Université de Technologie de Compiègne

MI01

Rapport de TP

5 et 6 - Traitement d'image

Automne 2014

Romain PELLERIN - Kyâne PICHOU
Groupe 1
29 décembre 2014

Table des matières

1 Introduction

L'objectif de ce TP est d'améliorer le traitement d'une image faite en C, en écrivant le traitement en code assembleur. Le traitement à améliorer est la détection de contours dans une image (grâce au masque de Sobel).

Pour commencer, il faudra convertir notre image RGB en niveaux de gris. Ensuite, il faudra effectuer une série de calculs sur cette image afin de détecter les contours. Tout le travail sera réalisé en assembleur afin de le comparer (en terme de temps d'exécution, donc de performance) avec le langage C.

2 Conversion en niveaux de gris

2.1 Présentation

2.1.1 Structure du logiciel

On utilise un petit logiciel fourni pour ce TP. Celui-ci permet d'effectuer le traitement sur l'image en C et en assembleur. Le code C est fourni, on se contentera donc de compléter le code assembleur effectuant le même traitement.

Notre traitement assembleur sera appelé depuis une fonction C. Elle prendra différent paramètres en entrée, passés sur la pile :

- **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 le premier pixel de l'image à traiter
- **img_temp1, img_temp2** : pointeurs 32 bits sur le premier pixel de deux images tampon (intermédiaires)
- **img_dest** : pointeur 32 bits sur le premier pixel de l'image finale après traitement.

2.1.2 Représentation d'une image

Les images traitées par le logiciel sont des images couleurs de 32 bits au format RGB. C'est-à-dire que chaque pixel est représenté sur un entier 32 bits non-signé contenant les différentes composantes de couleurs de ce pixel (Red/Green/Blue). Chaque composante est codée sur 8 bits (0 à 255) selon leur intensité. On obtient donc la structure ci-dessous :

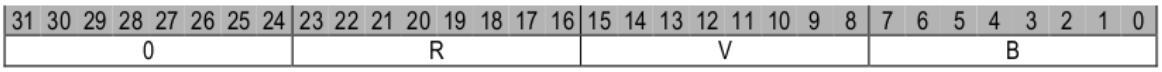


Figure 1 – Format d'un pixel

Une image étant composée de lignes et colonnes de pixels, elle peut être vue ainsi :

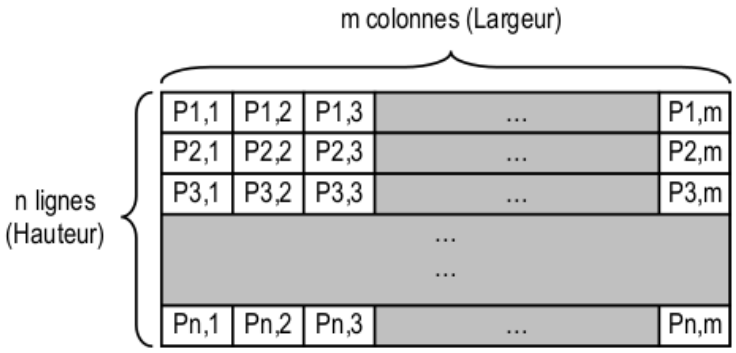


Figure 2 – Format de l'image

En mémoire, notre image sera représentée sous la forme d'un tableau d'entiers. Ce tableau de taille *largeur * hauteur* contiendra bout à bout chaque ligne de l'image.

2.2 Réalisation

2.2.1 Conversion niveau de gris

On réalise le code assembleur de la conversion en niveau de gris. On effectuera une boucle parcourant chaque pixel et effectuant le calcul suivant :

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

Notre pixel en niveau de gris sera toujours stocké sur 32 bits, or on souhaite que la valeur du gris soit stocké dans l'octet de la composante bleue (les autres composantes seront à 0). Il convient donc de vérifier que notre calcul ne provoque pas de débordement (pas plus de 8 bits, soit la composante bleue).

La somme des cœfficients que l'on applique à chaque composante est égale à 1 (0.299+0.587+0.114=1). Donc quelque soit les valeurs d'intensités des composantes, le résultat de conversion sera toujours inférieur ou égal à 255 (stocké sur 8 bits donc). **Il n'y a par conséquent aucun risque de débordement et notre composante de gris sera bien stockée sur 8 bits à la place de la composante bleue.**

Les calculs doivent être effectués en prenant en compte la virgule des coefficients.Les composantes de couleurs seront donc représentées sur 16 bits à virgule fixe avec un décalage de la virgule de +8. Par exemple on écrira 42 ainsi :

0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0

Partie entiere – Poids fort *Partie decimale – Poids faible*

Pour la partie décimale, il faut faire en sorte que celle-ci tienne sur 8 bits. On peut la décomposer sous la forme :

$$Partie\ decimale = 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-6} + 2^{-7} + 2^{-8} \tag{2.1}$$

Soit :

$$Partie\ decimale \times 256 = 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 \tag{2.2}$$

Ce qui nous permet donc de représenter le nombre sur 8 bits. Cependant on effectue une petite erreur inévitable de l'ordre de 2^{-8} lorsque l'on décalera de 8 bits à nouveau, après les calculs.

Après conversion et multiplication par 256 on trouve les valeurs héxadécimales arrondies suivantes :

Coef. du rouge	0.299	0x4D
Coef. du vert	0.587	0x96
Coef. du bleu	0.114	0x1D

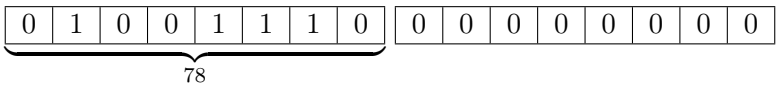
Ces valeurs sont celles que l'on mettra dans la partie décimale des coefficients (la partie entière étant 0).

Avant de produire le code assembleur, on s'intéresse au déroulement de la conversion d'un pixel en niveaux de gris. On prendra pour exemple le pixel suivant :

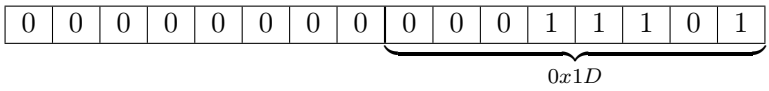
0	127	204	78

R *V* *B*

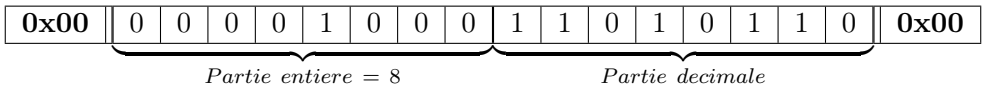
On veut obtenir le niveau de gris dans la composante bleu. On traite la composante bleue :



×



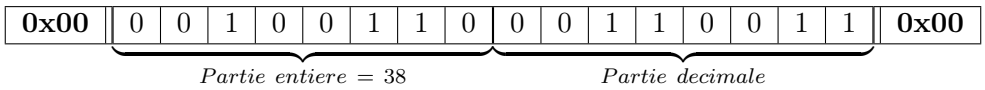
=



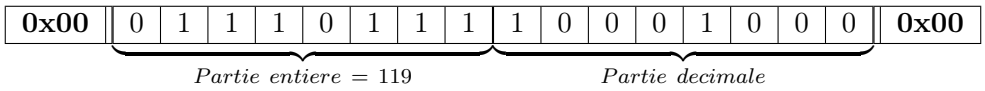
Du fait de la multiplication entre 2 nombres de 16 bits, on a un résultat sur 32 bits avec une virgule décalée de 16 bits. Cependant on notera que les 8 bits de poids forts sont toujours à zéro (la partie entière du résultat est toujours inférieure à 255). De plus le résultat attendu en niveau de gris est un entier, donc la partie décimale sera ignorée dans le résultat final.

En répétant l'opération sur les composantes verte et rouge on obtient les 2 mots de 32 bits suivants :

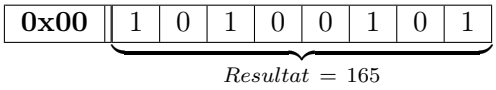
Rouge :



Vert :



On souhaite maintenant obtenir un niveau de gris sur 8 bits. On additionne donc nos 3 résultats de 32 bits et on effectue un décalage de 16 bits à droite pour éliminer la partie décimale. On obtient donc :



On ne manipulera que les 8 bits de poids faible qui constituent le résultat. Ici on a 165 tandis que la valeur théorique du calcul donne 166,613. Il y a donc bien une erreur provoquée par la manipulation utilisée pour considérer les parties décimales. Cependant l'œil humain ne différencie gère plus que 50 intensités différentes, et compte tenu de notre faible erreur, ceci n'est à priori pas visible à l'œil nu.

2.2.2 Code assembleur

On réalise ensuite notre programme en assembleur. **Dans les faits, nous manipulons les composantes de couleur sur 8 bits ainsi que les cœfficients de multiplication sur 8 bits, par conséquent nous obtenons un résultat de multiplication sur 16 bits, et nous ne conservons que les 8 bits de poids fort.**

Assembleur

```
; CODE FOURNI
; ...

    ;*************************************************************
    ;*************************************************************
    ; Ajoutez votre code ici
```

```
    ;*************************************************************
    ;*************************************************************

    dec ecx
    imul ecx,4

boucle:
    movzx ebx, byte ptr [esi+ecx] ; on stocke la valeur du bleu
    imul ebx, 29d ; 29 = 0.114*256
    mov eax,ebx

    movzx ebx, byte ptr [esi+ecx+1] ; on stocke la valeur du vert
    imul ebx, 150d ; 150 = 0.587*256
    add eax,ebx

    movzx ebx, byte ptr [esi+ecx+2] ; on stocke la valeur du rouge
    imul ebx, 77d ; 77 = 0.299*256
    add eax,ebx
    shr eax, 8 ; décalage de 8 bits pour ne conserver que la partie entière

    mov [edi+ecx], eax ; on sauvegarde la valeur du pixel que l'on copie dans l'
        image de destination

    sub ecx,4
    jne boucle
fin:

; ...
; CODE FOURNI
```

On test le programme en C et en assembleur et on obtient les temps suivants (à gauche en C et à droite en assembleur) :

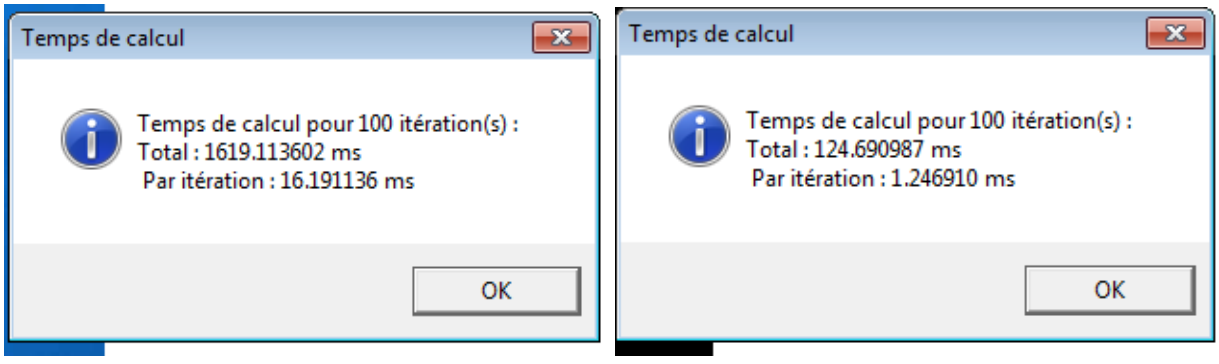


FIGURE 2.1 – Temps conversion niveaux de gris

On constate donc une très grande différence de temps d'execution. Notre code assembleur est 10 fois plus rapide que la fonction C. Il est fonctionnel (comparaison visuelle avec le traitement en C).

Pour obtenir une image en vrai noir et blanc (et non uniquement sur la composante bleue comme dans le sujet), il suffit de rajouter ce bout de code à la fin de notre code, qui copie la valeur du gris dans les composantes rouge et verte :

Assembleur

```
; ...
shr eax,8

; VRAI NOIR ET BLANC
mov ah,al
shl eax,8
mov al,ah

mov [edi+ecx], eax
; ...
```

Le résultat que l'on obtient est le suivant :

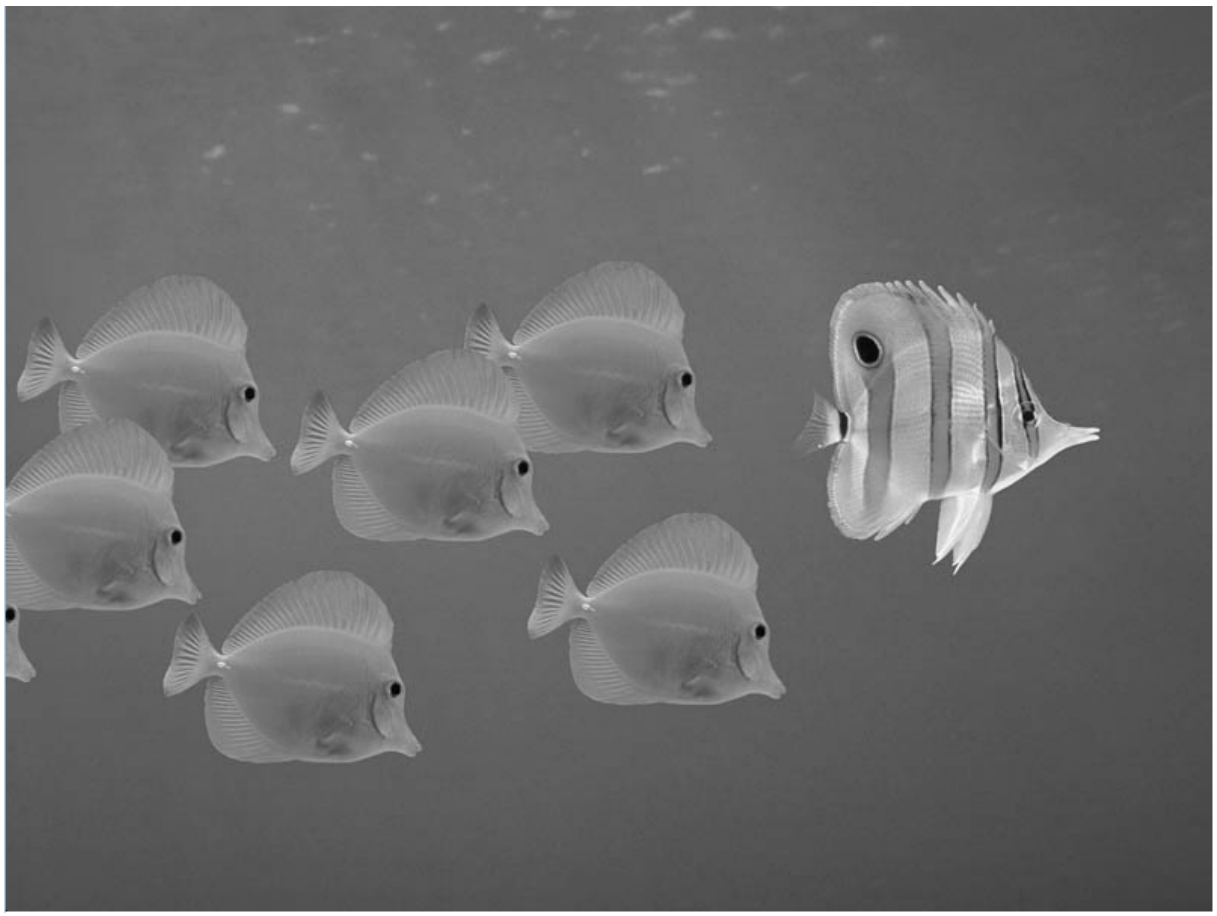


FIGURE 2.2 – Image en niveaux de gris

Nous pouvons donc passer à la partie suivante, à savoir la détection des contours dans l'image.

3 Détection des contours

3.1 Présentation

La détection des contours au sein d'une image est un traitement d'image assez classique, très utile pour la détection/reconnaissance d'objets par exemple.

3.1.1 Principe de détection

Les zones de contours sont caractérisées par un fort contraste. La détection de contours se réduit donc à la détection des zones à fort contraste. Pour se faire il y a 2 grands types de méthodes : l'analyse du laplacien de l'image et l'analyse du gradient. L'analyse du gradient sera implémentée ici.

On utilisera l'opérateur de Sobel qui permet une mesure, en deux dimensions, du gradient de l'intensité d'une image en niveaux de gris. Cet opérateur se présente sous la forme de 2 masques de convolution S_x et S_y qui fournissent le gradient de chaque pixel (respectivement dans les directions x et y).

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \text{ et } G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

La norme G du gradient est calculée comme suit :

$$|G| = \sqrt{G_x^2 + G_y^2} \tag{3.1}$$

Mais pour faciliter les calculs on nous propose de simplifier comme suit :

$$|G| = |G_x| + |G_y| \tag{3.2}$$

Le gradient de chaque pixel sera donc calculé en appliquant chaque masque sur chaque pixel, ligne par ligne. Un exemple nous est proposé :

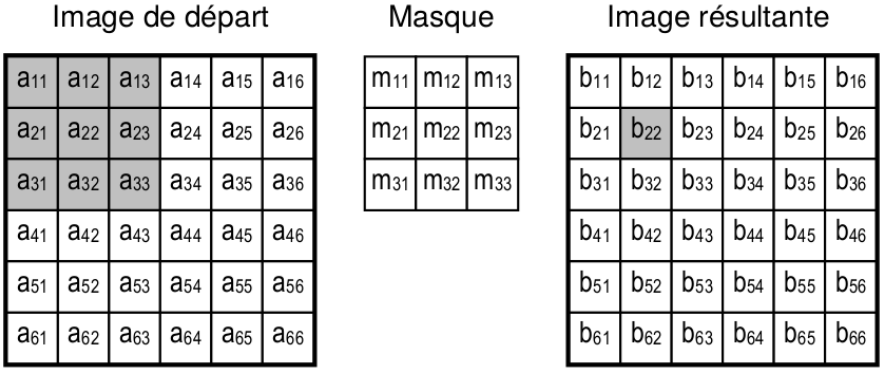


FIGURE 3.1 – Applicaton du masque

Ici la valeur du pixel b_{22} est :

$$b_{22} = m_{11}a_{11} + m_{12}a_{12} + m_{13}a_{13} + m_{21}a_{21} + m_{22}a_{22} + m_{23}a_{23} + m_{31}a_{31} + m_{32}a_{32} + m_{33}a_{33} \quad (3.3)$$

Evidemment les lignes et colonnes en bordure de l'image ne pourront pas être traitées par un masque de dimension 3x3, les pixels correspondant ne seront donc pas pris en compte (ils seront laissés noirs).

3.1.2 Algorithme de traitement

On va donc parcourir tout les pixels de l'image source pour calculer le gradient et le mettre dans le pixel correspondant de l'image de destination :

1. On applique G_x et G_y
2. On somme leurs valeurs absolues
3. On veut afficher l'image en noir sur fond blanc donc on "inverse" les pixels en faisant $255 - G$
4. Si la valeur est négative alors on met le pixel à 0
5. On stocke la valeur dans le pixel de l'image de destination

3.2 Réalisation

On va donc réaliser notre fonction de traitement. On considère que le niveau de gris de chaque pixel se trouve dans la composante bleue de celui-ci. On utilisera les registres de la manière ci-dessous :

— **ESI** : Adresse du premier pixel (de la matrice) source

— **EDI** : Adresse du pixel de destination

— **EBP** : Taille d'une ligne en pixel (sauvegarde du pointeur de pile avant cela)

— **ECX** : Poids forts (16 bits) = compteur de lignes restantes = nombre de lignes - 2
Poids faibles (16 bits) = compteur de colonnes restantes = nombre de colonnes - 2

— **EBX** : Valeur de G_x

— **EDX** : Valeur de G_y

Pour respecter cette organisation on initialise les registres servant à enregistrer les adresses sources et destination :

Assembleur

```
mov    esi, [ebp + 20]
mov    edi, [ebp + 24]
```

On initialisera `ebp` à `[EBP+8]` (la longueur d'une ligne) uniquement après avoir initialisé les autres registres. En effet `ebp` nous sert aussi de sauvegarde du pointeur de pile et est nécessaire pour récupérer les différents paramètres.

3.2.1 Double itération

Comme indiqué précèdement, le registre `ecx` sera utilisé en tant que double compteur. Les poids forts (16 bits) compteront les lignes restantes à traiter et les poids faibles les colonnes restantes à traiter.

Itération sur les lignes

On initialise les poids forts ainsi :

Assembleur
```
mov ecx, [ebp + 12] ; On déplace le nombre de ligne dans ecx
sub    ecx, 2 ; On soustrait de 2 (suppresion des bords)
shl    ecx,16 ; On décale la valeur sur les 16 bits de poids forts
```

En fin de boucle on décrémente les bits de poids forts et on vérifie que ce n'est pas fini, sinon on reboucle :

Assembleur
```
sub ecx, 65536 ; decremente de 1 les 16 bits de poids fort
cmp ecx,0 ; On compare avec zéro
jne traitement_ligne ; Si ce n'est pas égal à zéro on lance un nouveau
   traitement de ligne
```

À la fin du traitement d'une ligne `edi` et `esi` doivent avancer de deux pixels d'un coup pour passer à la ligne suivante et sauter la dernière colonne et la première colonne de pixels.

Itération sur les colonnes

Pour les colonnes on utilisera les 16 bits de poids faibles de `ecx` soit le registre `cx`. Au début d'une boucle sur une ligne on l'initialise au nombre de colonnes à traiter, soit la longueur d'une ligne (valeur stockée dans `ebp`) en enlevant 2 colonnes (la première et la dernière).
Dans cette boucle, après avoir traité un pixel, on passe au pixel suivant. On avance donc `edi` et `esi` d'un pixel soit 4 octets.

Implémentation de la boucle

On obtient donc notre structure de boucle suivante :

Assembleur
```
; ****** Détection de contours - boucle itérative ******
mov    esi, [ebp + 20] ; tmp1
mov    edi, [ebp + 24] ; tmp2

; On stocke le nombre de lignes à traiter dans les bits de poids fort de ECX (
   hauteur)
mov ecx, [ebp + 12]
sub    ecx, 2
shl    ecx, 16

mov ebp, [ebp + 8] ; on stocke la longueur d'une ligne = le nombre de colonnes =
    largeur
; On saute la première ligne et on passe à la deuxième case (+4)

lea edi, [edi+ebp*4+4]

traitement_ligne:
add ecx, ebp ; nombre de colonnes à traiter
sub cx, 2
```

```
    traitement_colonne:

    ; ***************************
    ; Traitement pixel ici
    ;***************************

    ; Après avoir traité le pixel, on avance d'un pixel
    add edi, 4
    add esi, 4
    dec cx ; on décremente le nombre de colonnes à traiter
    cmp cx, 0
    jne traitement_colonne

; Passage à la deuxième case de la ligne suivante
add edi, 8
add esi, 8

sub ecx, 65536 ; decremente de 1 les 16 bits de poids fort <=> decremente
    compteur de lignes
cmp ecx,0
jne traitement_ligne
```

3.3 Calcul du gradient

3.3.1 Implémentation

Il faut maintenant programmer le coeur de l'application : le traitement du pixel. On stockera G_x dans `ebx` et G_y dans `edx`.

Si le registre `esi` contient l'adresse de a_{11} de l'image source (et que `ebp` contient la longueur d'une ligne), alors l'adresse de :

- a_{12} : `[esi + 4]` => On avance d'un pixel (4 octets)
- a_{21} : `[esi + ebp*4]` => On avance d'une ligne (longueur ligne en pixels*4)
- a_{31} : `[esi + ebp*8]` => On avance de 2 lignes (2*longueur ligne*4)

Pour effectuer le calcul de G_x et G_y sur un pixel on multipliera l'intensité de chaque pixel par l'élément correspondant de la matrice puis on sommera les différents résultats. Par exemple pour G_x :

Assembleur

```
mov ebx, [esi]
mov eax, [esi+ebp*4]
imul eax, 2
add ebx, eax
add ebx, [esi+ebp*8] ; on saute 2 lignes
neg ebx
add ebx, [esi+8]
mov eax, [esi+8+ebp*4]
imul eax, 2
add ebx, eax
```

```
add ebx, [esi+8+ebp*8]
```

Le principe est le même pour le calcul de G_y, seuls les cœfficients changent.

Après le calcul des coefficients, il est nécessaire de récupérer la valeur absolue. Pour G_x on obtient :

Assembleur

```
add ebx, [esi+8+ebp*8] ; Dernière opération arithmétique du calcul de Gx
jns pasneg ; Si le résultat n'est pas négatif on saute la prochaine instruction
neg ebx ; Sinon on change le signe (on le rend donc positif)
pasneg:
; Suite du programme
```

On somme ensuite les 2 composantes et on effectue la fin de l'algorithme : normalisation par rapport à 255 et affichage en niveaux de gris « négatif » (bordure noir sur fond blanc).
On utilise le code ci-dessous :

Assembleur

```
add ebx, edx ; On additionne les 2 composantes

neg ebx
add ebx,255 ; ebx = 255-ebx
cmp ebx,0
jg endd
mov ebx,0

endd:
mov eax,ebx
shl eax,8 ; Via plusieurs décalage, on met la valeur du gradient dans chaque
    composante du pixel
add ebx,eax
shl eax,8
add ebx,eax
mov [edi], ebx ; On met le pixel dans l'image de destination
```

3.3.2 Résultat

On teste donc le code complet (disponible en Annexe) et on obtient le bon résultat (voir figure ci-dessous).

On détecte correctement les contours de l'image et on affiche bien le résultat en contours noirs sur fond blanc. On compare les temps d'execution (en C à gauche en assembleur à droite) sur la figure ci-dessous.

On constate que notre code assembleur est bien plus rapide que le traitement en C. Pour 100 itérations on gagne une seconde de traitement.

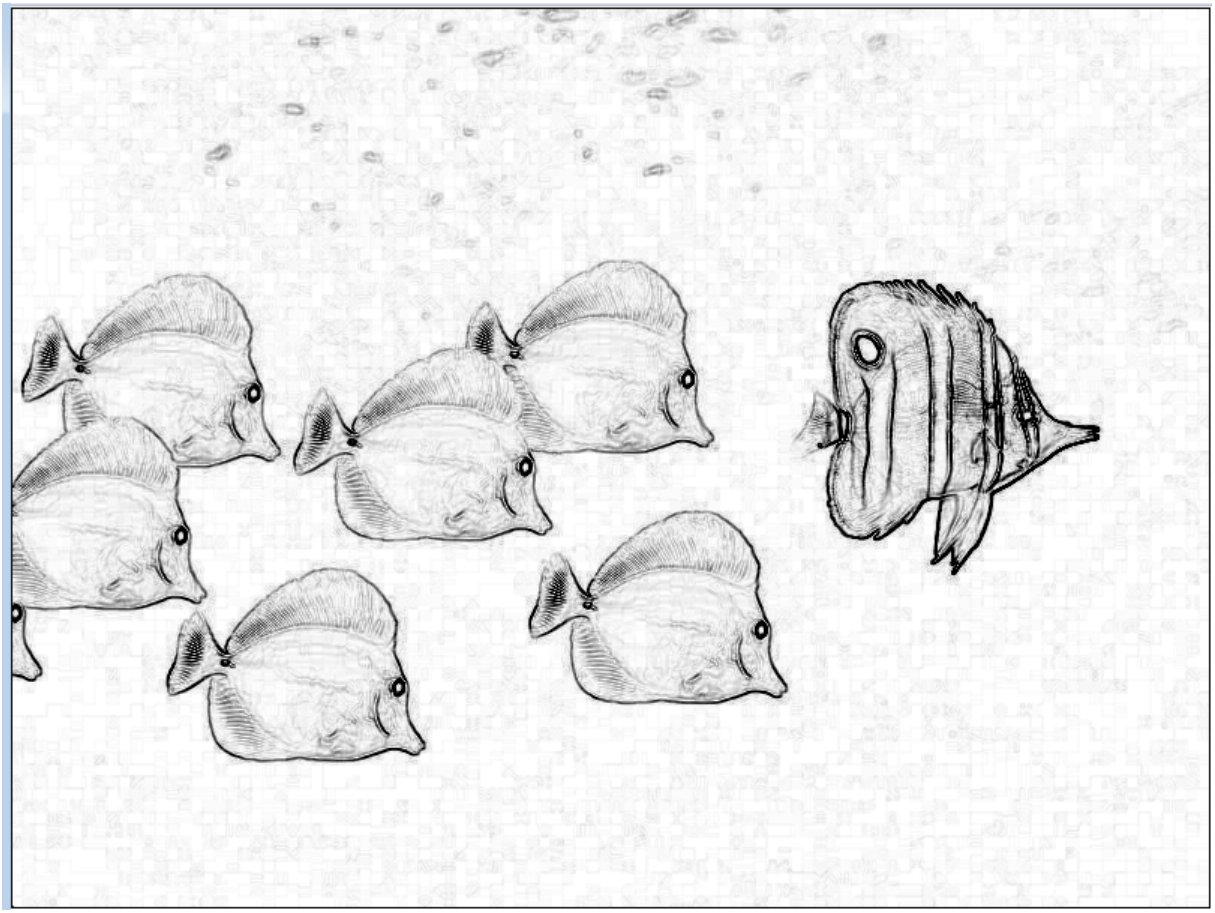


FIGURE 3.2 – Détection des contours

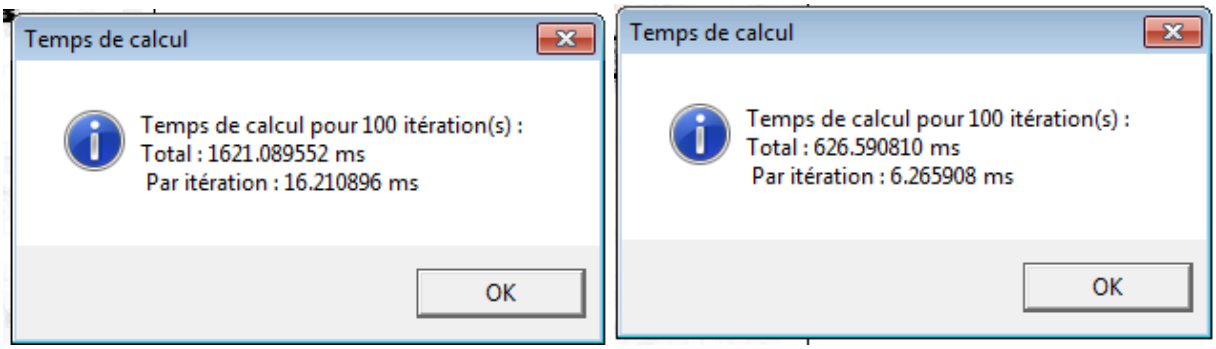


FIGURE 3.3 – Temps de détection totale des contours

4 Conclusion

Après cet exercice de programmation, on remarque que pour certains types de traitement de données (ici d'une image) l'optimisation du code en assembleur offre un réel gain en performance.

A Code final

Voici le code final des TP 5 et 6 :

Assembleur

```
.686
.MODEL FLAT, C
.DATA
.CODE
PUBLIC     process_image_asm
process_image_asm PROC NEAR      ; Point d'entrée du sous programme

push    ebp         ; construction du cadre de pile (sauvegarde de ebp)
mov     ebp, esp    ; sauvegarde du pointeur de pile
; SAUVEGARDE DES REGISTRES
push    ebx
push    esi ; image source
push    edi ; image tmp1
; multiplication de la largeur de l'image par la hauteur
mov     ecx, [ebp + 8]
imul    ecx, [ebp + 12]

mov     esi, [ebp + 16] ; esi = image source (adresse du premier pixel de l'image
    )
mov     edi, [ebp + 20] ; edi = image tmp1

dec ecx
imul ecx,4

boucle:
    movzx ebx, byte ptr [esi+ecx] ; on stocke la valeur du bleu
    imul ebx, 29d ; 0.114*256
    mov eax,ebx

    movzx ebx, byte ptr [esi+ecx+1] ; on stocke la valeur du vert
    imul ebx, 150d ; 0.587*256
    add eax,ebx

    movzx ebx, byte ptr [esi+ecx+2] ; on stocke la valeur du rouge
    imul ebx, 77d ; 0.299*256
    add eax,ebx
    shr eax,8

    mov [edi+ecx], eax ; on sauvegarde la valeur du pixel que l'on copie dans l'
        image de destination
    sub ecx,4
    jne boucle
```

```
    ; ****************************** TP6 ******************************

    mov     esi, [ebp + 20] ; tmp1
    mov     edi, [ebp + 24] ; tmp2
    ; On stocke le nombre de lignes à traiter dans les bits de poids fort de ECX
        (hauteur)
    mov ecx, [ebp + 12]
    sub     ecx, 2
    shl     ecx,16
    mov ebp, [ebp + 8] ; on stocke la longueur d'une ligne = le nombre de
       colonnes = largeur
    lea edi, [edi+ebp*4+4]

traitement_ligne:
    add ecx, ebp ; nombre de colonnes ? traiter
    sub cx, 2

traitement_colonne:
    ; Gx
    mov ebx, [esi]
    mov eax, [esi+ebp*4]
    imul eax, 2
    add ebx, eax
    add ebx, [esi+ebp*8] ; on saute 2 lignes
    neg ebx
    add ebx, [esi+8]
    mov eax, [esi+8+ebp*4]
    imul eax, 2
    add ebx, eax
    add ebx, [esi+8+ebp*8]
    jns pasneg
    neg ebx
pasneg:
    ; Gy
    mov edx, [esi+ebp*8]
    mov eax, [esi+4+ebp*8]
    imul eax, 2
    add edx, eax
    add edx, [esi+8+ebp*8]
    neg edx
    add edx, [esi]
    mov eax, [esi+4]
    imul eax,2
    add edx, eax
    add edx, [esi+8]
    jns pasneg2
    neg edx
pasneg2:
    add ebx, edx
    neg ebx
    add ebx,255
```

```
    cmp ebx,0
    jg endd
    mov ebx,0
endd:
    mov eax,ebx
    shl eax,8
    add ebx,eax
    shl eax,8
    add ebx,eax
    mov [edi], ebx

; On avance d'une case
    add edi, 4
    add esi, 4
    dec cx ; on décremente le nombre de colonnes à traiter
    cmp cx, 0
    jne traitement_colonne
; Passage à la deuxième case de la ligne suivante
    add edi, 8
    add esi, 8
    sub ecx, 65536 ; decremente de 1 les 16 bits de poids fort <=> decremente
        compteur de lignes
    cmp ecx,0
    jne traitement_ligne
fin:
    pop     edi
    pop     esi
    pop     ebx
    pop     ebp
    ret    ; Retour à la fonction MainWndProc
process_image_asm ENDP
END
```