TP06/TP07 - Traitement d'image - deuxième partie

Geoffrey PERRIN
Océane DUBOIS

Le but de ce TP est de construire un algorithme permettant de détecter les contours d'un image. Pour ce faire on utilisera la méthode du gradient, qui consiste à réaliser la dérivée de l'intensité des pixels, si on observe un maximum, on considère que c'est un contour.

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Figure 1: Masque de convolution de Sobel Sx

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Figure 2: Masque de convolution de Sobel Sy

Pour nous aider à retrouver le gradient des pixels de l'image, on utilisera l'opérateur de sobel qui est composé de 2 masques de convolution Sx et Sy définis par les figures 1 et 2.

Attention, on ne pourra pas appliquer les masques sur la première et dernière ligne de pixel ni sur la première et dernière colone de pixel car le masque dépasserait de l'image. Pour le TP on ignorera cet lignes et colones.

0.1 Calcul des adresses source et destination

Soit esi contenant l'adresse du premier pixel auquel on applique le masque et edi est l'adresse du premier pixel de l'image de destination (img_temp2). On stocke dans ebp la taille d'une ligne de pixels.

Ainsi ecx contient la hauteur de l'image, à laquelle on soustrait 2 pour ignorer la première et la dernière ligne.

Esi contient l'adresse du premier pixel de l'image tmp1 Edi contient l'adresse du premier pixel de l'image tmp2 Ebp contient le nombre de pixel sur la ligne d'une image.

Pour connaitre la taille d'une ligne il suffit de multiplier Ebp par 4(la taille d'un pixel en mémoire).

0.2 Contruction de la double itération

Dans le but d'économiser un registre, ecx contiendra sur la partie haute le compteur de lignes et sur sa partie base le compteur de colones.Ainsi on peut récupérer facilement le comtpeur des lignes dans cx.

0.3 Itération sur les lignes

Au début de la boucle on initialise les bits de poids fort avec ebp -2, pour ignorer les premières et dernièrers lignes.

A la fin de chaque ligne on soustrait 00010000h à ecx. Lorsque les bits de poids forts de ecx arrivent à 0, on arrête la boucle. Cela signifie qu'on a traité tous les pixels de l'image.

0.4 Itération sur les colones

On place donc dans la partie cx du registre ecx, le nombre de colones, stockées dans ebp.

A la fin de chaque colone, on décrémente ecx de 1, lorsque cx est égale à 0 on ira décrémenter la partie haute de ecx.

0.5 Calcul du gradient de chaque pixel

Le gradient de chaque pixel est calculé comme suit : Sur chaque pixel de l'image source, on applique la valeur du masque Sx et on stock le résultat de la somme de tous les pixels dans ebx. On réalise la même opération avec le masque Sy et on stocke le résultat dans edx.

Puis on doit trouver les valeurs absolues. On test si le résultat de ebx est positif, si il l'est on saute plus loin dans le programme, si il ne l'est pas on lui applique l'opérateur "neg", qui permet de rendre ebx positif.

On réalise la même opération avec edx. Puis lorsqu'on est sûrs que le résultat de ebx et de edx est positif, on somme les 2. Si le résultat des 2 est positif on saute a g_positif.

On compare donc ebx à 255, si le résultat est plus petit que 255 on passe à g_negatif car le résultat est inférieur à l'intensité maximale. Sinon on met le pixel à 255 qui est donc l'intensité maximale.

Puis pour inverser les couleurs, on prend le négatif de G(=ebx) et on lui ajoute 255.

On récupère ensuite chaque composante (R, V et B) et on les somme entre-elles. Puis on place ce résultat dans le pixel de l'image source.

0.6 Code

Voici le code que nous avons implémenté :

```
1
2  ;
3  ; MI01 - TP Assembleur 2 ? 5
4  ;
5  ; Réalise le traitement d'une image 32 bits.
6
7  .686
8  .MODEL FLAT, C
9
10 .DATA
11
12 .CODE
13
14 ;
       ********************************************************************
15 ; Sous-programme _process_image_asm
16 ;
17 ; R?alise le traitement d'une image 32 bits.
18 ;
19 ; Entr?es sur la pile : Largeur de l'image (entier 32 bits)
20 ;                       Hauteur de l'image (entier 32 bits)
21 ;                       Pointeur sur l'image source (d?pl. 32 bits)
22 ;                       Pointeur sur l'image tampon 1 (d?pl. 32 bits)
23 ;                       Pointeur sur l'image tampon 2 (d?pl. 32 bits)
24 ;                       Pointeur sur l'image finale (d?pl. 32 bits)
25 ;
       ********************************************************************
26 PUBLIC        process_image_asm
27 process_image_asm    PROC NEAR       ; Point d'entrée du sous
       programme
28
29     push     ebp
30     mov      ebp, esp
31
32     push     ebx
33     push     esi
34     push     edi
35
36
```

```
37          ;récupération des arguments dans les différents
        registres
38          mov     ecx, [ebp + 8]
39          imul    ecx, [ebp + 12]
40
41          mov     esi, [ebp + 16]
42          mov     edi, [ebp + 20]
43
44          xor     edx,edx
45          ;début de la boucle
46  boucle: dec       ecx
47          ;on récupére le pixel de l'image source à traiter
48          mov     ebx,[esi+ecx*4]
49
50          ;--------------------
51          ;calcul de B * 0,114
52          ;--------------------
53
54          mov       eax,ebx
55          ;on récupere la composante bleu dans eax
56          and       eax,000000FFh
57          ;on effectue le calcul
58          imul    eax,1Dh ;B * 0,114
59          ;on stock la somme (I) dans edx
60          mov       edx,eax
61
62          ;--------------------
63          ;calcul de V *,0,587
64          ;--------------------
65
66          ;on récupere la composante verte dans eax
67          mov       eax,ebx
68          and       eax,0000FF00h
69          shr     eax,8
70          ;on effectue le calcul
71          imul    eax,96h ;V *,0,587
72          add       edx,eax ;I=I+V *,0,587
73
74          ;--------------------
75          ;calcul de R*0,299
76          ;--------------------
77          ;on récupere la composante rouge dans eax
78          mov       eax,ebx
79          and       eax,00FF0000h
80          shr     eax,16
81          ;on effectue le calcul
82          imul    eax,4Ch
83          add       edx,eax ;I=I+R*0,299
84
```

```
85          ;on divise par 256
86          shr     edx,8
87          ;on stock I dans la composante bleu de l'image
        destination
88          mov     [edi+ecx*4],edx
89
90
91          cmp     ecx, 0
92          ja      boucle
93
94  ;tp6
95    mov ecx,[ebp+12];hauteur
96    sub ecx,2
97    shl ecx,16
98
99    mov      esi,edi;img tmp1
100   mov      edi, [ebp + 24];img tmp2
101   mov      ebp, [ebp + 8] ; largeur
102   mov    eax,ebp  ;eax=largeur
103   shl    eax,2    ; on multiplie la largeur par 4 ;taille  d'une
        ligne
104   add    edi,eax ; on ajoute au premier pixel de l'image tmp2 4*
        largeur
105   add    edi, 4
106
107
108
109 boucle3:  ;itération sur les lignes
110   add    ecx,ebp
111         sub      ecx,2
112
113 boucle2:  ;itération sur les colones
114
115   ;--------------
116
117     ;calcul de Gx avec le masque de convolution Sx (on effectue
        pas les multiplications par 1 pour économiser des
        instructions). Le résultat est stocké dans ebx, on effectue
        les calculs intermédiaires dans eax.
118
119     xor    ebx, ebx
120     mov    ebx, [esi]
121     imul   ebx,-1
122     add    ebx,[esi+8]
123     mov    eax,[esi+ebp*4]
124     imul   eax,-2
125     add    ebx,eax
126     mov    eax,[esi+ebp*4+8]
127     imul   eax,2
```

```
128     add    ebx,eax
129     mov    eax,[esi+ebp*8]
130     imul   eax,-1
131     add    ebx,eax
132     add    ebx,[esi+ebp*8+8]
133
134
135   ;calcul de Gy avec le masque de convolution Sy (on effectue
        pas les multiplications par 1 pour économiser des
        instructions). Le résultat est stocké dans edx, les valeurs
        temporaires sont dans eax.
136
137
138     xor    edx, edx
139     mov    edx, [esi]
140     mov    eax,[esi+4]
141     imul   eax,2
142     add    edx, eax
143     mov    eax,[esi+8]
144     add    edx, eax
145     mov    eax,[esi+ebp*8]
146     imul   eax,-1
147     add    edx,eax
148     mov    eax,[esi+ebp*8+4]
149     imul   eax,-2
150     add    edx,eax
151     mov    eax,[esi+ebp*8+8]
152     imul   eax,-1
153     add    edx,eax
154
155
156       cmp  ebx,0  ; on vérifie si le résultat de Gx est négatif
157       jg     gx_positif ; si il ne l'est pas on passe à "
        gx_positif"
158       neg     ebx  ;si le résultat est négatif on le passe en
        positif car on souhaite la valeur absolue.
159
160   gx_positif :
161     cmp edx, 0  ; on fait de meme pour Gy, on vérifie si il est
        négatif, si il l'est on le passe en positif
162     jg  gy_positif
163     neg edx
164
165   gy_positif :
166     add ebx, edx ; on ajoute ensemble la valeur absolue de Gx et
         Gy
167
168
169     xor eax, eax
```

```
170     mov eax, 255
171     sub eax, ebx  ;pour inverser les couleurs on soustrait à 255
         la valeur de chaque pixel
172       cmp  eax, 0 ;si la valeur obtenue est inférieur à 0 on la
        passe à 0 car on ne peut pas avoir une valeur d'intensité né
        gative.
173     jg     g_positif
174     xor eax,eax
175   g_positif:
176     mov edx, eax
177     shl edx, eax
178     add eax, edx
179     shl edx, 8
180     add eax, edx
181     mov [edi], eax
182   ;--------------
183
184
185
186     add esi, 4
187     add edi, 4
188     dec ecx
189     test ecx, 0000ffffh  ;on test si on est arrivé au bout de la
         ligne
190     jne     boucle2  ;on revient à l'itération sur les colones
191
192
193     add esi, 8
194     add edi, 8
195     sub ecx, 00010000h ; on passe à la ligne suivante
196
197     jnz     boucle3  ;on revient à l'itération sur les lignes
198
199 fin:
200
201     pop      edi
202     pop      esi
203     pop      ebx
204
205     pop      ebp
206         ret                           ; Retour a la fonction
        MainWndProc
207
208 process_image_asm   ENDP
209 END
```

Et voici les résultat observés avec l'assembleur pour 100 itérations :

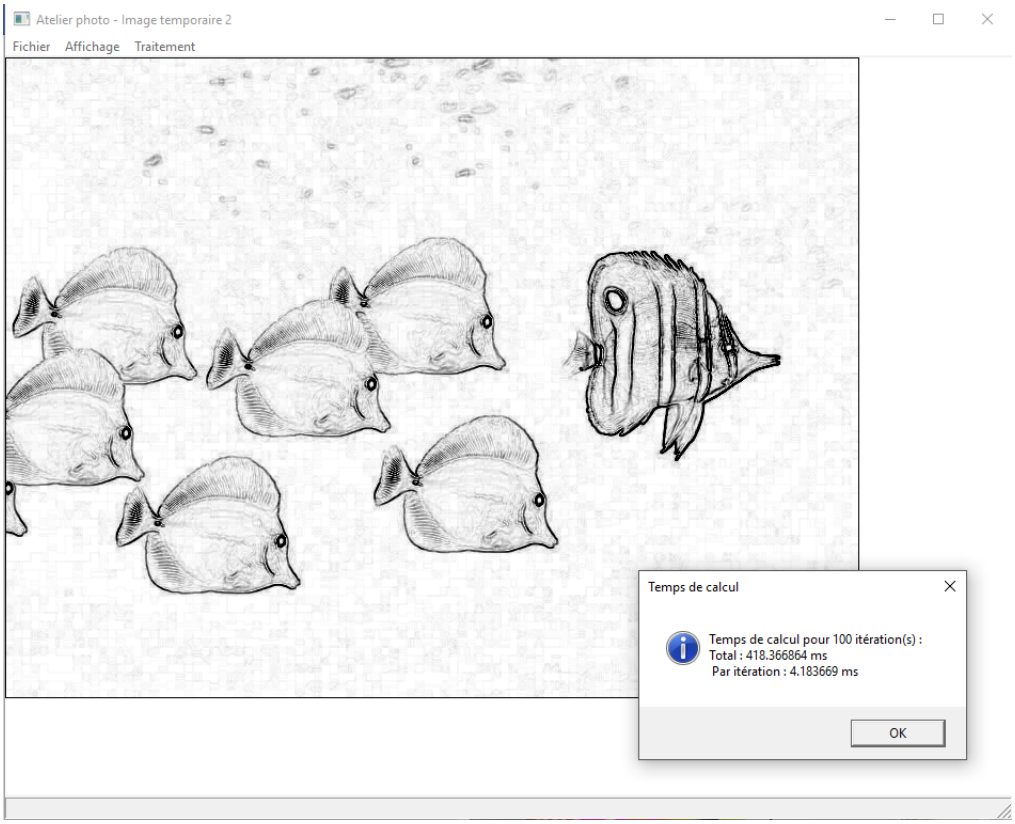

Figure 3: Résultat du programme en assembleur de détection de contours

Avec le code en C, pour 100 itérations on obtient 12,24ms par itérations, on voit donc que le code en assembleur est nettement plus rapide que le code en C.