Décompression des clichés JPEG

J'ai essayé de rassembler ici des explications sur le décodage des JPEG (du type de compression le plus fréquent), que l'on trouve surtout en langue anglaise, et pas forcément limpides… Ceci pour accompagner l'unité dyJPEG.pas que j'ai adaptée de LoadJPEG.cpp écrite en langage C. J'aimerais essayer de rendre plus véloce la décompression en Delphi !

Avertissements *:*

- *Seuls sont traités ici les fichiers JPEG reposant sur la compression séquentielle DCT avec ligne de base, et utilisant le codage couleur à 3 composantes YUV (YcbCr) entrelacées.*
- *Les nombres binaires sont exprimés en base hexadécimale sous la forme : $FFF et en notation binaire sous la forme #111.*
- *Le terme «mot» écrit en italique désigne des valeurs entières non signées codées sur 2 octets (Word, 16 bits). Byte désigne un entier non signé sur 8 bits. ShortInt et SmallInt désignent respectivement des entiers signés sur 8 et 16 bits.*

INTRODUCTION

Quelques notions de base sur le **format JPEG** (Joint Photographic Experts Group), tout d'abord :

- c'est un standard reposant sur la recommandation T.81 du CCIT (18 septembre 1992), reprise à l'identique comme norme ISO 10918-1 ;
- il utilise l'alignement Motorola = Big Endian (stockage des octets du poids le plus fort au plus faible, c'est-à-dire dans le même sens que leur valeur) ;
- un fichier JPEG contient plusieurs **sections** repérées par des **marqueurs** : ce sont des *mots* qui commencent tous par $FF ;
- il débute toujours par un marqueur **SOI** (Start Of Image = début d'image) valant **$FFD8** ;
- l'image compressée suit de peu le marqueur **SOS** (Start Of Scan=header+ ECS) valant **$FFDA** (ECS = Entropy Coded Stream = flux compressé).
- et il se termine toujours par un marqueur **EOI** (End Of Image = fin d'image) valant **$FFD9**.

A l'exception de ces trois marqueurs, chaque section d'un fichier JPEG obéit à une structure générale : le marqueur est immédiatement suivi d'un *mot* qui indique le nombre d'octets de la section. Les données d'une section sont donc limitées à 65533 octets, leur taille étant incluse dedans.

Ces sections renferment des données relatives à l'image et son codage, et des outils permettant sa décompression : tables de quantification, et tables de Huffman, à la base du décodage.

Grâce à des **marqueurs d'applications** (Application Markers) définis de **APP0 ($FFE0) à APP15 ($FFEF)**, certains clichés ajoutent à l'image elle-même des **méta-données** qui lui sont relatives, utilisables par les applications les lisant (vignette, et/ou données techniques). Ces marqueurs APPX suivent immédiatement le marqueur SOI, précédant donc le reste de l'image elle-même.

Le **format JFIF** (Jpeg File Interchange Format – Independant JPEG Group), fixe, utilise le marqueur **APP0 ($FFE0)**.

Il a été supplanté par **le format EXIF** (EXchangeable Image file Format – Japanese Electronic Industry Development Association), qui **utilise le marqueur APP1 ($FFE1)**.

Pour une explication sur ces formats, voir notamment mon tutoriel et l'unité dyExif.pas, disponibles sur ce site.

Avant d'aborder la décompression, il nous faudra examiner de façon simplifiée la logique de la compression, ses principes et l'ordre des opérations. Retroussons donc nos manches…

PRINCIPES DE LA COMPRESSION JPEG

La compression JPEG par **DCT** (Discrete Cosine Transform) séquentielle avec ligne de base fait appel à plusieurs niveaux de réduction des informations à coder. C'est un processus avec pertes, certains détails étant éliminés, en fonction de paramètres qui influencent la qualité et le niveau de la compression. Elle aboutit à coder par blocs de 8x8 valeurs appelés **DU** (Data Units).

- Sous-échantillonnage :

La première étape transforme les pixels codés par 3 canaux colorés RGB en 3 composantes YUV : la première (Y) code la luminance, et les 2 autres (U=Cb et V=Cr) la chrominance (luminance moins bleu ou rouge). Ceci est réversible et repose sur un simple calcul de matrice.
Ce changement d'espace de couleur se justifie par le fait que l'œil est beaucoup plus sensible à la luminance qu'aux couleurs : la rétine renferme en effet bien plus de cellules-bâtonnets, très sensibles à la lumière, que de cellules-cônes, qui permettent la vision colorée.
La première compression tient compte de cette imperfection pour coder généralement la luminosité de chaque pixel, mais seulement la chrominance moyenne de blocs de 2 ou 4 pixels adjacents.
Il s'ensuit qu'une DU de chrominance (64 valeurs) va coder pour un bloc de 8x16 ou 16x8 ou 16x16 pixels, tandis qu'une DU de luminance n'en codera que 8x8. D'où la notion de **MCU** (Minimum Coded Unit) ou portion d'image minimale, qui regroupe les DU nécessaires à la décompression des 3 composantes : par exemple, YYYYUUV si la composante Y est enregistrée pour chaque pixel, la U comme moyenne de 2 pixels, et la V pour 4. La MCU fait ici 16x16 pixels.
Nota : les DU balayent l'image de départ de haut en bas et gauche à droite. Et dans le mode - dit **entrelacé** - qui nous intéresse, les composantes se succèdent dans l'ordre YUV, dans le même *scan*.
Les valeurs des différentes composantes sont entières, codées de 0 à 255 sur un octet (*Byte*). On leur soustrait 128 (normalisation) pour obtenir une valeur signée de –128 à 127, toujours sur un octet (*ShortInt*). L'intérêt est de faire tendre le maximum de valeurs vers 0 lors des manipulations suivantes.

- Transformée Cosinus Discrète :

Chaque DU de chaque composante subit une sorte de transformée de Fourier à 2 dimensions, destinée à remplacer leurs valeurs par les amplitudes de fréquences spatiales correspondant aux détails de l'image : plus les détails sont fins (différences minimes entre pixels), plus ils engendrent des fréquences élevées.
Dans cette DU, de 8x8, la valeur en haut à gauche représente la fréquence la plus faible, appelée terme **DC** (Direct Coefficient), proportionnel à la moyenne des valeurs des pixels de départ. Les autres coefficients sont nommés termes **AC** (Alternative Coefficients), et représentent les détails.

- Quantification :

Elle va éliminer les amplitudes les plus élevées par 2 mécanismes :
- ré-arrangement des valeurs en zig-zag pour les ordonner selon l'amplitude (plus une valeur est proche du terme DC, moins elle en diffère : la couleur varie en principe peu d'un pixel à son voisin) ;
- division par un facteur défini dans la **table de quantification** de la composante : plus la fréquence est élevée, plus le facteur est grand. Ceci tire parti du seuil de perception des détails pour les atténuer, mais contribue aux pertes à la compression : 2 pixels de caractéristiques suffisamment proches pour ne pas être vus comme distincts sont codés comme s'ils étaient identiques. La table de quantification est le second élément de qualité de la compression, après le sous-échantillonnage.

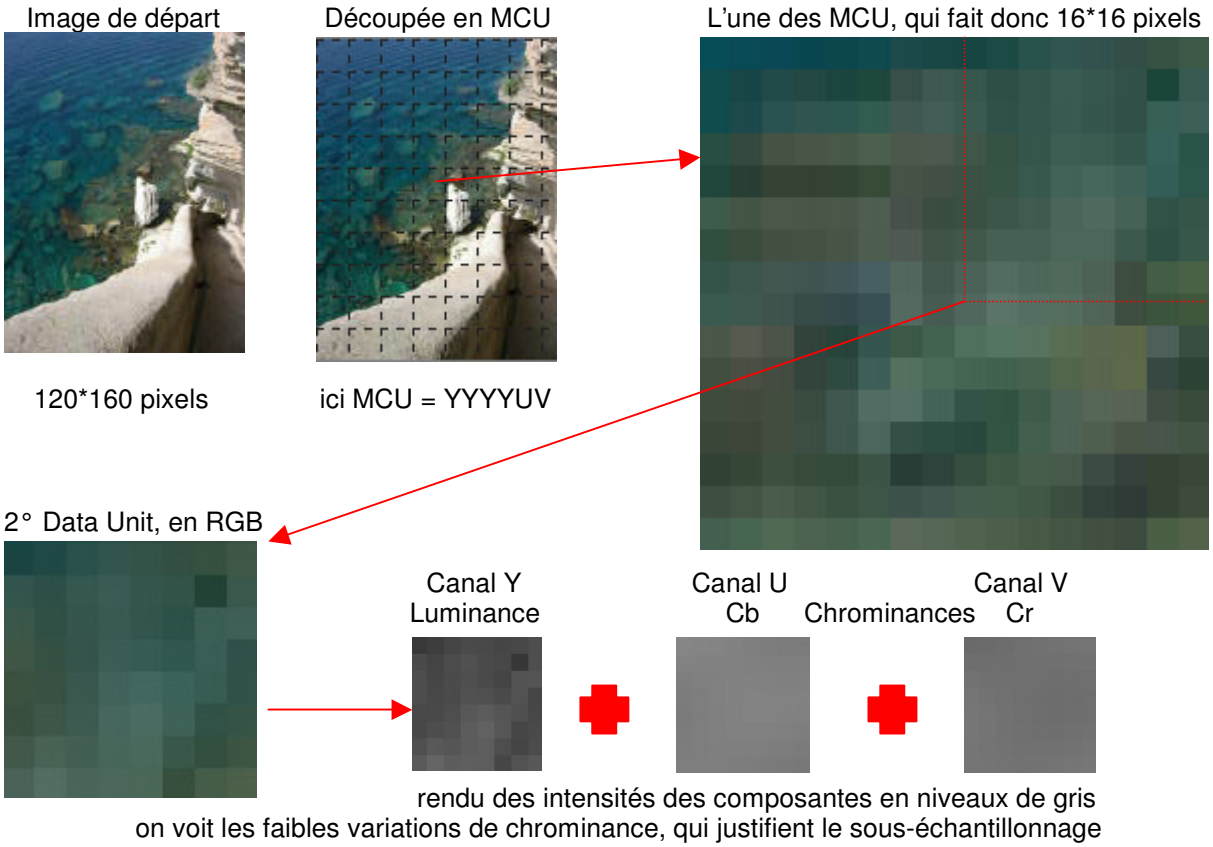
- Codage d'une DU :

C'est un tableau de 8x8 *SmallInt,* le premier étant le terme DC, suivi des coefficients AC, dont beaucoup sont nuls, du fait de la quantification.
Partant du principe que les couleurs varient lentement d'un pixel, donc d'un bloc à l'autre, la valeur du DC est codée dans la première DU de la composante, puis on ne code que ses variations dans les DU suivantes (**Differential Pulse Code Modulation** ; intérêt : plus petites valeurs à coder).
Pour les coefficients AC, on code le nombre de coefficients à zéro le précédant, avant d'en coder la valeur elle-même (**Zero Run Length Coding** ; intérêt : coder moins de valeurs, donc gagner en taille).

- Codage d'Huffman :

Il code les valeurs sur un nombre variable de bits (1 à 16), en minimisant cette taille. Aucun code n'est le début d'un plus long, ce qui en permet la lecture sans ambiguïté. Une **table d'Huffman** établit la correspondance entre les codes et les valeurs codées.
Les tables DC sont plus légères, car les valeurs codées représentent la catégorie du coefficient, c'est-à-dire la longueur minimale en bits nécessaire pour représenter sa valeur en binaire.
Les tables AC ont plus de valeurs à coder, chaque octet valant le nombre de zéros (0 à 15) sur les 4 bits de poids fort, et le nombre de bits nécessaires pour exprimer le coefficient AC (sa catégorie), sur ceux de poids faible.
Nota : le codage d'Huffman n'est donc pas aligné sur les octets. Si le flux s'interrompt avant que le dernier octet soit rempli, ajouter des #1 à due concurrence.

A titre d'exemple, voici ce qui se passe pour compresser une image BitMap en couleurs RGB codées sur 24 bits, avec un sous-échantillonnage de 1 coefficient de chrominance pour 4 pixels :



rendu des intensités des composantes en niveaux de gris
on voit les faibles variations de chrominance, qui justifient le sous-échantillonnage

Nota : les tableaux avec fond bleu-vert sont en exemple ; ceux à fond jaune valables en général.

Coefficients pour la Luminance :

58	39	52	63	72	82	98	115
54	24	22	54	55	62	78	110
60	19	39	53	58	67	94	113
70	30	49	63	73	79	96	108
73	51	66	81	95	97	103	113
86	78	79	94	106	105	104	117
103	98	100	106	109	108	111	121
118	116	117	118	119	118	121	126

Coefficients pour la Chrominance Cr :

107	102	105	105	114	118	123	127
107	98	98	99	109	113	118	123
113	101	100	99	107	110	116	122
113	105	104	109	105	108	115	123
119	114	113	106	110	114	120	123
121	116	116	113	116	117	122	125
123	119	118	117	122	122	125	127
125	124	123	124	125	127	127	128

A partir de la DU d'une composante, on soustrait 128 des coefficients, puis fait subir la DCT :

Coefficients DCT de Luminance : Coefficients DCT de Chrominance Cr :
Nota : les coefficients, réels, ont été arrondis

-568	-518	-471	-443	-450	-474	-457	-396
-431	-436	-470	-364	-423	-489	-536	-388
-417	-482	-419	-394	-429	-478	-387	-339
-396	-483	-416	-380	-379	-426	-414	-494
-444	-450	-382	-323	-272	-320	-374	-446
-431	-372	-387	-296	-230	-309	-460	-424
-376	-321	-319	-278	-285	-382	-474	-415
-302	-265	-248	-264	-277	-393	-406	-312

-175	-153	-146	-159	-119	-111	-85	-58
-123	-125	-136	-145	-112	-112	-116	-118
-94	-121	-135	-154	-129	-142	-140	-150
-104	-114	-129	-114	-161	-173	-171	-128
-79	-84	-94	-152	-149	-148	-131	-157
-79	-90	-96	-137	-128	-145	-126	-125
-84	-105	-116	-148	-102	-117	-98	-94
-89	-92	-128	-113	-87	-70	-91	-76

On ordonne ensuite les valeurs en traversant la DU en zigzag selon le tableau d'indices présenté ci-dessous (analyse de l'image en diagonale, qui réordonne les écarts par rapport au terme DC) :

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

Puis on divise chaque valeur par le coefficient correspondant de la table de quantification, qui est très généralement un (sous-)multiple de celle de référence proposée dans la norme :

Table pour la Luminance :

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

Table pour la Chrominance :
(adaptée pour sous-échantillonnage)

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

Nota : le facteur de qualité de compression que l'on ajuste entre 1 et 100 pour enregistrer une image sert à calculer un multiplicateur des coefficients de la table. Les tables sont utilisées telles qu'elles pour 50 (bonne qualité), divisées par 2 pour 75 (très bonne), multipliées par 2 pour 25 (acceptable).

Pour notre exemple, avec un facteur de qualité de 25 :

Vecteur pour la Luminance :

-18	-24	-13	-18	-10	-6	-4	-4
-20	-16	-16	-13	-8	-3	-4	-4
-15	-15	-13	-9	-5	-3	-3	-3
-14	-13	-9	-8	-4	-7	-3	-3
-9	-9	-4	-3	-2	-1	-1	-2
-9	-6	-4	-3	-3	-2	-1	-2
-3	-2	-2	-2	-2	-2	-2	-2
-3	-2	-2	-2	-2	-2	-2	-2

Vecteur pour la Chrominance Cr :

-5	-4	-1	-4	-1	-1	-1	-1
-3	-2	-2	-1	-1	-1	-1	-1
-2	-3	-1	0	0	0	0	0
-1	-1	-2	0	0	-1	-1	-1
-1	0	-1	0	0	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1
-1	0	0	0	-1	0	0	0

L'analyse et la pondération des détails de l'image en diagonale aboutissent à un « vecteur, » tableau des coefficients DC et AC représentant la DU. Codage de la variation du terme DC par rapport à la précédente DU de la composante, puis des coefficients AC. Le codage des valeurs des coefficients fait appel à un code de longueur variable, chaque valeur (de type *SmallInt*) étant codée sur le nombre de bits minimum, ce nombre de bits étant sa « **catégorie** » :

- catégorie 0 : codage sur 0 bit, le coefficient vaut donc 0 !
- catégorie n : pour un entier positif en représentation binaire sur le plus petit nombre de bits, celui de poids fort est forcément à #1 (par exemple, 5 se code en catégorie 3 : #101). Du fait que les positifs débutent par #1, la norme fixe de représenter leurs opposés dans les codes de même taille disponibles, débutant par #0… Leur stockage se fait par complément à 1 (ou négation, opérateur Not : -5 se stocke également en catégorie 3, sous la forme #010).

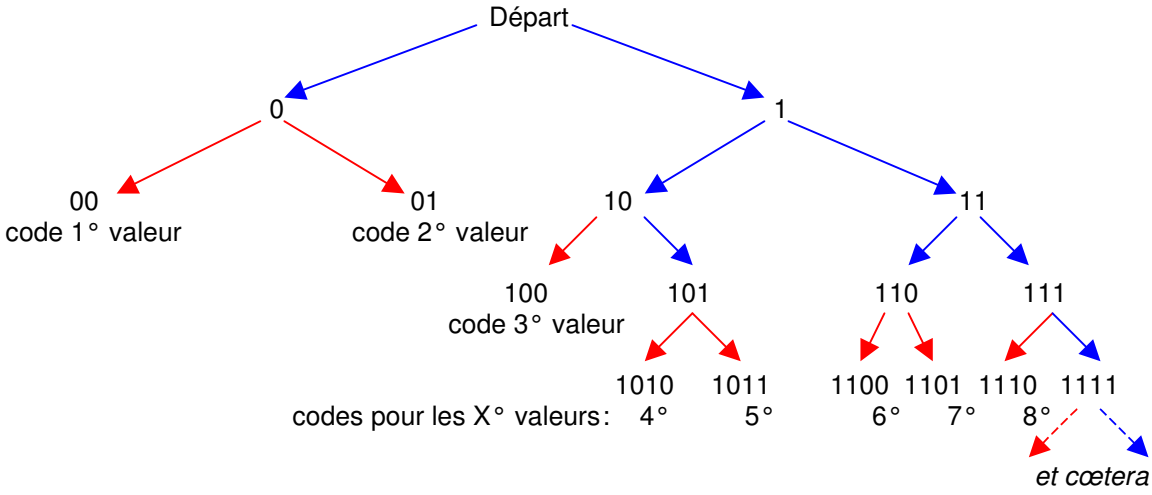
Pour le différentiel de coefficient DC, sa catégorie (nombre de 0 à 15) est codée selon Huffman, puis sa valeur (dans le nombre minimal de bits que l'on viendra donc juste de déterminer d'après l'arbre de Huffman.) Nous envisagerons ce principe de codage juste après celui des coefficients AC.

Nombre de coefficients AC sont nuls. On en tire parti pour coder selon Huffman une valeur d'un octet de long dont les 4 bits de poids fort représentent le nombre ($Z) de coefficients à 0 avant celui dont on code la catégorie ($C) dans les 4 bits de poids faible (Zero Run Length Coding : $(Z,C)). Ceci ne permet de coder que 15 zéros précédant une valeur non nulle. Pour coder une suite de 16 zéros, on utilise le code (15,0). Le code (0,0) signifie **EOB** : End Of Block : il n'y a alors plus que des coefficients nuls. Soit à coder la suite de 19 zéros avant un coefficient non-nul : (15,0) (3,C) soit 16 puis 3 zéros, avant le coefficient à lire de catégorie C.

Pour le codage de la chrominance, ici :

- Code d'Huffman de la catégorie terme DC, puis codage de sa valeur sur 3 bits.
- Codage des termes AC selon le ZRL :
  (0,-4) ; (0,-1) ; (0,-4) ; (0,-1) ; (0,-1) ; (0,-1) ; (0,-1) ; (0,-3) ; (0,-2) ; (0,-2) ; (0,-1) ; (0,-1) ;
  (0,-1) ; (0,-1) ; (0,-1) ; (0,-2) ; (0,-3) ; (0,-1) ; (5,-1) … (0,-1) ; (0,-1) ; (3,-1) ; (0,0) = EOB
  Chaque couple est représenté par le code d'Huffman correspondant au nombre de zéros précédant la catégorie de la valeur, la valeur elle-même étant codée sur le nombre de bits ainsi défini.

Si vous n'avez pas encore la migraine, passons au codage selon Huffman ! C'est un code de longueur variable (**Variable Length Code**) entre 1 et 16 bits. Les valeurs à coder les plus fréquentes sont représentées par les codes les plus courts. Pour permettre le décodage, aucun code n'est le début d'un autre (plus long, donc.) Les tables de Huffman établissent la correspondance entre code et valeur représentée. Elles résultent de l'arbre qui construit le code :



Exemple d'arbre de Huffman

Les tables pour les coefficients DC permettent de coder leur catégorie (de 0 à 15) et sont donc plus légères que celles des termes AC, qui codent pour zéros précédents et catégorie. Les coefficients sont codés successivement, dans l'ordre des DU composant les MCU, et les MCU les unes à la suite des autres.

Nota : si les dimensions de l'image ne sont pas multiples de celles de la MCU, on complète les lignes (colonnes) de chaque MCU incomplète du bord droit (inférieur) de l'image, par autant de lignes (colonnes) de son bord gauche (supérieur).

C'était la dernière étape de codage, hormis l'architecture du fichier : sections… Pour notre image test :

- `$FFD8 : SOI  pour marquer le début d'un fichier JPEG`
- `$FFDB : DQT  pour définir la table de quantification d'indice 0`
- `$FFDB : DQT  pour définir la table de quantification d'indice 1`
- `$FFC0 : SOF0 pour données images, composantes, sous-échantillonnage, et associer les tables de quantification aux canaux de couleur`
- `$FFC4 : DHT  pour définir la table de Huffman pour termes DC d'indice 0`
- `$FFC4 : DHT  pour définir la table de Huffman pour termes AC d'indice 0`
- `$FFC4 : DHT  pour définir la table de Huffman pour termes DC d'indice 1`
- `$FFC4 : DHT  pour définir la table de Huffman pour termes AC d'indice 1`
- `$FFDA : SOS  pour associer les tables de Huffman aux composantes YUV suivi d'ECS, sans marqueur : flux image`
- `$FFD9 : EOI  pour marquer la fin du fichier`

Nota : ici, pas de marqueurs Restart (ni de section DRI ; voir ci-après). Une même section DQT ou DHT peut coder plusieurs tables à la suite. On peut voir un autre type de section : **COM** (**$FFFE**) pour Comment : jusqu'à MaxWord-2 (65533) octets, à discrétion : commentaire en PChar par exemple.

Nous n'avons pas abordé ici le détail du codage des tables de Huffman, qui sera vu dans la partie décompression, et nous n'envisagerons pas les mécanismes qui permettent d'élaborer des tables optimisées, plutôt que d'utiliser celles proposées. En utilisant celles préconisées, la compression peut se faire en une seule passe, à l'instar de la décompression, sans analyse préalable de l'image.

PRINCIPES DE LA DECOMPRESSION JPEG

La décompression se fait en ordre inverse, après avoir lu les informations nécessaires, qui précèdent le flux compressé. Ces informations sont codées dans des sections identifiées par des marqueurs :

- **$FFDB : DQT** (Define Quantization Table) pour les tables de quantification ;
- **$FFCO : SOF0** (Start Of Frame Zero) pour les données de base du type de JPEG auquel nous nous intéressons ici : dimensions de l'image, nombre et caractéristiques des composantes (YUV, sous-échantillonnage, indices des tables de quantification)
- **$FFC4 : DHT** (Define Huffman Table) pour les tables de Huffman ;
- **$FFDA : SOS** (Start Of Scan) pour la liaison des tables de Huffman aux composantes, puis le flux compressé **ECS** (Entropy Coded Stream).

Ce flux peut être divisé en segments égaux par des marqueurs Restart. La section **DRI** (Define Restart Interval ; marqueur **$FFDD**) indique le nombre de MCU par segment. La fin d'un segment est matérialisée par un des marqueurs **$FFD0 (RST0)** à **$FFD7 (RST7)** qui se succèdent en boucle, dans cet ordre (Restart 0 à 7, sauf pour le dernier segment, terminé par EOI : End Of Image).
Ceci peut être utile pour se repérer dans l'image, en cas d'erreur, notamment lors de télétransmission de cliché.
Le marqueur Restart vient interrompre le flux, à la fin d'une MCU qui n'est pas forcément alignée sur un octet. Les bits de faible poids sont alors mis à #1, et la MCU suivante initie le nouveau segment, après le marqueur Restart.

- Lecture du fichier :

On s'assure d'avoir réellement affaire à un JPEG s'il commence bien par SOI ($FFD8).
On va ensuite chercher les marqueurs, et retenir ceux qui nous intéressent.
Si l'on rencontre une section inconnue, il suffit de sauter sa longueur (la longueur à sauter est celle contenue dans le *mot* suivant, diminuée de la longueur de ce *mot,* soit 2 octets). Attention : alignement Motorola (Big Endian), inverse de celui de Delphi.
En fin de flux d'image compressé (ECS), on vérifiera que le marqueur EOI ($FFD9) clôt bien le fichier.

- Lecture des sections :

Pas de difficulté ; leurs structures sont fixées par la norme JPEG. Pour notre cas, il s'agit de ne décoder que des fichiers à compression séquentielle DCT avec ligne de base, utilisant le codage couleur à 3 composantes YUV (YcbCr) entrelacées.
On doit donc trouver une section SOF0 qui signe le type de compression, donne la taille de l'image, la précision du codage des couleurs (classiquement 8 bits par canal, parfois 12), des informations sur ces canaux : nombre, nature, sous-échantillonnage (nombre de DU horizontales et verticales par MCU), et correspondance avec les tables de quantification précédemment codées.
On a ensuite les définitions des tables de Huffman, dont nous étudierons plus loin le codage et l'interprétation.
Finalement, la section SOS, qui associe les tables de Huffman aux composantes et précise des données qui ne sont utiles qu'aux JPEG progressifs… Immédiatement après, le flux image ECS.

- Tables de Huffman :

Leur structure est triviale : numéro et type (pour décodage de termes DC ou AC) en tête, puis nombre de codes de chaque taille (entre 0 et 15 bits), et finalement valeurs codées, dans l'ordre des codes.

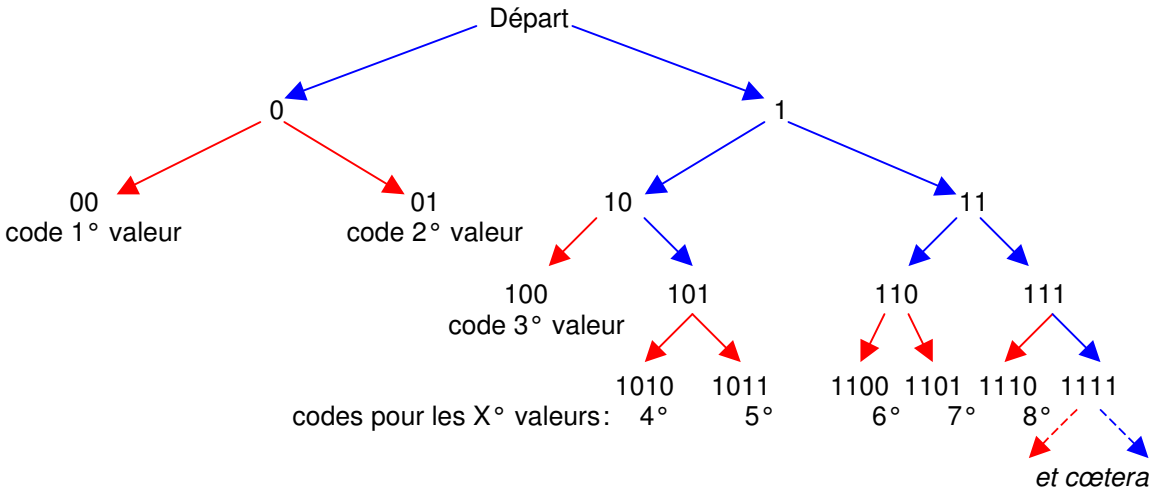
A charge pour nous de reconstituer l'arbre de Huffman pour en déduire à quel code correspond quelle valeur. Nous en savons assez, puisque nous nous souvenons qu'aucun code n'est le début d'un autre ! Reste à trouver la méthode qui permet de l'édifier et de le parcourir.

On va ajouter par commodité un champ code au tableau représentant la table.

Nous allons balayer les codes, dans l'ordre de longueur croissante. Dans l'exemple repris ci-dessous, il n'y a pas de code de 1 bit, mais 2 de 2 bits, 1 de 3 bits, 5 de 4 bits…

Le code est initialisé à 0.
La taille du code est initialisée à 0.
On attribue au fur et à mesure les codes de cette taille et incrémente pour avoir le prochain code.
Si on a atteint le dernier code de la taille considérée, on multiplie cette racine par 2 et on passe à la taille suivante.

Au final, on a donc parcouru l'arbre et reconnu ses feuilles car elles sont rangées au plus compact.



Exemple d'arbre de Huffman

L'arbre a été établi de manière statistique, de manière à associer les plus petits codes aux valeurs les plus souvent rencontrées. Souvenons-nous que les valeurs codées vont de :
- $00 à $0F pour les termes DC (catégorie) => gain en taille par rapport au codage d'octets ;
- $00 à $FF pour les termes AC (ZRL) => gain en moyenne, les codes pouvant être plus longs que l'octet, mais seulement pour les valeurs les plus rarement rencontrées.

- Décodage du flux :

Il va s'effectuer MCU après MCU, composante après composante, DU après DU. On comprend rapidement qu'une des difficultés tient au codage réalisé sur un nombre minimal et variable de bits, bousculant la barrière des octets. C'est la rançon de la compression !

Chaque DU débute par le terme DC. On doit d'abord lire sa catégorie, grâce à la table d'Huffman DC du canal. Pour cela, pas d'autre moyen que de lire bit à bit et chercher si le code correspond à une valeur dans l'arbre.

Le plus simple pour ceci est de faire un tableau des valeurs numériques des codes eux-mêmes, à parcourir au fur et à mesure des tailles de codes. La recherche est de plus en plus longue pour les codes des valeurs les plus rares : c'est le tribut à Huffman !

Une fois la catégorie décodée, reste à lire la valeur du coefficient, représentant un entier signé dans le nombre de bits spécifié. Ceci nous oblige à une « traduction » pour stocker la valeur dans un *SmallInt,* de taille fixe, le problème tenant aussi à la différence de représentation des valeurs négatives (nous ferons un rappel sur la représentation usuelle des entiers relatifs).

Le codage des entiers sur le nombre de bits minimal retenu pour le JPEG obéit à un mécanisme particulier :

- un positif est codé sans les bits de poids fort à #0 du *SmallInt*, et commence donc par #1 ; pour la valeur 87, catégorie 7, donc codage sur 7 bits : #1010111
- un négatif est codé comme le complément à 1 de son opposé et commence donc par #0 ; pour la valeur -87, codage sur 7 bits également (catégorie 7), par négation, donc : #0101000

Représentation des entiers signés binaires en complément à 2 :

C'est celle utilisée pour le codage classique des entiers relatifs, notamment sous forme de *SmallInt*, dans laquelle 87 est ainsi représenté : #0000000001010111

La représentation d'un nombre négatif s'obtient par :

- la traduction du nombre positif : (87 pour -87) #0000000001010111
- la négation de ses bits (complément à 1) : #1111111110101000
- l'ajout de la valeur 1 (complément à 2) : (soit -87) #1111111110101001

L'intérêt du complément à 2 réside dans le fait que l'addition d'un nombre et de son opposé donne zéro (ça tombe bien !) En effet, l'addition d'un nombre et de son complément à 1 donne un nombre dont tous les bits sont à 1. En lui ajoutant 1 (le fameux complément à 2), on annule donc tous les bits en les complétant à #10, ce qui propage une retenue.

Pour passer d'un entier sur n bits à un *SmallInt* de 16 bits, il faut:

- lire le bit de poids fort (#1 si positif, #0 si négatif) ;
- répliquer son inverse dans les 16-n bits de poids fort du *SmallInt* ;
- ajouter #1 s'il est négatif, pour passer du complément à 1 au complément à 2.

C'est une petite gymnastique bien astreignante, les octets se manipulant plus facilement que les bits !

Le coefficient DC étant stocké sous forme différentielle, il s'agit de l'ajouter à sa précédente valeur. On se souvient que le flux peut être interrompu à intervalles réguliers par des marqueurs Restart qui recalent le cadre de lecture en cours sur le début de l'octet suivant. Dans ce cas, la valeur du coefficient DC est codée dans la première DU du nouvel intervalle, et pas seulement son différentiel.

Une fois décodé, au tour des termes AC : grâce à la table d'Huffman correspondante, le code obtenu par lecture bit à bit livre un octet dont les 4 bits de poids fort contiennent le nombre de zéros précédant le coefficient, et les 4 bits de poids faible la catégorie du coefficient. On pourra donc le lire, comme précédemment le coefficient DC.
Nota : attention aux valeurs spéciales du ZRL : (15,0) signifie 16 zéros consécutifs, et (0,0) = End Of Block (plus que des zéros dans la DU).

Une fois la DU obtenue, on réordonne selon l'analyse en zigzag, et on déquantifie en multipliant par les coefficients de la table de quantification correspondante.

On applique la **Transformée Cosinus Discrète Inverse** (IDCT, voir ci-après) qui renvoie des *ShortInt* auxquels on ajoute 128 pour retrouver les *Byte* du canal de couleur…

Il suffit d'écrire la valeur dans le tampon image (au format tableau de triplets YUV), en tenant compte du sous-échantillonnage : elle peut être la moyenne de plusieurs pixels, et donc nécessiter d'être attribuée à chacun.

En fin de MCU (ou en fin d'image), on peut convertir les triplets YUV en triplets RGB par simple calcul.

En dernier lieu, on recopiera l'image dans un TBitMap aux dimensions de l'image et au format pf24Bit.

Enfin, on vérifiera que le flux dont on vient de terminer le décodage est bien suivi du marqueur de fin d'image EOI.

- IDCT :

La Transformée Cosinus Discrète Inverse est une opération fondamentale, qui permet de revenir aux valeurs de départ à partir des fréquences (coefficients AC) de leurs variations par rapport à la moyenne (coefficient DC). Cela fait appel à des calculs en boucle, complexes, dans les 2 dimensions de la DU, en une ou deux passes. La transformée à 2 dimensions est en effet très généralement pratiquée comme l'application successive d'une transformée à 1 dimension, sur les lignes, puis sur les colonnes de la DU.
L'implémentation classique de ce calcul s'applique à des réels. Un algorithme particulièrement performant est celui de Arai, Agui et Nakajima (**AA&N**'s algorithm), qui limite les multiplications, coûteuses en temps de calcul. Celui de Loeffler, Ligtenberg et Moschytz (**LL&M**'s algorithm) travaille sur des entiers, le passage aux réels s'effectuant par multiplication initiale et division finale. Il est plus rapide et reste très précis. On peut adapter l'AA&N's algorithm aux calculs sur entiers, de la même manière, mais en perdant en précision.

Un décodeur JPEG doit donc trouver à ce niveau un compromis entre précision et rapidité.

CONCLUSION

Ici se termine notre aperçu des principes de décodage du type le plus courant de fichier JPEG. Cette explication est plus particulièrement destinée à aider à la compréhension de l'unité dyJPEG.pas (disponible sur ce site) que j'ai adaptée en Pascal à partir de l'unité LoadJPEG.cpp, en C.

La norme JPEG impose un vocabulaire, une structure de données et des techniques de compression nécessitant une phase de familiarisation avant de s'y confronter. J'espère être parvenu à les présenter de manière à peu près intelligible.

Mon but est désormais d'essayer de rendre le décodage plus rapide, en améliorant ce code. Borland® est plus véloce, avec une adaptation de la LibJPEG de l'IJG. Intel® également, avec sa librairie ijl : 4 fois plus rapide, mais question processeurs, ils en connaissent un rayon…

Yves LEMAIRE
alias tourlourou

Grand Merci à tous les relecteurs, notamment René Kinzinger *alias Kr85*,
pour leur patience et leurs précieuses remarques.

Merci de signaler erreurs et omissions par mail : delphyves@wanadoo.fr

En dehors des spécifications, quelques articles, sites, programmes, ou codes précieux : merci à

- Cristi Cuturicu pour son article "A note about the JPEG decoding algorithm"
- Calvin Hass pour ses tutoriels et JPEGsnoop.exe (ImpulseAdventure.com)
- shenron666 pour son unité LoadJPEG.cpp (CodeS-SourceS.com)
- Thomas G. Lane et l'IJG pour LibJpeg (ijg.org)
- Patrick J. Van Fleet pour approfondir les DCT (dev.whydomath.org)