Université François-Rabelais de Tours
Institut Universitaire de Technologie de Tours
Département Génie Électrique et Informatique Industrielle

UNIVERSITE FRANCOIS-RABELAIS
TOURS
Institut Universitaire de Technologie
Département
GENIE ELECTRIQUE ET
INFORMATIQUE INDUSTRIELLE

Études et Réalisation Informatique Industrielle

Projet CamPainter

DESTOUCHES Thibaud
BONHOMME Alexandre
2ème Année – Q2
2008-2010

Enseignants
BRUNO Armel

Université François-Rabelais de Tours
Institut Universitaire de Technologie de Tours
Département Génie Électrique et Informatique Industrielle

UNIVERSITE FRANCOIS-RABELAIS
TOURS
Institut Universitaire de Technologie
Département
GENIE ELECTRIQUE ET
INFORMATIQUE INDUSTRIELLE

Études et Réalisation Informatique Industrielle

Projet CamPainter

DESTOUCHES Thibaud
BONHOMME Alexandre
2ème Année – Q2
2008-2010

Enseignants
BRUNO Armel

Sommaire

Introduction

Alors que les interfaces homme machine ne cessent d'évoluer, on le vois notamment avec l'arrivé massive des écrans tactiles. Cependant, il existe de nombreux inconvénients a ce type d'interface, telle que le contacte physique obligatoire, l'usure ou encore le manque de maniabilité. Des défauts qui font des interfaces actuelles, des outils temporaire.

En effet, on voit de plus en plus ce développer un nouveaux type d'interface. Plus adapté a la vie quotidienne et plus pratique a utiliser. Ces mode de contrôle d'un nouveau genre utilise des techniques de traitement d'image. Des principe similaire a ceux employé dans les logiciels de retouche d'images par exemple. Mais cette fois, non pas dans le but d'améliorer l'image, mais plutôt dans l'objectif d'en retirer des information. Comme la position d'un objet, la détection d'un mouvement d'une partie du corps ou encore le repérage d'un couleur particulière. Les possibilité apparaisse alors quasi infini, puisque l'on ce rapproche alors de l'a vision qu'a l'homme du monde.

C'est dans ce cadre que nous avons décidé de nous intéresser aux bases du traitement d'images. Nous tenterons alors de réaliser une application capable à tout d'abord de localiser une source lumineuse a partir d'un flux vidéo acquis au travers une simple webcam. Puis d'utiliser cette source lumineuse comme un « marqueur », afin de pouvoir « écrire » en temps réel sur l'image affiché du flux. Ce projet étant très faste, certaines fonctions comme l'effacement de l'écran ou le changement couleur de inscriptions feront partie des options de bases, mais beaucoup d'autres sont envisageable.

1. Cahier des charges

Etant donné qu'aucun cahier des charge ne nous a été imposé, nous avons du en fixer nous même. Bien évidement, ce cahier des charges a été revu plusieurs fois au cours du projet afin de mieux coller au but final.

- Le programme dois pouvoir fonctionner indépendamment du type de webcam utilisée
- Il doit être compatible Linux et Windows
- Il doit pouvoir fonctionner sur une machine de puissance moyenne
- Il doit pouvoir détecté une source lumineuse ou un objet de couleur particulière
- La couleur des « écritures » doit être modifiable ainsi que la taille du trait tracé

2. Recherches et documentation

Tout au long de nôtre projet, nous avons effectué beaucoup de recherches, et ce exclusivement sur internet. En effet, nôtre projet étant purement informatique et les informations recherchées souvent relatives a des problèmes rencontrés au cours de la programmation. Il nous étant impossible de les résoudre grâce à une quelconque source papier.

Nous n'avons donc principalement utilisé deux site internet. Qui sont « Le site du zéro » et la documentation en ligne de la librairie OpenCV.

3. Analyse technique du projet

3.1. Pré-requis

3.1.1. Les espaces de couleurs

Tout d'abord qu'es qu'un espace de couleur ? Un « espace de couleurs » est en faite un espace vectoriel dans lequel la couleur est représentées par différents vecteurs (le plus souvent trois ou quatre). Plus simplement, c'est une manière de représenté les couleur au travers de trois ou quatre valeurs selon l'espace.

L'espace le plus connu et le plus utilisé est le RVB, pour Rouge Vert Bleu (ou RGB en anglais, pour Red Green Blue). Il est composé de trois vecteurs, le premier représente la composante rouge de la couleur, le second la verte et le dernier la bleu. Ainsi, la couleurs est composé de l'assemblage de chaque composante. Le niveaux de chaque composante déterminera la couleur obtenu. De cette manière en assemblant du rouge et du vert on obtient du jaune (comme le montre l'illustration suivante). De la même manière en ouvrant complètement les trois composantes on obtient du blanc et en les fermant on obtient du noir.

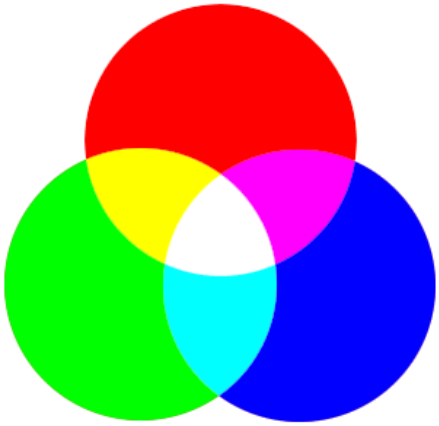


Illustration 1: Représentation classique de l'espace RVB

Le principal inconvénient de cet espace est qu'il ne sépare pas la Luminance[1] et la Chrominance[2]. Ce qui est problématique lors ce que l'on veut par exemple détecter un objet de couleur. Car la détection sera très influencé par la lumière. C'est là qu'intervient l'espace TSV.

L'espace de couleur TSV, pour Teinte Saturation Valeur (ou HSV en anglais, pour Hue Saturation Value) est issue d'une transformation non linéaire de l'espace RVB. Dans cet espace les deux premiers vecteurs représente la Chrominance. Le premier indique la teinte, ce qui correspond à la couleur pure. Le second indique la saturation de la couleur, son intensité, plus elle est faible plus la couleur sera fade. Enfin le troisième vecteur représente la Luminance, plus elle est faible plus la couleur sera sombre. De cette manière nous pourront plus facilement identifier une couleur.

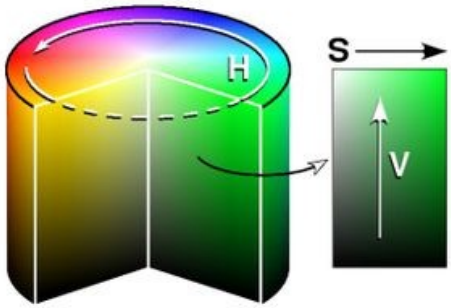


Illustration 2: Représentation de l'espace HSV

3.1.2. Les masques

Tout au long de nôtre projet nous utiliserons des masques. Il apparaît donc important, même si 'idée est très simple, d'expliquer ce qu'est un masque.

Un masque, dans notre cas, est une image binaire (Composé d'un seul vecteur de couleur, correspondant à une image en niveau de gris). Le masque est utilisé afin de ne modifier qu'une partie définie de l'image.

1 Luminance : C'est l'intensité lumineuse d'une source de lumière.
2 Chrominance : C'est l'information qui caractérise la couleur.

3.1.3. Principe d'un flux vidéo

Un fux vidéo est une succession d'images a une cadence fixe (Par exemple 25 images par seconde). Chaque images est composé de plusieurs lignes, sur lesquelles ce succèdes des points appelés Pixels[3].

3.1.4. La bibliothèque OpenCV

Afin de mener notre projet à bien, nous avons été contrait a un moment donné d'utiliser la bibliothèque « OpenCV ». Cette bibliothèque est une librairie spécialisé pour le traitement de l'image. Elle a initialement été développée par Intel et est actuellement disponible sous licence BSD[4]. On trouve au seins de cette bibliothèque diverse fonction qui nous permettrons par exemple d'acquérir facilement le flux vidéo de la webcam ou encore de convertir une image d'un espace RGB à HSV.

Il y en revanche une petite subtilité quand à la gestion de l'espace RGB par OpenCV. Effectivement, par défaut OpenCV traite les images dans l'espace BGR. Ce qui n'est pas un problème en soit, si l'on garde à l'esprit que les canaux Rouge et Bleu sont inversés.

3.2. Début du développement sous Borland C++ Builder

3.2.1. Principe de fonctionnement

Sous Borland C++ Builder en incluant les bibliothèques « vfw » et « windows » il est possible de gérer le traitement d'image et de flux vidéo. Pour capturer la vidéo, il faut tout d'abord récupérer le « handle » de la fenêtre ou le flux sera affiché, ensuite il faut capturer la vidéo à proprement parlé grâce à la fonction :

```
HDC GetDC( HWND hWnd );
```

Pour traiter ensuite cette vidéo il faut créer une image sur laquelle nous pourrons agir, pour cela il faut créer deux nouvelles variables, la première qui sera un masque temporaire poste-traitement et la deuxième qui sera en fait un tableau à deux dimensions contenant les valeurs et coordonnées de chaque pixels de l'image. Le flux vidéo capturer par la commande vue précédemment est donc copier sur le Canvas et ce sera donc celui-ci qui subira le traitement. Cette méthode bien que fonctionnelle est très peu efficace et manque énormément de fluidité, en effet le fait de copier l'image de la vidéo pour pouvoir la traiter demande énormément de ressources et obligé également à afficher la vidéo d'origine alors que cela ne nous intéresse pas. De plus étant donné que la méthode décrite ici copie le Canvas et non pas le flux vidéo à proprement parlé, le moindre déplacement de fenêtre effaçait la totalité du traitement.

3.2.2. Problèmes rencontrés

Nous avons rapidement rencontrés des difficultés dans la réalisation de notre projet sous BCB. Le premier a été le manque de fluidité de l'image final, ce qui venais de la technique de seuillage mise en place. Nous avons alors voulu utiliser une bibliothèque tierce afin de pouvoir travailler correctement sur les images. Cependant une fois de plus nous avons rencontrés des problèmes. Malgré les techniques décrite sur internet, il nous a été impossible d'utiliser la librairie

3 Pixels : Abréviation de l'anglais « Picture Element », signifiant « élément d'image ». C'est le plus petit élément composant une image.

4 Licence BSD : C'est une licence libre de droit qui permet de réutilisé tout ou une partie d'un logiciel.

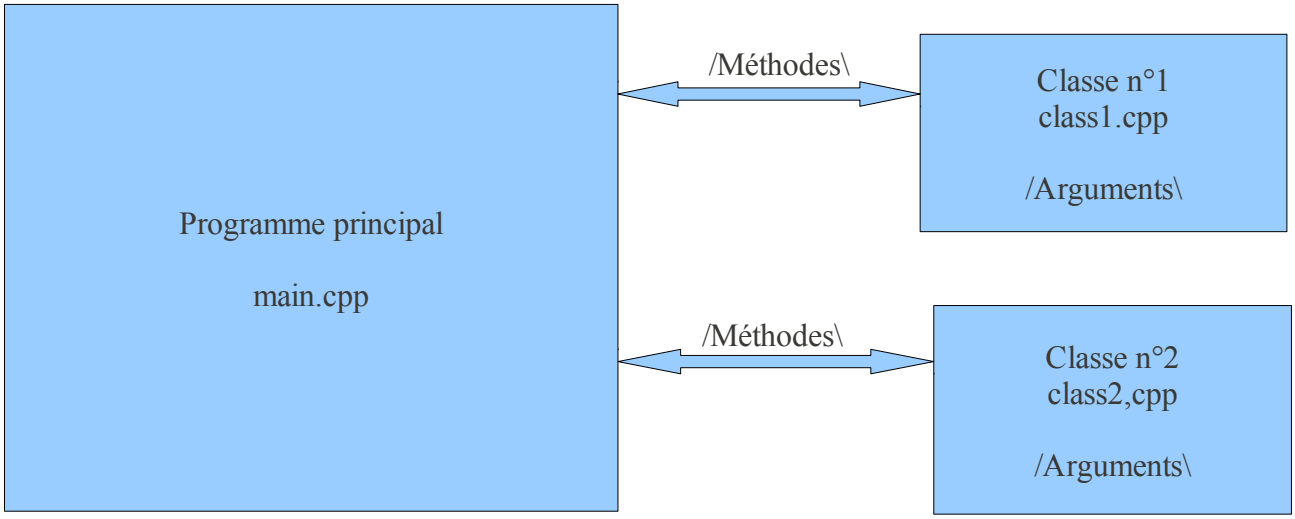
OpenCV sur BCB. Nous avons alors cherchés une alternative. Pour finalement décider de changer d'environnement de développement et nous orienter vers CodeBlocks.

3.2.3. Différences entre Borland C++ Builder et CodeBlocks

Tout d'abord la première différence que nous pouvons noté est au niveau du prix, en effet Borland C++ Builder (BCB) est un EDI payant, alors que Code Blocks (CB) est entièrement gratuit. A première vue il semblerait que BCB soit bien plus évolué que CB car ce dernier ne propose pas de bibliothèque de développement d'interface graphique pré-installé. Hors lorsque l'on y regarde de plus près il s'avère que CB est bien plus souple au niveau de l'utilisation, en effet il est possible d'installer la quasi totalité des bibliothèques disponible sur celui-ci alors que à part les bibliothèques proposés pour BCB ou à la rigueur Delphi sont utilisable sous l'EDI de Borland.

Comme dit juste avant, CB est bien plus souple que BCB mais pas uniquement pour sa possibilité d'importation de bibliothèque. Étant donné qu'aucune interface graphique n'est disponible de base sous CB un programme basique en C++ va se lancer sous la console, cela permet néanmoins une grande liberté, en effet cela oblige à apprendre la manipulation des classes, des méthodes et des arguments alors que celle-ci est très secondaire sous BCB.

Le langage C++ est dite programmation orienté objet, ces fameux objets prennent tout leur sens avec CB, en effet on peut les considérer comme des boîtes fermé, dont on ne voit pas l'intérieur qui sont comme des sous programmes (classes) avec lesquels on peut agir sur leurs variables (arguments) uniquement par le biais de fonction (méthodes). Cela permet dans un premier temps de protéger les variables de ces objets, donc on gagne en sécurité, et dans un deuxième temps cela permet d'alléger énormément le programme principal, en effet on agit sur ces sous programmes que lorsque nous en avons besoin, le reste du temps ils sont absents.



3.4. Développement sous CodeBlocks

3.4.1. Principe de fonctionnement

Sous CB, le principe de base est semblablement le même que sous BCB, dans le sens où le flux vidéo est tout d'abord capturer via la fonction :

```
CvCapture* cvCreateCameraCapture( int index );
```

Puis la vidéo est ensuite transformer en image grâce à la fonction :

```
IplImage* cvQueryFrame( CvCapture* capture );
```

C'est lors du traitement de l'image que le fonctionnement se différencie de BCB, en effet grâce à la bibliothèque OpenCV, qui est une bibliothèque spécifique au traitement d'image, tout est beaucoup plus simple car cette bibliothèque inclut des fonctions déjà toute fait pour traiter les images.

Cela nous permet donc d'avoir un programme bien plus optimisé qu'en utilisant l'EDI de Borland, le traitement des images étant allégé grâce à une bibliothèque spécifique, cela n'étant pas négociable lorsque que l'on sait les ressources que cela demande pour gérer un tel traitement en temps réel, l'affichage seul de l'image capturer via la webcam étant bien plus fluide que sous BCB.

De plus avec un découpage du programme en classe, il serai possible d'alléger encore plus le traitement, et de pouvoir optimiser encore plus le programme.

3.4.2. Détection de la lumière

La première méthode que nous avons employé pour détecter la lumière repose sur un seuillage de l'image. Nous analysons tous les pixels un par un et faisons la somme de chaque composantes BGR. Puis nous comparons cette somme à un seuil, que nous pouvons modifier grâce à une barre de réglage. Nous avons aussi crée une image binaire disposant de la même résolution que l'image initiale, que nous utilisons comme masque. De cette manière, si la somme des composantes est supérieur au seuil, nous modifions le pixel correspondant sur le masque afin de le rendre blanc. Ou dans le cas contraire pour le rendre noir.

Cependant, cette technique ne nous permet pas d'avoir un effet « mémoire ». Car le but étant « d'écrire » sur l'image. Il nous faut donc mémoriser les zones écrites. Nous avons alors rajouté une condition lors de la génération du masque. De telle sorte que si le pixel est déjà blanc nous ne pouvons plus le rendre noir.

```
for(int y = 0; y < CAM->height; y++)
        for(int x = 0; x < CAM->width; x++)
        {
                Pixel = cvGet2D(CAM, y, x);
                Px_Masque = cvGet2D(CAM_Masque, y, x);

                int BGR = Pixel.val[0] + Pixel.val[1] + Pixel.val[2];

                if((BGR/3) > seuil)
                {
                        if(Px_Masque.val[0] != 255)
                        {
                                Px_Masque.val[0] = 255;
                                cvSet2D(CAM_Masque, y, x, Px_Masque);
                        }
                }
                else
                {
                        if(Px_Masque.val[0] != 255)
                        {
                                Px_Masque.val[0] = 0;
                                cvSet2D(CAM_Masque, y, x, Px_Masque);
                        }
                }
```

```
}
```

Une fois le masque généré il faut effectuer le modification de l'image final en fonction des zones blanc ou noir. Nous avons alors effectué une simple comparaison. Si le pixel du masque est blanc, nous rendons blanc le pixel correspondant sur l'image final. Dans le cas contraire aucune modification n'est effectué.

Nous avons ensuite remplacé le blanc par une couleur qui sera paramétrable grâce à trois barres de réglages (Rouge, Vert et Bleu).

```
for(int y=0; y<CAM->height; y++)
        for(int x=0; x<CAM->width; x++)
        {
                Px_Masque = cvGet2D(CAM_Masque, y, x);

                if(Px_Masque.val[0] == 255)
                {
                        Pixel.val[0]=Blue;
                        Pixel.val[1]=Green;
                        Pixel.val[2]=Red;

                        cvSet2D(CAM, y, x, Pixel);
                }
        }
```

Nous avons aussi ajoutés une option qui nous permet par un appuie sur la touche « c » d'effacer toutes les écritures. Voici le détail de la fonction qui est appelé pour effacer le masque :

```
void ClearImg(void)
{
        for(int y = 0; y < CAM->height; y++)
                for(int x = 0; x < CAM->width; x++)
                {
                        Px_Masque.val[0] = 0;
                        cvSet2D(CAM_Masque, y, x, Px_Masque);
                }
}
```

Malheureusement, lors du l'utilisation de cette technique nous avons rencontrés des problèmes du aux zones clairs ou lumineuses qui n'était pas éliminables par le seuillage. Nous avons alors du avoir recours à l'utilisation d'un filtre.

3.4.3. Le filtre

Le but de notre filtre sera de supprimer les pixels blancs isolés. De manière a limiter au maximum les écritures parasites. Pour effectuer ce traitement nous avons d'abord mise en place ce système sur une matrice de pixel 3x3 (Un carré de trois pixels de largeur et trois pixels de hauteur). Nous nous somme par le suite tournée vers une matrice 5x5 pour plus d'efficacité. Mais le principe reste le exactement même.

255	255	0
255	255	0
0	0	255

Tableau 1: Exemple de matrice 3x3

Nous utilisons une matrice impaire (3x3, 5x5 ou 7x7) afin de pouvoir centrer le pixel a traité. De cette façon nous faisons la moyenne des pixel encadrant le point centrale. Et si cette moyenne est inférieur à un seuil on rend le pixel noir. Dans notre cas le seuil était de 127, car cela correspond à la moitié des pixels de la matrice blanc.

Ex: 255 + 255 + 0 + 255 + 0 + 0 + 0 + 255 / 8 = 127

La valeur 255 correspond à un pixel blanc et 0 à un pixel noir. On voit que si la moitier des pixel sont blanc la moyenne vaut 127. Donc le pixel centrale (qui n'est bien sur pas utilisé dans la moyenne) deviendra ou restera blanc.

En utilisant cette technique sur un matrice 5x5 nous avons pu éliminer énormément de pixels parasites. Cependant, la technique d'écriture par détection de la couleur, est très influencée par la luminosité ambiante. Nôtre projet ne pouvait donc pas fonctionner de manière optimale dans toute les conditions. Nous nous somme alors tournée vers l'écriture par détection de couleurs.

3.4.4. Détection de la couleur avec l'espace HSV

Afin de mieux détecter la couleur, nous avons décidé de passer de l'espace BGR à l'espace HSV. De cette manière nous pourrons mieux traiter la chrominance. Pour effectuer ce changement d'espace qui est une opération complexe nous avons utilisé une fonction fournis par la librairie OpenCV. Voici le prototype de la fonction :

```
void cvCvtColor( const CvArr* src, CvArr* dst, int code );
```

Comme le montre le prototype, on donne comme paramètre une image source nommé « src », une image de destination « dst » et un code qui correspondra à la conversion souhaité.

Attention, lors de la déclaration d'une image avec OpenCV le type utilisé est « IplImage ». Hors, dans les prototype des fonctions disponible dans OpenCV, on trouve toujours le type « CvArr ». « CvArr » est en réalité une metatype utilisé dans les prototype pour signaler que la fonction accepte plusieurs type de structure tel que « IplImage », « CvMat » ou encore « CvSeq ». Nous n'avons utilisez que le type « IplImage », mais nous aurions aussi pu utiliser « CvMat » le résultat serait le même.

Nous avons donc utilisé le code suivant pour effectuer notre conversion BGR/HSV :

```
cvCvtColor(CAM, CAM_HSV, CV_BGR2HSV);
```

Une fois notre image HSV crée, nous avions besoin de récupérer les valeurs de teinte et de saturation de l'objet à détecter. Pour cela vous avons crée une fonction qui récupère la valeur de teinte et de saturation du pixel cliqué.

```
void Souris(int event, int x, int y, int flags, void *param=NULL)
{
```

```
        if(event == CV_EVENT_LBUTTONUP)
        {
                CvScalar scal;
                scal = cvGet2D(CAM_HSV, y, x);
                Color = scal.val[0];
                Sat = scal.val[1];
                ClearImg(CAM_Masque_Cercle);
        }
}
```

Cette fonction doit être appelé à chaque action de la souris. Et si l'action est une clique sur le bouton gauche, alors on récupère les valeurs désiré. On en profite pour effacer le filtre et indirectement l'image final.

Pour exécuter nôtre fonction automatiquement nous avons utilisé une fonction fournis par OpenCV.

```
cvSetMouseCallback("HSV", Souris);
```

Le premier argument correspond au nom de la fenêtre sur la quelle on veut surveiller l'évènement de la souris. Et le deuxième est le nom de la fonction à exécuter en cas d'évènement.

Il nous a ensuite fallu procéder à la modification du masque d'une manière similaire à celle utilisé dans l'espace RGB. A l'exception prêt que cette fois la valeur du seuil sera fixé par les valeurs recueilli précédemment. Cette fois si nous ne chercherons pas si la couleur de l'image est au dessus du seuil mais si elle correspond au seuil. En effet nous voulons maintenant détecter un couleur précise et non plus un niveau de lumière.

Pour simplifier la condition qui nous permettra de détecter la couleur nous avons mis en place une macro qui est la suivante :

```
#define seuil(nb, cherche, tol) (nb < (cherche+tol) && nb > (cherche-tol))
```

On défini « seuil(nb, cherche, tol) » qui a pour arguments « nb », « cherche » et « tol ». Si « nb » est inférieur à « cherche + tol » et supérieur à « cherche - tol » alors « seuil(nb, cherche, tol » est vrai.

Nous allons maintenant récupérer les valeurs de chrominance de chaque pixels pour l'image HLV. Pour ensuite effectuer une comparaison grâce à la macro définie précédemment. Si le pixel rentre dans la marge de tolérance nous effectuons la même manipulation sur le masque que pour la détection de lumière.

3.4.5. Détection du centre de l'objet

Afin d'obtenir un tracé propre nous avons ajouté quelques modifications par rapport à la technique employé lors de la détection par couleur. Nous avons décidé de calculer la positon du centre de l'objet de manière à pouvoir remplacer la zone détecté par un cercle.

Pour arriver à nos fins nous avons du créer un nouveau masque, possédant les mêmes caractéristiques que le premier. De plus il a fallu apporter un modification à la partie détection de la couleur. L'idée étant de calculer la moyenne des coordonnée des points détectés.

Ainsi, a chaque fois qu'un pixel est détecté comme étant de la bonne couleur on additionne sa coordonnée en X à la somme des coordonnée en X et on effectue le même calcule pour les coordonnée en Y. De plus on incrémente un compteur à chaque pixel détecté. Ce qui nous permettra

de calculer la moyenne des X et des Y. Et nous obtenons les coordonnées du pixel au centre de la détection.

3.4.6. Tracé des cercles

Une fois les coordonnées calculés il ne nous reste plus qu'a tracer le cercle. Il est très simple de tracer un cercle grâce à OpenCV. Voici le prototype de la fonction le permettant :

```
void cvCircle( CvArr* img, CvPoint center, int radius, CvScalar color, int thickness=1,
               int line_type=8, int shift=0 );
```

Le premier argument correspond à l'image sur laquelle le point va être dessiné. Dans les arguments de cette fonction on découvre « CvPoint », qui est une structure définissant simplement un point. Elle dispose de deux paramètres, sa coordonnée en X et en Y. Les coordonnées de ce point définirons le centre du cercle. L'argument « radius » va comme son nom l'indique nous permettre de définir le rayon du cercle. On passe aussi en paramètre un vecteur « CvScalar », qui représentera la couleur de notre cercle. Sur OpenCV un pixel est représenté par la structure « CvScalar », qui contient les valeurs des différents vecteurs de couleurs. Le paramètre « thickness » permet de préciser l'épaisseur du trait lors du tracé du cercle. On peu aussi obtenir un cercle plein en le passant à -1. Les deux derniers paramètres nous permets de définir certaines options comme l'anti-crénelage[5] (Anti-Aliasing).

Nous allons donc tracé le cercle de cette façon :

```
cvCircle(CAM_Masque_Cercle, cvPoint(PosX,PosY), rayon, Pixel, -1);
```

On dessine ainsi sur le nouveau masque une cercle à chaque détection. « PosX » et « PosY » serons les coordonnées du centre calculé précédemment.

Cependant, si les mouvements sont trop rapides par rapport au temps de traitement le trait apparait comme une série de point. On obtient donc un trait discontinu. Pour comblé ces « vides » nous avons donc relié les points entre eux par des segments.

3.4.7. Tracé des lignes

Afin de tracer ces segments nous avons une fois de plus utilisé une fonction fournis par OpenCV :

```
void cvLine( CvArr* img, CvPoint pt1, CvPoint pt2, CvScalar color, int thickness=1,
               int line_type=8, int shift=0 );
```

On retrouve tout les paramètres présent dans la fonction de dessin des cercles. Cette fois « pt1 » sera le point de départ de la ligne et « pt2 » la fin.

Il nous faudra donc mémoriser les coordonnées du point précédent afin de le relier avec le suivant. Pour cela nous avons légèrement modifié la partie de dessin des cercle pour aboutir au code suivant :

```
if(SommeX > 0 && SommeY > 0 && Cpt  >NbPix)
{
        PosX = SommeX/Cpt;
        PosY = SommeY/Cpt;
```

5 Anti-crénelage : C'est un méthode permettant d'enlever le crénelage, l'effet « escalier » que l'on peut avoir sur certaines images.

```
            if(Write)
            {
                    cvCircle(CAM_Masque_Cercle, cvPoint(PosX,PosY), rayon, Pixel, -1);

                    if(PosXp != 0 && PosYp != 0)
                            cvLine(CAM_Masque_Cercle, cvPoint(PosXp,PosYp), cvPoint(PosX,PosY),
                                    Pixel, rayon*2);
            }

            PosXp = PosX;
            PosYp = PosY;
    }
    else PosX = PosY = 0;
```

Nous avons aussi ajouté une fonction permettant interrompre l'écriture lors d'un appuie sur la touche « p », ce qui passe le flag « Write » à « faux » et empêche le dessin des cercles et des lignes. Nous reviendrons sur quelques détails de cette fonction ultérieurement.

3.4.8. Inversion de l'image

Une fois l'écriture réalisée il ne nous reste plus qu'à afficher l'image modifié. Cependant, pour faciliter l'écriture nous avons effectué une inversion verticale de l'image.

```
void InvImg(IplImage *IMG_src, IplImage *IMG_dst)
{
        int Height = IMG_src->height;
        int Width = IMG_src->width;

        for(int y = 0; y < Height; y++)
                for(int x = 0; x < Width; x++)
                        cvSet2D(IMG_dst, y, x, cvGet2D(IMG_src, y, (Width - 1) - x));
}
```

On passe en paramètres une images source « IMG_src » et une image de destination « IMG_dst ». Les pixels sont alors inversé un par un.

3.4.9. Détails de la fonction « Pause »

Lors de l'appuie sur la touche « p » un texte s'affiche signalant si l'écriture est active ou ne l'est pas. Pour cela nous avons une fois de plus utilisé une fonction dédié d'OpenCV.

```
void cvPutText( CvArr* img, const char* text, CvPoint org, const CvFont* font, CvScalar color );
```

La seule chose inconnu dans ce prototype est la structure « CvFont ». Effectivement, pour écrire il faut définir une police d'écriture. Et bien-sur une fonction d'OpenCV s'en charge très bien :

```
void cvInitFont( CvFont* font, int font_face, double hscale, double vscale, double shear=0,
                int thickness=1, int line_type=8 );
```

Le premier paramètre est la structure « CvFont » que nous aurons précédemment déclaré. Ensuite pour l'initialiser nous devons préciser d'autres paramètres.

« font_face » défini le type de la police utilisé. Nous utiliserons dans nôtre cas « CV_FONT_HERSHEY_COMPLEX_SMALL ».

Les deux paramètres « hscale » et « vscale » correspondent respectivement à la taille horizontale et verticale du lettrage. Pour une taille normale nous mettrons ces paramètres à 1.

« shear » indique l'angle d'inclinaison de la police. Pour une police droite nous le laisserons par défaut à 0.

« thickness » indique l'épaisseur de l'écriture. Dans nôtre cas ce paramètre sera à 2.

Puis enfin, comme pour les cercles et les lignes, le dernier paramètre « line_type » défini la connexité de la police. Nous laisserons donc ce paramètre par défaut.

```
cvInitFont(&font, CV_FONT_HERSHEY_COMPLEX_SMALL, 1, 1, 0, 2);
```

Une fois nôtre police défini il ne reste plus qu'à l'afficher convenablement car à la fonction « cvPutText » :

```
if(Write) cvPutText(CAM_Inv, "Pen: ON", cvPoint(5,20), &font, cvScalar(0,255,0));
else cvPutText(CAM_Inv, "Pen: OFF", cvPoint(5,20), &font, cvScalar(0,0,255));
```

De cette manière, un texte vert « Pen: ON » sera affiché en haut à gauche de l'image lorsque l'écriture est effective. Et un texte rouge « Pen: OFF » dans le cas contraire.

3.4.10. Affichage de l'image final

L'affichage est effectué grâce une fonction fournis par OpenCV. Il faudra cependant préalablement déclarer une « fenêtre » dans laquelle nous afficherons l'image. Voici les deux fonctions que nous avons utilisé :

```
void cvNamedWindow(const char *titre_fenetre, int flag);
void cvShowImage(const char *titre_fenetre, IplImage *image);
```

La première fonction nous permet de créer la fenêtre en lui donnant un titre et en précisent en deuxième argument « CV_WINDOW_AUTOSIZE », ce qui permettra que la fenêtre s'adapte automatiquement à l'image.

La deuxième fonction a pour but d'afficher une image donnée en deuxième paramètre dans une fenêtre identifié par son nom passé en premier paramètre.

3.4.11. Structures annexes

Toute la partie traitement de l'image est réalisé dans une boucle. De cette manière on traite le flux vidéo image par image. La condition de sortie est un appuis sur la touche « q ». Pour capter cet appuis (et le principe est le même pour les touche « c » et « p » utilisé précédemment), nous utilisons une fonction fournis par OpenCV :

```
char cvWaitKey(int delay);
```

Cette fonction attend l'appuis sur une touche pendant un temps donné. Ce temps est passé en paramètre et est en millisecondes. Elle va nous permettre de temporiser notre boucle. Pour utiliser raisonnable les ressources du système nous avons fixé ce délais à 50ms.

Une fois le programme terminé, il faut libérer les ressources utilisées. Pour cela on « détruis » les fenêtres et la capture grâce à ces fonctions :

```
void cvDestroyWindow( constchar *titre_fenetre );
void cvReleaseCapture( CvCapture** capture );
```

3.4.12. Problème rencontrés

Nous avons rencontrés divers problèmes au cours de ce projet. Notamment des problèmes de gestion des ressources utilisé par nôtre programme. A commencé par la gestion de la mémoire.

1. La fuite mémoire

Nous avons en effet eu un problème de fuite mémoire au cours du développement. Nous nous somme retrouvé avec un programme saturant la mémoire de notre ordinateur. Après quelque recherche nous avons trouvé la source du problème qui venait du morceau de code suivant :

```
CAM_Inv = cvCloneImage(CAM);
```

Nous utilisions la fonction « cvCloneImage() » pour crée facilement une image ayant les mêmes caractéristiques que l'image source (l'image tiré du flux à un instant T). Cependant, cette partie du programme se trouvant au sein d'une boucle, l'image était constamment crée à chaque tour de la boucle. Ce qui venait très rapidement à occuper plus de 400Mo de mémoire vive.

Pour régler ce problème nous avons utilisé une méthode simple. Il ne fallait créer les images qu'une seule fois. Nous avons donc mit en place une condition avec un variable booléenne nous servant de « flag » pour exécuter une seule et unique fois la partie création des images. Nous avons par la même occasion changé la fonction utilisé pour la création des image. Car il semblerai que celle-ci ne soit pas la plus performante. Nous utilisons donc dorénavant la fonction « cvCreateImage() ».

```
if(flag)
{
	Size = cvGetSize(CAM);
	Depth = CAM->depth;

	CAM_Inv = cvCreateImage(Size, Depth, 3);
	CAM_Masque = cvCreateImage(Size, Depth, 1);
	CAM_Masque_Cercle = cvCreateImage(Size, Depth, 1);
	CAM_HSV = cvCreateImage(Size, Depth, 3);

	flag = false;
}
```

L'indicateur « flag » est préalablement déclaré et a pour valeur initial « vrai ». Une fois les images crée il passe à « faux » jusqu'à la fin du programme.

Les deux premières ligne servent à récupérer la taille « Size » et la profondeur des couleurs de l'image « Depth ». Pour la taille nous utilisons la fonction « cvGetSize() ». Cette fonction renvoie une structure du type « CvSize » qui contient les paramètres de hauteur et de largeur de l'image. Pour la profondeur on accède directement à cette valeur qui est un entier que l'on stocke dans la variable « Detph ». De cette façon nous avons toutes les informations nécessaire à la création des différentes images basées sur l'image du flux « CAM ».

2. L'utilisation importante des ressources

Un important problème au quel nous avons du faire face est l'utilisation trop importante des ressources du système. Nous n'avons actuellement pas réussi à régler ce problème. Malgré quelques améliorations, l'utilisation du processeur reste trop importante. A titre indicatif le programme en fonctionnement utilise environs de 50% à 60% du processeur et ce sur une machine de type Core2Duo à 3Ghz. En revanche l'utilisation de la mémoire est elle très correct puisque autour des 4Mo.

Nous avons en revanche quelques idées que nous avons pas eu le temps de mettre en place. Comme l'utilisation des « zone d'intérêt » qui nous permettrez non plus de travailler sur l'image entière, mais uniquement sur une zone restreinte autour de l'objet détecté. Puisque d'après nôtre analyse du problème l'utilisation importante du processeur viendrait des multiples boucles utilisées dans nôtre code.

3. Le portage de l'application sous windows

Le ayant été développé sur un système à base de noyau Linux (pour la partie du développement effectué sous CodeBlocks), nous voulions effectuer le portage du programme sur la plateforme Windows. Bien qu'en théorie ce portage semble facile, vu que l'EDI[6] CodeBlocks et la librairie OpenCV sont disponibles en version Windows et Linux, il c'est avéré impossible pour nous de l'effectuer. Le code qui est pleinement fonctionnel sous Linux génère une erreur inconnue lors de ca compilation sous windows. Malgré nos recherches ils nous a été impossible de décelé la source du problème. Nous avons quelques pistes comme un problème de version de librairie ou une mauvaise configuration de l'EDI sous Windows. Mais rien ne nous permet actuellement de faire fonctionner notre programme sous Windows.

6 EDI : Environnement de Développement Intégré (IDE pour Integrated Development Environment en anglais). C'est un programme regroupant des outils permettant le développement de logiciels. On y trouve en générale un éditeur de texte, une compilateur et un débogueur. On peu aussi y trouver des outils de création d'interfaces graphiques comme dans BCB.

Conclusion

Arrivé au terme de ce projet, du moins scolairement car il peut encore être grandement exploité, c'est sans doute un des plus intéressant sur lequel nous avons pu travailler au cours du DUT. Et ce en grande partie car le choix en était libre.

Ce sujet a donc été très enrichissent pour nous, car il nous a permis de nous plonger dans le traitement d'image, domaine d'on nous avions acquis les bases durant plusieurs cours. Il a été intéressant pour nous de mettre en pratique ces connaissances et d'en acquérir de nouvelles. Nous avons ainsi pu découvrir l'existence de librairies spécialisées et comprendre le fonctionnement basique du traitement de flux vidéo. Nous avons aussi découvert des techniques de traitements particulières des images, comme par exemple les opérateurs morphologique, les différents types de filtres applicable sur une image ou encore les techniques employées lors de la détection de contours. Même si nous n'avons pas mis en place toutes ces techniques au sein de nôtre programme, il a quand même été intéressant de les aborder durant nos recherches. Nous aurons certainement l'occasion de les utiliser dans de futurs projets.

Le bilan de ce projet est donc très positif, autant sur le plan technique que humains. Nôtre travaille en binôme c'est très bien déroulé et fut fortement enrichissant et très constructif.

Index des illustrations

Bibliographie