TP05 - Traitement d'image - première partie

Geoffrey PERRIN
Océane DUBOIS

0.1 squelette du sous-programme process_image_asm

Sachant que le compilateur empile les arguments dans l'ordre inverse de celui dans lequel ils sont passés à la fonction, ecx contient d'abord la largeur de l'image puis il est multiplié par la hauteur de l'image, ainsi on obtient le nombre de pixels de l'image. Dans esi, on met le pointeur vers l'image source et edi contient un pointeur vers l'image tempon 1.

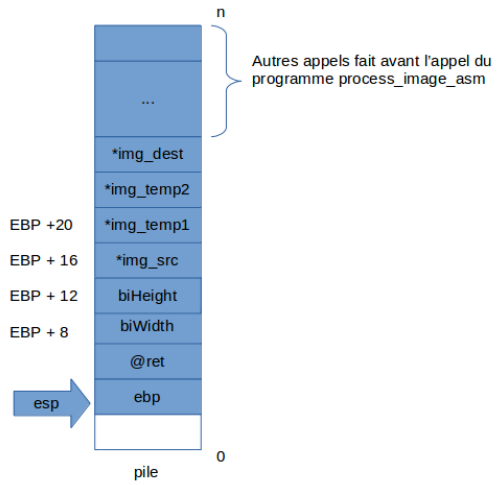

Figure 1: Etat de la pile juste après l'appel de process_image_asm

0.2 Conversion en niveaux de gris

Sachant que les pixels sont indexés de 1 à n, et que ecx contient l'index d'un pixel donné et esi contient un pointeur vers l'image source,alors l'adresse du pixel d'index ecx de l'image source sera: $esi + ecx * 4 - 1$.
L'adresse du pixel d'index ecx de l'image destination sera: $edi + ecx * 4 - 1$.
En effet on ajoute au pointeur de l'image de départ, l'index du pixel multiplié par 4 qui correpond à la taille d'un pixel (un pixel occupe 4 octets en mémoire) et on retire 1 car les pixels sont indexés de 1 à n et non de 0 à n-1.

Pour parcourir tous les pixels de l'image source, on utilise une structure de type :

```
1 do{
2   traitement de l'image;
3   ecx - -;
```

```
}while(ecx >=0);
```

En assembleur cela correspond à :

```
boucle: dec       ecx
        mov     ebx,[esi+ecx*4]
        ;Traitement
        cmp     ecx, 0
        ja      boucle
```

Il est plus facile de partir du dernier pixel de l'image car ecx contient déjà le nombre de pixel total de la photo.

Pour convertir une image en nuance de gris on utilise la formule suivante : $I = R * 0,299 + V * 0,587 + B * 0,114$

On converti pour la suite du TP les coefficients de ce calcul en hexadécimal. On obtient donc, en virgule fixe sur 16 bits avec un décalage à la virgule de +8 :

- Cr = 0,299 = 00,4Ch
- Cv = 0,587 = 00,96h
- Cb = 0,114 = 00,1Dh

Sachant que la partie entiere des coefficient et que la partie décimal des valeur de RVB sont toujours nulle on peut simplifier le calcul:

On décale de 8 bits a gauche les coefficients. (On peut se permmetre de perdre l'information de la partie entiere car elle est toujours nulle) Alors Cr = 4Ch, Cv = 96h, Cb=1Dh. Ainsi les coefficients sont réduit à des entiers de 8 bits.

De plus la partie décimal des valeurs de R,V,B étant nulle on peut réduire ces valeurs a des entier de 8 bit en ométant la partie décimal.

On a donc transformé la multiplication entre deux réels de 16 bits en une multiplication entre deux entiers de 8 bits.

Mais en faisant ce décalage de 8 bits vers la gauche on a multiplié le resultat final par 256. Il suffit donc de redécaler de 8 bits vers la droite le résultat final pour le diviser par 256 et donc trouver le résultat attendu.

Lors de la conversion des coefficient en hexadecimal sur 16 bits nous ne faisons qu'approcher la valeur des coefficient. Il y a une perte d'informaton lié au nombre limité de bit.

Sachant que la valeur maximal que peut prendre les coefficients R, V et B est 255 et que la somme des 3 coefficient du calcul de I est inférieur ou égale 1, on aura donc comme valeur max $1 * 255$, qui vaut donc 255. Sur 8

bits on peut coder au plus 255 valeurs, il n'y aura donc pas de débordements. Nous avons décidé d'arrondir au coefficient inferieur le plus proche pour que la somme des coeffcients reste infereiur ou égale a 1.

On souhaite éviter les débordements pour éviter de perdre de l'information sur l'image.

```
1  PUBLIC        process_image_asm
2  process_image_asm    PROC NEAR        ; Point d'entrée du sous
       programme
3
4          push     ebp
5          mov      ebp, esp
6
7          push     ecx
8          push   eax
9          push     edx
10
11
12         ;récupération des arguments dans les différents
       registres
13         mov      ecx, [ebp + 8]
14         imul     ecx, [ebp + 12]
15
16         mov      esi, [ebp + 16]
17         mov      edi, [ebp + 20]
18
19         xor     edx,edx
20         ;début de la boucle
21 boucle: dec       ecx
22         ;on récupére le pixel de l'image source à traiter
23         mov     ebx,[esi+ecx*4]
24
25         ;--------------------
26         ;calcul de B * 0,114
27         ;--------------------
28
29         mov       eax,ebx
30         ;on récupere la composante bleu dans eax
31         and       eax,000000FFh
32         ;on effectue le calcul
33         imul    eax,1Dh ;B * 0,114
34         ;on stock la somme (I) dans edx
35         mov       edx,eax
36
37         ;--------------------
38         ;calcul de V *,0,587
39         ;--------------------
40
```

```
41          ;on récupere la composante verte dans eax
42          mov       eax,ebx
43          and       eax,0000FF00h
44          shr     eax,8
45          ;on effectue le calcul
46          imul    eax,96h ;V *,0,587
47          add       edx,eax ;I=I+V *,0,587
48
49          ;--------------------
50          ;calcul de R*0,299
51          ;--------------------
52          ;on récupere la composante rouge dans eax
53          mov       eax,ebx
54          and       eax,00FF0000h
55          shr     eax,16
56          ;on effectue le calcul
57          imul    eax,4Ch
58          add       edx,eax ;I=I+R*0,299
59
60          ;on divise par 256
61          shr     edx,8
62          ;on stock I dans la composante bleu de l'image
        destination
63          mov     [edi+ecx*4],edx
64
65
66          cmp     ecx, 0
67          ja      boucle
68
69  fin:
70          pop      edx
71          pop      eax
72          pop      ecx
73          pop      ebp
74          ret                             ; Retour à la fonction
        MainWndProc
75
76  process_image_asm    ENDP
77  END
```

Le code en assembleur est plus performant que le code en C. L'éxecution du traitement dure en moyenne 0,7ms en assembleur contre 1,26ms en C.

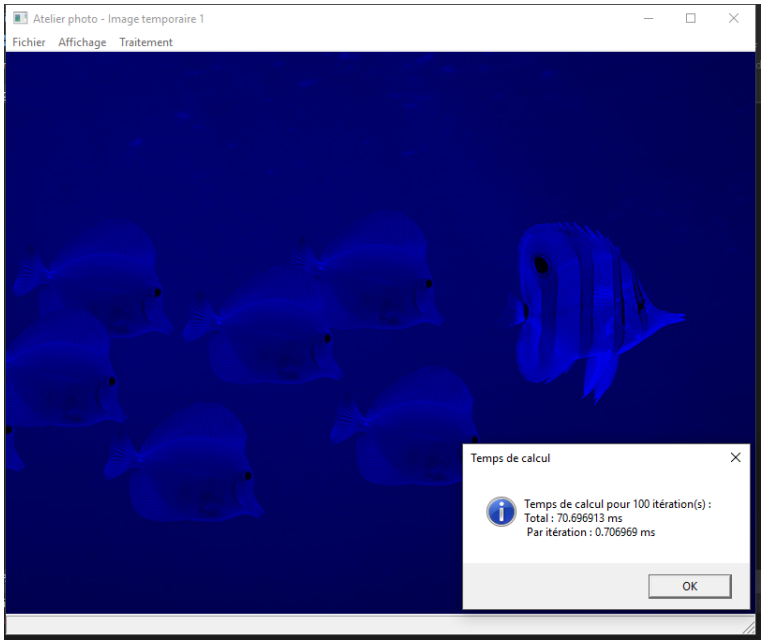

Figure 2: Execution de l'algorithme de traitement en Assembleur

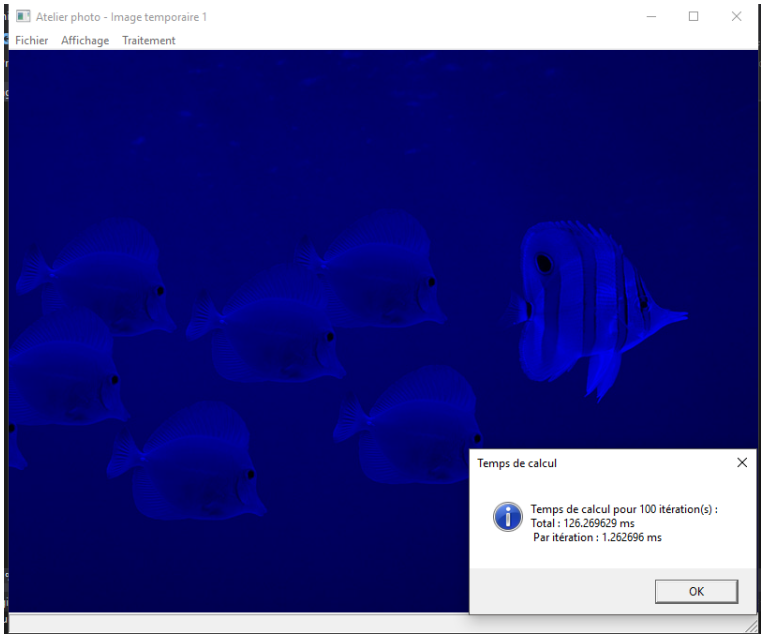

Figure 3: Execution de l'algorithme de traitement en C