XU Sisong
ZHANG Jingwei
PAPET Nicolas
LATOUR Rodolphe

PROJET TARGET DRONE

COMPTE RENDU DE PROJET

Groupe D2

TABLE DES MATIERES

INTRODUCTION

A l'issue du projet S8 de notre deuxième année d'enseignement, notre équipe à choisie de travailler sur le projet de suivi de personne par asservissement visuel d'une unité mobile, en particulier un drone Bebop II de l'entreprise Parrot. Différentes méthodes seront mises en avant durant ce rapport cherchant à respecter les critères imposés par notre cahier des charges. En particulier, par détection de couleur, reconnaissance de formes et par Deep Learning. Le délai de réalisation du projet est de cinq semaines, allant du 26 mars au 27 avril. Le drone présente l'avantage particulier de fonctionner avec le Middleware Robotic Operating System (ROS), ce qui offre une très grande versatilité quant à l'implémentation de différents programmes permettant son pilotage.

Ceci impliqua ainsi la nécessité de découvrir le fonctionnement de ce middleware et de savoir comment user de son potentiel pour l'implémentation de nos méthodes de détections et de la réaction comportementale du robot selon les instructions requises. Outre le système ROS, il était indispensable d'utiliser la librairie OpenCV pour le traitement d'image. Cette bibliothèque graphique contient une quantité importante de fonction qui nous permis de réaliser la grande majorité des méthodes nécessaires à l'aboutissement du projet. Il était également nécessaire d'avoir des connaissances préalables sur les langages de programmation C++ et Python, tous deux étant parfaitement adaptés pour fonctionner sur l'environnement ROS.

REPARTITION DU TRAVAIL AU SEIN DE L'EQUIPE

Le tableau suivant réfère la répartition des différentes tâches menant à l'aboutissement du projet effectuées par les membres de l'équipe.

Membres de l'équipe	Jingwei ZHANG	Sisong XU	Nicolas PAPET	Rodolphe LATOUR
Préparation de l'environnement de travail (installation de Linux Ubuntu 16.04 LTS, ROS Kinetic Kame…)	X	X	X	X
Création du nœud gérant le comportement du drone pour le suivi par couleur	X	X		
Création du nœud gérant la détection de couleur				X
Création du nœud gérant la détection de forme – HAAR Cascade	X			
Création du nœud gérant la détection de forme – HOG DPM		X		
Création du nœud gérant le comportement du drone pour la détection de forme			X	
Documentation pour le suivi par Deep Learning				X

#TODO

Cette étape fondamentale à la compréhension du fonctionnement de l'intégralité des outils étant à notre disposition fut certainement la méthode la plus simple à implémenter. Deux nœuds furent utilisés pour réaliser cette méthode.

DETECTION DE LA COULEUR

Afin de bien saisir le fonctionnement du nœud, chaque étape du code sera expliquée et détaillée.

Dans un premier temps, il est nécessaire d'implémenter toutes les librairies nécessaires. Nous utilisons dans ce code des fonctions permettant de transmettre les différents type de messages à travers des topics dans le système ROS ;

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <std_msgs/Int32.h>
#include <geometry_msgs/Point.h>
```

Trois fenêtres sont affichés lors de l'exécution du nœud : `OPENCV_raw` affiche l'image brut reçue du drone, `OPENCV_HSV` affiche l'image en HSV et `OPENCV_thresholded` affiche l'image binarisée avec en blanc la couleur cible.

```
static const std::string OPENCV_raw = "raw image";
static const std::string OPENCV_HSV = "HSV image";
static const std::string OPENCV_thresholded = "Thresholded image";
```

Le programme est écrit dans une classe nommée `ImageConverter`. Un `NodeHandle` est défini pour initialiser le note et pour pouvoir faire les Subcribers, Publisher et Advertiser nécessaires. Ensuite, on définit l'objet `it_` de la classe `ImageTransport` qui nous permet de publier et souscrire à des messages de type `image_transport`. En l'occurrence, ici, la vidéo que l'on veut recevoir. Egalement, on définit un Publisher `publ` qui servira à transmettre la position du barycentre et la quantité de pixel blanc sur l'image binarisée.

```
class ImageConverter
{
  ros::NodeHandle nh_;
  image_transport::ImageTransport it_;
  image_transport::Subscriber image_sub_;
  image_transport::Publisher image_pub_;
  ros::Publisher publ;

public:
  ImageConverter()
    : it_(nh_)
  {
```

Ici, nous nous abonnons au topic `/bebob/image_raw` qui correspond à l'image brute transmise par le Bebop II. On annonce ensuite que le nœud publiera des messages sur `/image_converter/output_video` de type `image_transport` et des messages sur `/cible` de type `geometry_msgs`.

```
// Subscrive to input video feed and publish output video feed
image_sub_ = it_.subscribe("/bebop/image_raw", 1,
  &ImageConverter::imageCb, this);
image_pub_ = it_.advertise("/image_converter/output_video", 1);
publ = nh_.advertise<geometry_msgs::Point>("/cible", 1000);
```

Nous définissons et nommons nos trois fenêtres qui dépendront de la bibliothèque openCV.

```
  cv::namedWindow(OPENCV_raw);
  cv::namedWindow(OPENCV_HSV);
  cv::namedWindow(OPENCV_thresholded);
}
```

Nous détruisons nos trois fenêtres si l'on appelle `~ImageConverter()` dans le main du programme.

```
~ImageConverter()
{
  cv::destroyWindow(OPENCV_raw);
  cv::destroyWindow(OPENCV_HSV);
  cv::destroyWindow(OPENCV_thresholded);
}
```

On définit la fonction `imageCb` qui contient l'intégralité des fonctions permettant le traitement de l'image. Elle reçoit en entrée la vidéo du drone qui est convertie pour être utilisée par la librairie d'OpenCV.

```
void imageCb(const sensor_msgs::ImageConstPtr& msg)
{
  cv_bridge::CvImagePtr cv_ptr;
  try
  {
    cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
  }
  catch (cv_bridge::Exception& e)
  {
    ROS_ERROR("cv_bridge exception: %s", e.what());
    return;
  }
```

On récupère dans une matrice d'OpenCV `imgRGB` l'image `cv_ptr` convertie par le module `cv_bridge` afin de pouvoir effectuer les transformations nécessaires. On crée ensuite une matrice HSV qui nous permettra de récupérer la couleur de manière plus précise par la suite.

```
cv::Mat imgRGB = cv_ptr->image;
cv::Mat imgHSV;
cv::cvtColor(imgRGB, imgHSV, cv::COLOR_BGR2HSV);
```

On crée ensuite la matrice qui contiendra l'image binarisée `imgThresholded` et définissons les valeurs de Hue, Saturation et Value nous permettant de récupérer ici une couleur orange.

```
cv::Mat imgThresholded;

int iLowH = 0;
int iHighH = 30;

int iLowS = 120;
int iHighS = 235;

int iLowV = 120;
int iHighV = 255;
```

Une fois ces valeurs définis, il nous reste plus qu'à définir ce que la matrice `imgThresholded` devra contenir grâce à la fonction `inRange`.

```
    // Perform binarisation
    cv::inRange(imgHSV, cv::Scalar(iLowH, iLowS, iLowV), cv::Scalar(iHighH,
iHighS, iHighV), imgThresholded);
```

La manipulation suivante nous permet en premier lieu de supprimer les petits objets de couleur orange pouvant survenir sur l'image et ensuite de remplir les trous de petites tailles compris dans des gros amas de pixels. Il s'agit en quelque sorte d'un effet de « blur » affinant la précision de notre tracking.

```
    //morphological opening (remove small objects from the foreground)
    cv::erode(imgThresholded, imgThresholded,
cv::getStructuringElement(cv::MORPH_ELLIPSE, cv::Size(5, 5)) );
    cv::dilate( imgThresholded, imgThresholded,
cv::getStructuringElement(cv::MORPH_ELLIPSE, cv::Size(5, 5)) );

    //morphological closing (fill small holes in the foreground)
    cv::dilate( imgThresholded, imgThresholded,
cv::getStructuringElement(cv::MORPH_ELLIPSE, cv::Size(5, 5)) );
    cv::erode(imgThresholded, imgThresholded,
cv::getStructuringElement(cv::MORPH_ELLIPSE, cv::Size(5, 5)) );
```

La binarisation effectuée, il ne nous reste plus qu'à calculer la position du barycentre, le nombre de pixels blanc présent dans la matrice, et à dessiner un réticule à cette position sur la matrice `imgRGB`. Pour les deux premières étapes, la fonction `Moments` d'OpenCV nous permet de le faire très simplement.

```
//Barycentre calculation

cv::Moments moment = cv::moments(imgThresholded, true);

float surface = moment.m00;
float x = moment.m10/surface;
float y = moment.m01/surface;


cv::Point point(x, y);
cv::circle(imgRGB, point, 15, cv::Scalar(255, 255, 255), 2);
cv::circle(imgRGB, point, 1, cv::Scalar(0, 0, 255), 2);
```

Ensuite, nous affichons les trois fenêtres contenant les matrices `imgRGB`, `imgHSV` et `imgThresholded`.

```
// Update GUI Window
cv::imshow(OPENCV_raw, imgRGB);
cv::imshow(OPENCV_HSV, imgHSV);
cv::imshow(OPENCV_thresholded, imgThresholded);
cv::waitKey(3);

// Output modified video stream
image_pub_.publish(cv_ptr->toImageMsg());
```

La dernière étape consiste à émettre notre message `publ` dans notre topic `/cible`. Nous définissons nous trois valeurs à émettre sur les trois composantes `x`, `y` et `z` du type de message `Point`. `ROS_INFO()` nous permet d'afficher dans ce nœud les valeurs qui seront émises.

```
//Publish position and surface

    geometry_msgs::Point myPos;
    myPos.x = x;
    myPos.y = y;
    myPos.z = surface;

    ROS_INFO("x : %f", myPos.x);
    ROS_INFO("y : %f", myPos.y);
    ROS_INFO("surface : %f", myPos.z );

  publ.publish(myPos);

  }
};
```

Nous définissons le `main` du programme. Il exécute d'abord l'initialisation de ce nœud nommé `image_converter`. Ensuite il exécute l'objet `ic` de la classe `ImageConverter` précédemment écrit. `ros::spin()` permet d'exécuter ce programme en boucle.

```
int main(int argc, char** argv)
{

  ros::init(argc, argv, "image_converter");
  ImageConverter ic;
  ros::spin();
  return 0;

}
```

Pour ce nœud, il sera nécessaire d'utiliser les utilitaires permettant les déplacements du drone.

```
#include <drone_test/keyboard_controller.h>
#include <iostream>
#include <ncurses.h>
#include <unistd.h>
```

On définit ensuite les variables qui seront utilisées dans ce programme. C'est également ici que les valeurs limites permettant le respect des conditions de déplacement sont imposées.

```
//position couleur
int couleur_x;
int couleur_y;
//distance entre couleur et drone
int distance_max=100;
int distance_x=400;
int distance_y=200;
//taille image
int taille_image=856;
//position centrale de l'ecran
//int x_min = taille_image*(5/11);
//int x_max = taille_image*(6/11);
int x_min = 375;
int x_max = 425;
//int pos y
int y_min = 150;
int y_max = 250;
//tracking actif a 1
int track=0;
//vitesse de deplacement
float vitesse_x = 0.0;
float vitesse_y = 0.0;
float taille_h;
float yaw;
```

La fonction posCallback permet de récupérer les valeurs émises par le node `image_converter`.

```
//recuperation de la position couleur
void  posCallback(const geometry_msgs::Point myPos)
{
    couleur_x=myPos.x;
    couleur_y = myPos.y;
    taille_h = myPos.z;
}
```

La fonction `keypressed` permet de vérifier si une saisie clavier a été effectuée.

```
int keypressed(void)
{
    int ch = getch();

    if (ch != ERR) {
        ungetch(ch);
        return 1;
    } else {
        return 0;
    }
}
```

Définitions de la fonction main dans lequel se trouve l'intégralité du programme. On initialise les fonctions de ROS nécessaire et s'abonne au topic `/cible` pour récupérer les valeurs de positions et de surface.

```
int main(int argc, char **argv)
{

  ros::init(argc, argv, "autocontroller");
  ros::NodeHandle n;
  ros::Rate loop_rate(30);
  ros::Subscriber sub = n.subscribe("/cible", 1000, posCallback);


  DroneController bebop;
  float avancement = 0.00, translation = 0.00, hauteur = 0.00, rotation = 0.00;
  initscr();

  cbreak();
  noecho();
  nodelay(stdscr, TRUE);
```

La condition tant que est respectée si la connexion du nœud à ROS est établie (i.e le programme n'a pas été arrêté volontairement ou involontairement). Ensuite, nous avons dans le cas où une saisie clavier est effectuée des messages qui seront transmis au drone. Cela permet de contrôle le drone depuis ce nœud. La saisie clavier « v » permet d'activer ou désactiver l'opération de tracking de cible. Toute autre saisie clavier référencée ici permet de désactiver le tracking.

```
  while(ros::ok())
  {
   if (keypressed())             // a key is pressed
   {
       switch(getch()) // ASCII values
        {
          case 116: // t: decoller
              bebop.sendTakeoff();
              track=0;
              break;

          case 110: // n: atterrir
             bebop.sendLand();
              track=0;
```

```
            break;

        case 105: // i: avancer
            avancement =  0.1;
            break;

        case 107: // k: reculer
           avancement =  -0.1;
            break;

        case 97: // a: rotation gauche
            rotation =  0.1;
             break;

        case 100: // d:rotation doite
            rotation =  -0.1;
             break;

        case 106: // j:translation gauche
            translation =  0.1;
            break;

        case 108: // l:translation droite
            translation =  -0.1;
            break;

        case 119: // w:monter en altitude
            hauteur =  0.1;
             break;

        case 115: // s:descendre en altitude
           hauteur =  -0.5;
            break;

        case 118: // v: track or stop tracking
            if(track!=1){
            track=1;
            }
            else if(track!=0){
            track=0;
            }
            break;
        default:
            break;
    }
     flushinp(); // on vide le buffer de getch
}
```

Dans le cas où aucune saisie clavier n'est effectuée, nous pouvons nous retrouver dans le cas où le tracking est activé (i.e. `track=1`). On affiche donc en boucle la position du barycentre et le nombre de pixel blanc à l'écran.

```
else  // a key is not pressed
{
   if(track!=0)
   {
       printf("couleur X %d\n\r",couleur_x);
       printf("couleur y %d\n\r" ,couleur_y);
       printf("taille  %f\n\r",taille_h);
```

Les conditions de déplacement sont ensuite définies. Dans le cas où le curseur serait au-delà d'un seuil défini pour les positions x max et min, et y max et min, alors le drone agira pour qu'il se retrouve dans des conditions hors de ces seuils. Il exécutera ce type de comportement pour les quatre cas. (i.e. curseur trop bas, trop haut, trop à gauche et trop à droite.) Pour les cas où la cible serait trop proche ou trop éloignée, le drone se réfère au nombre de pixels blancs à l'image. S'il est trop important, il recule, s'il se trouve en dessous d'un seuil bas, il avancera. Et si la cible est beaucoup trop loin où qu'aucun pixel n'est présent sur l'image, alors le drone ne bouge pas.

```
//calcul vitesse de deplacement en hauteur
if(couleur_y<y_min){
    vitesse_y = 0.1;
    printf("monter \n\r");
}else if(couleur_y>y_max){
    vitesse_y = -0.1;
    printf("descendre \n\r");
}else{
    vitesse_y = 0;
    printf("altitude OK \n\r");
}

printf("changement d'altitude %f\n\r",vitesse_y);
hauteur = vitesse_y;

//calcul vitesse de deplacement en rotation
if(couleur_x<x_min){
    vitesse_x = 0.1*((x_min-couleur_x)/50);
    printf("tourner gauche \n\r");
}else if(x_max<couleur_x){
    vitesse_x = -0.1*((couleur_x-x_max)/50);
    printf("tourner droite \n\r");
}else{
    vitesse_x = 0;
    printf("ne pas tourner \n\r");
}

printf("vitesse de rotation %f\n\r",vitesse_x);
rotation = vitesse_x;

//calcul vitesse de deplacement avant/arrière
        if((taille_h>2500)&&(taille_h<10000)){
        printf("rester\n\r");
        avancement=0;
}
        if(taille_h<2500){
    printf("avancer\n\r");
    avancement=0.1;
}
        if(taille_h>10000){
    printf("reculer\n\r");
    avancement=-0.1;
}
        if(taille_h<100){
// a key is pressed             avancement=0;
                printf("Cilbe perdu\n\r");
}
printf("vitesse avancement %f\n\r",avancement);
```

```
            }
            printf("\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r");
        }
        bebop.setCommand(avancement, translation, hauteur, rotation);
        avancement = 0.00, translation = 0.00, hauteur = 0.00, rotation = 0.00;
        ros::spinOnce();
        loop_rate.sleep();
    }
    endwin();
}
```

CONCLUSION