm that has thus far been presented for finding the disjoint Regions of Color in a Texture assumes that the Pixel Color in each Region will contain the *exact* same RGBA value. Although the algorithm will correctly do what is intended, the number of Regions may be in the thousands, due to the fact that many colors (uniquely identified by RGBA value) may exist in the image that one might expect to be only a single color by looking at it with the naked eye. This could be caused by a number of issues, such as compression artifacts from other, previously applied compression algorithms (such as JPEG encoding), or anti-aliasing due to an art program such as

Photoshop, GIMP, or Paint.Net attempting to make the image look more pleasant by smoothing the lines, shapes, and colors, causing several different RGBA values for a single intended Color.

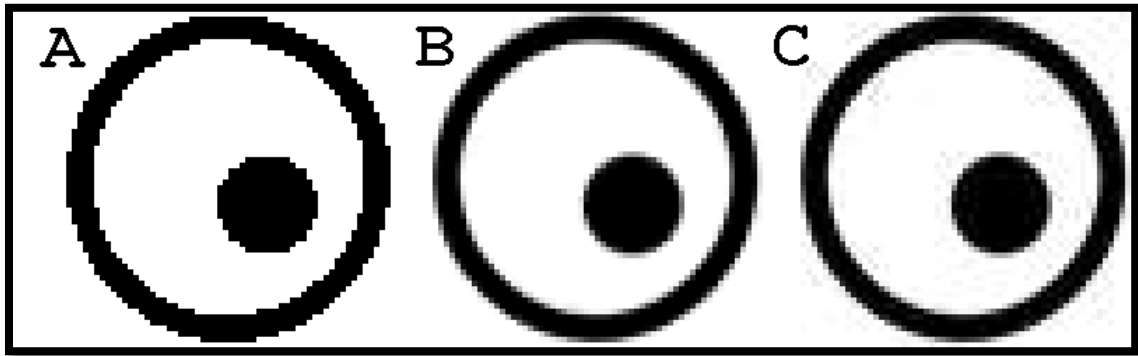


Figure 11. A) This shows a cartoon image with rigid color regions, and no noise or artifacts. B) This shows the application of anti-aliasing to the shapes in A, causing a smoother appearance but a much larger number of distinct colors. C) This shows the effects of JPEG artifacts when B is encoded using this lossy encoding; this increases the number of distinct colors even further. Note that all images are scaled 400% to show pixel detail.

It is clear that the assumption that all color regions in the image will be made up of the exact same RGBA value is an impractical one. This section attempts to reduce the number of artifacts and subtle color differences in order to greatly reduce the number of Regions, while not altering the image significantly.

Modification of Region Detection Algorithm

The region detection algorithm, as described, only expands a Region when adjacent pixels are the exact same color (by RGBA value) as the pixel from which it is being expanded. It is often the case that color regions which are perceived to be one color are actually composed of several unique colors with slightly different RGBA values, and because these RGBA differences are slight, the “Color Match” function can be altered to detect these slight differences.

```
ColorMatch(Color A, Color B) returns Boolean  
1.    return (A.RGBA - B.RGBA).Length <= Threshold
```

Figure 12. The modified “Color Match” algorithm

Note that the RGBA value of a color is a 4-dimensional vector, and thus the subtraction of two RGBA values is an RGBA value, of which the Euclidean Length can be found. The value of Threshold can be set by the creator of the file that is being encoded, as this is a parameter to be set to get the best appearance, due to the fact that this value contributes to the lossiness of the image. If Threshold is 0, then this function is the same as the previous function, comparing exact RGBA values, and thus resulting in a lossless image. For the test images used in this research, the Threshold that was determined to be the most perceptually pleasing was 4. Recall that each of the components in the RGBA vector ranges from 0 to 255, inclusive.

This approach of seeing if the Color Vector distance is under some threshold is applied to the Color Vector distance between the Pixel which is being considered to be added to the Region and the Pixel adjacent to it, which is already in the Region. This approach tends to work well for dealing with intentional gradients and artifacts of anti-aliasing, but it tends to do poorly with JPEG artifacts and other noise. This is due to the fact that Pixels are added to a Region in a “chain” fashion, where every link in the chain could be only a slight increase (that is less than the threshold), but over a long distance could pick up a huge difference in Color, thus inadvertently merging two color regions together into one Region object, when visually they should have been separate. This tends to happen more with random noise or JPEG artifacts because the Pixel Color values could be altered in such a way that allows this “Pixel Chain” to gap between two regions

that should be separate, whereas anti-aliasing artifacts usually are not dramatic enough to cause this to happen in the absence of noise.

One possible method for solving this problem would be to simply use a lower Threshold value, but this is not a good idea because it could then cause Pixels that should belong to the Region to not be included. Rather than balancing this value to find the best Threshold, a better approach is to use the average Color value of the region as the Color for comparison in the “Color Match” function, rather than the Color value of the adjacent pixel. In the Region object, store the average Color. Because it is known how many Pixels belong to the region, a new average Color can be found easily upon each addition of a new Color, so when the algorithm adds a new Pixel, the average Color is updated. This can be achieved by modifying line 12 of the “Create New Region” function to use this average Color instead of the adjacent Pixel’s Color.

This approach is much less likely to mistakenly merge two visually distinct color regions into one Region object. However, it is more likely to create a larger number of regions by not absorbing Pixels into a Region that is expanding when it should, because random noise or JPEG artifacts may cause a color difference to be higher than the Threshold from the average Region Color. The Threshold for this method, as opposed to using the nearest Pixel’s Color in the expansion, is much less sensitive, and a higher value of 12 is recommended, in general, but again this can be made an option to the end-user of the compression software to get the best results if appropriate. Finally, it should be noted that the method of using the average Region Color cannot be used for detecting Regions with color gradients, and often leads to severe posterization when faced with gradient

regions. This is discussed in more detail in the Gradient Detection and Application section.

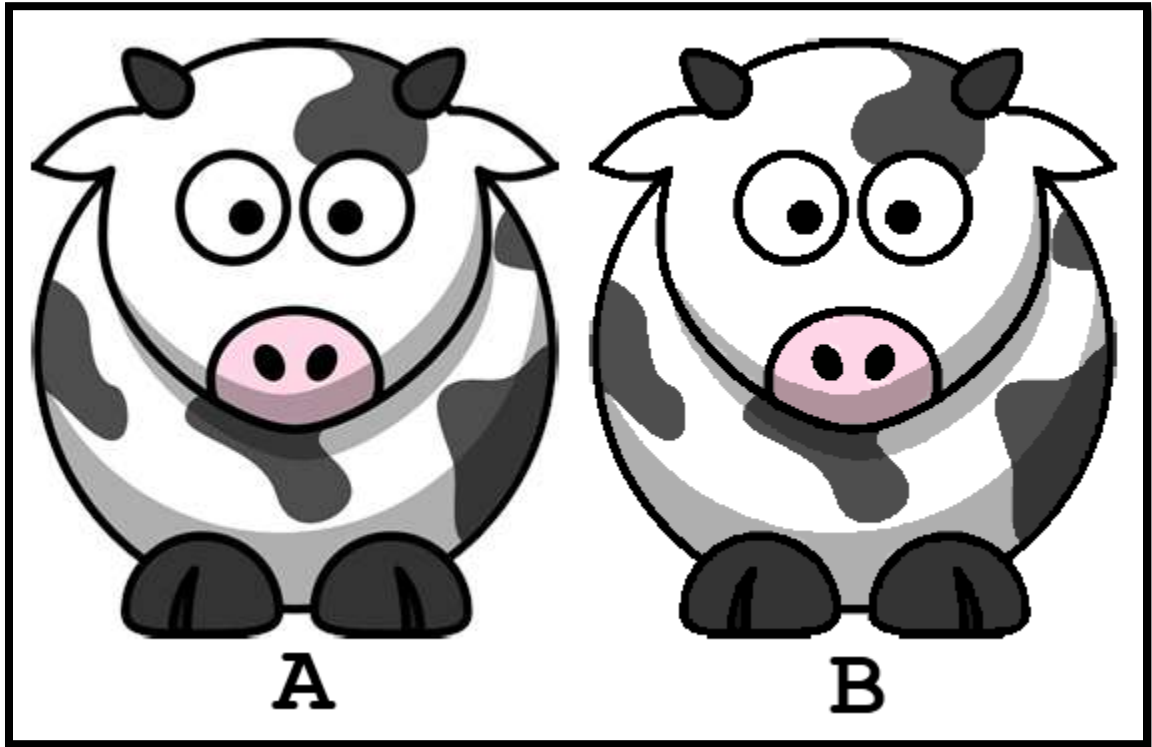


Figure 13. This shows the input image (A), which was created using a lossy JPEG encoder, and the resulting image (B) after color reduction was applied with color-average based Pixel absorption. The Threshold used to create B was 12.

The next section describes how to merge regions together after the previous process has been performed, but when there are several smaller regions very similar in Color value to larger, adjacent regions.

Region Concatenation

Due to the fact that nearest-pixel based region expansion is not practical for noisy images, it is recommended that Region Color average based expansion is used. Although this approach, with a reasonable Threshold value used, does not lead to two visually distinct color regions being mistakenly treated as one, it does occasionally result in one visually distinct color region being broken down into several smaller Regions of only

slightly different RGBA Color values. The goal of the Region Concatenation is to identify these adjacent regions that differ only slightly and concatenate them into one region. Doing this will save space when the image compression takes place, as fewer regions means fewer bits used to represent the image.

The process of merging the adjacent Regions together takes advantage of the fact that, even among images with noise or heavy artifacts, there will always be one region that is considerably larger than all of the other regions that have visually the same color (with different RGBA values) that should be absorbed into it. This is because the Region Detection Algorithm still merges some Colors together, even though due to noise or artifacts, it may be unable to find the largest Region of what is visually the same Color because the RGBA value of one of the Pixels affected by noise is just too far beyond the range of the Threshold. Thus, the Regions surrounding this larger Region, which are visually the same color, are usually rather small (less than 10 Pixels in area).

The algorithm, shown in Figure 14, works by progressively refining the image, Region by Region, until no Region concatenations can be made or the smallest Region size is too large (determined by some minimum area value).

```

ConcatenateRegions(Texture texture) returns nothing
1.   for i = 1 to MinimumRegionArea
2.       FillAllRegions(texture)
3.       Remove all Regions from texture.Regions containing less than
        or equal to i Pixels, and store these removed regions in the
        Linked List "removedRegions".
4.       if removedRegions has no entries
5.           return
6.       end if
7.       for each Region, r, in removedRegions
8.           for each Pixel, p, in r
9.               if (p.Region = "null")
10.                  Pixel q <- GetBestPixelMatch(texture, p)
11.                  p.Region <- q.Region
12.                  p.Color <- q.Color
13.                  q.Region.Add(p)
14.                  r.Remove(p)
15.               end if
16.           end for
17.       end for
18.   end for

```

Figure 14. The "Concatenate Regions" algorithm.

```

GetBestPixelMatch(Texture texture, Pixel p) returns Pixel
1.   Create a Queue of points, "bfsQueue", initially empty.
2.   bfsQueue.Enqueue(p.X, p.Y)
3.   Create "visited", a Hash Table of Points, initially empty.
4.   Create a Hash Table of Pixels, "toConsider", initially empty.
5.   while bfsQueue is not empty
6.       (xCurrent, yCurrent) <- bfsQueue.Dequeue()
7.       if visited.Contains(xCurrent, yCurrent) or
        !texture.Contains(xCurrent, yCurrent)
8.           continue (Go to line 5)
9.       end if
10.      if texture[xCurrent, yCurrent].Region != null
11.          toConsider.Add(texture[xCurrent, yCurrent])
12.          continue (Go to line 5)
13.      end if
14.      visited.Add(xCurrent, yCurrent)
15.      for each (xNew, yNew) in
        {(xCurrent - 1, yCurrent), (xCurrent + 1, yCurrent),
         (xCurrent, yCurrent - 1), (xCurrent, yCurrent + 1)}
16.          if texture.Contains(xNew, yNew)
17.              bfsQueue.Enqueue(xNew, yNew)
18.          end if
19.      end for
20.  end while
21.  return the Pixel, q, in toConsider which results in the lowest
    value from the DifferenceMetric(p, q) function.

```

Figure 15. The "Get Best Pixel Match" algorithm.

```

DifferenceMetric(Pixel A, Pixel B) returns float
1.   spatialDistance <- ((A.X, A.Y) - (B.X, B.Y)).Length
2.   colorDistance <- (A.RGBA - B.RGBA).Length
3.   return (1 - Weight) * colorDistance + Weight * spatialDistance

```

Figure 16. The “Difference Metric” algorithm.

This algorithm works by first applying the Region expansion algorithm, and then removing all Regions which contain only 1 Pixel. Thereafter, for each Pixel belonging to these removed Regions, a BFS algorithm is applied to find all neighboring Pixels which still belong to a Region (which was therefore large enough to not be removed). Finally, each of these Pixels is examined using a metric that compares both the Color Difference and the Euclidean Distance from the Pixel that started the BFS, and the Pixel that started the BFS has its Color and Region set to that of whichever Pixel in the set has the lowest Difference Metric. This process is repeated iteratively, where upon each iteration, Regions containing 1 more Pixel than the last iteration are removed, up until a certain Minimum Region Area (where the recommended value is 10, but again this can be made an available parameter to an end-user of some compression software).

The Difference Metric uses a Weight to compare the Euclidean Distance with the RGBA Color Distance. If only the Euclidean Distance were used (if the Weight is set to 1), then unclassified Pixels at the boundaries of visually distinct color regions may be absorbed by the incorrect region, simply because it is spatially closer, leading to odd-looking edges around these regions which should otherwise be smooth. On the other hand, if only Color Distance is used (if the Weight is set to 0), then noise-effected unclassified Pixels could be set to the incorrect Region. The value of the Weight can be made a parameter to the end-user of the compression software to get the best-looking results, but in the general case, the recommended Weight is 0.2.

Gradient Detection and Application

The methods for Region detection that have been described thus far assume that all Regions are made up of a solid, unchanging Color. This was, in fact, the central idea as to why cartoon images would compress better than generic images, but it is often the case that cartoons involve Color Gradients, which need additional detection and processing. This section describes what a Color Gradient is, how to detect these, and how to apply a Color Gradient when recreating an image.

A Color Gradient is composed of a primary and secondary Color (where each Color has an associated RGBA value, and the Colors are not the same). A Gradient consists of two Points on the image; these Points correspond to where the primary and secondary Colors are located. A Gradient also includes a shape and a function. The function determines how the Colors are interpolated between the primary and secondary Color at Pixels between the two respective Points; for the purposes of how the Gradients will be detected and applied for this paper, a linear interpolation will always be used. The Gradient shape is the manner in which the Pixels in the Region are set based on the interpolation. This paper will discuss three types of shapes, but an essentially limitless number of other shapes may exist. The types of shapes that will be described are a line, a circle, and a mirror. A line consists of all Pixels between the primary and secondary Pixel being linearly interpolated, with all other Pixels in the Region being interpolated similarly along lines parallel to the original. The mirror shape is similar, except this process happens a second time with the secondary Pixel being moved in a “mirror-like” fashion about the primary Pixel, across the line orthogonal to the line between the primary and secondary Pixel. The circle shape linearly interpolates along the line from

the primary to secondary Pixel, except rather than continuing in a parallel manner, the line radiates around the primary Pixel in a full circle, and the interpolation is performed along these radials.

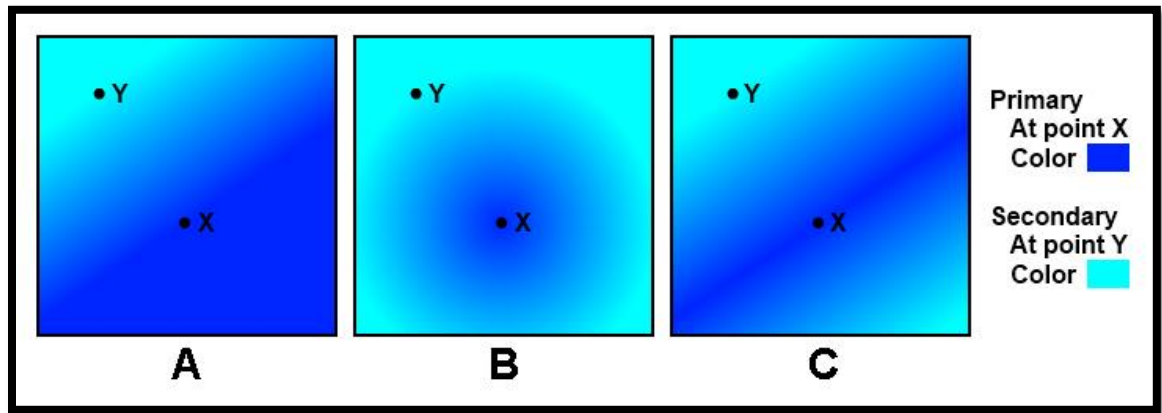


Figure 17. This example shows the application of a line (A), circle (B), and mirror (C) Gradient. The primary Color is at Point X and the secondary Color is at Point Y. In each of these, the Region is the square which contains the colors. Note that the shape of the Region is not the same as the shape of the Gradient.

Gradient Application

Gradient Application must be applied when the image is decoded, so as to fill each Region with the appropriate colors that are interpolated from the Gradient, but the Gradients are also applied at the time of encoding the image for difference comparison with the original. This section presents the methods for applying the Gradients to a Region. The process works by identifying the shape of the Gradient to apply, and based on the shape and the location of the Pixel whose Color is being set, linearly interpolating the Color. The algorithm is presented below.

```
ApplyGradient(Region region, Gradient gradient) returns nothing  
1.   Set the color of all Pixels in region to "null".  
2.   if gradient.shape = "line" or gradient.shape = "mirror"  
3.       ApplyLineGradient(region, gradient)  
4.   else if gradient.shape = "circle"  
5.       ApplyCircleGradient(region, gradient)  
6.   end if  
7.   SnapColors(region)
```

Figure 18. The “Apply Gradient” algorithm.

The “Apply Gradient” function takes the parameter of the Region to have the Gradient applied, where the Region has already been populated with the corresponding Pixels that make up the Region, and the Gradient to apply to the Region, where the Gradient object contains two Color objects (the “primary Color” and “secondary Color”), two Point (integer x-y pair) objects (the “primary Point” and “secondary Point”), and a shape (either “line”, “mirror”, or “circle”). The Gradient application works differently based on the shape of the Gradient to apply, so this function determines the shape of the Gradient and calls the appropriate function to apply the shape-specific Gradient, and once that has finished, sets the colors of all Pixels in the Region that were outside the range of the Gradient, and thus not applied, with the “Snap Colors” function. These functions are detailed in Figure 19 and Figure 20.

```

ApplyLineGradient(Region region, Gradient gradient) returns nothing
1.   length <- 1000
2.   x0 <- gradient.primaryPoint.x
3.   y0 <- gradient.primaryPoint.y
4.   xf <- gradient.secondaryPoint.x
5.   yf <- gradient.secondaryPoint.y
6.   dx <- xf - x0
7.   dy <- yf - y0
8.   orthoLine <- GetLine((x0 + length * dy, y0 - length * dx),
                          (x0 - length * dy, y0 + length * dx))
9.   for each Point, p, in orthoLine
10.    line <- GetLine((p.x, p.y), (p.x + dx, p.y + dy))
11.    loop 2 times
12.    for i = 0 to line.Count - 1
13.    if region.Contains(line[i].x, line[i].y)
14.    region.texture[line[i].x, line[i].y].color =
        InterpolateColor(gradient.primaryColor,
                          gradient.secondaryColor, i / (line.Count - 1))
15.    end if
16.    end for
17.    if gradient.shape = "line" then break out of loop
18.    else line <- GetLine((p.x, p.y), (p.x - dx, p.y - dy))
19.    end loop
20.  end for

```

Figure 19. The “Apply Line Gradient” algorithm.

The application of the linear Gradient works by first finding the line that is orthogonal to the line between the primary and secondary Points, where the orthogonal line passes through the primary Point. A line, as it is represented in the code, is a Linked List of Points, and these Points are found by applying Bresenham’s Line Algorithm, and the “Get Line” algorithm for this is presented in the Background section. The orthogonal line is used as a baseline for where to apply strips of pixel-thick Gradients to the Pixels in the Region. Because the Gradient specifies only the start and end point of the linearly shaped Gradient, not the width of the region, there is no way to determine what the length of this orthogonal line should be from the Gradient object. Therefore, this orthogonal line should be long enough to cover all applicable Pixels in the Region, and the length of this orthogonal line is set by the “length” variable in Line 1 of Figure 19. If this length is too short, then the Gradient will not get applied at all Pixels that it should be applied to, but if

it is overly long, then this will not make the application of Colors to Pixels in the Region incorrect, as if the corresponding Point is not in the Region, then it is simply ignored. The only drawback of making the length of the orthogonal line too long is that it may become a performance issue; however, only a single operation needs to be performed per Point in this line to determine if it belongs to the Region, so it is a minor performance concern.

Once the orthogonal line is found, for every Point in this line, a new line is found which is parallel to the original line between the primary and secondary Points, and this new line is the same length as the line between the primary and secondary Points. Recall that the orthogonal line passes through the primary Point, and thus the correct skew is given to the linear Gradient from each of these corresponding lines. For each of these lines off of the orthogonal line, the Color of the corresponding Pixel in the Region (if the Pixel is in the Region) is found by doing a simple linear interpolation between the primary and secondary Colors of the Gradient based on where the Pixel lies on the second line, scaled to be in the range of $[0, 1]$.

This procedure is applicable to a mirror Gradient as well by simply reversing the lines which are extended from the orthogonal line. The first point of these lines is the same, but the second point is “mirrored” across the orthogonal line, creating the mirror-shaped Gradient.

Figure 20 gives the algorithm that applies a circle Gradient to the Region.


```

ApplyCircleGradient(Region region, Gradient gradient) returns nothing
1.  x0 <- gradient.primaryPoint.x
2.  y0 <- gradient.primaryPoint.y
3.  xf <- gradient.secondaryPoint.x
4.  yf <- gradient.secondaryPoint.y
5.  finalDistance <- Ceiling(Distance((x0, y0), (xf, yf)))
6.  Create "bfsQueue", an initially empty queue of Points.
7.  Create "visited", an initially empty Hash Table of Points.
8.  bfsQueue.Enqueue(x0, y0)
9.  while bfsQueue.Count > 0
10.   currentPoint <- bfsQueue.Dequeue()
11.   visited.Add(currentPoint)
12.   currentDistance = Distance((x0, y0),
    (currentPoint.x, currentPoint.y))
13.   if currentDistance > finalDistance
14.     continue (Go to Line 9)
15.   end if
16.   region.texture[currentPoint.x, currentPoint.y].color =
    InterpolateColor(gradient.primaryColor,
    gradient.secondaryColor, currentDistance / finalDistance)
17.   for each Point, p, in
    {(currentPoint.x - 1, currentPoint.y),
    (currentPoint.x + 1, currentPoint.y),
    (currentPoint.x, currentPoint.y - 1),
    (currentPoint.x, currentPoint.y + 1)}
18.     if region.Contains(p) and !visited.Contains(p)
19.       bfsQueue.Enqueue(p)
20.     end if
21.   end for
22. end while

```

Figure 20. The “Apply Circle Gradient” algorithm.

The “Apply Circle Gradient” function is similar to the “Apply Line Gradient” function in that it finds the Color value for each Pixel in the Region that is within the area of the Gradient through linear interpolation; however, it does not use parallel lines to apply the Colors, but instead, it compares the distance of each Pixel in the area of the Gradient to the center Point of the circle for the linear interpolation. The algorithm works by first finding the maximum distance, called “final distance”, which is the Euclidean distance from the primary Point of the Gradient to the secondary Point. It then applies a Breadth-First Traversal from the Point at the center of the gradient circle outward, and at each new Pixel visited, if the distance from that Pixel to the center Pixel is less than the “final distance”, then the corresponding Color of that Pixel is found by linearly

interpolating the primary and secondary Color based on the ratio of that distance to the “final distance”, which must be in the range [0, 1], because if this were greater than 1 then the Pixel would not have been considered. Note that this will not set the value of all Pixels in the Region if the Gradient circle does not enclose the Region boundaries, but the unset Pixels are set with the “Snap Colors” function, which is described in Figure 21.

```
InterpolateColor(Color A, Color B, float value) returns Color  
1.   value <- Clamp(value, 0, 1)  
2.   Create a new Color, "toReturn".  
3.   toReturn.Red <- A.Red * (1 - value) + B.Red * value  
4.   toReturn.Green <- A.Green * (1 - value) + B.Green * value  
5.   toReturn.Blue <- A.Blue * (1 - value) + B.Blue * value  
6.   toReturn.Alpha <- A.Alpha * (1 - value) + B.Alpha * value  
7.   return toReturn
```

Figure 21. The “Interpolate Color” algorithm.

The “Interpolate Color” function simply returns a new Color object based on the two Colors (A and B) that are passed and the value, in the range [0, 1], that is passed. If the value is 0, then Color A is returned, and likewise, when the value is 1, then Color B is returned. Any value between 0 and 1 returns a proportional combination of the RGBA components of A and B.

```

SnapColors(Region region) returns nothing
1.   for each Point, p, in region
2.       if p.color = "null"
3.           Create "bfsQueue", an initially empty queue of Points
4.           Create "visited", an initially empty Hash Table of Points
5.           bfsQueue.Enqueue(p)
6.           while bfsQueue.Count > 0
7.               currentPoint <- bfsQueue.Dequeue()
8.               visited.Add(currentPoint)
9.               if region.texture[currentPoint.x, currentPoint.y]
                  .color != "null"
10.                  for each Point, q, in visited
11.                      region.texture[q.x, q.y].color
                        = region.texture[currentPoint.x, currentPoint.y]
                        .color
12.                  end for
13.                  break out of while loop
14.              end if
15.              for each Point, q, in
                  {(currentPoint.x - 1, currentPoint.y),
                   (currentPoint.x + 1, currentPoint.y),
                   (currentPoint.x, currentPoint.x - 1),
                   (currentPoint.x, currentPoint.x + 1)}
16.                  if region.Contains(q) and !visited.Contains(q)
17.                      bfsQueue.Enqueue(q)
18.                  end if
19.              end for
20.          end while
21.      end if
22.  end for

```

Figure 22. The “Snap Colors” algorithm.

The “Snap Colors” function is used to assign a Color value to all Pixels in a Region which have a “null” Color. Before the Gradient application is applied, regardless of shape, the Color of all Pixels in the Region is set to “null”. The application of the Gradient is based on the shape of the Gradient, and it is possible that the shape of the Gradient does not fully contain the shape of the Region, thus leaving some Pixels in the Region with “null” Colors, as they have not yet been set. This function sets those Colors.

The function works by iterating through all of the Pixels in the specified Region, and if the Color of a Pixel is “null”, then it applies a Breadth-First Traversal from that Pixel until it finds a Pixel whose Color is not “null”. Once finding such a Pixel, it will set the color of the original Pixel that began the BFT, as well as all other Pixels that were

visited in the BFT (and thus also have “null” Colors), to the Color of that last Pixel found. This “snaps” all “null” colored Pixels in a Region to be the Color of the nearest Pixel which is not null. Note that setting all of the colors in the visited list to be the same is not necessary for correctness, but it makes the algorithm run in $O(k)$ time in the worst-case instead of $O(k^2)$, where k is the number of “null” colored Pixels originally, due to the fact that a BFT will not have to be performed for every “null” colored Pixel in the worst-case.

Gradient Detection

This section describes how to detect and represent the Gradients in an input image, before the image is encoded. This uses the Region Detection algorithms described in Figure 8 to first find the Regions, and it then uses a variance-based Color difference comparison to analyze the regions to see if it is a Gradient for one of the three predefined shapes. This is extendable to any number of predefined Gradient shapes, but this paper only examines line, mirror, and circle Gradients. The algorithms to detect and verify Gradients, as well as use the Region concatenation algorithm given in Figure 14 with Gradients, are detailed in Figure 23.

The first algorithm that will be described is the process of getting the Gradient of a Region, which is an object which contains a “primary” and “secondary” Color object, a “primary” and “secondary” Point object, and a “shape” identifier. This function will return the Gradient, if a Gradient should be returned, and otherwise it will return a Color, which represents that solid, unvarying color of the region.

```

GetGradient(Region region) returns object (Color or Gradient)
1.   temp <- GetSolidColor(region, false)
2.   if temp != "null" then return temp
3.   startPoint <- GetMiddlePoint(region)
4.   highBestPoint <- GetLastPoint(region, startPoint, true)
5.   lowBestPoint <- GetLastPoint(region, startPoint, false)
6.   Create "forcedGradient", a new Gradient object with
      primaryPoint = lowBestPoint,
      primaryColor = region.texture[lowBestPoint.x,
      lowBestPoint.y].color,
      secondaryPoint = highBestPoint,
      secondaryColor = region.texture[highBestPoint.x,
      highBestPoint.y].color,
      shape = "null".
7.   GetGradientShape(region, forcedGradient)
8.   if forcedGradient.shape != "null"
9.       ApplyGradient(region, forcedGradient)
10.      return forcedGradient
11.  end if
12.  return GetSolidColor(region, true)

```

Figure 23. The “Get Gradient” algorithm.

This function is used to determine the Color to be assigned to a Region once its Pixels have already been found. This makes use of several additional functions, which will be described in Figure 24, Figure 25, Figure 26, and Figure 27. This algorithm first checks to see if the input Region is clearly not a Gradient (i.e., a solid color), and if not, it returns the solid Color. If this check does not determine the Region to be of a solid Color, then it might be a Gradient. It then finds the middle Point of the Region, and from this Point, a search for the Pixel with the highest RGBA average value is performed, as well as a search for the Pixel with the lowest RGBA value. Once these values are found, a Gradient object (the “forced Gradient”) is verified, and if the Gradient verification is successful, this returns the Gradient, and otherwise, the function returns the Color representing the solid Color of the Region, based on the mode Color.

Note that this function has two checks in place to verify that the Gradient is correct and to optimize the applied (not asymptotic) running time. The first is the initial call to the “Get Solid Color” function, which will check if the variance of the actual Pixel

Colors is low enough for the Region to be considered a solid Color, thereby avoiding any other Gradient checks that might need to be performed. This first verification is not necessary for correctness, but it reduces the running time by identifying Regions which are certainly not Gradients before Gradient checks are performed. The second verification happens during the “Get Gradient Shape” function, which attempts to match up the Gradient that was found with one of the three predefined shapes. If it does not match one of the shapes well enough, this indicates that the Region should not be a Gradient.

The “Get Solid Color” function, which takes as a parameter a Pixel-populated Region and a Boolean indicating whether or not to always return a solid Color, is described below.

```
GetSolidColor(Region region, Boolean returnColor) returns Color  
1.   Create an initially empty Linked List of Colors, "colors".  
2.   Fill colors with the Color of each Pixel in region, even if the  
     Color has already been added to colors.  
3.   Set "average" to be the average color from "colors", by  
     averaging the RGBA vector components.  
4.   if returnColor is false  
5.       Set "variance" as the variance of the Colors in "colors"  
         from "average", compared by the sum of RGBA vector  
         components.  
6.       if variance > MaximumSolidColorVariance then return "null"  
7.   end if  
8.   Return the mode Color from colors.
```

Figure 24. The “Get Solid Color” algorithm.

The “Get Solid Color” function simply compares the color variance to some Maximum Solid Color Variance, where the recommended value of this parameter is 25. This will ensure that regions that are already solid Colors (thus having a variance of 0) will not have Gradient checks performed. The value of 25 accounts for some noise or artifacts, but not enough to be considered a Gradient. If “return Color” is true, then this will always return a solid Color, but if “return Color” is false, this may return a solid

Color if the Region is not a Gradient or this might return “null” if the possibility of a Gradient cannot be excluded.

The next algorithm is the “Get Middle Point” function, which takes as a parameter a Pixel-populated Region and returns a Point (x-y integer pair) corresponding to the middle Point of the Region.

```
GetMiddlePoint(Region region) returns Point  
1.   Create “firstPoint”, a new Point with  
      x = region.boundingBox.Left + region.boundingBox.Width / 2,  
      y = region.boundingBox.Top + region.boundingBox.Height / 2  
2.   Apply a Breadth-First Search from “firstPoint” outward,  
      stopping when a Point is found that is inside the region (which  
      could be “firstPoint”).  
3.   Return the first Point found that is within the region.
```

Figure 25. The “Get Middle Point” algorithm.

The “Get Middle Point” function is an algorithm which finds the Point belonging to the Region which is closest to the center of the Region, as determined by the Region’s AABB center. Specifically, it accomplishes this by calculating the AABB center, and applying a Breadth-First Search from that point until it finds a Point which is within the Region, thereafter returning that Point. The BFS is necessary, as opposed to simply returning the AABB center, in order to account for “donut-shaped” regions which might have holes where the AABB center is located.

Next, the “Get Last Point” algorithm is described, which takes as parameters the Region, the starting Point, and a Boolean indicating whether or not to look in the ascending direction. This returns a Point, corresponding to the Pixel in the Region which is one of the two endpoints of the Gradient.

```

GetLastPoint(Region region, Point startPoint, Boolean ascending)
    returns Point
1.   farthestMetric <- -1
2.   farthestPoint <- "null"
3.   startDark <- GetNormalizedDarkness(region, startPoint)
4.   multiplier <- ascending ? 1 : -1
5.   Create "bfsQueue", an initially empty Queue of Points.
6.   bfsQueue.Enqueue(startPoint)
7.   Create "visited", an initially empty Hash Table of Points.
8.   visited.Add(startPoint)
9.   while bfsQueue.Count > 0
10.    currentPoint <- bfsQueue.Dequeue()
11.    Create "toAdd", an initially empty Linked List of Points.
12.    for each Point, p, in
        {(currentPoint.x - 1, currentPoint.y),
         (currentPoint.x + 1, currentPoint.y),
         (currentPoint.x, currentPoint.x - 1),
         (currentPoint.x, currentPoint.x + 1)}
13.        if !visited.Contains(p) and region.Contains(p) and
            ColorInRange(region, currentPoint, p, ascending)
14.            toAdd.Add(p)
15.            visited.Add(p)
16.        end if
17.    end for
18.    currentDark <- GetNormalizedDarkness(region, currentPoint)
19.    if multiplier * startDark <= multiplier * currentDark
20.        colorDist <- Abs(startDark - currentDark)
21.        euclidDist <- Distance(startPoint, currentPoint)
22.        metric <- colorDist - 0.5 * euclidDist
23.        if metric > farthestMetric
24.            farthestMetric <- metric
25.            farthestPoint <- currentPoint
26.        end if
27.    end if
28.    Sort "toAdd" by the color darkness corresponding to the
        Pixel color in the region. Do this in ascending order if
        "ascending" is true, and in descending order otherwise.
29.    Add each Point in "toAdd" to "bfsQueue" in order.
30. end while
31. return farthestPoint

```

Figure 26. The “Get Last Point” algorithm.

This algorithm traverses the Region from the center Point of the region in the direction of the perceived Gradient. It does this by modifying the Breadth-First Traversal algorithm in three ways. The first change is that new Points are only added to the BFS queue if they are believed to contribute to the Gradient, which is determined by the

“Color in Range” function (described in Figure 28). The second change is that instead of adding the neighbors to the BFS queue in any order, the Points to be added are first sorted by the corresponding color darkness, such that the Points which are more likely to lead to the endpoint of the Gradient are added first. This is done in order to find the path from the center that most quickly leads to the end Point of the Gradient, so as to reduce the likelihood that noise or artifacts will incorrectly contribute to the Gradient detection. The final change to the BFS algorithm is that a comparison metric is used to compare the Color distance from the current Point in the traversal to the starting Point, as well as the Euclidean Distance between the two Points. The metric is designed to give the darker Colors more weight, as long as they are within a close enough distance to the center as the next-darkest Color. For example, if the Euclidean distance is not taken into account, then the end point of the Gradient may be mistakenly identified as a noisy Pixel that happens to be darker, but is farther away from the Gradient that is being detected; taking the Euclidean distance into account reduces this effect.

The “Get Last Point” algorithm is intended to be executed twice—once to find the darkest end of the Gradient and one to find the lightest end. This is what the “ascending” Boolean corresponds to, and this affects the way that Colors are compared and the way that the Points to be added to the BFS queue are sorted. When Colors are compared in this function, the darkness values (closeness to the RGBA value of {0, 0, 0, 0}) are compared. It is not enough, however, to simply compare the Color darkness values of two Pixels, as the input image might be noisy or have artifacts that alter what the true value of the darkness should be, and therefore simply comparing two Pixel Colors by darkness does not always work. Thus, a buffer is used, wherein the average Color darkness of this

buffer, centered at the target Pixel, is used instead of simply the Color darkness of the target Pixel. This is the purpose of the “Get Normalized Darkness” function, which is given below.

```
GetNormalizedDarkness (Region region, Point point) returns float  
1.   radius <- 6  
2.   Create a Linked List of Points, "buffer", containing all Points  
     within "radius" distance of "point".  
3.   Return the average darkness (the RGBA vector distance of a  
     color to {0, 0, 0, 0}) of the Colors associate with the Points  
     in "buffer", found by indexing them in "region". Do not  
     consider Points in "adjacent" which are not in "region".
```

Figure 27. The “Get Normalized Darkness” algorithm.

The buffer used by “Get Normalized Darkness” is a circular buffer. The purpose of this is to reduce the effect of noise and artifacts in the original image contributing to the darkness value of a Pixel. Note that only the Points in the buffer that are contained in the region are used for the darkness average. The recommended radius is 6, but this can be altered, if necessary, to improve the look of the Gradient.

In addition to using a buffer to compare the darkness of two Colors, when Points are examined to see if they should be added to the BFS queue, they are compared by using a minimum bound for which Colors can be lighter than the Color at the current Point when the BFS executes. Ideally, when searching for the endpoint of a Gradient that is the darkest, only Pixels which are darker than the Pixel currently being visited should be added, but due to noise, there should be some windowed lower-bound. This is what the “Color in Range” function does, and it is shown below.

```

ColorInRange(Region region, Point A, Point B, Boolean ascending)
    returns Boolean
1.    aDarkness <- GetNormalizedDarkness(region, A)
2.    bDarkness <- GetNormalizedDarkness(region, B)
3.    if ascending then return bDarkness >= aDarkness - LowerBound
4.    else return bDarkness <= aDarkness + LowerBound

```

Figure 28. The “Color in Range” algorithm.

The Lower Bound is a configurable setting, which can be changed depending on the noise. If this value is too low, then it might be possible that noise prevents the Gradient from being detected to be as large as it truly should be, but if this value is too large, then the Gradient might be too large, as additional noise may be thought to be part of the Gradient. The recommended value for this Lower Bound is 4.

The last function to be described that is used by the “Get Gradient” function is the “Get Gradient Shape” function, which takes as a parameter the Pixel-populated Region and the Gradient object that is suspected of fitting the Region, which has the shape as “null”. This does not return anything, but it will set the shape of the Gradient that is passed if it is determined to fit, and leaves the shape as “null” otherwise. The “Get Gradient Shape” function has two purposes—to verify that the passed Gradient fits one of the predefined shapes and to find the shape that best fits the Gradient, if it does fit one of the shapes. The function works by applying the gradient for each shape to a copy of the Region, and then taking the shape which resulted in the lowest variance between the Pixel Colors in the two regions. The algorithm is given in Figure 29.

```

GetGradientShape(Region region, Gradient gradient) returns nothing
1.  bestVariance <- infinity
2.  bestShape <- -1
3.  shapes <- {"circle", "reversedCircle", "mirror",
    "reversedMirror", "line"}
4.  for i = 0 to 4
5.      regionCopy <- Copy(region)
6.      gradient.shape <- shapes[i]
7.      ApplyGradient(regionCopy, gradient)
8.      Let "variance" be the variance between the Pixel Colors of
        "region" and the Pixel Colors of "regionCopy".
9.      if variance < bestVariance
10.         bestVariance <- variance
11.         bestShape <- i
12.     end if
13. end for
14. if bestVariance < MaximumGradientVariance and bestColor != -1
15.     if shapes[bestShape] is "reversed", then swap the primary
        and secondary Point and primary and secondary Color of
        gradient
16.     gradient.shape <- shapes[bestShape]
17. else
18.     gradient.shape <- "null"
19. end if

```

Figure 29. The “Get Gradient Shape” algorithm.

The “Get Gradient Shape” function finds the shape, out of the three predefined shapes, by simply applying the Gradient to a copy of the Region and finding the variance of the Colors of the Pixels that make up the Regions (where the Region and the copy have the exact same number of Pixels and Pixel locations, even if the Colors are different). The Maximum Gradient Variance is used as an additional verification that the Gradient Passed to the function should be a Gradient, because if the best variance is higher than the maximum allowable variance then the Region should be considered to have a solid Color instead of a Gradient. The recommended value to use for the Maximum Gradient Variance is 600.

The algorithms that have thus far been described for detecting Gradients deal only with Regions that have already been found and filled with the correct Pixels from the input image. To find these Regions before the Gradient detection occurs, simply apply

the algorithm described in the Cartoon Image Region Detection section, using the nearest-neighbor Pixel color value as the metric for comparison, rather than the Region average color, so as to absorb the full Region with a possible Gradient instead of posterized segments. Because the nearest-neighbor approach is unwise with noisy images, but is the only way to absorb Gradient-based Regions, and because an image might exist with both Gradient and non-Gradient Regions, it is necessary to run the Region Detection algorithm twice—first with nearest-neighbor based Color comparison, and second with Region Color average based Color comparison. When the Region Concatenation takes place when the number of unique Colors is being reduced, Regions which contain a Gradient should not be removed. Furthermore, after a Gradient is detected, the Colors of the Pixels in that Region should be changed to those corresponding to the Gradient, so that when the Region Concatenation occurs, the Pixels which are later absorbed by the Region can be made the same as the Color of the closest Pixel belonging to the Gradient, creating a smoother appearance than simply setting the absorbed Pixels to be the average Color of the Gradient.

Data Compression

Idea

The point of the image processing that has thus far been described is to format an input image in a manner that greatly reduces the amount of space that the image takes up when stored in a computer. Before the image compression takes place, the pre-processing of the input image must take place, wherein all distinct Regions are detected, each having

a Color or Gradient component. The idea is to compactly encode the location, shape, and Color or Gradient of each Region, thereby fully representing the cartoon image.

Implementation

The implementation of the data compression makes use of a few resources, the first of which being Huffman Encoding. The basics of Huffman Encoding are described in the Background section, but in order to get the values used to create the Huffman Tree, the image must be processed twice. The first time is to get the list of values (non-negative integers) to be encoded by the Huffman Encoder, and the second pass is to use the Huffman Tree (which is generated immediately after the first pass) to encode the image. Rather than do computations a second time, an ordered Linked List of not necessarily unique, non-negative integer values should be created before the first pass, and a value should be added to this whenever it would be encoded using the Huffman Encoder (the encoding scheme is given in Table 1). This Linked List can then be used to create the Huffman Tree, and in addition, by keeping track of a pointer in this list, on the second pass of the data to be encoded, calculations will not need to be redundantly performed, as this list can simply be accessed where appropriate.

An additional resource that must be used for the file encoding is Binary Run-Length Encoding, which is also described in the Background Section. Binary RLE is used here for compression of Region data. Because a Region corresponds to a single Color or Gradient, Binary RLE can be used to represent the shape of the region by letting “1” represent a Pixel that belongs to the Region and “0” be any other Pixel. Before the values from the Binary RLE Encoding (the “RLE Stream”) can be found, the Axis-Aligned Bounding Box of the Region must be found. This “AABB” is calculated by finding the

minimum and maximum X value from all the Pixels in the Region and the minimum and maximum Y value from all the Pixels in the Region, and these represent the left, right, top, and bottom, respectively, of the AABB. The AABB of each Region should be pre-computed and stored in the Region structure before any image encoding takes place.

Using the AABB of a Region, the RLE Stream is found by iterating from the top-left Pixel in the AABB from left to right, then from top to bottom (similar to reading English), moving down a row when the last Pixel from the previous row is read. The algorithm to get the RLE Stream for a Region is given in Figure 30.

```
GetRLEStream(Region region) returns Linked List of integers
1.   Create the empty Linked List of integers, "toReturn"
2.   currentBit <- region.Contains[region.AABB.Top,
    region.AABB.Bottom]
3.   tally <- 0
4.   for y = region.AABB.Top to region.AABB.Bottom
5.       for x = region.AABB.Left to region.AABB.Right
6.           if region.Contains[x, y] != currentBit
7.               currentBit <- !currentBit
8.               toReturn.Add(tally)
9.               tally <- 0
10.          end if
11.          tally <- tally + 1
12.        end for
13.    end for
14.    toReturn.Add(tally)
15.    return toReturn
```

Figure 30. The “Get RLE Stream” algorithm.

Note that the Region that is passed to this function is assumed to have already undergone the processing described in the Cartoon Image Region Detection section and Color Reduction section, such that its color information and all Pixels belonging to it have been set. Note also that this returns every value in the Binary RLE that represents the region, although strictly speaking, the final value (added at line 14) is not needed, as if the number of values in the stream is known then the final value can be determined from the number of values and the area of the AABB.

The algorithm for filling a Region's Pixels based on an input RLE Stream is given in Figure 31.

```
FillRegion(AABB boundingBox, Boolean firstBit, Linked List of  
integers, "values") returns Region  
1.   Create a new Region object, "toReturn", with no Pixel values  
    set.  
2.   index <- 0  
3.   activeBit <- firstBit  
4.   switchValue <- values[index++]  
5.   streamCount <- 0  
6.   for y = boundingBox.Top to boundingBox.Bottom  
7.       for x = boundingBox.Left to boundingBox.Right  
8.           if activeBit  
9.               toReturn.Add(x, y)  
10.          end if  
11.          streamCount <- streamCount + 1  
12.          if streamCount >= switchValue  
13.              streamCount <- 0  
14.              activeBit <- !activeBit  
15.              switchValue <- values[index++]  
16.          end if  
17.      end for  
18.  end for
```

Figure 31. The "Fill Region" algorithm.

Note that before this algorithm is used, the RLE Stream and the first bit of that stream, as well as the AABB for the Region, must be read from the data file. These are stored before the values of the RLE Stream. The Region that is returned from this algorithm has all of the Pixel locations correctly stored, but it still needs to have its Color value set. This is also stored in the data file before the RLE Stream, and simply needs to be specified for the Region that is returned before the Texture can be rendered.

The final resource used by the encoding and decoding process is the use of a Color Palette. A Color Palette is a collection of unique Colors, where each color is represented by a 32 bit representation of its RGBA value (where each component takes on a value in [0, 255]). Because it is possible (in fact, likely) that at least two Regions share the same Color, all Colors are added to the Color Palette, and when encoded, the

index into the Color Palette is used, rather than the 32 bit RGBA value each time. This will likely save space in the encoded file. The Color Palette should be pre-computed before the encoding of the file by analyzing the Color of each Region (including both of the Colors of a Gradient), and adding the Color to an initially empty Linked List of RGBA values if it is not already in the list. This list is encoded near the beginning of the encoded file (see Table 1), and Colors are thereafter encoded by the index in this Linked List.

File Structure

This section details the exact file structure that is used to encode the image. Because most integer values are encoded using the Huffman Tree, which is generated by pre-processing the data, the exact bit lengths cannot be given for these values, as the bit lengths will vary due to the Huffman Encoding; “Huffman” is written in the table to indicate that the value should be pre-processed and encoded using the Huffman Tree. The bit lengths, where fixed, are given.

Table 1. The complete file structure.

Bits	Purpose
Varies	Encoded Huffman Tree, as described in the Background section.
Huffman	Image Width
Huffman	Image Height
Huffman	Number of colors in the color palette - 1, $n - 1$
32	Color 0
...	(And so forth for all n colors.)
Huffman	Number of gradient colors, m
Huffman	Palette index for the first color of gradient color 0
Huffman	Palette index for the second color of gradient color 0
Huffman	The X value of the first point in gradient color 0
Huffman	The Y value of the first point in gradient color 0

Huffman	The X value of the second point in gradient color 0
Huffman	The Y value of the second point in gradient color 0
2	The shape type for gradient color 0. Line = 00, Circle = 01, Mirror = 10.
...	(And so forth for all m gradient colors.)
Huffman	Number of Regions in the image - 1, $k - 1$
Huffman	Color palette index of the color for Region 0
Huffman	AABB Left value for Region 0
Huffman	AABB Top value for Region 0
Huffman	AABB Width for Region 0
Huffman	AABB Height value for Region 0
1	Leading bit of the RLE stream for the region.
Huffman	Number of values in the RLE stream, $s - 1$
Huffman	Stream value 0
...	(And so forth for only the first $s - 1$ stream values. The last value is not included.)
...	(And so forth for all regions.)

Before the image is actually encoded in this format, the numbers which are to be processed with a Huffman encoding must first be analyzed so that the frequencies can be accessed and the Huffman Tree can be generated. Thus, the entire encoding process needs to happen twice, and so the values should be stored the first time through so that the second pass can be made more efficient.

Additionally, when a Region decodes the RLE Stream, it does not need to know the last value in the completed stream. This is because, after the second-to-last value has been read, because the area of the AABB of the Region is known, as well as the number of stream values, the Pixels that would have been covered by this last stream value can be determined easily, and it is therefore redundant and a waste of space to store the last value of the RLE stream.

The decoding of this image is as simple as following this format and applying the RLE Streams to the appropriate regions at the appropriate location, as they are read from

the file. Once the image is decoded, it will appear to have “jaggy” edges due to the fact that a lossy process may have been used to encode the image. Smoothing of the image should be applied to reduce this effect, and the recommended procedure to do so is described in the Image Reconstruction via Low-Pass Filtering section.

Image Reconstruction via Low-Pass Filtering

Before an image is encoded with the proposed method, its visually distinct color regions are identified, and each Pixel of the image is set to belong to one of these regions. This process greatly reduces the size of the encoded file, but will result in a lossy image, due to the fact that all Colors in a Region, which are visually the same color, are not actually the same Color, according to RGBA values. This tends to clean-up images that suffer from JPEG artifacts or noise quite nicely, but it undoes any attempts that were made to anti-alias the lines and shapes of the original image. Figure 32 shows an image in which anti-aliasing was used to smooth the lines, creating a more visually pleasing image, and the same image after it has been filtered into visually distinct color regions.

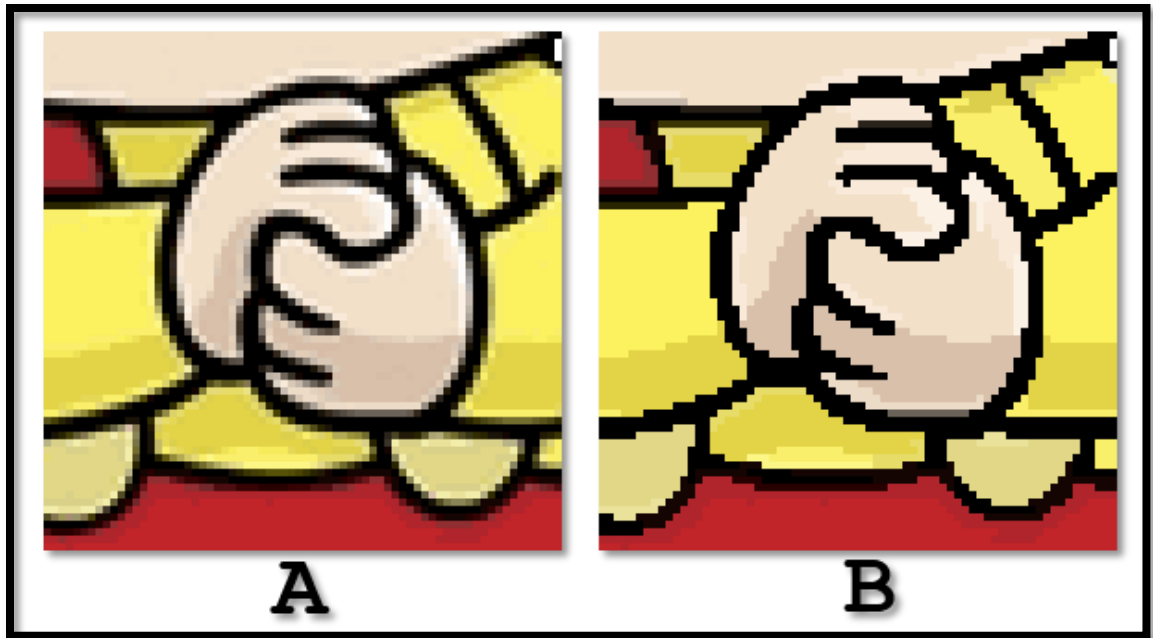


Figure 32. This shows an input image (A), which used anti-aliasing to create smooth, visually appealing shapes and edges, and the resulting image (B) when all of the encoding algorithm are applied, resulting in jagged edges. Images scaled 400% to show Pixel detail.

It is clear that there exists a presence of jagged edges, or “jaggies”, on the image due to the smooth gradients created by the anti-aliasing being stripped away. This section describes a method used by the decoder of the image to restore this smooth, anti-aliased look. The procedure uses a combination of edge detection and iterative low-pass filtering.

Idea

The objective is to restore the decoded image to be as smooth as the original image was, but because the Region shapes do not contain the line data without performing polygonal shape analysis, anti-aliasing cannot be applied directly. Instead, an iterative low-pass filter is used to smooth the image, and the effects are constrained only to the pixels between regions.

Detecting the Edges between Regions

Although high-pass filtering is often used for edge detection in general images, for cartoon images where the color regions are known, edge detection is simpler. The process works as follows. First, create a 2-D Boolean (bit) array, a “bit mask”, with the same width and height as the image, and set all values initially to 0. For each Region in the Texture, apply a Breath-First Traversal of the Pixels, starting from any Pixel belonging in the region. If a Pixel is visited and it is determined that its right neighboring Pixel or lower neighboring Pixel do not belong to the same Region (if they exist), then set the corresponding bit in the bit mask to true at the index corresponding to the right or lower neighboring Pixel. Once this process has been completed for all Pixels of all Regions, the bit mask contains the outlines of the Region, and this determines what Pixels will be altered in the low-pass filtering.

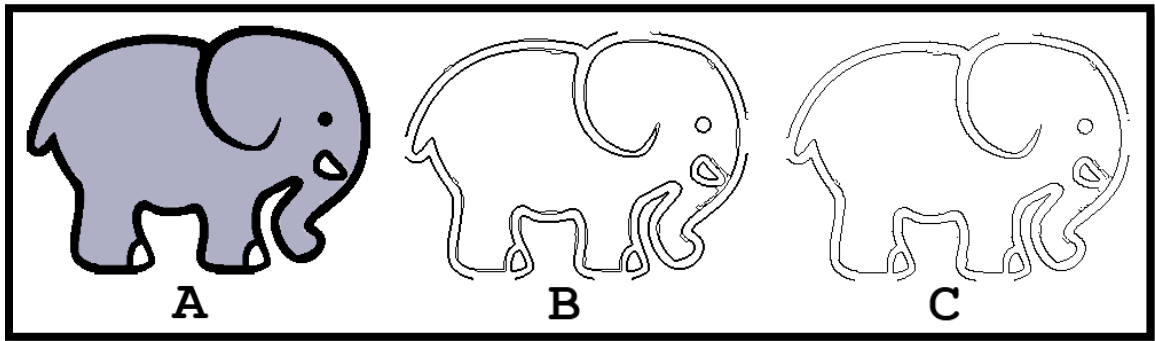


Figure 33. This shows the rigid image that was encoded after undergoing Region expansion and Color reduction (A), the 2-Pixel thick outline found by using all 4 orthogonally adjacent neighbors in the Breadth-First Traversal when identifying a Region boundary (B), and the 1-Pixel thick outline created by simply using only the lower and right adjacent neighbors in the BFT when identifying a Region boundary (C).

Note that only the right adjacent Pixel and bottom adjacent Pixel that do not belong to the same Region are used so that the outline of the Regions are only 1 Pixel thick. If the top, bottom, left, and right adjacent Pixels of different Regions are each used, then when one Region draws its outline, and then the adjacent region draw its outline, the

outline will accumulate a thickness of two Pixels, as seen in Figure 33. However, by using the right and the bottom Pixels only, this ensures that the thickness of the boundary line is only 1, as when the other Region's outline is found, these Pixels will correspond to the left and top, respectively, and therefore will not be set to true in the bit mask. Furthermore, this does not give the image a "skew" in the right-down direction because the low-pass filter, which is described in the Low-Pass Filtering section, applies an averaging of the color, and because it is impossible for two regions to meet at the borders of an image, this process is direction invariant. Thus, the fact that the right and bottom Pixels are used for expansion is irrelevant, and so this could be any combination of left, right, top, or bottom as long as one horizontally and one vertically adjacent Pixel is used.

Low-Pass Filtering

Traditionally, low-pass filtering is the process of altering a Pixel's Color based on the Pixels that surround it. Different functions can be used to apply these weights, such as weighted averaging, a Normal distribution, or a sinc function. Additionally, different shapes could be used to determine the neighboring Pixels that are used in this function; for example, only the top, bottom, left, or right Pixels might be used with averaging, or perhaps all pixels within a certain radius might be used with a smooth weighting function. Low-pass filtering techniques are described in detail in [5].

The low-pass filtering that is used for restoring the images that were encoded with the scheme presented in this thesis simply takes the average of the four orthogonally neighboring Pixels (above, below, and to the left and right). If the Pixel is at an edge of the image, the weights are distributed to the neighboring Pixels which are contained within the image.

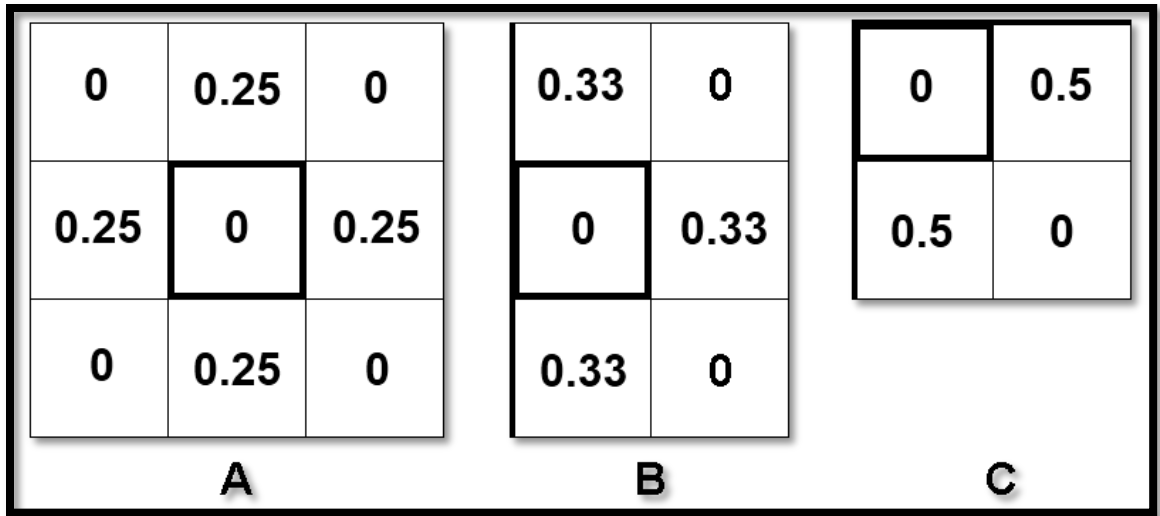


Figure 34. This image shows the weight distributes for the adjacent Pixels of pixel currently being filtered (the highlighted square). This shows the case where the filter Pixel is not along the border of the image (A), where it is along the border of the image but not at a corner (B), and at a corner (C).

For a single iteration of low-pass filtering, the process works as follows. First, make a copy of the input image, as the overwriting of Pixel Colors should not affect the processing of other Pixels in the same iteration. Using this copy, iterate through every Pixel, in any order, and using the appropriate function and neighboring Pixels, calculate what the new color of the current Pixel should be, and then set the corresponding Pixel in the actual image to be that color. The copy of the image should not be altered, and it can be deleted once the iteration has completed.

Applying low-pass filtering on the image as a whole, especially for several iterations, tends to make the image seem blurred and out of focus. See Figure 35-D. This is why, for the purposes of reconstructing the original input image that was encoded using the algorithms discussed in this paper, only the Pixels which are contained in the bit mask defining the edges of Regions (as described in the Detecting the Edges between Regions section) are altered. This keeps the image looking sharp, while at the same time,

allowing multiple iterations of the low-pass filtering to process to achieve a higher level of smoothness.

The filtered image, although visually smoother than the encoded image, is still not as smooth as the original. This is due to the fact that the low-pass filtering uses all orthogonal neighbor Pixels with equal weight, and after a few iterations, this tends to only extend the jagged edges of a region boundary. This is because long and narrow strips of bits in the bit mask give much more weight to the Pixels along the longer edge of this strip, and much less weight to Pixels along the narrow edge. To achieve results with this low-pass filter, the applied weights must be made a function of the width and height of these “strips” in the bit mask.

Before the width and height of a strip in the bit mask can be used in the weight calculation, the length and width must be determined. Suppose the Pixel at (x, y) is having the filter applied to it. It must be the case that the bit mask is “1” at the point (x, y) , because otherwise the filter would not be applied at this point, and so the width and height is at least 1, always. Find the width by traversing the bit mask at $(x-1, y)$, $(x-2, y)$, and so on until a bit in the bit mask is 0 or the edge of the image is reached, and count the number of bits that were “1” while doing so. Repeat for $(x+1, y)$, $(x+2, y)$, and so on in the same fashion, again counting the number of bits that were “1”. Add these two sums to 1 (for the Pixel at (x, y)), and this is the width of the strip. The height of the strip is found the same way, only traversing up and down rather than left and right.

At the point of applying a filter to a particular Pixel, once the corresponding width and height of the strip that it belongs to is found, use the width and height directly to scale the relative weights of the adjacent Pixels. For instance, in the case where the Pixel

is not at the edge of the image, if the width of the strip is 7 and the height is 1, the weight of the left (and of the right) neighbor would be $\frac{7}{(7+7+1+1)} = 0.4375$, and likewise the weight of the top (and bottom) neighbor would be $\frac{1}{(7+7+1+1)} = 0.0625$. Likewise, if the Pixel is at the edge of an image, these weights are simply skewed to not take Pixels which do not exist in the Texture into account. Finally, the low-pass filtering will iteratively run until the number of the current iteration, starting at iteration 0, is larger than the maximum of the widths and heights found in the image. When applying the averaging to a single Pixel, if the iteration number is larger than the maximum of the width and height of the strip that the Pixel belongs to, then do not apply the filtering. This keeps Pixels near short strips from being effected while longer strips are still being processed. The effects of this modification are shown below.

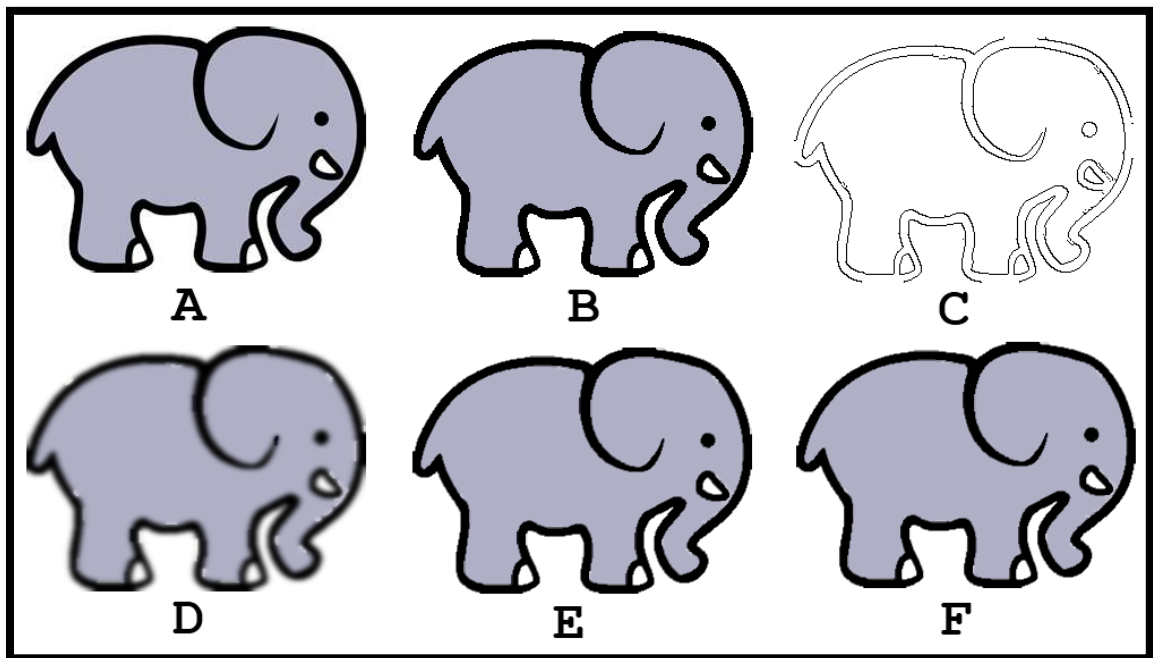


Figure 35. This shows the result of encoding an input image with lossy JPEG noise (A) to get a rigid image (B), which is encoded. The decoding process finds the 1-pixel thick outline (C) of B for filtering. The low-pass filter is applied to the image resulting in D, but only the pixels that lay in C are used to create both E and F from B. The width and height proportions were taken into account when creating F, but not E.

Results

This section compares the set of test images used for this research with the compression algorithms used for the PNG, JPEG, Quad-Tree Decomposition, and the Proposed Method. All image sizes are in bits per pixel, to provide a normalized metric for comparison between test images of different pixel areas. The Quad-Tree file was generated by using the approach proposed in [15], without the use of LZW encoding, and the file size for this is compared with and without the Color reduction proposed in this paper. In addition, the compression and decompression times are given for quad-trees and the proposed method.

Table 2. Results for Test Image 1.


Test Image 1 264x300 Pixels Originally JPEG				
PNG	Full-Quality JPEG	Quad-Tree without Color Reduction	Quad-Tree with Color Reduction	Proposed
5.989 bpp	3.765 bpp	6.749 bpp	0.826 bpp	0.808 bpp
Compression Time		1.94 sec	6.58 sec	6.50 sec
Decompression Time		0.46 sec	0.43 sec	1.70 sec

Table 3. Results for Test Image 2.

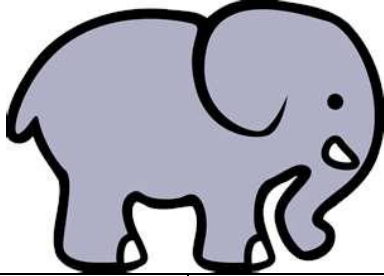
Test Image 2 300x217 Pixels Originally JPEG				
PNG	Full-Quality JPEG	Quad-Tree without Color Reduction	Quad-Tree with Color Reduction	Proposed
5.864 bpp	2.982 bpp	7.477 bpp	0.503 bpp	0.520 bpp
Compression Time		1.68 sec	4.53 sec	4.32 sec
Decompression Time		0.42 sec	0.50 sec	1.21 sec

Table 4. Results for Test Image 3.


Test Image 3 300x293 Pixels Originally JPEG				
PNG	Full-Quality JPEG	Quad-Tree without Color Reduction	Quad-Tree with Color Reduction	Proposed
8.832 bpp	3.905 bpp	13.674 bpp	1.457 bpp	1.080 bpp
Compression Time		3.22 sec	7.90 sec	7.36 sec
Decompression Time		0.53 sec	0.50 sec	1.66 sec

Table 5. Results for Test Image 4.

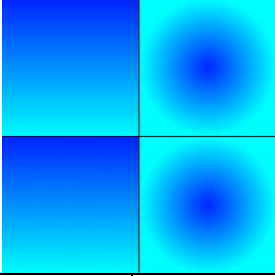
Test Image 4 514x514 Pixels Originally PNG				
PNG	Full-Quality JPEG	Quad-Tree without Color Reduction	Quad-Tree with Color Reduction	Proposed
5.742 bpp	3.442 bpp	9.778 bpp	0.391 bpp, but with severe posterization	0.00987 bpp
Compression Time		2.69 sec	6.76 sec	6.53 sec
Decompression Time		0.87 sec	1.28 sec	3.14 sec

Table 6. Results for Test Image 5.



Test Image 5 478x484 Pixels Originally JPEG				
PNG	Full-Quality JPEG	Quad-Tree without Color Reduction	Quad-Tree with Color Reduction	Proposed
5.653 bpp	3.641 bpp	7.981 bpp	0.763 bpp	0.536 bpp
Compression Time		6.94 sec	9.01 sec	12.33 sec
Decompression Time		1.12 sec	0.90 sec	2.59 sec

Table 7. Results for Test Image 6.

Test Image 6 269x300 Pixels Originally JPEG				
PNG	Full-Quality JPEG	Quad-Tree without Color Reduction	Quad-Tree with Color Reduction	Proposed
7.109 bpp	3.502 bpp	10.804 bpp	1.234 bpp, but with severe posterization	0.867 bpp
Compression Time		2.60 sec	7.48 sec	7.03 sec
Decompression Time		0.49 sec	0.51 sec	1.65 sec

Note that for the Quad-Tree compression, when the Color reduction is applied, Gradients cannot be taken into account. Thus, the resulting image is posterized, leading to higher compression ratios, but noticeable artifacts of several distinct Color Regions rather than one smooth Gradient.

Conclusion and Future Work

From the Results section, it is clear that the proposed method outperforms the PNG image standard, the JPEG image standard at full quality, and the lossless Quad-Tree Compression presented in [15] on all test images. Averaging the bits per pixel on all six test images, the proposed method outperforms all other methods; the proposed method was 13.75 times more compact than the PNG, 7.45 times more compact than the Full-Quality JPEG, 19.82 times more compact than the lossless Quad-Tree Compression, and 1.82 times more compact than the Quad-Tree Compression on the color reduced images.

Note that the color reduction algorithm used to reduce the colors of the images which were compressed with the Quad-Tree is the same that was proposed in this paper. [15] does not suggest that such steps be taken, but instead assumes that the image contains no noise, artifacts, or anti-aliasing, which is an unreasonable assumption based on the poor results demonstrated in the Results section. Thus, the color reduction algorithm proposed in this paper not only leads to a high compression ratio with the proposed compression algorithm, but it also leads to a higher compression ratio for the preexisting Quad-Tree cartoon image compression algorithm. Note that the Quad-Tree cartoon image compression algorithm cannot support Gradients by the nature in which the data is saved, and therefore, all Gradient Detection and application algorithms discussed in this paper are not applicable to the Quad-Tree compression algorithm. Therefore, the only way for Quad-Trees to be used for cartoon images with Gradients is through lossless compression (i.e., with no Color reduction), leading to very low compression ratios, or through posterizing the Gradients into several distinct Regions, which is visually noticeable. Figure 36 shows the posterization of a Gradient image.

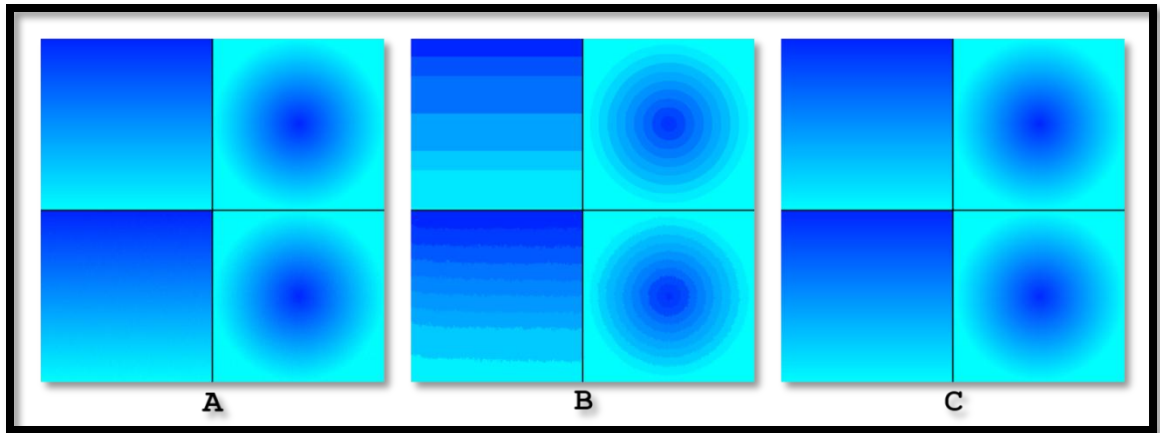


Figure 36. This shows a sample image with two types of Gradient shapes—a line and a circle. The original image (A) is saved in a PNG format, where the top two images are perfect Gradients generated by a mathematical function, and the bottom two images are the same Gradients with JPEG artifacts. Image B shows image A after Color reduction has been applied, but with no Gradient detection. Image C shows image A after both Gradient detection and Color reduction have been applied, and in image C, all artifacts of the JPEG were eliminated by this process, resulting in the bottom two Regions being the exact same as the top two Regions.

Figure 36 shows the result of the Gradient detection and application, which works well when the Gradient has either no noise or a small amount of noise. The Gradient detection algorithm that was described in this paper also works well for large Gradient Regions, even with larger amounts of noise, due to the fact that a large buffer is used for getting a Color's darkness value. Unfortunately, the Gradient detection, as presented in this paper, can have problems with small Regions with a large amount of noise surrounded by other Regions, which may be mistakenly absorbed into the smaller Region. The reason for this is that it is difficult to find a balance between false positives (Regions which should not be determined as a Gradient but are) and false negatives (Region which should be determined as a Gradient but are not). This is caused by the Breadth-First Search approach to finding the highest and lowest color Gradient, and treating these Points as the primary and secondary Points of the Gradient, when the primary and secondary Points should actually have been different. Because of the presence of noise, and because these Points are incorrect, the variance may too high when

there should be a Gradient detected or too low when there should not be a Gradient detected. One way to avoid this would be to use an exhaustive Gradient detection algorithm instead of a greedy Gradient detection algorithm, which would examine all reasonable combinations of Points in a Region until the best Gradient is found. Although this would get better results, it would likely be so slow as to be impractical to use. Improving the Gradient detection for small, noisy Regions is a topic for future research.

An additional topic for future research would be to extend the algorithms presented in this paper to be used in general image compression. The Motivation section described an approach to decompose a generic image, such as a photograph, into Color-based Regions in order to reduce the variability of the Discrete Cosine Transforms involved in the JPEG image compression. By saving the image as a cartoon, and along with that, saving only the DCT between that cartoon and the differences in Colors from the original on a Region-by-Region basis, it may be possible to achieve a more compressed image than current JPEG standards. This is an area for future research and another direct, practical application for the research presented in this paper.

Another area in which future work could be done with this research is the application of edge smoothing presented in [2]. The approach presented in [2] involves performing a case-by-case evaluation on the edges between Color Regions to determine how the resulting, filtered edges should look. According to [2], this approach leads to images with edges that are visually more appealing than low-pass filtering, anti-aliasing, and bi-cubic resampling. This could be applied instead of edge-restricted low-pass filtering to increase the visual quality of the decompressed images.

Finally, an area of future research is to extend this research to cartoon videos. It is the case that frames within a cartoon video are often the same, and if not, many regions between adjacent frames have little to no change in shape. [9] presents similar work dealing with regions which have been encoded in a vector format, rather than with Run-Length Encoding. Future work could involve finding simple algorithms for interpolating between two regions shapes that have been stored with RLE as efficiently as possible, thereby encoding the changes between frames in a cartoon video.

Bibliography

- [1] Bresenham, J. E., "Algorithm for computer control of a digital plotter," *IBM Systems Journal* , vol.4, no.1, pp.25-30, 1965
doi: 10.1147/sj.41.0025
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5388473&isnumber=5388471>
- [2] De Witte, Valérie, et al., "Morphological Image Interpolation To Magnify Images With Sharp Edges," *ICIAR 2006*. LNCS 4141, pp. 381-393, 2006.
- [3] Duce, Dave, "Portable Network Graphics (PNG) Specification (Second Edition)," *W3C Recommendation*, 10 November 2003. URL: <http://www.libpng.org/pub/png/spec/iso/index-object.html#11iCCP>
- [4] Fraenkel, A. S. and Klein, S. T., "Robust Universal Complete Codes for Transmission and Compression," *Discrete Applied Mathematics*, vol.64(1): pp.31-35, 1996.
- [5] Gonzalez, R.C, & Woods, R.E (2002). *Digital image processing (2nd edition)*. Prentice Hall.
- [6] Hamilton, Eric. *JPEG File Interchange Format*. C-Cube Microsystems, 1 Sept. 1992. PDF.
- [7] Huffman, D.A., "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE* , vol.40, no.9, pp.1098-1101, Sept. 1952
doi: 10.1109/JRPROC.1952.273898
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4051119&isnumber=4051100>

- [8] Jian Sun; Nan-Ning Zheng; Hai Tao; Heung-Yeung Shum, "Image hallucination with primal sketch priors," *Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on* , vol.2, no., pp. II-729-36 vol.2, 18-20 June 2003
doi: 10.1109/CVPR.2003.1211539
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1211539&isnumber=27266>
- [9] Koloros, Martin; Zara, Jiri, "Coding of Vectorized Cartoon Video Data", Conference on Computer Graphics 2006. Bratislava: Comenius University, 2006, p. 177-183. ISBN 80-223-2175-3
- [10] Li Zhe-lin; Xia Qin-xiang; Jiang Li-jun; Wang Shi-zi, "Full Color Cartoon Image Lossless Compression Based on Region Segment," *Computer Science and Information Engineering, 2009 WRI World Congress on* , vol.6, no., pp.545-548, March 31 2009-April 2 2009
doi: 10.1109/CSIE.2009.672
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5170759&isnumber=5170643>
- [11] M. Mainberger, et al., "Edge-based compression of cartoon-like images with homogeneous diffusion", *Pattern Recognition* (2010), doi: 10.1016/j.patcog.2010.08.004
- [12] Maleki, A.; Shahram, M.; Carlsson, G., "A near optimal coder for image geometry with adaptive partitioning," *Image Processing, 2008. ICIP 2008. 15th IEEE International Conference on* , vol., no., pp.1061-1064, 12-15 Oct. 2008

doi: 10.1109/ICIP.2008.4711941

URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4711941&isnumber=4711669>

- [13] Morse, B.S.; Schwartzwald, D., "Image magnification using level-set reconstruction," *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on* , vol.1, no., pp. I-333-I-340, vol.1, 2001

doi: 10.1109/CVPR.2001.990494

URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=990494&isnumber=21353>

- [14] Starck, J.-L.; Elad, M.; Donoho, D.L., "Image decomposition via the combination of sparse representations and a variational approach," *Image Processing, IEEE Transactions on* , vol.14, no.10, pp.1570-1582, Oct. 2005

doi: 10.1109/TIP.2005.852206

URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1510691&isnumber=32345>

- [15] Yi-Chen Tsai; Lee, M.-S.; Shen, M.; Kuo, C.-C.J., "A Quad-Tree Decomposition Approach to Cartoon Image Compression," *Multimedia Signal Processing, 2006 IEEE 8th Workshop on* , vol., no., pp.456-460, 3-6 Oct. 2006

doi: 10.1109/MMSP.2006.285350

URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4064600&isnumber=4064505>

- [16] Zhiwei Xiong; Xiaoyan Sun; Feng Wu, "Web cartoon video hallucination," *Image Processing (ICIP), 2009 16th IEEE International Conference on*, vol., no., pp.3941-3944, 7-10 Nov. 2009
doi: 10.1109/ICIP.2009.5414032
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5414032&isnumber=5413332>