Travail Pratique

**Département de génie logiciel et des TI**

|  |  |
| --- | --- |
| Numéro du laboratoire | 4 |
| Étudiant(s) | Desmarais, Gabriel  Benoit, Marie-Ève |
| Code(s) Permanent(s) | DESG24078908  BENM22568707 |
| Cours | GTI310 |
| Session | Automne 2010 |
| Groupe | 01 |
| Professeur | Stéphane Coulombe |
| Chargé de laboratoire | Jean-François Franche |
| Date | 10 Décembre 2010 |

**Table des matières**

[1. Introduction 1](#_Toc279761246)

[2. Choix de design 2](#_Toc279761247)

[3. Implémentation 3](#_Toc279761248)

[4. Analyse du projet 5](#_Toc279761249)

[5. Conclusion 6](#_Toc279761250)

[6. Questions supplémentaires 7](#_Toc279761251)

[7. Manuel de l’usager 8](#_Toc279761252)

[ANNEXE 1](#_Toc279761253)

# Introduction

La compagnie Squeeze-media désire pouvoir améliorer ses performances en stockant plus d’images dans leur base de données. Pour ce faire, l’utilisation d’une méthode de compression similaire à celle du JPEG a été proposée.

Le projet a pour but de compléter un cadre de développement déjà existant (Squeeze Light Media Codec) afin d’y ajouter les classes permettant la compression d’images au format ppm vers un format quasi-JPEG et vice-versa.

Pour ce faire, une classe ConvertColor se chargeant des conversion de modèle de couleur, une classe BlocManager s’occupant de fractionner l’image en bloc de 8x8, une classe DCTManager pour la gestion des coefficients produits par la DCT, une classe Quantification servant à réduire le nombre de bits représentant une valeur et une classe ZigZag pour changer l’ordre des coefficients et encoder les ACs ainsi que les DCs devront être implémentées.

Le présent document contient une section Choix de Design qui expose les différentes décisions possibles inhérentes à la conception, une section Implémentation contenant les détails de l’implémentation et l’analyse complète du projet est exposée dans la section Analyse du Projet. Un retour sommaire sur les notions abordées se trouve dans la section Conclusion, il s’en suit les réponses aux questions supplémentaires, le guide d’utilisateur ainsi que l’annexe.

# Choix de design

Pour réaliser le projet, il était nécessaire de respecter certaines contraintes. Ces éléments consistaient à ne pas utiliser de code JPEG disponibles sur le web, garder le code simple malgré un facteur de compression moins optimal, permettre à l’utilisateur de choisir le facteur de qualité pour chaque image à coder (Valeur entière allant de 1 à 100) et l’application doit également permettre le décodage des images.

Plusieurs questions ont été soulevées pendant la phase de conception.

1. Comment faire la séparation en blocs?

Deux approches ont été envisagées pour la séparation de l’image en bloc. La première consiste à faire une liste chaînée de tableaux tridimensionnels contenant des tableaux de dimensions 8x8 pour chaque couleur. Cette méthode permet d’ajouter dynamiquement des blocs à une liste et garde le code simple. La seconde méthode consiste à créer un tableau contenant les divers blocs. Cette méthode nous oblige à connaitre la grandeur du premier tableau à l’avance.

1. Comment tester l’application et avec quel degré de précision?

Comme suggéré dans l’énoncé du projet, l’utilisation de JUnit permet de valider les valeurs obtenues dans les fonctions des différentes classes et il a été envisagé d’en faire uniquement pour les classes suggérées, soit ConvertColor, BlocManager et DCTManager. Cette solution, bien que testant les éléments principaux de l’application, reste incomplète. C’est pourquoi il a été envisagé de faire une classe de test pour chacune des classes du programme, main comprit en créant un test validant plusieurs classes d’un seul coup, ce qui permettrait de faire des tests plus complets, mais exigerait beaucoup plus de temps et ne serait pas forcément nécessaire.

1. Utiliser un tableau d’ordre pour le ZigZag, ou des compteurs?

Lors de la conception de la classe ZigZag, l’utilisation d’un tableau de recherche permettait d’avoir un code plus concis et plus compréhensible. Les exemples en lignes utilisent par contre des méthodes constituées de compteurs alors cette deuxième méthode est plus document et utilisée. Elle deviendrait donc relativement plus sécuritaire à utiliser.

1. L’utilisation de listes ou de tableau pour garder les données?

Les données étaient toujours constituées en tant que chaine. Cette chaine devait être gardée à travers les étapes et transférée d’une classe à une autre à l’aide de variables. Ces variables pouvaient être constituées de tableau, de listes de tableaux, de tableau de listes, ou de listes de listes. Les choix sont grands mais une représentation claire et plus intuitive était priorisée.

# 3. Implémentation

Pour réaliser l’application solutionnant la problématique, plusieurs classes ont été ajoutées ou implémentées au cadre de développement Squeeze Light Media Codec.

Premièrement, une classe ConvertColor a été ajoutée afin de pouvoir convertir l’image au format YUV au moment de la compression et en RGB au moment de la décompression et contient deux méthodes : convertRGBToYUV() et convertYUVToRGB(). La première méthode reçoit en paramètre le tableau tridimensionnel (l’image) lue par le PPMReaderWriter, puis boucle sur le tableau pour convertir les valeurs RGB vers YUV et retourne un autre tableau tridimensionnel des nouvelles valeurs. La deuxième reçoit un tableau de couleur YUV reconstitué par la classe BlocManager et retourne un tableau des valeurs convertie en RGB en les normalisant entre 0 et 255.

Ensuite la classe BlocManager a été implémentée et contient une méthode split() qui sépare l’image en bloc de 8x8 provenant de ConvertColor et une méthode merge() qui reconstitue les blocs en un seul tableau à 3 dimensions. Pour la séparation en bloc, quatre boucles imbriquées se charge de lire le tableau et de le subdiviser en tableaux 8x8, ces tableaux sont ensuite ajoutés à une liste chaînée qui facilite la circulation et la manipulation individuelle des blocs et offre plus de flexibilité qu’un tableau de tableau statique.

Puis, le DCTManager a été ajouté afin de pouvoir obtenir une représentation séquentielle des pixels. Cette classe contient une méthode DCT() qui s’occupe de calculer les coefficients à partir de la liste de blocs du BlocManager et une méthode iDCT qui se charge de faire la manipulation inverse à partir de la liste de bloc provenant de Quantification. La première méthode fait une double boucle pour obtenir une liste de cosinus nécessaire au calcul des coefficients, puis enchaîne sur cinq boucles imbriquées qui multiplient chaque éléments du tableau passé en paramètre avec les cosinus correspondants, applique les formules fournies dans la documentation et ajoute les coefficients obtenues dans la liste chaînées des coefficients. La seconde méthode fait la même double boucle pour les cosinus pour ensuite faire le calcul inverse du DCT.

Une classe Quantification a également été ajoutée afin de réduire le nombre de bits nécessaires pour représenter une valeur, permettant ainsi la compression des images. Cette classe contient une méthode initMatrice() pour initialiser la matrice fournie dans l’énoncé du laboratoire et nécessaire au calcul de la compression, une méthode do() pour faire la compression de la liste de blocs du DCTManager et une méthode undo() pour l’action inversion sur la liste venant du ZigZag, le tout prenant en compte le facteur de qualité passée en argument à l’application.

Par après, une classe Zigzag a été implémentée pour changer l’ordre des coefficients pour le codage. Elle contient une méthode getAC() qui retourne le tableau à trois dimensions des coefficients AC encodés en couples à partir d’une liste de blocs, une méthode getDC() qui retourne un tableau à deux dimensions des DC encodés à partir d’une liste de blocs, une méthode createBlocs() qui retourne une liste de bloc à partir d’un tableau de DCs, de ACs, d’une hauteur et d’une largeur. Cette classe contient également une méthode findZigZagOrder() qui retourne l’emplacement dans le ZigZag d’un index passé en paramètre.

Finalement, les classes BlocManagerTest, ConverColorTest, DCTManagerTest, MainTest, QuantificationTest et ZigZagTest ont été ajoutées à l’application afin de valider les informations utilisées dans les classes ayant un nom similaire. Toutes ces classes implémentent JUnit et se terminent par test afin de les identifier comme classes de test. Les méthodes ont été nominées de façon similaire.

# 4. Analyse du projet

Tester l’application efficacement s’est avéré un problème durant le projet car le résultat final, l’image décompressée, n’était accessible qu’à la fin du processus, soit après que l’image ait été encodée puis décodée. C’est pourquoi les tests unitaires se sont avérés essentiels pour, minimalement, évaluer la validité des valeurs obtenus après chaque étape. Cependant, s’en tenir aux tests suggérés s’est vite avéré insuffisant, c’est pourquoi des tests ont été ajoutés pour chaque étape de l’encodage et du décodage en plus d’un test de la classe Main. Cette dernière classe de test a de particulier qu’elle n’utilise pas le Assert propre au test unitaire avec JUnit, mais test tout de même les valeurs au fur et à mesure de l’application à l’aide de data véritable, soit des images au format ppm tout en passant au travers de plusieurs classes.

La raison de ces tests manuels est que l’utilisation de valeurs arbitraires facilement vérifiables s’est avérée insuffisant puisque les tests individuels de chaque classe passent sans problème, mais que le résultat final différait grandement du résultat attendu. Cela s’explique par le fait que les tests unitaires ne prennent pas en compte les erreurs humaines lors de la réalisation des tests et que ces tests n’indiquent pas les erreurs non surveillées tel que les cas d’exceptions par exemple.

Deux endroits ont été principalement causes de problème, la première a été lors de la reconstruction de l’image à partir des divers blocs. La méthode merge servant à effectuer cette tâche était erronée et causait des erreurs lorsque le programme utilisait une image réelle. Les tests unitaires utilisaient des tableaux suffisamment petits pour ne pas causer d’erreur, ce qui a ralentie la détection et l’ajustement du problème.

Le deuxième endroit où la programmation s’est révélée ardue a été dans le calcul des ACs lors de la décompression. La logique derrière la liste des ACs n’était pas implémentée telle que voulue et cette différence avec la norme donnait des valeurs différentes, ce qui donnait de fausses images. Le code n’attribuait pas les valeurs normalement lors de la détection des fins de blocs. Le manque d’information de fin de bloc dans la suite du processus ne permettait plus l’affichage des ACs au bon endroit. La réécriture de cette partie de code s’est révélée obligatoire pour résoudre le problème et obtenir les valeurs attendues.

# 5. Conclusion

La programmation s’est révélée ardue car nous ne savions pas comment tester les données obtenues. Une mauvaise compréhension des notation de fin de bloc ne nous a pas permis d’aller plus rapidement, mais malgré toutes ces problème l’application est fonctionnelle et permet d’encoder et de décoder les images voulues.

Un dernier problème d’optimisation persiste car lors du décodage de grandes images, l’application prend énormément de temps. Cette optimisation serait la prochaine étape pour rendre l’application facile d’utilisation.

Malgré le délai encouru par l’application lors de la décompression, tous les objectifs reliés à la programmation ont été respectés et accomplis.

# 6. Questions supplémentaires

1. Pour les images de référence lena.ppm et peppers.ppm, faites un graphique (en utilisant des facteurs de qualité de 1, 10, 20, 30, 50, 70, 90 et 100) :

1. Du facteur de compression en fonction du facteur de qualité.

b. Du niveau de qualité subjective en fonction du facteur de qualité.

2. Quelles modifications devriez-vous apporter à votre programme afin qu’il puisse supporter la compression en tons de gris en plus de couleur (c.-à-d. l’usager peut décider de conserver seulement l’information de tons de gris d’une image couleur lors de la compression)? Donnez les changements majeurs sans entrer dans les détails inutiles.

# 7. Manuel de l’usager

L’environnement utilisé lors des tests était Windows 7 avec Java version 6 update 22.

Lors des tests effectués, les configurations fournies par Eclipse ont été utilisées pour gérer les arguments envoyés à l’application. Ces arguments étaient les suivants :

Pour compresser lena.ppm :

"Session 3\\GTI310\\SqueezeLightMediaCodec-TP4\\lena.ppm"

"% de compression" (Varie selon les tests)

Pour compresser mandrill.ppm:

"Session 3\\GTI310\\SqueezeLightMediaCodec-TP4\\mandrill.ppm"

"% de compression" (Varie selon les tests)

Pour compresser monalisa.ppm :

"Session 3\\GTI310\\SqueezeLightMediaCodec-TP4\\monalisa.ppm"

"% de compression" (Varie selon les tests)

Pour compresser peppers.ppm :

"Session 3\\GTI310\\SqueezeLightMediaCodec-TP4\\peppers.ppm"

"% de compression" (Varie selon les tests)

Pour décompresser une image en slz :

"Session 3\\GTI310\\SqueezeLightMediaCodec-TP4\\output.slz"

La méthode qui doit être utilisée en dehors de l’interface de développement est la suivante :

java <programme> <fichier d’entrée> <pourcentage de compression si compression>

java <programme> <fichier d’entrée>

Cette méthode permettra à un utilisateur de lancer l’application avec les arguments nécessaires.

# ANNEXE