Projet de programmation C
Compression d'images avec des quadtrees
Licence d'informatique
–2020 - 2021–

Table des matières

Hugo PETIOT
Johnson VILAYVANH

ARCHITECTURE

CRÉATION DU QUADTREE

Structure du quadtree

La première chose qu'il faut résoudre avant même de pouvoir parler de la création même du quadtre est la structure adopée pour créer ce quadtree, qu'on va retrouver dans le module **quadtree**.
Pour faciliter la reconnaissance d'un nœud et d'une feuille, nous avons créer le champ ***type***. De plus, on va également utiliser une union pour distinguer une feuille d'un champ : effectivement, un nœud devra pointer vers ses fils (***no***, ***ne***, ***so***, ***se***), alors qu'une feuille va disposer d'une ***color***.
Il ne restait plus qu'à ajouter une des valeurs essentiels pour la création du quadtree, qui est la valeur de l'erreur d'un nœud qui représente une zone de l'image d'origine, que nous avons ainsi appelé ***color_error***.

Pas de grande difficulté en ce qui concerne le calcule de la couleur moyenne d'une zone d'une image : selon la zone de l'image, on va calculer toutes les couleurs des pixels et en faire la moyenne. Pour indiquer quelle zone un nœud va représenter, on va, pendant la création du quadtree, utiliser une structure ***zone*** (dans le module **zone**), qui représente une zone de coordonnée dans laquelle le nœud devra calculer sa couleur moyenne. Cette structure possède ainsi 4 champs représentant 4 coordonnées, (x1, y1) et (x2, y2), respectivement le sommet haut-gauche de la zone, et le sommet bas-droite de la zone.
De plus, pour faciliter la division d'une zone en 4, on crée également 1 fonction pour chaque zone (les fonctions dans le module **zone**), fonction qui va renvoyer la sous-zone d'une zone (NO, NE, SE ou SO) par rapport à la zone récupérer en argument.

ListError

Maintenant que nous pouvons calculer la couleur moyenne d'une zone, il reste à déterminer quelle zone doit-on diviser. Naturellement d'après le sujet, il s'agissait de récupérer la zone qui possède la grande valeur d'erreur par rapport à ce qui est représenté à réalité sur l'image.
Cependant, lorsque notre arbre commence à devenir *très* grand, cela devient vite très long de parcourir toutes les feuilles à la recherche de celui-ci qui possède la plus grande valeur d'erreur.
Le module **list_errors** répond à ce problème à l'aide d'une liste chaînée triée ***ErrorList***.
Effectivement, en triant les feuille du quadtree dans cette liste par ordre décroissant de valeur d'erreur, il ne reste plus qu'à récupérer la 1ère valeur de cette liste qui représente la zone ayant la plus grande erreur.
De plus, lorsqu'on va diviser une zone, il faut également ajouter les 4 nouvelles zones créées dans la liste chaînée. On remarque qu'en règle générale, la valeur d'erreur des 4 sous-zones d'une zone qui vient d'être divisé tendent à être très petit.
Il était donc important de pouvoir accéder aux derniers éléments de la liste chaînée sans avoir à la parcourir entièrement : d'où l'idée de la structure ***ErrorList_FirstLast*** qui va pointer sur le 1er et dernière élément de cette liste chaînée.

Création du quadtree

Dans le module **create_quadtree** dans lequel on construit un quadtree à partir d'une image, à chaque tour de boucle de la fonction image_to_quadtree, on va récupérer le 1er élément de la liste chaînée ***ErrorList***.
Ce 1er élément, qui va pointer sur la feuille qui doit être divisée, va devenir un nœud pour être divisé en 4 feuilles, qui seront elles-mêmes diviser, *si le besoin se montre*.
Effectivement, il est important de noter que nous avons jugé que les feuilles qui dispose d'un color_error == 0 ne nécessitent pas d'être divisé plus tard, puisque le calcule de l'erreur de la zone qu'elles représentent indiquent que ces feuilles représentent déjà la couleur qui respecte la réalité. Nous n'avons donc aucun intérêt à les diviser.

MINIMISATION

Pour la minimisation, on va naturelle utilise le module **minimisation** pour stocker les fonctions relatives à cette fonctionnalité.

Minimisation sans pertes

La minimisation sans pertes n'a pas été très optimisée. En effet, seule la 1ère idée du sujet concernant la minimisation sans perte a été utilisé , à savoir remplacer tous les nœuds à hauteur 1 dont les 4 enfants ont la même couleur.
Un simple parcours préfixe de l'arbre qui contrôle que chaque nœud de hauteur 1 que nous rencontrerons devra ne pas respecter cette règle, auquel cas il sera remplacé par une de ses feuilles.

Minimisation avec pertes

La minimisation sans pertes permet de réduire considérablement un quadtree.
Cependant, on doit également noter que celle-ci est ***très peu*** optimisée par rapport à ce qui avait pu être demandé.
En effet, il était question de chercher tout couple d'arbre qui ont une distance minimale entre eux, pour ainsi les fusionner.
Cependant, malgré des recherches, nous sommes simplement arriver à une fonction qui va comparer toutes les sous-arbres d'un même nœud, mais pas les autres sous-arbres de tout le quadtree.
Effectivement, pour un nœud qui possède 4 feuilles NO, NE, SE et SO, la fonction ***minimize_loss_fuse_identical_subtree*** va simplement comparer ces 4 mêmes feuilles entre elles, et les fusionner si la distance entre elles est moindre (selon la valeur de ***DIST_MAX***) : en d'autres termes, on minimise très peu l'arbre et on réduit très peu le nombre de nœuds.

Une autre fonction de minimisation avec pertes a également été implanté, qui va réduire considérablement le nombre de nœuds dans l'arbre. Elle fonctionne cependant très « bêtement », puisqu'elle va simplement se charger de parcourir tout l'arbre et de supprimer tous les sous-arbres dont la racine dispose d'une couleur erreur qu'elle considère comme peu élevé (selon la valeur de la variable ***ERROR_MAX***).

En résumé, la minimisation avec pertes a des résultats peu convaincants. On peut réduire le nombre de nœuds, mais avec un algorithme bête et méchant qui va supprimer les nœuds qu'elles considèrent « assez proche » de la réalité.

ENCODAGE

Encodage/Decodage d'un quadtree non minimisé

Pour l'encodage comme le décodage en bit, il était nécessaire de créer un buffer de bit (module **bit_buffer**).
Ensuite, en ce qui concerne l'encodage, il suffisait de parcourir l'arbre en préfixe avec un buffer qu'on remplit en fonction des informations qui nous viennent : est-ce un nœud, est-ce une feuille ?
Puis on remplissait le buffer de l'information correspondante. Et lorsqu'on devait indiquer les couleurs d'une feuille, on codait bit par bit les informations concernant la valeur r, g, b, a de celle-ci.
Le décodage se déroule plus ou moins de la même manière : on parcourt le fichier à l'aide d'un buffer qui récupère les 0 et 1 du fichier binaire. Ensuite, selon le bit qu'on lit, on crée soit un nœud, soit une feuille, soit une couleur.
A noter que le décodage fonctionne avec les fichiers des autres groupes, contrairement aux graphes que nous allons voir.

Encodage/Decodage des graphes (quadtree minimisé)

Effectivement, première chose à noter : le décodage des graphes ne fonctionne pas avec les fichiers .gmc et .gmn de l'algorithme qui était demandé de base pour minimiser un quadtree.
En effet, puisque notre algorithme de minimisation avec pertes ne compare que les sous-arbres d'un même nœud, lorsqu'on décode un fichier, on va seulement noter si les feuilles d'un même nœud (= d'une même ligne) correspondent au même numéro de nœud.
On a donc des fonctions qui comparent si c'est le cas, qui nous permet d'éviter d'allouer 2 fois un quadtree pour un même nœud.

Pour l'encodge, c'est la même chose : on compare si les sous-arbres d'un nœud pointent vers le même sous-arbre : si c'est le cas, on leur attribut le même numéro de nœud, et on incrémente le nombre total de nœud si et seulement si un sous-arbre ne pointe pas sur le même sous-arbre qu'un autre.

PARTIE GRAPHIQUE

Le challenge de la partie graphique à été la jonction entre les fonctionnalités et les boutons. Il était difficile de faire cela proprement tout en integrant les incompatibilités d'actions et de format.
Pour la création des boutons, le plus dur à été de trouver une format qui les décrivent suffisement pour les afficher mais assez simplement pour que la lecture soit claire. C'est pourquoi ils sont subdivisés avec "box" qui contiennent leur coordonées et leur taille et faire cela proprement tout en integrant les incompatibilités d'actions et de format.

DÉROULÉ DU PROJET

PRISE DE CONNAISSANCE DU SUJET

Les bases du projet

En débutant le projet, il nous a semblé nécessaire de débuter en même temps malgré la distance qui nous a empêché de travailler au même endroit. Effectivement, sans structure du quadtree, on ne pouvait pas avancer.
Nous nous sommes donc retrouvés sur Discord avec un partage d'écran pour coder les fonctions de base du projet : la structure du quadtree, la création des nœuds, etc.

Répartition des tâches

Suite à cela, nous nous sommes réparti les tâches de la manière suivante : l'un se chargerait de la partie technique, et le second de la partie graphique.
Tout en se tenant informer comment chacun avançait, une fois que chaque partie était assez fonctionnel, nous les avons assembler.

C'est également de cette façon que nous avons travaillé sur redmine : la branche master restait intouché, et nous travaillions chacun sur notre branche respective.
Une fois que nos parties fonctionnaient, nous les avons mis en accord pour les faire fonctionner ensemble.

CONCLUSION

Ce projet a été très long, mais surtout avec une difficulté qui n'était pas forcément en accord avec les TP que nous avons pu avoir.
De plus, la distance ne facilitait pas les moments qui nous auraient permis de nous retrouver afin de discuter du projet.
Enfin, par manque de temps et une charge de travail beaucoup plus grande qu'anticipé, on peut déplorer la minimisation avec pertes qui est loin d'être achevé.