Compte Rendu TP3 POOCS

Compression d'une Image par Maillage

JOHN GLIKSBERG

M1 IHPS @ UVSQ

Introduction

Le but de ce TD/TP est de fournir un outil qui génère une maille de triangles à partir d'une image pour réduire la quantité d'information stockée d'une image.

Le maillage consistera en un ensemble de triangles dont les points sont des pixels de l'image. Chaque pixel retiendra la couleur de l'image en ce point. On pourra retrouver un affichage de l'image d'origine en dessinant les triangles remplis de la couleur moyenne de celles de ses points.

La "compression" se fait sur la formation de triangles qu'on privilégiera dans les zones avec des grandes differences de couleur.

On n'alignera *pas* les côtés des triangles avec les lignes qu'on pourra distinguer dans l'image mais on construira toujours de nouveaux triangles dans la zone qui diffère le plus de l'image (sauf lorsque l'on a atteint une résolution trop fine).

C'est ainsi un processus itératif car on ne prévoit pas où sera le prochain découpage.

On décide d'avance qu'on ne travaillera que sur des images en niveaux de gris. Pour la modélisation de notre outil on décide de se limiter à des fichiers en entrée au format `.pgm` et un enregistrement du maillage au format `.vtk`.

Définitions: Dans un maillage M donné, pour un triangle T dans M donné, si C_1, C_2 et C_3 sont les couleurs des points de T, on définit C la couleur moyenne du triangle, et l'*écart* en T correspond à la différence absolue entre C et la couleur de l'image au barycentre des points du triangle.

Ceci nous permet d'une itération à l'autre de choisir le triangle au plus grand *écart* dans notre maillage. On travaille maintenant sur le barycentre P de ce triangle, c'est le point à partir duquel on va rendre notre maillage plus précis.

Le processus précis est décrit dans l'énoncé et consiste à vider une cavité de triangles autour de P et reconstruire de nouveaux triangles plus petits.

Cet algorithme de maillage privilegiera souvent trop certaines régions de l'image, mais en limitant la granularité à celle du pixel, on répartit l'importance donnée aux zones de l'image.

Mise en place

On a implémenté les différentes parties de notre outil avec pour chacune une classe décrite entièrement dans un fichier `.hh` et un fichier `.cc` si ce n'est pas une classe `template` (et on construit un objet `.o` pour cette classe).

Voici un graphe d'inclusion qui représente ces classes selon l'inclusion de leur *header*.

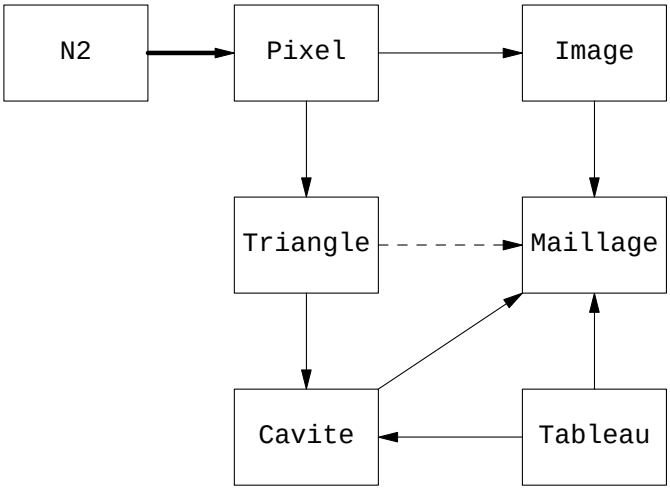


Tableau (`Tableau<T>`)

Cette classe `template` nous a fait un exercice sur les classes génériques: on peut accéder à ou modifier une liste d'élèments génériques. On réduit le nombre d'allocations nécessaires au passage à l'échelle en séparant les concepts de *taille* et de *capacité*.

La méthode `retaille` peut faire appel à `reserve` si on demande une taille qui excède la capacité (on choisit de ne pas *réallouer vers le bas*). Comme les objets stockés sont génériques, on ne fait pas de réallocation à la `realloc`, mais on fait un appel à `new` et on boucle avec des appels à `operator=`. De même lorsqu'on supprime un élèment on ne fait que le remplacer par le dernier; on fait encore un appel à `operator=` et on appelle le constructeur générique pour le dernier élèment déplacé. C'est relativement lent par rapport à des fonctions type `memcpy` mais on préfère préserver l'usage *POO*.

N2

On implémente ici une classe très simple représentant des coordonnées entières à deux dimensions, et quelques opérations de base.

Par exemple, `operator==` rendait aisé de tester si un point était présent dans un maillage sans avoir à implémenter `bool *_pixels_presents`. C'est par contre une méthode plus lente qu'avec ce tableau.

Pixel

Cette classe hérite de `N2` et lui ajoute une valeur de couleur *typedef'd* à un `int`. Sachant qu'on travaille avec le format `.pgm`, on aurait pû stocker la couleur dans un `unsigned char`, comme ce sera le cas dans une `Image`, mais pour pouvoir manipuler les `Pixel`s avec aisance on a fait ce choix.

On a ainsi prévu que pour le calcul d'un barycentre on voudra pouvoir écrire :

```
Pixel p = (p1 + p2 + p3) / 3;
```

Sans se soucier d'*overflow* sur les `Pixel`s intermédiaires.

La couleur de ce `Pixel` est C telle que définie plus haut.

Triangle

Cette classe représente un rassemblement de trois `Pixel`s et c'est à ce niveau qu'on implémente en particulier la méthode `bool cercleCirconscritContient(N2)`, au coeur de notre algorithme de maillage. Pour cette méthode on calcule le déterminant d'une matrice en stockant certaines valeurs uniquement par soucis d'écriture, on laisse le compilateur éviter les répétitions inutiles en général.

Cavité

On construit la région à mettre à jour dans notre modèle comme à la fois un tableau des triangles à supprimer (ceux dont le cercle circonscrit contient le point à ajouter) et un tableau des pixels à partir desquels on va insérer les nouveaux triangles.

C'est le constructeur `Cavite::Cavite(const Tableau<Triangle>&, const Pixel&)` qui détermine ces tableaux à partir des triangles de notre maillage et du point qu'on veut ajouter.

Image

Cette classe est quasiment une copie de la classe `PGMImage` fournie dans les documents de l'énoncé. On a décidé de ne pas séparer les concepts d'image et de lecteur de fichier image car notre but n'était pas d'implémenter un outil pour un panel varié d'images dans le sens abstrait, mais de tester si la méthode de compression était intéressante pour des images exemples.

Il n'était ainsi pas nécessaire de donner plus de sens à une image que le format qu'on comptait tester, représenté principalement par l'attribut `unsigned char *_data`.

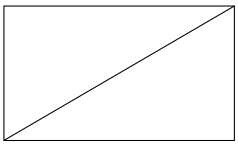
Pour revenir au stockage `int` des couleurs dans la classe `Pixel`, on stocke ici la couleur comme `unsigned char` mais on la distribue comme `int` soit *via* les accesseurs `pixel(...)` soit *via* l'accesseur `operator()(int, int)`.

Le lecteur de fichier `.pgm` se fait *via* le constructeur `Image::Image(const char*)`. Celui-ci fait peu de vérifications quant au contenu, mais encore une fois cet outil n'est pas fait pour un usage très varié.

Maillage

Voici la classe qui représente l'objet qu'on veut construire *in fine*. C'est un tableau de triangles avec l'image sur laquelle on travaille comme attribut intrinséquement lié au maillage qui va le représenter.

On initialise le maillage avec le constructeur `Maillage::Maillage(Image&)` qui pointe `Image* _img` vers l'image étudiée et initialise le maillage par défaut à ses dimensions. Voici le type de maillage initial qu'on construit (deux triangles) :



La méthode `ajoute(const int itermax, const int precision)` itère sur le maillage tant que *l'écart* en chaque triangle est trop important (et que `iter < itermax`). À chaque itération on détermine (parmis les triangles dont le barycentre n'est pas un point d'un triangle du maillage) quel triangle T vérifie le plus grand *écart*. On détermine la cavité à partir de son barycentre, on supprime les triangles mis en relief et on ajoute tous les nouveaux triangles.

La méthode `supprime` réordonne le tableau en sa fin, et on sait quoi supprimer uniquement par son index qu'on veut garder cohérent pendant la suppression. Avec la garantie qu'on a une liste croissante des index des triangles à supprimer, on itère en arrière sur ce tableau et on garantit qu'on a supprimé uniquement les bons triangles.

La couleur C du barycentre ne nous intéresse plus, on voudra insérer la couleur de l'image en son point avec les coordonnées du barycentre. `( p = Pixel(p, (*_image)(p)); )`

On construit les nouveaux triangles à partir d'un tableau de pixels. On fait bien attention à les construire dans le sens trigonométrique et à ne pas oublier le triangle qui contient les dernier et premier points du tableau.

Une fois qu'on a suffisement itéré, on peut `sauvegarder` le maillage dans un fichier `.vtk` qu'on peut visualiser avec *Paraview*. Plusieurs captures d'écran en `.png` des résultats sont à la racine du projet.

Conclusion

On arrive dans ce projet à construire l'outil demandé. La construction des maillages n'est pourtant pas vraiment satisfaisante:

• L'image telle qu'on peut l'afficher après compression a perdu beaucoup de qualité

• Pour une précision suffisamment correcte (< 10) et dans les limites de notre implémentation, le temps de calcul est beaucoup trop long (≈ 1 minute).

• On n'a pas particulièrement gagné en taille de fichier. Si on avait un format plus spécifique à notre maillage que le format `.vtk` (en binaire, etc...), on pourrait par contre réduire assez clairement la taille du fichier.