Langage C et Traitement d'images

Lionel Lacassagne – version Mai 2005

Introduction

Ces quelques pages présentes une séries de réflexions, et d'astuces de codages des données multi-dimensionnelles en langage C. L'origine provient du livre "Numerical Recipes in C" dont la principale "recette" est l'adressage nD (1D, 2D et 3D) avec des intervalles d'adressages ne commençant pas à zéro. Les indices peuvent être négatifs ou très grands sans que cela occasionne des pertes de mémoire.
Ce document reprend cette recette et l'applique au cas des images monochromes ou couleurs. L'approche décrite ici, permet de généraliser à bien d'autre type de données.
Le but recherché ici, est le codage le plus clair et le plus efficace possible d'algorithmes de traitement d'image. Cela signifie qu'un soin tout particulier doit être apporté aux fonctions d'accès mémoire et à leur portabilité d'un OS à un autre.

Les types de données

Il est nécessaire de les redéfinir, car d'une machine à l'autre ils peuvent changer :

```
typedef unsigned char  byte;
typedef unsigned char  int8;
typedef          short int16;
typedef          int   int32;
```

Attention le type int64 existe sous linux, il ne faut pas le redéfinir.
En fonctions des compilateurs pour processeurs 32 bits des implantation int64 ont été réalisées, mais de manière non standard.
Avec Microsoft VisualC :

```
typedef          __int64  int64;
```

Avec Metrowerk CodeWarrior :

```
typedef          long long  int64;
```

Pour le traitement d'images, il est aussi utile de définir des types pour les images couleurs. .nous définissons aussi des types pour les images couleurs, grâce à une structure :

```
typedef struct {byte  r; byte g; byte b;} rgb8;
typedef struct {byte  r; byte g; byte b; byte x;} rgbx8;
```

pour définir la transparence (alpha-channel / alpha-blending)

De même pour les images codées avec 16 bits par composantes (format disponible via les types d'images PNG et TIFF ainsi que DICOM pour l'imagerie médicale)

```
typedef struct {int16  r; int16 g; int16 b;} rgb16;
typedef struct {int16  r; int16 g; int16 b; int16 x;} rgbx16;
```

Remarque : il est possible, si nécessaire, de faire la distinction entre type signé et type non signé, plutôt que de laisser cela au compilateur, par exemple :

```
typedef          char  uint8;
typedef unsigned char  uint8;
typedef   signed char  sint8;
```

Afin de respecter cette cohérence de notation, il est donc aussi nécessaire de redéfinir les types flottants, ce qui permettrait d'inclure les format SIMD 128 bits.

```
typedef       float   float32;
typedef       double  float64;
typedef  long double  float80;
```

Le fichier `def.h` décrit toutes ces définitions, ainsi que des définitions sur les champs de bit (*bitfiled*).

Numerical Recipes

NRC est une librairie scientifique permettant de manipuler simplement et efficacement les objets mathématiques comme les vecteurs, matrices et les tenseurs/cubes.
Les fonctions développées par Numerical Recipes et enrichies par le PARC/LISIF (UPMC) puis le groupe AXIS/IEF (UPSUD) ont plusieurs avantages :

- les indices des tableaux ne commencent pas nécessairement à 0 et peuvent être négatifs
- il est possible de passer (dynamiquement) des objets multidimensionnels (2D et 3D) à une fonction sans qu'il soit nécessaire de connaître une dimension, comme c'est le cas, classiquement en C,
- il est possible de *mapper/wrapper* une zone mémoire 1D allouée par une fonction extérieure à NRC en une zone 2D, et ce, même avec un *padding* (lignes d'une image non contigues en mémoire, c'est le cas lors d'alignement 32 bits pour les images couleurs – souvent codées sur 24 bits, et maintenant des alignements de 64 bits et 128 bits pour des raisons de performances ainsi que des contraintes impératives sur les nombres 128 bits).

NRC utilise des acronymes pour ses indices :
`n` : number
`r : row` (1ere dimension)
`c : column` (2eme dimension)
`d : depth` (3eme dimension)
`l : low`
`h : high`

Ainsi, nous avons :
`[nrl..nrh]` pour les objets 1D `vector`
`[nrl..nrh] [ncl..nch]` pour les objets 2D `matrix`
`[nrl..nrh] [ncl..nch][ndl..ndh]` pour les objets 3D `tensor`
`[ndl..ndh] [nrl..nrh] [ncl..nch]` pour les objets 3D `cube`

Les notations en TI sont différentes :
`i` indice de ligne (2ème dimension)
`j` indice de colonne (1ère dimension)
`k` indice de page (3ème dimension)

Ainsi une ROI (*Region Of Interest*) classiquement définie par [i0..i1]x[j0..j1] sera décrite en C de la même façon que sa définition formelle et son utilisation dans un algorithme.

Les objets 1D

prototypes
```
byte* bvector     (long nl, long nh);
byte* bvector0    (long nl, long nh); // init à zero
void  free_bvector(byte* v, long nl, long nh);
```

exemple 1 : initialisation d'un vecteur
```
byte *v;
v = bvector(i0, i1);
```

```
for(i=i0; i<=i1; i++) {
  v[i] = 2*i;
}
```

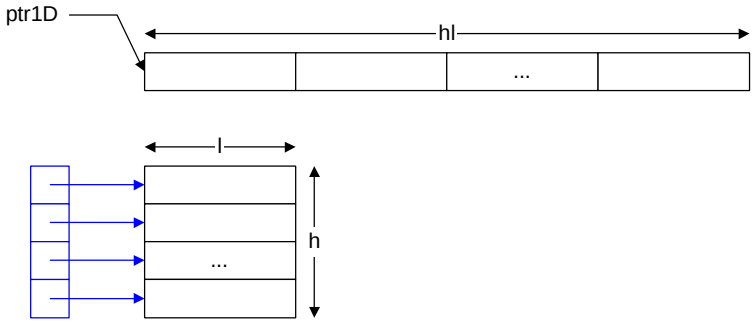
Les objets 2D

prototypes

```
byte** bmatrix     (         long nrl, long nrh, long ncl, long nch);
void   free_bmatrix(byte**m, long nrl, long nrh, long ncl, long nch);
```

Fonctionnement

La mémoire n'étant qu'1D, le but des allocations de NRC est de simuler la seconde dimension grâce à un tableau de pointeur pointant sur chaque début de ligne.



exemple 1a : initialisation

```
byte **m;
m = bmatrix(i0, i1, j0, j1);
for(i=i0; i<=i1; i++) {
  for(j=j0; j<=j1; j++) {
    m[i][j] = (i+j)%256;
  }
}
```

exemple 1b : initialisation

```
rgb8 **m;
m = rgb8matrix(i0, i1, j0, j1);
for(i=i0; i<=i1; i++) {
  for(j=j0; j<=j1; j++) {
    m[i][j].r = (i+j+0)%256;
    m[i][j].g = (i+j+1)%256;
    m[i][j].b = (i+j+2)%256;
  }
}
```

exemple 2a : addition N&B

```
add_bmatrix(byte **X, long i0, long i1, long j0, long j1, byte **Y, byte **Z)
{
int i, j;
for(i=i0; i<=i1; i++) {
  for(j=j0; j<=j1; j++) {
    Z[i][j] = X[i][j] + Y[i][j];
  }
}
```

exemple 2b : addition couleur

```
add_rgb8matrix(rgb8 **X, long i0, long i1, long j0, long j1, rgb8 **Y, rgb8 **Z)
{
int i, j;
```

```
for(i=i0; i<=i1; i++) {
  for(j=j0; j<=j1; j++) {
    Z[i][j].r = X[i][j].r + Y[i][j].r;
    Z[i][j].g = X[i][j].g + Y[i][j].g;
    Z[i][j].b = X[i][j].b + Y[i][j].b;
  }
}
```

Optimisation logicielle

Le But premier de ces formulations est de simplifier les notations. Ainsi l'accès 2D NRC `T[i][j]` s'écrit plus classiquement, lorsqu'on ne considère que la mémoire en 1D, `k=i*N+j, T[k]`.
Non seulement la notation classique est plus complexe, et donc susceptible de provoquer des bugs (lors de recopie « sauvage » de précédente ligne de code), diminue les possibilité de debug. (essayer donc de debugger un noyau de convolution 5x5 couleur, avec seulement des accès 1D), mais en plus elle est plus lente : 1 multiplication et 1 addition pour calculer l'adresse mémoire contre 2 additions pour la version NRC. Moins lisible et moins rapide, la version classique 1D ne présente que des problèmes.

Le codage NRC possède un second avantage : il permet d'optimiser les accès mémoire.
Une fois qu'une version correcte et validée d'un algorithme a été écrite, et seulement à ce moment là, il est très simple d'obtenir une version plus rapide. Il suffit alors de remplacer les formules 2D par des formules 1D, en utilisant des pointeurs de début de ligne, par exemple

exemple 2c : initialisation rapide

```
init8(int8 **X, long i0, long i1, long j0, long j1)
int i, j
int8 *Xi;
for(i=i0; i<=i1; i++) {
  Xi = X[i]; // pointeurs de ligne
  for(j=j0; j<=j1; j++) {
    Xi[j] = (i+j)&0xff;
```

Remarque : sur certaines machines, les registres internes du processeurs sont en nombre réduit (8 sur Pentium et ADM Athlon contre 128 sur PowerPC, Sun, HP, IBM). Utiliser un grand nombres de registres pour pointer sur des débuts de lignes peut alors avoir l'effet inverse de celui recherché : le code peut alors être plus lent. Il est donc véritablement nécessaire de toujours commencer par une version 2D.

exemple 2d : addition N&B optimisée

```
add_bmatrix(byte **X, long i0, long i1, long j0, long j1, byte **Y, byte **Z)
{
int i, j;
byte x, y, z;
byte *Xi, *Yi, *Zi;
for(i=i0; i<=i1; i++) {
  Xi = X[i]; Yi = Y[i]; Zi = Z[i]; // pointeurs de ligne
  for(j=j0; j<=j1; j++) {
    x = Xi[j]; y = Yi[j].
    z = x + y;
    Zi[j] = z;
  }
}
```

exemple 2e : addition couleur optimisée

```
add_rgb8matrix(rgb8 **X, long i0, long i1, long j0, long j1, rgb8 **Y, rgb8 **Z)
{
int i, j;
rgb8 x, y, z;
rgb8 *Xi, *Yi , *Zi;
```

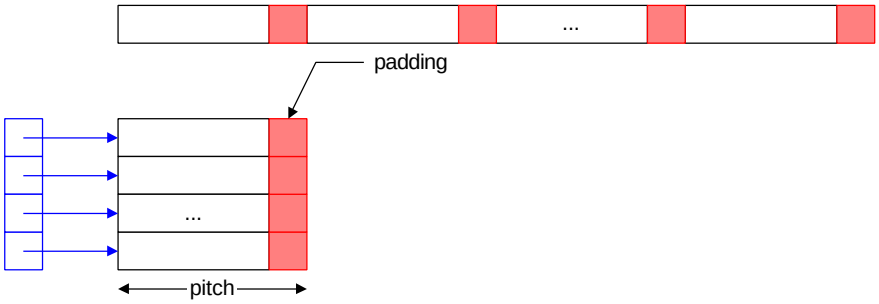
```
for(i=i0; i<=i1; i++) {
  Xi = X[i]; Yi = Y[i]; Zi = Z[i]; // pointeurs de ligne
  for(j=j0; j<=j1; j++) {
    x = Xi[j]; y = Yi[j].
    z.r = x.r + y.r; // calcul pour chaque composante
    z.g = x.g + y.g;
    z.b = x.b + y.b;
    Zi[j] = z;
  }
}
```

Ces versions optimisées ne sont a écrire qu'une fois que les versions "full NRC" fonctionnent correctement, ont été débugguées et documentées (quelques ligne dans le fichier header).

les mapping/wrapping 2D

Parfois l'utilisateur n'est pas maître de l'allocation mémoire, il faut donc pouvoir lire des zones mémoire allouées par un système autre que NRC : à partir d'une adresse pointant une zone mémoire (continue), un tableau de pointeur (vertical et en bleu sur la figure) est construit en prenant en compte les dimensions de la zone mémoire,

Pour des raisons d'optimisation des accès mémoire, les débuts de ligne nécessitent parfois d'être alignés sur des multiples de 4 octets. Cela ne pose donc pas de problème aux images 32 bits, mais les images 8 bits (noir et blanc) 16 bits (noir et blanc médical) et 24 bits (RGB, 8 bits par composante) peuvent se voir ajouter un *padding* en fin de ligne. Dans ce cas le dernier pixel d'une ligne n'est plus connexe au premier pixel de la ligne suivante.



Pour réaliser cela, et aussi afin de prendre en compte les mouvements de la mémoire (*garbage collecting*), le *mapping* se fait en deux temps : dans un premier temps, on ne construit que le tableau de pointeurs, et dans le second, on fait pointer la première case vers le début de la zone mémoire et les cases suivantes, contiennent l'adresse de la case précédente plus le *pitch* en octets. Attention, à cause de l'arithmétique des pointeurs, ces calculs doivent être fait avec un type 8 bits. Le *pitch* étant la "distance" en octets entre deux début de ligne.

Exemple 1 : wrapping N&B pour CVB (Common Vision Blox – Stemmer Imaging Inc)

```
byte** Wrapper_CVB_BYTE(IMG Img)
{
  long  Plane, lXInc, lYInc, lXIncrement, lYIncrement, pitch;
  long  lImageWidth, lImageHeight, lDimension;
  long  lXStatus, lYStatus;
  long  lSize, lRefCount;

  BOOL  b_Linear, b_XSwap, b_YSwap;

  void *lpBaseAddress; // adresse CVB
  byte  *data_1D;
  byte **ptrNRC;   // pointeur 2D NRC
```

```
  Plane = 0;
  lImageWidth  = ImageWidth       (Img);
  lImageHeight = ImageHeight      (Img);
  lDimension   = ImageDimension   (Img); // 1:B&W, 3:RGB
  lSize        = ImageToMemorySize(Img);

  lRefCount = RefCount(Img);
  b_XSwap   = FALSE;
  b_YSwap   = FALSE;
  b_Linear = GetLinearAccess (Img, Plane, &lpBaseAddress, &lXIncrement, &lYIncrement);
  data_1D = (byte*) lpBaseAddress; // adresse de debut
  data_1D = data_1D;
  pitch = lYIncrement;

   // vecteur de pointeurs
  ptrNRC = bmatrix_map(        0, lImageHeight-1, 0, lImageWidth-1);
  // pitch en octets
  bmatrix_map_1D_pitch(ptrNRC, 0, lImageHeight-1, 0, lImageWidth-1, data_1D, pitch);
  Test_Wrapper(ptrNRC, 0, lImageHeight-1, 0, lImageWidth-1);
  return ptrNRC;
}
```

Exemple 2 : wrapping couleur pour CVB

```
rgb8** Wrapper_CVB_RGB8(IMG Img)
{
  long  Plane, lXInc, lYInc, lXIncrement, lYIncrement, pitch;
  long  lImageWidth, lImageHeight, lDimension;
  long  lXStatus, lYStatus;
  long  lSize, lRefCount;

  BOOL  b_Linear, b_XSwap, b_YSwap;

  void *lpBaseAddress; // adresse CVB
  byte  *data_1D;
  rgb8 **ptrNRC;   // pointeur 2D NRC

  Plane = 0;
  lImageWidth  = ImageWidth       (Img);
  lImageHeight = ImageHeight      (Img);
  lDimension   = ImageDimension   (Img); // 1:B&W, 3:RGB
  lSize        = ImageToMemorySize(Img);

  lRefCount = RefCount(Img);
  b_XSwap   = FALSE;
  b_YSwap   = FALSE;
  b_Linear = GetLinearAccess (Img, Plane, &lpBaseAddress, &lXIncrement, &lYIncrement);
  data_1D = (byte*) lpBaseAddress; // adresse de debut
  data_1D = data_1D - 2; // a cause de l'offset de CVB (bug)
  pitch = lYIncrement;

   // vecteur de pointeurs
  ptrNRC = rgb8matrix_map(        0, lImageHeight-1, 0, lImageWidth-1);
  // pitch en octets
  rgb8matrix_map_1D_pitch(ptrNRC, 0, lImageHeight-1, 0, lImageWidth-1, data_1D, pitch);
  Test_Wrapper(ptrNRC, 0, lImageHeight-1, 0, lImageWidth-1);
  return ptrNRC;
}
```

Les lignes importantes ont été mises en gras. Comme on peut le voir, les versions monochrome et couleur sont très semblables, à part le décalage de 2 octets dans la version couleur, à cause d'un pseudo bug dans CVB (l'adresse retournée par CVB est celle du coin en bas à gauche de l'image et de la composante r –

premier plan image, alors que les composantes couleurs sont stockées dans l'ordre b-g-r, ce qui revient à donner la troisième composante couleur du premier pixel au lieu de donner la première composante)

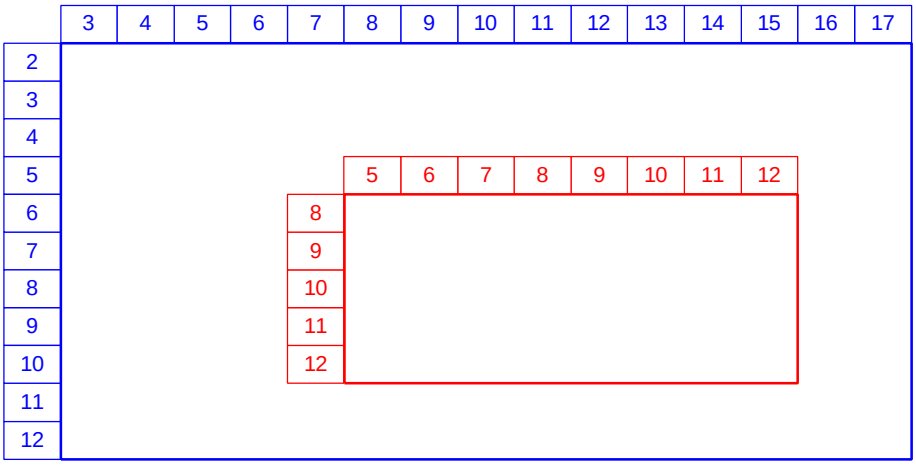
les mapping/wrapping de ROI

Le but ici est, à partir d'une matrice définie sur un certain intervalle, de définir un *wrapper* de ROI (*Region Of Interest*) défini sur un autre intervalle. Dans le cas présenté ici, la matrice d'origine (en bleue sur la figure) a pour intervalle :
`[nrl..nrh][ncl..nch] = [3..17][2..12]`
la ROI a pour intervalle (en rouge sur la figure)
`[nrl2..nrh2][ncl2..nch2] = [8..12][5..12]`
et on désire que le bord supérieur gauche de la ROI soit au point (dans les coordonnées de la matrice) `(nr0,nc0) = (6,8).`

Pour réaliser cela, il suffit d'appeler l'une des fonctions suivantes (`byte` ou `rgb8`). La matrice source (qui peut être un *wrapper* doit être allouée au préalable. La désallocation se fait en utilisant les fonctions de désallocation des *wrappers*.

```
byte** bmatrix_map_pitch_ROI
(byte **X, long nrl,  long nrh,  long ncl,  long nch, long pitch,
           long nrl2, long nrh2, long ncl2, long nch2,
           long nr0,  long nc0);

rgb8** rgb8matrix_map_pitch_ROI
(rgb8 **X, long nrl,  long nrh,  long ncl,  long nch, long pitch,
           long nrl2, long nrh2, long ncl2, long nch2,
           long nr0,  long nc0);
```



Les objets 3D

Les objets 3D définis dans NRC sont les `tenseurs`. Afin d'avoir une écriture plus cohérente, nous avons permuté les premier et troisième indices pour créer le type `cube`.
Un `cube` est un véritable objet 3D (`C[k][i][j]`) mais peut être vu comme une pile d'images 2D.
Ainsi les fonctions de manipulation de `cube` se basent sur les fonctions de manipulation de `matrix`. L'addition de cube s'écrit :

```
add_bcube(byte ***c1, long ndl, long ndh, long nrl, long nrh, long ncl, long nch, byte ***c2, ***c3)
// c3 = c1+c2
```

```
for(k=k0; k<=k1; k++) {
  add_bmatrix(c1[k], nrl, nrh, ncl, nch, c2[k], c3[k]);
}
```

Les opérateurs arithmétiques et logiques

Les principaux opérateurs arithmétiques (+,-,*,/) et logiques (OR, NOT, AND, XOR) ont été codés. ainsi que des fonctions de conversions, d'initialisation et de tri.
Ces fonctions se trouvent dans le fichier nrarith.c. Exemple :

```
add_bmatrix(byte **m1, long nrl,long nrh,long ncl, long nch, byte **m2, byte **m3)
```

Les Entrées / Sorties

Les fonctions d'ES se trouvent dans le fichier nrio.c. Elles permettent d'afficher à l'écran, dans une fenêtre en mode texte, des vecteurs, des matrices et des cubes pour les types les plus courants. Ces fonctions permettent aussi d'écrire en mode texte et en mode binaire ces mêmes objets.

Architecture logicielle

Un code (i.e. un algorithme codé) au format NRC, "rien qu'en NRC, tout en NRC" sera totalement portable et pourra être réutilisé dans n'importe quelle application. Cela deviendra un composant, une brique réutilisable.

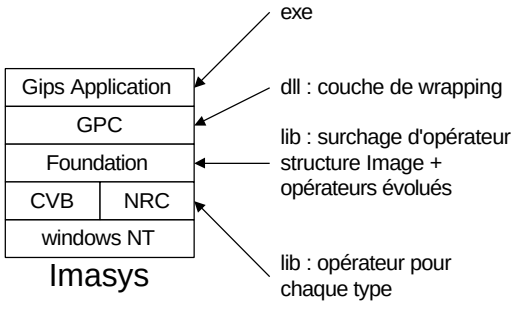
Séparation C/C++ et Algorithme/Interfaçage

Une autre notion très importante est la distinction entre les algorithmes et leur interfaçage avec l'existant (logiciel, librairie, …). Afin d'être efficace, aussi bien en temps de codage (en vue du debug) qu'en temps d'exécution, tout opérateur de TI doit être écrit en C, dans un fichier .c, et les fonctions d'interfaçage propres à la librairie existante (GipsVision/Pixy, CVB, Matrox, Cognex, Imaging, Lecky) dans un fichier C++ si la librairie manipule des objets, ou dans un fichier C, s'il n'y a pas d'objet.
Dans tous les cas, il faut absolument séparer les algorithmes (en NRC) du reste.

Voici maintenant quelques exemples d'architecture logicielle

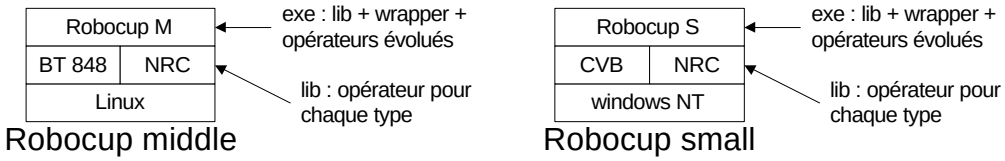
Imasys

En dessous de l'application Gips, se trouve la dll GPC qui se charge de *wrapper* le format d'image IMG de CVB (lib C++ se chargeant du pire à coder : les drivers et l'affichage) avec les fonctions évoluées de TI se trouvant dans la librairie Foundation. Cette librarie Foundation s'appuie quant à elle sur NRC pour la gestion mémoire. La gestion des bords d'une image (nécessaire lors d'opérations de type convolution est faite dans Foundation).



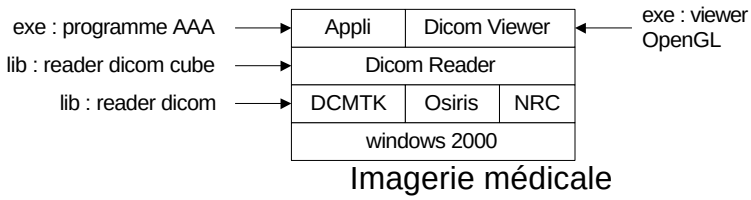
Robocup

L'OS utilisé n'est pas le même en small et en middle. Par contre beaucoup de fonctions de TI sont identiques dans les deux. Afin de ne pas les coder deux fois, elles sont en NRC et fonctionnent aussi bien sous linux que sous windows. De même pour les programmes "off line" tel que la calibration géométrique, la calibration couleur ou encore l'identification des "T-Shirts" adverses.



Imagerie médicale

Le problème ici est la lecture des fichiers médicaux au format DICOM. Plusieurs librairies existent, aucune ne fonctionnant correctement (dépend de l'origine de la machine qui a généré les images IRM, CT-Scan, angiographie et de la marque : GE, Siemens, Toshiba). Il faut donc faire une couche d'abstraction permettant de choisir le reader DICOM.



Architecture logicielle AXIS/APV (2003-2004)

Le problème est complexe. Il est nécessaire d'avoir des outils de base portables (Ansi-C, Ansi-C++ et OpenGL) des outils de traitement d'images bas niveaux (librairie IPL d'Intel, ou librairie Foundation de LL) de librairie très bas niveau (noyau NRC : nralloc.c, nrarith.c, nrio.c), mais aussi de librairie moyen niveau (OpenCV) ainsi que des interfaces graphiques portables comme GLUI, compatible OpenGL. La librairie GLUI est la plus simple des interfaces graphiques, il en existe d'autre plus evoluées comme GLOW, GTK ou FOX (www.fox-toolkit.com), ou encore Qt (www.trolltech.com/).

Le fichier nrio fourni les fonctions nécessaires pour faire des entrées-sorties écran/clavier/disque pour les différents formats précédemment présentés, ainsi que pour les formats de fichier PGM (monochrome 8 bits) et PPM (couleur 24 bits).

Le grand intérêt du format NRC dans ce contexte est qu'il permet de wrapper les formats d'images utilisés dans OpenCV (format iplImage d'Intel, format réutilisés dans IPP, l évolution d'IPL après la version 2.5).
NRC rend ces format plus lisible, grâce à l'adressage 2D. Cela est particulièrement vrai en couleur.
De plus, et comme les codes sources d'OpenCV sont disponibles, il est possible d'introduire dans la structure même d'iplImage, un double pointeur sur des données, rendant inutile l'utilisation explicite des wrappers par l'utilisateur. Des algorithmes développés en suivant les règles de codages NRC, pourront donc être directement intégrés dans OpenCV, sans qu'une ré-écriture de code, toujours source de bug, ne soit nécessaire.

Les alternatives

Il y a au moins trois alternatives a l'indexation NR :

- l’utilisation d’un #define
- l’appel à une routine d’acces au pixel
- l’utilisation de pointeurs de ligne

une routine typique serait
```
byte getPixel(byte *P, int i, int j, int width)
{  return P[(i*width+j)] ;}
```
Le define serait identique :
```
#define GETPIXEL(P,i,j,width) P[(i)*(width)+(j)]
```

Le #define est “plus rapide” que la routine (elle doit être inlinable) mais moins facilement debuggable. Il n'est pas plus rapide qu'une fonction inline, mais bien plus source de bugs.
L’utilisation de pointeurs de ligne, solution vers laquelle on tend lorsqu’on optimise un code d’accès 2D est la solution la plus efficace. C'était la solution systématiquement retenus il y a une dizaine d'année, car l'écriture sous forme de pointeur *(p+i) était plus rapide (le code généré par le compilateur) que lécriture sous forme de table p[i].
Mais ce style d'écriture ne permet pas de répondre à un des critères de NRC : la souplesse de codage. Un exemple de code serait :
```
byte *pi0, *pi1, *pi2 ;
p0 = &p[(i-1)*width];
p1 = &p[(i)*width];
p2 = &p[(i+1)*width];
```

La lisibilité de telles solution s’effondre définitivement lorsqu’on traite des images couleurs.
Pour des raisons d’optimisation d’accès mémoire, les images couleurs peuvent avoir des adresses de début de ligne alignées sur des multiples de 4 octets. Cela est transparent pour les images 32 bits, mais pas pour les images 24 bits. Il faut alors en plus prendre en compte le padding en fin de ligne et gérer un pitch (la longueur véritable de la ligne) : le dernier pixel d’une ligne n’est plus suivi par le premier pixel de la ligne suivante.
Ainsi pour appliquer un filre passe bas classique, nous aurons (pour la composante g, d’un rgb8)
```
g =   I[i-1][j-1].g + 2 * I[i-1][j].g +   I[i-1][j+1].g;
g+= 2*I[I  ][j-1].g + 4 * I[I  ][j].g + 2*I[I  ][j+1].g;
g+=   I[i+1][j-1].g + 2 * I[i+1][j].g +   I[i+1][j+1].g;
```

En utilisant une routine ou un #define le code C, après le passage du préprocesseur ressemblera à:
```
g =   p[3*(i-1)*n+3*(j-1)+1] + 2*p[3*(i-1)*n+3*j+1] +   p[3*(i-1)*n+3*(j+1)+1];
g+= 2*p[3*i*n+3*(j-1)+1]     + 4*p[(3*i*n+3*j+1]    + 2*p[3*i*n+3*(j+1)+1];
g+=   p[3*(i+1)*n+3*(j-1)+1] + 2*p[3*(i+1)*n+3*j+1] +   p[3*(i+1)*n+3*(j+1)+1];
```

Avec les trois pointeurs de lignes définis précédemment on obtient un code un peu plus lisible:
```
g =   p0[3*(j-1)+1] + 2*p0[3*j+1] +   p0[3*(j+1)+1];
g+= 2*p1[3*(j-1)+1] + 4*p1[3*j+1] + 2*p1[3*(j+1)+1];
g+=   p2[3*(j-1)+1] + 2*p2[3*j+1] +   p2[3*(j+1)+1];
```

Il est a noter que certaines personnes n'utilisent ni l'un ni l'autre et codent directement l'adressage 1D de leur image tel quel. A cause des 3 composantes couleurs, il y a deux multiplications supplémentaires : une pour i et une pour j, ainsi qu'encore une addition pour gérer l'offset +1 par rapport à l'adresse de base du pixel.

Lorsqu'il devient nécessaire d'accélérer les accès aux pixel, la seule solution est alors de développer les différentes expressions.
```
g =   p[3*(n*i-n+j)-2] + 2*p[3*(n*i-n+j)+1] +   p[3*(n*i-n+j)+4];
g+= 2*p[3*(n*i+j)-2]   + 4*p[3*(n*i+j)+1]   + 2*p[3*(n*i+j)+4];
g+=   p[3*(n*i+n+j)-2] + 2*p[3*(n*I+n+j)+1] +   p[3*(n*I+n+j)+4];
```

A ce stade, toute erreur de frappe devient difficile à détecter. Cela est vrai pour tout développer, mais encore plus pour des étudiants qui ne sont pas forcement habitués à débugger et/ou à gérer des indexations 1D d'objet 2D.

OpenCV

il est très facile d'intégrer NRC à OpenCV, car le code source est disponible. Plutôt que de créer des *wrappers* à chaque appel de fonction, il est possible, en ajoutant un champs à la structure de le faire une seule fois, lors de la création de l'image.

```
typedef struct _IplImage {
// definition de la structure IplImage ici
void **data2D;
}
IplImage;
```

Il faut ensuite modifier le constructeur d'image Ipl, pour que ce pointeur pointe vers une zone allouée.
Cela est actuellement fait a l'extérieur, pour des raisons de debug, mais peut être fait à l'intérieur du constructeur.

```
void wrap(IplImage *image)
{
  // choix du traitement
  image->data2D = NULL;
    switch(image->nChannels) {
    case 1: wrap_nb (image); break;
    case 3: wrap_rgb(image); break;
  }
}
void wrap_nb(IplImage *image)
{
  int width  = image->width;
  int height = image->height;

  byte **data_2D, *data_1D;

  /*
   * recuperation du pointeur 1D fourni par OpenCV
   * creation d'un tableau de pointeur de debut de ligne
   * pointage de chacun des debuts de ligne, dans notre cas,
   * pas de padding en fin de ligne
   */

  data_1D = (byte*) image->imageData;
  data_2D = bmatrix_map(0, height-1, 0, width-1);
  bmatrix_map_1D_pitch(data_2D, 0, height-1, 0, width-1, data_1D, width);
  image->data2D = (void**) data_2D;
}
void wrap_rgb(IplImage *image)
{
  int width  = image->width;
  int height = image->height;

  rgb8 **data_2D, *data_1D;

   /*
   * ATTENTION : le pitch en octets vaut 3x la largeur d'une ligne
   */

  data_1D = (rgb8*) image->imageData;
```

```
  data_2D = rgb8matrix_map(0, height-1, 0, width-1);
  rgb8matrix_map_1D_pitch(data_2D, 0, height-1, 0, width-1, data_1D, 3*width);
  image->data2D = (void**) data_2D;
}
```

Le pointeur data2D est de type void car les images peuvent être de différents type.
L'exemple ci dessous implémente l'opérateur NOT, en OpenCV classique (adressage 1D) et en NRC adressage 2D. Noter que dans les 2 cas, on utilise une fonction "abstraite" pour simuler la surchage d'opérateurs C++.

version OpenCV

```
void not_nb(IplImage *imageS, IplImage *imageD)
{
  int i, j;
  int   width  = imageS->width;
  int   height = imageS->height;
  char *dataS  = imageS->imageData;
  char *dataD  = imageD->imageData;

  for(i=0; i<height; i++) {
    for(j=0; j<width; j++) {
      *dataD++ = 255 - *dataS++;
    }
  }
}
void not_rgb(IplImage *imageS, IplImage *imageD)
{
  int i, j;
  int   width  = imageS->width;
  int   height = imageS->height;
  char *dataS  = imageS->imageData;
  char *dataD  = imageD->imageData;

  for(i=0; i<height; i++) {
    for(j=0; j<width; j++) {
      *dataD++ = 255 - *dataS++;
      *dataD++ = 255 - *dataS++;
      *dataD++ = 255 - *dataS++;
    }
  }
}
void not(IplImage *imageS, IplImage *imageD)
{
  // choix du traitement

  switch(imageS->nChannels) {
    case 1: not_nb_NR (imageS, imageD); break;
    //case 3: not_rgb_NR(imageS, imageD); break;
      case 3: not_rgb(imageS, imageD); break;
    default : printf("erreur n"), exit(-1); break;
  }
}
```

Version Numerical Recipes

```
void not_nb_NR(IplImage *imageS, IplImage *imageD)
{
  int i, j;
  int   width  = imageS->width;
```

```
  int   height = imageS->height;

  byte **X = (byte**) imageS->data2D;
  byte **Y = (byte**) imageD->data2D;

  // le NOT est volontairement faux (div 2)
  // pour verifier que le traitement est fait sur l'ensemble de l'image

  for(i=0; i<height; i++) {
    for(j=0; j<width; j++) {
      X[i][j] = 255 - X[i][j]/2;
    }
  }
}
void not_rgb_NR(IplImage *imageS, IplImage *imageD)
{
  int i, j;
  int   width  = imageS->width;
  int   height = imageS->height;

  rgb8 **X = (rgb8**) imageS->data2D;
  rgb8 **Y = (rgb8**) imageD->data2D;

  // le NOT est volontairement faux (div 2)
  // pour verifier que le traitement est fait sur l'ensemble de l'image

  for(i=0; i<height; i++) {
    for(j=0; j<width; j++) {

      Y[i][j].r = 255 - X[i][j].r/2;
      Y[i][j].g = 255 - X[i][j].g/2;
      Y[i][j].b = 255 - X[i][j].b/2;

    }
  }
}
void not_NR(IplImage *imageS, IplImage *imageD)
{
  // choix du traitement

  switch(imageS->nChannels) {
    case 1: not_nb_NR (imageS, imageD); break;
    case 3: not_rgb_NR(imageS, imageD); break;
    default : printf("erreur n"), exit(-1); break;
  }
}
```

SWAR = SIMD Within A Register

Les allocateurs de vecteurs et de matrices peuvent être redéfinis pour manipuler les vecteurs 128 bits SSE+ des architectures Intel/AMD ou Altivec des PowerPC.
Pour rappel:

- SSE manipule des flottantes 32 bits
- SSE2 manipules des entiers 8,16 ou 32 bits, mais aussi des flottants 64 bits
- SSE3 ajoute des opérations intra registres (réductions)
- Altivec, manipule des flottants 32 bits et des entiers 8,16 32 bits.

La première étape consiste donc à redéfinir les types de base, pour qu'un algorithme puisse être portable d'une architecture à l'autre. En Altivec, les opérations ne sont pas typées, ce sont les données qui le sont, en SSE, c'est le contraire

en SSE, nous aurons:
```
typedef __m128i vuint8;
typedef __m128i vsint8;
typedef __m128i vuint16;
typedef __m128i vsint16;
typedef __m128i vuint32;
typedef __m128i vsint32;
typedef __m128  vfloat32;
```

en Altivec, nous aurons
```
typedef vector   signed char  vsint8;
typedef vector unsigned char  vuint8;
typedef vector   signed short vsint16;
typedef vector unsigned short vuint16;
typedef vector   signed int   vsint32;
typedef vector unsigned int   vuint32;
typedef vector          float vfloat32;
```

Wrapper SIMD

Un point essentiel pour la mise au point de code SIMD est d'améliorer les capacités de debug des compilateurs, en créant des wrappers SIMD. Les buts sont doubles:

- debug : wrappers scalaires de zones SIMD
- vectorisation : wrappers SIMD de zones scalaires

Comme il n'existe pas de type vectoriel pour les images couleurs (rgb8), le type vectoriel utilisé pour wrapper sera le vuint8 ou le vsint8. Il est nécessaire que les zones scalaires soient alignées sur des multiples de 128 bits (16 octets) pour réaliser des accès mémoire alignés. Il peut donc être nécessaire de faire du *padding* en fin de ligne pour que les début de lignes soient alignés. Sur PowerPC, il est recommandé, afin de diminuer les échecs d'accès au cache (*cache misses*) d'allouer des lignes dont la longueur est un multiple impair de 16 octets.

Le problème récurrent de la gestion des bords en SIMD se pose aussi en SIMD. Si une image est définie sur [0..h-1] x [0..w-1] ou plus généralement [i0..i1] x [j0..j1] et que l'opérateur est de type TCL (Traitement Combinatoire Local) avec des bords de 1 ou plus généralement des bords de b, l'image SIMD allouées sera :
[i0-b..i1+b] x [j0-1..j1+1] tant que les bords ne dépassent pas 16 pixels.
L'allocation scalaire sera
[i0-b..i1+b] x [j0-b.card ..j1+b.card]
où card représente le nombre d'élément d'un vecteur:

- 16 pour un vecteur 128 bits avec des composantes 8 bits (vsint8, vuint8)
- 8 pour un vecteur 128 bits avec des composantes 16 bits (vsint16, vuint16)
- 4 pour un vecteur 128 bits avec des composantes 32 bits (vsint32, vuint32, vfloat32)
- 2 pour un vecteur 128 bits avec des composantes 64 bits (vfloat64) (jamais utilisé *right now*)

Pour être cohérent les quantités passées aux constructeurs sont en nombre de vecteurs. Par rapport au types scalaires, il faut donc diviser ces quantités par 1 en vertical et par card en horizontal.

Exemple:
Cas particuliers des images commençant en (0,0): [0..h-1][0..w-1], bord b=2 (Gauss 5x5 par exemple)

Afin de simplifier l'écriture, la largeur des bords horizontaux est b et non pas ceil(b/16)

```
vuint8 **vX;
uint8  **sW;
uint8  **sW0;

int h, w, b, pitch;

vX  = vui8matrix(0-b, h-1-b, 0-b, w/16-1+b);
sW  = ui8matrix_map(0-b, h-1-b, 0-b*16, w-1+b*16);
sW0 = ui8matrix_map(0-b, h-1-b, 0-b*16, w-1+b*16);

pitch = w + 2*b; // en octets
sW  =ui8matrix_map_1D_pitch(sW, 0-b,h-1-b,0-b*16,w-1+b*16,&vX8[0-b][0-b-16],pitch);
sW0 =ui8matrix_map_1D_pitch(sW0,0,  h-1,  0,     w-1,     &vX8[0][0],       pitch);
```

Le pointeur sW wrappe l'image vectorielle vX, bords compris, alors que le pointeur sW0, ne wrappe que la zone "utile", sans les bords.

D'autre exemplse se trouvent dans le répertoire test.