Introduction à la Programmation

—

Examen partiel sur machine

—

G1: mathis.petrovich(at)enpc.fr
G2: thomas.belos(at)enpc.fr
G3: clement.riu(at)enpc.fr
G4: nicolas.audebert(at)cnam.fr
G5: pascal.monasse(at)enpc.fr
G6: laurent.bulteau(at)u-pem.fr
G7: abderahmane.bedouhene(at)enpc.fr

—

06/11/20

—

1 Enoncé

1.1 Kaléidoscope

Nous allons à partir d'une image faire un kaléidoscope. On sépare l'image en blocs de dimensions identiques et on applique une permutation (une bijection) entre ces blocs. On sépare alors chacun des blocs obtenus en sous-blocs de la même manière pour les permuter également et ainsi de suite pour les sous-sous-blocs. Au bout de suffisamment d'itérations, on retombe sur l'image initiale (voir figure page suivante).

Utilisez comme projet Imagine++ de base celui de l'exercice 2 (avec l'image). Reportez dans le code en commentaire les questions auxquelles vous répondez (`Q1`, `Q2`, etc). Le type `Img` dans le code s'utilise un peu comme une structure (c'est en fait une classe) représentant un tableau bi-dimensionnel de pixels de type `Color`. Pour une variable `img`, le pixel en haut à gauche est Color c = img(0,0);. On peut écrire aussi par img(1,0)=c; (ces deux lignes copient le pixel haut-gauche dans son voisin à droite). La fonction `load` permet de charger un fichier image en mémoire. Pour créer un nouvelle image *ex nihilo*, utiliser `Img img(w,h);` pour une dimension $w \times h$. On peut demander sa taille à une image par img.width() et img.height(). Le pixel en bas à droite est img(img.width()−1,img.height()−1).

Il est plus important de livrer un code clair (commenté et indenté) et qui compile sans warning, même s'il ne répond pas à toutes les questions. Pour cela, vérifiez à chaque étape que votre programme compile et se lance correctement. À la fin de l'examen, nettoyez votre code (indentation...) et vérifiez qu'il compile. Créez alors une archive contenant votre code, le fichier `CMakeLists.txt` et l'image.

1.2 Outils de base

1. Dans un fichier *séparé*, créer une structure `Bloc` qui sert à désigner un rectangle de l'image (coordonnées entières). Les fonctions de cette section seront placées dans ce fichier.
2. Écrire une fonction `voisinh` qui prend un bloc en argument et renvoie celui juste à sa droite.
3. De même `voisinv` pour le bloc juste en-dessous.
4. Programmer un `operator<=` qui indique si son argument gauche est inclus dans son argument droite.
5. Écrire une fonction `copie` qui prend une image et un bloc pour renvoyer la sous-image correspondante.
6. Écrire une fonction `colle` qui prend une image à la fois source et destination, une image (plus petite) à coller et la position (x, y) du coin haut-gauche de collage.
7. Écrire une fonction `translate` qui prend une image à la fois source et destination, un bloc et le vecteur de déplacement (dx, dy) indiquant où dupliquer ce bloc de l'image (on peut supposer que les zones source et destination ne se recouvrent pas).

8. Sécuriser `copie` et `colle` par des assert vérifiant qu'il n'y a pas de débordement.
9. Écrire une fonction `presente_image` qui affiche l'image en argument et fait une pause d'environ une demi-seconde (remplacer la pause par une attente de clic dans la phase de mise au point du code).

1.3 Effets miroir

10. Écrire une fonction `miroirh` qui prend une image et un bloc et échange les deux moitiés côte à côte (utiliser le copier-coller défini ci-dessus).
11. Écrire une fonction `miroirh` prenant seulement une image et appelant de façon répétée la précédente : on échange les deux moitiés (on présente le résultat), puis les moitiés de chacune (on présente le résultat) et ainsi de suite jusqu'à obtenir l'image initiale inversée horizontalement. Faire une deuxième itération pour retrouver l'image originale. Pour les boucles, utiliser `voisinh` et `operator<=` des blocs.
12. Faire de même avec un miroir vertical.
13. Définir un miroir mixte (horizontal/vertical, fonction `miroirhv`) qui échange les deux moitiés horizontales (présente le résultat), puis chacune des moitiés verticalement (présente le resultat), puis chacun des blocs à nouveau horizontalement et ainsi de suite. On obtient enfin l'image tournée de 180°. Itérer une deuxième fois pour retrouver l'image initiale. Attention, on a besoin dans ce cas de deux boucles imbriquées pour appliquer l'échange de moitiés sur tous les blocs.

1.4 Rotation

14. Définir une fonction `tourne` prenant une image et un bloc, séparant le bloc en quatre quarts et les faisant tourner. On pourra utiliser le principe que pour passer de la liste (a, b, c, d) à (b, c, d, a), on peut faire : tmp=a; a=b; b=c; c=d; d=tmp;
15. La fonction `tourne` prenant simplement une image fait touner les quatre quarts, puis chacun des quatres quarts de ces derniers, et ainsi de suite jusqu'à obtenir l'image tournée de 90°. Itérer quatre fois pour retrouver l'image initiale.

1.5 Bonus

16. Les fonctions `tourne` et `miroirhv` requièrent que les images soient carrées de taille 2^n (c'est le cas de l'image *Lena* fournie, 512×512). Pour une image autre, extraire la plus grande sous-image centrée vérifiant ces conditions.
17. Question algorithmique : pourquoi l'implémentation de ces kaléidoscopes par fonction récursive ne donnerait pas l'effet souhaité ? (répondre en commentaire dans le code).
18. Pour les ambitieux : plutôt qu'appliquer les permutations élémentaires brutalement, on peut les faire de façon progressive en faisant les translations par étapes (translation de n pixels en n translations de 1 pixel) pour un effet plus fluide. Le plus simple est de garder l'image avant translation, d'en faire une copie, copier les blocs dans l'image initiale pour coller dans la copie. Une fois toutes les translations effectuées, on peut remplacer l'image initiale par la copie finale.

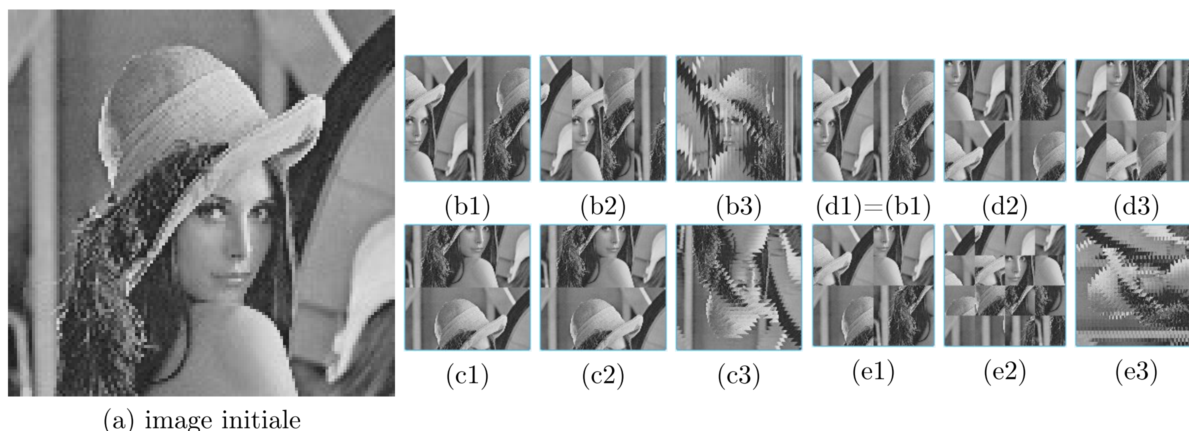

FIGURE 1 – (b) Application de `miroirh` (Q11) aux itérations 1, 2 et plus tard. (c) Idem pour `miroirv` (Q12). (d) Itérations 1, 2, 3 de `miroirhv` (Q13). (e) `tourne` (Q15) aux itérations 1, 2 et plus tard.