Universitat Autònoma de Barcelona

escola d'enginyeria

5132-1 DETECCIÓN DE PERSONAS DESDE AVIONES NO TRIPULADOS UAV

Memòria del Projecte Fi de Carrera
d'Enginyeria en Informàtica
realitzat per

Jordi Ventayol Marimon

i dirigit per

Antonio Manuel López Peña

Bellaterra, 19 de Juny de 2013

Agradecimientos

En primer lugar dar las gracias a Antonio Manuel López Peña por la dirección y supervisión de este trabajo. Agradecer su atención y disponibilidad a lo largo de estos meses y sus consejos. También agradecer que bajo ninguna obligación me proporcionara herramientas mejores que las mías y su implicación en algunos días festivos.

En segundo lugar agradecer a Jiaolong Xu, por su paciencia y consejos a lo largo del desarrollo del proyecto, y contestarme en un margen de 24h todas y cada una de las dudas que me han surgido, así como contagiarme una gran ilusión en el mundo de detección de personas mediante visión por computador.

En tercer lugar agradecer a Daniel Ponsa por sus consejos en el desarrollo de la aplicación y a todas las personas que han participado en este proyecto, incluidos mis compañeros de Ingeniería Informática que con su experiencia o consejos también me han ayudado a tomar decisiones sobre el proyecto.

Por último gracias a los lectores, que serán pocos pero espero que encuentren un proyecto interesante y que les aporte algo nuevo sobre sus conocimientos en este mundo.

Tabla de contenidos

Capítulo 1:

Introducción:

1.1 Contexto

La búsqueda de víctimas en escenarios donde se ha producido algún tipo de desastre, o la localización de personas perdidas en bosques es uno de los objetivos principales de cualquier operación de salvamiento. En muchas ocasiones, la zona o el desastre es de tal magnitud que acceder a la zona es de gran dificultad para los responsables de salvamiento.

Todo y que se ha avanzado mucho en la búsqueda de víctimas humanas mediante vehículos terrestres las dificultades que presenta normalmente el escenario del desastre hace que aún no se disponga de prototipos lo suficientemente fiables para acceder en dichos escenarios. Pero la incorporación de los vehículos aéreos no tripulados de peso ligero y poco costosos, (Unmanned aerial vehicles; UAVs), ha hecho que los escenarios del desastre sean más accesibles desde el aire y se puedan recoger imágenes en busca de personas.

Uniendo estos vehículos con algoritmos de detección de personas se consigue una potente herramienta para la localización de personas en zonas casi inaccesibles, reduciendo el tiempo de búsqueda e intervención humana.

El objetivo final consiste en poder enviar una flota de UAVs en escenarios donde se ha producido una catástrofe o donde se busque a personas perdidas. Una vez se ha detectado alguna víctima mediante los algoritmos de detección de personas se pueden tomar las medidas adecuadas con intervención humana.

Para ello hay que desarrollar una aplicación que controle los UAVs desde los cuales se recogerán imágenes y se aplicarán los algoritmos de detección de personas, para determinar si la escena que se está grabando contiene personas parcialmente o totalmente visibles. Para facilitar la detección hará falta aplicar y adaptar técnicas usadas en procesamiento de imágenes para conseguir el mayor acierto y el menor error a la hora de decidir.

El objetivo principal de este proyecto será desarrollar una aplicación para pilotar un AR.Drone y obtener imágenes en tiempo real, así como información de los sensores de éste para incorporar los algoritmos de detección de personas ya probados en vehículos terrestres. Para ello se estudiarán los resultados y analizarán los nuevos problemas en este entorno y se propondrán soluciones al respecto.

1.2 Objetivos

Para abordar el objetivo final, es decir, la detección de personas desde el UAV, se ha dividido este objetivo en sub-objetivos más simples. Estos son de la siguiente forma:

1.- Desarrollar una aplicación capaz de comunicarse con el AR.Drone para su pilotaje.

2.- Adaptar la aplicación para poder obtener información de los sensores, desde la cámara para la obtención de imágenes, hasta la velocidad y ángulos de navegación.

3.- Estructurar la gran cantidad de datos que se registran y clasificarlos para que cumpla con los requisitos del algoritmo de detección de personas y su posterior análisis.

4.- Evaluar el algoritmo de detección de personas usado en el vehículo terrestre.

1.3 Planificación del proyecto

La planificación del proyecto abarca desde octubre de 2012 hasta julio de 2013. Durante este periodo las distintas tareas a realizar se han repartido de la siguiente forma:

Id. Tarea	Nombre	Inicio	Final	Horas
T0	Lectura y pequeño estudio de modelos de detección.	01/10/2012	15/01/2013	15 h
T1	Instalación y preparación del entorno de desarrollo.	05/11/2012	22/01/2013	10 h
T2	Primera aplicación para el manejo del AR.Drone.	22/01/2013	15/03/2013	40 h
T3	Adaptación de la aplicación para la recolección de imágenes y información de sensores.	15/03/2103	30/04/2013	60 h
T4	Creación de una Base de Datos, aplicar modelos de detección y estudio de los resultados.	06/05/2013	03/06/2013	30 h
T5	Generación de la memoria y desarrollo de la presentación	27/05/2013	17/06/2013	30 h

1.4 Estructura de la memoria

El contenido de la memoria se distribuye en los siguientes capítulos:

- Capítulo 2, Marco teórico y estado del arte: En este capítulo se introducirán los conceptos básicos y se conocerán las herramientas de desarrollo a lo largo del proyecto.

- Capítulo 3, Desarrollo de la aplicación: Se expondrá cómo se han abordado los objetivos del proyecto, explicando las decisiones tomadas a lo largo de éste y cómo se han solucionado los problemas surgidos, se analizarán los resultados obtenidos así como la solución final.

- Capítulo 4, Conclusiones: Se expondrán las conclusiones del proyecto y las sensaciones personales, así como se valorarán propuestas de futuro.

Capítulo 2:

Marco teórico y estado del arte

2.1 Parrot AR.Drone

En esta sección se conocerá la herramienta de trabajo A.R.Drone, desde su funcionamiento hasta su configuración y programación. Se detallará el método de comunicación con el dispositivo portátil , su arquitectura interna y finalmente cómo funcionan las librerías para programarlo.

2.1.1 Introducción

A.R.Drone es un cuadrotor compuesto mecánicamente por cuatro rotores o hélices en los extremos, conectados a una batería y otros componentes hardware específicos.
Cada par de hélices opuestas gira en el mismo sentido, un par en el sentido de las agujas del reloj y el otro par gira en sentido anti-horario. De esta manera se le proporciona estabilidad y una fuerza vertical para que se eleve.
Como se muestra en la siguiente imagen (Fig.1.1), cada rotor recibe un nombre específico, empezando por la izquierda: Left, Front, Rear y Right.

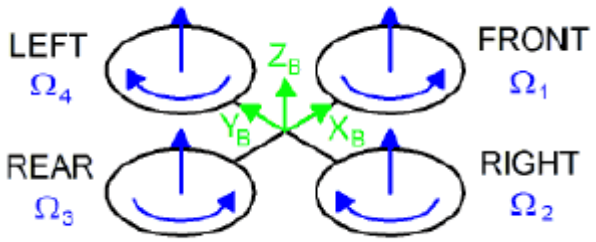


Fig.1.1: Hélices del AR.Drone.

2.1.2 Desplazamiento

Para conseguir los distintos movimientos como pueden ser avanzar, retroceder y girar entre otros, se modifican los valores de los ángulos *pitch*, *roll* y *yaw* del A.R.Drone. Para entender estos ángulos la siguiente figura (Fig.1.2) muestra la dirección que emprende el vehículo al incrementar o decrementar estos ángulos. A su vez, y para modificar estos ángulos, se modifica la velocidad de los rotores individualmente para conseguir los ángulos deseados, e inminentemente el movimiento deseado por el piloto.

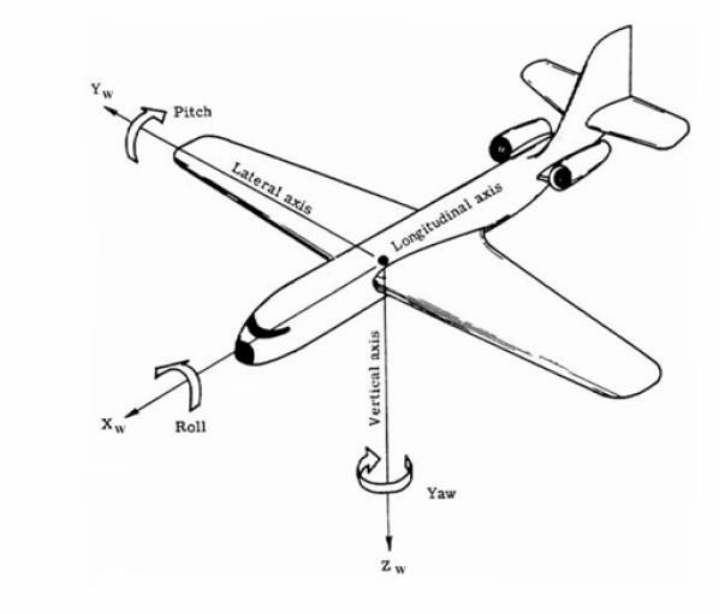


Fig.2: Ángulos de navegación.

- Modificando la velocidad de los rotores Left y Right se consigue mover el A.R.Drone por el campo del ángulo *roll,* permitiendo avanzar y retroceder el vehículo.
- Modificando la velocidad de los rotores Front y Rear se consigue mover por el campo del ángulo *pitch*.
- Finalmente modificando la velocidad de cada par de rotor se consigue mover por el campo del ángulo *yaw*. Esto permite girar a la izquierda o a la derecha.

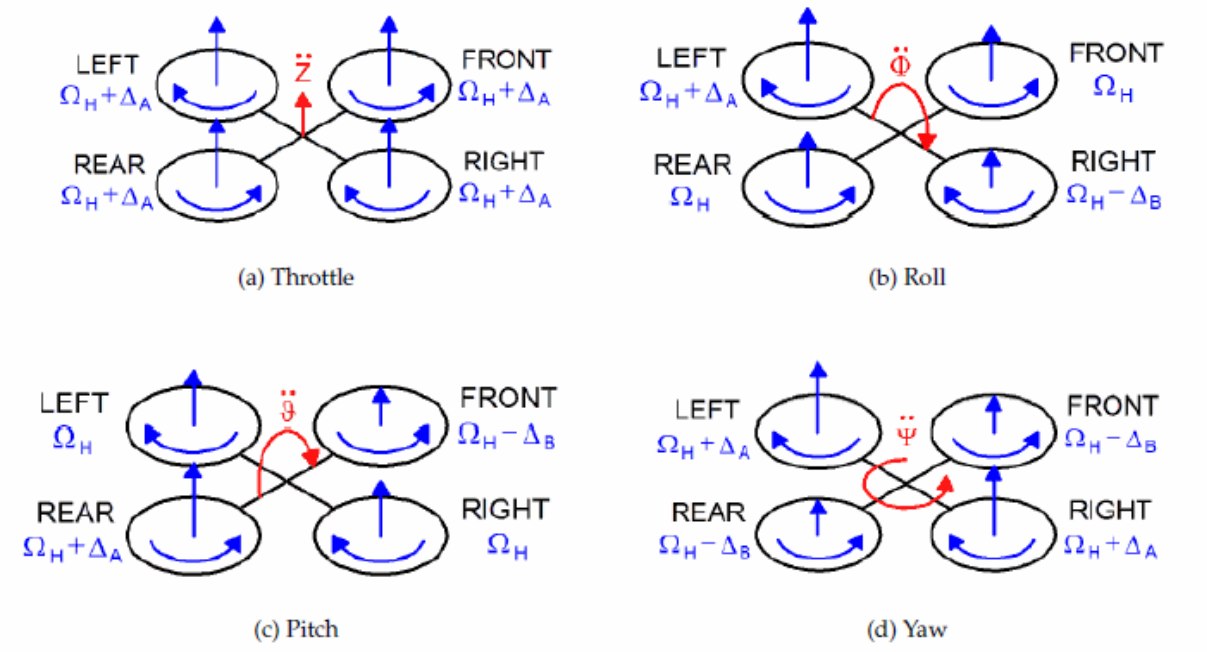


Fig.1.3: Ángulos y velocidad de las hélices [13].

Para compensar entre desplazamiento y seguridad al mismo tiempo hay dos tipos de estructuras para el A.R.Drone, una para interiores y otra para exteriores. La estructura para interior protege las hélices completamente para evitar colisionar con objetos en el escenario protegiéndolo lateralmente, mientras que la estructura para exterior carece de protección lateral ya que las condiciones climatológicas como el viento lo hacen más vulnerable cuanta más superficie disponga.

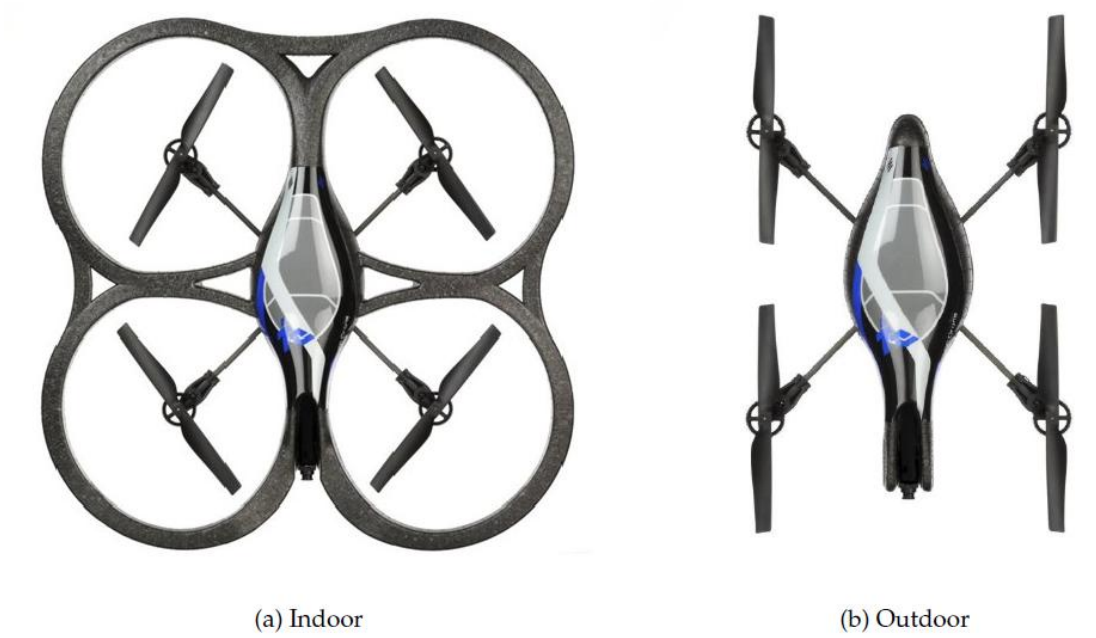


Fig.1.4: Estructuras AR.Drone.

2.1.3 Sensores

El AR.Drone cuenta con varios sensores de movimiento. Todos ellos están situados en la parte inferior del casco central. Contiene una pequeña unidad de medida que proporciona el software para las medidas de pitch, roll y yaw. Estas mediciones proporcionan automáticamente el control para la estabilización del vehículo.
Un telémetro ultrasónico proporciona medidas de altitud para la estabilización automática de la altitud y el control de velocidad vertical.
Finalmente una cámara apuntando hacia el suelo, junto al medidor de velocidad, proporciona medidas de velocidad de desplazamiento automático.

2.1.4 Conexión por red Wifi

El AR.Drone puede ser controlado por cualquier dispositivo que soporte el modo Wifi ad-hoc. Esto significa que es una red inalámbrica descentralizada, que no se basa en una infraestructura existente, tales como un router o un punto de acceso, lo cual hace que la determinación de qué nodos encaminan sus datos se realiza dinámicamente sobre la base de la conectividad de red.

El proceso de conexión consiste en los siguientes pasos:

1.- El AR.Drone crea una red WIFI con una ESSID (Extended Service Set Identifier), que es un nombre incluido en todos los paquetes de una red inalámbrica WIFI para identificarlos como parte de esa red , con el formato ardrone_xxxx y se asigna una dirección IP libre.

2.- El usuario conecta el dispositivo que actúa como cliente a la red con ese ESSID.

3.- El dispositivo que actúa como cliente pide una dirección IP y sus parámetros de configuración al servidor mediante el protocolo DHCP del AR.Drone.

4.- El servidor DHCP del AR.Drone otorga al cliente una dirección IP según la versión:

- La misma dirección IP del AR.Drone más 1 (versiones inferiores a la 1.1.3).
- La misma dirección IP del AR.Drone más un número entre el 1 y el 4 (versiones superiores o iguales a 1.1.3).

5.- El dispositivo que actúa como cliente ya puede mandar paquetes de datos con solicitudes a la IP y puertos acordados.

2.1.5 Comunicación con el AR.Drone

Para controlar el AR.Drone el proceso de comunicación se realiza mediante 3 servicios principales de comunicación.

El control y la configuración del AR.Drone se efectúa mediante el envió de comandos AT en el puerto UDP 5556. La latencia de transmisión de estos comandos es fundamental para el buen control del AR.Drone, ya que se deben enviar de forma regular 30 veces por segundo (generalmente).

La información del AR.Drone, como el estado, la posición, velocidad, velocidad de rotación de las hélices, etc. Es la información llamada "navdata" (información de navegación) que es enviada por el AR.Drone al dispositivo que actúa como cliente en el puerto UDP 5554. Esta información de navegación también incluye la información de detección de etiquetas que por ejemplo se usa en crear juegos de realidad aumentada. Toda esta información también es enviada unas 30 veces por segundo.

Las secuencias de vídeo son enviadas por el AR.Drone al cliente por el puerto 5555. Las imágenes de estas secuencias de vídeo pueden ser descodificadas usando el códec incluído en el SDK (Software Developer Kit).

El cuarto y último canal de comunicación, llamado el puerto de control, se establece en el puerto TCP 5559 para transferir datos críticos, por eso se usa a diferencia del resto de canales un puerto TCP, ya que la pérdida de los otros datos no produce situaciones críticas. Se usa para recibir datos de configuración, y para verificar el correcto envío de la información importante.

2.1.5.1 Comandos AT

Los comandos AT son cadenas de caracteres enviados al AR.Drone para controlar sus acciones. Estas cadenas son generadas por las librerías ARDroneLib y ARDroneTool, proporcionados en el SDK.

Las cadenas de caracteres se codifican como caracteres ASCII de 8 bits, con un carácter de avance de línea (valor de byte 10 (10)), señalado <LF> adelante, como delimitador de línea nueva.
Un comando consiste en los tres caracteres AT * (es decir, tres palabras de 8 bits con valores de 41 (16), 54 (16), 2a (16)) seguido de un nombre de comando, e el signo igual, un número de serie, y opcionalmente una lista de argumentos separados por comas cuyo significado depende del orden.
Un solo paquete UDP puede contener uno o más comandos, separados por saltos de línea (valor de byte 0A (16)). Un comando AT debe residir en un solo paquete UDP, dividirlo en dos o más paquetes UDP no es posible.

AT*PCMD=21625,1,0,0,0,0<LF>**AT***REF=21626,290717696<LF>

Fig.2.1:Ejemplo comando AT.

La longitud máxima total de todos los comando no puede superar los 1.024 caracteres, de lo contrario todo la línea de comandos es rechazada. Este límite está codificado en el software del AR.Drone y no es modificable.
Los comandos AT con formatos incorrectos son ignorados por el AR.Drone. Sin embargo, el cliente debe asegúrese siempre de que envía paquetes UDP con el formato correcto.

La mayoría de los comandos aceptan argumentos, que pueden ser de tres tipos diferentes:

- Un entero con signo, almacenado en la cadena de caracteres del comando con una representación decimal (por ejemplo: el número de secuencia).

- Valor de una cadena de caracteres almacenado entre comillas (por ejemplo: los argumentos de AT * CONFIG).

- Un IEEE-754 (numero de punto flotante de precisión simple), (es decir, un float).

Con el fin de evitar el procesamiento de comandos antiguos, se asocia un número de secuencia al envió de cada comando AT, y se almacena como el primer número después del signo "igual". El AR.Drone no ejecutará ningún comando cuyo número de secuencia sea menor que el último comando AT válido recibido.

Este número de secuencia se pone a 1 dentro del AR.Drone cada vez un cliente se desconecta del puerto UDP (actualmente esta desconexión se hace al no enviar ningún comando durante más de 2 segundos), y cuando se recibe un comando con un número de

orden establecido en 1. Un cliente deberá por lo tanto respetar las siguientes reglas con el fin de ejecutar correctamente los comandos del AR.Drone:

- Enviar siempre 1 como el número de secuencia del primer comando enviado.

- Siempre enviar los comandos AT con un aumento en el número de secuencia. Si varios subprocesos de software envían comandos AT, se puede generar el número de secuencia y el envío de paquetes UDP por una sola función dedicada a proteger esta variable por un mecanismo de exclusión mutua.

AT command	Arguments[1]	Description
AT*REF	input	Takeoff/Landing/Emergency stop command
AT*PCMD	flag, roll, pitch, gaz, yaw	Move the drone
AT*FTRIM	-	Sets the reference for the horizontal plane
AT*CONFIG	key, value	Configuration of the AR.Drone
AT*CONFIG_IDS	session, user, application ids	Identifiers for AT*CONFIG commands
AT*LED	animation, frequency, duration	Set a led animation on the AR.Drone
AT*ANIM	animation, duration	Set a flight animation on the AR.Drone
AT*COMWDG	-	Reset the communication watchdog

Fig.2.2:Resumen comandos AT.

2.1.6 Arquitectura de capas y librería

Un esquema para entender el funcionamiento y la arquitectura de una aplicación desarrollada en el SDK del AR.Drone sería el siguiente. En la (Fig.2.3) se muestra dicha arquitectura de capas y como se organizan y comunican.

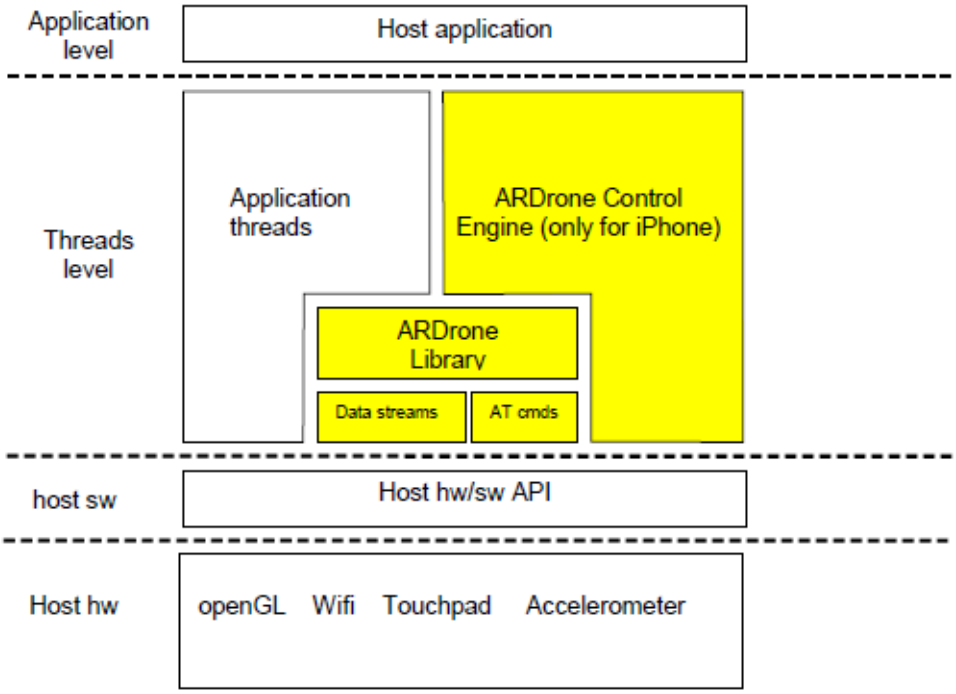


Fig.2.3: Arquitectura por capas.

Como se aprecia en la (Fig.2.3) la parte central y una de las más importantes es la "ARDrone Library", esta es una librería de código abierto con unas APIs de alto nivel para acceder al AR.Drone.

Se compone de los siguientes elementos:

- **SOFT**: Es el código especifico del AR.Drone, incluye:
    - **COMMON**: Cabecera (.h) archivos que describen la estructura de la comunicación usada por el AR.Drone. Es imprescindible en la compilación de las aplicaciones propias que estos archivos estén enlazados.
    - **Lib/ardrone_tool**: Un conjunto de herramientas para hacer más fácil el manejo del AR.Drone, como un AT comando enviando un thread y un bucle, un thread que recibe datos de navegación, uno listo para usar un pipeline para video y listo para usar la función principal main.
- **VLIB**: La librería que procesa video. Contiene las funciones para recibir y descodificar la secuencia de video.
- **VPSDK** : Librerías de propósito general, incluyen:
    - **VPSTAGES** : Para construir un proceso de video pipeline.

- **VPOS** : "wrappers" multiplataforma para funciones a nivel de sistema como gestionar la memoria y gestionar threads.

- **VPCOM** : "wrappers" multiplataforma para las funciones de comunicación. (Wifi, bluetooth, etc.).

- **VPAPI** : Ayuda para la gestión de pipelines de video y threads.

ArDroneTool a su vez está compuesto por:

- **Ardrone_tool.c** : Contiene un función estándar main en C que inicializa la conexión Wifi y inicializa todas las comunicaciones con el AR.Drone.

- **UI** : Contiene código con la gestión de un mando de control listo.

- **AT** : Contiene todas las funciones que el usuario puede llamar al controlar el AR.Drone. Muchas de ellas hacen referencia directamente a comandos AT que son automáticamente construidos con la sintaxis correcta y la secuencia de números, y posteriormente enviados al control de threads.

- **NAVDATA** : Contiene código para recibir y decodificar el sistema de navegación del AR.Drone.

Así pues, la librería ARDroneTool implementa en correcto orden los cuatro servicios explicados en el apartado 1.2.5 con la siguiente estructura de flujo:

- Un thread para la gestión de comandos AT, que recoge todos estos comandos enviados por todos los otros threads y los envía de forma ordenada y con los números de secuencia correctos.

- Un thread para la gestión de los datos de navegación, que automáticamente recibe la secuencia de estos datos, la decodifica y prepara a la aplicación cliente para la devolución de estos mismos datos.

- Un thread para la gestión de vídeo, que automáticamente recibe la secuencia de vídeo y prepara a la aplicación cliente para la confirmación de los datos.

- Un thread de control que se ocupa de otras peticiones como las de control de correcta entrega de los datos y correcto recibimiento.

Todos estos threads se encargan de conectarse con el AR.Drone en su creación y lo hacen bajo el uso de la librería vp_com, que se encarga de reconectarlos al AR.Drone si por algún caso fuera necesario y se perdiera la conexión momentáneamente.

Estos threads y su correcta inicialización son creados y gestionados por la función main, también ofrecida por la herramienta ARDroneTool en el archivo ardrone_tool.c

Por lo tanto todo lo que el programador tiene que hacer es aplicar su código especifico en estas funciones y gestionarlas de manera apropiada.

2.2 Sistemas de detección de personas

Una vez presentados los detalles del AR.Drone, el siguiente paso es conocer cómo funcionan los algoritmos de detección de personas que se aplicarán en las imágenes obtenidas con la aplicación desarrollada para el manejo y obtención del AR.Drone.

En este apartado se introduce el estado del arte en la detección de personas en imágenes.

Vehículos en movimiento

En un campo de investigación muy similar, como es la detección de peatones desde un vehículo terrestre, hay métodos que funcionan muy eficientemente, pero estos funcionan cuando las personas son totalmente visibles y aparecen en un rango de posturas muy limitado, como puede ser caminar o estar de pie. Se han obtenido muy buenos resultados con modelos monolíticos descriptivos HOG [1], y a su vez, optimizando los resultados introduciendo a estos métodos movimiento [2],[3] y características del color [4]. Por ello, estos modelos no son eficientes en escenarios donde las personas pueden aparecer parcialmente escondidas o tapadas, y con una variabilidad de posturas enormes lo que dificulta muchísimo más su detección.

Fig.3.1: Personas totalmente visibles [5].

Vehículos aéreos

Sin embargo, los modelos basados en representaciones monolíticas son poco eficaces en escenarios complejos, como ocurre en la búsqueda y rescate de personas [6]. En particular están muy cuestionados por oclusiones parciales y altas variabilidades en las posturas de las personas, que se producen con frecuencia en tales situaciones. La figura (Fig 3.2) muestra varias imágenes de muestra obtenidos de la cámara de a bordo de un UAV, lo que demuestra la complejidad de la detección de las personas en escenario como estos. Las personas están parcialmente ocluidas y se muestran en una amplia gama de poses en un ambiente típicamente muy desordenado. Usar un sistema basado en la representación monolítica de persona para detectar el cuerpo completo es propenso a fallar en estos escenarios.

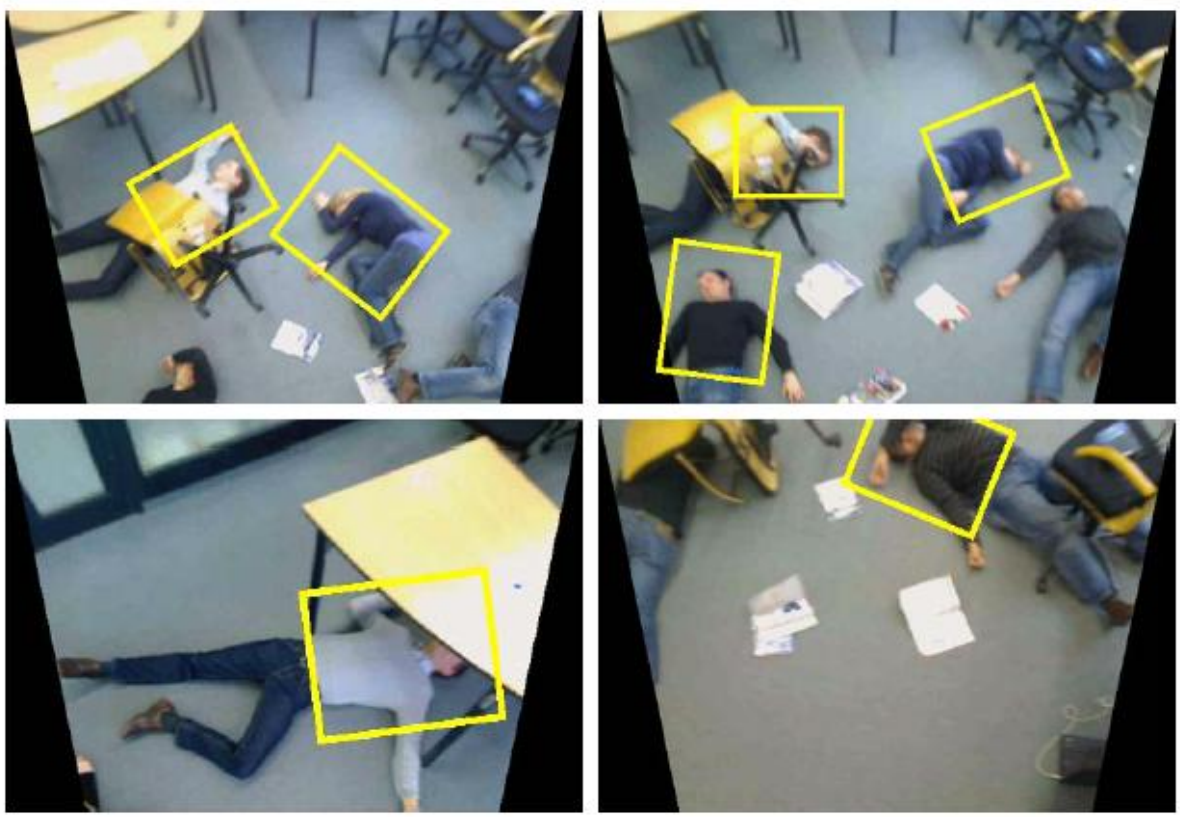


Fig.3.2: Personas parcialmente visibles [7].

Deformable por partes

Otro tipo de métodos de detección de personas, los llamados "part based models" descomponen el aspecto de las personas en múltiples componentes o partes [8], [9], [10]. Estos están mejor adaptados a abordar la detección de seres humanos parcialmente visibles utilizando un gran número de detectores de partes especializados, los cuales han sido entrenados para detectar regiones del cuerpo con determinadas características y articulaciones muy flexionadas. Estos modelos son mucho más adecuados para la detección de personas desde UAVs.

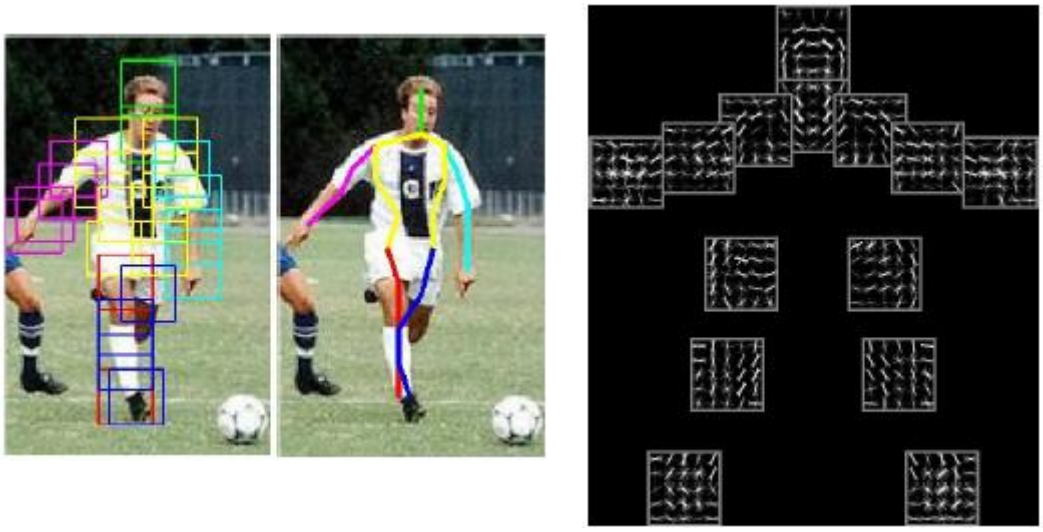


Fig.3.3: Part based model [11].

Existen dos modelos referentes a "part-based-models":

El modelo PS trabaja con partes correspondientes a los brazos y las piernas, el torso y la cabeza y necesita de partes etiquetadas para el aprendizaje. Es más preciso en la estimación de la escala de personas en una imagen, en comparación con DPM, pero es menos preciso en la localización de personas aunque ofrece peor rendimiento. Utiliza un solo modelo para todo el cuerpo [7].

El modelo DPM también se basa en estructuras pictóricas como PS, todas las partes del cuerpo están conectadas directamente a la raíz y difiere de PS en la interpretación de las partes. Ya que, DPM descubre automáticamente las partes del cuerpo que se visualizan en las imágenes de entrenamiento. Requiere un mayor número de imágenes para entrenar el detector DPM y combina dos modelos para un cuerpo completo, como son la parte superior del cuerpo y la inferior [7].

En varios estudios se ha demostrado la combinación de varios de estos modelos que pueden aumentar la fiabilidad del sistema. También que el rendimiento de detección se puede mejorar sustancialmente mediante la integración de la altura y de información proporcionada por otros sensores incorporados en el UAV.

Capítulo 3:

Desarrollo de la aplicación

3.1 Preparación del entorno de desarrollo

Para llevar a cabo los objetivos del proyecto se analizarán a continuación los distintos elementos que se necesitan y formarán parte del desarrollo del proyecto.

Existen multitud de librerías para el pilotaje y navegación del AR.Drone, escritas en C, C++ y Java entre otros. Un paso de gran importancia es la de escoger el mejor entorno de programación y aquel que ofrezca más garantías para el éxito del proyecto. Como se explicará a lo largo de este capítulo el proyecto se empezó a desarrollar en lenguaje C, bajo el sistema operativo Ubuntu, pero el análisis de los posteriores pasos que se tenían que efectuar, demostraron que la tarea se complicaba demasiado cuando hay que manejar imágenes en C, y que el lenguaje C++ ofrecía una mayor facilidad para el manejo del AR.Drone y su tratamiento en imágenes. Así pues se optó por desarrollar el proyecto con estas herramientas y no otras:

Hardware:

UAV (Unnamed Aerial Vehicle) será el motor y el eje principal del proyecto, se encargará de desplazarse por las zonas y recoger las imágenes, además proporcionará información de sus sensores para facilitar el trabajo de los algoritmos de detección y la identificación de personas dentro de las imágenes. Se comunicará con el dispositivo que contendrá el software desarrollado.

Se dispone de un Parrot AR.Drone 2.0 con las siguientes características:

- Una cámara HD, 720p 30 fps integrada con un objetivo gran angular de 92º diagonal.
- Procesador ARM Córtex A8 32 bits a 1GHz con DSP de vídeo a 800 MHz
- RAM DDR2 1 GB a 200 MHz i USB 2.0 de alta velocidad.
- Giroscopio, Acelerómetro i Magnetómetro de 3 ejes.
- Sensores de presión, ultrasonido y cámara para la medición de la velocidad respecto al suelo.

Dispositivo Tablet, Smartphone o Portátil:

Dispositivo desde el cual se controlara el AR.Drone remotamente, la comunicación se realizará mediante conexión WIFI y debe contener y ejecutar el software que implementará la obtención de imágenes y el entorno de desarrollo que ejecutará este en el lenguaje escogido.

En el caso de Tablet o Smartphone la única restricción es que sea compatible con la aplicación AR.freeflight, pero para facilitar el desarrollo del proyecto el dispositivo escogido es el siguiente dispositivo portátil:

Se dispone de un Intel con las siguientes características:

- Procesador Intel(R) Core(TM) i7-2860QM CPU @ 2.50 GHz
- Intel Smart Cache 8 MB
- Memoria principal RAM 16,0 GB (15,9 GB disponibles)
- Tarjeta red Intel(R) 82579LM Gigabit Network Connection

Clúster o sistema distribuido:

Para el análisis de las imágenes y la ejecución del software de detección de personas podría ser necesario un sistema con una gran capacidad de procesamiento y una gran cantidad de memoria, ya que el código se compone de muchas instrucciones, por lo tanto los test a gran escala se podrían ejecutar si fuera necesario en el clúster del Centre de Visió per Computador.

Software:

Sistema Operativo Windows XP:

Sistema operativo donde se desarrollarán las aplicaciones y ejecutará el entorno de desarrollo. Sistema de 32 y 64 bits

Disponible en el software de la Universidad Autónoma de Barcelona.

Visual Estudio 2010:

Entorno de desarrollo necesario para desarrollar y modificar el software del proyecto y librerías externas. Se ejecutará sobre el sistema Operativo Windows XP y el lenguaje de desarrollo será C++.

Disponible en el software libre de la Universidad.

AR.Drone:

Librería implementada en C++ donde las funciones incorporan a alto nivel las instrucciones necesarias para comandar y pilotar el parrot AR.Drone. También permite añadir nuevas funciones personales y modificar las existentes, así como obtención de imágenes.

OpenCV:

Librería de dominio libre disponible en C++, C, Python y Java. Soporte para Windows, Linux, MAC OS, IOS y Android. Esta librería está construida con el fin de ser altamente eficiente computacionalmente en aplicaciones de tiempo real y que trabaja con imágenes. Escrita en un optimizado C/C++ la librería saca ventaja de los procesadores multicore.

Disponible en la web:

Enlace: http://sourceforge.net/projects/opencvlibrary/files/opencv-win/2.4.5/OpenCV-2.4.5.exe/download

CV_Drone:

Esta librería libre incluye las dos librerías anteriormente nombradas en una para ser usadas en una misma solución de Microsoft Visual Studio 2010. Así se puede combinar la librería de alto nivel para el AR.Drone con OpenCV para tratar las imágenes que el dispositivo AR.Drone recoge.

Disponible en la web:

Enlace : https://github.com/puku0x/cvdrone

Código PartBased Detector:

Código para diferentes aplicaciones (Matlab, octave, python) donde se proporciona la parte de Train, Test y otros métodos para la detección de personas.

Disponible en github:

Enlace : https://github.com/wg-perception/PartsBasedDetector

3.2 Aplicación para comandar el AR.Drone

Como ocurre con cualquier aplicación basada en C, el punto de entrada inicial para cada aplicación de AR.Drone es una función "main". Cuando se crea una nueva aplicación con la librería ARDroneTool, no se tiene que escribir de nuevo esta función, ya que esta librería ya proporciona una por defecto. Esta librería ARDroneTool incluye una versión de esta función con todo el código necesario para iniciar una aplicación. Todo lo que se tiene que hacer es escribir el código específico de la aplicación, y compilarlo junto con la librería ARDroneLIB para conseguir una aplicación para el AR.Drone completamente funcional.

En la (Fig 4.1) se muestra la función main de una aplicación para ARDrone. Se encuentra en el archivo ardrone_tool.c y no requiere ninguna modificación. Todas las aplicaciones que se crean deben tener una función "main" casi idéntica a ésta.

```
int main(int argc, char *argv[])
{
    ...
     ardrone_tool_setup_com( NULL );
     ardrone_tool_init(argc, argv);
    while( SUCCEED(res) && ardrone_tool_exit() == FALSE )
     {
         res = ardrone_tool_update();
     }
    res = ardrone_tool_shutdown();
}
```

Fig.4.1: Función main.

Esta función lleva a cabo las siguientes tareas:

- Configura la red WIFI.

- Inicializa los puertos de comunicación (comandos AT, NAVDATA, Video y Control).

- Llama a la función ardrone_tool_init_custom. Su estructura se define en el archivo de cabecera ardrone_tool.h, y debe ser definido y personalizado por el desarrollador. En esta función podemos encontrar:

    - La inicialización local para nuestra propia aplicación.
    - La inicialización de los dispositivos de entrada, con la función ardrone_tool_input_add.
    - La puesta en marcha de todos los threads, excepto el navdata_update y el ardrone_control que son iniciados por la función main.

    En la figura (Fig 4.2) se muestra un ejemplo de cómo podría ser un ejemplo de la misma.

- Inicia el thread navdata_update que se encuentra en el archivo ardrone_navdata_client.c. Para ejecutar correctamente esta rutina, el usuario debe declarar una tabla ardrone_navdata_handler_table . En la figura (Fig 4.2) se muestra un ejemplo de cómo declarar esta tabla. La macro se encuentra en el archivo de cabecera ardrone_navdata_client.h.

- Inicia el thread ardrone_control que se encuentra en el archivo ardrone_control.c. Para enviar un evento se debe utilizar la función ardrone_control_send_event declarada en el archivo ardrone_control.h.

- Reconocer el AR.Drone para indicar que estamos listos para recibir el NAVDATA, es decir, la información de navegación.

- Por último llama a la función ardrone_tool_update en el bucle principal. La aplicación no vuelve de esta función hasta que finaliza. Esta función recupera la información del dispositivo para enviársela al AR.Drone. El usuario puede declarar la función ardrone_tool_update_custom, que será llamada por la función ardrone_tool_update.

```
C_RESULT ardrone_tool_init_custom(int argc, char **argv)
{
gtk_init(&argc, &argv);
/// Init specific code for application
ardrone_navdata_handler_table[NAVDATA_IHM_PROCESS_INDEX].data = &cfg;
// Add inputs
ardrone_tool_input_add( &gamepad );
// Sample run thread with ARDrone API.
START_THREAD(ihm, &cfg);
return C_OK;
}
```

Fig.4.2: Ej. Función init_custom.

Durante la ejecución de la aplicación, la librería ARDroneTool llama automáticamente a un conjunto de funciones definidas por el usuario de respuesta cada vez que algún navdata (paquete de información de navegación) llega desde el AR.Drone.
Para declarar una función de respuesta se debe añadir esta a la tabla NAVDATA_HANDLER_TABLE. En la figura (Fig.4.3) se muestra un ejemplo:

```
BEGIN_NAVDATA_HANDLER_TABLE //Mandatory
  NAVDATA_HANDLER_TABLE_ENTRY(navdata_ihm_init, navdata_ihm_process,
      navdata_ihm_release,
NULL)
END_NAVDATA_HANDLER_TABLE //Mandatory
//Definition for init, process and release functions.
C_RESULT navdata_ihm_init( mobile_config_t* cfg )
{.
..
}
C_RESULT navdata_ihm_process( const navdata_unpacked_t* const pnd )
{.
..
}
C_RESULT navdata_ihm_release( void )
{
}
```

Fig.4.3: Navdata_handler_table.

Una vez declarada en la tabla ya se puede gestionar, donde desde un puntero podemos acceder a la distinta información de navegación que nos proporciona el AR.Drone. Un ejemplo de ello sería el de la siguiente figura (Fig 4.4):

```
/* Receving navdata during the event loop */
inline C_RESULT demo_navdata_client_process( const navdata_unpacked_t* const
    navdata )
{
  const navdata_demo_t* const nd = &navdata->navdata_demo;

  printf("Navdata for flight demonstrations\n");

  printf("Control state : %s\n",ctrl_state_str(nd->ctrl_state));
  printf("Battery level : %i/100\n",nd->vbat_flying_percentage);
  printf("Orientation   : [Theta] %f  [Phi] %f  [Psi] %f\n",nd->theta,nd->phi,nd
      ->psi);
  printf("Altitude      : %i\n",nd->altitude);
  printf("Speed         : [vX] %f  [vY] %f\n",nd->vx,nd->vy);

  printf("\033[6A");  // Ansi escape code to go up 6 lines

  return C_OK;
}
```

Fig.4.4: Navdata_handler_table.

Hasta este punto, se ha mostrado cómo se inicializa una aplicación ARDrone y cómo se gestionan los eventos NAVDATA y de control. Además de los aspectos de la aplicación creada, aparte de eso también hay pequeños detalles que deben tenerse en cuenta en la creación de la aplicación final.

Es responsabilidad del usuario manejar los threads. Para ello, debemos declarar una tabla de threads con MACRO definida en el archivo vp_api_thread_helper.h. En la figura (Fig 4.5) se muestra cómo declarar esta tabla. Los threads navdata_update y ardrone_control no necesitan ser ejecutados ni liberados, ya que de esto se encarga la función ARDroneMain para todos los demás threads, se debe utilizar la macro denominada START_THREAD y JOIN_THREAD.

```
BEGIN_THREAD_TABLE //Mandatory
THREAD_TABLE_ENTRY( ihm, 20 ) // For your own application
THREAD_TABLE_ENTRY( navdata_update, 20 ) //Mandatory
THREAD_TABLE_ENTRY( ardrone_control, 20 ) //Mandatory
END_THREAD_TABLE //Mandatory
```

Fig.4.5: thread table.

3.2.1 Estructura de la aplicación

Como se ha explicado en el capítulo 3.1, la aplicación se desarrolla en lenguaje C para pilotar el AR.Drone mediante el manejo de los threads anteriores. Pero nuestra aplicación no solo se centra en el pilotaje y manejo del AR.Drone, sino que queremos obtener imágenes y al mismo tiempo debe ser posible editarlas o aplicar algoritmos complicados. Dado esto, el desarrollo de la aplicación final mediante este método, y como se empezó a desarrollar el proyecto, se podía complicar de manera excesiva, por lo que se decidió trabajar un nivel por encima de este entorno compactando estas instrucciones C en métodos de la clase AR.Drone.

3.2.1.1 Estructura general

Con la librería AR.Drone, todo el manejo de los threads queda relegado a un segundo plano y de manera más cómoda para el programador, ya que en C++ la clase AR.Drone se trata como una más y sus funciones definidas se encargan de la parte más engorrosa.

Esto nos permitirá dividir y desarrollar nuestra aplicación en 2 grandes bloques, representadas por clases y estructurarla de manera que cada modulo se encargue de sus funciones. En la figura (Fig.5.1) se muestra una imagen general de los módulos que componen la aplicación final y como se comunican entre ellas.

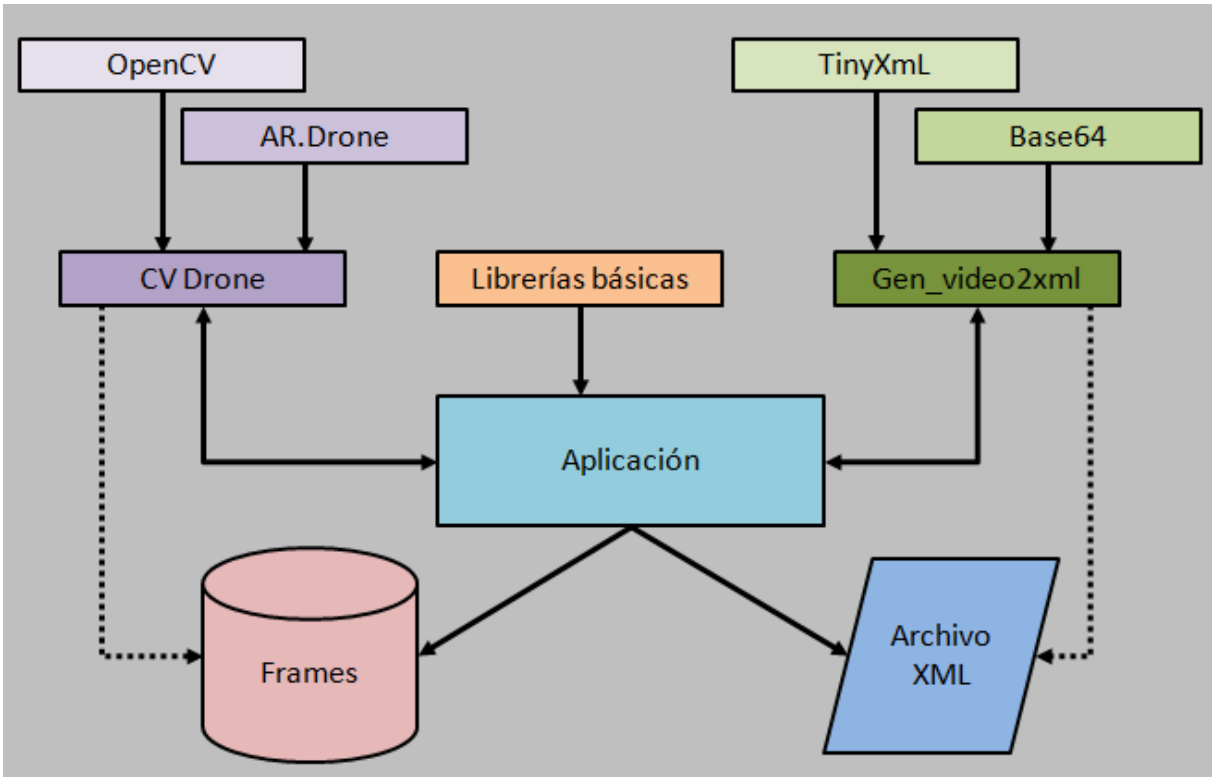


Fig.5.1: Estructura general.

En la imagen se puede apreciar cómo se estructura nuestra aplicación, que depende sobre todo de los módulos de CV Drone y Gen_video2xml, y como no, de librerías básicas de Windows para gestionarlas a ellas y a la distinta información que generan.

- **CV Drone**: Es la librería compuesta por las dos librerías ARDrone y OpenCV. Desde ella se puede pilotar, configurar y obtener imágenes del AR.Drone y aplicar las conocidas funciones de OpenCV para editar y gestionar las imágenes.

- **Librerías básicas**: Son las librerías necesarias para programar en C y C++, en ellas se gestiona la entrada y salida de ficheros y todo tipo de estructuras de datos necesarios para adaptar los datos a los tipos correctos de parámetros de las funciones.

- **Gen_video2xml:** Es la clase que gestiona los datos obtenidos por los sensores del AR.Drone y los adapta para guardarlos en el archivo XML. Se extenderá la explicación con más detalles en el apartado 3.4 de este capítulo.

Esta aplicación genera dos tipos de datos de salida:

- **Frames**: Son el conjunto de imágenes que forman el video o la secuencia de imágenes obtenidas por el AR.Drone desde una de sus dos cámaras, tanto la

Frontal como Zenital. Se extenderá la explicación con más detalles en el apartado 3.3 de este capítulo.

- **Archivo XML**: Es el archivo que contiene la información de los sensores del AR.Drone. Se extenderá la explicación con más detalles en el apartado 3.4 de este capítulo.

3.2.1.2 Estructura módulo ARDrone

Para conocer y entender la estructura y el funcionamiento de la aplicación es esencial conocer cómo trabaja el modulo ARDrone y como se ha producido el salto cualitativo para el programador de lo explicado en el capítulo 3.1. Para ello a continuación se expone como integra esta librería las instrucciones y el control de threads para manipular el AR.Drone en el lenguaje de alto nivel y orientado a objetos.

El modulo ARDrone se representa con la clase ARDrone, esta contiene funciones públicas para el usuario para manipular el AR.Drone y otras privadas para gestionar la configurar interna, como son el manejo de los threads. En la figura (Fig 6.1) se muestra la declaración de estas funciones empezando por el constructor y el destructor y terminando por funciones menos críticas como la obtención de imágenes.

```
// AR.Drone class
class ARDrone {
public:
    // Constructor / Destructor
    ARDrone(const char *ardrone_addr = NULL);
    virtual ~ARDrone();
    // Initialize
    int open(const char *ardrone_addr = ARDRONE_DEFAULT_ADDR);
    // Update
    int update(void);
    // Finalize (Automatically called)
    void close(void);
    // Get an image for OpenCV
    IplImage* getImage(void);
    // Get AR.Drone's firmware version
    int getVersion(int *major = NULL, int *minor = NULL, int *revision = NULL);
    // Get sensor values
    double getRoll(void);       // Roll angle  [rad]
    double getPitch(void);      // Pitch angle [rad]
    double getYaw(void);        // Yaw angle   [rad]
    double getAltitude(void);   // Altitude    [m]
    double getVelocity(double *vx = NULL, double *vy = NULL, double *vz = NULL); // Velocity [m/s]
    // Battery charge [%]
    int getBatteryPercentage(void);
    // Take off / Landing / Emergency
    void takeoff(void);
    void landing(void);
    void emergency(void);
    // Move with velocity [m/s]
    void move(double vx, double vy, double vr);
    void move3D(double vx, double vy, double vz, double vr);
    // Change camera channel
    void setCamera(int channel);
    // Others
    int  onGround(void);                          // Check on ground
    void setAnimation(int id, int duration);      // Flight animation
    void setLED(int id, float freq, int duration); // LED animation
    void startVideoRecord(void);                  // Video recording for AR.Drone 2.0
    void stopVideoRecord(void);                   // You should set a USB key with > 100MB to your drone
```

Fig.6.1: Clase ARDrone.

Si vamos a la implementación de una función, por ejemplo la función takeoff() mostrada en la figura (Fig 6.2) para levantar el AR.Drone, vemos como la función genera el comando AT necesario y genera su número de secuencia correctamente y con la exclusión necesaria para el correcto funcionamiento del programa. Tal y como se ha explicado en el capitulo 2.1.5.1. Esta implementación se repite para todas las funciones mostrada en la figura (Fig 6.1) y de esta manera es como la librería AR.Drone se comunica con el AR.Drone.

```
// -------------------------------------------------------
// ARDrone::takeoff()
// Description  : Take off the AR.Drone.
// Return value : NONE
// -------------------------------------------------------
void ARDrone::takeoff(void)
{
    // Reset emergency
    resetWatchDog();
    resetEmergency();

    // Send take off
    WaitForSingleObject(mutexCommand, INFINITE);
    sockCommand.sendf("AT*REF=%d,290718208\r", seq++);
    ReleaseMutex(mutexCommand);
}
```

Fig.6.2: Función takeoff.

A su vez esta librería se comunica con 5 clases para organizar y repartir los distintos tipos de trabajo que debe realizar en su comunicación con el AR.Drone.

1. **Command.cpp** : Incluye las funciones de movimiento y las de aterrizaje y despegue.

2. **Navdata.cpp**: Incluye las funciones referentes a los datos de navegación, el valor de los sensores de los ángulos, la velocidad, la altitud y el porcentaje de batería.

3. **Udp.cpp**: Incluye las funciones para gestionar la comunicación, la creación del socket y enviar y recibir por él.

4. **Versión.cpp**: Incluye las funciones para gestionar la versión del AR.Drone, ya que hay diferencia en algunos protocoles dependiendo de la versión.

5. **Video.cpp**: Incluye las funciones para obtener video e imagen, así como la conversión y aplicación de los datos con los códecs necesarios.

3.2.2 Recolección de imágenes

Una parte esencial del proyecto es la obtención de imágenes y video desde el AR.Drone, para poder aplicar los algoritmos de detección de personas más adelante. Para ello es imprescindible adaptar la obtención de imágenes del SDK a nuestra librería y posteriormente a nuestra aplicación.
A continuación se presenta como es el proceso de obtención de imágenes desde el principio que sale del AR.Drone hasta el final que es almacenada como archivo.

El SDK incluye métodos para gestionar el flujo de vídeo. Todo el proceso está gestionado por un pipeline de video, construido como una secuencia de etapas que llevan a cabo los pasos básicos, tales como la recepción de los datos de vídeo desde el socket, la descodificación de los frames, y la visualización de estos.

Una etapa está definida en una estructura denominada vp_api_io_stage_t que se define en el archivo <VP_Api/vp_api.h>.

Definición de la estructura de una etapa:

En la figura (Fig 7.1) se muestra cómo construir un pipeline con etapas. En esta ejemplo se abre un socket y se recibe una secuencia de video.

```
#include <VP_Api/vp_api.h>
#include <VP_Api/vp_api_error.h>
#include <VP_Api/vp_api_stage.h>
#include <ardrone_tool/Video/video_com_stage.h>
#include <ardrone_tool/Com/config_com.h>
vp_api_io_pipeline_t pipeline;
vp_api_io_data_t out;
vp_api_io_stage_t stages[NB_STAGES];
video_com_config_t icc;
icc.com = COM_VIDEO();
icc.buffer_size = 100000;
icc.protocol = VP_COM_UDP;
COM_CONFIG_SOCKET_VIDEO(&icc.socket, VP_COM_CLIENT, VIDEO_PORT, wifi_ardrone_ip);
pipeline.nb_stages = 0;
stages[pipeline.nb_stages].type = VP_API_INPUT_SOCKET;
stages[pipeline.nb_stages].cfg = (void *)&icc;
stages[pipeline.nb_stages].funcs = video_com_funcs;
pipeline.nb_stages++;
```

Fig 7.1: Pipeline por etapas.

En la figura (Fig 7.2) se muestra cómo ejecutar el pipeline. Este código debe ser implementado por el usuario, para ello, se puede utilizar un thread dedicado.

```
res = vp_api_open(&pipeline, &pipeline_handle);
if( SUCCEED(res) )
{
int loop = SUCCESS;
out.status = VP_API_STATUS_PROCESSING;
while(loop == SUCCESS)
{
if( SUCCEED(vp_api_run(&pipeline, &out)) )
{
if( (out.status == VP_API_STATUS_PROCESSING || out.status ==
VP_API_STATUS_STILL_RUNNING) )
{
loop = SUCCESS;
}
}
else loop = -1; // Finish this thread
}
vp_api_close(&pipeline, &pipeline_handle);
}
```

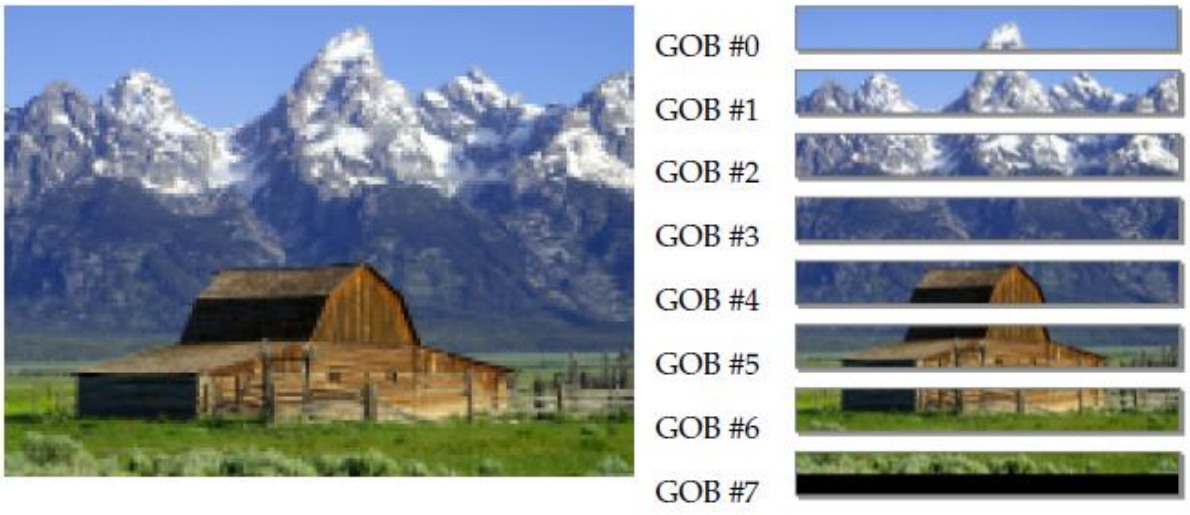
Fig 7.2: Ejecutar pipeline.

Las funciones se definen en el archivo <VP_Api/vp_api.h>:

- vp_api_open: Inicialización de todas las etapas declaradas en el pipeline.
- vp_api_run: Ejecución de todas las etapas, en un bucle.
- vp_api_close: Cierra todas las etapas.

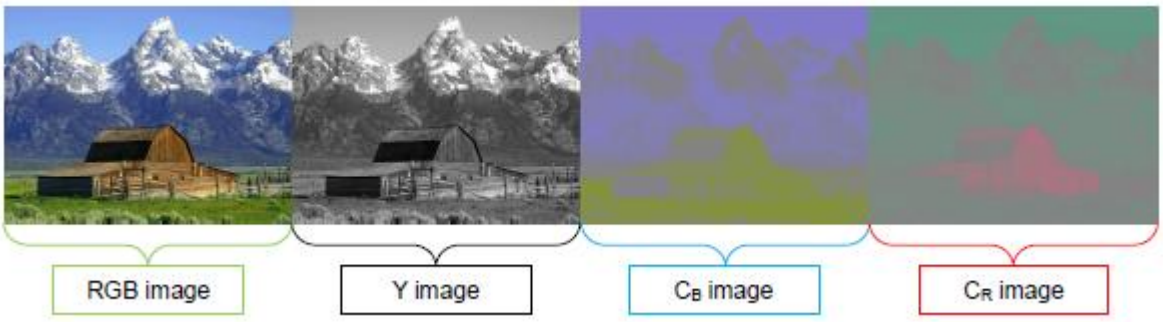
Una vez generado el pipeline, ya podemos comunicarnos con el AR.Drone para obtener imágenes y video por él. A continuación veremos cómo se codifica la imagen para viajar por el pipeline y acabar en nuestra aplicación decodificada y guardada en formato JPEG.

Una vez captada por el sensor, es decir la cámara del AR.Drone, la imagen es cortada en grupos de bloques (GOB), cada uno de 16 líneas de altura, quedando de la siguiente forma:



A su vez estos GOB son divididos en Macro bloques, que representan una imagen 16x16 pixels.

Estos macro bloques contienen información de una imagen 16x16 pixels en formato YCbCr, sampleado a 4:2:0.



La imagen es finalmente guardada en memoria en 6 bloques de 8x8 de la siguiente forma:

- 4 bloques (Y0,Y1,Y2 y Y3) que forman la imagen Y de 16x16 pixels de la componente de luminancia.
- 2 bloques de la componente de color azul y rojo.

En la obtención de imagen hay dos codecs disponibles, el UVLC y el p264. Las librerías de estos codecs vienen enlazadas a la solución de la aplicación, de esta manera son aplicables en la compresión de estas imágenes en JPEG.

Para comprimir las imágenes en JPEG el códec ejecuta cuatro pasos:

1. A cada bloque de 8x8 pixels del macro bloques se le aplica la Transformada discreta del coseno (DCT).
2. A cada elemento 8x8 pixels con la DCT aplicada es divido por una matriz de cuantizacion establecida por una formula.
3. Se reordena el bloque de 8x8 pixels en zigzag.
4. Cada bloque es finalmente codificado usando un método de "variable length coding" , es decir, codificar los valores con mayor frecuencia con las palabras código más cortas en longitud de bits, para bajar el numero de bits transferido.

En la siguiente imagen veríamos este proceso de forma grafica:

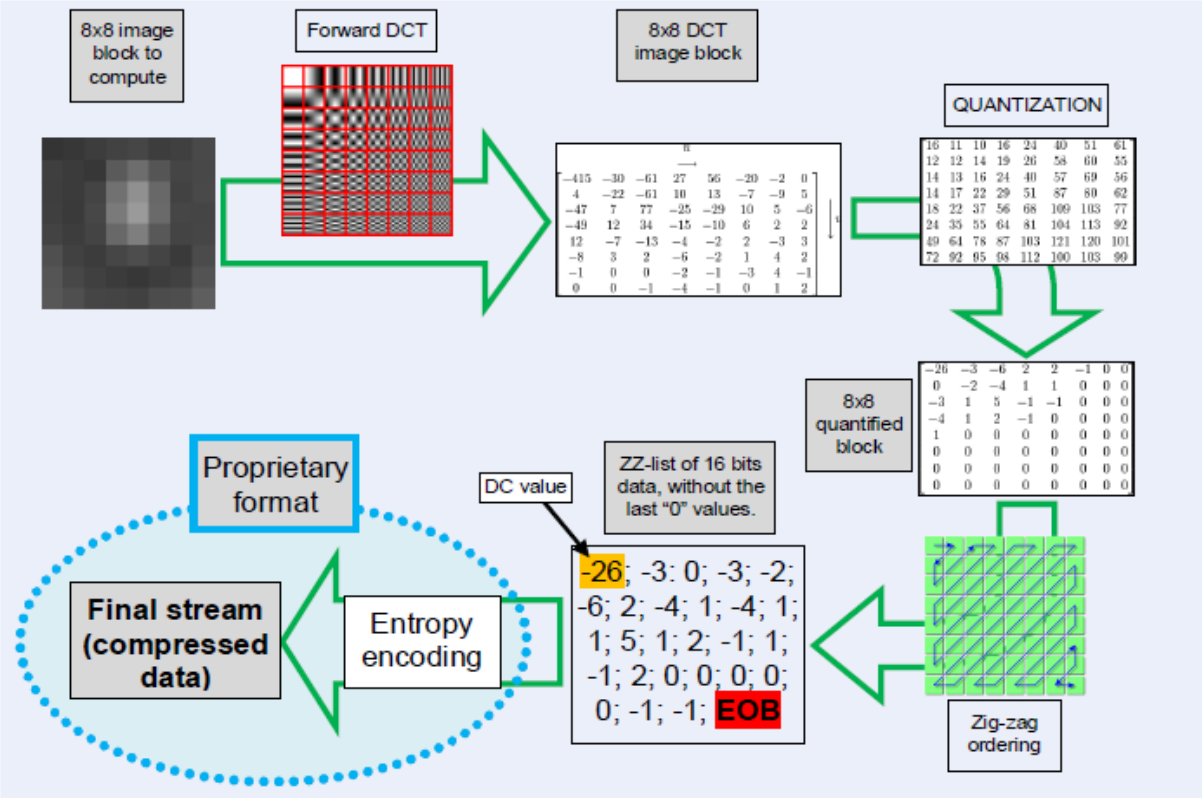
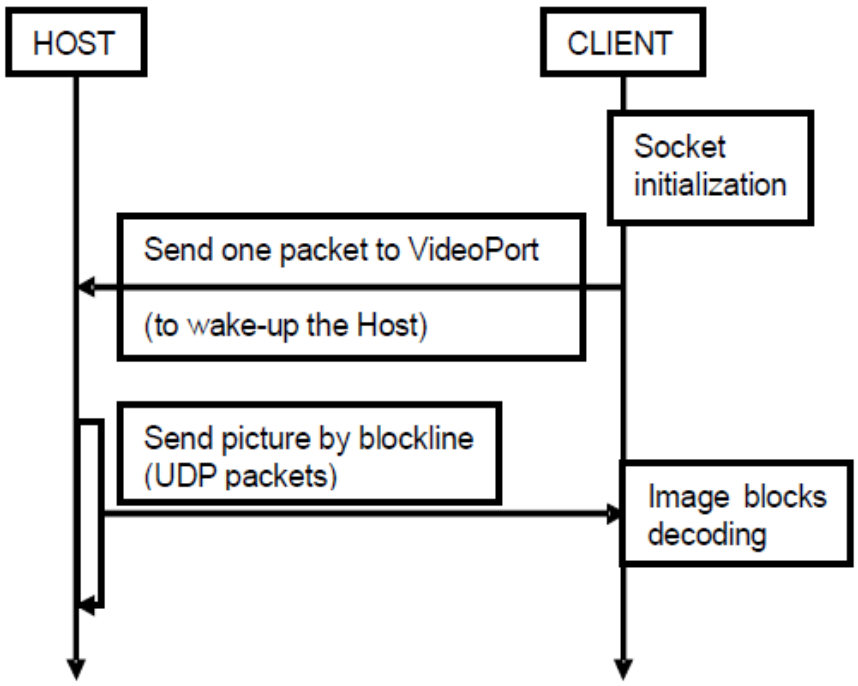


Fig 7.3: Codificación Jpeg.

Una vez codificada la imagen, se genera el datagrama que se mandará por el puerto UDP a nuestra aplicación.



Como ocurre en el capitulo anterior, las diferentes funciones de ARDrone gestionan estos procesos de decodificación y recepción, hasta llegar a la función del punto más alto ardrone.getimage() que nos permite obtener una imagen manejable y expresada en el tipo de dato IplImage*.

La estructura IplImage se heredada de la Biblioteca de procesamiento de imágenes Intel, en el que el formato es nativo. De esta manera podemos utilizar las imágenes en las funciones de OpenCV, per ejemplo cvShowImage o cvSaveImage, ya que la librería CVDrone esta creada y optimizada para precisamente este acoplamiento entre las dos librerías, ARDrone y OpenCV.

Al final del recorrido, y con la imagen en formato IplImage*, solo queda guardarla con el formato adecuado. Para ello se usa la función cvSaveImage, que recibe dos parámetros:

- **Filename**: Es un parámetro muy importante y vital aunque no lo parezca, de él depende de que quede bien ordenado el frame dentro de los otros miles de frames para su posterior montaje. En la aplicación se ha definido un estándar a seguir para facilitar esta tarea. El nombre de cada frame se formará concatenando las cadenas de caracteres siguientes:

  - **S1 = Nombre de sesión** : Será el nombre de sesión introducido al inicio de la aplicación introducido por el usuario. No se puede modificar una vez introducido.

  - **S2 = REC_XXX :** Es la carpeta que contiene cada video, las 'XXX' indican el numero de video de esa sesión y que se representan con 3 dígitos. Empieza en 001 y acaba en 999, permitiendo un máximo por sesión de casi 1000 videos.

  - **S3 = HHMMSS :** Representa la hora, los minutos y segundos en dos dígitos cada uno. Esta cadena de caracteres se genera cada vez que se inicia un video y es para todos los frames el mismo, correspondiente a la hora cuando se capturo el primer frame.

  - **S4 = Numero de frame :** Con estos 6 dígitos se representa el numero de frame, empezando desde el 000000 hasta el 999999. Con este formato se solucionaron los problemas del principio donde se ordenaban incorrectamente los frame por problemas de dígitos. Permite obtener 1 millón de frames de un mismo video, obteniendo 10 frames por segundo da un máximo de 27 horas para cada vídeo.

  - **S5 = Extensión del archivo**: Como la imagen se codifica en JPEG, lógicamente la extensión es .jpg haciendo el archivo reconocible por los visualizadores de imágenes de los sistemas operativos.

  Por lo tanto el Filename concatenará estas cadenas de caracteres mediante el carácter "_", quedando de la forma siguiente: S1_S2_S3_S4.jpg

- **Image**: Es la estructura donde se almacena la imagen. Almacenada en una variable de tipo ‘ IplImage * ’ compatible con las funciones que reciben ‘ const cvArr * ’.

De esta manera se obtiene el conjunto de imágenes o frames de un video donde cada una es identificable por su nombre y le corresponde un frame posterior y otro anterior. Facilitando el aprendizaje del sistema de detección de personas.

En la figura (Fig 7.4) siguiente se muestra un ejemplo de 9 frames consecutivos y demostrando lo anteriormente explicando:

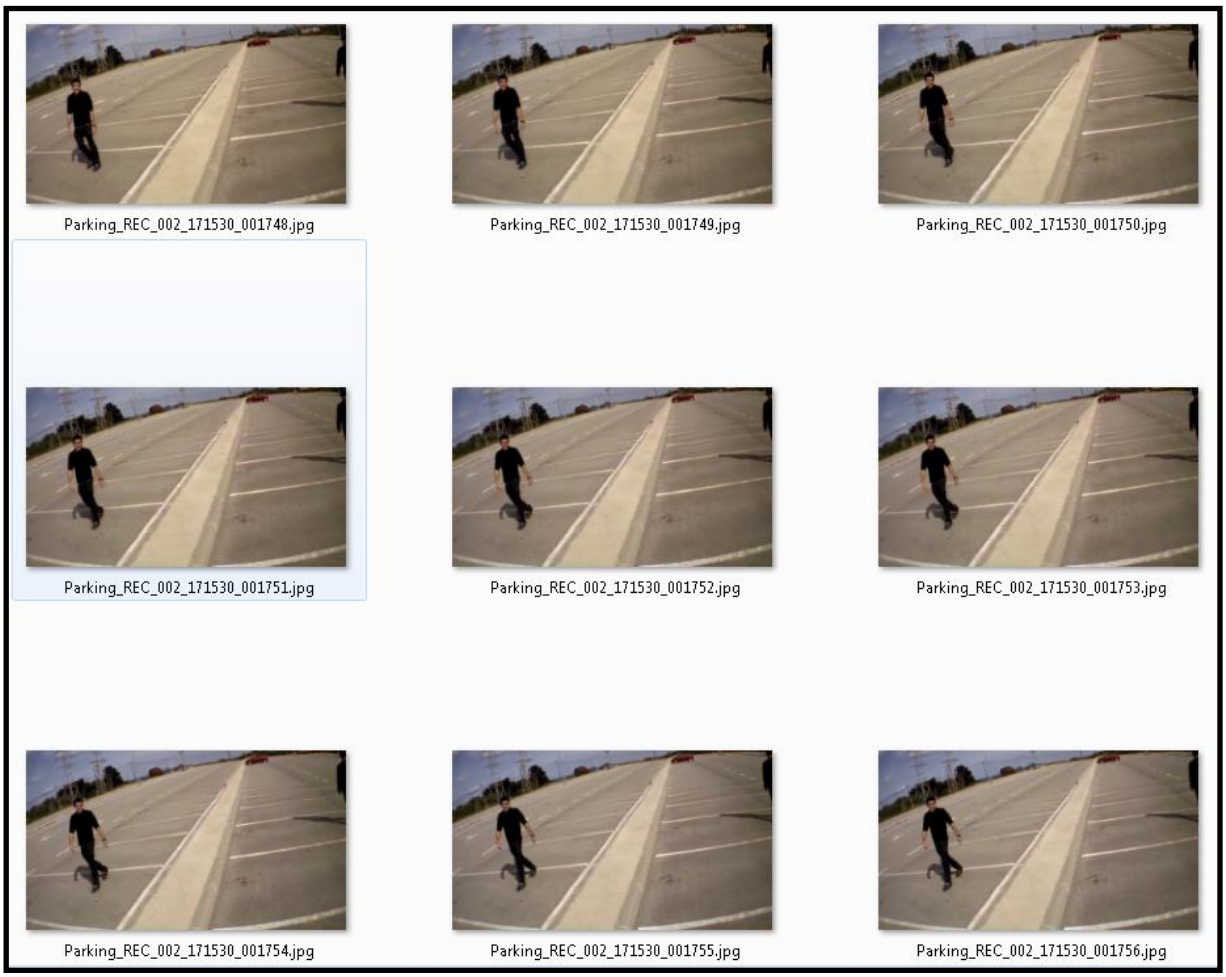


Fig 7.4: Secuencia de frames.

En la imagen se aprecia los frames del segundo video de la sesión Parking, esta grabación fue iniciada a las 17:15:30 y podemos observar la secuencia del frame 1748 hasta el 1756.

3.2.3 Recolección de información de los sensores

La segunda parte esencial del proyecto después de la obtención de las imágenes es la obtención de la información de los sensores. Estos son de gran importancia ya que como

veremos en el análisis de los resultados nos proporcionan una información no visible en la imagen y son una herramienta más a favor del usuario y al servicio del algoritmo de detección de personas.

Tal y como hemos hecho para las imágenes, vamos a ir paso a paso desde el nivel más bajo disponible para el programador, para entender el funcionamiento de envió y recepción de los datos de navegación, hasta posteriormente como los trata nuestra librería y finalmente nuestra aplicación.

Los datos de navegación (o navdata) son una media dada a la aplicación cliente para recibir periódicamente (en un tiempo inferior a los 5 ms) información sobre el estado del AR.Drone (ángulos, la altitud, la cámara, la velocidad, la detección de etiquetas resultados ...). A continuación se muestra cómo recuperarlos y decodificarlos.

Los datagramas de navdata son enviados por el AR.Drone desde y hacia el puerto UDP 5554. La información se almacena en formato binario y consisten en varias secciones de bloques de datos denominados opciones.

Cada opción consta de un encabezado (2 bytes) que identifica el tipo de información contenida en el mismo, un entero de 16 bits almacena el tamaño del bloque, y varia información se almacena en enteros de 32 bits de precisión simple de punto flotante, o vectores. Todos estos datos se almacenan en 'Little Endian'.

Header 0x55667788	**Drone state**	**Sequence number**	**Vision flag**	**Option 1**			…	**Checksum block**		
				id	size	data	…	cks id	size	cks data
32-bit int.	32-bit int.	32-bit int.	32-bit int.	16-bit int.	16-bit int.	…	…	16-bit int.	16-bit int.	32-bit int.

Fig 8.1: Datagrama Navdata.

Todos los bloques comparten esta estructura común:

```
typedef struct _navdata_option_t {
uint16_t tag; /* Tag for a specific option */
uint16_t size; /* Length of the struct */
uint8_t data[]; /* Structure complete with the special tag */
} navdata_option_t;
```

Para recibir el Navdata, se debe enviar un paquete con algunos bytes al puerto NAVDATA_PORT del host. A continuación pueden pasar dos cosas:

- El AR.Drone empieza en modo bootstrap, solo el estado y el numero de secuencia son enviados.
- El AR.Drone está siempre activo, un 'navdata demo' es enviado.

Para finalizar el modo bootstrap, el cliente debe enviar un comando AT para modificar la configuración del AR.Drone. Entonces enviar una serie de comandos AT y el AR.Drone se inicializará y enviará un 'navdata demo'.

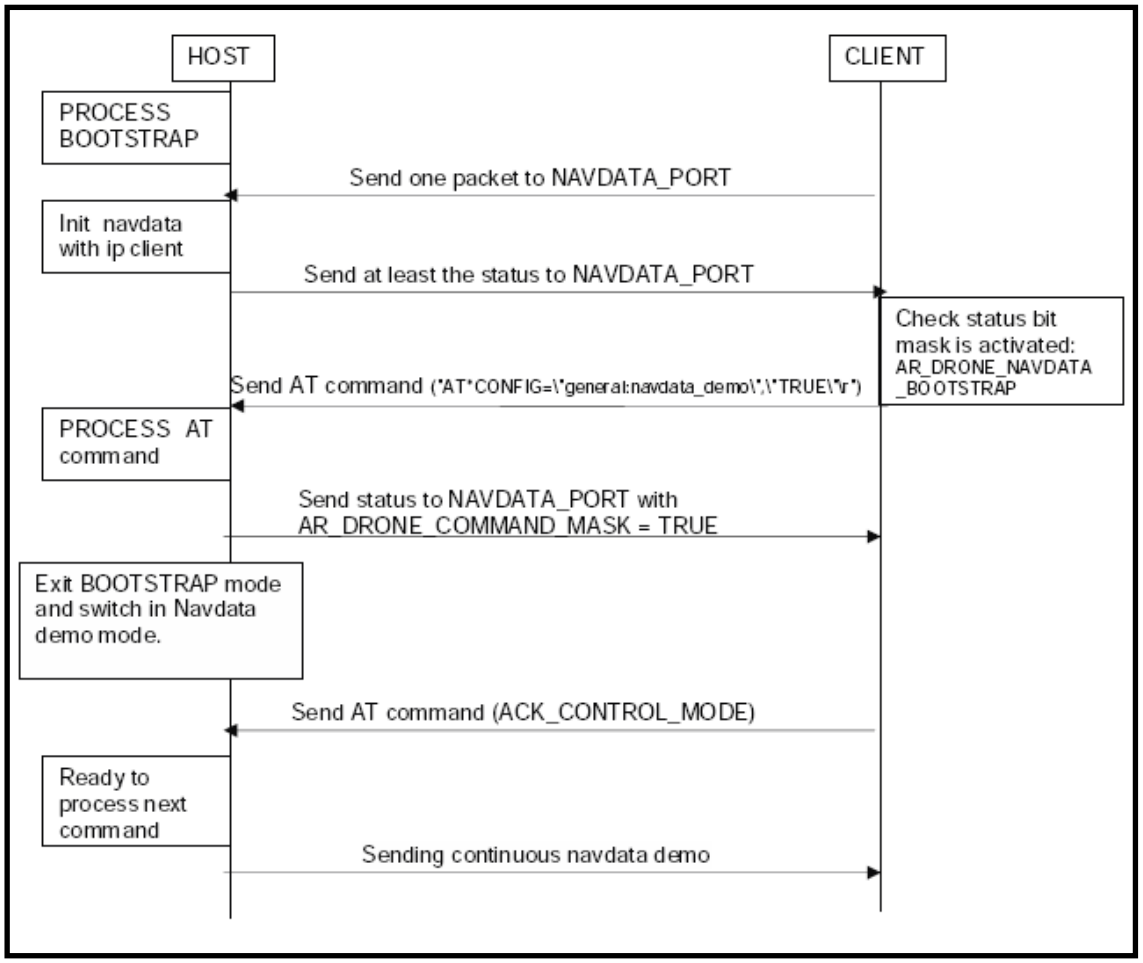


Fig 8.2: Comunicación para Navdata.

Una vez que sabemos cómo se obtiene el navdata_demo, vamos a ver como lo implementa nuestra librería ARDrone. En ella obtenemos la clase navdata.cpp, donde después de inicializar la función initNavdata y getNavData se ejecutan todas las instrucciones necesarias y explicadas anteriormente. Una vez obtenido nuestro navdata_demo, las funciones para obtener la información de los sensores en concreto solo tienen que acceder a campos de esta estructura. En la siguiente figura (Fig 8.3)se muestra un ejemplo de las funciones getRoll y getPitch obteniendo la información de los ángulos, aplicando su correspondiente conversión a radianes.

```
// ------------------------------------------------
// ARDrone::getRoll()
// Description  : Get current role angle of AR.Drone.
// Return value : Role angle [rad]
// ------------------------------------------------
double ARDrone::getRoll(void)
{
    return navdata.demo.phi * 0.001 * DEG_TO_RAD;
}

// ------------------------------------------------
// ARDrone::getPitch()
// Description  : Get current pitch angle of AR.Drone.
// Return value : Pitch angle [rad]
// ------------------------------------------------
double ARDrone::getPitch(void)
{
    return navdata.demo.theta * 0.001 * DEG_TO_RAD;
}
```

Fig 8.3: Ejecutar pipeline.

Ahora en nuestra aplicación, con la clase ARDrone declarada y del mismo modo que obteníamos las imágenes, podemos obtener esta información llamando a las funciones correspondientes.
Tal y como son llamadas estas funciones, son mandados los datos a la clase Gen_video2xml, nuestra clase encargada de codificar y generar el archivo XML.

Esta clase Gen_video2xml incluye una clase TinyXmL preparada para generar y recoger datos de ficheros XML. TinyXmL trata sus archivos XML en estructura de árbol donde el documento es la raíz del árbol y todas las etiquetas del documento son nodos hijos.
También incluye otra clase Base64 que se ha añadido, encargada de codificar los valores numéricos de los sensores para almacenarlos en el archivo XML.

En esta figura (Fig 8.4) se muestra un pequeño trozo de cómo funciona esta clase, donde se puede apreciar la generación de los nodos, su especificación de los atributos, la especificación del valor de las etiquetas y la codificación de los valores.

```
int Gen_video2XML::saveSequence(char* SeqName, char* Camera, char* id_frame, char* time_hms, (
{
    //Information frame
    TiXmlElement E_frame( "Frame" );
    TiXmlElement E_Nameframe( "Name" );
    TiXmlElement E_Time( "Time" );
    TiXmlElement E_roll( "Roll"); E_roll.SetAttribute("Encoding","base64");
    TiXmlElement E_pitch( "Pitch"); E_pitch.SetAttribute("Encoding","base64");
    TiXmlElement E_yaw( "Yaw");  E_yaw.SetAttribute("Encoding","base64");
    TiXmlElement E_altitude( "Altitude");  E_altitude.SetAttribute("Encoding","base64");
    TiXmlElement E_velocity( "Velocity");  E_velocity.SetAttribute("Encoding","base64");

    //Values Tags
    TiXmlText T_id_frame( id_frame );
    TiXmlText T_time( time_hms );

    //Encode Base64 values
    stringstream auxstr("");
    auxstr << val_roll;
    string auxdouble = auxstr.str();
    auxdouble = base64_encode( (const unsigned char*) auxdouble.c_str(),auxdouble.length() );
```

Fig 8.4: Clase Gen_video2XML.

Esta clase pues, nos generará un archivo XML como el de la figura (Fig 8.5) .
Para la creación de este archivo se asignaron también unos estándares. El nombre del archivo será el mismo que el de los frames, pero sin la secuencia de caracteres que representa el numero de frame. Este se encontrará dentro del archivo para relacionar la información con el frame correspondiente. Para que sea más entendible veamos el ejemplo de la figura (Fig 8.6):

```
- <Sequence>
    <Name>ARDrone_VideoRecorder</Name>
    <Camera>Frontal</Camera>
  - <ListFrames>
    - <Frame>
        <Name>frame_000000</Name>
        <Time>17:21:37</Time>
        <Roll Encoding="base64">LTAuMDE4NTAwNQ==</Roll>
        <Pitch Encoding="base64">MC4wODkxNTE0</Pitch>
        <Yaw Encoding="base64">Mi4yMjcz</Yaw>
        <Altitude Encoding="base64">MA==</Altitude>
        <Velocity Encoding="base64">MA==</Velocity>
      </Frame>
    - <Frame>
        <Name>frame_000001</Name>
        <Time>17:21:37</Time>
        <Roll Encoding="base64">LTAuMDE4NTM1NA==</Roll>
        <Pitch Encoding="base64">MC4wODkwOTkx</Pitch>
        <Yaw Encoding="base64">Mi4yMjc2Mw==</Yaw>
        <Altitude Encoding="base64">MA==</Altitude>
        <Velocity Encoding="base64">MA==</Velocity>
      </Frame>
```

Fig 8.5: Cuerpo archivo XML.

En ella podemos ver la información de los dos primeros frames relativos a estas dos imágenes con la cabecera del archivo XML:

Fig 8.6: Archivo XML.

Como se puede apreciar en las dos imágenes correspondientes a los frames 0 e 1, el AR.Drone en estos dos primeros frames aun estaba en el suelo, preparado para despegar. Al no tener aun movimiento en las hélices, vemos como los dos valores devueltos y codificados de altitud y velocidad están en 'MA==', que es la representación en base64 del valor 0.0. Mientras que los otros valores son prácticamente iguales excepto a por pequeñas variaciones en los últimos decimales a causa de sensibilidad en los sensores.

3.2.4 Aplicación final

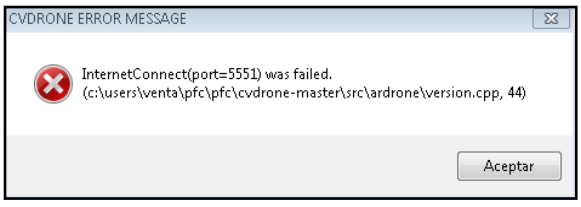
Llegados a este punto nuestra aplicación ya dispone de todos los medios y herramientas para cumplir con los objetivos con los que se empezó el proyecto.

A continuación se presenta el aspecto con que terminó la aplicación final, eliminando las instrucciones poco importantes y las modificaciones de variables que no son necesarias para entender el funcionamiento del código.

La aplicación consiste en declarar una variable de clase ARDrone y inicializarla. En este punto de inicialización, nuestra aplicación se conectará con el AR.Drone en el que hayamos establecido conexión, obtendrá la versión para configurar correctamente los protocolos y inicializará las tres clases: Command, NavData, Video. Si se produce algún error durante este proceso aparecerá un error como el de la figura 1.

```
// AR.Drone class
ARDrone ardrone;

// Initialize
if (!ardrone.open()) {
    printf("Failed to initialize.\n");
    return -1;
}
```



Si todo es correcto, habremos establecido conexión con el AR.Drone y ya estaremos listos para usar las funciones públicas de la clase ARDrone. La figura siguiente muestra el cuerpo principal respetando siempre las estructuras mencionadas en los capítulos anteriores y como estas se han programado. En ella se aprecia el bucle principal con la función update necesaria para el correcto funcionamiento, como gestiona el despegue y el aterrizaje y como realiza la modificación de las variables de movimiento en otras.

```
// Main loop
while (!GetAsyncKeyState(VK_ESCAPE)) {
    // Update
    if (!ardrone.update()) break;

    // Get an image
    IplImage *image = ardrone.getImage();

    // Take off / Landing
    if (KEY_PUSH(VK_SPACE)) {
        if (ardrone.onGround()) ardrone.takeoff();
        else                    ardrone.landing();
    }

    // Move
    double vx = 0.0, vy = 0.0, vz = 0.0, vr = 0.0;
        if (KEY_DOWN(VK_UP))    vz =  1.0;
        if (KEY_DOWN(VK_DOWN))  vz = -1.0;
        if (KEY_DOWN(VK_LEFT))  vy =  1.0;
        if (KEY_DOWN(VK_RIGHT)) vy = -1.0;

    ardrone.move3D(vx, vy, vz, vr);

    // Change camera
    static int mode = 0;
    if (KEY_PUSH('C')) ardrone.setCamera(++mode%4);

    // Display the image
    cvShowImage("camera", image);
    cvWaitKey(1);
}
```

Dentro de este bucle principal, y en cada iteración se gestiona la recolecta de imágenes y la de información de los sensores. Para gestionar la obtención de imágenes simplemente se obtiene y se guarda cada imagen correctamente en la carpeta adecuada y con el nombre adecuado, como se explica en el capitulo 3.1.2, y cumpliendo estas normas que acordamos para una buena organización y escalabilidad.

- Cada vez que se ejecuta la aplicación el usuario introduce un nombre de sesión. Esto es útil ya que varios videos con el mismo escenario pueden agruparse y obtener indirectamente información de por ejemplo en que escenario se capturaron.

- Con la aplicación en marcha se pueden grabar tantos videos como se desee. Cada vez que se almacena un video lo hace en forma de frames y estos se almacenan en

una misma carpeta, que a su vez esta forma parte de la carpeta sesión anteriormente nombrada, separando todos los frames de cada video.

- La variable que controla el número de imágenes por segundo está fijada en 10, ya que un número mayor podría penalizar demasiado la ejecución del código y provocar pérdidas de conexión y errores por demora.

En la siguiente figura se muestra cómo se organiza la información con los estándares marcados anteriormente. Un ejemplo de salida de archivos después de haber adquirido videos en distintos escenarios seria este que se muestra en la figura 2.

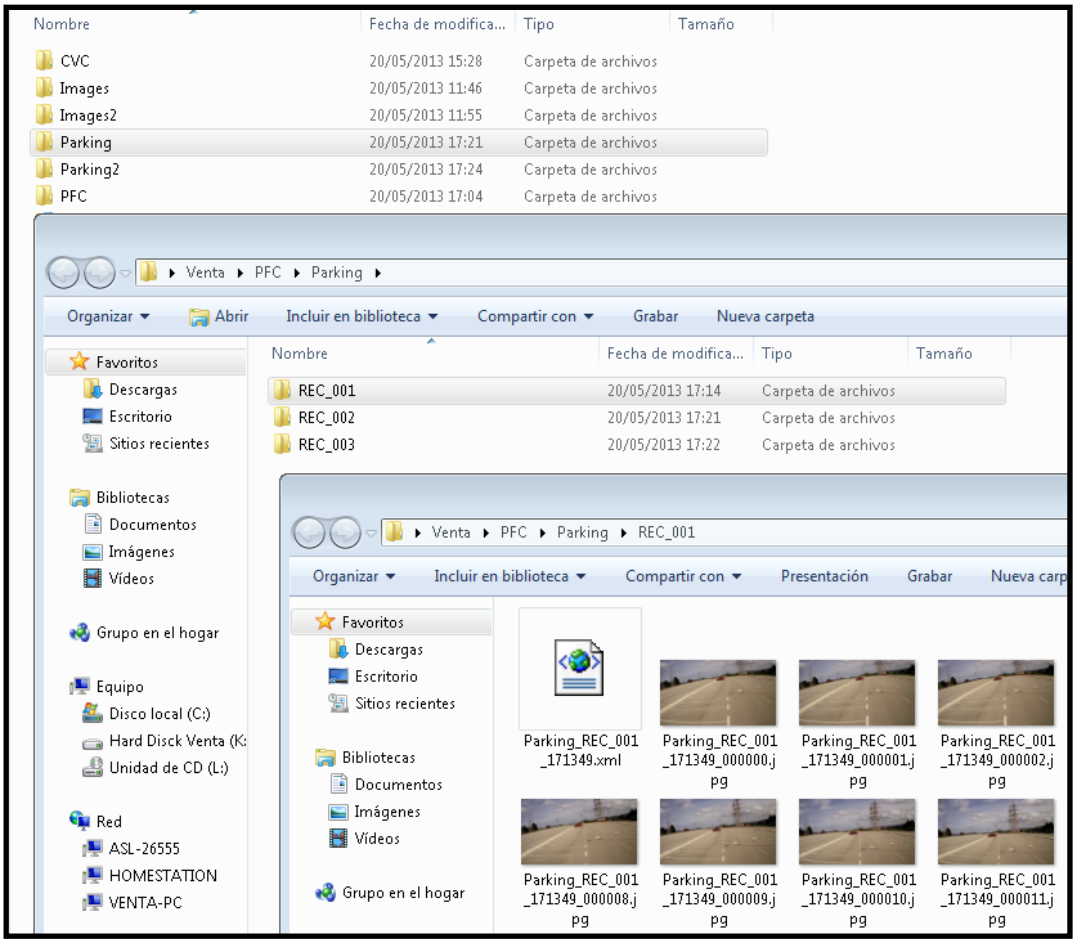


Fig 8.7: Organización de resultados.

Como se puede apreciar, se han adquirido videos de distintos escenarios, cada vez que se han capturado imágenes de distintos escenarios se ha creado una carpeta que contiene todos los videos de esa sesión. En la imagen, la sesión Parking contiene tres carpetas, que representan tres videos diferentes en el mismo escenario. Dentro del primer video, REC_001, encontramos todos los frames del video seguido por el archivo XML, que enlaza los frames con la información de los sensores en ese mismo instante en que se capturó el frame. Este formato se repite siempre y se corresponde con las otras carpetas REC_002 y REC_003 que nos encontramos.

Por otro lado, una vez gestionada la obtención de imágenes, el siguiente paso es gestionar la información de los sensores, esto se consigue mediante la clase Gen_video2xml como se explica en el capitulo 3.1.3. Así pues lo único que realiza el programa principal es crear esta clase, y llamar a las dos funciones saveSequence y savePacksequence en el orden correcto. Cada vez que se obtiene una imagen, se llama a la función saveSequence para que almacene la información del frame, y una vez finalizado el video y con la información del todos los frames, se llama a la función savePackSequence para que guarde y cree el árbol desde la raíz con todas las secuencias para el fichero XML.
La siguiente figura muestra las funciones en la aplicación con sus parámetros correspondientes:

```
//XML class
Gen_video2XML gvxml;

//Save sensors frame information
gvxml.saveSequence(SeqName, Camera, id_frame, (char*) auxss.str().c_str(), ardrone.getRoll(),

//Save all the frame in packseq (represents the video.)
gvxml.savePackSequences( FilexmlName, SeqName, Camera);
```

De esta forma funciona nuestra aplicación y que cumple con los requerimientos de los objetivos 1,2 y 3 propuestos al principio del proyecto.

3.3 Aplicando algoritmo de detección de personas.

Una vez obtenidas las imágenes y los archivos XML, se aplicaron los algoritmos de detección de personas en las imágenes obtenidas por el AR.Drone, para ver como respondía la aplicación en un escenario diferente al habitual.
Hay que recordar que el modelo que se uso para estas imágenes está entrenado en el vehículo terrestre Tazzari Zero, con lo que las imágenes se toman desde una perspectiva plana y casi a la misma altitud del suelo que los viandantes y personas. En este escenario, el AR.Drone varia en altitud desde el medio metro de altura hasta los 3 metros, haciendo que la misma imagen pueda parecer totalmente distinta a nivel de bytes y complicar la tarea del algoritmo de detección de personas.

A continuación se mostrara el resultado de aplicar el algoritmo de detección a los frames de una secuencia de imágenes. La secuencia de video escogida es la siguiente, iniciada a las 17:24:19 y finalizada a las 17:26:45, por lo que la grabación duró 2 minutos 26 segundos y se obtuvieron 2436 frames.

La figura (Fig 9.1) muestra unos cuantos frames la secuencia de video y a continuación el resultado de aplicar el algoritmo:

Fig 9.1: Aplicando algortimo.

Como se aprecia para detectar a una persona en una imagen el algoritmo se basa en interpretar sus extremidades. Una vez localizadas las dos extremidades inferiores, las dos superiores y la cabeza ese conjunto de pixeles es interpretada como un persona.
Este resultado no siempre es así, y durante todo el video se producen algunos resultados diferentes a los esperados, vamos a analizarlos:

Diferentes posturas

La postura del cuerpo que se va a encontrar en el escenario nuestro AR.Drone es la estándar con las piernas en un ángulo de 30º y los brazos casi pegados al torso. En esta secuencia comprobamos distintos movimientos para ver como se comportaba el algoritmo.

Fig 9.2: Brazos extendidos.

En la figura (Fig 9.2) vemos como con los brazos extendidos todo y que no se detecta correctamente las extremidades superiores la personas es plenamente detectable. Interpretamos que el algoritmo debe considerar que las extremidades están escondidas y no en 90 grados.

Fig 9.3: Brazos extendidos recorrido.

En la figura (Fig 9.3) se aprecia el mismo movimiento pero con recorrido, detectando por el mismo motivo que en la figura tal pero sin comprender el movimiento de las extremidades superiores.

Otro resultado positivo fue la detección de personas en movimiento, en la figura (Fig 9.4) se aprecia la secuencia de una persona corriendo desde el lateral, y en todos los frames de esta secuencia el algoritmo responde bien.

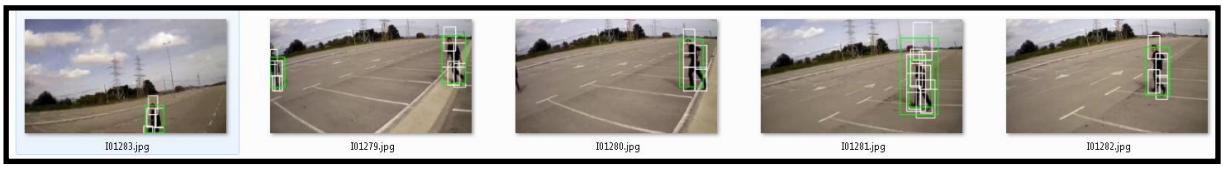


Fig 9.4: Persona en movimiento.

Diferentes escalas

Una situación que se produce muchísimo en la captura de imágenes es encontrarse personas más alejadas que otras desde un mismo punto, hecho que hace que en la imagen se aprecie una persona mucho más pequeña, la más lejana, y otra mucho mas grande. El algoritmo respondió bien y mal a partes iguales ante esta situación.

En la figura (Fig 9.5) vemos un ejemplo de uno de los pocos casos en que funcionó, mientras que en la figura (Fig 9.6) se aprecia lo que se produjo en la mayoría de frames de la secuencia de video, la persona alejada no era bien detectada. Esto se debe a que el parámetro de escala está fijado en un tamaño para no confundir al algoritmo, reajustando este parámetro se podría mejorar el resultado en estos casos.

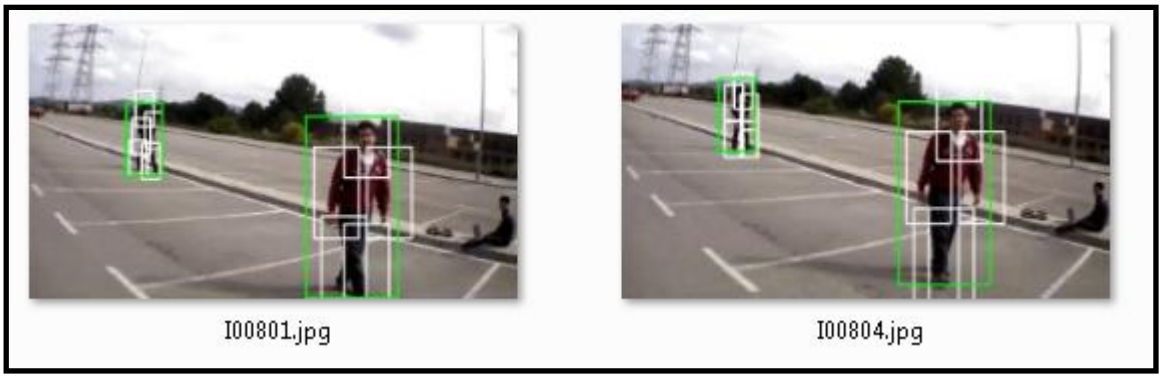


Fig 9.5: Acierto en escala.

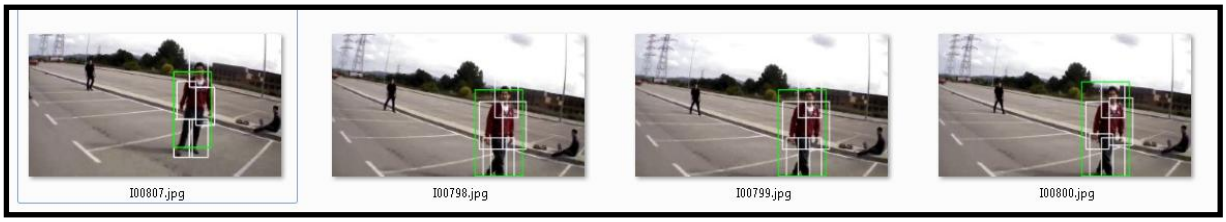


Fig 9.6: Fallo en escala.

Falsos positivos

Se conoce como un error falso positivo, comúnmente llamado falsa alarma, cuando el resultado indica que se han cumplido unas condiciones que realmente no se han cumplido. Esto solo es aplicable en sistemas en que su salida es positiva o negativa. En nuestro caso es aplicable ya que hay, o no hay, una persona en la imagen.

En toda la secuencia de video solo se produjo un falso positivo, fue en la imagen del medio de la figura (Fig 9.7), donde detectó que el background era una persona. En la imagen es apreciable que en el instante en que se capturó el frame el AR.Drone sufrió algún tipo de movimiento incontrolado, que hizo que la imagen se tomara pixelada a causa del movimiento del AR.Drone en el momento en que el obturador estaba abierto.

Considerando los frames anteriores y posteriores se vio que este falso positivo no se repitió en ninguno más, refutando nuestra idea de que la pixelación pueda ser la causa, no un mal funcionamiento del algoritmo. Por lo tanto, si nuestro algoritmo incorpora información temporal, vería que solo en uno de todo el conjunto de frames de esa toma contiene ese error y lo eliminaría.

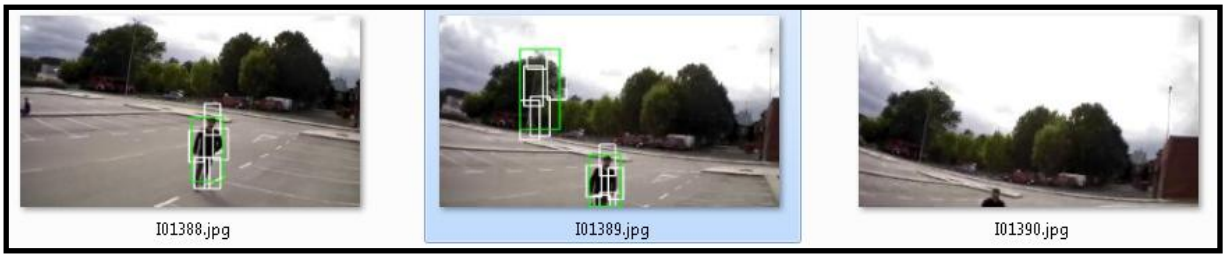


Fig 9.7: Falso positivo.

Errores en detección completa

En muchos frames no se detectaron personas totalmente visibles y en ellas intentamos buscar respuestas del por qué. Estas imágenes fueron algunas de las que todo y haber una persona completamente visible el algoritmo no lo detecto y el porqué se pudo producir ese error. Este caso también incluiría los errores por diferentes escalas pero se han obviado ya que se han analizado en el apartado anterior.

Fig 9.8: Persona sentada.

En la primera imagen de la figura (Fig 9.8) la persona no es detectada, esto sucede ya que la posición del cuerpo se encuentra de manera lateral y las extremidades están muy pegadas al cuerpo, siendo imposible para el algoritmo de detectarlas. Los frames posteriores donde las piernas están más separadas el algoritmo actúa con normalidad y correctamente, corroborando esta suposición. En las dos restantes se aprecia como el ángulo en el que aparece la persona respecto el suelo es muy horizontal, dificultando muchísimo la labor del algoritmo. Esto viene provocado por la poca estabilidad del AR.Drone en el aire, debido a la fuerza del viento. También hay otro punto a destacar en la imagen del medio donde se puede ver como la persona se encuentra sentada en el suelo. La detección es correcta pero el algoritmo interpreta que la parte inferior esta oculta, cuando en realidad esta horizontalmente del suelo.

Errores en detección parcial

Estos fueron algunos casos donde no se detectó correctamente la persona porque aparecía parcialmente en la imagen. Este caso en concreto con información temporal, es decir, de los frames anteriores y posteriores, se podría mejorar mucho este problema, ya que si una persona aparece completamente visible en algún frame, y en el siguiente parcialmente como en estos casos, el algoritmo tiene mucha más información para procesar correctamente

El algoritmo responde bien cuando una la parte superior esta visible pero no la parte inferior.

En la siguiente figura (Fig 9.9) se aprecia algo curioso, el algoritmo solo detecta bien a la persona de la izquierda de la imagen cuando esta es totalmente visible, ya que en el frame anterior está cortada por el margen de la imagen.

Fig 9.9: Persona parcialmente visible horizontalmente.

Para los casos donde la persona no es visible en el eje vertical vemos varios errores con la parte inferior visible (primera imagen de la figura (Fig 9.10)), y a continuación con la parte superior visible. Además hay que agregar la dificultad que el cuerpo se encuentra lateralmente, haciendo que esta pueda ser otra causa de la no detección.

Fig 9.10: Persona parcialmente visible verticalmente.

3.4 Software corrector

En el análisis de las imágenes obtenidas se identificaron varios puntos problemáticos que dificultaban la correcta detección de los algoritmos detectores de personas. Para ello se abordó uno de esos problemas que se presentaban en la toma de imágenes desde el AR.Drone. Este problema es el de la inestabilidad del AR.Drone en el aire que produce que muchas imágenes estén rotadas.

Para ello se ha desarrollado una pequeña y simple aplicación que corrige la rotación de las imágenes respecto el ángulo roll. Esto se consigue gracias a las decisiones del capítulo 3.2 sobre la estructura de la aplicación, donde para una aplicación exterior, es fácil y escalable obtener y decodificar para cada frame los valores de los sensores del AR.Drone en el momento exacto de la toma de imágenes.

Gracias a la estructura del archivo en XML, todo una secuencia de imágenes se trata como un árbol que tiene el nombre de video como raíz y donde cada etiqueta, en este caso nos interesan las <frame>, son los nodos hijos.

En la figura (Fig 10.1) se muestra un ejemplo de este código implementado en Matlab, donde se puede apreciar el tratamiento en estructura de árbol y como se obtienen los atributos.

```
allimages = ls('*.jpg');

try
    DOMnode = xmlread('Parking_REC_003_172137.xml');
catch
    error('Failed to read XML file.');
end

allframes = DOMnode.getElementsByTagName('Roll');
vec = zeros(1,allframes.getLength);
b64_roll_ant = '';

display('Rotating image command roll...');
for k = 0:allframes.getLength-1

    thisframe = allframes.item(k);
    b64_roll = thisframe.item(0).getTextContent;
    if(~(b64_roll==b64_roll_ant))
     y=base64decode(b64_roll.toCharArray,'','matlab');
     v_roll = str2double(char(y));
     b64_roll_ant = b64_roll;
    end
    vec(1,k+1)=v_roll;

     thisimage = allimages(k+1,:);
     im = imread(thisimage);
     c_im = imrotate(im,v_roll);
     outim = strcat(thisimage(1:30),'_rollcheck.jpg');
     imwrite(c_im,outim,'jpg');


end
display('Task Done!');
```

Fig 10.1:Corrector de ángulo.

Al lanzar la aplicación obtendremos en el mismo directorio todas las imágenes corregidas con el mismo nombre más la extensión de rollcheck para identificar que se ha corregido la imagen.

Al analizar los resultados de esta corrección, se aprecia como en las imágenes no resultan realmente tan afectadas por este ángulo, ya que no se alejan en demasiadas unidades al

punto de partida y estabilidad de 0.0. Las nuevas imágenes no se ven en exceso modificadas y no aportan calidad para la detección de personas. En la figura (Fig 10.2) podemos apreciar un ejemplo. En los bordes se puede distinguir un margen negro que se produce al rotar la imagen, pero como el valor del ángulo es tan pequeño apenas se aprecia diferencia..

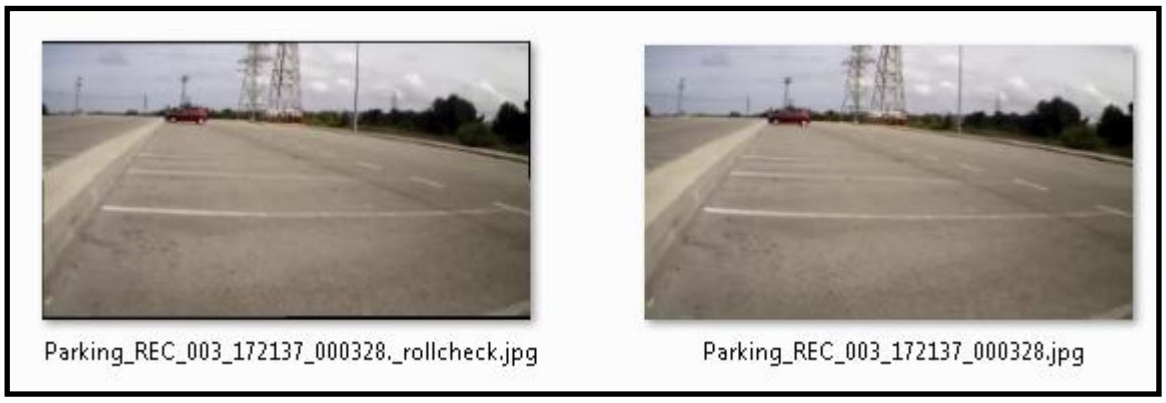


Fig 10.2:Imagen corregida.

Para realizar una rotación más exacta, sería necesario reparar la imagen respecto los tres ángulos disponibles con un sistema similar, no solo con el roll, y analizar los nuevos resultados con los algoritmos de detección de personas, para comparar los resultados.

Capítulo 4:

Conclusiones

4.1 Conclusiones

En primer lugar, el objetivo principal del proyecto se ha visto cumplido. Hemos podido implementar una aplicación para recoger imágenes del AR.Drone y aplicar los algoritmos de detección de personas. Posteriormente se han podido analizar los resultados para obtener algunas conclusiones para la mejora de la detección de personas en UAVs.

La aplicación resulta fiable y no se producen efectos de cuello de botella al tener que almacenar tanta cantidad de datos, lo que podría provocar pérdidas de conexión con el AR.Drone y fallos en la aplicación, se comprobó con secuencias de imágenes superiores a los 5 minutos.

La estructura de los datos de salida es portable y escalable, permitiendo futuras mejoras y adaptaciones a nuevos enfoques u objetivos.

El algoritmo detectó personas sentadas en el suelo como parcialmente visibles. Eso se debe a que como en el eje vertical solo existe la parte de arriba del torso (cabeza, brazos y espalda) la parte de abajo del torso queda en un eje horizontal, por lo que el algoritmo interpreta que esa parte está detrás de un objeto.

Aplicando métodos de información temporal en el algoritmo se podría mejorar mucho el rendimiento, ya que el sistema puede predecir el movimiento de las personas detectadas en los frames anteriores para mejorar la predicción en los posteriores con personas parcialmente visibles.

Debido a la inclinación del AR.Drone muchas personas no fueron detectadas, ya que el algoritmo de detección no ha sido entrenado para detectar personas en posición horizontal. Esto se podría corregir con software de corrección de imágenes, como el que se ha implementado en el proyecto, u otros métodos como adaptar el algoritmo a la detección en horizontal.

Un factor esencial en la detección de personas en las imágenes es que las personas a detectar cumplan con unos valores de escala predefinidos por el programador. Se debe ajustar el valor de la escala, ya que una persona que se visiona en un tamaño muy pequeño pasa desapercibida para el detector.

Los mejores resultados se han observador a una altura entre 3m y 50m. Menos de ese rango de altitud no permite un plano general de la persona y más de ese rango no permitiría ver con claridad las articulaciones de las personas.

A nivel personal como estudiante de Ingeniería Informática de último curso, siempre me ha llamado la atención la tecnología que nos rodea y como esta funciona. Cuando esta tecnología, o el desarrollo de esta, es aplicable directamente a la sociedad para ayudar o facilitar tareas complejas, es cuando todo el trabajo y horas de dedicación cobran sentido.

Escogí este proyecto porque combina diferentes aspectos en los que me he formado durante estos 5 años, empezando desde programación simple en C y C++, incorporar librerías externas para utilizar herramientas eficientes y consistentes, y finalizando por el manejo de datos complejos como son las imágenes con OpenCV. Así pues, con estas tres o cuatro herramientas, se puede desempeñar un proyecto tan bonito y a la vez complejo como pilotar un UAV para obtener y identificar personas en cualquier escenario.

Como se comenta en el apartado 1.1 de esta memoria, aún hay mucho camino por recorrer en la detección de personas mediante vehículos autónomos. Los vehículos terrestres, se encuentran con varios problemas a la hora de acceder al escenario, por eso, una nueva vía de investigación que evita ciertos problemas de estos vehículos, pero introduce otros, como son los vehículos aéreos, resulta motivador y interesante para ver cómo funcionan los algoritmos de detección de personas ya desarrollados en este nuevo entorno.

4.2 Propuestas de futuro

Corrección de imágenes

En el apartado 3.4 de esta memoria se ha presentado un sencillo software que corrige el ángulo de inclinación de las imágenes para compensar la inestabilidad del AR.Drone en el aire, este software solo trabaja con uno de los tres ángulos disponibles.

Si se quisiese maximizar la corrección de las imágenes obtenidas se tendría que optimizar ese software o uno parecido para que incorporara corrección de los ángulos roll y pitch.

También se tendría que corregir la distorsión causada por la lente, como se parecía en algunas imágenes, donde las líneas rectas parecen impresas encima de una esfera. Esto facilitaría mucho el trabajo de los algoritmos de detección de personas.

Inteligencia temporal

Muchos de los errores expuestos en este proyecto en la detección de personas se podrían corregir con inteligencia artificial o sistemas de predicción temporal. Toda la información generada por los frames anteriores sería de gran utilidad para ayudar la predicción de los futuros frames y así incrementar el acierto en imágenes difíciles de identificar.

Referencias

[1] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. CVPR 2005.

[2] N. Dalal, B. Triggs, and C. Schmid. Human detection using oriented histograms of flow and appearance. ECCV, 2006.

[3] C. Wojek, S. Walk, and B. Schiele. Multi-cue onboard pedestrian detection. CVPR 2009.

[4] M. Villamizar, J. Scandaliaris, A. Sanfeliu, and J. Andrade-Cetto. Combining color-based invariant gradient detector with HoG descriptors for robust mage detection in scenes under cast shadows. ICRA, 2009.

[5] Advanced Drive Assistance Systems, Computer Vision Center, UAB, http://www.cvc.uab.es/adas/site/?q=node/8

[6] R. Murphy. Human-robot interaction in rescue robotics. IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews, 34(2):138–153, 2004.

[7] Mykhaylo Andriluka, Paul Schnitzspan, Johannes Meyer, Vision Based Victim Detection from Unmanned Aerial Vehicles, 2010

[8] M. Andriluka, S. Roth, and B. Schiele. Pictorial structures revisited: People detection and articulated pose estimation. CVPR 2009.

[9] L. Bourdev and J. Malik. Poselets: Body part detectors trained using 3d human pose annotations. ICCV 2009.

[10] P. F. Felzenszwalb, D. McAllester, and D. Ramanan. A discriminatively trained, multiscale, deformable part model. CVPR 2008.

[11] Pedro F. Felzenszwalb, Member, IEEE Computer Society, Ross B. Girshick, Student Member, IEEE, David McAllester, and Deva Ramanan, Member, IEEE,Object Detection with Discriminatively Trained Part-Based Models, 2010.

[12] Yi Yang, Member, IEEE, and Deva Ramanan, Member, IEEE, Articulated Human Detection with Flexible Mixtures-of-Parts

[13] Stephane Piskorski, Nicolas Brulez, Pierre Eline, AR.Drone Developer Guide, 2011 v SDK 1.7