Summer 8-21-2006

A hybrid format for storing raster images

Bardia Khalili
University of Colorado Boulder

A Hybrid Format for Storing Raster Images

Submitted to
Dr. Michael G. Main, Chair
Dr. Andrzej Ehrenfeucht
Dr. Roger A. King

Computer Science Department
University of Colorado
UCB 430 Boulder, CO 80309-0430
August 21, 2006

by
Bardia Khalili
University of Colorado
bardiakh@gmx.net

Abstract

Many different formats exist for storing different types of images. Photographs of natural scenes usually are best suited to be stored as Raster images. But drawings are better off if stored as Vector graphics. Some formats can even store both types of images, but none of the existing formats can change itself to adopt a balance between both methods, based on the content of the image.

This project introduces a new format that has a built-in capability to store both Raster images and Vector graphics. Unlike others, this format automatically breaks apart an input image and stores them in appropriate formats; some parts are stored as Raster, others as Vector. This format allows better quality resizing. Also, images stored in this format occupy less disk space in some cases, when compared with other existing formats. Because of its new approach, this format has the potential to be used in applications where other formats may not be able to perform effectively.

Table of Contents

I) Introduction

Every image created with a digital camera or a scanner must be stored according to a specific format. The format for these images can be Jpeg, Tiff or any of the many other available formats, which are generally called "Raster" formats. Every image in Raster format is, in fact, a collection of colored dots arranged in a two dimensional table. Each dot, also known as a "pixel", forms a small fraction of the original image, and is represented by a number called the Color Value. A second method for defining an image is called "Vector Graphics". In this method, instead of using pixels, a series of instructions and commands are utilized to draw the image from scratch. Each method has its own benefits and shortcomings. The Raster method is preferred when it is important to preserve the small details of an image. On the other hand, the Vector method gives better results in case the image should go through a resizing process.

This project aims to devise a new hybrid format, which uses both Raster and Vector schemes. This new method inherits the advantages of both predecessors. The goal is to have an image that its details are well preserved while the results after resizing would also be as good as the results usually obtainable only with Vector format. In this method, the original picture is dismantled into two individual Raster images, which will be processed separately. The first one will contain the elements of the picture with fewer details, such as a plain sky, while the other will have the elements with more details, such as tree leaves. The low detailed image will later be converted into Vector graphics through a process called "tracing". The result of this process, along with the high detailed Raster image, will make up the final hybrid picture.

Figure 1. Separating an image(left) into "low detailed"(middle) and "high detailed"(right)

The following report is intended to describe the different algorithms used to process the image and the theories behind them. Also, the performance of the method with regards to time, picture quality and the file size are analyzed.

II) Theories, Models, and Hypotheses

II-A) Numeric representation of a color

There are many different ways to represent a color with numeric values. A common method is RGB, where a color is represented by 3 separate values for red, green and blue. Each number is usually stored in a byte, which means it can be a value between 0 and 255. By choosing appropriate values, any color can be made. This method is commonly used for displaying images on a computer monitor. In other words, any picture in any other format must be converted to RGB to be suitable for displaying on a computer monitor.

There is another format called YCbCr. Here, instead of 3 values for red, green and blue, we have Y, Cb and Cr, and their values can be calculated using the following formulas:

Y = 0.3 Red + 0.59 Green + 0.11 Blue
Cb = Blue - Y
Cr = Red - Y

The Y value is the brightness, or in fact, a black & white representation of the original color. The Cb and Cr act as color information that, added to the Y, will create the original color. One can conclude that in this method, the black & white and color information sets are separated from each other, which simplifies the independent processing of each one. At first glance, this method does not seem to have any significant advantage over the RGB method. After all, we can simply recalculate the RGB values, using these formulas:

Red = Y + Cr
Blue = Y + Cb
Green = (Y – 0.3 Red – 0.11 Blue) / 0.59

The main advantage of this method originates in how a color image is viewed by the human eyes. The human eye is not able to see color in very small details of a picture. In other words, for small details of a picture, only the black & white information of the picture are important. Even if there are very small details in color, the eye will see them in a color which is the same as that of the surrounding. Consequently, when the small color elements in a picture are not visible, it does not make much difference to the picture if such information is removed from the picture, provided that the small detail *does* appear in the black & white part of the image. Removing unnecessary information is beneficial because it makes the overall picture data smaller.

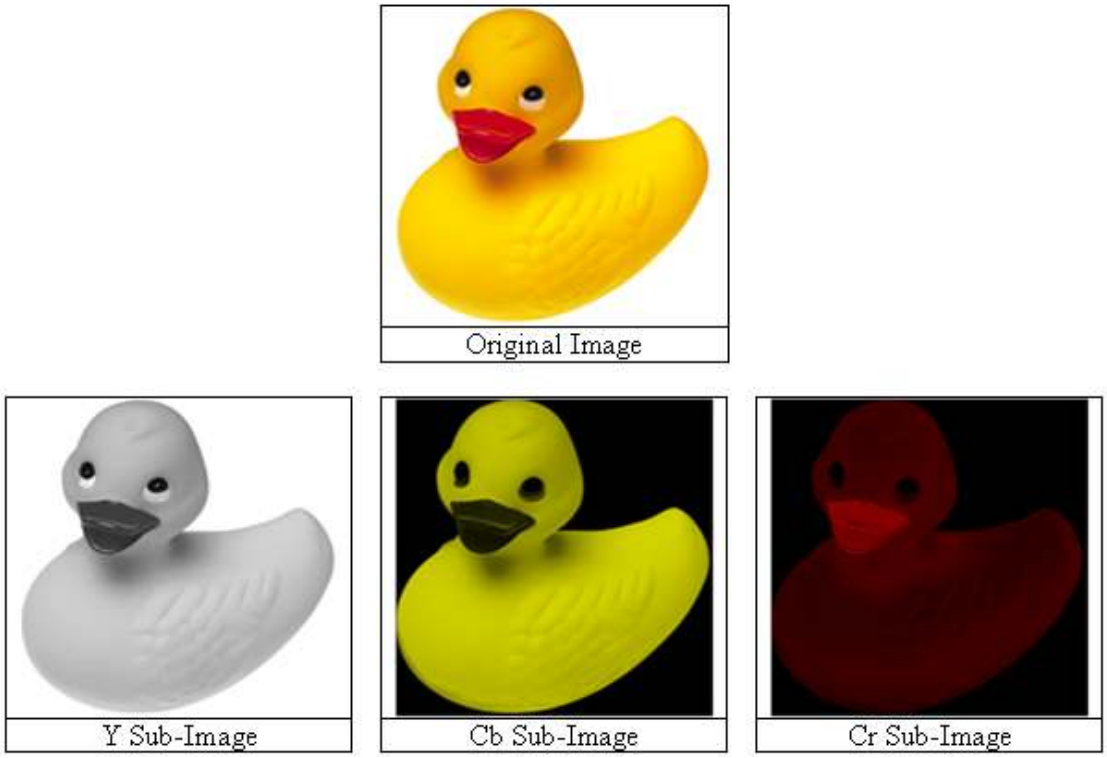

Figure 2. Decomposing an image into Y, Cb and Cr sub-images

This method is used in both image compression, such as the Jpeg format, and analog television broadcasts. In both of these applications, the color information is significantly reduced comparing to the black & white information, although the resulting color image does not seem much different from the original image

The method that is used in this project is YCbCr. As mentioned before, the values for red, green and blue in the RGB format are commonly stored as a byte, which ranges between 0 and 255. In this project, the values for Y, Cb and Cr are also stored in a byte. However, observing the conversion formulas, it becomes obvious that for some values of red, green and blue, the values of Y, Cb and Cr will be out of the byte range. For example, with red=255, green=255 and blue= 0, the Cb will be: Cb = Blue – Y = -226.95. Therefore, the Cb and Cr values need to be normalized to fit in the byte range. The value of Y always fits in the byte range, and does not need to be normalized. The normalization formulas are as follows:

Cb = Blue – Y = Blue – (0.3 Red + 0.59 Green + 0.11 Blue)
Cb = 0.89Blue – (0.3 Red + 0.59 Green)
The maximum value for Cb is when Red=0, Green=0 and Blue=255 so: $Cb_{max} = 227$
The minimum value for Cb is when Red=255, Green=255 and Blue=0 so: $Cb_{min} = -227$
Therefore, the normalized value for Cb (Cbn, as we call it) is as follows:

$$Cbn = \left[Cb + \frac{(Cb_{max} - Cb_{min})}{2} \right] * \frac{255}{(Cb_{max} - Cb_{min})}$$

$$\boxed{Cbn = (Blue - Y + 227) * \frac{255}{2 * 227}}$$

Similarly for Cr:
Cr = Red – Y = Red – (0.3 Red + 0.59 Green + 0.11 Blue)
Cr = 0.70 – (0.11 Blue + 0.59 Green)
The maximum value for Cr is: Cr_{max}=178.5; the minimum value: Cr_{min}=–178.5.
Therefore, the normalized value for Cr (Crn, as we call it) is as follows:

$$Crn = \left[Cr + \frac{(Cr_{max} - Cr_{min})}{2} \right] * \frac{255}{(Cr_{max} - Cr_{min})}$$

$$\boxed{Crn = (Red - Y + 178.5) * \frac{255}{2 * 178.5}}$$

For displaying the picture, we need to convert these normalized values back to RGB. The formulas for that are as follows:

Cbn = (Blue – Y + 227) * (255 / (2 * 227)) => $\boxed{Blue = \frac{2 * 227}{255} * Cbn + Y - 227}$

Crn = (Red – Y + 178.5) * (255 / (2 * 178.5)) => $\boxed{Red = \frac{2 * 178.5}{255} * Crn + Y - 178.5}$

Y = 0.3 Red + 0.59 Green + 0.11 Blue => $\boxed{Green = \frac{Y - 0.3\,Red - 0.11\,Blue}{0.59}}$

III) Materials and Methods

III-A) The overall strategy for encoding an image

III-A-1) In this step, the input image, which is in the RGB format, will be converted to the YCbCy format. In other words, the color input picture will be transformed into 3 monochrome sub-pictures, each image to be processed separately. As explained before (Section II-a), the Y image resembles the black & white version of the input image, and Cb & Cr images are the color information. The following steps describe the process for only one of these sub-pictures, but the same process must be repeated for each sub-picture separately.

III-A-2) In this step, a "low detailed" version of the input image will be made. To do that, two separate processes will be applied to the input images.

- First, the number of colors in the input image will be reduced. Based on experiments, a 16 color image gives best result for typical photographs.
- Second, the details of the image should be removed using a Median filter. For more information on the Median filter, please see appendix A.

III-A-3) In this step, high details of the input image will be separated into a new image. To do that, the "low detailed" image from the last step will be subtracted from the original image. By subtracting we mean the value of each pixel in the "low detailed" image will be subtracted from the value of the corresponding pixel in the input image.

III-A-4) In this step, the "low detailed" image will be traced and converted into vector graphics.

III-A-5) Steps III-A-2 through III-A-4 will be applied to the three Y, Cb and Cr sub-pictures. Therefore, each produces one Vector and one Raster image. Overall, each color image will be converted into 6 different monochrome sub-pictures. It should be noted that based on the content of the original image, some of these 6 sub-images may not contain any useful information, and therefore, it would be possible to omit some of them completely. For example, a black & white picture does not have any color information. So, the Cb and Cr sub-images do not contain any useful information, and can be safely discarded.

III-B) Reducing the color levels

In order to facilitate the tracing algorithm, the number of colors in the image must be reduced. Having too many diverse colors in an image will result in an image with very small areas of homogenous color. Since the tracing algorithm must deal with each of these areas separately, having too many of them will decrease the efficiency of the algorithm, and also produce an unnecessarily huge output.

It must be emphasized here that in this project, as explained before (Section III-A-1), the color pictures first convert into three monochrome sub-pictures for Y, Cb and Cr. Since these sub-pictures still contain color information, the term "color" is used to refer to their content, even though only a single integer number represents the color of each pixel. Throughout this report, the term "color" or "color value" for a monochrome image refers to the integer value that is representing the information of a single dot in the image. The single dot is also known as a Pixel. The color values are typically within the byte range, meaning that they can have any value between 0 and 255.

The color reduction means selecting a subset of color values used in a picture, and changing the color of each pixel to the closest value from that subset. The selected subset of values must be from the most frequently used values in the original picture, but it must also contain the values that are close to less frequently used colors as well. For example, if the image is mostly dark but has a few small light areas, the selected subset of colors must include a few values close to the ones used in lighter areas. If the selected subset

exclusively consists of only the frequently used color values, which are the dark colors, the color of lighter areas will have no choice but to change to one of those darker colors. Although the lighter areas are small, a large change in their color will be very noticeable. Therefore, a few light color values must exist in the selected color set to eliminate this problem.

The color reduction algorithm is as follows:

III-B-1) A histogram must be created for the input image. A histogram is a two dimensional graph that shows the distribution of different color values in an image. The X-axis is the different possible values that a point in the picture can have. If the values in the picture are stored in bytes, the maximum value for X-axis is 255. The Y-axis is the number of pixels in the image that have the same value as the one indicated by X-axis.

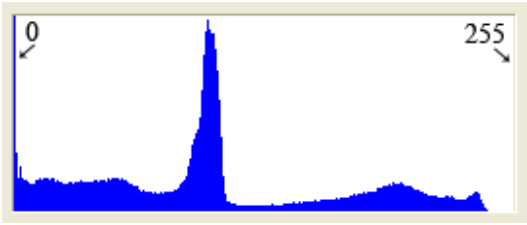

Figure 3. A typical histogram

The benefit of using a histogram is that it can easily show which values are mostly used in an image.

III-B-2) The histogram will be checked around 0 and 255. If there were no values for Y-axis around both ends of the graph, then new lower and/or upper limits for the histogram will be selected. These limits are the closest points to 0 and 255 on the X-axis that have a non-zero value for Y-axis. If there were any pixels in the image with values 0 and 255, the lower and upper limits would remain at 0 and 255.

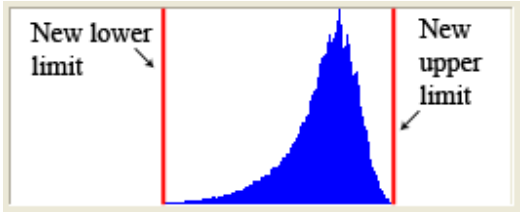

Figure 4. Selecting new Lower and upper limits

III-B-3) The range between the lower and the upper limits of the histogram will be divided by the number of different colors in the output image. For example, if the

histogram ranges between 0 and 255, and the output image should have 32 colors, then the histogram will be divided into 32 partitions, and each partition will contain 8 values.

III-B-4) In this step, each partition will be examined separately and only one value will be selected for each partition to replace all the color values in that partition. This value must be close to the one with the largest number of pixels (Value with highest Y). At the same time, the number of pixels for other values must be taken into consideration as well.

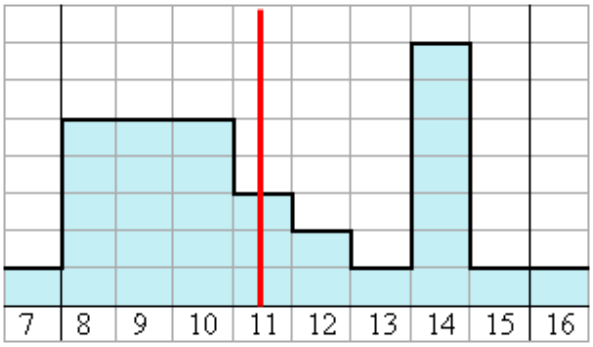

Figure 5. Selecting an X value in a partition within a histogram

A formula that takes all of the above into consideration can be written as follows. This formula was tested in the test program, and produced the most favorable results.

$$Selected\_X = \frac{\sum_{i=X_{Low}}^{X_{High}} Xi * f(Xi)}{\sum_{i=X_{Low}}^{X_{High}} f(Xi)}$$

Where X_{Low} is the X value for the start of the partition, X_{High} is the X value for the end of the partition, and f(X) is the number of pixels in the image with the color value of X. In other words, f(X) is the value of Y in the histogram for the given X. Using this formula for values in figure 5, will give 11 as the selected X.

The selected X must be replaced in all the pixels of the image that have a value greater than or equal to X_{Low} , and less than or equal to X_{High}.

Note: After the color reduction process, the histogram graph will lose its continuity, and will consist only of discrete values. The number of these discrete values is obviously the same as the number of different colors in the output picture.

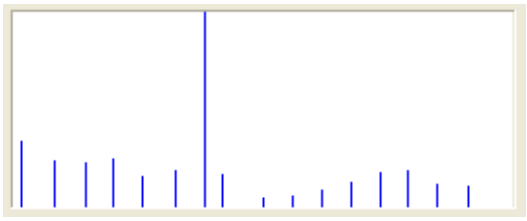

Figure 6. Discrete histogram after color reduction

III-C) The Tracing Algorithm

Tracing is the process of converting a Raster image into Vector graphics. In this project, the goal is to keep the outlines of the Vector image as close to the original picture as possible. The reason for that is because the Vector graphics is only one part of the final image. The other part, which is the Raster, will include all the differences of the vectored image with the original. Any unnecessary deviation in vector image will only result in more differences being transferred to the Raster image, and will increase its size unnecessarily. For this reason, the tracing process will only produce/use a collection of dots along with the straight horizontal and vertical lines with varying lengths that closely follow the outline of each "area" in an image. In this report, an "**area**" is defined as a collection of adjoining pixels, all with the same color. An image with a lot of details has many small areas, while a less detailed image has fewer areas with bigger sizes.

One of the initial problems was to find an appropriate method to dismantle the original picture and create two separate images; one for low details and the other for high details. The image with low details should have certain characteristics to facilitate the tracing process and conversion to Vector graphics. The tracing works best when the input image is made up of areas with different colors that are clearly distinctive from each other. Borders with blended colors will usually confuse the tracing algorithm. In addition, fewer colors in the image would help the tracing process as well. The best method to create a low detailed image was found to be the utilization of a Median filter. This filter preserves the sharp contrast between different color areas. Moreover, it does not create any new intermediary color on the borders, and consequently keeps the overall number of colors at minimum. For more information about the median filter, please refer to appendix A. It should be noted that other methods such as Gaussian filter and filter made by DCT/IDCT were also tested, but all failed to preserve a sharp distinction on the border between different color areas, and so were discarded. For more information about the Gaussian filter, please refer to appendix B. For more information about the DCT/IDCT, please refer to appendix D.

The complete output of the tracing process is a collection of areas. Each area can be drawn on the screen by using dots and lines. Also, for each area, a color value is stored and in addition, one or more points inside the area are selected to mark the starting point for flood filling (See appendix C) the area with the area color during the drawing process. These points will be explained in more detail in section III-C-4.

The proposed trace algorithm starts by detecting the presence of each area and finding all the information for that area in 4 separate phases. Note: Each phase starts presuming that the previous phase is completed. Therefore, it is not possible to process different phases simultaneously in different threads.

The trace algorithm detects a new area by scanning the image from top to bottom and from left to right. We call this "**area discovery loop**". Obviously, at the very beginning, the pixel at the position X=0 and Y=0 is identified as a point in a new area (because none of the areas in the picture were detected yet). Then, the area containing this point will be fully traced, and all the points in that area will be marked, so that they will not be traced again. Then, the area discovery loop will continue. It will skip all the previously marked pixels of previously detected areas until it finds an unmarked pixel, which would belong to the next area to be processed. The same process will be repeated until the area discovery loop reaches the end of the image.

As mentioned before, each area will be traced in 4 separate phases. We will generally call them "**the area processing routines**". They are as follows:

III-C-1) Phase 1: In this phase, the goal is to detect all the points in an area. The input is a single pixel in the area. The output is an assorted list of all points on the border of the area. Also, all the points in the area will be marked to indicate that they were already investigated. The algorithm for finding all the pixels in an area is very similar to the flood fill algorithm (See appendix C). It will start from the point given in the input and finds all the surrounding points with the same color as the first point. In this way, all the points in the area will be detected and marked. During the detection loop, a list of pixels, which are on the border of the area, will be made as well. **Border pixels** are those that are either located on the edge of the image, or have at least one neighboring pixel with a different color from their own. During this phase, the border pixels are added to the list as they are detected at no specific order. Since all pixels in an area have the same color, there is no need to store all the points of an area. Only the position of the border points and the color of the area is enough information for recreating the area.

It should be noted that it is possible to have one area inside another. In this case, the outer area will have two kinds of border points; those on the border with inner area (inner border), and those on the border with outer areas (outer border).
For each area, only the outer border is important. The inner border can be ignored as long as the orders of areas are correct. That is, the outer area is drawn *before* the inner area.

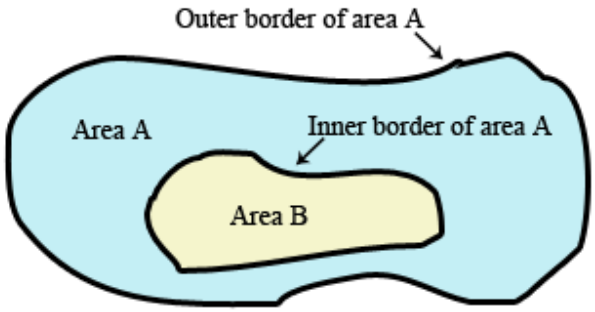

Figure 7. Inner and outer border of an area

III-C-2) Phase 2: In this phase, border points from the last phase will be ordered and also any "inner" border point will be removed from the list. The input is a list of assorted points on the border of the area, plus one starting point on the border. The output is an ordered and refined list of border points.
The starting point, which is mentioned above, is the same starting point used in phase1, which is also the same point initially found by the area discovery loop (explained at the beginning of the tracing algorithm). Remember that the area discovery loop scans the image from left to right and from top to bottom, and this is the first encountered point in any area. That means this point is the most upper left point in the area. Obviously, this point must be located somewhere on the outer border of the area. The algorithm in this phase is very simple. Starting from the "start point", the list of the border points is searched for a point right next to the start point. This newly found point will be added to a new list for ordered border points, and will be removed from the old list. Next, the old border list will be searched again for a point next to the last found border point. The same process will be repeated until all the points on the outer border are traversed, with the last found point being a neighbor of the start point (from the other side). At this point, there might still be some point left in the old border list, which belongs to the inner borders. However, they are not needed, and will be simply discarded. At this stage, we have an ordered list of points for the outer border of the area.

III-C-3) Phase 3: This phase is simply a format conversion for border points. The input is a list of ordered border points obtained from phase 2, plus a starting point, which is the same starting point used in previous phases. The output is the same starting point plus a list of "Directions", which are referenced to the starting point. These directions can be either up, down, left or right. The conversion algorithm is a simple loop, starting from the start point and traversing through all the points in the border list. In each step, the current point will be compared with the next point in the list, and the relative direction between the two points will be stored in a new list for the directions.

Note 1: The border points always form a closed circuit, which is obviously the border of the area, so in the newly created direction list, the number of left directions is always equal to the number of right directions. Similarly, the number of up directions is equal to the number of down directions.

Note 2: After writing and testing the program for the phase 3, a problem was discovered. In cases such as the one depicted in figure 8, it is not possible to determine exactly which pixel is the next pixel. Hence, the direction cannot be determined.

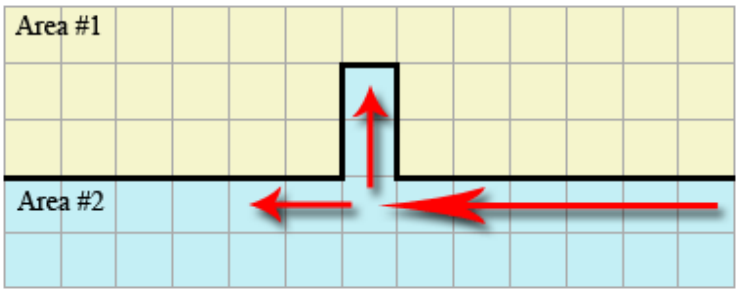

Figure 8. Problem in initial trace algorithm

To solve the problem, the following solution was devised, tested and proved to be capable of solving the problem:
Before starting the trace, the image size is tripled in both vertical and horizontal directions. In other words, each pixel will be replaced by a 3x3 matrix of pixels (See figure 9). None of the previously explained algorithms needs any changes, except the Phase 3, which needs some modifications as follows: Instead of adding one "direction" to the direction list for every traversed border point, in the new scheme, one direction will be added to the direction list for every 3 border points traversed in one direction.

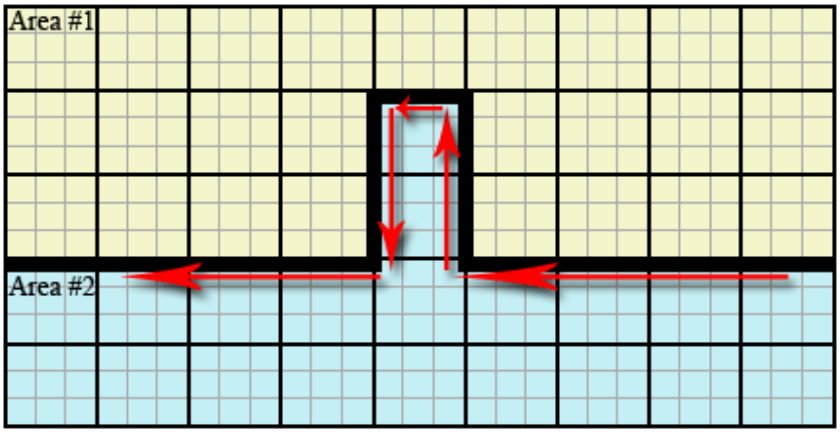

Figure 9. Solution to the problem in the trace algorithm

III-C-4) Phase 4: The redrawing of the area requires us to first draw the border and then fill the enclosed area with the designated color of that area, using the flood fill algorithm (See appendix C). For the flood fill algorithm, a point inside the area must be known so

that it can be used for initiating the algorithm from that point. Note that an area may be shaped in a way that requires more than one starting point for flood fill (See figure 10).

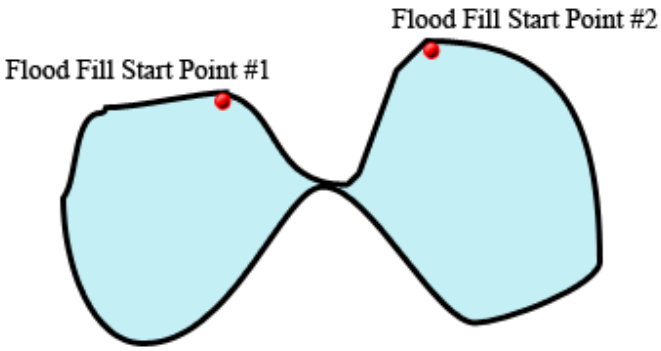

Figure 10. The case where an area needs to be flood filed from two different starting points

It is important to mention that this phase can be completely omitted from the tracing algorithm. Instead, it can be done prior to drawing. However, since the goal is to keep the drawing process as simple as possible, Phase 4 is carried out here. The algorithm for finding these start points for flood fill is as follows:
First, a point inside the area will be chosen. This point can be anywhere in the area. Then, a test flood fill will be initiated from that point. Next, all the points in the area will be checked to see if they are all affected by the flood fill. If there were still some unaffected points left, one of them will be selected and a second flood fill will be initiated from that point. The same process will be repeated until every point in the area is affected by one of the test flood fills. At the end, we will have a list of "inside points" that later will be used as starting points for flood fills during the drawing process.

At the end of all phases of the area processing routines, the following information is gathered for each area:

1- A starting point for drawing the area border
2- A list of directions, relative to the starting point
3- The color of the area
4- One or more points inside the area to indicate from where the flood filling should start in the drawing process

III-D) Re-drawing Algorithm

Each picture has two parts: one Vector and one Raster. For the drawing, first the vector part will be drawn. Then the raster part will be decoded separately, and added to the Vector image. The drawing algorithm for the vector part is as follows:
It starts by reading the information of the first area from the disk file. Then, starting from the border start point of that area, and reading the directions from the border list, the border around the area will be drawn. This border is always a closed loop. Finally, using the flood fill method, the area will be filled with its color.

III-E) Saving the output

It has been explained so far that every color image will produce at most 6 sub-pictures. Three of them will be saved as Raster images, and the other 3 as Vector graphics. The tree Raster sub-images can be saved using any of the existing methods. However, two methods were selected for further testing and analysis, as explained below:

III-E-1) The Gif format. The compression in this format is non-lossy. Since the saving of the vector parts is also non-lossy, using this format will make the overall saving method a non-lossy one. It should be noted that the GIF pictures can only utilize a maximum of 256 different colors in one image. But that does not cause any problem in our case, as our Raster sub-images do not have more than 256 colors (or values) at any rate.

III-E-2) Monochrome Jpeg format. The compression of the Jpeg format is lossy. Therefore, using this method will make the overall saving method a lossy format. This format only works for monochrome images and the files it makes are smaller than the ordinary Jpeg format.

The vector part will be saved using a custom format. In this format, basically all of the information gathered for all areas will be stored in a binary file. For more information about the file structure in this format, please see the source code of the accompanying software libraries. The only compression used in this format is for storing the list of directions that defines each area. Since there are only four choices for each direction, which are up/down/left or right, only 2 bits are enough to store a single direction. That means four directions can be packed into a single byte.

III-F) Programming Environment

This project was initially developed in Microsoft Windows environment. However, the main encoding and decoding routines can be easily ported to other operating systems, such as Linux and Mac OS. For the software development, the Borland C++ Builder [1] is used. One of the main advantages of C++ Builder is that there are compilers that can directly compile this source code into Linux binary file.

IV) Results and Analysis

There are different aspects to investigate for analyzing the efficiency and performance of this proposed method. It can be analyzed with regards to the time it takes to process, the quality of the converted image and the size of the converted image.

IV-A) Time Complexity Analysis

Each image in this method goes through a series of processes. Below is the time analysis for each step. In the following analysis, "*n*" is defined as the number of pixels in an image, which is width * height. Also, it should be noted that since these steps are performed one after another, the overall time complexity is the sum of all steps.

IV-A-1) Conversion to YCbCr. In this step, the value of each pixel will go through a set of calculations, which are not related to *n*. Instead, they use a constant amount of time for each single pixel. Therefore, the time is a linear function of *n*: $Time_1 = K_1 * n$

IV-A-2) Color reduction. This step has three phases as follows:

IV-A-2-a) Building the histogram. This is a simple loop that reads all the pixels in the image. So, the time is a linear function of *n*: $Time_{2a} = K_{2a} * n$

IV-A-2-b) Selecting the target color. This phase makes calculation on an array, the size of which is the number of colors in the input image. The number of calculations depends on the size of this array and the number of colors in the output picture. Since both numbers are usually constant, the time for this phase is also constant: $Time_{2b} = K_{2b}$

IV-A-2-c) Color replacement process. This phase scans the input image and replaces the value of each pixel with a new value. Therefore the time is a linear function of *n*: $Time_{2c} = K_{2c} * n$

IV-A-3) The median filter. This step performs a series of the same calculations for each pixel. The time of these calculations depends on the radius of the filter. Since this radius is usually constant, the time for this filter is a linear function of *n*: $Time_3 = K_3 * n$

IV-A-4) Making the "high-detailed" image. In this step, each pixel in the "low-detailed" image is subtracted from the original image. Therefore, the time is simply a linear function of n: $Time_4 = K_4 * n$

IV-A-5) Tracing the "low-detailed" image. This step has multiple phases that are analyzed below:

IV-A-5-a) The area discovery loop. This is a loop that checks all the pixels in the image, and invokes the area processing routines, whenever it finds a new one (See section III-C). The worst case is when every single pixel in the image is a new area with a color different from that of its neighbors. Even in this case, the process of checking each pixel itself, is a linear function of n: $Time_{5a} = K_{5a} * n$

IV-A-5-b) The processes in phase 1 and 4 of area processing are very similar (Sections III-C-1 and III-C-4). Only the phase 1 will be discussed here, but the same results apply to the phase 4 as well.

In this phase, all the pixels of each area will be processed. The process for each pixel is a simple compare and mark task, which runs in linear time. Note that this process is invoked from the "area discovery loop", which runs in linear time by itself. So, at the first glance it may appear that the combination of these two tasks forms a process that runs in quadratic time. But this conclusion is not correct, as the phase 1 of area processing does not process the entire image, but only a single area of that image. It is only after the subsequent calls to this

function that it will process the other parts of the image as well. So, in fact, after this function is called for all the areas in the image, each single pixel in the image is only processed just once (and not *n* times). Therefore, the combination is not quadratic; it consists of two linear functions with their times multiplexed, which means the result is still linear.

IV-A-5-c) The phase 2 of the area processing routines (Section III-C-2) contains a search that is performed for every border point in an area. In the worst case, that search routine may check all of the border points in that area, which makes the time a quadratic function of the number of border points (not a quadratic function of *n*). In the absolute worst case scenario, where all the areas are only two pixels wide, and the image is practically made of only border points of different areas, the time will become a quadratic function of *n*. This, of course, is a very rare situation and will not happen very often for ordinary images.
Note that in either case, since this process is invoked by the area detection loop - which has a linear time function- the overall time function for this section is at worst, a 3rd degree function of *n*.

IV-A-5-d) the phase 3 of the area processing routines (Section III-C-3) is a simple format conversion for the border points of each area. These points are obviously a subset of the pixels of the entire image. So, for typical images, this process runs in sub-linear time. The worst case scenario is when all areas are only 2 pixels wide, and therefore, all the pixels in the image are border points. In that case, the time is a linear function of *n*: $Time5_d = K_{5d} * n$
Note that in either case, since this process is invoked by the area detection loop - which has a linear time function- the overall time function for this section is at worst, a quadratic function of *n*.

IV-A-6) **Time Analysis Conclusion**: The time complexity of the whole process is a combination of linear, quadratic and 3rd degree functions of *n*. For pictures with higher number of areas, the quadratic and 3rd degree parts will have more influence. In other words, we can conclude that the time complexity of this method depends on the contents of the input image. For simple pictures, the time is almost a linear function of *n*. For images with more details, the time will tend to become a 3rd degree function of the size of the image.

IV-B) Picture Quality Analysis
As explained in the section III-E, the picture can be saved using either lossy or non-lossy methods. In case of lossy methods, the judgment on the quality of the output picture depends greatly on the viewer's personal opinion, and cannot be challenged. In the case of non-lossy methods, the output is exactly the same as the input. In this case, the picture quality is always perfect, and so there will be nothing to analyze.

IV-C) Picture's File Size Analysis
Since different methods are used to process different aspects of an image, the overall size of the output image is greatly influenced by the content of the image. However, to give a

general view on how this method compares with other popular methods, a few test images were processed with this method, as well as Jpeg and Gif methods. The results of this test are summarized in the table below:

	Image Content Description	Image Size	Size in Jpeg	Raster Jpg-ed	Size in Gif	Raster Gif-ed	Vector Zipped
1	Plastic toy, plain background	500x546	18.5	17.1	121	103.2	17.2
2	Natural Scene	421x640	40.3	38	215.8	168.7	77.5
3	Electrical Diagram	332x255	15.1	0	8.3	0	12.8
4	Comic Book	350x245	19.6	0	21.9	0	48.1

All sizes are in Kilo Bytes.
"Image Size" is the size of the image in pixels.
"Size in Jpeg" is the size of the input image if stored as Jpeg.
"Raster Jpg-ed" is the size of the "high detailed" part of the image if stored as Jpeg
"Size in Gif" is the size of the input image if stored as Gif.
"Raster Gif-ed" is the size of the "high detailed" part of the image if stored as Gif
"Vector Zipped" is the size of the "low detailed" part after it has been traced and the zipped with an external program.

To keep the test simple, only black & white pictures were used as input. So the output would be just two images (instead of 6 for color images). The output consists of the Vector part (for low details) plus a Raster part (for higher details). The Raster part for each picture is saved twice. Once compressed with Jpeg method and the second time, it is compressed with Gif method. This is done for comparison purposes. Otherwise, they are both the same and either one along with the Vector part can be used to draw the complete output image.

Analyzing the data; For the first image, we see that if the input image (the whole input image) is compressed as an ordinary Jpeg file, its size will become 18.5 KB. But if the same file is compressed as an ordinary Gif file, the size will be 121 KB. This is normal for Gif files because their compression method is not lossy. Then the same input file will go through the whole processes of the proposed format and will create two kinds of output. First, the Vector part with the size of 17.2 KB, And then the Raster part, which is compressed as both Jpeg and Gif. Their sizes are 17.1 KB and 103.2 KB.

Here we can see that adding the Vector part with the "Gif-ed" Raster part will give: 17.2+103.2 = 120.4 KB, which is slightly smaller than the size of the input file if only compressed as Gif.

Another point to mention is the value of 0 for Raster images of lines 3 and 4. That is because there was no useful information in Raster part and so they were discarded.

V) Conclusion

One of the greatest advantages of this method is that although the output looks like a Raster image, it contains the Vectored version of the image as well. That Vectored

part can be used to quickly create thumbnails for previews without the need to read the whole image. Also, it can be used as clip arts in software programs such as MS-Word. With regards to the file size, this format in its current state does not show any advantage over the commercial, Raster-only methods such as Gif or Jpeg. But we must not forget that this format decomposes the image in many different aspects. So there are many opportunities for removing the unnecessary data, which deserve further studies. Over all, this project is an implementation of a new idea. With more investments of time and effort, it has the potential to become a competitive format for storing images.

VI) Recommendations

The following recommendations can be considered for any future expansion of this project:

VI-A) The sub-pictures, which are in Vector graphics, can be further compressed using popular compression methods such as LZW.

VI-B) In Vector graphic sub-image, there are usually a considerable number of areas that are made up of only a single pixel. This will unnecessarily increase the size of the area list. These single pixels can be either stored in a separate list (to occupy less space), or transferred to the Raster part of the image.

VI-C) In color reduction algorithm, if the image creates a discrete and scattered graph in the histogram, the algorithm may not choose the best possible colors. It would be beneficial to investigate the possibility of improving this algorithm.

VI-D) In sub-images converted to Vector graphics, the starting points for the flood fill process (for drawing) are not essential data, and only added to improve the performance of the drawing process. Instead, they can be calculated later, using the information already stored for that sub-image. One option would be to develop such an algorithm, so that these points can be removed from the Vector graphic file and consequently reduce its size.

VI-E) Depending on the contents of the source image, some of the generated sub-images can be discarded without any significant loss in the overall quality of the image (See section III-A-5). So far, the decision to keep or discard any of these sub-images is made by the operator. It would be a good improvement to devise an algorithm to automatically make this decision.

VII – References

[1] Borland, "Borland® Developer Studio 2006 Feature Matrix", [Online document], (2005), Available at HTTP: http://www.borland.com/resources/en/pdf/products/delphi/bds2006_feature_matrix.pdf

[2] Wikipedia, "Flood fill", [Online document], (19 May 2006), Available at HTTP: http://en.wikipedia.org/wiki/Flood_fill

[3] Wolfram Research, " Discrete Cosine Transform", [Online document], (2006) Available at HTTP: http://documents.wolfram.com/applications/digitalimage/UsersGuide/ImageTransforms/ImageProcessing8.4.html

[4] Ken Nist," What exactly is ATSC?", [Online document], (27 Jan. 2006), Available at HTTP: http://www.hdtvprimer.com/ISSUES/what_is_ATSC.html

[5] Dave Marshall, " The Discrete Cosine Transform (DCT)",[Online document], (4 Oct 2001), Available at HTTP: http://www.cs.cf.ac.uk/Dave/Multimedia/node231.html

VIII) Appendixes

VIII-A) Appendix A

Median Filter Algorithm

A graphic "filter" refers to a process that changes the appearance of a picture. Examples of such changes include: sharpening, blurring, color adjusting and so on. The Median filter removes the small details in a picture. The algorithm for this filter is much simpler than those of other types of filters. The main idea is to sort the color value of each pixel along with its neighbors, and pick the value located exactly in the middle of the list. The name Median is thus adopted for this filter. For each Media filter, a radius must be defined, which means how many neighbors can participate in the filter. For example with radius = 2, a total of 24 neighbors will participate (See figure A.1).

Here is an example to demonstrate the algorithm for the Median filter. Consider the image fragment in Figure A.1, which will be used as input image. The output color value corresponds to the pixel in the middle of this image fragment (i.e., '5'). To find the output color value here, first the color values of the middle pixel and its surrounding pixels are copied into a separate list, and sorted in an ascending order. The result will be a list like this: 0, 1, 2, 3, 3, 3, 4, 4, 5, 6, 6, 33, 38, 39, 40, 41, 41, 42, 42, 43, 43, 43, 46, 61 and 87

43	42	43	61	87
41	39	46	33	6
42	40	5	3	1
43	41	4	6	3
38	4	3	0	2

Figure A.1, an example for a portion of an input image

After that, the value in the middle of this list will simply be selected. In this example it is the 13th item with the value of 38, which will be assigned to the corresponding output pixel. This process so far, produces a single pixel in the output image. The same process should be repeated for every pixel in the input image to create a complete output.

One of the significant properties of this filter is that it does not perform any kind of mathematical calculation to produce the pixels for the output image. For that reason, the filter does not introduce any new color value into the output image, and the color values of the output image are exactly the same as those available in the input image.

VIII-B) Appendix B

Gaussian Filter Algorithm

A graphic "filter" refers to a process that changes the appearance of a picture. Examples for such changes include: sharpening, blurring, color adjusting and so on. The Gaussian filter produces a blurred picture. The algorithm for this filter basically calculates the average of the color values of each pixel, along with that of its surrounding pixels. However, the Gaussian filter differentiates between each surrounding pixel. The pixels closer to the center will participate in the calculation with a higher ratio compared to the pixels which are further away. In practice, the color value of each pixel will be multiplied by a constant number called "weight", which is different for different pixels in different positions. The values of these weights at different positions correspond to a bell shaped curve or Gaussian curve; hence, the name Gaussian filter is adopted.

A Gaussian filter uses a matrix of weights similar to the one depicted in Figure B.1. The process of calculating each new pixel in the output image is done using the corresponding pixel in the input picture along with its neighbors. In order to calculate the color value of a new pixel, first the color value of the corresponding pixel in the input image must be multiplied by the weight value at the center of weight matrix. In this example, the weight value is 9. Also, all the neighboring pixels must be multiplied by the corresponding weight value from the matrix. For example, the color

0	1	2	1	0
1	3	4	3	1
2	4	9	4	2
1	3	4	3	1
0	1	2	1	0

Figure B.1, a typical weight matrix for Gaussian filter

value of the pixel immediately above the center must be multiplied by the weight value, which is immediately above the center weight. The same process must be repeated for all the neighboring pixels. To calculate the average, all the results of the above calculations must be added together and then divided by the sum of all weights in the weight matrix. Here, the sum of weights is 53. The same process must be repeated for all pixels in the input image to create a complete output.

VIII-C) Appendix C

Flood Fill Algorithm

Flood fill is an algorithm that starts from one point in an image, locates all points of the same color that are either connected to the start point or have a path to it which is made of points with the same color, and changes their color to a new one. This algorithm is usually used in popular paint programs in the "paint bucket" tool. Besides its application in paint programs, the flood fill algorithm is useful for detecting all the adjoining nodes in a two dimensional array that contains the same numeric value.

There are many ways in which the flood fill algorithm can be structured, but they all make use of a queue or stack data structure, either explicitly or implicitly [2] . The implicit use of stack is in fact through the recursive function calls. This method is not suitable for environments with limited stack size. The queue method is straight forward and does not have the above problem. It is explained below:
The flood fill algorithm takes three parameters: a start point, a target color, and a new color. It starts by checking the start point to see if its color is the same as the target color. If it is not, the algorithm terminates and there is nothing else to do. If it was the same color, that point will be inserted into an empty queue, and a loop will start. Inside the loop, first a point will be read (and removed) from the queue. The color of this point, which is already verified as the target color, will be changed to the new color. Then the color of the points that are above, below, to the left and to the right of this point will be checked. If any of them was the same as the target color, that point will be inserted into the queue. The loop will terminates when all the points in the queue were processed and the queue became empty.

VIII-D) Appendix D

DCT / IDCT

The Discrete Cosine Transform (DCT) is a method that can separate the image into parts that are different in importance with regards to the visual quality of the image. It transforms an image from a spatial domain to the frequency domain [5]. The DCT can have different forms for single or multidimensional array of integers. In this report, we are interested in DCT for two dimensions, which correspond to a flat (2D) image. In this transform, the input is a 2D array with a size of N1 by N2, and the output is also a 2D array with exactly the same size. Each element in the array is calculated by the following formula [3]:

$$X[k_1,k_2] = a[k_1] * a[k_2] * \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \left[x[n_1,n_2] * Cos\left(\frac{\pi(2n_1+1)k_1}{2N_1} \right) * Cos\left(\frac{\pi(2n_2+1)k_2}{2N_2} \right) \right]$$

$$a[k] = \begin{cases} \sqrt{\frac{1}{N}}, \text{for K} = 0 \\ \sqrt{\frac{2}{N}}, \text{for K} = 1,2,...\text{N} - 1 \end{cases}$$

In the output matrix, the values located at the upper left correspond to the areas with fewer details in the original image. In fact, the value at index [0,0] is the average of the values in the input image (also known as the DC offset). Similarly, the values located at the lower right of the output matrix represent the small details in the original image. (See the figure D.1)

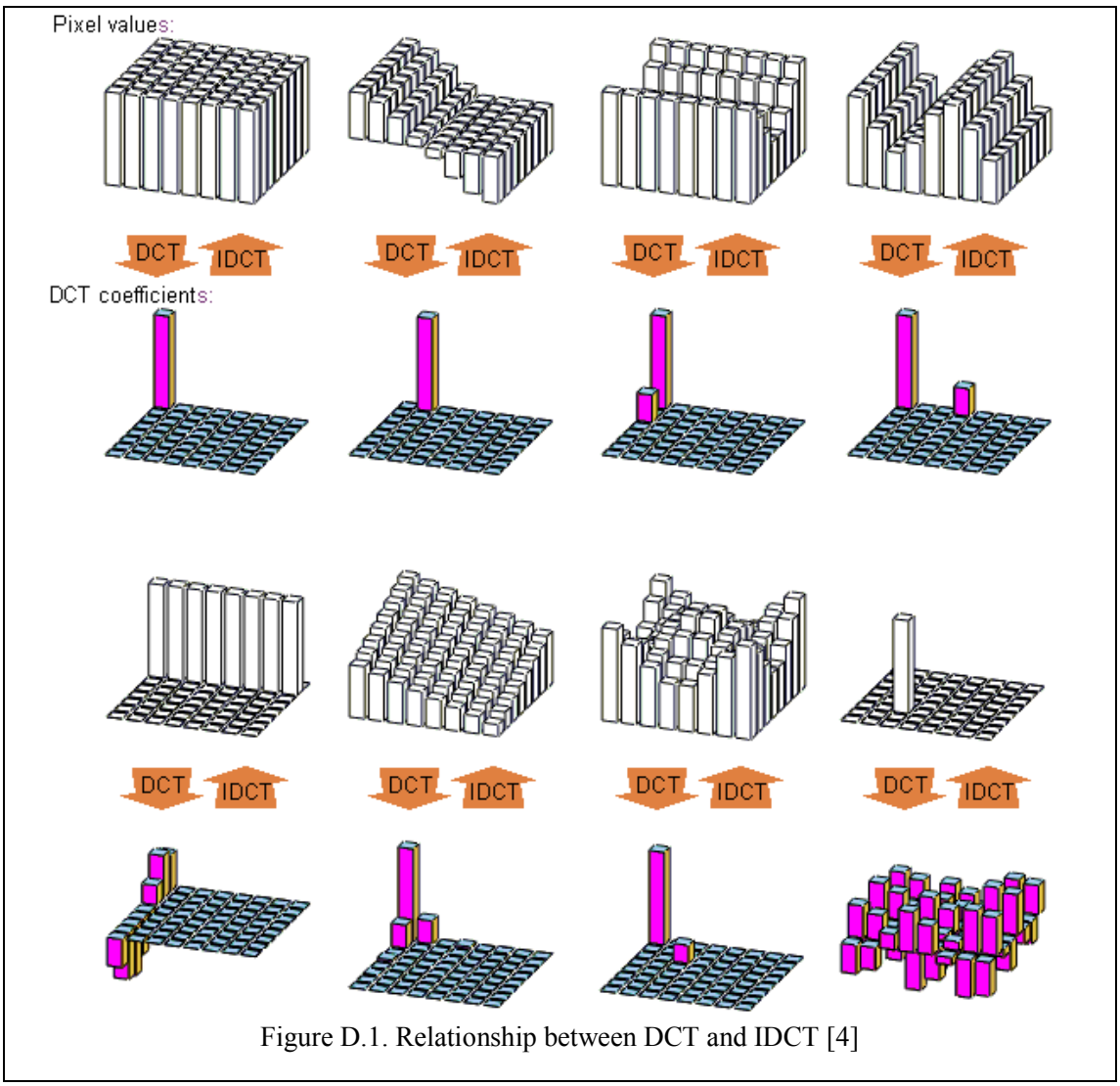

Figure D.1. Relationship between DCT and IDCT [4]

The Inverse Cosine Transform (IDCT) behaves exactly in the opposite way. It transforms a matrix from the frequency domain to spatial domain. The formula for a two dimensional IDCT is as follows [3]:

$$x[n_1,n_2]=\sum_{k1=0}^{N_1-1}\sum_{k2=0}^{N_2-1}\left[a[k_1]*a[k_2]*X[k_1,k_2]*Cos\left(\frac{\pi(2n_1+1)k_1}{2N_1}\right)*Cos\left(\frac{\pi(2n_2+1)k_2}{2N_2}\right)\right]$$

Theoretically, the DCT and IDCT can be used to create a low pass or a high pass filter for an image. To do that, first the DCT of the original image will be calculated. Then, in the resulting matrix, the values, which correspond to the unwanted level of details, will be replaced with 0. For example, to have a low pass filter, all the small details should be eliminated. So, all the values in the lower right of the transformed matrix should be replaced with 0. Finally, the IDCT will be calculated to form the filtered image (See figure D.2).

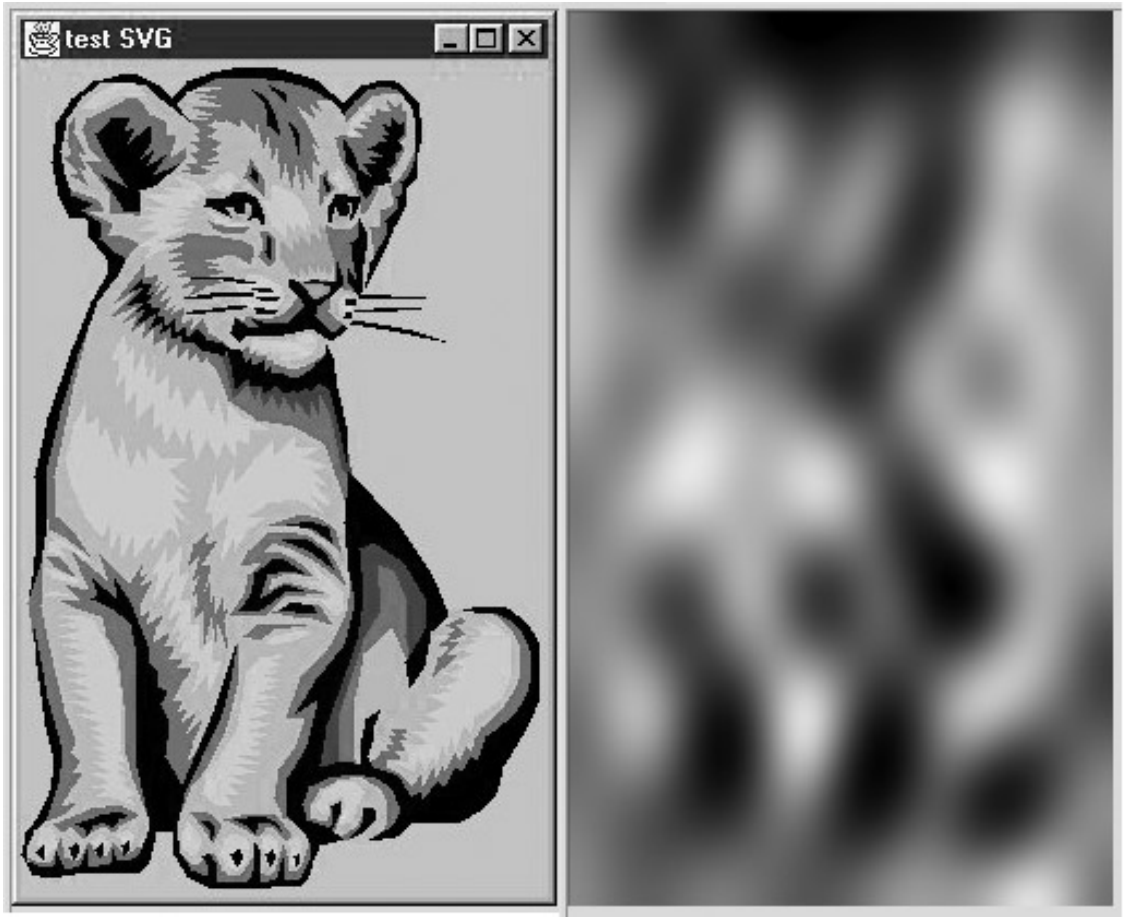

Figure D.2. Using DCT/IDCT to make a low pass filter

However, in practice, the time required for DCT/IDCT is so large that for the images with ordinary sizes, this method will take an unacceptably long time to compile.