ROUSSILLON Cyril **TS3**

TPE 2001 - 2002

Nous avons choisi avec Jean-Baptiste DODANE comme sujet les **Les formats graphiques de compression**, en rapport avec le thème **Image**.

Planning

13/09/01 - Choix du sujet : le format informatique de compression d'images JPEG.
- Premières recherches générales sur Internet.

20/09/01 - Implémentation en C++ de la Transformée en Cosinus Discrète et de son inverse.

27/09/01 - Etude de la structure des images informatiques.

- Implémentation de la conversion RGB / Luminance - Chrominance.

04/10/01 - Etude de la théorie du codage de Huffman.

11/10/01 - Implémentation en C++ du codage de Huffman.

18/10/01 - Mise au point du contenu de nos TPE

25/10/01 - Traduction du programme en Turbo Pascal pour Delphi
- Découverte de la méthode du codage prédictif

01/11/01 - / (férié)

08/11/01 - Etude structure des fichiers .BMP
- Ecriture d'une fonction avec Delphi lisant un fichier BMP
- Implémentation du codage prédictif

15/11/01 - Etude des effets du sous échantillonnage de la chrominance ou luminance.

22/11/01 - Recherche sur Internet de la méthode de compression pour les pixels du bord de l'image dans le jpeg.

29/11/01 - / (Olympiades de la Physique)

06/12/01 - / (Forum du Haut-Doubs)

13/12/01 - Vérification expérimentale des fondements du jpeg

20/12/01 - Découverte d'une méthode pour compresser les pixels du bord de l'image

10/01/02 - / (passage répétition exposé)

17/01/02 - Correction des fonctions pour lire les fichiers BMP
- Implémentation de la gestion des pixels du bord en jpeg

24/01/02 - Organisation du logiciel et des sources

31/01/02 - Implémentation EQM

21/02/02 - Recherche d'applications de la compression graphique dans le domaine des images satellites

13/09

J’ai commencé les premières recherches sur Internet sur le format de compression JPEG, puisque j’ai décidé de faire ma partie dessus. J’ai trouvé à l’adresse *http://perso.libertysurf.fr/IPhilGood/Codage/Compressionjpeg.htm* une page expliquant le principe de la compression JPEG en plusieurs étapes :

- Préparation : convertir les donnée RGB en Luminance/Chrominance (YIQ) suivant les formules :
Y = 0.30 R + 0.59 G + 0.11 B
I = 0.60 R - 0.28 G - 0.32 B
Q = 0.21 R - 0.52 G + 0.31 B
On peut ensuite prendre pour les valeurs I et Q (Chrominance) la moyenne de quatre pixels car les yeux sont moins sensibles aux écarts de couleurs qu'aux différences d'intensités lumineuses.

- La transformée de cosinus discrète (DCT) : clé du processus de compression, semblable à la FFT. Elle est effectuée sur une matrice 8x8 de valeurs de pixels, et elle donne une matrice 8x8 de coefficients de fréquence.

- Quantification : on diminue la précision dans les hautes fréquences car l’œil y est beaucoup moins sensible.

- Remplacement : on remplace chaque élément par sa différence avec le précédent

- Linéarisation : on linéarise la matrice de 8x8 pour avoir une suite de 64 nombres, on suit une méthode dite de zigzag car les plus basses fréquences donc les plus importantes se situent en haut à gauche et on aura de longues suites de zéros à la fin.

A l’adresse *http://saturn.umh.ac.be/~olivier/CompressionInformatique/node41.html* j’ai ensuite appris que l’on effectuait un codage RLE (le plus simple des codages de compression : remplacer une suite de mêmes nombres par une partie indiquant le nombre d’itération puis le nombre lui-même, même principe que remplacer des additions par une multiplication) puis un codage statistique (type Huffman : le but est de coder avec le moins de bits possible les caractères les plus fréquents).

20/09

Calcul de la DCT :

Les formules mathématiques de la DCT et de son inverse étaient données sur le site *http://saturn.umh.ac.be/~olivier/CompressionInformatique/node41.html* et j'ai commencé à l'implémenter (voir feuille DCT). J'ai d'abord essayé de la coder en pascal (avec Borland Delphi) mais je me suis vite rendu compte de ses limitations quant à la gestion dynamique de la mémoire (impossibilité de transmettre des tableaux multidimensionnels que ce soit par référence ou par valeur, statiques ou ouverts à une fonction). Il aurait été tout de même possible de réaliser cette fonction en pascal (en déclarant les matrices en variables globales) mais en commençant à réfléchir sur l'algorithme de Huffman je me suis rendu à l'évidence qu'il nécessitait une gestion dynamique de la mémoire et des pointeurs très poussée. C'est pourquoi j'ai choisi le C++ (avec Borland C++Builder), étant donné qu'il répond à ces nécessités et que je maîtrise déjà ce langage.

J'ai optimisé ces fonctions (ou plutôt procédures au sens stricte du terme puisqu'elles ne renvoient aucune valeur) pour être utilisées pour la compression JPEG puisque la taille des tableaux est fixée à 8x8. Les matrices sont transmises par adresse (en C++ tous les tableaux sont automatiquement transmis par référence, jamais par valeur).
La fonction DCT : j'ai commencé par définir la constante Pi (elle n'est pas disponible dans C++Builder) puis le coefficient 1/V(2) afin de gagner du temps de calcul (3-4). Ensuite c'est le calcul proprement dit, les deux premières boucles imbriquées (6) servent à traiter tous les points de la matrice (i et j dans la formule). On calcule ensuite les coefficients c(i) et c(j) de la formule (8-10), puis deux variables *Fast* (11-12) dont les valeurs ne changent pas en fonction de x et y et donc permettent de gagner du temps de calcul (autrement on ferait ces calculs 64 fois pour rien). La variable *Temp* en tant que flottant permet d'éviter les erreurs d'arrondis cumulés, et les deux boucles imbriquées suivantes (14) correspondent aux sigmas x et y. On finit en multipliant cette somme par les coefficients devant les sigmas dans la formule et en enregistrant le résultat (16).
La fonction IDCT (DCT inverse) : elle repose sur le même principe en étant adaptée à la formule IDCT.

J'ai ensuite testé ces fonctions en vérifiant qu'elles donnaient bien les mêmes résultats que l'exemple donné sur le site.

27/09

Les images informatiques :

Il existe deux modes de stockages des images en informatique :
- le bitmap : on échantillonne l'image en points auxquels on associe une couleur ce qui forme un pixel. Cette couleur peut être stockée sur 1 bit (noir et blanc), 4 (16 couleurs), 8 (256 couleurs), 24 (couleurs vraies RGB environ 16 millions de couleur) et même plus. Ce mode est utilisé par la plupart des formats.
- le vectoriel ou métafichiers : l'image est stockée sous forme d'une définition mathématique (plus précisément des instructions GDI Windows). Par exemple un cercle est stocké sous la forme d'une instruction qui définit sa taille, sa position, sa couleur etc. L'avantage est que les fichiers sont généralement plus petits, et que l'on peut zoomer dessus en améliorant la précision mais en revanche il n'est adapté qu'aux schémas et logos.

Conversions RGB / Luminance chrominance :

Pour stocker les images en couleurs réelles on stocke la couleur de chaque pixel sur trois octets : un pour la quantité de rouge, un pour le vert et un pour le bleu. Cependant le modèle Luminance / Chrominance est plus adapté à l'interprétation des couleurs par notre œil. En effet notre œil est moins sensible aux écarts de couleurs (chrominance) qu'aux différences d'intensité lumineuse (Luminance) c'est pourquoi dans le JPEG on va pouvoir limiter la précision de la chrominance.
J'ai trouvé sur *http://perso.libertysurf.fr/IPhilGood/Codage/Compressionjpeg.htm* les conversions de RGB en Luminance (Y) / Chrominance (I et Q) qui sont des simples combinaisons linéaires des intensités de rouge (R), vert (G) et bleu (B) :

Y = 0.30 R + 0.59 G + 0.11 B
I = 0.60 R - 0.28 G - 0.32 B
Q = 0.21 R - 0.52 G + 0.31 B

On constate que pour Y la somme des coefficients est égale à 1 (donc quand toutes les composantes sont au maximum la luminance est au maximum : logique) et que nos yeux doivent être plus sensibles au vert puisqu'il a le plus grand coefficient. Pour I et Q la somme des coefficients est nulle (donc quand les composantes sont identiques la chrominance est nulle : logique). Les deux variables pour la chrominance servent certainement pour permettre de résoudre le système en proposant trois équations.

En revanche pour l'opération inverse je n'ai rien trouvé. J'ai donc pensé avoir à résoudre un système de trois équations à trois inconnues pour chaque conversion :

$$\begin{cases} Y = 0.30R + 0.59G + 0.11B \\ I = 0.60R - 0.28G - 0.32B \\ Q = 0.21R - 0.52G + 0.31B \end{cases}$$

Mais étant donné que les coefficients sont constants on peut commencer à le résoudre :

$$\begin{cases} Y = 0.300R + 0.590G + 0.110B \\ 2Y - I = 1.460G - 0.540B \\ 0.7Y - Q = 0.933G - 0.233B \end{cases} \qquad \begin{cases} Y = 0.300R + 0.590G + 0.110B \\ 2Y - I = 1.460G - 0.540B \\ 0.844Y - 0.933I + 1.46Q = 0.844B \end{cases}$$

Il suffit donc ensuite de calculer successivement B, G et R puisque l'on connaît Y, I et Q.

04/10

Théorie du codage de Huffman :

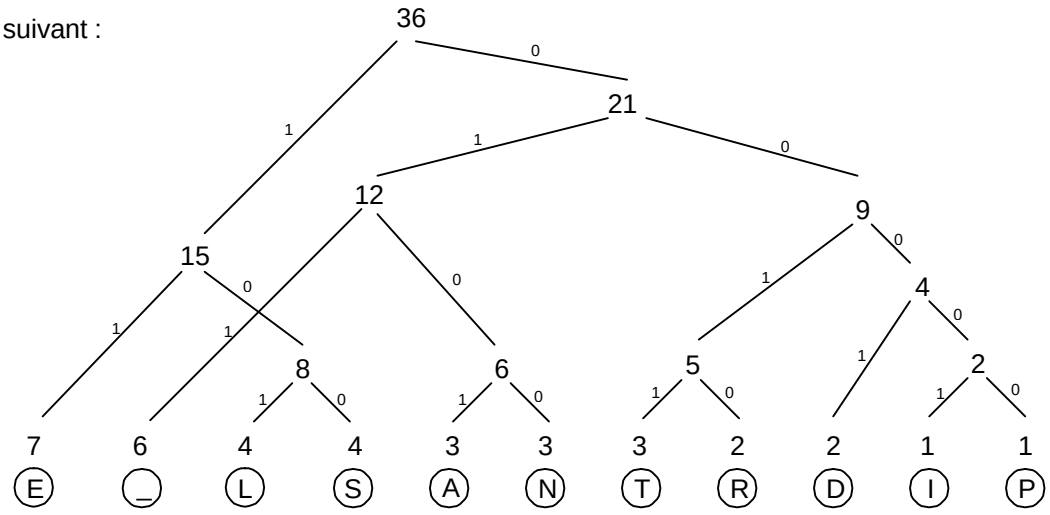
Le codage de Huffman est un **VLC préfixé** : c'est un codage statistique ou entropique, c'est à dire qu'il se base sur la fréquence d'apparition d'un caractère pour le coder : plus le caractère apparaît souvent plus son code sera court et vice-versa, ce qui explique le **VLC** (**V**ariable **L**ength **C**ode, code à taille variable). Mais en plus chaque code n'est le préfixe d'aucun autre, d'où **préfixé**, ce qui permet un décodage unique.
Le codage de Huffman est le plus efficace de toutes les techniques de codage statistique.

Dans la pratique, pour déterminer le code de chaque caractère on construit l'arbre de Huffman :
- On calcule la fréquence d'apparition de chaque caractère (ou poids).
- On rassemble les deux caractères de plus faible poids pour former un nœud, dont le poids est égal à la somme des poids des deux caractères qui le composent.
- On affecte la valeur 0 au caractère de plus petit poids et 1 au caractère de plus grand poids des deux.
- On recommence les deux étapes précédentes en considérant chaque nœud formé comme un caractère, jusqu'à n'avoir plus qu'un nœud.

Par exemple pour la phrase « LE PRESIDENT EST ENTRE DANS LA SALLE » on a le tableau de fréquences d'apparition suivant :

E	Espace	L	S	A	N	T	R	D	I	P
7	6	4	4	3	3	3	2	2	1	1

Et on construit l'arbre suivant :



Ainsi la lettre E la plus fréquente est codée sur 2 bits : 11, le I caractère le moins fréquent est codé sur 5 bits : 00001. Pour coder une lettre on part de celle-ci et on remonte jusqu'à la racine en relevant le « chemin », et pour décoder on part de la branche et on choisit son « chemin » en suivant les 0 ou 1 du code pour arriver à la lettre. On constate qu'ainsi on n'a pas besoin de dire quand le code est terminé puisqu'il est impossible de descendre plus bas quand on est arrivé à une feuille.

Par exemple pour cette phrase, stockée en ASCII elle ferait 288 bits alors que compressée elle n'en fait que 118.

Il existe trois variantes de cette algorithme :
- *non adaptative :* l'arbre est élaboré à partir d'une table des fréquences fixe ce qui fait qu'il est adapté à un seul type de donnée
- *semi-adaptative :* comme dans l'exemple, on lit une fois le fichier à compresser pour créer la table des fréquences. C'est la méthode la plus efficace mais elle présente l'inconvénient de nécessiter l'arbre que l'on devra introduire dans une en-tête.

adaptative : la table des fréquences est élaborée au fur et à mesure de la lecture du fichier et l'arbre est reconstruit à chaque fois. On réalise l'arbre avec les premiers caractères (qu'on ne compresse pas) et on s'en sert pour compresser les caractères suivants que l'on utilise pour réactualiser l'arbre etc. Elle est un peu moins efficace que la méthode semi-adaptative mais le résultat final est meilleur pour les petits fichiers.

11/10

En commençant à réfléchir sur la façon de modéliser l'arbre de Huffman, je me suis rendu compte que le mieux était de le modéliser en fonction de ses « nœuds ».
J'ai donc créé une structure (voir feuille « Code source en C++ de la modélisation et création de l'arbre de Huffman »), *sNoeud1* (1), l'arbre étant composé d'un tableau de 511 de ces structures. Elle est donc composée de trois pointeurs sur d'autres structures du tableau représentant les deux nœuds fils et le nœud père (3), et d'un octet (4) représentant le code du nœud, c'est à dire s'il est le 1° ou 2° fils du nœud père (voir schéma arbre semaine dernière).
Mais le stockage de 511 de ces structures représente une assez grande quantité de mémoire. En effet un pointeur est stocké sur 4 octets, ce qui fait 4 * 3 + 1 = 13 octets par structure soit 6643 octets pour l'arbre.
Cela était possible mais je me suis rendu compte qu'il était possible de les stocker avec beaucoup moins de place. En effet un pointeur repère la structure dans toute la mémoire, alors que nous n'avons besoin de la repérer que parmi 511 autres, donc 9 bits suffiraient à la place de chaque pointeur. Le langage C++ offre la possibilité de réaliser des champs binaires : on spécifie pour chaque variable le nombre de bits qu'elle doit occuper.
J'ai donc refait une structure (*sNoeud*) utilisant des champs binaires (10), qui fait désormais 4 octets (on est obligé d'arrondir à la taille d'une variable supérieure, soit un entier de 32 bits). L'arbre n'occupe plus que 511 * 4 = 2044 octets. J'ai ensuite définit une méthode (un constructeur plus précisément) qui initialise les données (11).

Je définis ensuite un tableau *Arbre* de 511 structures *sNoeud* qui représente l'arbre de Huffman (15), et un tableau Poids de 511 entiers (16) qui stockera le poids de chaque nœud (ou le nombre d'occurrences de chaque caractère pour les feuilles). Je n'ai pas intégré cette donnée à la structure *sNoeud* de façon à pouvoir supprimer le tableau quand on n'en a plus besoin et libérer ainsi la mémoire occupée.

La procédure *Donner2PlusPetits* renvoie dans le tableau *Res* les indices des deux nœuds de plus petit poids dans les *Taille* premiers éléments du tableau *Poids*.
On commence en affectant au tableau *Res* les deux premiers éléments et on déclare un pointeur *Temp* (21) qui pointera vers l'élément de *Res* de plus fort poids (22) (l'opérateur & renvoie l'adresse, que l'on affecte au pointeur).
On met ensuite une boucle pour tester tous les éléments du tableau *Poids* (23-28). On teste si le nœud concerné a déjà été traité (si il ne l'a pas été code = 3 comme initialisé sinon 0 ou 1, voir après), et si son poids est inférieur au deuxième plus petit déjà repéré (le plus grand élément de *Res* sur lequel pointe *Temp*) (24). Si tel est le cas on remplace l'élément de plus grand poids de *Res* par celui-ci (26), et on fait pointer *Temp* sur l'élément de Res de plus fort poids (27).
A la fin de la boucle on a les deux nœuds non traités de plus faible poids, et on finit en vérifiant que le premier nœud est le nœud de plus fort poids et dans le cas contraire on les intervertis, de façon à toujours avoir l'indice du nœud de plus fort poids dans le premier élément de *Res*.

La procédure *ConstruireArbre* procède comme son nom l'indique à la construction de l'arbre de Huffman du fichier contenu dans le tableau d'octets *Source* de taille *TailleS*. Elle utilise la procédure précédente *Donner2PlusPetits*.
On commence par remplir le tableau *Poids* qui contiendra les occurrences des octets du fichier source (35) dans ses 256 premiers éléments. Par exemple les occurrences de l'octet 147 sont stockés dans le 147° élément de *Poids*, celles de l'octet 18 dans le 18° élément etc. A chaque rencontre d'un octet on incrémente son occurrence dans le tableau *Poids* et à la fin on a le nombre d'apparition de chaque octet.
Pour les 256 premiers éléments de l'*Arbre* qui sont les feuilles on place dans la donnée ndFils0 la valeur de l'octet auquel il est attaché (36) car cette donnée ne sert pas pour les feuilles, et ainsi pendant la décompression on saura à quel octet on est arrivé.
On déclare ensuite le tableau *Petits* de deux entiers *short* (16bits) (37) qui récupèrera les deux nœuds de plus faible poids renvoyés par la procédure *Donner2PlusPetits*.
On a ensuite une boucle (38-48) qui construit le reste de l'*Arbre* au fur et à mesure.
On récupère les deux nœuds de plus faible poids (40) en ne cherchant que dans les nœuds déjà construits, qui seront les deux nœuds fils du nœud traité.
On affecte au poids du nœud traité la somme des poids des nœuds fils (41), puis on enregistre les nœuds fils comme tels (42-43), de même pour le nœud père (44-45), et enfin le code de chaque fils (46-47) : le fils qui a été enregistré dans la donnée *ndFils0* a la valeur 0 et celui qui a été enregistré dans la donnée *ndFils1* la valeur 1, et cela marque en même temps le fils comme déjà traité puisque la donnée *Code* ne contient plus 3.
A la fin le dernier élément est le nœud racine, qui n'a donc pas de père mais on lui affecte la valeur 511 pour le repérer.

Pour coder les données on part donc de la feuille (nœud) correspondante à l'octet, on regarde la valeur de *Code* qui fixe le premier bit, puis on passe dans son père (donnée *ndPere*) et on fait de même jusqu'à atteindre la racine (indice 510, dernier élément de *Arbre*) et on a notre code !
Pour décoder on part de la racine, si le premier bit est 0 on part dans *ndFils0* autrement dans *ndFils1*, jusqu'à atteindre une feuille (*ndFils1* = 511 car ils ont été initialisés ainsi et on n'y a pas touché comme ce sont des feuilles) dans laquelle on a dans *ndFils0* l'octet décodé !

18/10

Nous avons avec Jean-Baptiste défini plus précisément le contenu de nos TPE, maintenant que nous avons une meilleure connaissance de notre sujet.

Je vais réaliser comme prévu un programme qui compressera des images enregistrées au format BMP en jpeg. J'ai trouvé sur Internet des documents officiels sur la norme JFIF (JPEG File Interchange Format) qui est le format officiel .jpg utilisant la méthode jpeg. Il en ressort qu'il est très complexe de respecter cette norme avec tous les paramètres qu'elle exige (tables de Huffman, de quantification, etc). C'est pourquoi j'ai décidé de n'utiliser que la méthode jpeg en laissant tomber toutes les normes qui sont loin d'ailleurs d'être la partie la plus intéressante.

Nous avons donc décidé de réorienter nos tpe vers les méthodes de compression plus que vers les formats, et de créer nos propres formats (simplifiés) afin de pouvoir enregistrer des fichiers que nous compresserons avec différentes méthodes.

J'ai pris conscience en faisant les recherches sur le format JPEG que ce format n'était plus à la pointe de la l'innovation et qu'il était même sur le point d'être supplanté par des méthodes plus performantes telles les ondelettes ou les fractales. Etant parti pour traiter le jpeg, je ne pense pas avoir le temps de traiter ces méthodes, mais j'essaierai d'en expliquer le principe général et leurs performances.

Je vais ensuite après avoir terminé la théorie et la pratique du JPEG (ou peut-être avant), faire diverses expériences sur les pertes occasionnées par le jpeg. J'agirai sur des paramètres comme les coefficients de quantification pour vérifier les rapports entre le taux de compression et les dégradations des images. J'essaierai aussi de vérifier expérimentalement les bases du jpeg, c'est-à-dire notre sensibilité à la luminance, aux hautes fréquences, afin de le justifier.

Jean-Baptiste s'occupera pour sa part de méthodes conservatives, comme la rle, et essaiera aussi de les mettre en œuvre. Nous verrons ensuite à la fin selon le temps qu'il nous reste, pour étudier d'autres méthodes (le plus possible) voire les implémenter, et pouvoir ainsi comparer leurs performances.

25/10

Nous avions choisi de réaliser notre programme en C++ car ce langage offre une gestion plus poussée de la mémoire (pour les tableaux dynamiques), et que je maîtrisais dans ce langage les pointeurs qui sont indispensables pour ce projet. Mais j'ai entre temps appris à m'en servir en pascal avec Delphi, et étant donné que mon camarade avait quelques difficultés à s'habituer au C++, nous avons décidé de retraduire le programme en turbo pascal pour Delphi.
Un autre problème avec Delphi qui m'avait poussé à ne pas le choisir, était qu'il n'était pas possible de transmettre de tableaux multi dimensionnels dynamiques à des fonctions, ce qui était indispensable. J'ai découvert que cela n'était qu'à moitié vrai car il existe une alternative pour arriver au même résultat : il suffit de déclarer un type tableau bidimensionnel, et l'utiliser comme paramètre de la fonction.

Le seul principal défaut du programme Delphi par rapport à C++Builder concerne le stockage de l'arbre de Huffman. En effet Delphi ne propose pas de champs binaires (voir 11/10) donc l'arbre occupe plus de place en mémoire.

Par contre Delphi présente tout de même quelques avantages, par exemple il propose plus de constantes préenregistrées (Pi …) et de routines de traitement des nombres (arrondis…), et surtout il est plus populaire dans l'enseignement, et est installé sur les ordinateurs du lycée.

Nous avons également découvert avec Jean-Baptiste une méthode de compression qui nous avait échappée jusque-là, et qui semble simple à réaliser étant donné que j'ai déjà réalisé le codage de Huffman.
Il s'agit du codage prédictif : étant donné que des pixels proches d'une image ont de grandes chances d'être proches, on va essayer de prédire la valeur d'un pixel à l'aide de ceux l'entourant, puis on va seulement enregistrer la différence entre la prédiction et la valeur réelle. Cette différence sera généralement petite, et si ces valeurs sont petites, cela veut dire qu'elles reviennent plus souvent, et donc sont plus fréquentes et donc un codage statistique tel que le codage de Huffman sera efficace. Cette méthode est complètement conservative puisqu'elle restitue exactement la même image.

Je vais donc essayer d'implémenter cette méthode.

08/11

J’ai trouvé à l’adresse *http://compressions.multimania.com/bmp.html* la structure des fichiers bitmaps windows, le format non compressé le plus répandu, et dont je vais me servir comme point de départ pour mon programme de compression jpeg.

En-tête du fichier :

BMP_FileType	W	‘BM’
BMP_FileSize	DW	taille du fichier
BMP_Reserved	DW	toujours 0
BMP_BitMapOffset	DW	offset de l’image

En-tête du bitmap :

BMP_HeaderSize	DW	taille de l’entête en octets
BMP_Width	DW	largeur en pixels de l'image
BMP_height	DW	hauteur en pixels de l'image
BMP_Planes	W	nombre de plans utilisés : toujours 1
BMP_BitsPerPixel	W	nombre de bits par pixels
BMP_Compression	DW	méthode de compression (cf BMP_CMP_xxx)
BMP_SizeOfBitMap	DW	Taille de l’image en octets
BMP_HorzResolution	DW	résolution horizontale en pixels par mètre
BMP_VertResolution	DW	résolution verticale en pixels par mètre
BMP_ColorsUsed	DW	résolution verticale en pixels par mètre, si 0: palette entière si BitPerPixel <=8
BMP_ColosImportant	DW	nombre de couleurs importantes, masque pour les modes >8 bits par pixels

J’ai ensuite écrit une procédure pour lire un fichier bitmap et le dessiner sur le canevas d’une image, en utilisant les méthodes *FileOpen*, *FileRead*, *FileClose* (voir source et programme compilé).

J’ai également implémenté le codage prédictif, qui s’est d’ailleurs révélé assez simple à coder puisque j’avais déjà écrit le code pour Huffman.
En revanche nous avons été très surpris par les performances de cet algorithme puisqu’il concurrence les formats tels le PNG ou le TIFF qui utilisent la LZW ! D’autant plus que ce n’est qu’une première mouture, et qu’il peut certainement être amélioré.

15/11

On a vu que l’on convertissait les données RGB (Rouge Vert Bleu : un octet pour chaque couleur) en luminance / chrominance. On peut ensuite sous-échantillonner les deux octets de chrominance car nous sommes beaucoup moins sensibles aux écarts de couleur (chrominance) qu’aux écarts de luminosité (luminance).
J’ai donc voulu vérifier cela, et j’ai ajouté à la fiche de lecture du bitmap deux procédures pour sous-échantillonner la chrominance ou la luminance (voir code source et programme compilé sur disquette).

J’ai travaillé avec une image bitmap de 40 pixels de côté en 24bits.

On constate ainsi que :
- quand l’on prend pour chaque pixel la moyenne de 4, on ne voit aucune différence en taille réel avec l’original, mais lorsque l’on zoome, on constate un palissement des couleurs, et surtout au niveau des yeux du sujet qui comportaient quelques pixels bleus ont été perdus. En revanche la taille de l’image est passée de 40^2 * 3 = 4800 octets à 20^2 * 2 + 40 ^2 = 2400 octets, et a donc été divisée par 2 pour une qualité quasiment identique.
- Si l’on fait la même chose avec la luminance, on voit de gros blocs de 4*4 pixels qui sont pratiquement de la même couleur, et on a l'impression que la résolution a été divisée par deux. De plus la taille de l'image n'est passée que de 4800 à 3600 octets.
- quand on sous-échantillonne encore plus la chrominance, les couleurs palissent de plus en plus jusqu'à devenir niveaux de gris. C'est logique puisque seule la chrominance contient les informations sur la couleur.

On comprend donc pourquoi il est utile de convertir les données RGB en luminance / chrominance. En effet en gardant les données RGB, il est très difficile de réduire la taille des données sans occasionner de graves pertes. Le maximum raisonnable est de stocker un pixel sur 16bits (comme pour les écrans, 4bits pour le rouge et le bleu et 5 pour le vert), où les pertes demeurent négligeables, mais le gain n'est que de 1/3.

22/11

Je suis maintenant en possession de toutes les informations nécessaires pour pouvoir finir d'écrire mon programme de compression jpeg, sauf d'une : la technique de compression utilisée pour les pixels du bord de l'image. En effet on doit décomposer l'image en blocs carrés de 8 pixels de côté, et si la taille de l'image n'est pas multiple de 8 il reste des pixels sur le côté auxquels on ne peut appliquer de DCT.
J'ai donc recherché spécifiquement cette donnée sur Internet, mais je n'ai réussi à trouver aucune information. Tous les sites qui expliquent le fonctionnement du jpeg disent que pour les pixels sur les côtés une autre méthode est utilisée … mais ils ne disent pas laquelle.

étant donné qu'on a vu la semaine dernière que l'on pouvait facilement diviser la taille des données par 2 sans perte notable de qualité, je vais certainement utiliser cette méthode, c'est à dire sous-échantillonner les octets de chrominance. Je regarderai ensuite si cela vaut la peine de leur appliquer l'algorithme de Huffman, ou une quantification. De toute manière le taux de compression pour ces pixels sera inférieur à ceux du reste de l'image.

13/12

j’ai écrit une procédure qui permet d’appliquer la DCT sur une image, la quantifier avec des tables que l’on peut modifier à volonté dans une grille, puis la déquantifier et appliquer l’IDCT pour obtenir l’image décompressée. j’ai travaillé sur une image représentant un carré noir sur fond blanc afin de faire ressortir la moindre perte.

Quand on met 10000 dans la ligne du haut de la luminance (sauf le coefficient continu), donc on supprime ces composantes, l’image devient floue dans le sens horizontal, et quand on met 10000 dans la première colonne l’image devient floue dans le sens vertical, ce qui confirme que les axes de la DCT représentent les axes de l’image.

Quand on met 10000 dans les 5 cases en haut à gauche (sauf le coefficient continu), on constate que les zones qui étaient uniformes sont déformées alors que l’on voit toujours très nettement la rupture qui forme le contour du carré noir, ce qui montre que l’on a supprimé les fréquences basses ce qui confirme aussi la définition de la DCT. On revérifie cela en mettant 10000 dans toutes les autres cases (sauf le coefficient continu) et en remettant 1 dans les 5 en haut à gauche. En effet on constate cette fois ce sont bien les hautes fréquences qui ont été supprimées puisque les bords du carré apparaissent flous tandis que les parties uniformes sont parfaitement conservées.

Cela permet aussi de vérifier que nous sommes beaucoup plus sensibles aux basses fréquences qu’aux hautes fréquences puisque dans le 1° cas, le carré est méconnaissable alors que moins de 8% des coefficients ont été supprimés, et dans le 2° cas le carré est assez bien conservé alors que plus de 90% des coefficients ont été supprimés.

20/12

Concernant les pixels des bords de l'image dont je parlais le 22/11, j'ai enfin trouvé une méthode. en effet j'ai examiné des images compressées en JPEG par des logiciels de dessin et j'ai constaté que les bords de l'image avaient exactement les mêmes propriétés que le reste de l'image (des blocs de 8x8 incomplets se détachent) donc la méthode est semblable au reste de l'image.
en fait il est quand même possible de réaliser une dct même si la matrice n'est pas complète puisqu'il suffit de compléter les autres cases avec des 0 par exemple, et ne garder à la décompression que ceux que l'on veut. Mais il faut stocker les 64 éléments même si 8 seulement servent. Comme la dct fait ressortir les fréquences, je me suis dis qu'en périodisant la matrice d'entrée on pourrait peut être simplifier la dct.

J'ai donc fait plusieurs essais, et ça marche quand par exemple on recopie la même colonne 8 fois puisque la dct ne comporte plus que 8 valeurs non nulles, ce qui allège beaucoup le stockage. J'ai refait d'autres essais avec plusieurs colonnes, et en fait j'ai découvert que le but est de reproduire les colonnes que l'on a dans la matrice, en les retournant à chaque fois, de manière à ce qu'il n'y ait pas de discontinuité aux limites (c'est logique puisque c'est justement la définition de la dct par rapport à la transformée de fourier : on périodise le signal en retournant l'intervalle à chaque fois, de cette manière la fonction représentant le signal devient paire et sa transformée de Fourier ne comporte plus que des cosinus : la transformée en cosinus). Ainsi on peut réduire la matrice dct au même nombre que la matrice d'entrée, sauf pour 3 colonnes puisqu'on ne peut pas les reporter un nombre entier de fois. Mais en insérant deux colonnes de 0 au milieu il n'y a que 4 colonnes à stocker. Pour un nombre de colonnes supérieur à 4 ça ne marche plus, mais par contre on peut séparer la matrice en deux inférieures à 4, par exemple pour 5 une de 4 et une de 1 ce qui fera aussi 5 colonnes dct à stocker.

on peut résumer ainsi le codage, les lettres représentant les colonnes :

- 1 → AAAAAAAA → x0000000 = 1 colonne à stocker
- 2 → ABBAABBA → x000x000 = 2 colonnes à stocker
- 3 → ABCxxCBA → x0x0x0x0 = 4 colonnes à stocker
- 4 → ABCDDCBA → x0x0x0x0 = 4 colonnes à stocker
- 5 → ABCDDCBA + EEEEEEEE → x0x0x0x0 + x0000000 = 5 colonnes à stocker
- 6 → ABCDDCBA + EFFEEFFE → x0x0x0x0 + x000x000 = 6 colonnes à stocker
- 7 → ABCDEFGx → xxxxxxxx = 8 colonnes à stocker

Bien entendu les résultats sont les mêmes pour les lignes.

17/01

J’ai implémenté le codage des pixels du bord, mais j’ai eu beaucoup de mal.

tout d’abord j’ai constaté que les procédures que j’avais écrites pour lire les fichiers BMP ne fonctionnaient pas pour des fichiers de dimensions non multiples de 8. J’ai cherché un moment pourquoi, et j’ai découvert que pour écrire une ligne, il faut que le nombre total d’octets soit multiple de 4. S’il ne l’est pas, on complète avec des 0. Donc comme je n’en tenais pas compte, j’interprétais ces 0 comme les pixels de la ligne suivante et cela provoquait un décalage sur tout le reste de l’image.

J’ai donc corrigé cette erreur, et j’en ai profité pour améliorer les fonctions afin de gérer les images en 2, 4 ou 8 bits, puisqu’elle ne gérait que les images 24 bits avant.

pour les pixels du bord, j’ai également amélioré la méthode décrite la séance précédente, car pour trois colonnes par exemple, on peut la stocker sous la forme 2 + 1, ce qui fait 3 colonnes à stocker contre 4, et pour 7 on peut faire 4 + 2 + 1, ce qui fait 7 au lieu de 8.

Pour l’implémentation, c’était délicat car il faut à chaque fois regarder la taille de la matrice, calculer la dct en plusieurs fois avec une fonction qui est devenue récursive (qui s’appelle elle-même) et n’enregistrer que les éléments qu’il faut. Pour le décodage il faut également savoir que le bloc que l’on lit est en plusieurs morceaux, ce qui complique tout.

24/01

Je me suis occupé de l’aspect pratique et esthétique du logiciel que nous réalisons.

Je l’ai organisé comme une application MDI (multiple document interface), c'est-à-dire avec une fiche parent (principale) qui accueille à l’intérieure d’elle d’autre fiches pour gérer les différents formats. J’ai ajouté des éléments pour afficher la progression, les informations sur les images, leur affichage …

J’ai également organisé le code source, en le répartissant dans différents fichiers suivant le format pour lequel il sert. J’ai mis dans un fichier ‘_Bibliotheque’ tout ce qui sert pour plusieurs formats, c'est-à-dire Huffman, lecture/écriture BMP …

j’ai aussi organisé le code source dans le fond, pour le rendre plus élégant, en regroupant des fonctions et données allant ensemble dans des structures, simplifiant leur utilisation.

31/01

Nous avions du mal à comparer les performances de notre format JTPE avec le format répandu JFIF, car il était difficile de juger de la qualité d'une image par rapport à une autre.
Nous avons trouvé sur internet une grandeur, l'Erreur Quadratique Moyenne (EQM), qui calcule les différences d'une image par rapport à l'original avec la formule suivante :

$$EQM = \frac{1}{N}\sum_{i=1}^{i=N}\left(n'_i - n_i\right)^2$$

où *n'i* est la donnée récupérée et *ni* la donnée originale, *N* le nombre total de données

apparemment, c'est une simple moyenne des carrés des différences (quadratique). Le carré doit être là pour éviter que les erreurs se compensent (toujours avoir quelque chose de positif)

on peut cependant discuter cette grandeur, car le principe du jpeg est justement de perdre de la précision où l'oeil y est le moins sensible, et cet EQM n'en tient pas compte.

j'ai tout de même décidé de l'implémenter, car étant donné que les deux formats utilisent la même méthode, cela devrait convenir pour les comparer.

finalement le jfif apparaît plus efficace, mais comme c'est une moyenne quadratique, en prenant les racines carrés des EQM on trouve que la différence n'est que de 0.5 à 1.5 unités sur 256, ce qui est négligeable, surtout que l'on ne voit pas de différences à l'oeil nu, mais nous avons tout de même été déçus …

21/02

Nous avons cherché des informations sur Internet qui pouvaient enrichir notre dossier, notamment sur les applications des algorithmes sur lesquels nous avons travaillé.
La compression est très utilisée dans les images satellites, par exemple pour les cartes topographiques : une carte nationale numérique au 1:25000 est fournie à une résolution de 500dpi et couvre une zone de 17 sur 12 km.

Pour couvrir la Suisse, il faut 260 images de 14000 x 9600 pixels. Actuellement ces images sont livrées en 8 couleurs, mais pour inclure les reliefs, on utilise 24 bits. Ce qui donne (24 * 14000 * 9600 * 260 / 8) octets, soit plus de 97 Gigaoctets en Bitmap. Mais les images sont distribuées en format TIFF, autre format très connu, qui permet de compresser des images avec plusieurs algorithmes. Et dans le format TIFF-LZW, les images font 4 gigaoctets pour toute la Suisse (gain de 95% !).