MI01 TP5 – Traitement d'image – Première partie.

L'éditeur de logiciels Ladaube® voit les parts de marché de son célèbre logiciel Atelier Photo™ se réduire au profit de son concurrent principal, dont on taira le nom malgré une consonance certaine entre les noms des deux sociétés. Afin d'y remédier, il a été décidé d'optimiser certains traitements faits par le logiciel pour prendre la concurrence de vitesse au sens propre.

Vous avez été embauché en tant qu'expert en programmation IA32 pour accélérer la détection de contours dans une image, qui est le point fort du logiciel.

Le but de ce TP et des suivants sera donc de développer petit à petit en assembleur un logiciel de détection de contours dans une image, en utilisant un algorithme simple.

1 Mise en place

1) Sur le site Moodle de l'UV, téléchargez le fichier projet du TP « tp_image_p1.zip » et copiez le dossier *tp_image* dans votre dossier de projet (sur le disque Z:)
2) Lancez Visual Studio 2005.
3) Dans le menu « Fichier », choisissez « Ouvrir→Projet/Solution… », naviguez jusqu'au dossier où vous avez décompressé le fichier et sélectionnez le fichier « tp_image.sln ». Votre projet est prêt à être utilisé.

Lancez l'exécution du programme. Si le projet est correctement configuré, la génération de l'exécutable doit se faire sans erreur et l'application « Atelier photo » démarre.
Fermez « Atelier photo ».

2 Structure du logiciel

Le logiciel se présente sous la forme de deux modules :

- un module en langage C (*appli.c*) constituant l'interface graphique ; c'est lui qui contient l'appel au traitement que vous allez écrire ;
- un module en assembleur (*image.asm*), que vous allez compléter au fur et à mesure de l'implémentation de l'algorithme.

Appel du traitement assembleur : l'extrait suivant de *appli.c* montre la déclaration du sous-programme assembleur *process_image_asm*, ligne 103, et son appel, ligne 584. C'est l'implémentation de ce sous programme que contient le fichier *image.asm* et que vous allez compléter.

```
 […]
extern UINT __cdecl process_image_asm(UINT biWidth, UINT biHeight, BYTE *img_src,
                            BYTE *img_temp1, BYTE *img_temp2, BYTE *img_dest);
 […]
        case IDM_LAUNCH_ASM:
          QueryPerformanceCounter(&beginProcess);
          for (i = 0; i < nRepeats; i++) {
            process_image_asm(Bitmaps[0]->biWidth,
                            Bitmaps[0]->biHeight,
                            Bitmaps[0]->pBits,
                            Bitmaps[1]->pBits,
[…]
```

Le sous-programme *process_image_asm* prend six paramètres sur la pile en entrée :

biWidth : largeur d'une ligne de l'image en pixels (entier 32 bits, non signé)
biHeight : nombre de lignes de l'image (entier 32 bits, non signé)
img_src : pointeur 32 bits sur les pixels de l'image à traiter
img_temp1, img_temp2 : pointeurs 32 bits sur les pixels de deux images tampon
img_dest : pointeur 32 bits sur les pixels de l'image finale après traitement.

3 Représentation d'une image

Les images traitées par le logiciel sont des images couleur 32 bits. Chaque pixel est représenté par un entier non signé exprimé sur 32 bits contenant les valeurs des composantes rouge (R), verte (V) et bleue (B) du pixel (Figure 1). La valeur de chaque composante est un entier compris entre 0 et 255, 0 étant l'intensité minimale, 255 l'intensité maximale.

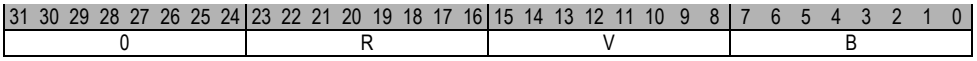

Figure 1 – Format d'un pixel

L'image est un tableau de pixels (Figure 2) dont la taille totale est *Largeur* x *Hauteur*. Les lignes de l'image sont stockées de manière contiguë en mémoire (Figure 3).

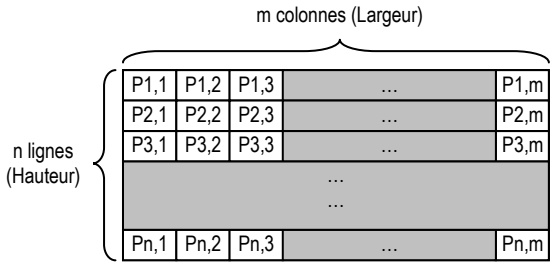

Figure 2 – Format de l'image

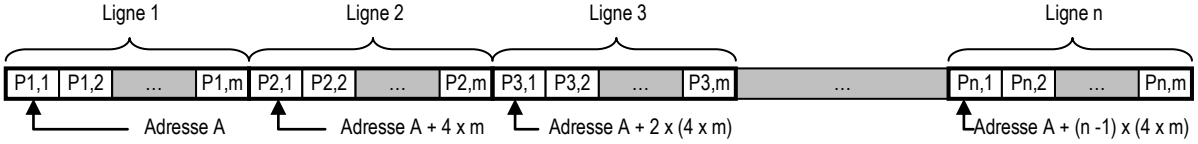

Figure 3 – Stockage en mémoire, A est l'adresse de l'image

4 Travail à réaliser

La détection des contours d'une image couleur impose une première étape : convertir l'image couleur source en niveaux de gris. Dans un premier temps, vous modifierez le sous-programme *process_image_asm* pour transformer l'image fournie dans le tampon *img_src* en niveaux de gris. L'image résultante sera stockée dans le tampon *img_temp1*.

Une implémentation en C est déjà fournie dans le programme *appli.c*, vous pouvez la lancer après avoir chargé une image au format BMP 24 ou 32 bits dans le logiciel (*Fichier->Ouvrir l'image source…*) en utilisant le menu *Traitement->Lancer le traitement C*. Le traitement sera répété autant de fois que vous l'aurez programmé dans le menu *Traitement->Répétitions*… Vous pouvez lancer le traitement en assembleur en utilisant le menu *Traitement->Lancer le traitement assembleur*.

A chaque fin de traitement, une estimation du temps de calcul vous est fournie. Pour obtenir une meilleure estimation, n'hésitez pas à augmenter le nombre de répétions (par exemple 10).

4.1 Squelette du sous-programme process_image_asm.

En vous basant sur la manière dont le module *appli.c* fait appel au sous-programme *process_image_asm*, donnez la signification du contenu des registres ECX, ESI et EDI après leur initialisation dans le sous-programme.

4.2 Conversion en niveaux de gris.

Supposons que les pixels d'une image sont indexés de 1 à n, n étant le nombre total de pixels de l'image.

En supposant que ECX contient l'index d'un pixel donné, donnez l'adresse du pixel correspondant dans l'image source en fonction de ESI et de ECX et son adresse dans l'image de destination en fonction de EDI et de ECX.

Proposez une structure itérative permettant de traiter un à un tous les pixels de l'image source et donnez le code assembleur correspondant. Indication : on peut parcourir l'image source en partant du dernier pixel.

La conversion en niveaux de gris *I* d'un pixel RVB se fait selon la formule suivante :

$$I = R \times 0{,}299 + V \times 0{,}587 + B \times 0{,}114$$

Si chaque composante R, V, et B représente 256 intensités de couleurs différentes, l'intensité résultante *I* comprend elle aussi 256 valeurs différentes. Il est donc possible de représenter l'image résultante avec 256 niveaux de gris régulièrement espacés (l'œil humain n'est pas capable de distinguer beaucoup plus de 50 niveaux d'intensité différents).

Afin de réaliser ce calcul, on représente chaque composante R, V et B comme un réel en virgule fixe sur 16 bits avec un décalage de +8 de la virgule.

Donnez l'expression en hexadécimal des trois coefficients constants utilisées dans le calcul dans ce format. Est-il possible que le résultat du calcul de *I* dépasse 16 bits ? Pourquoi veut-on éviter les débordements ?

Décomposez le calcul de *I* en opérations élémentaires sur des nombres en virgule fixe. Prenez garde aux décalages nécessaires pour les produits. Donnez l'algorithme de calcul résultant. Est-il possible de le simplifier ? Indication : dans la représentation choisie, la partie décimale des composantes R, V et B ainsi que la partie entière des coefficients qui les multiplient sont toujours nuls.

Donnez l'algorithme complet permettant de traiter chaque pixel de l'image source. Le résultat sera stocké dans la composante B de chaque pixel de l'image temporaire 1.

Ecrivez le programme assembleur correspondant. Votre code est-il plus performant que le code généré par le compilateur C ?

Important : le compilateur s'attend à ce que les registres EBX, ESI, EDI, EBP soient préservés par les appels de sous-programmes (en d'autres termes : une fonction C ne sauvegarde ni EAX, ni ECX, ni EDX).