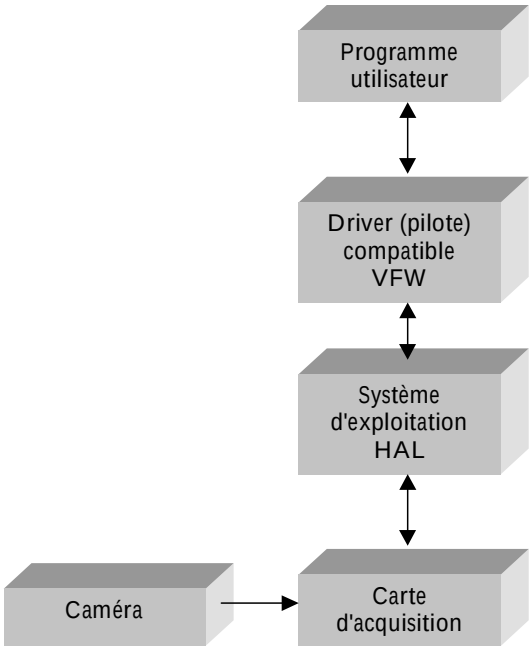
INTERFAÇAGE AVEC UNE CARTE D'ACQUISITION D'IMAGE

UTILISATION DE VIDEO FOR WINDOWS

Afin de pouvoir capturer des images fixes ou animées, un certain nombre de fonctions ont été écrites et regroupées au sein d'un ensemble appelé video for Windows. Ces fonctions sont du C classique, mais peuvent être utilisées dans les fonctions membres de classes C++, pour la plupart d'entre-elles.
Le principe d'utilisation suit la figure ci-dessous
La caméra envoie son signal par l'intermédiaire soit d'une carte connectée au bus PCI, soit par le port parallèle (imprimante), soit par le port USB (port série rapide).

Le signal numérique est utilisable par le programmeur par l'intermédiaire d'un pilote ou driver, programme qui doit respecter les entrées/sorties standards de Video For Windows. Ce driver, ce son côté, fait appel à des fonctions du système d'exploitation, et plus précisément à une couche de Windows NT dite HAL ou Hardware abstraction Layer, qui permet une meilleure stabilité du système en évitant que les programmes utilisateurs accèdent directement au matériel. Cette dernière couche, elle, peut accéder directement à l'acquisition des données sur le bus PCI.



Remarque préliminaire: pour compiler correctement le projet, il faut lui préciser le lien avec les fonctions de VFW, en ajoutant dans *projet->Settings->Link* : **vfw32.lib** dans la case *Object/library modules*. Il sera également nécessaire d'inclure **vfw.h** dans tous les fichiers faisant appel à des fonctions Video For Windows.

1 Acquisition des données

Pour ouvrir une fenêtre permettant de visualiser l'image reçue par la caméra, il est nécessaire de procéder à quelques initialisations.

1.1 Ouverture d'une fenêtre de capture

Dans la fonction OnInitDialog d'une boite de dialogue, il faut ajouter la l'appel suivant

hWndC = capCreateCaptureWindow ("Fenetre de Capture",WS_CHILD | WS_VISIBLE ,0, 0, 320, 240, m_hWnd, 0);

Les paramètres sont : le titre de la fenêtre, le style (la visualisation sera ici prisonnière de la boite de dialogue), la position initiale (0,0) et la taille initiale (320,240), le Handle de la fenêtre courante, et enfin un numéro identifiant la fenêtre, qui peut être 0.

La fonction retourne un Handle vers la fenêtre ainsi ouverte. Ce Handle doit être déclaré de la manière suivante :

HWND hWndC; / / en variable membre de la classe associée à la boite de dialogue.

1.2 Connection au driver

Il faut ensuite connecter la fenêtre de capture au driver existant, par l'intermédiaire de la fonction

capDriverConnect (hWndC, 0);

Le premier paramètre doit être le Handle de la fenêtre de capture précédemment ouverte. Le second paramètre représente le numéro du driver auquel on se connecte (nécessaire dans le cas de cartes d'acquisition à plusieurs entrées).

1.3 Acquisition et paramétrage des données

On peut alors lancer l'acquisition de manière à visualiser l'image en "direct" :

capPreviewRate(hwndC,10);
capPreview(hwndC, TRUE);

La première fonction spécifie la vitesse de visualisation (le deuxième paramètre est le temps entre deux trames, en millisecondes).

La deuxième fonction active la visualisation. Pour un traitement rapide, on peut très bien ne pas activer cette visualisation !

La configuration par défaut oblige l'utilisateur à cliquer pour activer la fenêtre au moins une fois, ce qui n'est pas très pratique à l'usage. Pour modifier cette configuration, et récupérer par la même occasion un certain nombre de données concernant l'image acquise, il est nécessaire d'appeler la fonction suivante :

```
capCaptureGetSetup(hWndC,&capParms,sizeof(CAPTUREPARMS)); / / Récup. des info du driver
capParms.fYield=TRUE; / / activer la fenetre sans etre obligé de cliquer!
capCaptureSetSetup(hWndC,&capParms,sizeof(CAPTUREPARMS)); / / forcer les paramètres du driver
```

capParms aura été déclaré en variable membre comme élément de structure CAPTUREPARMS:

CAPTUREPARMS capParms;

Une autre structure permet de récupérer un certain nombre d'information comme la taille de l'image. Il s'agit de la structure CAPSTATUS définie ainsi :

```
typedef struct {
 UINT uiImageWidth;
 UINT uiImageHeight;
 BOOL fLiveWindow;
 BOOL fOverlayWindow;
 BOOL fScale;
 POINT ptScroll;
 BOOL fUsingDefaultPalette;
 BOOL fAudioHardware;
 BOOL fCapFileExists;
 DWORD dwCurrentVideoFrame;
 DWORD dwCurrentVideoFramesDropped;
 DWORD dwCurrentWaveSamples;
 DWORD dwCurrentTimeElapsedMS;
 HPALETTE hPalCurrent;
 BOOL fCapturingNow;
 DWORD dwReturn;
 UINT wNumVideoAllocated;
```

```
UINT wNumAudioAllocated;
} CAPSTATUS;
```

Pour l'utiliser, on déclare un élément de cette structure :

CAPSTATUS capStatus;

Puis on appelle la fonction :

capGetStatus(hWndC, &capStatus, sizeof(capStatus));

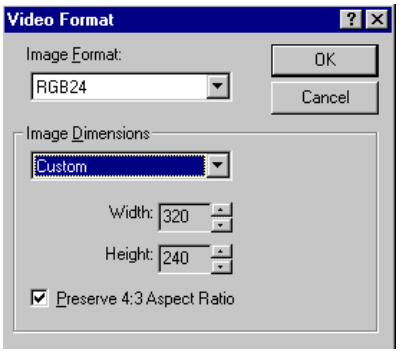
On peut alors récupérer la hauteur et la largeur de l'image en déclarant deux UINT en variables membres de la classe de la boite de dialogue.

Largeur=capStatus.uiImageWidth; // à faire après ouverture de la fenêtre de capture
Hauteur=capStatus.uiImageHeight;

Ces deux valeurs sont ici accessibles en lecture seulement. Pour les modifier, il faut impérativement passer par une boite de dialogue prévue (théoriquement !) dans tout driver standard VFW.
Pour accéder à cette boite de dialogue, il suffit d'appeler la fonction suivante :

capDlgVideoFormat(hWndC);

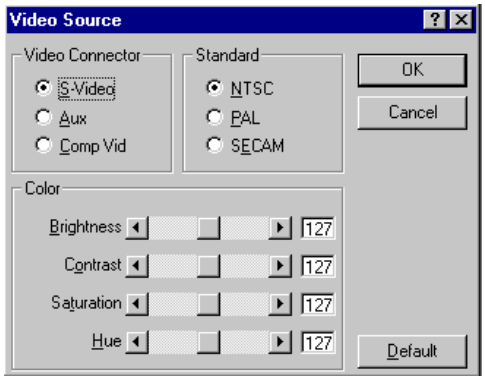
Dans le cas du driver de la carte Intel, la boite ci-contre s'affichera.
Elle permet de choisir le format des données (ici RGB 24 c'est à dire que chaque pixel est codé sur 24 bits, 8 bits par couleur) et la dimension de l'image, ici 320x240 pixels.
Cette boite peut différer légèrement suivant les cartes d'acquisition, mais devrait théoriquement fournir toujours les mêmes informations.



Il existe une seconde boite de dialogue dans le standard VFW. Elle permet de configurer lla source d'entrée de la carte d'acquisition, les couleurs, la contraste, la saturation, etc.
On l'obtient par l'appel :

capDlgVideoSource(hWndC);

Les changements sont immédiatement répercutés sur la fenêtre d'acquisition.
Il est à noter que ces deux fonctions admettent en paramètre le Hande de la fenêtre de capture. On ne peut donc les appeler qu'après initialisation de celle-ci (hWndC doit être non nul).



Enfin, dans la fonction DestroyWindow qu'il est nécessaire de surcharger, il faut libérer le driver :

CapDriverDisconnect(hWndC) ;

2 Traitement dynamique des données de l'image

Les fonctions utilisées précédemment permettent la visualisation "en direct" de l'image acquise par la caméra. Il est possible de réaliser un traitement en "temps réel" du signal vidéo, et de visualiser le résultat de ce traitement à la place de l'image d'origine. Pour cela, on utilise une fonction dite "CallBack". Cette fonction n'est pas appelée directement par notre programme : c'est le driver qui l'appelle, à chaque nouvelle image reçue de la caméra. Or ce driver ne sait pas, par défaut, le nom de la fonction que l'on veut appeler ainsi. Il est donc nécessaire de fournir au driver le nom de la fonction (c'est à dire en fait l'adresse de la routine) à appeler à chaque image, de la manière suivante :

capSetCallbackOnFrame(hWndC, CDialVideo::Traitement);

Le premier paramètre doit être le Handle de la fenêtre de capture. Le deuxième est le nom de la fonction qui effectue le traitement. Celle ci doit être déclarée de la manière suivante dans la classe CDialVideo (par exemple) :

static LPVOID CALLBACK Traitement(HWND hWndC, LPVIDEOHDR pV);

Le type et les paramètres de cette fonction sont imposés. Le mot clef *static*, obligatoire (uniquement dans le fichier .h) signifie qu'il n'existe qu'un exemplaire de cette fonction pour tous les objets de classe CDialVideo, et que cette fonction ne peut accéder qu'aux variables globales (et pas aux variables membres de CDialVideo). Pour accéder tout de même à ces variables, on déclarera en global un pointeur vers CDialVideo :
CDialVideo *PDial ;

Et dans la fonction OnInitDialog, on écrira
PDial=this ;
Et dans la fonction static, de traitement, on accédera aux données par l'intermédiare du pointeur
Ex : **PDial->Largeur** pour accéder à la largeur.

Dans la fonction de traitement, on récupère en premier paramètre le Handle de la fenêtre de capture, ainsi qu'un pointeur vers une structure de type LPVIDEOHDR (définie dans vfw.h).

```
typedef struct videohdr_tag {
   LPBYTE    lpData;              /* pointer to locked data buffer */
   DWORD     dwBufferLength;      /* Length of data buffer */
   DWORD     dwBytesUsed;         /* Bytes actually used */
   DWORD     dwTimeCaptured;      /* Milliseconds from start of stream */
   DWORD     dwUser;              /* for client's use */
   DWORD     dwFlags;             /* assorted flags (see defines) */
   DWORD     dwReserved[4];       /* reserved for driver */
} VIDEOHDR, NEAR *PVIDEOHDR, FAR * LPVIDEOHDR;
```

lpData est le pointeur vers les données (les valeurs de chacun des pixels de l'image).
dwBufferLength représente le nombre de données au total.
Dans le cas d'une image RGB, 3 octets sont nécessaires pour coder chaque pixel. Voici donc un exemple de traitement :

```
LPVOID CALLBACK CDialVideo::Traitement(HWND hwndC, LPVIDEOHDR pV)
{
  long  l;
  unsigned char *PImage;
  PImage=pV->lpData; / / PImage devient le pointeur vers les données
  int c,g1,g2,g,coul;
  long p;
  for (l=0;l<PDial->Hautgeur-1;l++) / / hauteur-1 pour éviter les débordements
  {
          for (c=0;c<PDial->Largeur-1;c++)
          {
                  for (coul=0;coul<3;coul++) / / pour les trois couleurs R, V et B.
                  {
                  p=(Pdial->Largeur*l+c)*3+coul; / / position dans le tableau de données
                  g1=abs((int)PImage[p]-(int)PImage[p+( PDial->Largeur*3)+3]);
                  g2=abs((int)PImage[p+3]-(int)PImage[p+( PDial->Largeur*3)]);
                  g=g1+g2;
                  if (g>255) g=255; / / test sur une éventuelle saturation
                  PImage[p]=g; / / gradient de Roberts
                  }
          } / / c
  } / / l
  return 0;
}
```

Si l'on ouvre une fenêtre de capture avec ce traitement, on verra donc le gradient de Roberts s'afficher en "temps réel". La vitesse dépendra évidemment de la taille de l'image et du processeur de la machine. On peut espérer faire ce type de traitement à une cadence de 8 images/s en 320x240. La carte Intel ne permet pas de faire correctement l'acquisition en RGB à un format beaucoup plus grand. Il faudrait alors dégrader le format (15 bits ou moins par pixel), et adapter le traitement en conséquence.

3 Enregistrement et traitement statique des données d'une image

Pour effectuer un traitement de type industriel, il peut être nécessaire de tout effectuer dans la fonction de traitement de type "callback". Pour détecter des défauts, par exemple, on effectuera le traitement nécessaire à leur mise en évidence, puis on prendra la décision de rejeter ou non le produit observé. Dans ce cas, il n'est pas nécessaire de sauvegarder les données. Par contre, si l'on souhaite conserver le résultat du traitement d'une seule image, il est possible d'enregistrer les données dans un fichier. On pourrait le faire en utilisant simplement une objet de classe CFile, et la fonction Write pour écrire toutes les données, dans un format de fichier "propriétaire". Il existe toutefois un grand nombre de formats de fichiers contenant des données de type image, et pour permettre une meilleure communication entre les programmes de traitement, il est préférable d'utiliser un de ces formats. Un des plus communs et pas trop difficile à coder est le format BMP. Ce format consiste en une zone "Entete" contenant des informations sur le fichier (taille de l'image, nombre de bits par pixel, etc), puis des informations éventuelles sur le codage des couleurs (la palette), et enfin les données. Dans la version la plus simple des BMP, les données ne sont pas compressées.
Quatre structures permettent de stocker les informations de format : BITMAPINFO, BITMAPINFOHEADER, RGBQUAD et BITMAPFILEHEADER. Elle sont définies comme suit :

```
typedef struct tagBITMAPFILEHEADER
{ WORD bfType; DWORD bfSize; WORD
bfReserved1; WORD bfReserved2;
                                        DWORD bfOffBits; }
                                        BITMAPFILEHEADER;

                                        typedef struct tagBITMAPINFO {
```

```
    BITMAPINFOHEADER    bmiHeader;
    RGBQUAD
bmiColors[1];
} BITMAPINFO;

typedef struct tagRGBQUAD { //
rgbq
    BYTE    rgbBlue;
    BYTE    rgbGreen;
    BYTE    rgbRed;
    BYTE    rgbReserved;
} RGBQUAD;
 typedef struct
tagBITMAPINFOHEADER{ // bmih
DWORD biSize;
LONG biWidth;
LONG biHeight;
WORD biPlanes;
WORD biBitCount
DWORD biCompression;
DWORD biSizeImage;
LONG biXPelsPerMeter;
LONG biYPelsPerMeter;
DWORD biClrUsed;
DWORD biClrImportant;
} BITMAPINFOHEADER;
```

3.1 Classe image couleur 24 bits :

Les images codées en RGB sont les plus simples à traiter car elles ne contiennent qu'un entête de fichier et un entête d'image. Les images codées sur 256 couleurs ou 256 niveaux de gris comportent en plus une table de correspondance de couleur, qu'il est nécessaire de gérer afin que l'affichage se fasse correctement.

Pour créer une classe permettant d'ouvrir des images BMP existantes, les traiter, ou encore créer des fichiers BMP à partir de données acquises, il faudra donc dériver une classe d'une MFC permettant d'ouvrir une fenêtre et de dessiner l'image. On choisira une CFrameWnd ou une CMDIChildWnd suivant le type de fenêtre (indépendante ou non) que l'on désire.
Dans cette classe, on ajoutera comme variable membre deux pointeurs :

BITMAPINFO Info; / / entête de l'image
BITMAPFILEHEADER Tete; entête du fichier

On ajoutera également quelques variables comme la Hauteur, la Largeur, etc.
On définira également dans le fichier .h une structure rgb :

struct rgb {BYTE r;BYTE g;BYTE b;};

qui permettra ensuite de définir un pointeur vers les données :

rgb * PImage;

3.1.1 Constructeur à partir d'un fichier BMP existant :

```
CImageBase::CImageBase(CString InitFileName)
{
        FileName = InitFileName;
        if (Fichier.Open(FileName, CFile::modeRead))
        {
                Fichier.Read(&Tete,sizeof(BITMAPFILEHEADER));
                if (Tete.bfType==19778) / / = 'MB'
                {
                        Fichier.Read(&Info,sizeof(BITMAPINFOHEADER));
                        Larg = Info.bmiHeader.biWidth;
                        Haut =Info.bmiHeader.biHeight;
                        / /allocation memoire du plan image
                        PImage = (rgb*) malloc(Larg*Haut*sizeof(rgb));
                        Fichier.Read(PImage,Larg*haut*sizeof(rgb));
                }
                Fichier.Close();
        }
}
```

3.1.2 Sauvegarde de l'image

```
void CImageBase::OnSauve(CString NomFichier)
{
FileName=NomFichier;
CFile Fichier; // variable fichier BITMAP
if (Fichier.Open(FileName,CFile::modeWrite|CFile::modeCreate))
  {
        Tete.bfOffBits = sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER);
        Tete.bfSize = Tete.bfOffBits+Larg*Haut*sizeof(rgb);
        Tete.bfType = 'MB'; // obligatoire
        Tete.bfReserved1 = 0;
        Tete.bfReserved2 = 0;
        Fichier.Write(&Tete,(UINT) sizeof(BITMAPFILEHEADER)); // entête du fichier
        Fichier.Write(&Info,sizeof(BITMAPINFOHEADER));   // entête de l'image
        Fichier.Write(PImage,Larg*Haut*sizeof(rgb));              // écriture des données
        Fichier.Close();

}// si ouverture du fichier ok
}
```

3.1.3 Constructeur d'une image "vide"

```
void CImageBase::CImageBase(long InitLarg, long InitHaut)
{
        Larg=InitLarg;
        Haut=InitHaut;

        Info.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
        Info.bmiHeader.biWidth = Larg;
        Info.bmiHeader.biHeight = Haut;
        Info.bmiHeader.biPlanes = 1;
        Info.bmiHeader.biSizeImage = sizeof(rgb)*Larg*Haut;
        Info.bmiHeader.biBitCount = 24;
        Info.bmiHeader.biClrUsed = 0;
        Info.bmiHeader.biCompression =  BI_RGB;
        Info.bmiHeader.biXPelsPerMeter = 0;
        Info.bmiHeader.biYPelsPerMeter = 0;
        Info.bmiHeader.biClrImportant = 0;

        PImage = (rgb*) malloc(Larg*Haut*sizeof(rgb)); // mémoire pour les données
}
```

3.1.4 Affichage de l'image

```
afx_msg void CImageBase::OnPaint(void)
{
        CPaintDC dc(this);
        SetDIBitsToDevice(dc.GetSafeHdc(),0,0, Larg, Haut , 0,0,0,Haut,PImage,&Info,DIB_RGB_COLORS);
        ReleaseDC(&dc); // le dc est relaché

} // fin OnPaint
```

3.1.5 Création de la fenêtre

Pour éviter d'appeler une fonction Create trop chargée à chaque nouvelle création de fenêtre, on peut surcharger cette fonction par exemple de la manière suivante :

```
void CImageBase::Create(LPCSTR szTitle, CMDIFrameWnd * parent)
{
        CRect rect(0,0,640,480);
        Long style = WS_CHILD | WS_VISIBLE | WS_OVERLAPPEDWINDOW;
        CMDIChildWnd::Create( AfxRegisterWndClass( CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW |
        CS_BYTEALIGNWINDOW | CS_SAVEBITS, NULL,(HBRUSH)(COLOR_WINDOW+1),NULL),
        szTitle, style ,rect,parent);
}
```

La création se résumera alors par exemple à :

```
CImageBase *Image=new CImageBase("exemple.bmp");
Image->Create("Titre",this);
```

3.2 Acquisition de l'image

Pour réaliser cette acquisition, il faut "figer" l'image en mémoire. On peut le réaliser à partir de :

```
capGrabFrameNoStop(hWndC); / / acquisition sans arrêter la visualisation
```

ou

```
capGrabFrame(hWndC); / / arrêt de la visualisation
```

Pour enregistrer l'image courante dans un format compatible avec celui de la classe CImageBase, il suffit d'appeler la fonction :

```
capFileSaveDIB(hWndC,"Nom.bmp");
```

Où le deuxième paramètre est le nom du fichier de sauvegarde.
Cette fonction transforme automatiquement les données en RGB 24 bits même si l'acquisition en cours est dans un autre mode.

Cette méthode nous oblige à passer obligatoirement par un fichier intermédiaire. Si l'on ne veut pas créer de fichier, nous avons besoin de connaître l'entête de l'image (taille, etc.) et ses données. L'entête est récupéré de la manière suivante :

```
BITMAPINFO Info;
DWORD s;
s=capGetVideoFormat(hwndC,NULL,0); / / s=taille nécessaire à l'entête
capGetVideoFormat(hwndC,&Info,s);
```

Les informations sont récupérées dans la variable Info.
Pour ce qui est du pointeur des données, il faut déclarer un pointeur en variable globale :

```
LPVIDEOHDR GlobalVideoPtr;
```

et dans une fonction de traitement, recopier le pointeur des données reçu par cette fonction en paramètre dans le pointeur global :

```
LPVOID CALLBACK CVideo::VideoFrame(HWND hwndC, LPVIDEOHDR pV)
{
        GlobalVideoPtr=pV;
        return 0;
}
```

On peut alors définir un nouveau constructeur pour la classe CImageBase, ce constructeur recevant les directement les informations d'images et ses données.

```
CImageBase::CImageBase(BITMAPINFO InfoInit, BYTE *PInit)
{
        Info=InfoInit;
        Larg=Info.bmiHeader.biWidth;
        Haut=Info.bmiHeader.biHeight;
        PImage = (rgb *)PInit;
}
```