Manual de usuario del Módulo de Visión
Proyecto de grado Visión de Robots

Gonzalo Gismero
http://www.fing.edu.uy/~pgvisrob - gegismero@gmail.com

Tutores: Facundo Benavides, Gonzalo Tejera, Serrana Casella
InCo - FIng- UDELAR
Montevideo - Uruguay

6 de septiembre de 2012

Resumen

El presente documento describe el uso del módulo de Visión desarrollado durante el proyecto de grado Visión de Robots, así como la localización y valor por defecto de los parámetros de configuración.

Índice

1. Introducción

1.1. Objetivo

El objetivo del documento es brindar al usuario las especificaciones de las funciones que debe invocar para utilizar el módulo de visión y los pasos previos que deben realizarse, así como la explicación y localización de los parámetros calibrados que deban modificarse en los casos que amerite.

Se explica además, el uso del componente Filtro de Zigbee, utilizado para envío y recepción de mensajes a través de Zigbee.

1.2. Arquitectura del módulo de visión

El módulo desarrollado está constituido por tres capas:

- Capa de acceso a HaViMo

  Encargada del acceso al hardware de cámara.

- Capa de visión

  Encargada de realizar el proceso de Visión.

- Capa de comunicación

  Encargada de la comunicación entre los robots, informando de los objetos identificados localmente.

La figura 1 muestra las capas del módulo de visión y su interacción con los demás componentes del sistema. Una explicación más detallada de la arquitectura y componentes del sistema puede ser encontrada en [1].

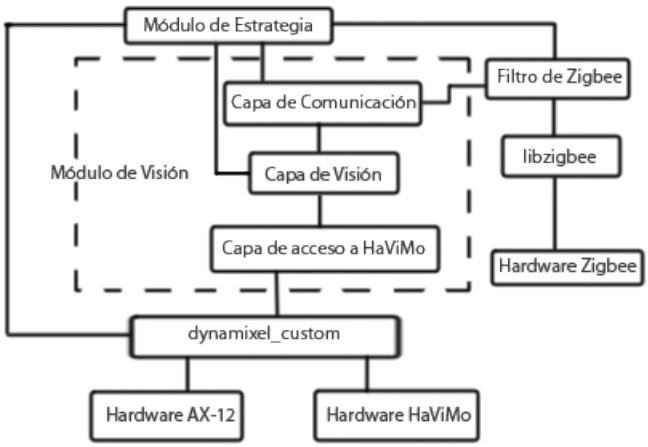

Figura 1: Arquitectura del módulo de Visión

1.3. Recomendaciones de uso

El módulo desarrollado fue creado específicamente para el ambiente de fútbol de robots, para la liga humanoide, *kid-size* de *RoboCup Soccer* [2].

El módulo podrá ser utilizado mientras se respeten las restricciones de forma de los objetos y se cuente con colores diferenciables entre ellos.

1.3.1. Montaje de hardware de cámara

El montaje de la cámara debe tener el cuidado de proveer un amplio ángulo de barrido vertical. Se alienta de esta manera la construcción de un cuello móvil, que permita ver objetos tanto en la base del propio robot como a la altura del horizonte.

Se debe tener especial cuidado de no tapar objetos que se encuentren cerca de la base del robot, por ejemplo, con las rodillas. Esto puede ocasionar que la distancia obtenida para una pelota a los pies del robot, sea mayor que la verdadera. Se recomienda realizar pruebas previo al montaje, cerciorándose que, para esta configuración, las imágenes obtenidas contienen más de la mitad de la pelota.

Por último, un cuello móvil puede permitir reducir el ruido del módulo, provocado por el movimiento del robot al caminar. Se alienta además la implementación de un componente que nivele la cámara al ángulo deseado, según la posición del robot en la caminata.

1.3.2. Configuración de Zigbee

El dispositivo Zigbee utilizado debe cumplir los siguientes requerimientos:

- Utilizar modo *Broadcast*.

  Por defecto, los dispositivos Zig110 se encuentran pareados con su correspondiente Zig100, permitiendo únicamente comunicación entre los mismos. [3]contiene información sobre cómo cambiar el modo de Zig110 y Zig100.

- Encontrarse en el mismo canal de comunicación.

  Los dispositivos Zigbee de BIOLOID cuentan con cuatro canales de comunicación. Por defecto, los modelos Zig110 se encuentran en un canal de comunicación distinto a Zig100. Si se desea interacción entre los distintos dispositivos, es necesaria la modificación del canal utilizado, siguiendo las especificaciones descritas en [3].

1.3.3. Uso del módulo de visión

Como recomendaciones a usuarios del módulo de visión, se alienta a:

- Mantener historial de datos, de manera de no perder información de visión en caídas.

- Utilizar el promedio de la posición encontrada en varias iteraciones para los objetos que presenten alto grado de error.

- Desarrollar un componente que obtenga de manera exacta la orientación de la cámara, basado en la posición de los motores del robot.
- Generar ambiente con luz dispersa, evitando sombras en objetos. Esto es posible utilizando focos en varias posiciones, obteniendo colores mas suaves y parejos en los objetos, facilitando la calibración, reduciendo el ruido en la segmentación y mejorando la identificación de objetos.
- En caso de contar con cuello móvil, se recomienda enfocar la cámara a la base de los objetos de interés, mejorando el cálculo del modelo vectorial y reduciendo ruido de fondo.

2. Acceso a resultados del módulo de visión

El módulo de visión local tiene la responsabilidad de, a partir del resultado del procesamiento de HaViMo, identificar los objetos presentes en el campo, y obtener para ellos la distancia y ángulo al agente[1].

2.1. Resultado de la capa de Visión

El resultado del procesamiento local del módulo puede ser accedido en la variable:

```
struct vision_obj* objects;
```

presente en el archivo **vision.h**.

La variable *objects* representa un arreglo de objetos de tipo *struct vision_obj*, de tamaño variable.

El tamaño del arreglo *objects* puede ser accedido en la variable:

```
unsigned short int obj_size;
```

presente también en el archivo **vision.h**.

2.2. Resultado de la Capa de comunicación de visión

El resultado de la comunicación de la visión con los demás agentes del campo puede ser accedido en la variable:

```
struct vision_obj** allied_objects;
```

presente en el archivo **vision_com.h**.

La variable *allied_objects* representa un arreglo de arreglos de objetos de visión, correspondiéndose con la visión local de los demás agentes. El tamaño de *allied_objects* es determinado por la constante *CANT_ALLIES* definida en **vision_com.h** (ver sección 6.3.2).

El tamaño de cada uno de los arreglos en *allied_objects*, es definido por la variable:

```
unsigned short int* allied_obj_size;
```

que contiene en cada índice *i*, el tamaño para cada arreglo *allied_objects*[*i*].

2.3. Objeto retornado

El tipo de objeto retornado por el módulo de visión se encuentra en **vision.h**, definido como:

```
struct vision_obj {
  unsigned short int type;
  int x;
  unsigned int y;
  unsigned int z;
  unsigned short int likeness_level;
};
```

Donde:

type define el tipo de objeto detectado.

x es el ángulo desde el frente del robot, al objeto detectado, en grados. El 0 se encuentra hacia el frente, creciendo en sentido antihorario.

y es la distancia desde la base del robot, al objeto detectado, en centímetros.

z se encuentra actualmente sin uso, mantenido por extensibilidad.

likeness_level es el nivel de semejanza entre el conjunto de *blobs* y el modelo interno que se usa para identificar el objeto. Abarca los valores de 0 (menor semejanza) a *MAX_LIKENESS_LEVEL* (mayor semejanza).

Los tipos de objetos retornados se encuentran definidos en **vision.h**, y se componen de:

- BALL: Representa la pelota del campo.
- PARTNER: Representa los aliados del agente en el campo.
- ENEMY: Representa los rivales del agente en el campo.
- RIGHT_LANDMARK: *Landmark* derecho de la cancha.
- LEFT_LANDMARK: *Landmark* izquierdo de la cancha.
- MY_GOAL: Representa el arco propio.
- OPP_GOAL: Representa el arco rival.
- MY_LATERAL_POST: Poste lateral del arco propio. Visto únicamente cuando no se retorna un objeto MY_GOAL.
- OPP_LATERAL_POST: Poste lateral del arco oponente. Visto únicamente cuando no se retorna un objeto OPP_GOAL.

Cabe destacar que los objetos RIGHT_LANDMARK y LEFT_LANDMARK no son transmitidos por la Capa de comunicación de visión, ya que son de interés únicamente para la localización propia.

3. Funciones accesibles del módulo de visión

3.1. Funciones de usuario en Capa de visión

La Capa de visión presenta funciones que pueden ser accedidas directamente por componentes superiores. Esto permite ejecutar la visión local, sin utilizar comunicación. Las siguientes funciones se encuentran definidas en **vision.h**.

3.1.1. Inicializar la Capa de visión

```
int vision_initialize(int HaViMo_dxl_ID, int baudnum, unsigned int team_color);
```

HaViMo_dxl_ID: identificador dynamixel de HaViMo.

baudnum: *baudnum* con que se inicializará la librería dynamixel o custom_dynamixel.

team_color: color del equipo aliado, debe ser CYAN_COLOR o MAGENTA_COLOR, incluidos en **iHaViMo.h**. Considera que el color no escogido es de los rivales.

Retorna: 1 en caso de que la inicialización haya sido exitosa, 0 en caso contrario.

Crea las estructuras que utiliza la capa de Visión e inferiores, inicializando el reconocimiento de objetos, la librería de comunicación con dispositivos dynamixel y la capa de acceso a HaViMo, invocando el procesamiento de *frames*.

3.1.2. Finalizar la Capa de Visión

```
void vision_terminate(void);
```

Finaliza la Capa de visión y el reconocimiento de objetos, liberando la memoria reservada del módulo. Finaliza también la capa de acceso a HaViMo y la librería dynamixel.

3.1.3. Establecer ángulo de pan y tilt en radianes

```
void vision_setPanAndTilt(double pan, double tilt);
```

pan: ángulo de *pan* de la cámara, en radianes.

tilt: ángulo de *tilt* de la cámara, en radianes.

Establece los ángulos de *pan* y *tilt* de la cámara en radianes.

El ángulo de *pan* se corresponde con el ángulo de la cámara contra el plano del suelo, positivo hacia abajo, negativo hacia arriba. El ángulo de *tilt* se corresponde con el ángulo de la cámara contra el frente del robot, antihorario. Ver figura 2.

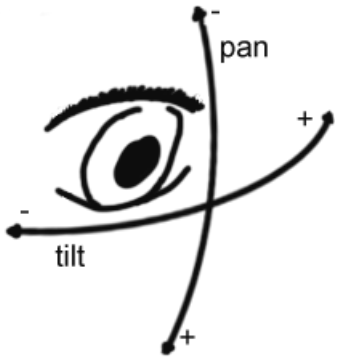

Figura 2: Ángulos de pan y tilt

3.1.4. Establecer ángulo de pan y tilt en grados

```
void vision_setPanAndTiltInDegrees(int pan, int tilt);
```

pan: ángulo de *pan* de la cámara, en grados.

tilt: ángulo de *tilt* de la cámara, en grados.

Establece los ángulos de *pan* y *tilt* de la cámara en grados.

El ángulo de *pan* se corresponde con el ángulo de la cámara contra el plano del suelo, positivo hacia abajo, negativo hacia arriba. El ángulo de *tilt* se corresponde con el ángulo de la cámara contra el frente del robot, antihorario. Ver figura 2.

3.1.5. Establecer altura del robot

```
void vision_setRobotHeight(double height);
```

height: altura del robot, en centímetros.

Establece la altura del robot.

Debe ser invocada antes de procesar cada imagen. En caso de que no se especifique ninguna altura, se utiliza *DEFAULT_ROBOT_HEIGHT*, definido en **vision.h**.

3.1.6. Procesar imagen

```
unsigned short int vision_checkVision(unsigned short int force_frame);
```

force_frame: con un valor mayor a 0 ejecuta la función en modo bloqueante, retornando únicamente cuando se haya procesado un nuevo *frame*. 0 para que no sea bloqueante.

Retorna: 1 si se procesó un nuevo *frame*, 0 en caso contrario.

Verifica si HaViMo procesó un *frame* e invoca nuevamente la captura y procesamiento sobre HaViMo.

En caso de que el procesamiento de un *frame* esté disponible, se realiza el proceso de visión identificando los objetos en la imagen. El resultado puede ser accedido en el arreglo global *objects*, que tiene tamaño *obj_size*, ambos definidos en **vision.h** (ver sección 2.1).

3.2. Funciones de usuario en Capa de comunicación

Las funciones accesibles desde la Capa de comunicación involucran la comunicación entre agentes además de la ejecución del proceso de visión. Las siguientes funciones se encuentran en **vision_com.h**.

3.2.1. Inicializar la Capa de comunicación

```
void vision_com_initialize(unsigned short int ag_id, unsigned int cant_frames_for_broadcast, unsigned short int min_likeness_level, unsigned int max_parsed_messages, unsigned int time_to_live);
```

ag_id: identificador del agente. Comienza en 0, hasta *CANT_ALLIES* - 1.

cant_frames_for_broadcast: cantidad de *frames* que deben ser procesados por la Capa de visión para que se distribuya la visión local a los demás agentes. Se recomienda el valor *DEFAULT_CANT_FRAMES_FOR_BROADCAST*, definido en **vision_com.h**.

min_likeness_level: nivel de semejanza mínimo para que un objeto sea transmitido. De 0 a *MAX_LIKENESS_LEVEL*. Se recomienda el valor *DEFAULT_LIKENESS_LEVEL_TO_SEND*, definido en **vision_com.h**.

max_parsed_messages: cantidad de mensajes de visión recibidos que se deben analizar como máximo en cada invocación. Se recomienda el valor *DEFAULT_PROCESSED_MSG_PER_ITERATION*, definido en **vision_com.h**.

time_to_live: cantidad de iteraciones que serán válidos los mensajes recibidos de aliados. Se recomienda el valor *DEFAULT_TIME_TO_LIVE*, definido en **vision_com.h**.

Inicializa las estructuras necesarias de la Capa de comunicación.

No se inicializa la Capa de visión.

3.2.2. Finalizar la Capa de comunicación

```
void vision_com_terminate(void);
```

Finaliza la comunicación de visión y libera la memoria dinámica utilizada por dicha capa.

No se finaliza la Capa de visión.

3.2.3. Analizar mensajes

```
unsigned short int vision_com_parseMessages(unsigned short int force_vision);
```

force_vision: especifica si se debe forzar la obtención del *frame* en la Capa de visión.

Retorna:

NO_VISION_NO_MSG Si no se identificó ningún objeto y no se recibió ningún mensaje.

VISION_NO_MSG Si se identificaron objetos y no se recibieron mensajes.

NO_VISION_AND_MSG Si no se identificó ningún objeto y se recibieron mensajes.

VISION_AND_MSG Si se identificaron objetos y se recibieron mensajes.

Recorre los mensajes recibidos de visión del componente ZigbeeVisionFilter, almacenando los datos recibidos. Serán analizados hasta *max_parsed_messages* (ver sección 3.2.1).

Realiza además la invocación a la Capa de visión para procesar el siguiente *frame*, distribuyendo el resultado, cuando se hayan procesado *cant_frames_for_broadcast frames* (ver sección 3.2.1).

3.2.4. Actualizar los objetos recibidos de aliados

```
void vision_com_updateObjects(void);
```

Analiza los datos recibidos de los demás aliados, y los presenta a los componentes superiores.

El resultado de la Capa de visión puede ser accedido en el arreglo global *objects*, de tamaño *obj_size*, definidos en **vision.h** (ver sección 2.1).

El resultado de la Capa de comunicación puede ser accedido en el arreglo global *allied_objects*, de tamaño *CANT_ALLIES* (ver sección 2.2).

3.3. Funciones de usuario en Filtro de Zigbee

El Filtro de Zigbee permite utilizar la tecnología Zigbee de manera transparente, evitando colisiones con mensajes del módulo de visión. Para ello, los mensajes enviados por el usuario no pueden finalizar con el parámetro *ZIGBEE_AUTHENTICATION_HEADER* (ver sección 6.4.1). Las siguientes funciones se encuentran definidas en **zigbeeVisionFilter.h**.

3.3.1. Inicialización del Filtro de Zigbee

```
void vision_zigFilter_initalize(int dev_index, int vision_buffer_length, int msg_buffer_length);
```

dev_index: parámetro *dev_index* de libzigbee[4].

vision_buffer_length: largo del *buffer* para mensajes del módulo de visión.

msg_buffer_length: largo del *buffer* para mensajes externos al módulo de visión.

Inicializa el Filtro de Zigbee, y la librería libzigbee de BIOLOID.

3.3.2. Finalización del Filtro de Zigbee

```
void vision_zigFilter_terminate(void);
```

Finaliza el Filtro de Zigbee y la librería libzigbee, liberando la memoria reservada para los *buffers*.

3.3.3. Envío de mensaje

```
int vision_zigFilter_sendMsg(int msg);
```

msg: mensaje a enviar.

Retorna: 1 si el mensaje es enviado exitosamente, 0 sino.

Envía un mensaje a través de Zigbee.

3.3.4. Corroborar si hay un mensaje

```
int vision_zigFilter_hasNextMsgBuffer(void);
```

Retorna: 1 si el *buffer* de mensajes no esta vacío.

Verifica si hay un mensaje en el *buffer* de mensajes.

3.3.5. Obtener mensaje

```
int vision_zigFilter_getNextMsg(void);
```

Retorna: el siguiente mensaje en el *buffer* de mensajes.

Obtiene el siguiente mensaje del *buffer* de mensajes y lo desencola.

3.3.6. Verificar si el *buffer* se encuentra lleno

```
int vision_zigFilter_isFullMsgBuffer(void);
```

Retorna 1 si el *buffer* de mensajes normales esta lleno.

Verifica si el *buffer* de mensajes se encuentra lleno.

4. Etapa de calibración de HaViMo

Previo al uso del módulo de visión, es necesario que el hardware de cámara sea calibrado por el programa del proveedor. De esta manera se asigna a conjuntos de píxeles, regiones preestablecidas.

4.1. Calibración de regiones

La correcta ejecución del módulo requiere que el hardware de cámara sea calibrado, utilizando el software del proveedor[5], según las siguientes especificaciones:

1. La región Ball debe ser calibrada para los colores de la pelota.
2. La región My Goal debe ser calibrada con los colores del arco propio.
3. La región Opp Goal debe ser calibrada con los colores del arco rival.
4. La región Robot debe ser calibrada con los colores del torso del robot.
5. La región Cyan debe ser calibrada con uno de los colores de los equipos.
6. La región Magenta debe ser calibrada con el color de equipo opuesto escogido para la región Cyan.

4.2. Recomendaciones sobre calibración

Se recomienda al momento de calibrar:

- Calibrar la cámara frente a variaciones de luz del ambiente.
- La calibración debe abarcar la totalidad del objeto, en especial la región inferior de los mismos. De otra manera se generarán errores en la distancia obtenida a los objetos.
- Se debe realizar la calibración desde varios puntos de vista de los objetos.
- Para evitar situaciones de ruido donde se detectan objetos fuera de la cancha se recomienda durante la calibración, quitar las secciones asignadas que no interesan utilizando la región Unknown, luego de haber calibrado el objeto de interés.

4.3. Calibración sin uso de alimentación externa

HaViMo en su versión 1.5 fue desarrollado para el controlador CM-5, por lo que para ser calibrado utilizando el controlador CM-510 se deben realizar los siguientes pasos de manera de proveer alimentación a la cámara:

1. Conectar la cámara al controlador CM-510 a través del conector TTL.
2. Conectar el controlador CM-510 al adaptador USB2dynamixel a través del conector TTL.

3. Cambiar el adaptador USB2dynamixel para utilizar la salida TTL.

4. Conectar el adaptador al PC.

Una vez conectada la cámara al PC, se realiza la calibración utilizando el programa de calibración USB.

5. Etapa en ejecución

Una vez realizada la calibración, el módulo es capaz de identificar los distintos objetos de la cancha, y obtener la distancia y ángulo respecto al frente del robot.

Se distinguen dos modos de funcionamiento:

- Visión sin comunicación: Donde los objetos obtenidos se corresponden únicamente con los identificados por el módulo en el agente.
- Visión con comunicación: Donde además de la visión local, se tiene acceso a los objetos identificados por los demás agentes aliados en el campo.

5.1. Visión sin comunicación

La visión sin comunicación realiza el proceso de visión sin interactuar con los demás agente y consiste en el uso del módulo sin la Capa de comunicación, accediendo directamente a las funcionalidades de la Capa de Visión.

5.1.1. Requerimientos de hardware

1. El agente debe tener conectada la cámara HaViMo 1.5 o compatible, previamente calibrada según sección 4.1.

5.1.2. Requerimiento de software

1. Se debe compilar utilizando las librerías:
   - libdynamixel_custom.a
   - libhavimo.a
   - libvision.a
2. Se debe importar los siguientes archivos de cabezal:
   - **vision.h**

5.1.3. Ejemplo

```
#include "vision.h"
#define HAVIMO_DXL_ID                    100

int main(void) {

...

  //Incializando dynamixel y localizacion
  //Los aliados estan de cyan, los rivales de magenta
  //Cambiar por MAGENTA_COLOR en caso de usar las otras camisetas
```

```
  vision_initialize(HAVIMO_DXL_ID, 1, CYAN_COLOR);

  while(1) {
    //Los componentes superiores deben obtener el pan, tilt
    //y altura de la cámara
    double pan = getCameraPan();
    double tilt = getCameraTilt();
    double height = getCameraHeight();

    vision_setPanAndTilt(pan, tilt);
    vision_setRobotHeight(height);

    if(vision_checkVision(0)) {
      for(int i = 0; i < obj_size; i++) {
        switch(objects[i].type) {
          case PARTNER:...;
          case ENEMY:...;
          case BALL:...;
          case RIGHT_LANDMARK:...;
          case LEFT_LANDMARK:...;
          case MY_GOAL:...;
          case OPP_GOAL:...;
          case MY_LATERAL_POST:...;
          case OPP_LATERAL_POST:...;
        }
      }
    }
  }

  vision_terminate();
...

}
```

5.2. Visión con comunicación

La visión con comunicación realiza el proceso de visión, comunicándose con los demás agente del campo, distribuyendo sus resultados locales y recibiendo la visión de otros agentes.

5.2.1. Requerimientos de hardware

1. El agente debe tener conectada la cámara HaViMo 1.5 o compatible, previamente calibrada según sección 4.1.

2. El agente debe tener conectado hardware de Zigbee compatible con la librería

libzigbee[1].

5.2.2. Requerimiento de software

1. Se debe compilar utilizando las librerías:
   - libdynamixel_custom.a
   - libzigbee.a
   - libzigbee_vision_filter.a
   - libhavimo.a
   - libvision.a
   - libvision_com.a
2. Se debe importar los siguientes archivos de cabezal:
   - **vision.h**
   - **vision_com.h**
   - **zigbeeVisionFilter.h**

5.2.3. Ejemplo

```
#include "vision.h"
#include "vision_com.h"
#include "zigbeeVisionFilter.h"

#define HAVIMO_DXL_ID       100

//Modificar para cada agente, desde 0 a CANT_ALLIES - 1
#define AG_ID               0

int main(void) {

...

  //Inicializando filtro de zigbee.
  //La estrategia debe enviar los mensajes de Zigbee a partir de
  //esta interfaz (zigbeeVisionFilter.h)
  vision_zigFilter_initalize(0, 100, 100);

  //Incializando dynamixel y localizacion. los aliados estan de cyan,
  //los rivales de magenta
  //Cambiar por MAGENTA_COLOR en caso de usar las otras camisetas
```

[1]Para BIOLOID, existe Zig110.

```
vision_initialize(HAVIMO_DXL_ID, 1, CYAN_COLOR);
vision_com_initialize(AG_ID, CANT_FRAMES_FOR_BROADCAST,
      LIKENESS_LEVEL_TO_SEND, MSG_FOR_ITERATION, TTL);

while(1) {
  //Los componentes superiores deben obtener el pan, tilt
  //y altura de la cámara
  double pan = getCameraPan();
  double tilt = getCameraTilt();
  double height = getCameraHeight();

  vision_setPanAndTilt(pan, tilt);
  vision_setRobotHeight(height);

  result = vision_com_parseMessages(0);

  if(result == VISION_NO_MSG || result == VISION_AND_MSG) {
    //Datos obtenidos localmente
    for(i = 0; i < obj_size; i++) {
      switch(objects[i].type) {
        case PARTNER:...;
        case ENEMY:...;
        case BALL:...;
        case RIGHT_LANDMARK:...;
        case LEFT_LANDMARK:...;
        case MY_GOAL:...;
        case OPP_GOAL:...;
        case MY_LATERAL_POST:...;
        case OPP_LATERAL_POST:...;
      }
    }
  }

  if(result == NO_VISION_AND_MSG || result == VISION_AND_MSG) {
    //Actualizar según mensajes recibidos
    vision_com_updateObjects();

    //Datos recibidos de comunicación de visión
    for(j = 0; j < CANT_ALLIES; j++) {
      for(i = 0; i < allied_obj_size[j]; i++) {
        switch(allied_objects[j][i].type) {
          case PARTNER:...;
          case ENEMY:...;
          case BALL:...;
          case MY_GOAL:...;
          case OPP_GOAL:...;
```

```
                case MY_LATERAL_POST: ...;
                case OPP_LATERAL_POST: ...;
            }
          }
        }
      }
    }

    ...

    vision_com_terminate();
    vision_terminate();
    vision_zigFilter_terminate();

}
```

5.3. Filtro de Zigbee

El componente Filtro de Zigbee abstrae el uso de Zigbee y proporciona 2 *buffers* distintos, el primero para los mensajes enviados por los componentes de la visión y el segundo para ser utilizado por módulos externos a la visión. Si se utiliza la Capa de comunicación del módulo de visión, es necesario utilizar el Filtro de Zigbee en lugar acceder directamente a la librería libzigbee provista por ROBOTIS.

5.3.1. Requerimientos de hardware

1. El agente debe tener conectado hardware de Zigbee compatible con la librería libzigbee.

5.3.2. Requerimiento de software

1. Se debe compilar utilizando las librerías:
   - libzigbee.a
   - libzigbee_vision_filter.a
2. Se debe importar los siguientes archivos de cabezal:
   - **zigbeeVisionFilter.h**

5.3.3. Ejemplo

```
#include "zigbeeVisionFilter.h"

#define BUFFER_LENGTH           100
#define VISION_BUFFER_LENGTH    100
int main(void) {
```

```
  vision_zigFilter_initalize(0, VISION_BUFFER_LENGTH, BUFFER_LENGTH);

  ...

  //Envio de mensaje
  int msg = ...;

  if(vision_zigFilter_sendMsg(msg)) {
    //Mensje enviado correctamente
  }

  //Recepción de mensaje
  if(vision_zigFilter_hasNextMsgBuffer()) {
    msg = vision_zigFilter_getNextMsg();
  }

  ...;

  vision_zigFilter_terminate();
}
```

6. Parámetros calibrados

Puede ser necesario realizar la recalibración de los parámetros internos del módulo. Casos como:

- Cambios en medidas de objetos de la cancha
- Cambios en restricciones de diseño de robots
- Cambios en resolución de la cámara
- Cambios en la cantidad de aliados en la cancha

requieren la modificación de los mismos para el correcto funcionamiento del módulo.

Además de cambiar el valor de los parámetros en cuestión, se deberá realizar una recompilación del módulo, utilizando los archivos *makefile* puestos a disposición junto con el código fuente.

A continuación se describen los parámetros configurables, según su localización en el código fuente.

6.1. Capa de acceso a HaViMo

6.1.1. iHaViMo.h

Cantidad de regiones retornadas por HaViMo

CANT_REGIONS 15

Índice de región inválida

INVALID_REGION 0

Tipo de región detectada por HaViMo

UNKOWN_REGION 0

BALL_REGION 1

FIELD_REGION 2

MY_GOAL_REGION 3

OPP_GOAL_REGION 4

ROBOT_REGION 5

CYAN_REGION 6

MAGENTA_REGION 7

Resolución de HaViMo

IMAGE_WIDTH 160

IMAGE_HEIGHT 120

Distancia focal para los métodos de modelo vectorial y de proporciones respectivamente. El valor se corresponde con la distancia al plano de la imagen en píxeles, obtenido a partir de distancias conocidas (d(cm) * ancho(px) / ancho(cm)).

DISTANCIA_FOCAL_HAVIMO 164

DFH_FOR_PROPORTION 179

6.2. Capa de Visión

6.2.1. clasification.h

Mínimo largo y ancho de píxeles válidos

MIN_PX_SIZE_FOR_BLOB 2

Milisegundos de espera entre invocaciones sobre HaViMo, al utilizar visión bloqueante

FORCE_FRAME_MS_DELAY 10

Umbral de distancia en píxeles entre *blobs* para agrupación en eje Y

THRESHOLD_Y 7

Umbral de distancia en píxeles entre *blobs* para agrupación en eje X

THRESHOLD_X 7

Umbral de distancia en píxeles entre *blobs* de arco para agrupación en eje Y

GOAL_THRESHOLD_Y 15

Umbral de distancia en píxeles entre *blobs* de arco para agrupación en eje X

GOAL_THRESHOLD_X 15

6.2.2. objRecognition.h

Umbral de proximidad de pelota al margen inferior en píxeles

BOTTOM_MARGEN_PROXIMITY IMAGE_HEIGHT - 3

Mínimo coeficiente de redondez para la pelota

MIN_BALL_ROUNDNESS 0.6

Área mínima de los postes de los arcos, en píxeles al cuadrado.

MIN_GOAL_POST_AREA 25

Umbral de diferencia de ancho entre *blobs* de robots de costado.

TWO_BLOB_ROBOT_WIDTH_THRESHOLD 5

Umbral de proximidad de postes al margen superior en píxeles

LATERAL_POST_TOP_MARGEN_PROXIMITY 10

Ancho mínimo en píxeles para objetos de tipo poste lateral

LATERAL_POST_MIN_WIDTH_PX 15

Ancho mínimo para *blobs* de postes de objeto de tipo arco

MIN_POST_BLOB_WIDTH 10

Separación en X para apilar un *blob* sobre otro al detectar postes

POST_MIX_BLOBS_THRESHOLD 10

6.2.3. vision.h

Tipo de objetos retornados

MYSELF 0

BALL 1

PARTNER 2

ENEMY 3

RIGHT_LANDMARK 4

LEFT_LANDMARK 5

MY_GOAL 6

OPP_GOAL 7

MY_LATERAL_POST 8

OPP_LATERAL_POST 9

Máximo nivel de semejanza en la identificación

MAX_LIKENESS_LEVEL 15

Valores sugeridos para inicialización del modulo de Visión

DEFAULT_HAVIMO_ID 100

DEFAULT_BAUDNUM 1

Altura por defecto del piso a la cámara

DEFAULT_ROBOT_HEIGHT 29.5

Ancho del objeto *landmark* en centímetros

LANDMARK_WIDTH 15.0

Tamaño mínimo de *blob* para que el error del método de proporciones sea despreciable

TWO_BLOB_LANDMARK_MIN_WIDTH_THRESHOLD 26

Umbral de distancia para comparación entre métodos, para elección de tipo en de *landmark,* en centímetros

DISTANCE_METHOD_THRESHOLD 5

Ancho de travesaño divido la altura (170/10), en centímetros

CROSSBAR_PROPORTION 17

Ancho del arco divido la altura (170/90), en centímetros

GOAL_PROPORTION 1.88

Altura del arco en centímetros, solo hasta el inicio del travesaño

GOAL_HEIGHT 80

Altura del travesaño en centímetros

CROSSBAR_HEIGHT 10

Distancia máxima para la que se retornan objetos, en centímetros

MAX_DISTANCE 800

6.3. Capa de Comunicación

6.3.1. zigAdapter.h

Tipos de objetos del mensaje

OBJ_ALLY 0

OBJ_RIVAL 1

OBJ_BALL 2

OBJ_MY_GOAL 3

OBJ_OPP_GOAL 4

OBJ_MY_LATERAL_POST 5

OBJ_OPP_LATERAL_POST 6

OBJ_ME 7

Parámetros de los objetos en mensajes

PAR_X 0

PAR_Y 1

PAR_Z 2

6.3.2. vision_com.h

Cantidad de aliados

CANT_ALLIES 2

Cantidad máxima de objetos que se pueden recibir

CANT_OBJECTS_FOR_AGENTS 8

Cantidad sugerida de *frames* que deben procesarse antes de distribuir nuevamente la visión local

DEFAULT_CANT_FRAMES_FOR_BROADCAST 16

Cantidad por defecto que se asigna al filtro de objetos para *broadcast*, en caso de que el valor asignado quede fuera de rango

DEFAULT_LIKENESS_LEVEL_TO_SEND 1

Cantidad sugerida de mensajes que se procesan por iteración

DEFAULT_PROCESSED_MSG_PER_ITERATION 30

Cantidad sugerida de iteraciones durante las cuales son válidos los mensajes recibidos

DEFAULT_TIME_TO_LIVE 16

6.4. Filtro de Zigbee

6.4.1. zigbeeVisionFilter.h

Tamaño sugerido para *buffers* de mensajes

DEFAULT_BUFFER_LENGTH 100

Parámetro encolado al final de los mensajes enviados por la Visión; se corresponde con los últimos ZIGBEE_AUTHENTICATION_LENGTH bits del mensaje. Los mensajes de componentes externos no pueden terminar con ZIGBEE_AUTHENTICATION_HEADER.

ZIGBEE_AUTHENTICATION_HEADER 0x0

Largo de *ZIGBEE_AUTHENTICATION_HEADER*

ZIGBEE_AUTHENTICATION_LENGTH 3

Mascara para la autenticación. Debe reflejar a *ZIGBEE_AUTHENTICATION_LENGTH*

ZIGBEE_AUTHENTICATION_MSK 0x0007

7. Problemas frecuentes

El módulo no identifica objetos que distan más de un metro

- Verificar que la calibración siga siendo válida.
- Los objetos deben calibrarse para distintos puntos de vista, tanto cercanos como lejanos.
- El módulo no puede identificar objetos más allá de dos metros y medio.

El ángulo retornado a los objetos no es correcto

- Verificar que las unidades (radianes o grados) de tilt y pan de la cámara se correspondan con el método utilizado para establecerlos.

La distancia a los objetos no es correcta

- Verificar que la cámara esté enfocando la base de los objetos de interés.
- A mayor distancia a los objetos, el error en la distancia aumenta.

El arco no es identificado a menos de un metro

- No es posible identificar el arco a esta distancia. En contraparte, se retornan los postes laterales que conforman el arco.

El programa de calibración no reconoce la cámara

- Verificar que el adaptador USB2Dynamixel se encuentre establecido para el puerto TTL.
- Verificar que el CM-510 se encuentre encendido.
- Verificar que ningún otro programa esté usando el puerto de USB2Dynamixel.

No se reciben mensajes desde el Filtro de Zigbee

- Verificar que los dispositivos Zigbee se encuentren en modo *broadcast*.
- Verificar que el canal de *broadcast* de los dispositivos Zigbee sea el mismo.

8. Programas

Calibrador de HaViMo

Aplicación para calibración de HaViMo.
Disponible en `http://sourceforge.net/projects/havimo/`

WinAVR

Suite de desarrollo y compilador GNU GCC para microprocesadores Atmel AVR.
Disponible en `http://winavr.sourceforge.net/`

Custom Dynamixel

Librería libdynamixel de ROBOTIS modificada para el uso de HaViMo.
Disponible en `http://www.fing.edu.uy/~pgvisrob/`.

Zigbee Filter

Módulo de acceso a Zigbee, utilizado para distinguir autores de mensajes.
Disponible en `http://www.fing.edu.uy/~pgvisrob/`.

Vision SDK

Módulo de Visión, conteniendo librerías necesarias, así como los ejemplos descritos en el manual. Se recomienda tomar como base los archivos makefile de los ejemplos para compilar aplicaciones.
Disponible en `http://www.fing.edu.uy/~pgvisrob/`.

Librerías de ROBOTIS

Contiene archivos cabecera, librerías, fuentes de librerías y código de ejemplo. Las librería libzigbee se encuentra también en Vision SDK.
Disponible en `http://support.robotis.com/en/software/embeded_c/cm510_cm700.htm`.

ROBOPLUS Terminal

Programa terminal de ROBOTIS, utilizado para descargar código al controlador. La aplicación forma parte de la suite de programas de ROBOTIS para BIOLOID y se encuentra disponible en el CD provisto junto con el robot.
Por información sobre cómo descargar código al controlador, visitar: `http://support.robotis.com/en/software/embeded_c/cm510_cm700/programming/bootloader.htm`.

RoboPlus Manager

Utilizado para la administración de dispositivos de BIOLOID. Permite modificar el modo de transmisión de ZIG110 y restaurar el firmware de CM-510. Disponible en el CD provisto junto con el ROBOT.

Por información sobre como modificar el modo de transmisión de ZIG110, visitar: `http://support.robotis.com/en/product/auxdevice/communication/zigbee_manual.htm`.

Referencias

[1] Gonzalo Gismero. Arquitectura del módulo de visión. http://www.fing.edu.uy/˜pgvisrob. 2012.

[2] 2010 RoboCup Soccer Humanoid League. *RoboCup Soccer Humanoid League Rules and Setup for the 2010 competition in Singapore*. RoboCup Soccer Humanoid League, 1 edition, 2010.

[3] ROBOTIS. Zigbee e-manual. http://support.robotis.com/en/product/auxde vice/-communication/zigbee_manual.htm.

[4] ROBOTIS. Roboits e-manual. http://support.robotis.com/en/.

[5] Hamid Mobalegh PhD. student of Freie Universität Berlin. Embedded vision module for bioloid - quick start.