A JPEG Decoder Implementation in C

Chris Tralie ELE 201 Fall 2007
Due 1/11/2008 Professor Sanjeev Kulkarni

1. Introduction

The purpose of this project is to create a decoder program in C that can interpret bit streams from .JPG files and display the result to the screen using the Windows API. Since there are multiple steps in this process, modular programming will be practiced. That is, each stage of the decoding process will be a component that has well-defined input and output. As such, the project will serve as a good review both for ELE 201 and COS 217. Someone who has taken both of these courses should be able to understand my code and report.

NOTE: During incremental development and debugging, the GCC c complier was used under Cygwin, and the open-source hex editor "Hack" was used to make sure I was on the right track understanding JPG files. The final executable program was compiled with Dev C++ for Windows, linking with the Windows API libraries.

NOTE ALSO: In this discussion, I will precede hex values with the notation **0x** (e.g. 0x18 is really 24 in base 10). This notation is also convenient since each hex value is a nibble.

2. Background Information

An uncompressed, color bitmap image, usually uses 24 bits for each pixel: 8 for the red component, 8 for the blue component, and 8 for the green component (RGB format). Although bitmap files are very straightforward to interpret, they store much unneeded data. For instance, there is a certain level of "redundancy" in most images. Parts of the image may not change very quickly, and there is often a very strong correlation between pixels in a region. Also, bitmap does not fully take advantage of all the properties of the human visual system, and thus stores more information than we can even detect under some circumstances. JPEG is a (usually) lossy method of compression that attempts to save space by taking advantage of these properties.

2.1: RGB to YCbCr and chroma subsampling

As it turns out, humans are much less sensitive to changes in chromiance, or color, than they are to changes in luminance, or brightness, in an image (Hass, Calvin). To take advantage of this, one of the first steps of JPEG is to make a "color space transformation." This maps each pixel, which is ordinarily stored in RGB format, to a new format: YCbCr (Y is Luminance, Cb is chroma blue, and Cr is chroma red). The mappings are as follows (Cuturicu, Cristi):

RGB to YCbCr (the transform an encoder uses):

$Y = 0.299*R + 0.587*G + 0.114*B$ (this weighting shows that the eye is more sensitive to green than it is to red and blue when it comes to perceived brightness)

$Cb = -0.1687*R - 0.3313*G + 0.5*B + 128$

$Cr = 0.5*R - 0.4187*G - 0.0813*B + 128$

YCbCr to RGB (the one the decoder must use)

$R = Y + 1.402 * (Cr - 128)$

$G = Y - 0.34414*(Cb - 128) - 0.71414*(Cr - 128)$

B = Y + 1.772 * (Cb - 128)

It should be noted that, convert a color image to a grayscale image, one merely has to calculate the Y component from the original RGB values for each pixel, and then to do the inverse mapping back to RGB with Cb and Cr values of zero.

The Cb and Cr components are less vital to the perceived quality of the image than the Y component, so the encoder can choose to quantize these values more harshly (more about quantization later), or it can "subsample." This means that instead of taking the Cb and Cr values at every pixel like an encoder would with the Y values, the encoder can choose to take them every other pixel, or perhaps only once every 2x2 block. The most common Y Cb Cr sampling schemes are as follows (Hass):

*1x1 chroma subsampling (Cb and Cr values are taken at every pixel)

*2x1 chroma subsampling (For every 2x2 block of pixels, the Cb and Cr values are taken from one 2x1 column)

*2x2 chroma subsampling (For every 2x2 block of pixels, the Cb and Cr values are only taken from one pixel). 2x2 is the most common for JPEG, and it is the one that seems to be used by MS Paint.

2.2 The Discrete Cosine Transform on 8x8 blocks

Instead of representing a JPEG image as a 2-dimensional spatial set of pixels, it is more efficient to transform it somehow into the frequency domain of separable, 2-dimensional cosine functions. This addresses part of the issue of the original image having redundancy; in an image with little detail and little change, the higher frequency components will be very small and can thus be compressed more (this option is not available in the spatial domain). A clever method is to perform the transform on 8x8 sub-blocks of the image instead of the whole image at once. This way, if there is an 8x8 block that is nearly constant and has little detail, it can be systematically compressed, while an 8x8 block that has more detail in the higher frequency components can retain more detail.

The Fourier analysis method of choice for JPEG is the "Discrete Cosine Transform" (DCT). The DCT assumes an even extension of the function on which it operates, so it only has cosine components for each frequency. This means that for an 8x8 block of 64 pixels, 64 DCT coefficients are needed. On the other hand, the complex Discrete Fourier Transform would need 128 frequency coefficients (64 for real components and 64 for imaginary components), so it would not be a wise choice. The formulas for the DCT and the Inverse DCT (the one my decoder must use) are listed below: (Cuturicu)

*NOTE: Since Y, Cb, and Cr components are stored in an unsigned byte (range from 0 to 255), a DC offset of 128 must be subtracted from the components before performing the transform

<u>Forward 8x8 DCT</u>

$$F(u,v) = 0.25c(u,v)\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y)\cos\left[(2x+1)\frac{\pi}{16}u\right] * \cos\left[(2y+1)\frac{\pi}{16}v\right]$$

<u>Inverse 8x8 DCT</u>

$$f(x,y) = 0.25\sum_{u=0}^{7}\sum_{v=0}^{7} c(u,v)F(u,v)\cos\left[(2x+1)\frac{\pi}{16}u\right] * \cos\left[(2y+1)\frac{\pi}{16}v\right]$$

Where f(x, y) represents the pixel component value at pixel offset (x, y) in the 8x8 block, F(u, v) represents a coefficient of frequency (u, v), and c(u, v) = 0.5 when u = v = 0 and 1 otherwise.

This transform is taken on each 8x8 block of sampled Y values, then on each 8x8 block of sampled Cb values and Cr values. Note that if 2x2 chroma subsampling is used, the first 8x8 Cb block will contain the Cb values taken from the first 16x16 block.

2.3 Zigzag Ordering and Quantization

After the 2D DCT coefficients have been calculated for each 8x8 block of each component, they are placed in "zigzag order" into a 1-dimensional array. This ensures that the lower frequencies in the upper left hand corner of the 8x8 block are all grouped together towards the beginning of the array, followed by the mid-range frequencies, followed by the high frequencies in the lower right corner. The table below maps a frequency (u,v) at position (u,v) in the table to a position within an array (Cuturicu). Note that frequency coefficients range from (0,0) to (7,7), and array indexes range from 0 to 63.

0	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

Once the frequency coefficients have been re-arranged as such, they are "quantized." This is done by dividing each coefficient by a chosen value and rounding it to the nearest integer. The rounding means that quantization is the primary lossy operation of the JPEG standard. Since lower frequencies are more important to the overall feel of the 8x8 block, their divisors are small (one would like to preserve as much of their resolution as possible). Conversely, since higher frequencies are much less important and detectable by the eye, they are often divided by a much higher value (and more of their information is lost). Hence, lower frequencies will generally take more minimum bits to store than higher frequencies. In many cases, the highest frequencies will likely be rounded to zero, especially with the Cb and Cr components, as colors tend to change slowly from pixel to pixel (Cuturicu). Variants of Huffman and run-length coding will be used to take advantage of these properties.

2.4 Zero Run-length coding variant

One would expect the quantized array of coefficients to have many runs of zeros, especially towards the high frequency region. Hence, encoding the number of zero coefficients before a nonzero coefficient would be advantageous. For instance, (5, 12); (3, -4) would be used in place of (0, 0, 0, 0, 0, 12, 0, 0, 0, -4). This is the exact scheme that JPEG uses, with a few restrictions. First of all, the length of a run of zeros can only range from 0-15. This means that a run of 38 zeros followed by a 5, for example, would be coded (15, 0); (15, 0); (6, 5). This is so that the run length can be stored on a nibble (4 bits), which will come in handy later with Huffman coding. Note that with this scheme, (15, 0) actually represents a run of 16 zeros, not 15, as it represents a zero preceded by 15 zeros. Also, there is a special marker (0,0), known as the EOB marker (end of block), that indicates that all of the rest of the coefficients are zero. This is extremely helpful in compression. For a Y block of constant luminance, for example, all that is needed is the (0,0) frequency coefficient and an EOB marker. To demonstrate this entire process further, consider this example of a set of coefficients:

(14, 0, 0, 0, -3, 8, 0, 0, 4, 23) is encoded as (0, 14); (3, -3); (0, 8); (2, 4); (0, 23)

2.5 Variable-length Huffman Coding

After zero run-length coding, the coefficients can be compressed further with an extremely clever Huffman coding scheme. First of all, the encoder needs to change its representation of each nonzero coefficient. It groups each value with a "category," which is a number representing the minimum number of bits needed to store the value (Cuturicu). It then associates each encoded category with a bit string of that minimum length. If the bit string starts with a 1, then use its ordinary unsigned representation of that bit stream for the coefficient. Otherwise, use the negative of the binary NOT of the bit string. For instance:

13 would be encoded as category 4: 1101	-13 would be encoded as category 4: 0010
42 would be category 6: 101010	-42 would be category 6: 010101

This scheme accommodates up to 16 bit two's complement signed precision for coefficients, so the maximum category number is 15. Therefore, the category number can be represented on a nibble. To change the previous run-length coding scheme to adhere to this new format, store the previous runs on the high nibble of the byte and the category on the low nibble. Then, put the representation bits directly after each grouping. Using the previous example:

(0, 14); (3, -3); (0, 8); (2, 4); (0, 23) turns into

(0, 4): 1110; (3, 2): 00; (0, 4): 1000; (2, 3): 100; (0, 5): 10111

The byte that contains the preceding zero run length and the category for the current coefficient can then be Huffman encoded. The advantage of doing it this way is that one would expect many coefficients to fall within the same category in an image, so they would probably have the same byte. This increases the frequency of that byte and decreases the length of the Huffman string for the byte. For example, a subset of coefficients, -7, 5, -6, 4 are all of category 3. Hence, assuming that each one of these coefficients had no zeros before them, they would all be represented by the same byte (0x03), followed by 3 bits. Also, in a big image, the EOB marker (still (0, 0) followed by nothing), would occur very frequently, so it would also take very few bits with a good Huffman table. If implemented this way, the combined Huffman and run-length coding can yield respectable compression ratios.

One special value, the DC coefficient (frequency (0,0)), which corresponds to the average component value over the 8x8 block and is the first coefficient of the DCT, is encoded separately from the rest of the coefficients. Since it obviously has no previous zeros, the byte that is Huffman coded simply contains the category of its value. Instead of storing the value of the coefficient directly, though, the difference between the DC coefficient and the last 8x8 block scanned is recorded (the first block starts with a previous value of zero). For example, if the first DC coefficient was 64 and the second one was 130, the second one would be encoded as (130-64 = **66**) : 0x07: 1000010. This provides a link between the 8x8 block and helps to further exploit the redundancy of the image; one would anticipate the average values of the components of adjacent blocks to be correlated.

3. Decoder Implementation Details

As stated before, the decoder that I created reads in streams of binary data from a file with the .JPG suffix. One major point to take into consideration before decoding is that JPEG stores information in its header (3.1) in “Big-Endian format” (I found this out the hard way using a Hex Editor). This means that for variables that are more than one byte long, the most significant byte comes first, and the least significant byte comes last. This is exactly the opposite of the Little-Endian Intel processor for which this program was written (all of the variables I was storing in memory had the least significant byte first), so I had to define some special endian swapping functions in a file **bitOps.c** before I could properly interpret the header information. I also put some other bit operation functions in that file, such as functions that can read a bit or write to a bit position within a byte, since I could not interpret the data byte by byte in the variable-length data code section.

3.1 The JPEG header

Before any data about DCT coefficients can be understood, the JPEG decoder must parse a very complicated header. A header is a portion right at the beginning of a binary file with fields that let a decoder know how to interpret the subsequent bits in the file. In the case of JPEG, the header must parse fields corresponding to different subsampling schemes, Huffman Tables, quantization values, etc.

The basic method to parse the header was to read the data in, byte by byte, from the beginning of the file. As special “markers” were encountered, I could redirect my program flow to fill in data in different structs that I had defined to store info about the header. A marker is the word (2-byte variable) 0xFFbb, where bb is any nonzero byte. Each marker is followed by another word that stores the length of the data following the marker, including the length variable. Here is a table summarizing a few vital header markers (Weeks, James). Please refer to my source code in **JPGObj.c** for more detailed information about the exact alignment of bytes within the header.

Marker (big endian)	Meaning	Information Following the marker / Notes
0xFFD8	Start of image	If the image does not start with this, then it's probably not a JPEG file
0xFFE0	JFIF Header	This header is not very important for my implementation. It has some information about the version, dot pixels per inch, and a possible thumbnail in RBG format (not common)
0xFFDB	DQT (Define Quantization Table)	WORD length, BYTE index (number associated with this table), and 64 bytes for the quantization values, stored in zigzag format NOTE: There are usually two quantization tables in a JPEG file.
0xFFC4	DHT (Define Huffman Table)	The Huffman trees are not present in the header. Instead, the DHT header states how many Huffman codes there are of each bit length, from 1 to 16, followed by an array that has all of the bytes that the Huffman codes represent. The array contains the bytes from left to right for each level and in increasing depth in the tree (more about this in section 3.2). NOTE: There are usually 4 Huffman tables present; a DC table and an AC table for the Y component, and a DC and AC table for the Cb and Cr coefficients. The BYTE index variable stores whether it is a DC table in the high nibble, and what its ID is in the low nibble.

0xFFC0	SOF (Start of Frame)	This stores the horizontal and vertical resolution of the image (words), the number of components (1 for grayscale, 3 for color), the subsampling factors for each component, and the quantization table used with each component. Normally, one quantization table is used for the Y component, and a different one is used for Cr and Cb components.
0xFFDA	SOS (Start of Scan)	This stores which Huffman tables are associated with which components. The program starts decoding the data section directly after it reads in this header.
0xFFFE	Comment marker	This can be anything the encoder wants. It is usually a null-terminated string that says something about the image.
0xFFD9	End of image	This should be the last word that is read in by the decoder. When my program was not working properly at first, this helped in debugging, as I knew I should hit 0xFFD9 the instant I finish decoding the data section, not before I am finished. As it turns out, the section of code that detected EOB markers was originally incorrect. Now everything seems to line up properly with the variable-length decoding as far as I can see.

All of the header structs and the functions that fill them in are defined in the **JPGObj.c**.

3.2 Decoding the Data Section

Once the header has been parsed properly, decoding the data section is relatively straightforward. The program decodes one "minimum coded unit" (MCU) at a time (Hass). An MCU includes one 8x8 block of the lowest sampled component and however many 8x8 blocks are needed of the other components to accommodate those pixels. The most common MCU occurs with 2x2 subsampling, in which 4 8x8 Y components are decoded (upper left, upper right, lower left, lower right), followed by one 8x8 block of subsampled Cb components, followed by one 8x8 block of subsampled Cr components. Hence, each MCU in this scheme takes up a 16x16 pixel block. For each of the 8x8 component blocks that make up the MCU, the program locates the appropriate DC and AC Huffman tables and quantization table, and it does the Huffman and run-length coding in reverse.

The trickiest algorithmic part of the decoding process was designing a function to fill in binary trees based on the "define Huffman table" portion of the header. To do this, I designed a recursive function in the file **HTree.c** (all of the decoding functions reside here) called **setValues**, which fills in the bytes from left to right at a particular level (the caller looped through each level from 1 to 16 and filled in the values at that level). Then, whenever I encountered a zero bit in the file stream, I knew to go to the left child of a node in this tree, and whenever I found a 1 bit, I knew to go to the right (Hass). I repeated this process until I reached a leaf node. At this point, I retrieved the byte that corresponded to the bit string I had read so far, and I performed run-length decoding to recover the quantized coefficients. Finally, I looped through and multiplied the coefficients by the values in the quantization table to "dequantize" them. This process was repeated until all of the coefficients were decoded, and the program was then ready to do the inverse DCT on a block and to render an RGB image to the screen.

4. Results/Conclusion

The program works surprisingly well on most JPEG images. There is only one known bug at present that gives rise to slight artifacts. When I first got all of the blocks rendered to the screen in order, I noticed that there seemed to be a high amount of scattered noise throughout the image. I recalled that during the Inverse DCT phase, I had stored the Y, Cb, and Cr values in floating point format for maximum precision, but I then had to store my RGB values back on an unsigned byte after conversions. I noticed that before casting RGB values to a byte, some of them were slightly negative or slightly over 255 (both outside of the range of an unsigned byte). This meant that when they were casted, the most significant information was usually lost, and they would take on seemingly random values. I tried halving each of the DCT coefficients to prevent this type of overflow/underflow, but the cost in contrast in my images was too great (they all became very dim). I found that, perceptually, a better solution was to map all RGB values over 255 to 255 before conversion, and to map all RGB values below 0 to 0. I cannot explain why this happened, but perhaps it has something to do with deficiencies of floating point numbers, or perhaps there was something wrong with my quantization tables that caused the amplitudes of certain frequencies to be too great.

In spite of this minor problem, I am very pleased with how well the program works. Although debugging was quite painful at times, this project met the design specifications and was an overall success. My program can display JPEG images to the screen of resolution 800x800 or less in full color. This was an excellent learning experience for me as well. Not only did it solidify my understanding of coding methods, but I had never dealt with a binary file directly before. Additionally, this project can be easily extended because of the code and data types that have already been mapped out. One natural avenue for further work is a JPEG encoder. If I decide to proceed to the encoder phase, I will probably use my program for JPEG steganography.

4. Executing the Decoder with Examples

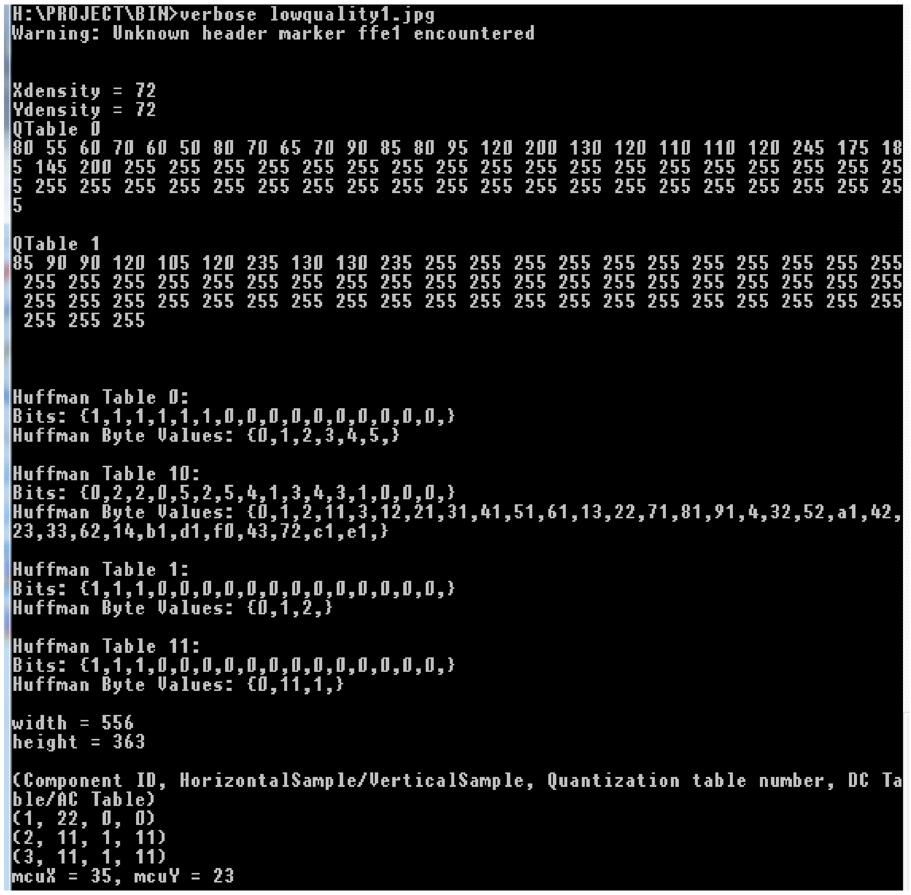
NOTE: I tested this on the undergraduate EE labs computers, so it should at least work there.

I have provided a CD with this assignment. At the root of the CD, there should be a file **Start.bat**. Double click on this batch file to begin execution. Once the batch file has started, there are two commands you can type in: **display** and **verbose**. Display will render and image to the screen, while verbose will display information to the console about the image's header. Here are the sample images I have provided:

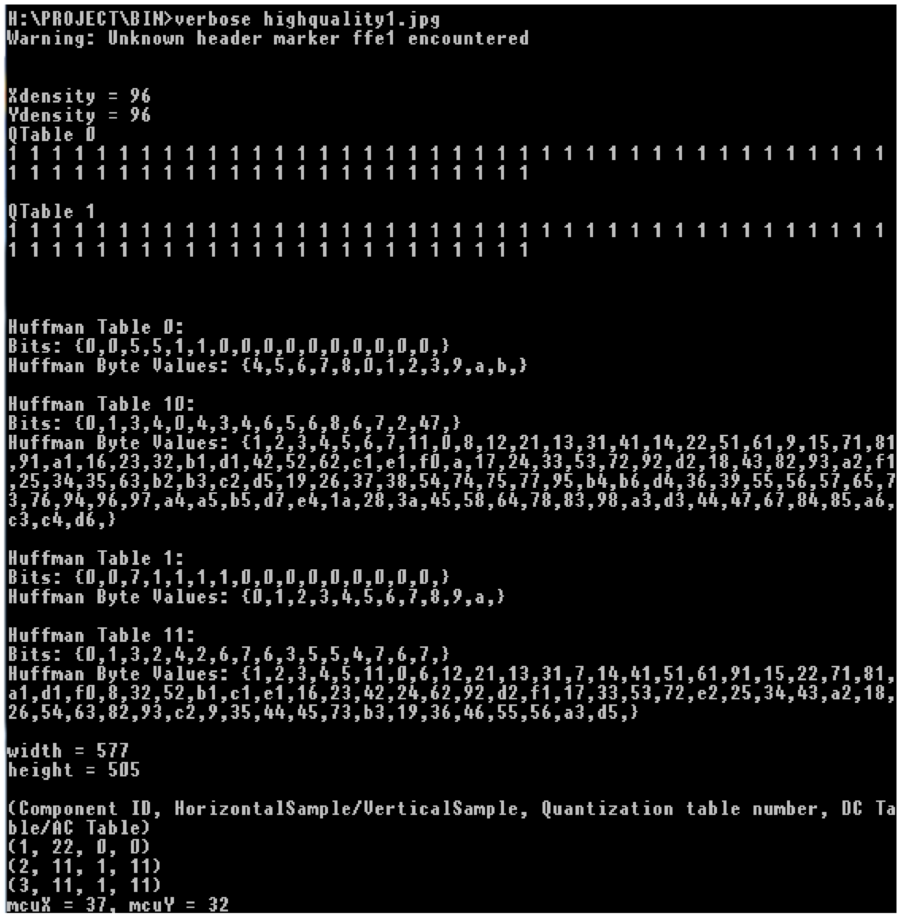
Typical Photographic Quality	High Quality	Low Quality
photo1.jpg (a freshman engineering outing)	highquality1.jpg (part of my desktop)	lowquality1.jpg (rocket at Johnson Space Center)
photo2.jpg (minigolf)	highquality2.jpg (a building)	lowquality2.jpg (a violin)
photo3.jpg (my senior portrait)	highquality3.jpg (Princeton emblem)	lowquality3.jpg (a skateboarder)

Example executions of the “verbose” command are shown on the next page. First, I typed **verbose lowquality1.jpg** at the prompt, and then I typed **verbose highquality1.jpg** at the prompt. These two images were generated with the help of “The Gimp,” an open-source Photoshop equivalent. The Gimp had a slider that allowed me to specify percent quality of the images. All of my low quality images were saved with 10% quality, while all of my high quality images were saved with 100% quality. Notice how the quantization table for the high quality image is all ones (no quantization), while the low quality table is much more coarse (all but the low frequencies have quantization divisors of 255, the max possible).

The “display” command is much more straightforward. Simply type display followed by the name of the image to display (e.g **display photo1.jpg**). It is also possible to display your own images if you provide an absolute path to your image (e.g. **display C:\images\image1.jpg**). *NOTE: It may take a few seconds to display images because of the computationally intense inverse DCT. Also, to view my source code, double click on the file* ***source.bat****.*



```
H:\PROJECT\BIN>verbose lowquality1.jpg
Warning: Unknown header marker ffe1 encountered


Xdensity = 72
Ydensity = 72
QTable 0
80 55 60 70 60 50 80 70 65 70 90 85 80 95 120 200 130 120 110 110 120 245 175 18
5 145 200 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 25
5 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 25
5

QTable 1
85 90 90 120 105 120 235 130 130 235 255 255 255 255 255 255 255 255 255 255 255
 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
 255 255 255



Huffman Table 0:
Bits: {1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,}
Huffman Byte Values: {0,1,2,3,4,5,}

Huffman Table 10:
Bits: {0,2,2,0,5,2,5,4,1,3,4,3,1,0,0,0,}
Huffman Byte Values: {0,1,2,11,3,12,21,31,41,51,61,13,22,71,81,91,4,32,52,a1,42,
23,33,62,14,b1,d1,f0,43,72,c1,e1,}

Huffman Table 1:
Bits: {1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,}
Huffman Byte Values: {0,1,2,}

Huffman Table 11:
Bits: {1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,}
Huffman Byte Values: {0,11,1,}

width = 556
height = 363

(Component ID, HorizontalSample/VerticalSample, Quantization table number, DC Ta
ble/AC Table)
(1, 22, 0, 0)
(2, 11, 1, 11)
(3, 11, 1, 11)
mcuX = 35, mcuY = 23
```



```
H:\PROJECT\BIN>verbose highquality1.jpg
Warning: Unknown header marker ffe1 encountered


Xdensity = 96
Ydensity = 96
QTable 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

QTable 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1



Huffman Table 0:
Bits: {0,0,5,5,1,1,0,0,0,0,0,0,0,0,0,0,}
Huffman Byte Values: {4,5,6,7,8,0,1,2,3,9,a,b,}

Huffman Table 10:
Bits: {0,1,3,4,0,4,3,4,6,5,6,8,6,7,2,47,}
Huffman Byte Values: {1,2,3,4,5,6,7,11,0,8,12,21,13,31,41,14,22,51,61,9,15,71,81
,91,a1,16,23,32,b1,d1,42,52,62,c1,e1,f0,a,17,24,33,53,72,92,d2,18,43,82,93,a2,f1
,25,34,35,63,b2,b3,c2,d5,19,26,37,38,54,74,75,77,95,b4,b6,d4,36,39,55,56,57,65,7
3,76,94,96,97,a4,a5,b5,d7,e4,1a,28,3a,45,58,64,78,83,98,a3,d3,44,47,67,84,85,a6,
c3,c4,d6,}

Huffman Table 1:
Bits: {0,0,7,1,1,1,1,0,0,0,0,0,0,0,0,0,}
Huffman Byte Values: {0,1,2,3,4,5,6,7,8,9,a,}

Huffman Table 11:
Bits: {0,1,3,2,4,2,6,7,6,3,5,5,4,7,6,7,}
Huffman Byte Values: {1,2,3,4,5,11,0,6,12,21,13,31,7,14,41,51,61,91,15,22,71,81,
a1,d1,f0,8,32,52,b1,c1,e1,16,23,42,24,62,92,d2,f1,17,33,53,72,e2,25,34,43,a2,18,
26,54,63,82,93,c2,9,35,44,45,73,b3,19,36,46,55,56,a3,d5,}

width = 577
height = 505

(Component ID, HorizontalSample/VerticalSample, Quantization table number, DC Ta
ble/AC Table)
(1, 22, 0, 0)
(2, 11, 1, 11)
(3, 11, 1, 11)
mcuX = 37, mcuY = 32
```

Bibliography

Cuturicu, Cristi. “CRYX's note about the JPEG decoding algorithm.” 4 January, 2008 <http://www.opennet.ru/docs/formats/jpeg.txt>.

Hass, Calvin. “JPEG Snoop – JPEG File Decoding Utility.” 4 January, 2008 <http://www.impulseadventure.com/photo/jpeg-snoop.html>.

King, K.N. *C Programming: A Modern Approach*. New York: Norton & Company, 1996.

Kingsbury, Nick. “Sync and Headers.” 4 January, 2007 <http://cnx.org/content/m11097/latest/>.

“Specifying Attributes of Variables.” 4 January, 2007 <http://docs.freebsd.org/info/gcc/gcc.info.Variable_Attributes.html>.

Weeks, James R. “JPEG Header Information.” 5 January, 2007 <http://www.obrador.com/essentialjpeg/headerinfo.htm>

“Windows API Programming.” 4 January, 2008. <http://www.toymaker.info/Games/html/windows_api.html>.