Jpeg Compression : A primer

Sam Heller

Graphics and Gui Programming

Jim Mahoney

Submitted : 5/4/04
Revised : 5/5/04

An attempt to create a relatively easy to understand primer on the basics colorspace and JPEG encoding.

Introduction

This document is an attempt to clearly explain the basic principles of image compression, JPEG encoding in particular. The entire process of creating a digital image is fundamentally based on the creation and manipulation of pixels. A pixel is a single point of color. Viewed on a computer or television screen, they are a composite of varying intensities of the primary colors Red, Green, and Blue. Image compression is essentialy the process of manipulating these three basic colors in any number of esoteric ways to reduce the amount of information needed to represent an image and to control the level of detail.

The process of JPEG conversion is twofold. First, image information is converted into different formats which allow more exacting control over minute aspects. Once a certain level of control has been reached, certain parts of the image, which due to the limitations of the human eye, can be removed without any noticeable effect. Once those uneccesary aspects are removed, the image is then viewed as pieces of data, which are rearranged and re-ordered with the intent of storing the maximum possible amount of information in the smallest possible space.

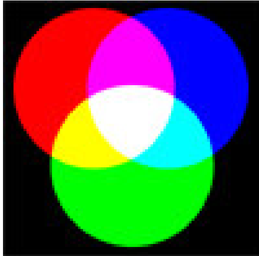
The goal of this paper is to provide what is more or less a laymans guide to this entire process. In order to do this several basic concepts of image and data representation as well as conceptual abstractions must be explored. The process of JPEG conversion is much more detailed and complex than is within the scope of this paper. That said, this is still somewhat of a technical document, but every attempt is being made to explain these concepts in a clear and concise manner. The downside to this is that omissions and generalizations are neccesary. This is by no means an authoritative document, and should be used as a guide, not a primary reference source.

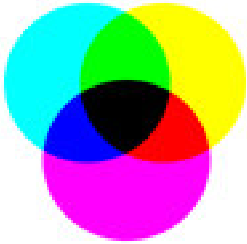
Color

RGB

When creating color on a pixel, each component of Red, Green, and Blue light is defined as a value of intensity. For the purpose of this paper color intensity is defined in a range of 0 to 255, 0 being complete dark and 255 being as intense as it gets. Because they are primary colors they can be combined in varying ratios to create any color desired. However, there is a substantial difference between mixing these colors and the mixing of, for instance, paint colors. The mixing of paint is done in what is called a subtractive manner. When you mix equal parts of red, green, and blue paint, you'll come up with black, or at the very least a really icky brown color. RGB mixing, on the other hand, is an additive process. If you mix equal parts of each together you create white.

This additive quality, among other things, makes RGB very well suited for pixel representation but when trying to store large amounts of image data, a little bit of a snag can occur. For example, take these images demonstrating the difference between additive and subtractive mixing. They're each 100x100 pixel images with each pixel requiring 3 seperate components. So in order to display just one of these relatively small images, there are 3 x 100^2 values that need to be stored. Thats 30,000 values. It's fairly obvious how that could balloon into something absurd very quickly. So the problem is, how do you represent these images without taking up such absurd amounts of space?


Additive Mixing


Subtractive Mixing

Taken from RGB entry on the wikipedia

YUV

Another method of representing color information is known as YUV. While they both consist of three seperate image components, the way they represent data is drastically different. Instead of using

three basic values of color to create pixel values, YUV is composed of a Luminance component (Y) which describes light intensity, and two Chrominance components (Cb and Cr) which define color. This method of representation, in addition to more closely mirroring the functioning of our own visual systems, allows for slightly more control over various aspects of the pixel. Since the intensity and color of a pixel are seperated, they can each be manipulated independent of the other. This is not something really possible with the RGB model, because of the very basic nature of how it defines color, you can't manipulate a specific color without affecting the intensity as well.

This seperation of components is vital, especially because the Chrominance values contain far more information than the human eye can interpret. This fact combined with the ability to alter the chrominance values independently allows us to begin to weed out information that is doing nothing but taking up space. The next progression in this attempt to more effeciently control these color values is a rather large jump into the world of differential equations.

The Discrete Cosine Transform

The Discrete Cosine Transform is a mathematical formula which belongs to a family of functions known as Fourier Transforms. The Mathematics of the DCT are intense and fairly confusing for those who aren't intimately familiar with higher level math. To put it concisely, when a DCT is applied to a set of numeric data, it looks at the entire set and interprets it as a waveform. If you give it a set of 64 numbers representing a waveform it will give you back a new set of 64 numbers representing the sinusoidal values of the waveform.

$$f_j = \sum_{k=0}^{n-1} x_k \cos\left[\frac{\pi}{n} j (k + 1/2)\right]$$

The Discrete Cosine Transform

Copied from wikipedia entry on the DCT.

Now, unless you harbor an intimate love of math, that last paragraph probably made little, if any, sense. Defining this in laymans terms can be a little tricky, and rife with innacuracies that are fairly irrelevant as far as long as you're more concerned with the effect of the formula as opposed to the way it works.

For the purpose of image compression, this formula is applied to an 8x8 block of individual

pixel components (Y, Cb, or Cr independent of the others). It crunches over all 64 values in the block and returns a completely different set of 64 values referred to as DC-components. This new set of 64 DC-components represent various components of the entire 8x8 block, as opposed to components of each individual pixel. The DC-Components are ordered from lowest to highest in terms of complexity, or detail.

The first few DC-components are fairly vauge representations of whichever image component the DCT was run on. For instance, if the DCT was run on a block of Luminance values, the first few DC-components it returned would be generalized values of Luminance which were present over the entire block. Like RGB, these DC-components are combined in an additive nature. Each new DC-coeffecient provides greater levels of detail, eventually representing the same amount of information as the original block, but in a completely different notation.

It is at this point where severe manipulation of the image data occurs, through a process called image quantization.

Quantization

In sharp contrast with the DCT, image quantization is a refreshingly straightforward process. As already noted, there is quite a lot of excess information contained in images. The DCT has seperated out these levels of complexity which means that they can now be altered or completely removed. Quantization is not much more than simple division.

Depending on the image component that the DCT was run on, each DC-component in the 8x8 block is divided by a corresponding value contained in the appropriate quantization table.

Luminance Quantization Table

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Chrominance Quantization Table

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

After the quantization occurs, the resulting value is rounded off and saved. Many values end up rounding off to zero because of the value of the number they are being divided by. The design of the quantization table can be completely arbitrary, however, the ones shown above are the standard tables defined by the ITU-T81 JPEG standard and do a pretty good job of getting rid of excess information without degrading or effecting the quality of the image.

The quality of the image can be further tweaked by multiplying the quantization values by a (usually small) number choosen by the encoder. Generally speaking though, a value of two or greater renders the image more or less unrecognizable.

At this point, we've skimmed pretty much all of the fat that we can off of the image data. The next stage of the process involves various encoding schemes designed to futher minimize the size of the data.

Encoding

In order to understand the full value of encoding data, a basic understanding is required for how data in general is stored on a computer, specifically the concepts of hexadecimal and binary notation.

Bytes, Bits, and Hex's

At the absolute lowest level, all data on a computer is stored in the binary notation of 1's and 0's called bits. However, these binary values are generally managed in groups of hexadecimal values called bytes. Each byte contains 8 bits which can be used to represent any number of things. For example, the character "A" is represented by the hexadecimal value 41 which has the binary value of 01000001. So that's one byte with a value of 41, stored in the 8 bits 01000001.

Runlength Coding

Often in data streams there is quite a bit of repetition. Runlength Coding (RLC) is a very simple

way to take advantage of that repetition to further decrease the space required to store that data. It works is by replacing long strings of repeating values with one instance of the value and a number indicating how often that value is repeated. Look at the following stream of data

AAABAACAACDDAAAAABABABBBBBBDDD

That's 30 separate values stored in 240 bits. Now if we rewrite it using RLC, we get :

(3A) (B) (2AAC) (DD) (5A) (2BA) (6B) (3D)

That's 18 Values stored in 144 bits, almost half of what we had initially. There are more complex implementations of RLC, including the one used by JPEG which is called zero runlength coding. It's not something worth explaining at this point, just keep this concept in mind.

Huffman Encoding

The Huffman encoding scheme is one which concentrates on a lower level of data storage. Instead of merely dealing with the end values, it creates an alternate model for the binary representation and storage of values. Like RLC, it also looks at repetition in data, but it completely ignores the ordering, concentrating instead on frequency of character occurence. Consider the following string of data

AAHHHBCCDHHDDDHFFEEEEGEGABCDBEFHHEFAHHBDA

The first step to encoding this string is to look at the number of time each individual character occurs and build a frequency table ordered by occurrence values. Once we have the table, we use it to build a data model called a Huffman Tree. This tree is just a way to visually represent an abstract data structure. It's construction logic is straightforward but doesn't neccesarily make sense if until

you see the end result. For now just worry about the construction of the tree, not the reasoning behind it's design.

Building the Tree

The first thing we do is look for the two lowest occurrence values (G and C) and group them together to start the tree.

Character	G	C	F	B	A	D	E	H
Occurence	2	3	4	4	5	6	7	10

G C

Now, instead of removing G and C from the frequency table, combine their values, so they're now worth 5. Now start again, looking for the next two lowest values (F and B) and add them to the tree.

Character	F	B	GC	A	D	E	H
Occurence	4	4	5	5	6	7	10

G C F B

Now, rebuild the frequency table, and repeat the process. At this point the two lowest values are GC and A. This means that we just add A onto the GC section of the tree..

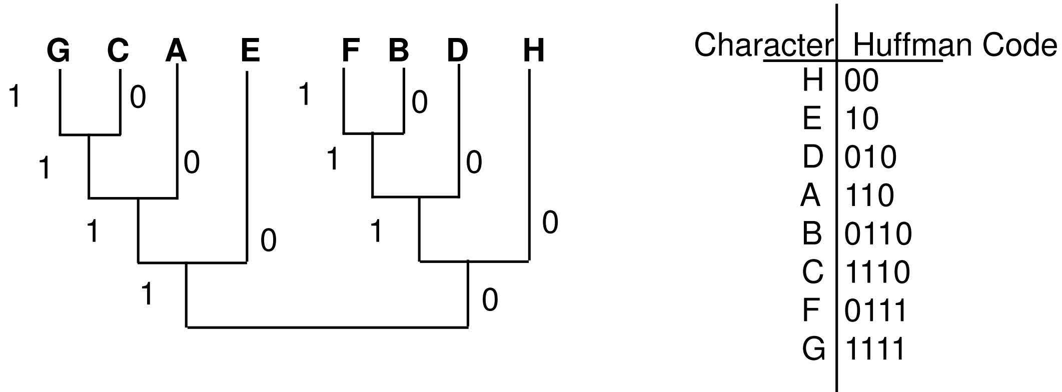
G C A F B

We continue this process, until every value is present in the tree, at which point we get a model that looks like this.

G C A E F B D H

If we start at the bottom of the tree and want to reach any particular value, we travel in a series left and right movements up the tree. For instance, if we wanted to reach A, then we would go Left, Left, then Right. If we describe this path up the tree in binary by stating that a 1 value indicates a

move left, and a 0 value indicates a move right, we have a simple and economical method to represent this movement.



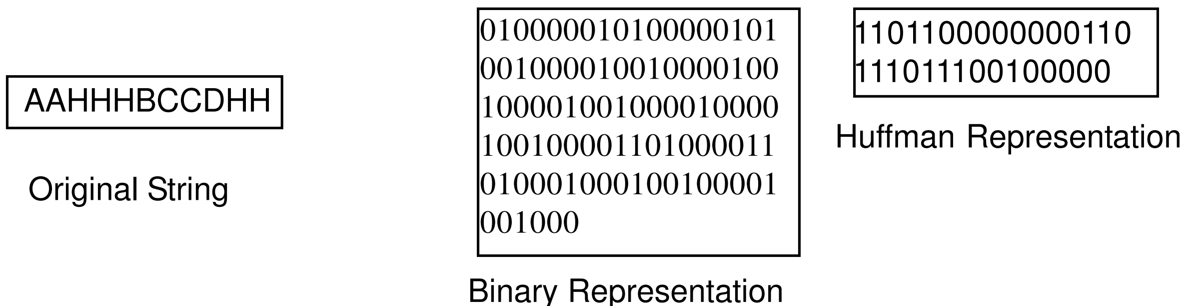
This manner of representation can be recorded in what a Huffman Table, which is a list of binary values called Huffman codes corresponding to a list of the characters.

The reasoning behind the tree structure is hopefully clearer at this juncture. The values that occur most commonly have the shortest Huffman code, while the values that occur less frequently have longer Huffman codes. Now it all comes together.

The Huffman table is encoded in the file preceding the data. This table has 8 characters and 26 binary values, so it comes out to a total of 90 bits. Now instead of writing out the hexadecimal value of each character we can write out the Huffman code for it.

In order to write out 10 of the characters of the string this table was built for in regular hexadecimal, it would take 80 bits to represent.However, writing it out using the Huffman codes only requires the use of 32 bits..

In this particular case, the requirement to include the 90 bit Huffman table increases the size of



the Huffman encoded string to 122. However, if the entire string is encoded with this method, you have more respectable savings. In straight hexadecimal notation, the entire string takes up 328 bits. Huffman encoding is able to represent the string in 209 bits (including the 90 bit table). It should be fairly obvious at this point that the Huffman scheme works better with larger amounts of data. The compression ratio is directly linked to the amount of data being encoded.

JPEG Encoder Walkthrough

Okay, we've gone over pretty much everything required to understand the JPEG encoding process. This section is meant to provide a concrete demonstration of the concepts discussed throughout this paper.

Step 1 : Colorspace Conversion

The first thing we need to do is get the image data out of the RGB colorspace and into YUV. Eric Hamilton's paper "JPEG File Exchange Format" details formulae for this conversion.

This is a simple matrix multiplication. Basically it says to multiply red by the first value in each row, green by the second value, and blue by the third. Then total it all up and add 128 to the Cb and Cr values, and nothing to the Y value.

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.299 & + & 0.587 & + & 0.114 \\ -0.1687 & - & 0.3313 & + & 0.5 \\ 0.5 & - & 0.4187 & - & 0.0813 \end{bmatrix} \text{X} \begin{bmatrix} Red \\ Green \\ Blue \end{bmatrix} + \begin{bmatrix} +0 \\ +128 \\ +128 \end{bmatrix}$$

Lets Start with the following values :

Reds

33	32	34	34	32	29	38	28
44	51	44	42	42	41	40	29
56	53	52	82	88	70	38	30
95	85	89	143	140	75	40	37
148	144	122	150	121	73	57	49
169	190	143	140	117	134	119	70
175	228	181	187	197	226	190	99
220	241	219	210	244	235	219	123

Greens

18	23	30	34	30	20	28	22
21	33	30	32	32	26	28	20
23	25	28	61	68	50	22	18
52	46	54	109	110	46	15	21
96	95	77	107	82	38	27	25
111	136	90	90	68	89	80	39
116	170	125	130	143	173	143	61
159	182	162	152	186	177	165	79

11	18	27	32	31	21	29	22
15	29	27	30	31	29	30	23
16	21	24	58	67	52	23	20
43	39	50	107	108	48	18	22
85	88	71	101	77	36	27	23
99	124	82	81	61	84	75	36
100	156	112	119	133	165	135	52
141	166	145	138	172	165	153	66

After applying the conversion formula we get these :

Y Values

-106.31	-102.87	-97.14	-94.22	-97.28	-105.19	-96.896	-104.206
-100.80	-90.074	-94.15	-93.23	-93.12	-97.17	-96.184	-104.967
-95.931	-95.084	-93.28	-61.06	-54.13	-71.79	-101.10	-106.184
-64.169	-71.137	-63.99	-9.062	-9.258	-73.10	-105.18	-102.102
-17.706	-19.147	-38.22	-8.827	-34.90	-79.76	-92.03	-96.052
-1.026	22.778	-23.06	-24.07	-46.14	-26.11	-36.909	-80.073
3.817	57.746	12.262	17.789	30.006	59.93	28.141	-56.664
47.187	69.817	49.105	39.746	73.746	64.97	51.778	-37.326

Cr Values

8.06909	4.90649	2.2439	0.16259
11.9878	9.3252	7.2439	5.1626
17.0691	14.3252	12.3252	10.7439
22.2317	20.0691	17.8252	17.1626
26.8943	25.0691	22.9878	21.9878
29.9756	27.9756	27.1504	25.7317
30.8008	30.1382	29.0569	29.3943
31.9634	30.8008	29.8821	30.1382

Cb Values

-0.6634	2.83969	6.77029	9.13777	9.10769	4.9451	7.16117	5.54754
-0.6185	4.80301	5.08330	6.85444	7.35444	6.72191	7.32401	5.9451
-2.2091	0.73064	2.29996	13.1456	16.4015	12.5345	4.36001	4.34266
3.75074	3.63651	8.19668	25.7647	26.7377	9.82352	1.75505	4.06236
14.3519	16.5598	12.3675	21.6500	15.3706	4.42596	2.98958	2.40545
17.3122	25.4413	13.8942	13.9003	8.50926	16.4456	14.7743	4.89893
16.6344	33.9043	21.3240	23.6462	28.5285	38.6423	30.7094	7.27777
28.1183	36.3136	30.1876	28.5372	38.6750	36.9914	34.0882	9.63263

0.91870	4.4187	4.9187	3
5.0813	7.2561	5.8374	4.2561
10.0813	9.8374	7.9187	5.8374
15.1626	14.3374	12.2561	7.9187
19.9065	17.6626	15	12.1626
25.0691	22.9065	19.9065	15.7439
27.813	27.1504	24.1504	19.7317
0.1382	29.9756	27.9756	23.0569

Step 2 : Discrete Cosine Transform

Due to the annoyingly complex nature of the DCT, before these values can be run through it they all need to have 128 subtracted from them. The explanation would take at least another 10 pages and probably another two weeks.

Post DCT Y's

-2839.684	-2110.108	421.9667	-288.6896	155.88795	144.81830	166.51424	62.793205
-76.55416	40.155171	-29.4407	-30.80738	8.1155219	113.28241	-195.5582	537.81862
-515.8054	-305.0000	242.9430	311.49422	-257.0510	-98.06543	7.9502593	29.452906
-39.40645	56.972388	-73.9698	92.662595	2.0637053	35.642327	-484.3606	577.09627
-94.39451	-100.6143	-41.1075	-111.2262	215.65232	-52.44706	20.779365	4.3598763
-3.059665	33.730500	-10.4739	43.337697	-47.84591	20.502824	-124.1418	284.50073
-214.0486	-117.0599	114.1007	-48.18856	30.374948	4.0808427	13.935403	20.556045
-40.04132	50.953505	-65.2861	107.60766	-61.65439	23.592612	-28.57956	61.993949

Post DCT Cb's

1108.186	-383.3392	-31.5466	-13.12400	-6.459576	0.1250999	-6.674715	0.7351888
-2.69333	7.7657266	-3.82009	4.5208844	-0.162558	5.0549597	-2.763259	139.37678
-4.29980	-54.11245	18.41341	8.3706858	4.0120400	4.8281583	3.2768756	7.1559714
-2.99068	10.178810	0.776009	9.3787279	-3.387893	12.860224	-9.154700	68.410796
-12.3600	-62.18321	-1.23797	-6.207398	0.2013820	-2.165932	2.6949863	2.4962227
-1.29278	1.7737930	0.908273	6.9747337	-5.071009	8.3684342	-1.967254	57.618788
0.044956	-46.95568	4.155120	-5.628569	0.3295392	-7.785915	0.6923177	1.6984639
-1.62035	2.7969660	-0.99463	2.7965586	1.1023176	4.2847946	-3.991238	43.448326

Post DCT Cr's

872.15656	-354.1337	110.04863	-78.69675	45.697784	40.178537	-51.29957	12.792144
-20.05875	5.0561958	-7.675064	-22.16592	1.5894958	20.237869	-56.05146	46.923676
-160.6473	-30.53594	59.387422	94.214057	-82.48934	-31.10950	0.4861114	6.0760250
-9.576603	11.458551	-21.63120	24.074417	0.9153078	3.4126870	-134.1091	118.47385
-31.06748	18.120246	-11.68223	-29.38433	62.304479	-12.14458	7.3675185	-1.558818
-2.585503	7.7186281	-3.000186	10.429191	-14.20979	-0.729168	-36.68985	43.665873
-63.01157	1.9517068	29.842508	-7.926997	7.0729022	4.1746896	7.2089793	2.9872271
-9.711238	17.180228	-17.90075	29.629483	-14.08521	0.1563493	-7.445540	-17.97507

Step 3 : Quantization

Now that we have the quantized values for each block of data, we pull in the quantization tables and use them to weed out the information we don't need. It is at this point that any image quality choices should be made, uniform increase or decrease in the quantization table will directly affect the end quality of the image.

Quantized Y Values

-177	-191	42	-18	6	3	-3	1
-6	3	-2	-1	0	1	-3	9
-36	-23	15	12	-6	-1	0	0
-2	3	-3	3	0	0	-6	9
-5	-4	-1	-1	3	0	0	0
0	0	0	0	0	0	-1	3
-4	-1	1	0	0	0	0	0
0	0	0	1	0	0	0	0

Quantized Cb Values

65	-21	0	0	0	0	0	0
0	0	0	0	0	0	0	1
0	-3	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Quantized Cr Values

51	-19	1	0	0	0	0	0
0	0	0	0	0	0	0	0
-3	-1	0	3	-1	0	0	0
0	0	0	0	0	0	-1	1
-1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

It should be clear now why quantization is responsible for the biggest reduction in image filesize. At this point, the image data is almost ready for encoding, however there are two small steps to take before that happens.

The first step is to Zig-Zag re-order the values in each of the blocks. Zig-Zag reordering is a pretty simple task. All you do is take the values from the DCT and rearrange them in a Zig-Zag order

in the same matrix. The re-ordering table below is a representation of where the old values go in the new matrix. Although it doesn't really work in this case, the majority of the time this allows for easier runlength encoding of the quantized values.

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

Zig-Zag Reordering Table
From Page 30 of ITU-T81

Y Values Zig-Zagged

-177	-191	3	-3	-3	9	3	0
42	6	1	1	-36	-3	0	0
-18	-6	0	-23	3	-6	0	0
3	-1	15	-2	9	0	0	0
-2	12	0	-5	0	0	0	0
-6	0	-4	0	-1	0	0	0
-1	-1	0	3	1	0	1	0
-1	3	-4	-1	0	0	0	0

CB Values Zig-Zagged

65	-21	0	0	0	1	0	0
0	0	0	0	0	0	0	0
0	0	0	-3	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Cr Values Zig-Zagged

51	-19	0	0	0	0	0	0
1	0	0	0	-3	0	0	0
0	0	0	-1	0	-1	0	0
0	0	0	0	1	0	0	0
0	3	0	-1	0	0	0	0
-1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

The second step is to remove the first value of each block and put it aside seperately before encoding the rest of the data. This value is called the DC-Coeffecient, and Cristi Cuturicu does an excellent job of explaining it's importance in his "Note about the JPEG Decoding Algorithm" :

> [The DC-Coeffecient] is the coefficient in the quantized vector corresponding to the lowest frequency in the image (it's the 0 frequency) , and (before quantization) is mathematically = (the sum of 8x8 image samples) / 8 . (It's like an average value for that block of image samples). It is said that it contains a lot of energy present in the original 8x8 image block. (Usually it gets large values).

The DC values are later encoded in a slightly different matter seperately from the rest of the image data (referred to as the AC-Coeffecients).

Using this scheme, the encoded values for each component are :

Encoded Y Data

(0, -191)	(0, 3)	(0, -3)	(0, -3)	(0, 9)	(0,3)	(1, 42)
(0,6)	(0, 1)	(0, 1)	(0, -36)	(0, -3)	(2, -18)	(0, -6)
(1, -23)	(0, 3)	(0, -6)	(2, 3)	(0, -1)	(0, 15)	(0, -2)
(0, 9)	(3, -2)	(0, 12)	(1, -5)	(4, -6)	(1, -4)	(0, -1)
(3, -1)	(0, -1)	(1, 3)	(0, 1)	(1, 1)	(0, -1)	(0, 3)
(0, -4)	(0, -1)	(3, 0)				

Encoded Cb Data

(0, -21) (3, 1) (13, -3) (15, 0) (15, 0) (11, 0)

Encoded Cr Data

(0, -19) (6, 1) (3, -3) (6, -1) (1, -1) (6, 1)
(4, 3) (1, -1) (4, -1) (15, 0) (6, 0)

After all the 8x8 blocks in the image have been encoded, a (once again) slightly modified Huffman encoding is performed. When encoding an entire image, all of the component values for all the blocks are used to build the frequency count. Since we've only got one of each here, we'll just use that data, but it's important to remember that at this point we are no longer operating on individual 8x8 blocks of data.

Before the frequency count of is taken, the strings of zero-runlength encoded data are altered. The second value of each runlength pair is replaced by the minimum number of bits required to represent that value and the binary representation of the actual value is placed after the runlength pair. When encoding negative numbers, the bits required to store that number as a positive value are counted, the bits are inverted, and that value is stored after the runlength pair.

For Example:

Initial Pair	**Positive Value In Binary**	**# of bits required**	**Binary Value**	**New Runlength String**
(0, -191)	10111110	8	01000001	(0, 8) 01000001
(0, 12)	00001100	4	1100	(0, 4) 1100
(1, -23)	00010111	5	01000	(1, 5) 01000

Once everything has been converted in that manner a frequency count is taken of the runlength pairs, but **not** the binary value.

The DC Coeffecient

Now We've got all the information we need to write the AC-coeffecients out to a file, but we still need to code the DC. In yet another weird little trick, the designers of the JPEG standard decided that because the DC-Coeffecients of progressive blocks where generally fairly close together, they would encode not the value of the DC, but rather the difference between the current DC and the last one. So essentialy, the value encoded = ($DC_{current}$ - DC_{last}). For the first DC value of any image, DC_{last} is equal to zero.So in order to store these values, you first need to convert them all to the difference values, and then proceed to zero runlength and huffman encode those values.

The Final Bit-Stream

Now we've got our fully encoded image data just ready to be written to a JPEG file. The JPEG file is structured in a fairly intricate manner. After the markers identifying it as a JPEG, the quantization tables used are written to the file. After that each individual Huffman table is recorded, and finally, the image data is written out.

So uh...Now what?

That's pretty much all there is to tell. If you're at all interested in pursuing this concept any further, I've got a minor reading list assembled that you might find amusing.

Books

Video and Image Processing in Multimedia Systems
Borko Furht Stephen W. Smoliar and Hongjiang Zhang
Kluwer Academic Publishers

Image and Video Compression Standards
Vasudev Bhaskaran and Konstantinos Konstantinides
Kluwer Academic Publishing

Files

The Jpeg File Interchange Format by Eric Hamilton
ITU-T81 (The Last Free JPEG Standard)
A note about the JPEG decoding algorithm by Cristi Cuturicu
The Wikipedia....It's awesome, in the original sense of the word.

Appendix A : JpegEnc.pl

```
#!/usr/bin/perl
```

Declaring Modules :

GD is used for importing image data
Math::FFT is used to apply the DCT
Algorithm::Huffman builds the huffman tables.

```
use GD;
use Math::FFT;
use Algorithm::Huffman;

####
# Grab image data from the command
# line and get info about it.
####
```

Getting the Image :

We take the name of the picture from the command line, at this point this script only accepts PNG images.

```
my $pic = shift;
my $img = GD::Image->newFromPng($pic, 1);
```

Get image height and width information, use it to create variables describing the number of 8x8 blocks in the image. After all these, undefine the $pic variable.

```
my ($width, $height) = $img->getBounds();
my $wblock = ($width  / 8);
my $hblock = ($height / 8);
my $blocks = ($wblock * $hblock);
undef $pic;
my @HuffCount;
my @data;
```

Tables :

Define the quantization and the zig-zag tables here. The quantization tables have already been zig-zag reorderd for ease of use.

```
my @LuminanceQT = qw
(16 11 40 51 60 55 29 51 10 24 61 58 14 22 87 55 16 12 26 13 17 80
35 64 12 19 16 14 62 24 81 121 14 24 56 18 77 104 103 120 40 69 22
103 113 87 101 112 57 37 109 92 78 72 98 100 56 68 49 64 92 95 103 99);

my @ChrominanceQT = qw
(17 18 99 99 99 99 99 99 24 99 99 99 24 99 99 99 47 18 99 26 56 99 99
99 21 66 56 47 99 99 99 99 26 99 99 99 99 99 99 99 99 99 99 99 99 99
99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99);

my @ZigSort = qw
(0 1 5 6 14 15 27 282  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);
```

Colorspace Conversion :

Here we take the RGB values for each pixel and convert them into YUV.

```
my $ct = 0;
my $blocktag = 0;
my $ct1 = 0;
my $ypos = 0;
```

This loop goes over each pixel, one 8x8 block at a time. Once it's done a single 8x8 block, it passes all of it's information to the encoding subroutine, waits for it to finish, and then processes the next 8x8 block

```
While ($ypos != $height)
   {
   for (my $x = 0; $x != $width; $x++)
   {
   $ct++;
   for my $y (0 .. 7)
       {
```

Color Conversion

```
       my  ($r,$g,$b)   = $img->rgb($img->getPixel($x,$y));
       push (@YVals,  ((0.299 * $r + 0.587 * $g + 0.114 * $b + 0) - 128));
       push (@CbVals, ((-0.1687 * $r - 0.03313 * $g + 0.5 * $b + 128) - 128));
      push (@CrVals, (( 0.5    * $r - 0.4187  * $g - 0.0813 * $b + 128) - 128));
      $ct1++;
      }
   if ($ct == 8)
       {
```

This calls two subroutines for each component. The call to the encoder subroutine (which is run first) and then passes the output of the subroutine to the Parsing subroutine. The output of the parsing subroutine is a count of how many components it handles, and is placed in the nested array $Count

```
       $Count{0}{$blocktag} = Parser(EncodeBlock(\@YVals,\@LuminanceQT),0,$blocktag);
       $Count{1}{$blocktag} = Parser(EncodeBlock(\@CbVals, \@ChrominanceQT,1,
             $blocktag),1,$blocktag);

       $Count{2}{$blocktag} = Parser(EncodeBlock(\@CrVals, \@ChrominanceQT, 2,
             $blocktag), 2, $blocktag);
          undef @YVals;
          undef @CbVals;
          undef @CrVals;
          $blocktag++;
          $ct = 0;
                         }
                 }
   $ypos += 8;
         }
$blocktag—;
```

After all of the image data has been encoded and parsed, we build Huffman Tables

```
my $YHuff = Algorithm::Huffman->new(@HuffCount[0]);
my $YEncodeHash = $YHuff->encode_hash;

my $CbHuff = Algorithm::Huffman->new(@HuffCount[1]);
my $CbEncodeHash = $CbHuff->encode_hash;

my $CrHuff = Algorithm::Huffman->new(@HuffCount[2]);
my $CrEncodeHash = $CrHuff->encode_hash;
```

At this point we have all the neccesary data to write the JPEG stream. This code doesn't actually deal with writing this stream, as it's an additional level of complexity that's quite a lot to handle.

```
sub EncodeBlock
   {
```

This is the encoding subroutine. It gets passed an array reference to whatever image data it's currently working on, as well as a reference to whichever quantization table should be used for that block of image data.

```
   ($arrayref, $qt) = @_;
```

Here we build the Fourier Transform object and run the DCT on whatever data the encoder block has been passed. After that the data is zig-zag re-ordered.

```
   my $fft   = new Math::FFT($arrayref);
   my $dct  = $fft->ddct();
   for $ct (0 .. 63) {push (@Zigged, $dct->[$ZigSort[$ct]]);}
```

Get rid of the FFT object and th DC-coeffecients.

```
   undef $dct;
   undef $fft;
   undef $arrayref;
   untie $fft;
```

Quantize all the data, round it off, and stick it in the @Rounded array.

```
   for $ct (0 .. 63)
       {
       push (@Rounded, int($Zigged[$ct] / ($qt->[$ct] + 0.5)));
       }

   undef $qt;
   undef @Zigged;
```

Here we take the DC-Coeffecient off of the @Rounded array so that we can encode The AC-components seperately.

```
   @RunValues = (shift (@Rounded));
   @RunZeroes = 0;
```

Zero-Runlength Encoding of the AC-Coeffecients. $zct is the zero counter, which is used to keep track of the amount of preceding zeroes for each value.

```
   $zct = 0;
   foreach (@Rounded)
       {
```

If the current value is zero, increment the zero counter;

```
       $zct++  if ($_ == 0);
```

If there are 16 zeroes in a row, create a RLC value of (0, 15) and reset the zero counter.

```
       if ($zct == 16)
           {
           push (@RunZeroes, 15 );
           push (@RunValues, 0 );
           $zct = 0;
           }
```

If there is a non-zero value, pass that to the @RunValues array and record the number of preceding zeroes in @Runzero. Then reset the zero counter.

```
       if ($_ != 0)
           {
           push (@RunZeroes, $zct);
           push (@RunValues, $_);
           $zct = 0;
           }
       }
```

```
undef $zct;
undef @Rounded;
```

Here we encode the run-values in the binary representation used by the JPEG format.

```
foreach (@RunValues)
    {
```

If the value is negative, make it positive and change the value $IsNeg to 1, so that we can remember if it's negative.

```
    $val = ($_ * -1) and $IsNeg = 1 if ($_ < 0);
    $val = $_ and $IsNeg = 0 if ($_ > 0);
```

Here we convert tha value in $_ into binary by using pack and unpack. By wrapping the entire process in int() any leading zeroes that aren't required are removed.

```
    $_ = int(unpack(“B*”, pack(“n”, $val)));
```

If the value is a negative value, we flip the bits. This method of bit-inversion is string based, not numeric. In an functional piece of code, this would be done in a numeric manner for sake of effeciency.

```
    s/0/a/g if ($IsNeg == 1);
    s/1/b/g if ($IsNeg == 1);
    s/a/1/g if ($IsNeg == 1);
    s/b/0/g if ($IsNeg == 1);
```

Record the minimum number of bits required to encode the value, and record it in @RunCategory.

```
    for ($ct = 0; $ct != 16; $ct++)
        {
        $category = $ct if /(\w{$ct,})/;
        }
    push (@RunCategory, $category);
    }
```

Here we create the huffman value of the DC and AC components.

```
$DcHuff = join ‘’, unshift(@RunZeroes), unshift(@RunCategory);
for $ct (0 .. $#RunZeroes)
    {
    push (@AcHuff, join ‘’, $RunZeroes[$ct], $RunCategory[$ct]);
    }
```

This returns the image data and huffman values of the AC and DC components, as well as a count of the number of huffman codes.

```
return ($DcHuff, $DcCoeff, \@AcHuff, \@RunValues, $#RunZeroes);
}
```

```
sub Parser
    {
```

All the data from the encoder subroutine is passed to this one, as well as a value indicating the type of data (Y = 0 Cb = 1 Cr = 2). Also passes the Block ID tag.

```
($DcHuff, $DcCoeff, $AcHuff, $AcCoeff, $Count, $Type, $Block) = @_;

for $ct (0 .. $Count)
    {
```

The Huffman value and Coeffecient value are recorded in the a nested hash which references an hash.

```
    $data[$Type][$Block][$ct]->{huff} = $AcHuff->[$ct];
    $data[$Type][$Block][$ct]->{coeff} = $AcCoeff->[$ct];
```

Build Frequency tables for the Huffman Encoding Algorithm.

```
if ($HuffCount[$Type]->{$AcHuff->[$ct]})
    {
    $HuffCount[$Type]->{$AcHuff->[$ct]}++ ;
    }
else
    {
    $HuffCount[$Type]->{$AcHuff->[$ct]} = "1";
    }
}
```

Store the DC information and return the number of ac components for that specific block.

```
$data[$Type][$Block][0]->{dchuff} = $DcHuff;
$data[$Type][$Block][0]->{dccoeff} = $DcCoeff;
return ($Count);
}
```