Projet Génie Logiciel

Rapport de projet

Tom Daguerre - Zoé Coudray

Encadrant : Jean-Yves Ramel

Département informatique 4ème année

Table des matières

1 En quoi consiste notre projet ?

1.1 Ce qui nous a été demandé

Pour ce projet, nous devions appliquer une opération de Split and Merge à une image 3D. Cela implique que nous puissions lire et écrire ce type d'image. Il s'agissait donc en réalité de lire l'image entrée dans le programme, de réaliser le Split and Merge et enfin d'écrire une nouvelle image au même format contenant les résultats. Un des points importants de ce projet est la rapidité d'exécution et donc la complexité de nos algorithmes.

1.2 Algorithme Split and Merge

Pour effectuer un algorithme Split and Merge on a besoin de prendre une image composée de pixels, définis par une couleur (RGB ou non), et de définir un critère d'homogénéité. On va ensuite diviser cette image en parties les plus grandes possibles respectant ce critère. Pour se faire l'image va passer à travers deux filtres. D'abord un Split qui va prendre une partie de l'image (en commençant par l'image entière) et soit la garder entière et arrêter le programme, soit la diviser si elle ne respecte pas le critère d'homogénéité et rappeler le split sur chacune des nouvelles parties. Ensuite le merge va prendre chacune des parties obtenues en fin de split et les comparer entre elles. Si ces deux parties sont voisines et respecteraient le critère d'homogénéité même en étant combiné alors il les combine. Bien qu'elles ne le laissent pas paraître ainsi ces deux parties sont extrêmement simples à réaliser pour un ordinateur, la partie la plus compliqué comme nous le découvriront plus tard est un peu cachée à première vue et à lieu entre le Split et le Merge : la création du graphe de voisinage.

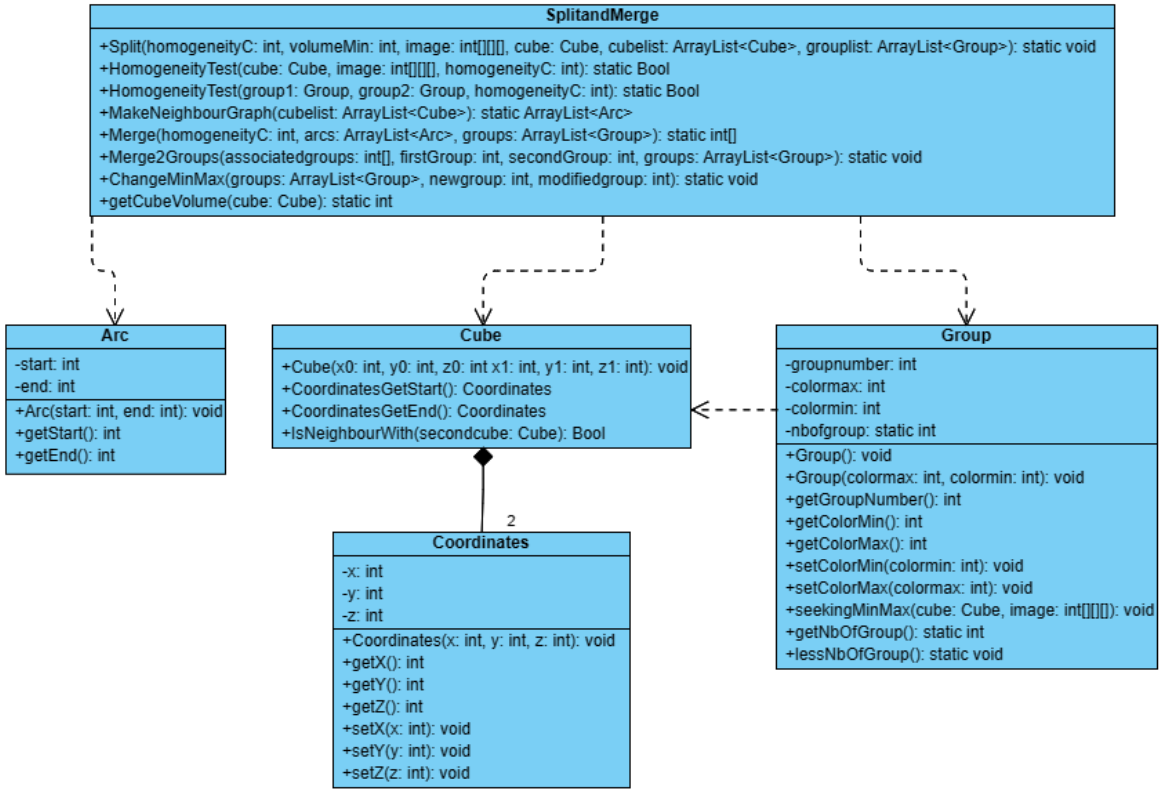
1.3 Outils utilisés

En ce qui concerne le langage utilisé, nous avons opté pour Java, c'est un langage orienté objet avec lequel nous sommes plutôt familiers et qui permet une plutôt bonne performance. Nous avons programmé à l'aide d'IntelliJ et nous utilisions gitHub pour communiquer et nous coordonner. Pour lire et écrire les images 3D, nous avons utilisé la bibliothèque simpleITK. Enfin, afin de visualiser des images 3D, nous avons utilisé ITK-SNAP.

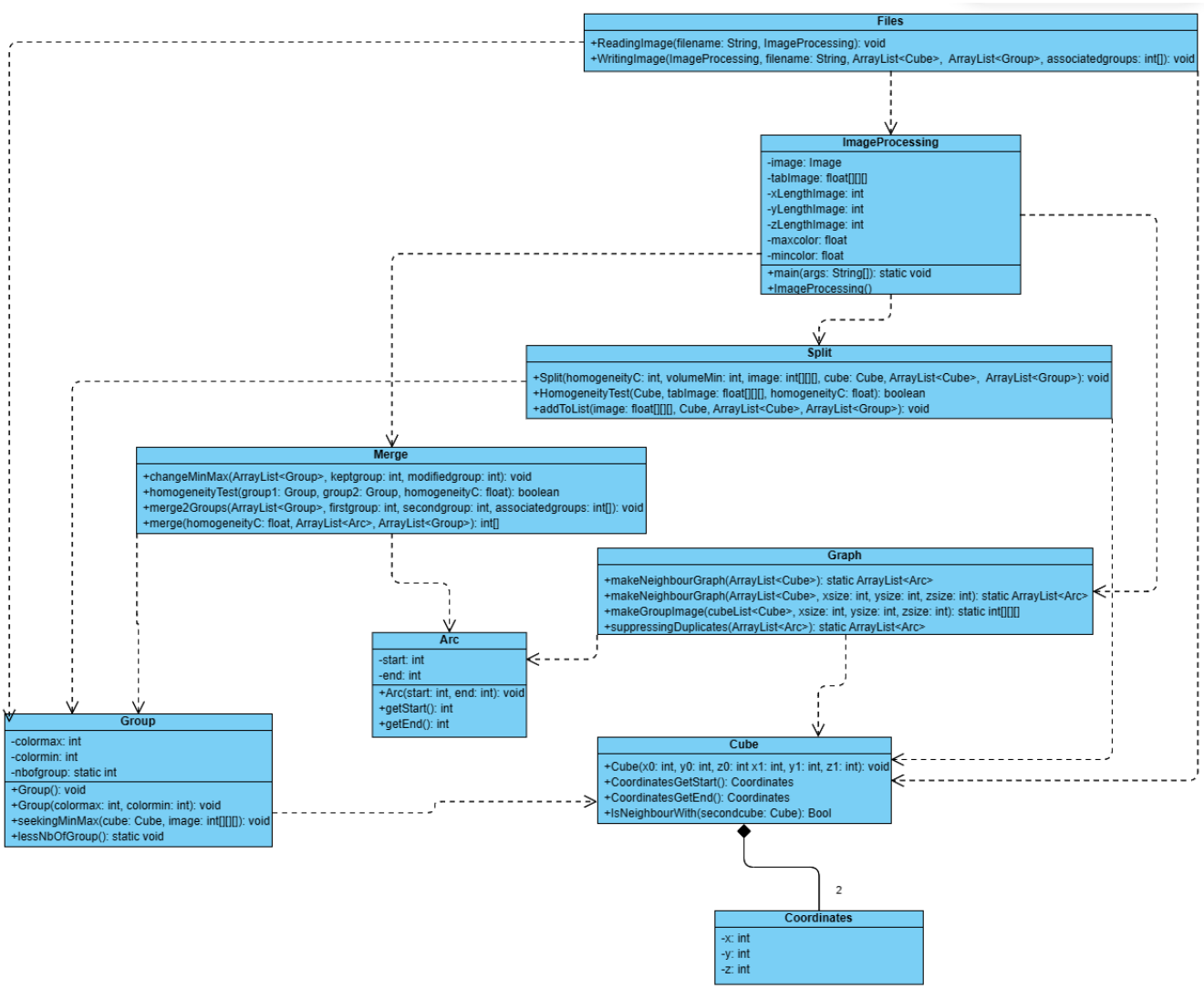
2 Modélisation et Spécifications (12 heures)

2.1 Modélisation

Une des premières étapes de notre projet et ce qui nous a pris le plus de temps a été de définir les classes, leurs liens, les fonctions principales et leurs algorithmes. Nous avons pour cela un document entier appelé Specifications. Ci-dessous notre idée première de modélisation UML.



Désormais, notre modélisation est un peu différente, la classe SplitandMerge a été divisée en trois classes différentes. De plus, nous avons maintenant une classe ImageProcessing contenant le main et une classe Files qui contient les méthodes de lecture et écriture des fichiers.



Dans cette deuxième version nous avons 4 classes types objets qui permettent de représenter des coordonnées, des cubes, des arcs et des groupes de pixels. Nous avons également créer 4 classes qui contiennent uniquement des méthodes static et n'ont pas vocation à être instancié. Il y a les classes Split, Graph et Merge qui correspondent aux différentes parties de l'algorithme et la classe Files qui va gérer l'ouverture des fichiers.

2.2 Spécifications

Les Spécifications ont été réalisées en quasi parallèle de la modélisation. Après la première phase de modélisation nous avons pu nous tourner vers les spécifications des méthodes principales et les plus complexes sous forme d'algorithme. Ces spécifications nous ont permis de détailler les entrées et sorties mais aussi de comprendre plus en profondeur l'algorithme que nous allions utiliser. Cette partie spécification a été très utile pour redéfinir nos besoins en terme de modélisation, certaines classes devenant obsolètes ou pas assez détaillés. Par exemple nous avions pensé à une classe Pixel contenant une couleur et un numéro de groupe et grâce à elle, représenter notre image à l'aide d'un tableau 3D de pixel. Cependant cette représentation demandait énormément de modifications lors des algorithmes et nous nous sommes tournés vers une classe Group qui en association avec la classe Cube regroupe toutes les informations nécessaires

et permet des modifications rapides.

Ces spécifications ont également été très utiles pour communiquer avec notre encadrant. Grâce aux spécifications il a été plus facile de nous faire comprendre et il a été plus facile pour lui de pointer certains problème et de nous aiguiller sur des points trop obscurs qui pourraient poser problème à l'avenir.

3 Les étapes de notre projet et nos difficultés

3.1 Les classes principales(2 heures)

Grâce à nos spécifications relativement détaillés nous avons pus en seulement 2 heures créer toutes nos classes objets, et une première ébauche de la fonction main permettant la lecture et écriture d'une image 3D simple. Cette première phase a permis de commencer nos algorithmes complexes sur une base fonctionnelle et qui nous permettait également d'avoir un résultat direct et lisible à l'aide de l'image renvoyé par l'algorithme. Cette image était au début exactement identique à l'image d'origine mais on avait déjà mis en place une boucle sur chaque pixel permettant de modifier leur couleur.

3.2 L'algorithme Split and Merge(6 heures)

La création de nos classes fonctionnelles a mis environ 6 heures, durant ces 6 heures nous avons pu mettre en code nos algorithmes précédemment réfléchis. Nous avons commenté et décrit brièvement toutes nos classes. Nous avons également codé une méthode assez complexe : IsNeighbourWith. Cette méthode de la classe Cube va comparer les coordonnées de 2 cubes et retourner vrai s'ils sont voisins. Grâce à cette méthode nous avons pu créer nos premiers graphes de voisinage et tester notre algorithme complet. Après cette étape nous avons obtenu une première image résultat de notre algorithme Split and Merge.

3.3 Création du graphe de voisinage(4 à 6 heures)

La création du graphe de voisinage est au cœur de notre programme, elle permet d'exécuter le merge facilement et est la méthode la plus longue de tout notre processus. Nous avions en tout premier pensé faire une méthode comparant les pixels adjacent mais lorsque nous avons abandonné la classe pixel, permettant de stocker le groupe du pixel et sa couleur dans le même tableau, nous avons abandonné cette méthode qui semblait trop longue à exécuter. Nous nous sommes donc tournés sur une double boucle tournant sur chacun des couples de cubes obtenus à la fin du split et notre méthode IsNeighbourWith. Cependant comme vous pourrez le voir dans la partie performance cette méthode est très peu efficace nous avons donc demander conseil à M Ramel qui nous a parlé de comparer les pixels voisins, ce qui nous a rappelé notre première idée. Nous avons donc décidé d'écrire une nouvelle méthode permettant de créer notre graphe sur cette idée et cette partie prenant de plus en plus d'importance dans le projet nous avons créé une nouvelle classe : la classe Graph contenant chacune des méthodes aidant à créer le graphe de voisinage.
Une fois les deux méthodes créés nous avons pu comparer leur efficacité et la deuxième méthode était en effet bien moins complexe à l'exécution.

3.4 Les tests unitaires

Pour tester notre programme nous avons d'abord préférés faire des test manuels. Notre première version fonctionnelle a été assez vite réalisée comme nous l'avons vu précédemment, pas plus de 8h entre le début du code et la première version. Grâce à cela les tests étaient assez simple à réaliser à la main car nous avions visuellement le résultat de notre programme grâce à l'image en sortie. D'un autre côté nous voulions vraiment réaliser des tests unitaires mais comme nous ne réussissions pas à les mettre en place à cause d'un problème de librairie nous les avons laissés de côté relativement longtemps. Une fois notre problème résolu nous avons testé nos méthodes sur lequel nous avions le plus de doute et celles qui pouvaient provoquer des bugs de manière transparente. Il est assez difficile d'estimer le temps que nous ont pris les tests à réaliser car ils ont étés répartis sur plusieurs séances. On pense y avoir consacré environ 3/4 heures chacun.

3.5 Quelques petits problèmes

Nous avons rencontré différents problèmes au cours de notre projet sur des aspects très différents. Ces problèmes restant relativement simple ils ont en général nécessité entre 1 et 3 heures de travail pour l'un d'entre nous.

Le graphe de voisinnage L'un des premiers problèmes que nous avons eu a été la création du graphe de voisinage. Nous n'avions pas suffisamment détaillé la fonction permettant de créer le graphe lors des spécifications, nous avons donc rencontré plus de difficultés que lors de l'écriture des autres méthodes. La méthode de comparaison entre deux cubes par exemple était bien plus complexe à coder que nous le pensions.

Une modification anodine Ensuite lors du codage de la deuxième méthode de création du graphe il est très vite apparu que si on compare les pixels entre eux deux groupes voisins seront ajoutés de très nombreuses fois dans le graphe de voisinage. Nous avons pensé qu'il était plus simple de supprimer tous les duplicatas à la fin plutôt que de vérifier à chaque fois qu'on ajoute un arc s'il n'est pas déjà dans le graphe. Cependant pour supprimer les duplicatas nous avons du réécrire la méthode equals de la classe arc et également changer fondamentalement la classe car nous n'avons pas réussi à modifier la méthode equals afin que l'arc (3,4) soit considéré identique à l'arc (4,3). Le constructeur de la classe a donc été modifié de manière à ce que l'origine soit toujours plus petite en valeur que la destination. Cette modification aurait pu causer des problèmes car elle change assez fondamentalement la classe mais heureusement ce changement est resté transparent aux autres classes.

Multiplier les tests sur les entrées Un autre problème que nous avons rencontré à été avec la lecture des images nifti. Au cours de l'élaboration de notre programme nous avons testé toutes nos méthodes avec une seule image et nous nous sommes rendus compte assez tard que notre code ne fonctionnait pas avec la plupart des autres images nifti nous avons donc passé un peu de temps à corriger notre main pour résoudre ce problème.

4 Nos tests de performance

Une fois les deux méthodes de création de graphe implémentées, nous avons réalisé des tests sur six images différentes pour comparer les temps d'exécution et les résultats obtenus.
Afin de trouver des images NIFTI, nous avons du chercher des dépots github en mettant à disposition. Nous avons fini par en trouver un proposant des images de volume variable, ce qui était exactement ce dont nous avions besoin afin de réaliser nos tests.
Les six images que nous avons retenues sont :

- Image de cerveau de caille donnée par le professeur
- Image de cerveau humain
- Image de corps humain (grand volume)
- Image d'iguane (grand volume)
- Image de crâne d'humain (grand volume)
- Deuxième image de cerveau d'humain (grand volume)

Les images marquées comme ayant un grand volume et les deux premières n'ont pas été testées avec les mêmes paramètres. En effet, sur l'ordinateur utilisé pour les tests, le split and merge sur les images à grand volume en utilisant des paramètres similaires à celles ayant un plus petit volume prenait trop de temps, on a donc du les tester avec des paramètres différents.
Nos tests visaient à comparer les temps d'exécution de nos deux méthodes de création de graphe afin de déterminer laquelle était la plus efficace. Nous avons aussi comparé les images obtenues pour voir si le changement de méthode entraînait un changement dans l'image résultante.
En ce qui concerne les images renvoyées par le programme, nous n'avons pas constaté de changement à l'oeil nu. Les deux méthodes permettent donc d'obtenir les mêmes résultats mais avec des temps d'exécution qui diffèrent.
Nous avons trouvé, en réalisant les tests sur six images avec, à chaque fois, deux jeux de paramètres différents, que la deuxième méthode est bien plus efficace. Ci-après une image du tableau Excel contenant les résultats de nos tests.

	Taille				Paramètres		Temps Modèle 1			Temps Modèle 2		
Image	X	Y	Z	Tot/1000	homogénéité	volume min	Split	Graphe	Merge	Split	Graphe	Merge
1-caille	130	250	160	5200	0,3	30	0	6	0	0	4	0
1-caille	130	250	160	5200	0,2	25	0	8	0	0	4	0
2-crâne humain	196	240	256	12042,24	0,5	50	1	95	6	1	3	5
2-crâne humain	196	240	256	12042,24	0,4	50	2	272	16	1	8	11
3-corps humain	255	178	256	11619,84	0,5	50	1	15	0	1	4	1
3-corps humain	255	178	256	11619,84	0,35	40	1	63	4	1	6	75
4-iguane	210	256	179	9623,04	0,5	50	1	27	2	1	3	2
4-iguane	210	256	179	9623,04	0,4	50	1	43	5	1	4	1
5-cerveau humain	176	188	144	4764,672	0,3	30	0	92	10	0	3	6
5-cerveau humain	176	188	144	4764,672	0,2	25	0	518	38	0	4	29
6-cerveau humain- 2	188	256	190	9144,32	0,5	50	0	143	12	1	5	7
6-cerveau humain- 2	188	256	190	9144,32	0,4	50	1	273	17	4	12	21

5 Les limites de notre programme

Un programme déterministe ? Notre programme est déterministe, si vous le lancez 10 fois d'affilé sur une même image vous obtiendrez 10 fois le même résultat mais ce n'est pas le seul résultat possible. Pour des raisons de rapidité de calcul nous avons décidé de ne pas faire de priorité lors du merge, c'est à dire prioriser les groupes les plus proches du point de vue du critère d'homogénéité. Notre programme va donc lors du merge prendre les groupes dans l'ordre des arcs de notre graphe et les regrouper si possible, comme notre graphe est créé de la même manière à chaque fois on obtient le même résultat mais si celui-ci change le résultat final sera très probablement différent. L'un des exemples les plus simples permettant d'observer celà serait de tourner l'image de départ ainsi les premiers arcs créés et testés ne seraient pas du même côté de l'image.

Les images traitées Notre programme permet de traiter les images au format nifti, donc en 3D et principalement en noir et blanc. Dans le cas d'une image en couleur notre programme pourrais potentiellement fonctionner mais donnerai un rendu en noir et blanc.

6 Annexes

6.1 Journal de bord

Nous avons décidé de créer un journal de bord afin de pouvoir facilement décrire nos actions quotidiennement et de communiquer notre avancement avec notre encadrant. Il nous a été très utile afin de se remémorer à chaque séance le travail effectuer la fois précédente. De plus nous avons ajouter à quasiment chaque séance un point "TO-DO" qui décrivait les taches auquel nous avions déjà réfléchis et qui devaient être faites assez rapidement.

6.2 Spécifications

Nous avons également écris un document de spécification au début de notre projet. Ce document nous a permis en premier lieu de décrire de manière textuelle notre modélisation puis nous sommes entrées dans les détails de nos différentes méthodes parfois jusqu'à préciser un algorithme. Nous avons ajouté quelques notes en rouge afin de préciser des modifications ayant eu lieu lors du développement donc après l'écriture.

6.3 Tableur Résultat des tests de performance

lien tableur excel

6.4 Javadoc

Notre programme est documenté à l'aide de l'outil javadoc. Nous la mettons à disposition avec notre code source, sur github.