Integración de Vídeo Streaming en la Teleoperación del AmigoBot

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Junio 2013

Autor

José Antonio Crespo Toral

Tutores

Jesús Fernando Rodríguez Aragón

Iván Álvarez Navia

Belén Curto Diego

D. Iván Álvarez Navia, Dña. Belén Curto Diego y D. Jesús Fernando Rodríguez Aragón, profesores del Departamento de Informática y Automática de la Universidad de Salamanca.

CERTIFICAN:

Que el trabajo titulado "*Integración de Vídeo Streaming en la Teleoperación del AmigoBot*" ha sido realizado bajo sus supervisiones por **José Antonio Crespo Toral** con DNI **52968553-J**, y constituye la memoria del trabajo realizado para la superación de la asignatura Proyecto fin de grado de la titulación Grado en Ingeniería Informática de esta Universidad.

Y para que así conste a todos los efectos oportunos.

En Salamanca, a 20 de junio de 2013

D. Iván Álvarez Navia	Dña. Belén Curto Diego	D. Jesús Fernando Rodríguez Aragón
Dpto. Informática y Automática	Dpto. Informática y Automática	Dpto. Informática y Automática
Universidad de Salamanca	Universidad de Salamanca	Universidad de Salamanca

Lista de cambios

Versión	Fecha	Descripción
1.0	04/07/2012	Elaboración de la memoria primera versión. Apartados 1,2,3 y 4
1.1	11/07/2012	Elaboración de la memoria primera versión. Apartado 5
1.2	16/07/2012	Revisión de los apartados 1, 2, 3 y 4.
1.3	18/07/2012	Revisión de los apartados 5 y 7.
1.4	23/07/2012	Inserción de imágenes en los apartados realizados
1.5	25/07/2012	Elaboración de la memoria apartados 6,8,9 y 10
1.6	7/08/2012	Inserción de imágenes en los últimos apartados
1.7	20/08/2012	Revisión de la memoria
1.8	25/08/2012	Inserción de formato
1.9	20/06/2013	Realización e inserción de resumen y palabras claves en español e inglés

Resumen

La tecnología busca solucionar problemas y necesidades individuales y colectivas del ser humano mediante la construcción de sistemas técnicos. Resulta indudable la aceleración que se ha producido en el desarrollo tecnológico durante el siglo XX y XXI.

Algunos de los avances más significativos se pueden observar en el ámbito de la telefonía, construyendo dispositivos móviles cada vez más potentes y ligeros. Este hecho junto al desarrollo y avance de la robótica, que es la ciencia que se ocupa del estudio, desarrollo y aplicaciones de los robots, forman las bases del proyecto que aquí se presenta.

La elección de estos temas, por tanto, se debe al gran auge de los dispositivos móviles en la sociedad actual y a que la robótica es un campo muy interesante que cuenta con una gran proyección de futuro.

En el ámbito de los dispositivos móviles existen grandes plataformas como son Windows Phone OS, Android e iOS y en este proyecto se ha optado por trabajar con el sistema y dispositivos iOS de Apple, ya que son considerados uno de los más extendidos y potentes en el mercado actual. Otra razón más de esta elección es aprovechar la oportunidad que la Universidad de Salamanca nos ofrece para trabajar con estas herramientas que de manera habitual no están al alcance de todos, como son el robot Amigobot, dispositivos iOS, pc Apple con MAC OS con un IDE (*Integrated Development Environment*, «Entorno de desarrollo integrado») con el que poder programar, etc.

El principal objetivo del proyecto es implementar un software que permita la navegación de un robot Amigobot a través de códigos QR mediante el uso de dispositivos iOS. Esto se realiza gracias al envío de las imágenes, capturadas por una cámara conectada al PC Debian que controla al robot, al dispositivo iOS que realizará un tratamiento de dichas imágenes.

El software de este dispositivo iOS sirve como interfaz de usuario, permitiendo controlar al robot mediante un *joystick* situado en la pantalla táctil para dirigirlo hacia algún código QR y leer su contenido a través del tratamiento de las imágenes de la webcam. En caso de que dicho QR contenga una dirección, el robot de manera autónoma, se dirigirá hasta la posición indicada utilizando un sistema de localización basado en la odometría de las ruedas del robot. Además, para salvar o remediar en cierta medida los grandes errores de la odometría, se realiza un segundo tratamiento de las imágenes recibidas para situar al robot siempre en una posición centrada que le permita leer el siguiente código QR.

Esto puede derivar en una gran cantidad de utilidades en múltiples y diferentes ámbitos. Por ejemplo, podría emplearse en el área de vigilancia y seguridad como cámara móvil, es decir, el robot realizaría la navegación mediante QR mientras que el

guardia observa el recorrido a través del dispositivo, pudiendo teleoperar en caso de que observara algo extraño. También podría utilizarse para que el robot (en una oficina o museo) guiara al usuario hasta un despacho o escultura. Éstas son sólo algunas ideas y aplicaciones útiles del proyecto.

El ámbito del proyecto nos ha hecho trabajar y profundizar con varios *frameworks* y programas:

- La biblioteca OpenCV (*Open Source Computer Vision*), nos servirá para realizar el tratamiento de las imágenes recibidas en el iPhone. La biblioteca está escrita en C++, aunque en el proyecto está adaptada a Objective-C para poder usarla en la programación de dispositivos iOS.

- Los *frameworks* de Cocoa Touch de Apple, nos sirven para la implementación de la aplicación en dispositivos iOS y están escritos en Objective-C. Su IDE, XCode 4, cuenta con importantes mejoras con respecto a sus predecesores.

- La biblioteca *Zxing*, nos servirá para realizar la decodificación de los códigos QR en el iPhone. Está escrita para varios lenguajes, entre ellos Objective-C.

- La biblioteca ARIA *(Advanced Robot Interface for Applications*) de *MobileRobots*, se utilizará para el desarrolllo del servidor que transmitirá las órdenes al Amigobot. Esta biblioteca está escrita en C++.

- El programa *MJPG-Streamer* para distribuciones Linux, que utilizaremos de servidor para que obtenga las imágenes de la webcam y las transmita a través de HTTP (*Hypertext Transfer Protocol*, «protocolo de transferencia de hipertexto») al cliente.

- Se ha creado un protocolo propio para la comunicación entre el servidor de imágenes, que se encuentra en el ordenador que controla el Amigobot, y el cliente iOS.

Durante el desarrollo del proyecto estudiaremos y profundizaremos en estos dos lenguajes de programación orientada a objetos: Objective-C y C++.

Para finalizar, se consideran superados los objetivos técnicos y de documentación del software a construir, al igual que los objetivos personales que se marcaron al comienzo del proyecto. La construcción del proyecto, así como la realización de toda la documentación, me ha servido para afianzar y aplicar todos los conceptos teóricos y prácticos adquiridos a lo largo de la carrera así como para prepararme para la vida y mercado laboral.

Summary

The technology seeks to solve problems and individual and collective needs of human by building technical systems. The acceleration that has occurred in the technological development during the twentieth and twenty-first century is clearly visible.

Some of the most significant developments may be observed in the field of mobile communications, where more and more powerful and lightweight mobile devices are built. This fact together with the development of robotics, which is the science that deals with the study, development and application of robots, form the basis of the current project.

Therefore, the choice of these fields is due to the boom of mobile devices nowadays and because of robotics is a very interesting field that has a future with great prospects.

Great platforms like Windows Phone OS, Android and iOS exist in the field of mobile devices. Apple system and Apple's iOS devices has been chosen to work with in this Project since they are considered one of the most widespread and powerful in the current market. Another reason for this choice is the opportunity that the University of Salamanca offers to work with these tools that are not available to everyone, such as the Amigobot robot, iOS devices, Apple MAC OS computer with an IDE (*Integrated Development Environment*) with which to program, etc.

To implement a software that allows navigation of an Amigobot robot through QR codes by using iOS devices is the main objective of this project. The onboard camera, connected to the Debian computer that controls the robot, will capture images at every time instant. The images will be sent to the iOS device that will perform the image processing.

The software of the iOS device is used as the user interface. It allows theuser to control the robot using a joystick on the screen in order to send the navigation orders to the robot to reach a QR code. Once the robot is in front of a QR code, it will be able to read its content using the captured images. In case the QR code contains information about a goal position, the robot will perform an autonomous navigation using the information of Wheel odometry for its localization. A second image processing is performed to place the robot in a centered position respect to the next QR code. This mitigates the large odometry errors and improves the performance of the localization method, functioning as a correction every time a QR code is found.

This can result in a lot of applications in multiple and different environments. For example, it can be used in a security and surveillance area as a mobile camera, i.e., the robot will perform an autonomous navigation using QR codes while the guard observes the path through a device. The robot can be teleoperated in any moment if something strange happens or appears in the captured images. Another application may be as a

guide in an office environment or a museum. The robot may guide the user to a department or a work of art in particular.

The area of the project made us work and deepen with several frameworks and programs:

- The *OpenCV* library (*Open Source Computer Vision*) is used for the processing of the images received in the iPhone. The library is written in C++, although it has been adapted to Objective-C in this project so it can be used for iOS devices programming.

- The Apple *Cocoa Touch* frameworks have been used for implementation and deployment in iOS devices and they are written in Objective-C. The IDE, XCode 4, has significant improvements over previous versions.

- The *Zxing* library is used for decoding QR codes on the iPhone. It is written for several languages, including Objective-C.

- The MobileRobots *ARIA* library (*Advanced Robot Interface for Applications*) is used for the implementation of a server that will send the navigation orders to the Amigobot. It is written in C + +.

- *MJPG-Streamer* is used as the image server. Images are captured and sent to the iOS device using HTTP protocol.

- A custom protocol has been created for the communication between the Amigobot server, which is deployed on the onboard computer of the Amigobot, and the iOS device.

During the project development we will study and will deepen in these two object-oriented programming languages: Objective-C and C + +.

To conclude, I have overcome the technical objectives and build the software documentation, as well as personal objectives that were marked at the beginning of the project. The development of the project and its documentation has helped me to consolidate and apply all the theoretical and practical concepts acquired during the career as well as to prepare myself for life and labor market.

Palabras clave

- iOS
- Teleoperación
- Robot Amigobot
- Transmisión de vídeo en tiempo real
- Códigos QR
- Odometría

Keywords

- iOS
- Teleoperation
- Robot Amigobot
- Video Streaming
- QR Codes
- Odometry

Tabla de contenidos

Lista de figuras

Lista de tablas

1. Introducción

En este primer apartado de la memoria se pretende hacer una descripción general del tema que abordará el proyecto y de la forma en que quedará estructurada dicha memoria.

1.1 Presentación del tema

El proyecto se ha realizado con carácter meramente docente y sus destinatarios finales son los miembros del Tribunal de Evaluación del TFG (Trabajo Fin de Grado), así como los mencionados tutores y cualquier otro estudiante que pueda aprovecharse del mismo únicamente con fines académicos.

A la hora de elegir el proyecto me propuse como objetivos hacer de él algo entretenido, divertido y no un trabajo excesivamente pesado. Tenía claro hacia dónde quería orientarlo, deseaba algún tipo de software para dispositivos móviles y me interesaba el área de la robótica, ya que me parecen campos interesantes e intrigantes.

En un primer momento, la elección del proyecto fue crear una aplicación para dispositivos iOS que controlara el vuelo y la recepción de imágenes del *Quadracopter* de Parrot. Pero, al comenzar, tuvimos que desecharlo debido a fallos en la funcionalidad y operatividad del Quadracopter con la aplicación original de Parrot (FreeFlight 1.0).

Tras unas semanas, elaboré en colaboración con mis tutores, una nueva propuesta para el proyecto "*Integración de Vídeo Streaming en la Teleoperación del AmigoBot*" sin desviarme de los temas antes mencionados. El objetivo en un principio, como el propio título nos indica, fue poder recibir vídeo stream de una webcam conectada al pc Debian y realizar un pequeño tratamiento de las imágenes. Tras entregar la propuesta, en colaboración con los tutores, el propósito principal del proyecto se fue desviando haciendo de la integración de vídeo streaming en la teleoperación del robot, una herramienta más para conseguir un objetivo más completo y elaborado. El nuevo objetivo del Trabajo de Fin de Grado se convertía de esta manera en la implementación de un software que permitiera la navegación de un robot, el Amigobot de *Mobile Robots*, a través de códigos QR mediante el uso de dispositivos iOS, integrando así el objetivo anterior. Se intentó cambiar el título del proyecto a "QrNav: Navegación de un robot a través de códigos QR usando un dispositivo iOS" pero debido a problemas administrativos no se pudo realizar el cambio.

Este software sirve como interfaz de usuario, permitiendo controlar al Amigobot (robot) mediante un *joystick* situado en la pantalla táctil del dispositivo iOS y obtener las imágenes de una webcam conectada al pc (con distribución Linux) para dirigirlo hacia algún código QR y poder leer su contenido. En caso de que dicho QR contenga una dirección, el robot de manera autónoma, se dirigirá hasta la posición indicada gracias a la odometría del mismo. Además, para salvar o remediar en cierta medida los grandes errores de la odometría, se realiza un tratamiento de las imágenes recibidas para situar al robot siempre en una posición centrada que le permita leer el código QR. El

software también dispone de un sistema de opciones persistente que permite mejorar el manejo del Amigobot y algún aspecto de la interfaz al gusto del usuario.

El proyecto, por lo tanto, permite a cualquier autómata realizar una navegación a través de códigos QR. Esto puede tener una gran cantidad utilidades en múltiples y diferentes ámbitos, como por ejemplo, podría emplearse en el área de vigilancia y seguridad como cámara móvil, es decir, el robot realizaría la navegación que los QR le indicaran mientras el guardia observaría a través del dispositivo la sala, pudiendo teleoperar en caso de que observara algo extraño. También podría utilizarse para que el robot (en una oficina o museo) guiara al usuario hasta un despacho o escultura. Éstas son sólo algunas de las muchas ideas y aplicaciones para las que podría ser útil el proyecto.

El ámbito del proyecto nos ha hecho trabajar y profundizar con varios *frameworks* y programas:

- La biblioteca OpenCV (*Open Source Computer Vision*), nos servirá para realizar el tratamiento de las imágenes recibidas en el iPhone. La biblioteca está escrita en C++, aunque en el proyecto está adaptada a Objective-C para poder usarla en la programación de dispositivos iOS.

- Los *frameworks* de Cocoa Touch de Apple, servirán para la implementación de la aplicación en dispositivo iOS y están escritos en Objective-C. Su IDE (*integrated development environment*, «Entorno de desarrollo integrado»), XCode 4, cuenta con importantes mejoras con respecto a sus predecesores.

- La biblioteca *Zxing*, nos servirá para realizar la decodificación de los códigos QR en el iPhone. Está escrita para varios lenguajes, entre ellos Objective-C.

- La biblioteca ARIA *(Advanced Robot Interface for Applications*) de *MobileRobots*, se utilizará para el desarrollo del servidor que transmitirá las órdenes al Amigobot. Está desarrollada bajo el sistema operativo Debian aunque es compatible con cualquier distribución Linux. Esta biblioteca está escrita en C++.

- El programa *MJPG-Streamer* para distribuciones Linux, que utilizaremos de servidor para que obtenga las imágenes de la webcam y las transmita a través de HTTP (*Hypertext Transfer Protocol*, «protocolo de transferencia de hipertexto») al cliente.

- Un protocolo que implementa un cliente *HTTP* para recibir el flujo de imágenes del servidor.

Durante el desarrollo del proyecto estudiaremos y profundizaremos en estos dos lenguajes de programación orientada a objetos: Objective-C y C++.

1.2 Contenido y recursos de la memoria

Previamente a la fase de implementación se llevará a cabo el diseño del software siguiendo una adaptación de Proceso Unificado, a partir de la cual se crearán una serie de documentos (además de éste) que conformarán la memoria del proyecto, así como otros artefactos del software propios de la implementación.

Se ha tomado como referencia para realizar la documentación de este proyecto: Guía de realización y documentación del Proyecto de Fin de Carrera en ITIS (Ingeniería Técnica en Informática de Sistemas) de García Peñalvo, Maudes Raedo, Piattini Velthuis, García-Bermejo Giner y Moreno García. Versión 1.52 del 2000, así como el documento que especifica la definición de las normas de estilo, extensión y estructura del TFGM.

Se han seguido las directrices que proporcionaba dicho documento adaptando aquellas que eran relevantes y fundamentales para una aplicación software de mis características.

De acuerdo con esta guía, la memoria consta de dos partes: Descripción del proyecto y Documentación técnica.

- **Descripción del proyecto:**

  - *Introducción*: En este apartado inicial se plantea el tema que concierne al proyecto del que se está documentando. Se realiza una descripción concisa y clara de la principal finalidad del proyecto y su utilidad en el entorno actual. Además se detalla la estructura de la memoria, especificando en cada apartado qué se describe.

  - *Objetivos del proyecto*: En este punto vamos a establecer cuáles son los objetivos fundamentales que la aplicación plantea haciendo referencia, a modo de introducción, a los requisitos de software necesarios para el sistema. Con esta información conseguiremos comprender mejor la funcionalidad que tendrá la aplicación y qué es lo que se ha buscado con el desarrollo de dicho proyecto software.

    Claro está que esto es sólo una pequeña anotación, pues en los anexos que comprenden esta documentación podremos conocer de manera detallada cada aspecto de los objetivos y de los requisitos funcionales y de información, entre otras cuestiones.

  - *Conceptos teóricos*: La razón de este apartado es lograr una mejor comprensión para el usuario si no está familiarizado con el tema que se trata en este proyecto. De esta forma, definiremos los ámbitos más importantes que se han de conocer para comprender la gestión.

  - *Técnicas y Herramientas*: Este apartado abarcará más temas que los anteriores, pues se definirán cada una de las técnicas metodológicas

empleadas durante el desarrollo, así como las herramientas que se han utilizado para la generación del sistema software final. Se proporcionará una explicación breve de cada una de ellas, indicando su finalidad en el desarrollo y su importancia. Estarán incluidos en este apartado, por lo tanto, los lenguajes de programación, bibliotecas y entornos de desarrollo que se han aprendido para la realización de este proyecto.

- *Aspectos relevantes del desarrollo del Proyecto*: Este apartado constará de tres secciones muy diferenciadas. Por un lado, se indicarán las partes que conforman el desarrollo, es decir, el ciclo de vida, especificando cada fase de análisis, diseño e implementación. Por otro lado, se comentará la experiencia práctica aprendida durante la elaboración de este proyecto, donde se destacarán de manera personal los aspectos más relevantes del aprendizaje. Y, por último, completará este apartado la descripción del sistema, que engloba todos aquellos aspectos que durante la realización del sistema software han sido problemáticos y son destacables por su mayor dificultad.

- *Trabajos relacionados*: En este apartado se comentarán los trabajos y proyectos ya realizados en el campo del proyecto en curso.

- *Conclusiones y líneas de trabajo futuro*: En este capítulo incluiremos los resultados y opiniones que se derivan de su desarrollo, así como las posibles líneas de mejora y de trabajo futuro.

- *Glosario*: Se definen y comentan ciertos términos utilizados en la memoria.

- *Bibliografía*: Recogerá el conjunto de documentación que se ha consultado para la construcción de este sistema o los libros de los que se ha obtenido información importante para el desarrollo o la implementación.

❖ **Documentación técnica**: Documentos derivados del ciclo de vida del proyecto software.

- *Anexo 1. Plan del Proyecto Software*: Esta documentación se dirigire principalmente hacia un punto en concreto, la planificación temporal del proyecto o lo que es lo mismo la elaboración del calendario o programa de tiempos. El principal objetivo de esta tarea es recoger de forma gráfica todas las actividades necesarias para producir el resultado final, así como la definición del mismo en función de los documentos y demás artefactos requeridos para la entrega del proyecto.

- *Anexo 2. Especificación de Requisitos del Software*: En este anexo se especificarán con cierto nivel de detalle cada uno de los objetivos que se alcanzan en el sistema, así como los requisitos funcionales, no funcionales y de información que definen la creación del software. Todo esto conllevará a la utilización de métodos como casos de uso o diferentes diagramas que completarán las especificaciones detalladas en las tablas REM. Esta parte se denominará Dominio del Problema.

- Anexo 3. Especificación de Diseño: Este anexo se basará en la fase de diseño, donde se concretarán cada una de las soluciones a los problemas planteados durante la recopilación de requisitos software. Esta fase es fundamental para una posterior implementación óptima, pues en el diseño se plantean cada una de las necesidades reales que se solicitan desde la parte de análisis. Esta parte se denominará, por tanto, Dominio de la Solución. El Anexo 3 seguirá la plantilla establecida en la Documentación Técnica, según la Guía de Realización y Documentación adaptada al paradigma orientado a objetos.

- *Anexo 4. Documentación Técnica de Programación*: Este documento nos servirá de guía para mostrar cada una de las clases y paquetes que componen la implementación de nuestro sistema. Además, se especificarán los métodos y funcionalidades de las que se encarga cada módulo de implementación y será un refuerzo para comprender mejor el código fuente de cara a futuros desarrolladores que necesiten nuestro sistema.

- *Anexo 5. Manuales de Usuario*: Este último anexo se centra en una guía para el usuario del sistema. De tal forma que se detallarán los conceptos necesarios para el uso sencillo de la aplicación, con imágenes de ejemplo y especificaciones de cada funcionalidad.

2. Objetivos del proyecto

2.1 introducción

Este apartado pretende detallar de manera concisa los objetivos que se persiguen con la realización del proyecto (distinguiendo entre los objetivos técnicos y de documentación), una breve introducción a los requisitos generales del software a construir y los objetivos personales.

2.2 Objetivos técnicos y de documentación

Se definen a continuación una serie de conocimientos necesarios previos al desarrollo del código:

- Aprender y profundizar en el lenguaje de programación Objective-C, usado por el SDK (*software development kit*, «Kit de desarrollo de software») de Apple para dispositivos iOS.
- Aprender y profundizar en el lenguaje de programación C++, usado en nuestro caso por diversas bibliotecas.
- Nuevo paradigma de desarrollo software, Orientación a Objetos. Objective-C y C++ siguen esta filosofía de objetos y clases.
- Familiarizarse con XCode 4, único entorno posible que permite desarrollar aplicaciones para dispositivos iOS.
- Utilización del *framework* Cocoa Touch, una API (*Application Programming Interface*, «Interfaz de programación de aplicaciones») para la creación de aplicaciones para iPad, iPhone e iPod Touch. Basado en el conjunto de herramientas que proporciona la *API* de *Cocoa* para crear programas sobre la plataforma Mac OS X. Por tanto, también se estudiará la programación para dicha plataforma, además de para los dispositivos iOS.
- Utilización de la biblioteca *OpenCV* para iOS, para realizar el tratamiento de las imágenes en el dispositivo.
- Utilización biblioteca *Zxing* para iOS, para realizar la decodificación de los códigos QR.
- Utilización de la biblioteca *ARIA* de MobileRobots, que ofrece una *API* para acceder a las características del Amigobot y realizar operaciones con él, bajo alguna distribución Linux (en nuestro caso Debian).
- Uso de *MJPG-Streamer* para realizar una transmisión de vídeo streaming en tiempo real entre un pc con distribución Linux y un dispositivo iOS.

2.3 Objetivos generales del software a construir

El Trabajo Fin de Grado consiste en una aplicación desarrollada para dispositivos iOS que permita la navegación a través de códigos QR del robot Amigobot.

Durante el proceso de desarrollo se pretenden conseguir una serie de objetivos:

- Crear tres aplicaciones:
  - Una aplicación (cliente) que servirá de interfaz para el usuario y que deberá realizar diversas operaciones. Será el encargado de decidir qué proceso debe ejecutarse en cada momento en función de los valores que obtenga y enviar los datos al servidor. También actuará como cliente para recibir las imágenes que el servidor (*MJPG-Streamer*) capta de la webcam en tiempo real. Esta aplicación funcionará en dispositivos iOS.
  - Un programa (servidor) que se utilizará para recibir los datos enviados por la aplicación y para controlar al robot en función de esos datos.
  - Otro programa que realizará la función de servidor streaming para el envío de un flujo continuo de imágenes a través de *HTTP*.
- Estas tres aplicaciones seguirán el conocido patrón de diseño Cliente-Servidor, en este caso un Cliente-Servidor de dos capas.
- Las dos primeras aplicaciones se desarrollarán siguiendo el paradigma de orientación a objetos.
- El desarrollo seguirá una adaptación del Proceso Unificado de la Ingeniería del Software.
- La aplicación cliente:
  - Se desarrollará en Mac OS X Lion en el *IDE* XCode 4 con el *SDK* 5.1 en el lenguaje Objective-C.
  - Se desarrollará para dispositivos iOS, centrándose en crear una interfaz útil, sencilla e intuitiva y gráficamente trabajada que permite:
    - Controlar la teleoperación del robot.
    - Leer un código QR de las imágenes recibidas del servidor mediante biblioteca *Zxing*.
    - Poner en marcha la recepción de vídeo y pausarla.
    - La desconexión con el robot en caso de emergencia.

- Mostrar en todo momento el estado del robot.

- Contendrá un sistema de opciones persistentes con el que podremos mejorar el manejo del robot con diferentes características y editar algún aspecto gráfico de la interfaz.

- Será capaz de realizar un tratamiento de las imágenes obtenidas buscando un cuadrado rojo alrededor del QR para colocar al robot y, posteriormente, leer un código QR mediante la biblioteca *OpenCV*.

❖ La aplicación servidor:

- Se desarrollará para GNU/Linux (*GNU is not Unix*, «GNU no es Unix»), en concreto, para la distribución Debian y en el lenguaje C++.

- Usará la biblioteca *ARIA* para la comunicación con el robot.

- Ordenará y actuará en función de los parámetros recibidos.

- Conducirá al robot hasta la posición recibida.

- Colocará al robot para realizar la lectura en función de los parámetros obtenidos al realizar el tratamiento.

❖ Servidor para la transmisión de vídeo en tiempo real mediante el programa *MJPG-Streamer*.

Esta breve introducción a los objetivos del software tiene como objetivo inciar al lector en el ámbito del proyecto que se pretende desarrollar. Para más información consultar el Anexo 2 "*Especificación de Requisitos del Software*", incluido junto a este documento en la memoria del proyecto.

2.4 Objetivos personales

❖ Superar la asignatura Trabajo Fin de Grado para poder obtener el título de Grado en Ingeniería informática.

❖ Sumergirme y profundizar en un mundo totalmente nuevo, como es la programación para dispositivos iOS que me servirá en un futuro próximo para el desarrollo de nuevas aplicaciones.

❖ Aprender y profundizar en los lenguajes de programación Objective-C y C++.

❖ Comprender y afianzar el paradigma orientado a objetos. A pesar de que se haya estudiado en la carrera, los conocimientos sobre él eran muy básicos.

- Evaluar mi capacidad para aprender un lenguaje y desarrollar un sistema software completo, desde cero, con un grado de dificultad mayor al de las materias de la carrera.
- Obtener capacidad de análisis temporal de los proyectos de desarrollo de software, algo importante para el mundo profesional.
- Capacidad de creación y desarrollo de un protocolo de comunicación.
- Obtener la capacidad de trabajar con un robot y profundizar en la rama de robótica.

3. Conceptos teóricos

3.1 Introducción

Este apartado está dedicado a describir una serie de conceptos teóricos necesarios para la comprensión del proyecto y del resto de la memoria.

Esta sección se va a separar en dos apartados, el primero tratará conceptos teóricos generales y necesarios para la comprensión de esta memoria y de los anexos adjuntos. La mayor parte del público al que va dirigido este documento ya conocerá dichos conceptos, pero con el objetivo de ayudar a posibles lectores no familiarizados con algunos de estos conceptos se hará una breve descripción.

El segundo apartado será una explicación de las partes técnicas correspondientes a los *frameworks y* conceptos usados.

3.2 Conceptos teóricos generales

3.2.1 Apple. iOS y sus dispositivos. Cocoa Touch.

Apple Inc. es una empresa multinacional estadounidense con sede en California, que diseña y produce equipos electrónicos y software. Entre los productos de hardware más conocidos destacan los equipos Macintosh, el iPod, el iPhone y el iPad. Entre el software de Apple se encuentran el sistema operativo Mac OS X, el sistema operativo iOS, el explorador de contenido multimedia iTunes, la suite iLife (software de creatividad y multimedia), la suite iWork (software de productividad), el navegador web Safari y un gran cantidad más de software.

El sistema operativo iOS (anteriormente denominado iPhone OS) es un sistema operativo móvil de Apple. Originalmente desarrollado para el iPhone, siendo después usado en dispositivos como el iPod Touch, iPad y el Apple TV. Apple Inc. no permite la instalación de iOS en hardware de terceros.

Algunas de sus características son:

- La pantalla principal (llamada «*SpringBoard*»), donde se ubican los iconos de las aplicaciones y el *Dock* que es la parte inferior donde se pueden anclar aplicaciones de uso frecuente. La pantalla tiene una barra de estado en la parte superior para mostrar datos, tales como la hora, el nivel de batería y la intensidad de la señal y el resto está dedicado a la aplicación actual.

- Con la actualización iOS 5 el sistema de notificaciones se rediseñó por completo. Las notificaciones ahora se colocan en un área a la que se puede acceder mediante un desliz desde la barra de estado hacia abajo. Al hacer un toque en una notificación, el sistema abre la aplicación que envió esta notificación.

- Antes de iOS 4, la multitarea estaba reservada para aplicaciones por defecto del sistema. A Apple le preocupaba los problemas de batería y rendimiento si se permitiese correr varias aplicaciones de terceros al mismo tiempo. A partir de iOS 4, dispositivos de tercera generación y posteriores permiten el uso de 7 *APIs* para multitarea específicamente:
  - Audio en segundo plano
  - Voz IP
  - Localización en segundo plano
  - Notificaciones push
  - Notificaciones locales
  - Completado de tareas
  - Cambio rápido de aplicaciones

Sin embargo, no consiste en una verdadera multitarea, pues las aplicaciones ajenas al sistema operativo quedan congeladas en segundo plano no recibiendo un solo ciclo de reloj del procesador.

- Game Center es una aplicación preinstalada que permite encontrar adversarios o amigos en función de tus juegos y jugadores favoritos.
- Tecnologías no admitidas. iOS no permite Adobe Flash ni Java. Steve Jobs escribió una carta abierta donde critica a *Flash* por ser inseguro, con errores, consumir mucha batería, ser incompatible con interfaces multitouch e interferir con el servicio App Store. En cambio iOS usa HTML5 (*HyperText Markup Language*, «lenguaje de marcado de hipertexto») como una alternativa a *Flash*. Esta ha sido una característica muy criticada tanto en su momento como en la actualidad. Sin embargo por métodos extraoficiales se le puede implementar aunque conllevaría la pérdida de la garantía.

Entre los dispositivos que usan iOS podemos encontrar:

- **iPhone**: es una familia de teléfonos inteligentes multimedia con conexión a Internet, pantalla táctil capacitiva y escasos botones físicos. Carecen de un teclado físico pero integran uno en la pantalla táctil con orientaciones tanto vertical como horizontal.
- **iPad:** es un dispositivo electrónico tipo Tablet desarrollado por Apple Inc. Se sitúa en una categoría entre un "teléfono inteligente" (*smartphone*) y una computadora portátil, enfocado más al acceso que a la creación de aplicaciones y temas.

Las funciones son similares al resto de dispositivos portátiles de Apple, aunque la pantalla es más grande y su hardware más potente. Funciona a través de una NUI (Interfaz natural de usuario) sobre una versión adaptada del sistema operativo iOS. Esta interfaz de usuario está rediseñada para aprovechar el mayor tamaño del dispositivo y la capacidad de utilizar software.

- **iPod Touch**: es un reproductor multimedia, PDA, videoconsola y plataforma móvil Wi-Fi. Sus funciones son bastantes similares al iPhone pero no dispone de SIM y, por tanto, de aplicaciones derivadas del uso de la SIM (Teléfono, Mensajes, etc.).

Cocoa Touch es la capa más alta en la programación de dispositivos iOS, los provee de las utilidades de desarrollo presentes en Mac OS X, añadiendo partes nuevas y poniendo especial énfasis en las interfaces multitouch y la optimización necesaria en estos dispositivos.

EL *framework* principal en cualquier aplicación para dispositivos iOS es UIKit. Éste provee de las herramientas básicas necesarias para implementar aplicaciones gráficas en iOS. UIKit ofrece acceso a controles gráficos, como botones, vistas para los dispositivos iOS…

Aparte de UIKit, el Cocoa Touch ofrece una colección de *frameworks* que incluyen todo lo necesario para crear aplicaciones con gráficos 3D, audio profesional, conectividad, control de las características del dispositivo... Algunos de estos *frameworks* son los siguientes:

- CoreData
- AudioToolBox
- Foundation
- CFNetwork
- CoreGraphics
- Etc

3.2.2 Ingeniería del software. Proceso Unificado

Podemos encontrar diversas definiciones sobre qué es la Ingeniería del Software:

- Es el establecimiento y uso de principios sólidos de ingeniería, orientados a obtener software económico que sea fiable y trabaje de manera eficiente en máquinas reales.

- Es la aplicación de herramientas, métodos y disciplinas de forma eficiente en cuanto al coste, para producir y mantener una solución a un problema de procesamiento real automatizado total o parcialmente por el software.

Todas estas definiciones quieren decir lo mismo con distintas palabras, por lo que podemos concluir que la Ingeniería del Software es la disciplina que ofrece métodos y técnicas para desarrollar y mantener software. La Ingeniería del Software trata de sistematizar este proceso con el fin de tener un menor riesgo de fracaso.

Consiste en aplicar, en la construcción y desarrollo de proyectos, métodos y técnicas para resolver los problemas que surgen en todas las etapas de la creación de un proyecto software.

Proceso Unificado es un proceso de desarrollo de software: "Conjunto de actividades necesarias para transformar los requisitos del usuario en un sistema software". También podríamos definirlo como un marco de trabajo genérico que puede especializarse para una variedad de tipos de sistemas, diferentes áreas de aplicación, tipos de organizaciones, niveles de aptitud y diferentes tamaños de proyectos.

Las características principales del Proceso Unificado son:

- **Dirigido por casos de uso.** Un caso de uso representa una pieza de funcionalidad en el sistema que le devuelve al usuario un resultado de valor. Los casos de uso sirven para capturar requerimientos funcionales. Todos los casos de uso de un sistema conforman el modelo de casos de uso. Se dice que el Proceso Unificado está manejado por casos de uso porque el desarrollador creará modelos de implementación y diseño que harán efectivos a los casos de uso planteados en la etapa de análisis. Cada modelo del proceso tendrá una correspondencia con el modelo de casos de uso.

- **Iterativo e incremental.** Para el Proceso Unificado la vida de un sistema se encuentra dividida en ciclos que terminan con un lanzamiento de diferentes modelos del producto. Cada ciclo se divide en cuatro fases: concepción, elaboración, construcción y transición. Cada fase se encuentra subdividida en iteraciones. Al final de cada fase se produce un "hito" o punto de revisión. Una iteración es un mini proyecto que concluye con la entrega de algún documento o implementación interna. Dentro de cada iteración se realizan actividades de captura de requerimientos, análisis, diseño, implementación y prueba.

- **Centrado en la arquitectura.** El Proceso Unificado asume que no existe un modelo único que cubra todos los aspectos del sistema. Por dicho motivo existen múltiples modelos y vistas que definen la arquitectura de software de un sistema. Se toma como arquitectura de referencia el modelo de arquitectura de 4+1 vistas de Philipe Kruchten, en el cual cada vista es una parte del modelo:

- *La vista de Casos de Uso.* Describe secuencias de interacciones que se desarrollarán entre un sistema y sus actores en respuesta a un evento que inicia un actor principal sobre el propio sistema.

- *La vista Lógica.* Describe la descomposición del sistema en una serie de abstracciones clave, tomadas del dominio del problema en forma de objetos o clases de objetos. También identifica mecanismos y elementos de diseño comunes a diversas partes del sistema.

- *La vista de Procesos.* Describe el diseño de los aspectos de concurrencia y sincronización.

- *La vista de Despliegue.* Describe el ordenamiento físico de los recursos computacionales, como computadores y sus interconexiones (nodos) en tiempo de ejecución.

- *La vista de Implementación.* Describe el empaquetamiento físico de las partes reutilizables del sistema llamadas componentes, es decir, muestra la implementación de los elementos de diseño.

3.2.3 Patrones de Diseño

Podemos definir un patrón como una regla que establece una relación entre un contexto, un sistema de fuerzas que aparecen en dicho contexto y una configuración.

Los patrones de diseño son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces.

Un patrón de diseño es una solución a un problema de diseño. Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que se debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser reutilizable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.

Existen diferentes clasificaciones de los patrones de diseño como son:

- **POSA**. Clasifican los patrones según la categoría del patrón y la categoría del problema como se muestra a continuación en la Figura 1.

POSA	Arquitectónicos	Diseño	Idiomas
Estructuración	*Layers; Pipes&Filters; Blackboard*		
Sistemas distribuidos	*Broker* *Pipes&Filters* *Microkernel*		
Sistemas interactivos	*MVC; PAC*		
Sistemas adaptables	*Microkernel, Reflection*		
Descomposición estructural		*Whole-Part*	
Organización del trabajo		*Master-Slave*	
Control acceso		*Proxy*	
Gestión		*Command Processor* *View Handler*	
Comunicación		*Publisher-Subscriber* *Forwarder-Receiver* *Client-Dispatcher-Server*	
Manejo recursos			*Counted Pointer*

Figura 1 - Clasificación POSA

- **GoF (Gang of Four).** Es una de las clasificaciones más referenciadas. Clasifican los patrones según su propósito y su ámbito (Figura 2).

Design Patterns				
		Purpose		
		Creational Patterns	Structural Patterns	Behavioral Patterns
Scope	Class	• Factory Method	• Adapter (de clases)	• Interpreter • Template Method
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Adapter (de objetos) • Bridge • Composite • Decorator • Façade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

Figura 2 - Clasificación GoF (Design Patterns)

- **GRASP**. Clasifican los patrones en aspectos fundamentales y avanzados del diseño.

3.2.3.1 Patrón Singleton

El patrón de diseño *Singleton* (instancia única) es un patrón de creación de objetos, está diseñado para restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto.

Intención, consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.

Problema, diferentes objetos precisan referenciar a un mismo elemento y asegurarnos de que no hay más de una instancia de ese elemento.

Solución, asegurar que únicamente existe una instancia.

Aplicabilidad, cuando deba haber exactamente una instancia de una clase y ésta deba ser accesible a los clientes desde un punto de acceso conocido. Y cuando la única instancia debería ser extensible mediante herencia y los clientes deberían ser capaces de utilizar una instancia extendida sin modificar su código.

El patrón *Singleton* se implementa generando en nuestra clase un método que crea una instancia del objeto sólo si todavía no existe niguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor (con atributos como protegido o privado).

El patrón *Singleton* provee una única instancia global gracias a que:

- La propia clase es responsable de crear la única instancia.
- Permite el acceso global a dicha instancia mediante un método de clase.
- Declara el constructor de clase como privado para que no sea instanciable directamente.

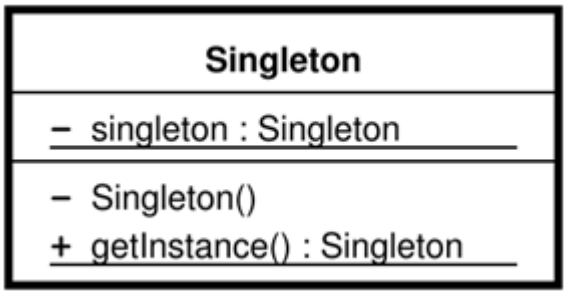

Figura 3 - Patrón Singleton

3.2.3.2 Patrón MVC

MVC (Modelo Vista Controlador) es un patrón de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario y la lógica de negocio en tres componentes distintos.

Intención, consiste en realizar un diseño que desacople la vista del modelo, con la finalidad de mejorar la reusabilidad. Así las modificaciones en las vistas impactan en menor medida en la lógica de negocio o de datos.

Problema**,** la propensión que tienen las interfaces de usuario para cambiar, extender funcionalidad de una aplicación o portarla a un entorno gráfico diferente y que construir un sistema flexible es caro y propenso a errores si la interfaz de usuario está altamente entremezclada con el núcleo funcional.

Solución**,** separar la aplicación en tres áreas (Figura 4):

- **Modelo**: Este componente encapsula los datos y la funcionalidad central y es independiente de la entrada y la salida.
- **Vista**: Presenta la información al usuario. Una vista obtiene los datos del modelo, pudiendo haber múltiples vistas del modelo.
- **Controlador**: Los controladores reciben la entrada, normalmente eventos que son trasladados para servir las peticiones del modelo o de la vista. El usuario interactúa con el sistema sólo a través de los controladores.

Aplicabilidad**,** siempre que se quiera desacoplar la vista del modelo para un menor impacto de las modificaciones y con vistas a la reusabilidad.

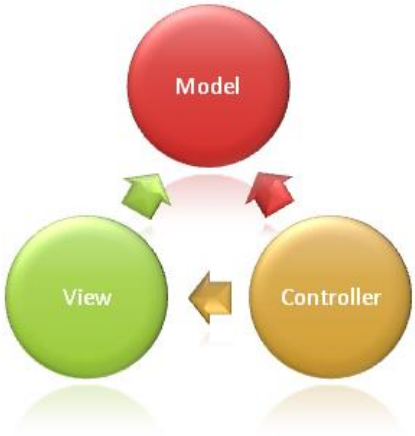

Figura 4 - Patrón MVC

3.2.3.3 Patrón Delegation

En Ingeniería de Software, el patrón de diseño de *Delegate* es una técnica en la que un objeto de cara al exterior expresa cierto comportamiento pero, en realidad, delega la responsabilidad de implementar dicho comportamiento a un objeto asociado.

La mejor manera de comprender este patrón es por medio de un ejemplo dentro del ámbito en el que estamos, como es el *framework* Cocoa Touch. Imaginemos que tenemos una clase que es el controlador de una vista, para ello tiene que heredar de *UIViewController*, y queremos que esa clase también nos implemente los métodos para manejar la *TableView* que tenemos en nuestra vista *UIView*.

Si no utilizamos el patrón *Delegate*, la solución sería la herencia múltiple, con todos los problemas que conlleva y además no es soportada por una gran cantidad de lenguajes (incluido Objective-C) y deberíamos implementar todos los métodos de *UIViewController*, muchos de los cuales no necesitamos. En cambio, el patrón *Delegate* nos permite tener nuestra clase que hereda de *UIViewController* y, además, controlar los eventos que lleguen de esa *TableView* simplemente delegándoselos a nuestra clase, es decir, nuestro patrón delegación lo que hace realmente es delegar un conjunto de operaciones de un objeto en otro objeto como podemos ver en la Figura 5.

Por ello podemos decir:

Intención, consiste en que un objeto expresa cierto comportamiento pero en realidad delega la responsabilidad de implementar dicho comportamiento a un objeto asociado.

Problema, que nuestro lenguaje de programación no soporte la herencia múltiple y que para implementar un determinado comportamiento se necesite heredar de otra clase más.

Solución**,** delegar un conjunto de operaciones de un objeto en otro objeto.

Aplicabilidad, cuando el lenguaje de programación no soporte la herencia múltiple y no se deseen todos sus problemas añadidos y, además, que la clase que hereda no necesite todos los métodos de su clase padre.

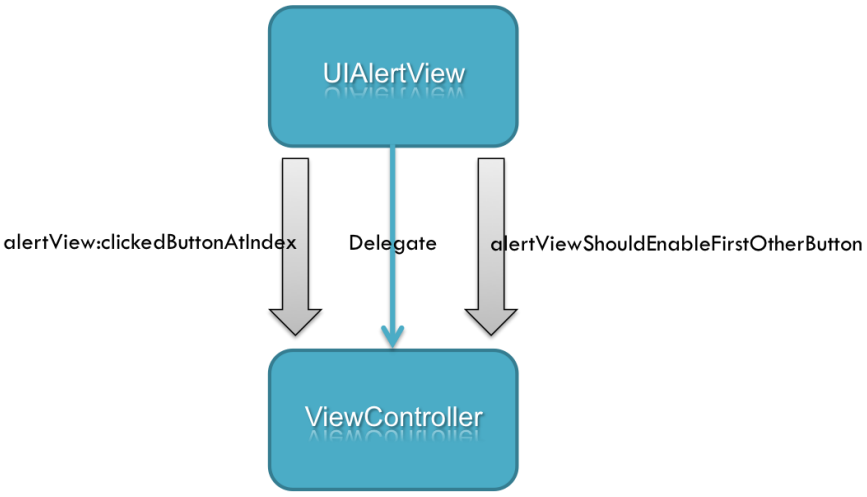

Figura 5 - Patrón Delegate

3.2.3.4 Arquitectura Cliente servidor

La arquitectura Cliente-Servidor es un modelo de aplicación distribuida en el que las tareas se reparten entre los proveedores de recursos o servicios, llamados servidores, y los demandantes, llamados clientes. Un cliente realiza peticiones a otro programa, el servidor, que le da respuesta. Esta idea también se puede aplicar a programas que se ejecutan sobre una sola computadora, aunque es más ventajosa en un sistema operativo multiusuario distribuido a través de una red de computadoras.

En esta arquitectura la capacidad de proceso está repartida entre los clientes y los servidores, aunque son más importantes las ventajas de tipo organizativo debidas a la centralización de la gestión de la información y la separación de responsabilidades, lo que facilita y clarifica el diseño del sistema.

La Figura 6 nos muestra la arquitectura C/S (Cliente-Servidor), donde el **remitente de una solicitud** es conocido como cliente. Sus características son:

- Inicia solicitudes o peticiones y tiene por tanto un papel activo en la comunicación.
- Espera y recibe las respuestas del servidor.

- Por lo general, puede conectarse a varios servidores a la vez.
- Normalmente interactúa directamente con los usuarios finales mediante una interfaz gráfica de usuario.
- Al contratar un servicio de redes, debe tener en cuenta la velocidad de conexión que le otorga al cliente y el tipo de cable que utiliza.

Al **receptor de la solicitud** enviada por el cliente se le conoce como servidor. Sus características son:

- Al iniciarse espera a que lleguen las solicitudes de los clientes, desempeña entonces un papel pasivo en la comunicación.
- Tras la recepción de una solicitud, la procesa y luego envía la respuesta al cliente.
- Por lo general, aceptan conexiones desde un gran número de clientes (en ciertos casos el número máximo de peticiones puede estar limitado).
- No es frecuente que interactúen directamente con los usuarios finales.

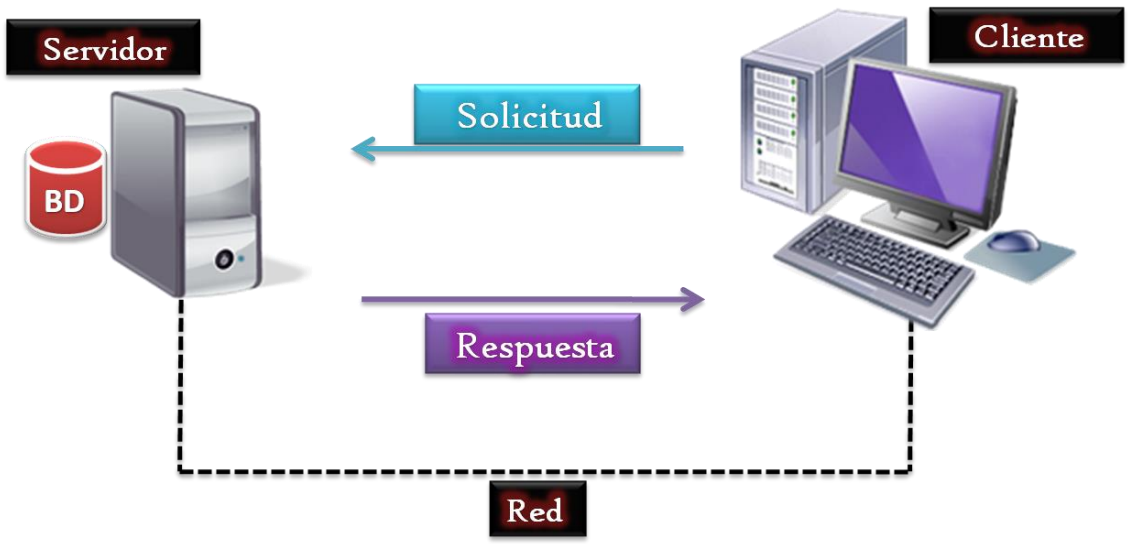

Figura 6 - Arquitectura Cliente-Servidor

3.2.4 Modelo de Referencia OSI.

El modelo OSI (*Open System Interconnection*) es un modelo de interconexión de sistemas abiertos, es el modelo de red descriptivo creado por la Organización Internacional para la Estandarización (ISO[1]) en el año 1984. Es decir, es un marco de referencia para la definición de arquitecturas de interconexión de sistemas de comunicaciones.

[1] http://www.iso.org/iso/home.html

Su arquitectura está estratificada en siete niveles. Cada nivel realiza un conjunto de funciones necesarias para comunicarse con otros sistemas y, además, cada nivel se sustenta en la capa inmediatamente inferior que realizará funciones más primitivas y proporcionará servicios al nivel inmediatamente inferior. Sus niveles son (Figura 7):

- **Nivel Físico**
  - Transmisión de flujo de bits a través del medio.
  - Especifica cables y conectores del medio de transmisión.
- **Nivel de Enlace**
  - Sincronización y entramado.
  - Control de errores.
  - Regulación de flujo.
  - Arbitrar el acceso al enlace cuando lo comparten varios sistemas. También llamado coordinación de la comunicación.
- **Nivel de red**
  - Encamina los paquetes de origen a destino.
  - Control de la congestión.
  - Resolver problemas de interconexión de redes heterogéneas.
- **Nivel de Transporte**
  - Ofrecer un sistema de transferencia de datos fiable y homogéneo entre dos procesos en máquinas remotas, de forma independiente de la tecnología de la subred subyacente.
- **Nivel de Sesión**
  - Gestión y control del diálogo.
  - Sincronización. Insertar puntos de sincronización para poder retransmitir desde esos puntos y no retransmitirla entera.
  - Gestión de Actividad.
- **Nivel de Presentación**
  - Sintaxis y semántica de la información. Ofrece la posibilidad de intercambiar por la red estructuras complejas de datos conservando su significado aunque varíe su representación interna.
  - Comprensión.

- Seguridad. Cifrado de los datos.

❖ **Nivel de Aplicación**

- Resuelve ya problemas directos del usuario con la red.

En una conexión, cada capa del modelo *OSI* se comunica con la misma capa del otro lado de la comunicación (Figura 7). La comunicación entre las dos capas debe ser la indicada en el protocolo que se opte por adoptar.

El modelo especifica el o los protocolos que pueden usarse en cada capa. Algunos de los protocolos más conocidos a nivel de redes son TCP (*Transmission Control Protocol*, «Protocolo de Control de Transmisión»), UDP (*User Datagram Protocol*) o IP (*Internet Protocol*).

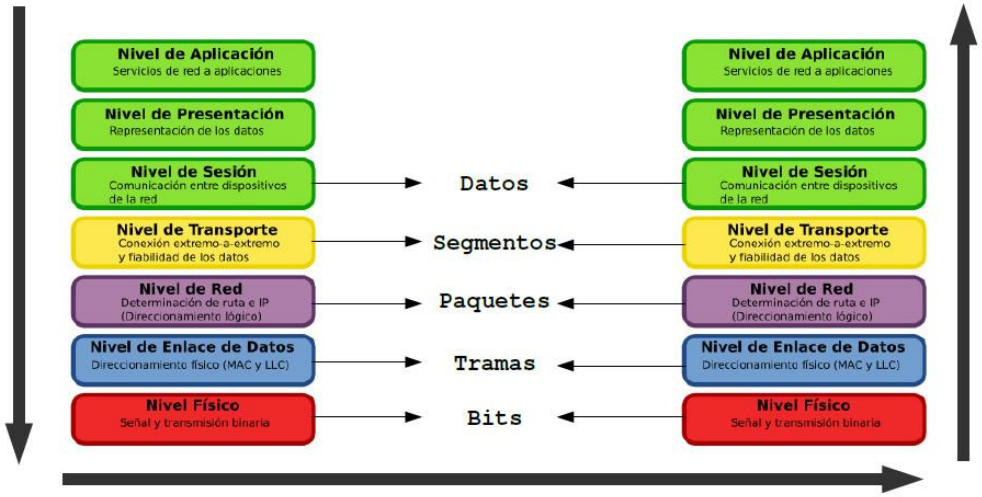

Figura 7 - Capas OSI

3.2.4.1 Sockets

Un *socket* es un mecanismo que proporciona un punto final para establecer una comunicación a través de la red. Podríamos definirlo como un punto de acceso a un servicio perfectamente identificado en la red por el que se mandan y aceptan paquetes de información.

En el modelo *OSI* los *sockets* ofrecen al nivel 5 una interfaz de acceso al nivel 4.

Las propiedades de un *socket* dependen de las características del protocolo en el que se implementan. El protocolo más utilizado es *TCP* y una alternativa común a éste es *UDP*.

Cuando se implementan con el protocolo *TCP*, los *sockets* tienen las siguientes propiedades:

❖ Son orientados a la conexión.

❖ Se garantiza la transmisión de todos los octetos sin errores ni omisiones.

- Se garantiza que todo octeto llegará a su destino en el mismo orden en que se ha transmitido.

Estas propiedades son muy importantes para garantizar la corrección de los programas que tratan la información.

El protocolo *UDP* es un protocolo no orientado a la conexión. Sólo se garantiza que si un mensaje llega, llegue bien. En ningún caso se garantiza que llegue o que lleguen todos los mensajes en el mismo orden que se mandaron. Esto lo hace adecuado para el envío de mensajes frecuentes pero no demasiado importantes, como por ejemplo, un streaming de audio.

En el proyecto QrNav se utiliza un *socket TCP* en lugar de un *socket UDP*. Se ha optado por este protocolo por la razón de que se requiere un servicio de transporte fiable.

El *API* de *socket* en C es bastante sencillo y cuenta con un número de funciones como podemos ver en la **¡Error! La autoreferencia al marcador no es válida.** :

- **Socket().** Crea un *socket* con algún protocolo y te devuelve un descriptor para poder acceder a él.
- **Bind().** Asocia una dirección a nuestro *socket*.
- **Listen().** Sirve para especificar el número de conexiones máximas que pueden estar en espera y para avisar al sistema de que comience a atender la recepción de clientes.
- **Connect().** Sirve para solicitar la conexión por parte del cliente a algún servidor.
- **Accept().** Acepta las conexiones de clientes. Una vez aceptada una, pasa a aceptar la petición del siguiente cliente en la cola. Si no hubiera se quedará bloqueada hasta la llegada de nuevos clientes.
- **Read() y Write().** Sirven para recibir los datos o para enviarlos. Tanto cliente como servidor deben saber qué datos esperan recibir y enviar y en qué formato.
- **Close().** Cierre de la comunicación y del *socket*.

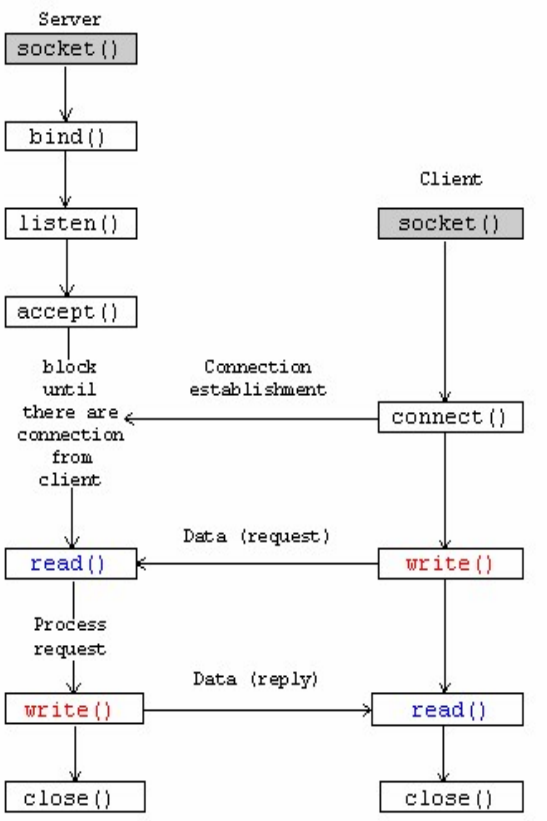

Figura 8 - Socket TCP

3.3 Conceptos Teóricos Específicos

3.3.1 Cocoa Touch.

Cocoa Touch es un conjunto de *framework*s ofrecidos por Apple para el desarrollo de software para dispositivos iOS. Está basado en el conjunto de *frameworks* de programación para MAC OS, pero se han eliminado los *frameworks* que no pueden utilizarse o no están entendidos para usarse en un dispositivo móvil. Además se han incluido adaptaciones de *frameworks* y clases que son necesarias sobre todo por la pantalla *multitouch* del dispositivo, su novedosa forma de interactuar con el usuario y la optimización necesaria en estos dispositivos que no poseen un hardware tan potente como un portátil o sobremesa.

Basados en el paradigma Modelo-Vista-Controlador, Cocoa Touch proporciona una base sólida para la creación de aplicaciones. Cuando se combina con la herramienta Interface Builder, es fácil y divertido de utilizar para el diseño de las aplicaciones iOS.

Gracias a su potente bajo nivel permite que existan *frameworks* de alto nivel como Game Kit (para el juego multijugador), Core Data, (que ofrece un alto rendimiento y es fácil de usar en la gestión de datos), Core Animation (para los efectos impresionantes), WebKit (el motor de navegador más innovador en dispositivos móvil), etc.

Gran parte de Cocoa Touch se lleva a cabo en Objective-C, un lenguaje orientado a objetos que se compila para correr a una velocidad increíble, sin embargo, emplea un

tiempo de ejecución muy dinámico por lo que es muy flexible. Debido a que Objective-C es un superconjunto de C, es fácil de mezclar C y C++, incluso en las aplicaciones Cocoa Touch.

Existen dos *frameworks* básicos que constituyen la capa de *Cocoa Touch* como son UIKit y Foundation. Estos dos *frameworks* son por tanto imprescindibles en cualquier aplicación iOS y si los eliminásemos de nuestro proyecto no podríamos compilar ni siquiera un Hola Mundo. El resto pueden o no ser utilizados en un proyecto. Como veremos más adelante, existen muchos más *frameworks* que los dos básicos que constituyen la capa *Cocoa Touch*. Como se puede ver en la Figura 9 existen, por ejemplo, relacionados con Multimedia, (como reproducción de audio o vídeo), relacionados con Servicios (como accesos a la agenda de contactos del iPhone), y de nivel de sistema operativo y *Kernel* (como *threading*, acceso de bajo nivel a archivos, redes, etc).

Figura 9 - Arquitectura de capas de iOS

3.3.1.1 UIKit

El *framework* UIKit (Figura 10) es uno de los más importantes de Cocoa Touch. Este *framework* proporciona las clases necesarias para construir y gestionar la interfaz de usuario de una aplicación iOS.

Proporciona un objeto de aplicación, manejo de eventos, el modelo de dibujo, ventanas, vistas y los controles diseñados específicamente para una la pantalla táctil.

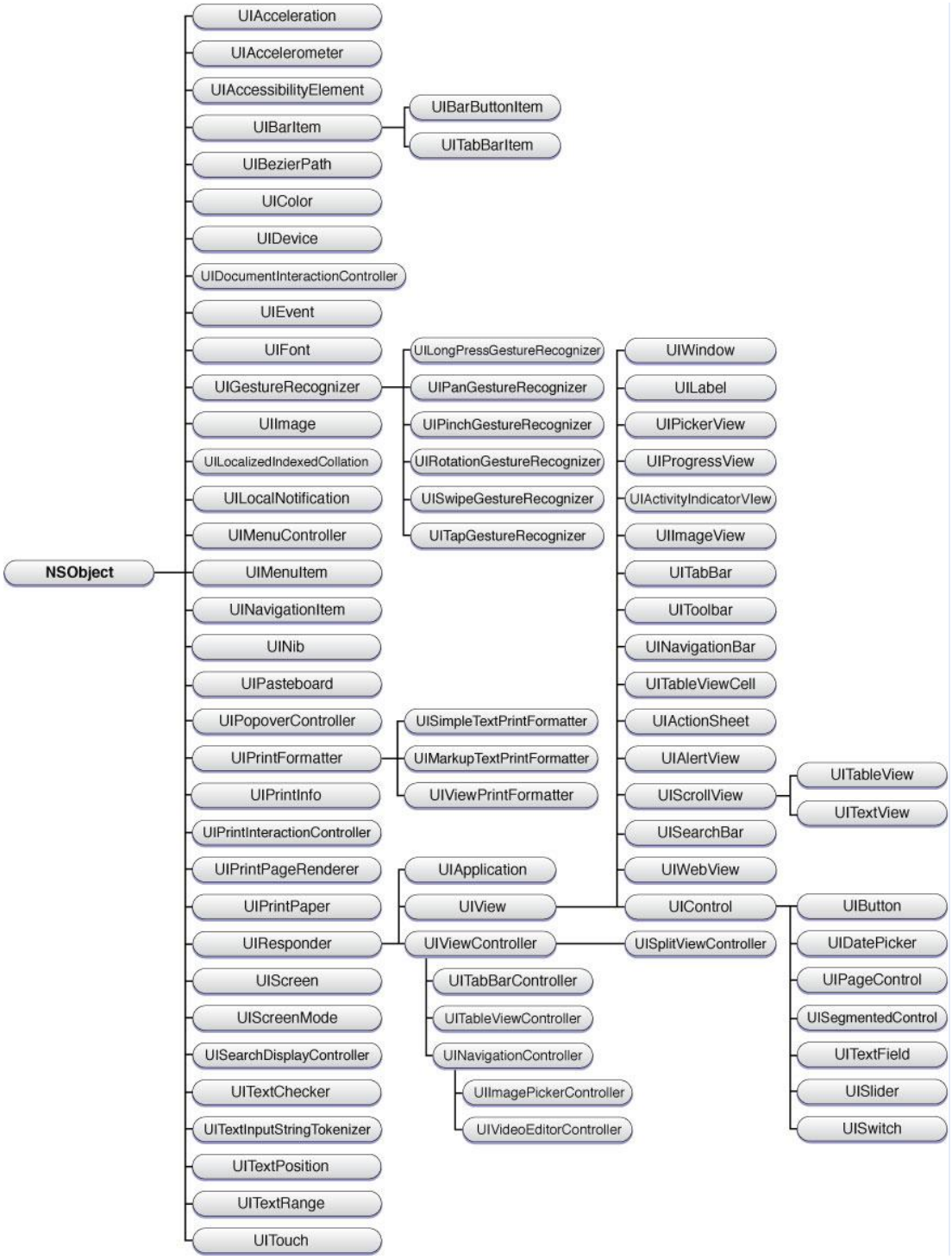

Figura 10 - UIkit

3.3.1.1.1 UIView

La clase *UIView* define un área rectangular en la pantalla y las interfaces para la gestión del contenido en esa área. En tiempo de ejecución, un *view* se encarga de cualquier contenido en su área y también se ocupa de las interacciones con ese contenido por parte del usuario.

El comportamiento básico de la clase *UIView* es rellenar el área rectangular con un color de fondo. Puede existir contenido más complejo que se representa por medio de subclases *UIView* e implementando lo necesario para dibujar y gestionar eventos en el código.

El *framework* UIKit también incluye un conjunto de subclases estándar que van desde simples botones hasta tablas complejas. Por ejemplo, un objeto *UILabel* dibuja una cadena de texto y un objeto *UIImageView* dibuja una imagen.

Debido a que los objetos *UIView* son la parte principal de la aplicación con la que interactúa el usuario, tienen una serie de responsabilidades. Éstas son sólo algunas de ellas:

- Dibujo y animación, las *view* dibujan el contenido en su área rectangular utilizando tecnologías tales como UIKit, Core Graphics, y OpenGL ES[2].
- Diseño y gestión de *subviews*, las *view* pueden contener otras *views*, pudiendo crear una sofisticada jerarquía de vistas con relaciones Padre-Hija entre las vistas contenidas (*subviews*) y las vistas contenedoras o padres (*superview*). Una *superview* puede contener cualquier número de *subviews*, pero cada *subviews* sólo tiene una *superview*, que es responsable de posicionar sus *subviews* adecuadamente. Podemos ver un ejemplo en la Figura 11.

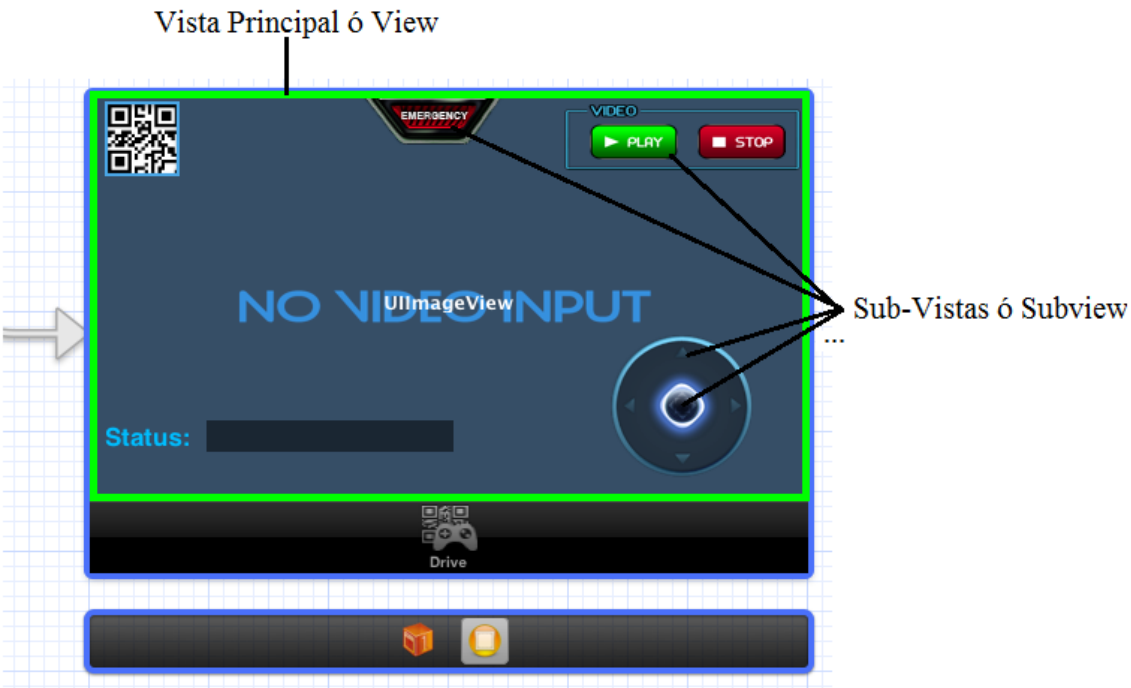

Figura 11 - View con Subview

[2] http://www.opengl.org

- Gestión de eventos, una *view* es la encargada de manejar los eventos *Touch* (toque en la pantalla por parte de un usuario) y otros eventos definidos por la clase *UIResponder*, incluso puede usar métodos para reconocer gestos *multitouch.*

3.3.1.1.2 UIViewController

La clase *UIViewController* nos proporciona las funcionalidades fundamentales para el manejo de las vistas en las aplicaciones iOS. Un *UIViewController* gestiona un conjunto de *view* que forman parte de la interfaz de usuario de la aplicación. Se encargan de:

- Cambiar el tamaño y establecer sus *view*.
- Ajustar el contenido de sus *view*.
- Actúa por sus *view* cuando el usuario interacciona con alguna de ellas.

El caso más sencillo de utilización, es una instancia de *UIViewController* para gestionar una jerarquía de vistas, normalmente, una vista y varias subvistas, tales como botones, interruptores, cajas de texto, etc.

Existen otros *UIViewController* más complejos (subclases de *UIViewController*), como por ejemplo *UINavigationController* que sólo muestra el contenido de una *view* en un momento dado y las demás *view* se muestran en serie. Las *view* están dispuestas en una pila de modo que, cuando una se retira, la anterior vuelve a aparecer. Las *view* adyacentes en la pila de navegación a menudo comparten una base de datos de relación e intercambio con los demás. Otro controlador sería *UITabBarController* que muestra sólo una *view* a la vez, pero no establece una relación inherente entre las *view*.

Los *ViewController* están estrechamente vinculados a las *view* que gestionan y participan en la cadena de respuesta utilizada para controlar los eventos que se producen. Los *UIViewController* heredan de la clase *UIResponder*, por lo que es capaz de responder a los eventos básicos producidos en el ciclo de vida de una vista. Por tanto, si su correspondiente vista no maneja un evento, este será remitido a su controlador.

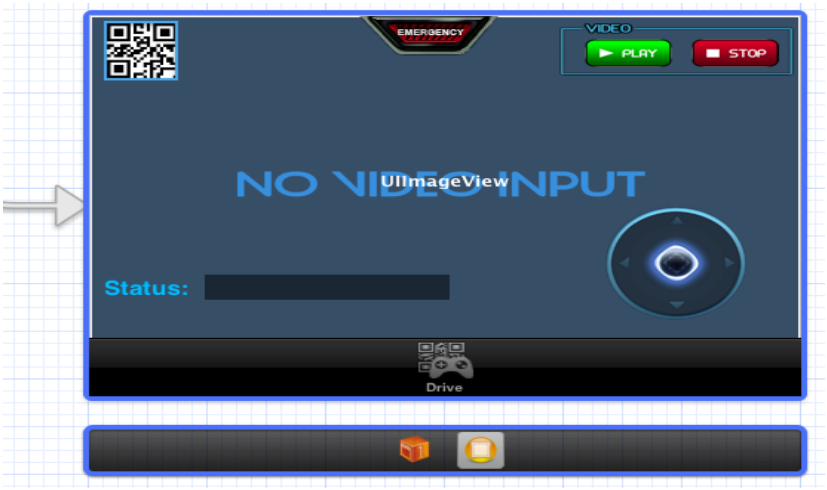

Figura 12 - UIView personalizada

3.3.1.1.3 UITabBar

La clase *UITabBar* implementa un control para seleccionar dos o más botones, llamados ítems.

El uso más común de una barra de pestañas, como vemos en la Figura 13, es implementar una interfaz donde tocar un ítem cambia la *view*. Es muy útil cuando queremos separar nuestra aplicación en diferentes secciones o áreas.

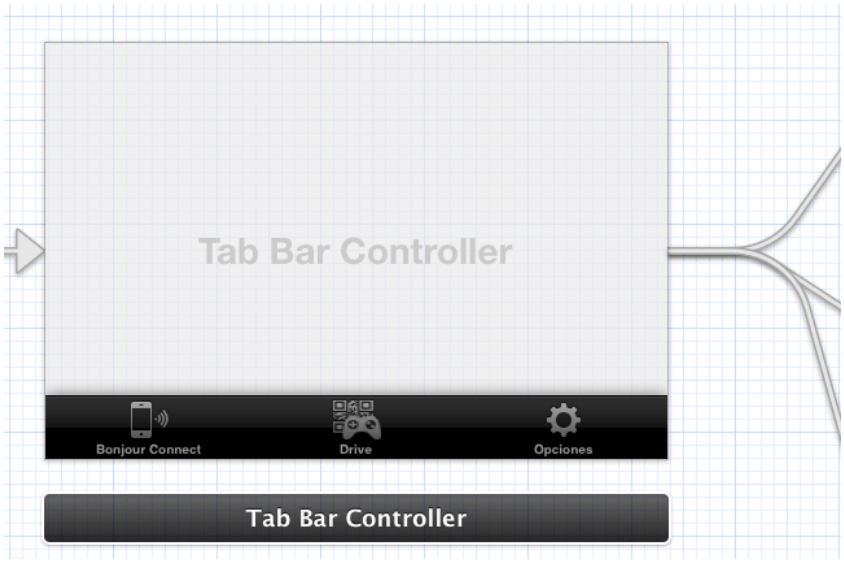

Figura 13 - UITabBar

La clase *UITabBar* proporciona la capacidad para personalizar la barra de pestañas, tales como reordenación, agregar y quitar elementos a la barra, etc. Se puede utilizar un delegado de *TabBar* para aumentar el control sobre dicho *TabBar*.

Una característica bastante buena y cómoda es que el *TabBar* gestiona automáticamente el número de ítems, si asciende a seis o superior, automáticamente el

TabBar genera un ítem "*More*" (Figura 14) que servirá para seleccionar entre los demás ítems.

Figura 14 - TabBar personalizada con más de 4 ítems

3.3.1.1.4 UITableView

Una instancia de *UITableView* es un medio para mostrar y editar listas jerárquicas de información, es decir, una serie de componentes que se muestran en una vista preliminar. Al tocar sobre un componente se desplegará una nueva vista con más información sobre el mismo.

Una *UITableView* se limita a una sola columna, ya que está diseñado para un dispositivo con una pantalla pequeña. *UITableView* es una subclase de *UIScrollView*, que permite a los usuarios desplazarse verticalmente a través de la tabla pero nunca horizontalmente.

Las celdas que componen la tabla son objetos *UITableViewCell* que representan los objetos individuales que queremos mostrar en la tabla. Estas celdas se pueden personalizar al gusto del programador con títulos, imágenes, botones, etc.

Una *UITableView* se compone de cero o más secciones, cada una con sus propias celdas. Las secciones se identifican por su número de índice en la *UITableView* y las celdas se identifican por su número de índice dentro de una sección.

Podemos ver el aspecto de una *TableView* en el Interface Builder en la Figura 15.

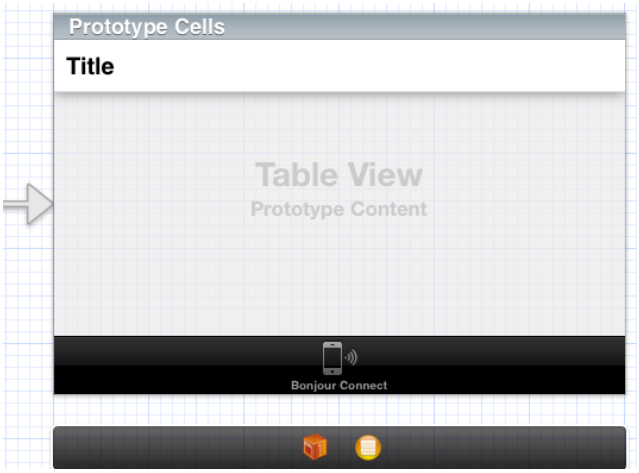

Figura 15 - Table View

Un objeto *UITableView* debe tener un objeto que actúe como fuente de datos y un objeto que actúe como delegado. La fuente de datos debe adoptar el protocolo *UITableViewDataSource* y el delegado debe adoptar el protocolo *UITableViewDelegate*. La fuente de datos proporciona información que necesita para construir *UITableView* y gestiona el modelo de datos (inserción, eliminación, reorganización y modificación). El delegado proporciona las celdas utilizadas en las tablas y realiza otras tareas, como por ejemplo, qué hacer cuando se selecciona una celda. En la Figura 16, podemos ver cómo están realizadas las conexiones en el Interface Builder.

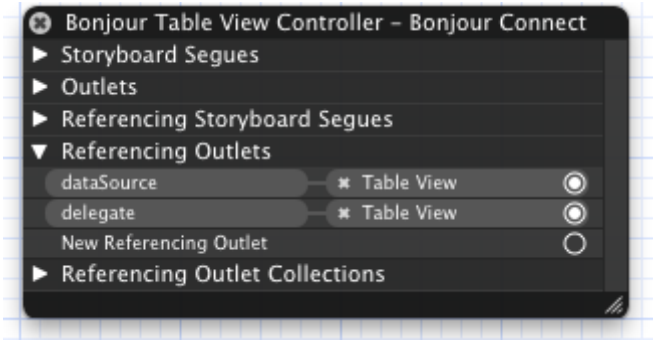

Figura 16 - Conexiones Table View

3.3.1.1.5 Core Data

Core Data es una *API* que nos permitirá tratar el acceso a la capa persistente de forma completamente orientada a objetos. Core Data es uno de los *framework* proporcionados por Cocoa Touch.

Stack es el término utilizado para describir una colección de objetos del *framework*, que trabajan en conjunto para obtener los objetos modelados a partir de los datos y guardar los datos en una *Persistent Store*. Un *Persistent Store* es un archivo donde están guardados nuestros datos. Conceptualmente, un *Persistent Store* es como una base de datos, con tablas y registros.

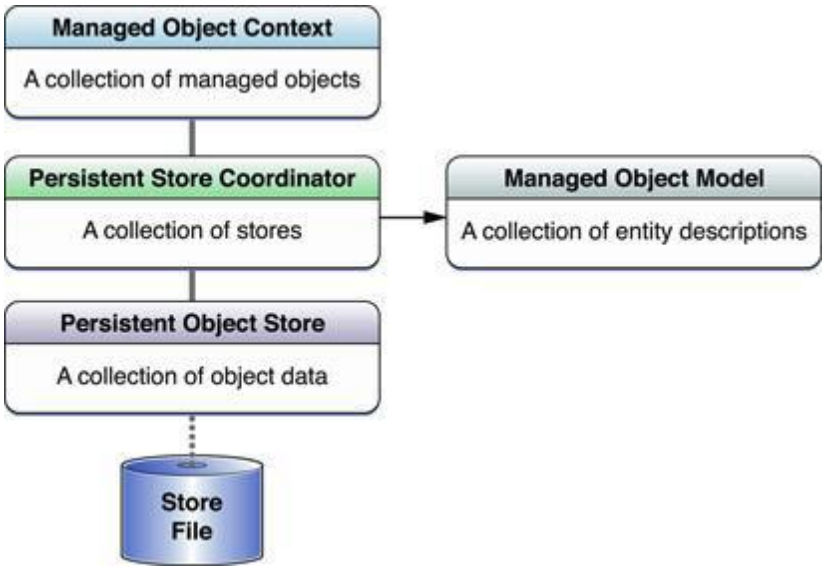

Figura 17 - Core Data Stack

Como vemos en la Figura 17, *Core Data Stack* está compuesto de los siguientes objetos: uno o más *Managed Object Contexts* conectados a un único *Persistent Store Coordinator*, el cual, a su vez está conectado a uno o más *Persistent Stores*. Un *Stack* contiene todos los componentes de Core Data que se necesitan para buscar, crear y manipular *Managed Objects*.

A continuación se explica en qué consiste cada uno de estos componentes:

- **Managed Objects,** puede ser una instancia de *NSManagedObject* o una subclase de *NSManagedObject*. Básicamente representaría un registro en una base de datos y, conceptualmente, es una representación de una tupla guardada dentro de una tabla de SQL (*Structured Query Language*). Por ejemplo, si tenemos una tabla de alumnos, un *Managed Object* sería Pedro, con todos su datos (edad, DNI, curso...)

- **Managed Object Context,** es la clase en la que se guardarán temporalmente los *Managed Object*. Se usa como borrador, es decir, los *Managed Object* guardados en él no se salvarán en memoria persistente hasta que no se ejecute el mensaje correspondiente en su *Managed Object Context*. Los *Managed Object Context* tendrán asociado un *Managed Object Model*.

- **Managed Object Model, e**s una instancia de *NSManagedObjectModel*. Representa al modelo de datos, es decir, una colección de entidades que definen los *Managed Objects*. Además, cada entidad está compuesta principalmente por atributos. XCode nos ofrece una herramienta con la que crear el *Managed Objects Model*, para ello realizaremos de forma gráfica la representación de nuestras clases (Figura 18). Cada una de estas representaciones será un *Entity*. Una vez creados todos los *Entity* de nuestro modelo, podremos hacer que XCode cree el correspondiente *Managed*

Object para cada *Entity*, creándonos una clase en nuestro proyecto para cada uno.

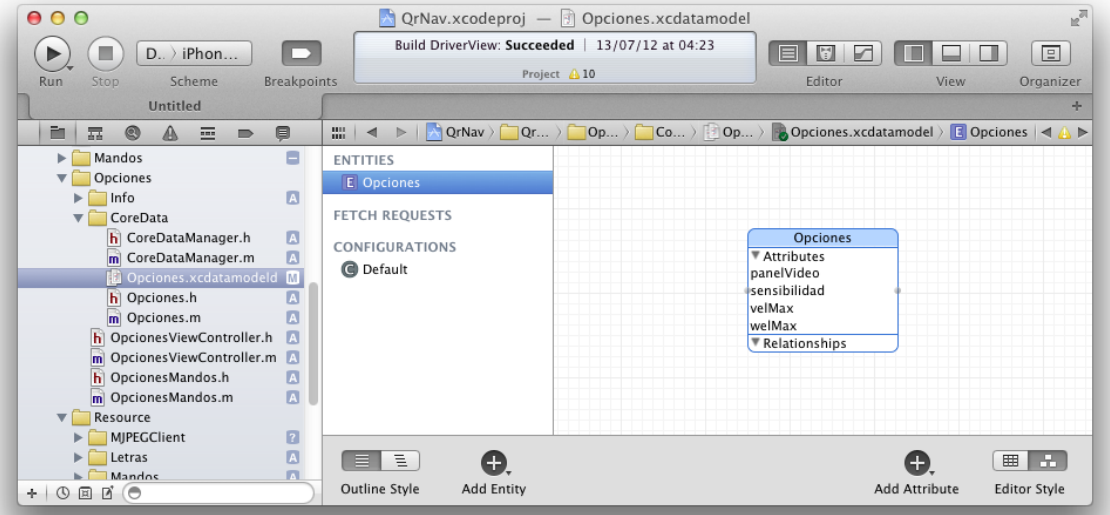

Figura 18 - Managed object Model en Xcode

- **Persistent Store Coordinator,** es una instancia de *NSPersistenStoreCoordinator*. Básicamente maneja una colección de *Persistent Object Stores*. Por lo general no se interactúa de forma directa con el *Persistent Store Coordinator*.

- **Persistent Object Store,** representa un repositorio externo. Hay diferentes clases de *Persistent Object Store*, que representan las distintas formas de almacenar la información. Se puede implementar nuestro propio *Persistent Object Store* si se desea.

- Para las consultas tenemos **Fetch Request.** Es una instancia de *NSFetchRequest* que representa una búsqueda de *Managed Objects* sobre un *Managed Object Context*. Podemos obtener todos los elementos simplemente definiendo el nombre de la entidad, establecer filtros (*NSPredicate*), especificar el orden de los datos (*NSSortDescriptor*), etc. El resultado de la ejecución de una consulta con *Fetch Request* es un array de *Managed Objects*.

Por tanto, podemos concluir que, para utilizar el *framework* Core Data debemos:

- Crear un *Managed Object Model*, que tendrá un esquema parecido a un diagrama de clases de cómo serán los datos que queremos guardar (relaciones entre clases incluidas) con la herramienta proporcionada por XCode, que nos creará una clase funcional para cada *Entity* del *Managed Object Model*.

- A continuación debemos asociar nuestro *Managed Object Model* a un *Managed Object Context*, una especie de contexto en el que guardaremos los *Managed Objets* que vayamos creando.
- Para que los *Managed Objets* se guarden de manera persistente sólo tendremos que indicarle al contexto que los guarde.
- Para añadir objetos, sólo tendremos que crear una instancia de uno de los *Managed Objets* usando la *API* del *Managed Object Model*.
- Para obtener objetos tendremos que realizar una consulta con *Fetch Request*.

3.3.1.2 Foundation

Este *framework* define la capa base de las clases de Objective-C. Además de proporcionar un conjunto de útiles clases de objetos primitivos, introduce varios paradigmas que definen la funcionalidad y que no están cubiertos por el lenguaje Objective-C. El *framework* Foundation se ha diseñado con los siguientes objetivos en mente:

- Proporcionar un pequeño conjunto de clases con utilidades básicas.
- Hacer más fácil el desarrollo de software mediante la introducción de convenciones coherentes para diferentes aspectos, tales como cancelación de asignación (*dellocation*).
- Soporte a cadenas *Unicode*, la persistencia de objetos y los objetos distribuidos.
- Proporcionar un nivel de independencia del sistema operativo, para mejorar la portabilidad.

El *framework* Foundation incluye la clase raíz de los objetos, las clases que representan tipos de datos básicos (como cadenas y arrays), las clases de colección para guardar otros objetos, las clases que representan la información del sistema (tales como fechas), las clases que representan los puertos de comunicación, etc.

3.3.1.3 Ciclo de vida de una aplicación

Como puede apreciarse en la Figura 19, el desarrollo para iOS sigue el patrón *MVC*. El punto de entrada a nuestra aplicación es siempre una aplicación delegada de *UIApplication*, por lo que crearemos una clase que herede de *UIApplicationDelegate*. A continuación tendremos al menos un *ViewController* que controlará una vista de tipo *UIWindow*.

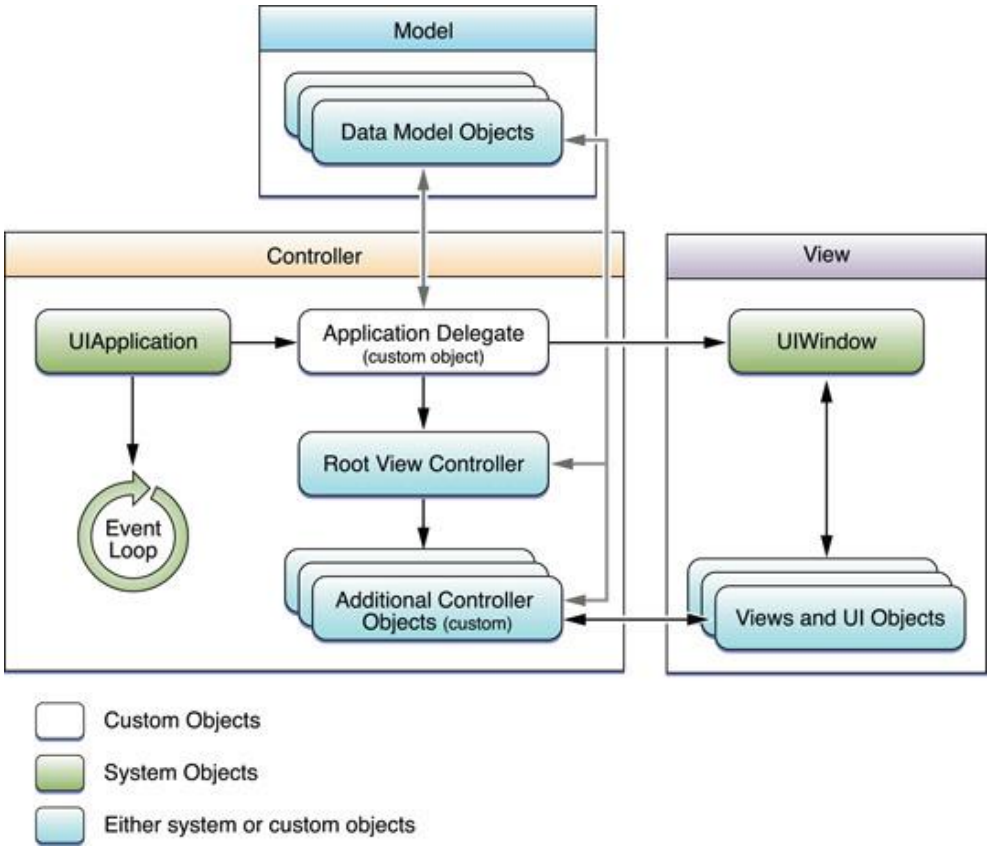

Figura 19 - Estructura de componentes de una típica App para iOS

Desde la versión iOS4.x y sólo en dispositivos que soportan multitarea, una aplicación puede encontrarse en los siguientes estados:

- **Sin ejecución (Not running):** La aplicación no está en la memoria de ejecución del dispositivo.
- **No activa (inactive):** La aplicación se encontraba antes en primer plano pero ha sucedido algo que la deja, momentáneamente, fuera de foco.
- **Activa (Active):** La aplicación está en primer plano y está en ejecución.
- **Segundo plano (Background):** La aplicación se encuentra ejecutando sin estar mostrando su interfaz de usuario.
- **Suspendida (Suspended):** La App se encuentra "congelada" y será candidata a ser cerrada y pasar al estado de *Not Running* si el sistema necesita recursos. Si el usuario, mediante el interfaz del dispositivo, vuelve a lanzar una app, que estaba en este estado, la encontrará exactamente como la dejó (mostrando la última vista que usó, con sus mismos datos, etc).

Veamos ahora el siguiente esquema (Figura 20) que muestra el ciclo de vida de una aplicación de terceros (third-party apps) en la que sí se soporta multitarea:

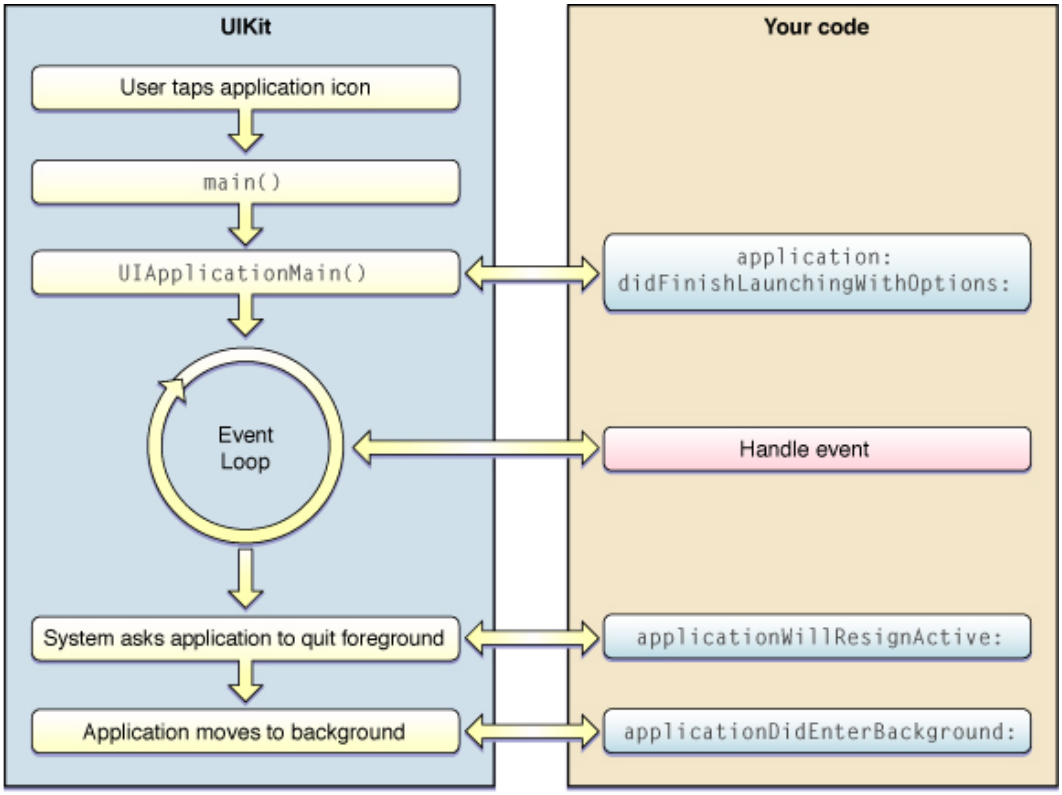

Figura 20 - Ciclo de vida de una aplicación de terceros

El ciclo de vida de una aplicación comienza, al igual que cualquier programa de C, con la llamada al método *main* por parte del sistema operativo cuando tocamos el icono de nuestra aplicación.

Si *main* es la puerta de entrada para nuestra aplicación, *UIApplicationMain* es la puerta de entrada a Cocoa, y es aquí donde entra en juego el UIKit.*framework*, ya que *UIApplicationMain* es una función que pertenece a dicho *framework*.

Lo que ocurre después es que el sistema pasará la ejecución a aquella clase que responda a algún evento ocasionado por el ciclo de vida de la aplicación. De nuevo entra en juego el UIKit.*framework* proporcionándonos el protocolo *UIApplicationDelegate* con los siguientes métodos que nuestra clase raíz implementa:

- ***applicationDidFinishLaunchingWithOptions***: Los programas deben comenzar la ejecución con una clase (que llamaremos clase raíz) que implemente este protocolo y responda al menos a este mensaje. XCode 4 siempre crea una clase que se llame *AppDelegate* y convierte a esa clase en la clase raíz de nuestro programa, haciendo que implemente el protocolo *UIApplicationDelegate*.

- ***applicationDidReceiveMemoryWarning***: Este método deberíamos implementarlo si queremos que nuestra aplicación sea capaz de "sobrevivir" ante una situación de poca memoria en el sistema. Si no lo hacemos liberando recursos innecesarios cuando este evento es llamado, nuestra aplicación será cerrada forzosamente por iOS.

Cuando nuestra aplicación se cierra nuestro código aún tiene la capacidad de responder implementando los métodos:

- ***applicationWillTerminate***: Es la oportunidad para guardar el estado actual en el que se encontraba nuestro programa, para que el usuario tenga la sensación de que la aplicación se encuentra en el mismo estado en el que la dejó.

- ***applicationDidEnterBackground***: Existe la posibilidad de que el sistema reclame memoria y la aplicación pase de estar suspendida a cerrarse definitivamente. En este caso no recibiríamos nunca el evento *applicationWillTerminate* y tendríamos que actuar en el momento de pasar a segundo plano, es decir, en este evento.

- ***applicationWillResignActive y applicationDidBecomeActive:*** Para responder ante los posibles eventos o cambios de estados antes mencionados y que, en cierto modo, dejan nuestra aplicación en segundo plano durante ese momento.

- ***applicationDidEnterBackground:*** Antes de que la App vaya a pasar a estado suspendido, entra en este evento durante un máximo de 5 segundos. Si nuestro código se ejecuta durante más tiempo sin haber solicitado entrar en "*task completition background*" mediante la llamada a *beginBackgroundTaskWithExpirationHandler* entonces la App es suspendida por iOS.

3.3.1.4 Nib Files (Next Interface Builder)

Los archivos *Nib* (*Next Interface Builder***)** desempeñan un papel importante en la creación de aplicaciones iOS. Con los *Nib files*, podemos crear y manipular las interfaces gráficas de usuario, usando XCode en lugar de mediante programación. Al poder visualizar los resultados de los cambios al instante, se puede experimentar con diferentes diseños y configuraciones muy rápidamente. También se pueden cambiar numerosos aspectos de la interfaz de usuario sin tener que escribir ninguna línea de código.

Para las aplicaciones construidas utilizando el *framework* UIKit, los *Nibs Files*, adquieren una importancia adicional. Este *framework* soporta el uso de *Nibs Files*, tanto para la colocación de las ventanas, vistas y los controles como para la integración de estos elementos con el código de la aplicación de gestión de eventos. XCode trabaja en conjunto con este *framework* para ayudar a conectar los controles de la interfaz de usuario con los objetos del proyecto que responden a estos controles. Esta integración reduce significativamente la cantidad de configuración que se requiere y también hace que resulte fácil cambiar las relaciones entre el código y la interfaz de usuario más tarde.

Estos archivos *Nib* están en desuso gracias a la herramienta de Apple, los archivos *Xib*. La diferencia entre ambos es que el archivo *Xib* nos ofrece una forma más amigable de trabajar que los archivos *Nib*. Es decir, desde un punto de vista técnico, los archivos *Xib* son la versión XML (*eXtensible Markup Language*, «Lenguaje de marcas extensible») de los archivos *Nib*. En tiempo de compilación XCode automáticamente convertirá los *Xib* en *Nib*.

Los contenidos de un archivo *Nib* pueden ser variados, aunque hay una serie de elementos que se repiten usualmente. La razón de ello es que los *Nib* suelen usarse casi siempre para configurar las vistas.

3.3.1.4.1 Storyboard

Con la llegada del XCode 4.2 y el *SDK* 5, apareció una nueva forma de crear las interfaces gráficas de usuario de manera más rápida, cómoda y sencilla.

Se trata de los *Storyboard*, que son una representación visual de la interfaz de usuario de una aplicación para iOS y que muestran el contenido de las pantallas y las conexiones entre dichas pantallas. Un *Storyboard* se compone de una secuencia de escenas y cada una de ellas representa un *ViewController* y su *View*. Las escenas están conectadas por objetos *segue* que representan una transición entre dos *ViewController*, es decir, una navegación entre una escena y otra.

XCode ofrece un editor visual para *Storyboard* donde se puede visualizar y diseñar la interfaz de usuario de su aplicación mediante la adición de *views* como botones, *TableView* y *TextView* a las escenas. Además, un *Storyboard* le permite conectar una *View* con su *ViewController* y gestionar la transferencia de datos entre *ViewControlers*.

El uso de *Storyboard* es el método recomendado para diseñar la interfaz de usuario de su aplicación, ya que le permiten visualizar el aspecto y el flujo de su interfaz de usuario en un lienzo.

En la Figura 21 podemos observar el *Storyboard* usado en nuestra aplicación.

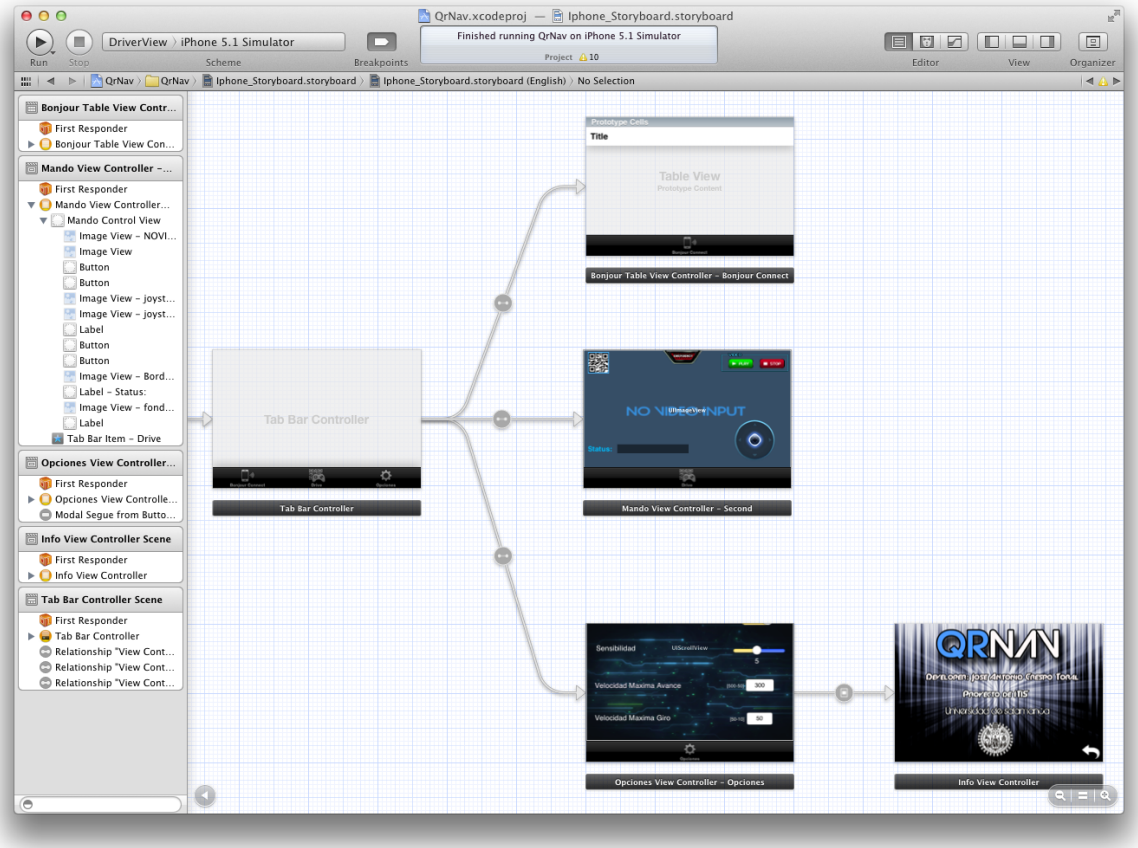

Figura 21 - Storyboard en XCode

Al usar *Storyboard*, los múltiples ficheros *Xib* que antes teníamos se sustituyen por un único fichero ".storyboard". Este fichero se edita desde el propio XCode 4.

Gracias a esta nueva forma de diseñar interfaces podemos establecer cuál es la navegación para cada uno de los elementos de la interfaz e incluso personalizar la forma en que se produce la transición entre los mismos, con una serie de efectos predefinidos pudiendo personalizar otros, sin tener que escribir ni una sola letra de código.

3.3.2 Código QR

Muchos son los que confunden los códigos QR con los códigos BIDI, pero son diferentes. Los códigos BIDI son una particularidad de la compañía Movistar creados en 2008, mientras que los códigos QR son libres y tienen mayor capacidad para codificar un texto libre, una url, una dirección email o un sms y, además, permiten más posibilidades.

Un código QR (*quick response code*, «código de respuesta rápida») es un sistema para almacenar información en una matriz de puntos o un código de barras bidimensional creado por la compañía japonesa Denso Wave en 1994.

Se caracteriza por los tres cuadrados que se encuentran en las esquinas y que permiten detectar la posición del código al lector. La sigla «QR» se deriva de la

frase inglesa Quick Response, pues los creadores «Euge Damm y Joaco Retes» aspiran a que el código permita que su contenido se lea a alta velocidad. Los códigos QR son muy comunes en Japón y de hecho son el código bidimensional más popular en ese país.

Aunque inicialmente se usó para registrar repuestos en el área de la fabricación de vehículos, hoy día los códigos QR se usan para administración de inventarios en una gran variedad de industrias. Recientemente, la inclusión de software que lee códigos QR en teléfonos móviles ha permitido nuevos usos orientados al consumidor, que se manifiestan en comodidades como el dejar de tener que introducir datos de forma manual en los teléfonos. Las direcciones y las *URLs* (*Uniform Resource Locator*) se están volviendo cada vez más comunes en revistas y anuncios. El agregado de códigos QR en tarjetas de presentación también se está haciendo común.

Los códigos QR pueden leerse desde PC, Smartphone o Tablet mediante un dispositivo de captura de imagen (un escáner, la cámara, etc), programas que lean los datos QR y mediante una conexión a nternet para las direcciones web.

El estándar japonés para códigos QR (JIS X 0510) fue publicado en enero de 1998 y su correspondiente estándar internacional ISO (ISO/IEC18004) fue aprobado en junio de 2000.

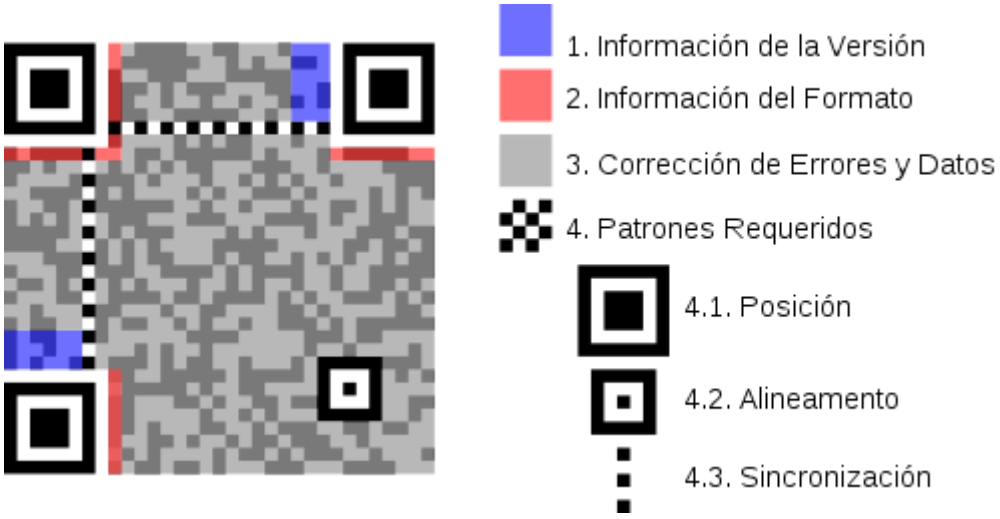

Figura 22 - Estructura de un código QR

Como observamos en la Figura 22, el código QR está compuesto por tres etiquetas de posición situadas en la parte superior izquierda, superior derecha e inferior izquierda. Esto es, de hecho, una característica única que puede ser utilizada para distinguir los códigos QR de otros *2Dbarcodes*. Éstos se utilizan para localizar el código QR en una imagen y ayudar a servir como marcadores. También son útiles para determinar el grado de inclinación y rotación del código en la imagen.

La versión y formato de la información se almacena en al menos dos lugares alrededor de las etiquetas de posición. Esto permite una mejor recuperación de datos en el caso de que una esquina del código esté dañada u obstruida. Los diferentes tipos de datos pueden tener diferentes representaciones en el Código QR por lo que es vital que,

como parte del proceso de decodificación, la información de formato sea recuperable. De lo contrario los datos codificados serían ilegibles. La información de la versión puede ser usada para la capacidad "*backward*", en soporte a antiguas revisiones de la codificación.

Hay también una etiqueta de alineamiento, que puede ser utilizada para los cálculos de alineación adicionales.

El patrón de cuadros continuado, tanto vertical como horizontalmente entre dos etiquetas de posiciones vecinas, permite una rutina de calibración para realizar cálculos y determinar el tamaño de las regiones individuales en blanco y negro. Una de las principales fortalezas del Código QR son sus múltiples niveles de corrección de errores. Se utiliza una técnica de corrección de errores de Reed-Solomon y, dependiendo del nivel de corrupción de datos, puede soportar hasta un 30% de la restauración de éstos.

La capacidad de almacenamiento de datos (Tabla 1) en el código QR varía dependiendo del tipo de dato empleado:

Tipo de dato	Tamaño Máximo
Numérico	Máx. 7.089 caracteres
Alfanumérico	Máx. 4.296 caracteres
Binario	Máx. 2.953 bytes
Kanji/Kana	Máx. 1.817 caracteres

Tabla 1 - Tipo de datos en QR

Y la capacidad de corrección de errores tiene varios niveles (Tabla 2):

Nivel	Restaurar
Nivel L	7% de las claves se pueden restaurar
Nivel M	15% de las claves se pueden restaurar
Nivel Q	25% de las claves se pueden restaurar
Nivel H	30% de las claves se pueden restaurar

Tabla 2 - Corrección de errores QR

3.3.3 Odometría

La odometría es el estudio de la estimación de la posición de vehículos con ruedas durante la navegación, es decir, son las técnicas de posicionamiento que emplean la información procedente de la rotación de las ruedas para obtener una aproximación de la posición real en la que se encuentra un sistema móvil, en un determinado instante y respecto a un sistema de referencia inicial (Figura 23).

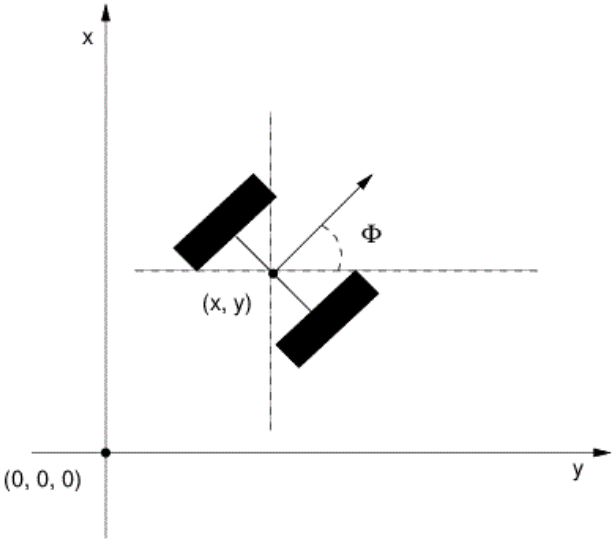

Figura 23 - Sistema de referencia en la odometría

La posición del robot respecto a este sistema de referencia inicial es uno de los parámetros más importantes de los que debe disponer un robot móvil.

En general, son suficientes tres parámetros (X, Y, θ) para conocer la posición de un sistema móvil:

- La posición respecto al eje X que para nuestro robot es el eje vertical.
- La posición respecto al eje Y que en el caso del robot es el eje horizontal.
- La orientación del robot Th o θ que indica el ángulo hacia el que se encuentra orientado.

En la Figura 24 se puede apreciar la posición de los ejes con respecto al Amigobot:

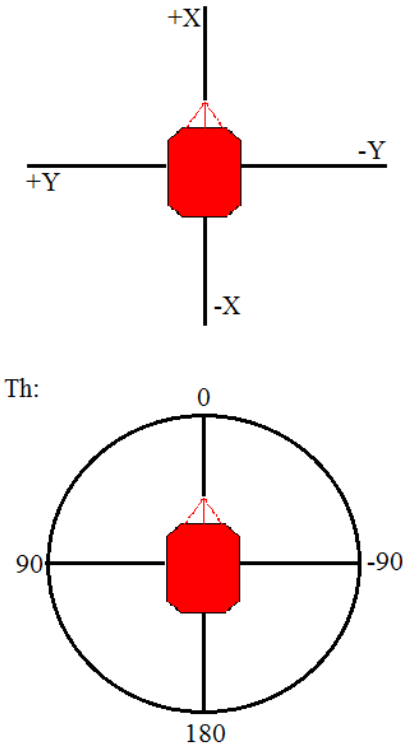

Figura 24 - Coordenadas y ejes del AmigbBot

Cuando el robot se enciende y conecta con un dispositivo, toma la posición actual en la que se encuentra como posición inicial de referencia, es decir, en ese momento estará en la posición (0, 0, 0). Esta posición también es tomada como referencia para establecer los ejes X e Y y para la orientación, ya que cuando el robot conecta toma la orientación que tenga como *Th*=0.

La odometría proporciona una buena precisión a corto plazo, es barata de implantar y permite tasas de muestreo muy altas.

Sin embargo la idea fundamental de la odometría es la integración de información incremental del movimiento a lo largo del tiempo, lo cual conlleva una inevitable acumulación de errores. En concreto, la acumulación de errores de orientación causa grandes errores en la estimación de la posición, que van aumentando proporcionalmente con la distancia recorrida por el robot. A pesar de estas limitaciones, la odometría es una parte importante del sistema de navegación de un robot y debe usarse con medidas del posicionamiento absolutas para proporcionar una estimación de la posición más fiable.

La odometría se basa en ecuaciones simples que se pueden implementar fácilmente y que utilizan datos de *encoders* situados en las ruedas del robot. Sin embargo, la odometría también está basada en la suposición de que las revoluciones de las ruedas pueden ser traducidas en un desplazamiento lineal relativo al suelo. Esta suposición no tiene una validez absoluta. Un ejemplo extremo se produce cuando las ruedas patinan: si por ejemplo, una rueda patina sobre una mancha de aceite y la otra no, entonces el *encoder* asociado registrará revoluciones en la rueda, a pesar de que éstas no correspondan a un desplazamiento lineal de la rueda. Además de este ejemplo hay muchas otras razones más sutiles por las que se pueden producir imprecisiones en la

traducción de las lecturas del *encoder* de la rueda a un desplazamiento lineal. Todos estos errores se pueden agrupar en dos categorías: errores sistemáticos y errores no sistemáticos.

Entre los **errores sistemáticos** destacan:

- Los diámetros de las ruedas no son iguales.
- La media de los diámetros de las ruedas difieren del diámetro de fábrica de las ruedas.
- Mal alineamiento de las ruedas.
- Resolución discreta (no continua) del *encoder*.
- La tasa de muestreo del *encoder* es discreta.

Entre los **errores no sistemáticos** se encuentran:

- Desplazamiento en suelos desnivelados.
- Desplazamiento sobre objetos inesperados que se encuentren en el suelo.
- Patinaje de las ruedas debido a:
  - Suelos resbaladizos.
  - Sobre-aceleración.
  - Derrapes (debidos a una rotación excesivamente rápida).
  - Fuerzas externas (interacción con cuerpos externos).
  - No hay ningún punto de contacto con el suelo.

Una clara distinción entre errores sistemáticos y no sistemáticos es de gran importancia a la hora de reducir los errores en la odometría. Los errores sistemáticos son específicamente graves, porque se acumulan constantemente. En muchas superficies no rugosas de entornos interiores, los errores sistemáticos contribuyen a producir más errores en la odometría que los errores no sistemáticos. Sin embargo, en superficies que agarran bien con irregularidades significativas, son los errores no sistemáticos los que predominan. El problema de los errores no sistemáticos es que pueden aparecer inesperadamente (por ejemplo cuando el robot pasa por encima de un objeto que se encuentra en el suelo) y pueden causar errores muy grandes en la estimación de la posición.

4. Técnicas y herramientas

En este apartado se realizará un pequeño resumen de los lenguajes de programación utilizados en el desarrollo del proyecto y de las herramientas de software y hardware que se han empleado.

4.1 Lenguajes de programación

Un lenguaje de programación es un idioma artificial diseñado para expresar procesos que pueden ser llevados a cabo por máquinas como las computadoras. Pueden usarse para crear programas que controlen el comportamiento físico y lógico de una máquina, para expresar algoritmos con precisión o como modo de comunicación humana.

Está formado por un conjunto de símbolos y reglas sintácticas y semánticas que definen su estructura y el significado de sus elementos y expresiones. Al proceso por el cual se escribe, se prueba, se depura, se compila y se mantiene el código fuente de un programa informático se le llama programación.

4.1.1 Objective-C

Objective-C es un lenguaje de programación orientado a objetos creado como un superconjunto de C para que implementase un modelo de objetos parecido al de Smalltalk. Originalmente fue creado por Brad Cox y la corporación StepStone en 1980. En 1988 fue adoptado como lenguaje de programación de NEXTSTEP y en 1992 fue liberado bajo licencia GPL (*GNU General Public License*, «Licencia Pública General de GNU») para el compilador GCC (*GNU Compiler Collection*, «Colección de compiladores GNU». Actualmente se usa como lenguaje principal de programación en Mac OS X, iOS y GNUstep. Este lenguaje ha sido utilizado para desarrollar la aplicación QrNav para dispositivos iOS.

Objective-C consiste en una capa muy fina situada por encima de C y, además, es un estricto superconjunto de C. Esto significa que es posible compilar cualquier programa escrito en C con un compilador de Objective-C y también puede incluir libremente código en C dentro de una clase de Objective-C.

El modelo de programación orientada a objetos de Objective-C se basa en enviar mensajes a instancias de objetos. Esto es diferente al modelo de programación al estilo de Simula, utilizado por C++, y esta distinción es semánticamente importante. En Objective-C uno no llama a un método; uno envía un mensaje, y la diferencia entre ambos conceptos radica en cómo el código referido por el nombre del mensaje o método es ejecutado. En un lenguaje al estilo Simula, el nombre del método es, en la mayoría de los casos, atado a una sección de código en la clase objetivo por el compilador, pero en Smalltalk y Objective-C, el mensaje sigue siendo simplemente un nombre y es resuelto en tiempo de ejecución.

4.1.2 C++

C++ es un lenguaje de programación diseñado a mediados de los años 1980 por Bjarne Stroustrup. La intención de su creación fue extender al exitoso lenguaje de programación C con mecanismos que permitan la manipulación de objetos. En ese sentido, desde el punto de vista de los lenguajes orientados a objetos, el C++ es un lenguaje híbrido. Este lenguaje ha sido el empleado para el desarrollo del servidorQrNav para Debian.

Posteriormente se añadieron facilidades de programación genérica, que se sumó a los otros dos paradigmas que ya estaban admitidos (programación estructurada y la programación orientada a objetos). Por esto se suele decir que C++ es un lenguaje de programación multiparadigma.

Una particularidad de C++ es la posibilidad de redefinir los operadores y de poder crear nuevos tipos que se comporten como tipos fundamentales.

4.2 Herramientas utilizadas

4.2.1 XCode 4. SDK

XCode es el conjunto completo de herramientas para el desarrollo de OS X y las aplicaciones de iOS, o dicho de otra manera, el entorno de desarrollo integrado (*IDE*) de Apple para facilitar el trabajo a los desarrolladores de su plataforma. Se suministra gratuitamente junto con Mac OS X.

XCode puede compilar código C, C++, Objective-C, Objective-C++, Java y AppleScript mediante una amplia gama de modelos de programación, incluyendo, pero no limitado a Cocoa, Carbon y Java.

Entre las características más apreciadas de XCode está la tecnología para distribuir el proceso de construcción a partir de código fuente entre varios ordenadores, utilizando Bonjour.

Tiene diversas herramientas como Instruments para detectar *memory leaks* (fugas de memoria) , que nos permiten optimizar nuestra aplicación en el tema de reserva de memoria, hecho que sin duda es uno de los mayores quebraderos de cabeza en la programación para dispositivos iOS puesto que no disponen de recolector de basura.

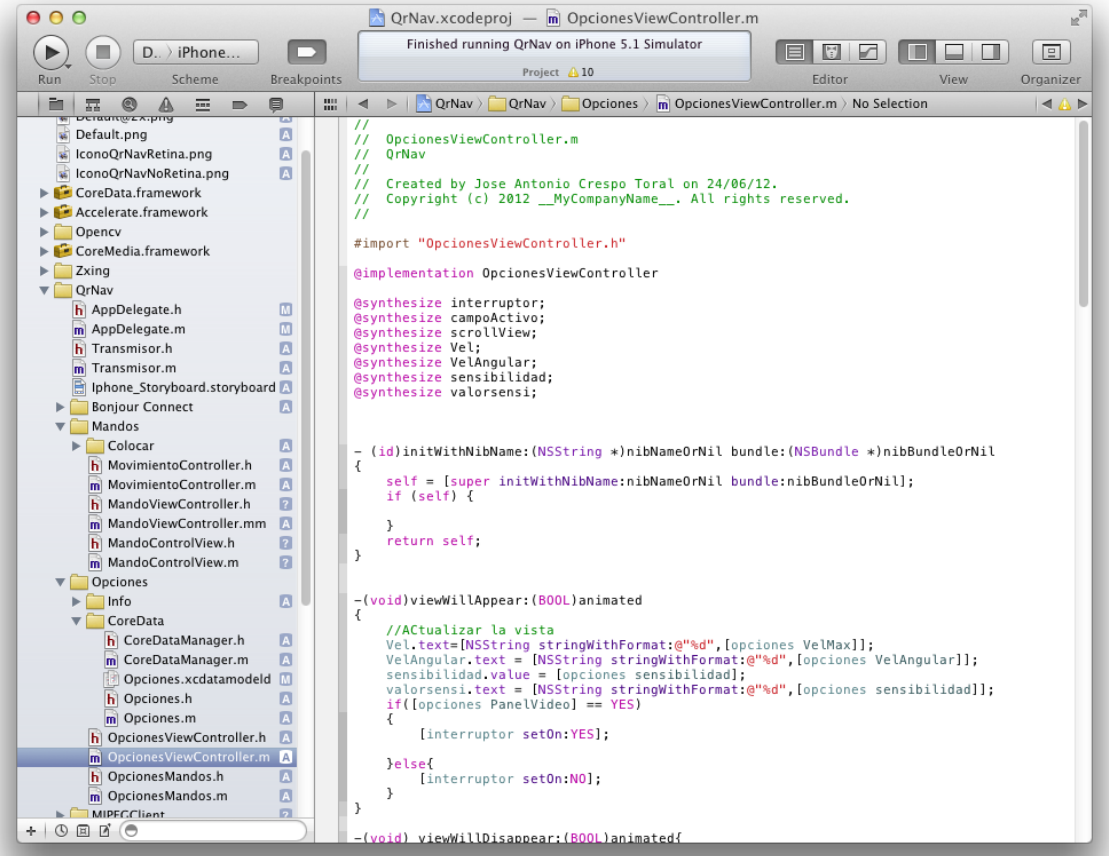

Figura 25 - XCode 4

En el proyecto se ha trabajado con XCode 4 (Figura 25) que permite desarrollar aplicaciones para la versión iOS 5, que incluye importantes cambios. Con XCode 4, las herramientas se han rediseñado para ser más rápido, más fácil de usar y resultar más útil que nunca. XCode comprende cada detalle del proyecto que se esté realizando, identifica errores (tanto sintácticos como lógicos) e incluso sugiere soluciones en los errores cometidos. En pocas palabras, XCode 4 ayuda a escribir un código mejor.

XCode 4 tiene una nueva interfaz de usuario, construida con tecnologías que la propia Apple utiliza para crear OS X y iOS y que han producido más de un cuarto de millón de aplicaciones OS X y iOS.

En XCode 4, Interface Builder (IB), aplicación con la que se crean nuestros archivos *Xib* para diseñar las interfaces gráficas de la aplicación, ha sido completamente integrado en XCode, tal y como se muestra en la Figura 26.

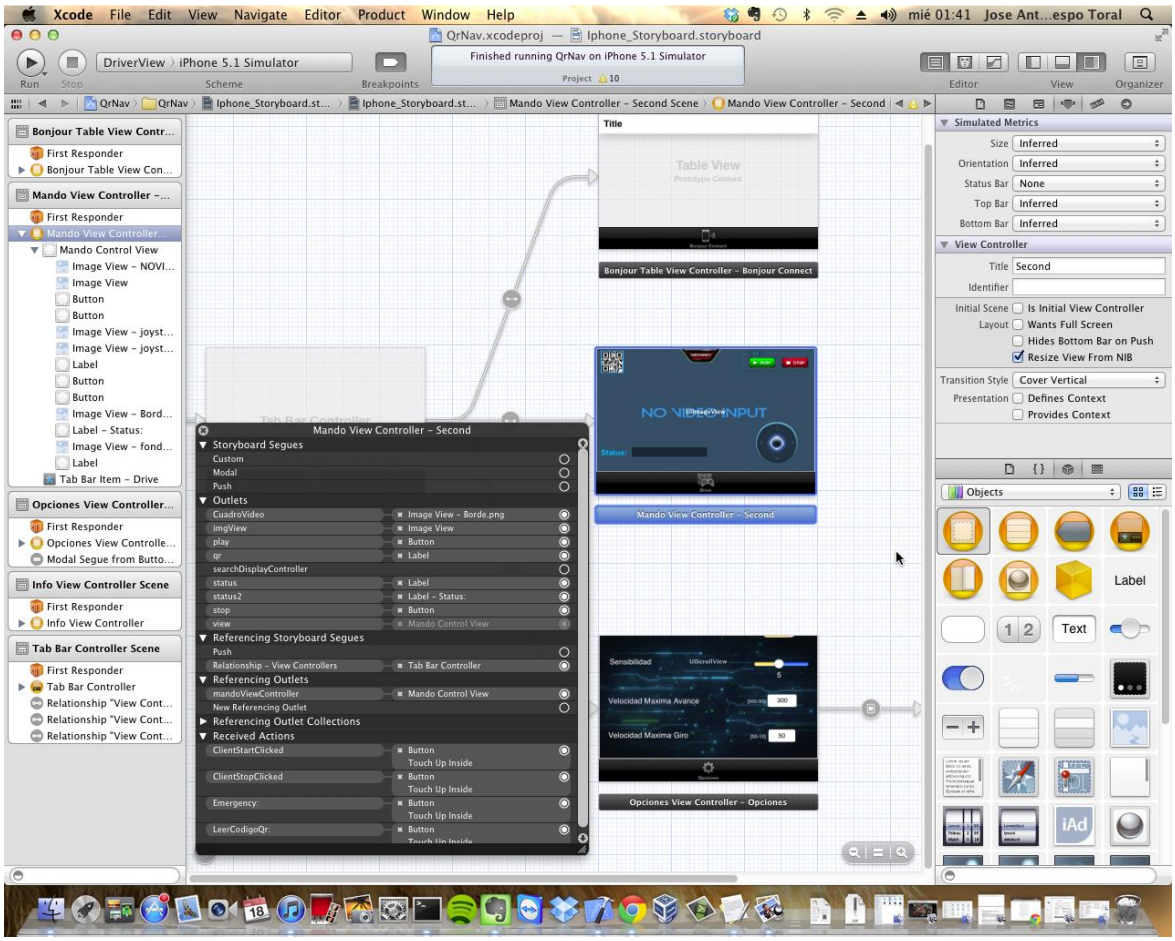

Figura 26 - Interface Builder integrado en XCode

Si seleccionamos un archivo de interfaz (*Nib* / *Xib*) de un proyecto, se abrirá el editor de IB en XCode y un área de útiles a la derecha que mostrará la gama completa de la interfaz del *inspector*, así como la librería de controles y objetos de la interfaz de usuario. Para diseñar una interfaz, sólo tenemos que arrastrar un control de la librería o subview hasta la *view* donde queremos colocarlo (también permite cambiar sus atributos).

Lo mejor de todo es que se pueden realizar las conexiones arrastrando directamente desde el diseño de interfaz de usuario hasta el código fuente. Esto se produce gracias al nuevo diseño de XCode 4 para editar que, mediante la opción de editar separando dos archivos en la misma pantalla, podemos establecer de forma fácil la conexión entre las *actions* y los *outlets* con un único gesto. Además, si aún no tenemos escrito el código al que poder conectarnos, sólo tenemos que arrastrar desde el IB hasta un espacio en blanco del código y XCode creará un nueva *action* o *outlet* generando el código.

Entre otras características, XCode 4 ha mejorado la organización de los proyectos al incluir: un panel de navegación, realizar sugerencias para solucionar errores de forma automática, un nuevo depurador LLVM (*Low Level Virtual Machine*, «Máquina Virtual de Bajo Nivel») diseñado desde cero para consumir mucha menos memoria y ser un bastante más rápido en cuanto a rendimiento, han mejorado instrument, etc.

En último lugar, comentar que el iPhone *SDK* es un Kit de desarrollo de software para la plataforma iOS de Apple, que permite a terceros desarrollar aplicaciones nativas para iOS. En el proyecto se ha empleado el *SDK* 5.1 que era la última versión existente en el momento de comenzar con el proyecto.

4.2.2 ARIA

ARIA[3] (*Advanced Robot Interface for Applications*) de MobileRobots, es una biblioteca en C + + (kit de herramientas de desarrollo de software o SDK) para todas las plataformas MobileRobots / ActivMedia, que nos permite controlar dinámicamente la velocidad del robot, la partida, la partida relativa y otros parámetros de movimiento a través de simples comandos de bajo nivel o a través de su alto nivel. *ARIA* también recibe estimaciones de posición, las lecturas de sonar y otros datos operativos enviados por la plataforma del robot.

Otras herramientas útiles para la construcción de aplicaciones de robótica están incluidas en *ARIA* o se encuentran disponibles como bibliotecas independientes, incluyendo síntesis de voz y reconocimiento, la reproducción de efectos de sonido, funciones matemáticas, multiplataforma (Windows / Linux), implementaciones de hilos y sockets, etc.

El acceso a la mayor parte de la *API* de *ARIA* también está disponible en los idiomas de Java y Python. El desarrollo de C++ con *ARIA* es compatible con *GNU* / Linux con G++ y en Windows con MS Visual C++.

ARIA viene con un código fuente completo bajo la Licencia Pública General de *GNU*. La licencia permite la redistribución del código, siempre y cuando todo se distribuya gratuitamente. *ARIA* incluye un manual de referencia de la *API* completa y códigos de ejemplos.

Como ya se ha mencionado, *ARIA* incluye una serie de *APIs* para programación en distintos niveles, por lo que podemos acceder directamente a nuestro robot usando la clase *ArRobot* y mandarle órdenes directas como pueden ser “cambiar la velocidad” o “cambiar la velocidad de giro”.

Por otra parte, *ARIA* dispone de una serie de mecanismos llamados *Actions* que nos permiten trabajar con el robot a un nivel más alto de desarrollo, más cercano a la inteligencia artificial.

En el proyecto QrNav, se ha optado por la primera opción por contar con los requisitos suficientes para nuestro caso ya que nos permite de forma rápida y sencilla dar órdenes al robot.

Para comprender el uso del robot, se van a detallar aspectos básicos y necesarios para entender la parte técnica del proyecto.

[3] http://robots.mobilerobots.com/wiki/ARIA

En primer lugar tendremos que conectar con *ARIA* y, posteriormente, conectar con el Amigobot:

```
Aria::init();
ArSimpleConnector connecto(&argc, argv);
ArRobot robot;
Connector.connectRobot(&robot);
```

Tabla 3 - Conectar con Aria y AmigoBot

La función intentará conectar con un robot. Nuestro Amigobot puede estar conectado a distintos puertos o ser un robot virtual (como es el simulador MobileSim ofrecido por *ARIA*). El servidor conectará con un robot real o virtual dependiendo de los parámetros pasados a éste al ejecutar la aplicación.

Una vez establecida la conexión, podemos invocar diversas funciones para modificar un parámetro del robot u obtener algún valor de sus sensores.

```
robot.setVel(Velocidad);
robot.setRotVel(velocidadAngular);
X = robot.GetX();
Y = robot.GetY();
Th = robot.GetTh();
```

Tabla 4 - Funciones de ARIA

4.2.3 ZXing

ZXing[4] (pronunciado "zebra crossing") es un código abierto, multi-formato de imagen de código de barras 1D/2D y biblioteca de procesamiento implementado en Java con soporte a otros lenguajes. La biblioteca se centra en utilizar la cámara incorporada en los teléfonos móviles para escanear y decodificar códigos de barras en el dispositivo, sin necesidad de comunicarse con un servidor. Sin embargo, el proyecto se puede utilizar para codificar y decodificar códigos de barras en los escritorios y también en los servidores. Actualmente, se admiten los siguientes formatos:

- UPC-A y UPC-E
- EAN-8 y EAN 13
- Código 39
- Código 93
- Código 128
- ITF
- Codabar
- RSS-14 (todas las variantes)
- Código QR
- Data Matrix
- Azteca ("beta" de calidad)
- PDF 417 ('alpha' de calidad)

La elección de esta biblioteca para decodificar los códigos QR se realizó teniendo en cuenta la funcionalidad que ésta ofrecía y la que el proyecto necesitaba (decodificar

[4] http://code.google.com/p/zxing/

un código QR de una UIImageView). En apartados posteriores se explicará con más detalle esta elección.

Esta biblioteca se divide en varios componentes principales:

- **Core**: La biblioteca de imágenes de decodificación y el código de prueba.
- **Javase**: J2SE código específico del cliente.
- **Zxingorg:** El código en zxing.org/w.
- **Android**: El cliente para Android, llamado BarcodeScanner.
- **Androidtest**: Android test app.
- **Android-integration**: Permite la integración con la aplicación Barcode Scanner a través de un Intent.

También hay módulos adicionales que se aportan y son mantenidos de forma intermitente:

- **cpp**: Parcial C + +
- **iPhone**: iPhone cliente para Objective C / C + + (código QR solamente)
- **zxing.appspot.com**: La fuente de nuestro generador de código de barras basado en la web
- **csharp**: Parcial C #
- **JRuby**

Hay, por último, algunos módulos que ya no se mantienen, sino que están disponibles en versiones anteriores, como son:

- **JavaME** : el cliente JavaME
- **aro** : RIM / Blackberry específicos del cliente a construir

4.2.4 MJPG-Streamer

MJPG-streamer[5] es una aplicación de línea de comandos para realizar un stream de archivos JPEG (*Joint Photographic Experts Group*, «Grupo Conjunto de Expertos en Fotografía») a través de una red basada en *IP*, desde una webcam a clientes como Firefox[6], VLC[7], un dispositivo Windows Mobile o teléfono móvil que tenga un navegador web.

[5] http://sourceforge.net/projects/mjpg-streamer/
[6] http://www.mozilla.org/es-ES/firefox/new/
[7] http://www.videolan.org/vlc/

La selección de este programa como servidor de imágenes se debe a que permite enviar un flujo de imágenes obtenidas desde la webcam a los clientes en tiempo real. En apartados siguientes se explicará como se llegó a esta decisión.

Se puede hacer uso de la compresión de hardware de ciertas webcams con el fin de reducir los ciclos de la *CPU* (*Central Processing Unit*, «unidad central de procesamiento») del servidor. Esto hace que sea una solución con menos ciclos de *CPU* para dispositivos embebidos y servidores regulares, que no deben pasar la mayor parte de su potencia de cálculo en la compresión de *videoframes*. Por ejemplo, se consume menos del 10% de *CPU* en un equipo de 200 MHz cuando se transmite un vídeo de 960x720 píxeles.

Este programa cuenta con unos *plugin* de entrada (*input-plugin*) y unos *plugin* de salida (*output-plugins*). Los *input-plugin* obtienen imágenes y las copian en memoria. Hay varios *output-plugins* que recogen esas imágenes y las ofrecen por varias salidas. El más popular es el servidor web, que permite el transporte de las imágenes a un navegador en el que se pueden visualizar.

El trabajo de *MJPG-streamer* consiste en unir el único *input-plugin* con múltiples *output-plugins* (casi todas las acciones importantes son llevadas a cabo por los *plugin*).

Podemos encontrar diversos *input-plugin*, todos ellos tienen una parte común, y es que copian las imágenes *JPEG* a una memoria de acceso global y se lo indican a los procesos que están esperando. Diferenciamos entre:

- **input_testpicture.so**, este *plugin* de entrada obtiene las imágenes de disco y se puede utilizar sin una cámara web. Convierte la *testpictures* de archivos *JPEG* en un archivo de cabecera. Los archivos de cabecera contienen las imágenes compiladas en el *input-plugin*. Una vez activado, el *plugin* sirve estos compilados de imágenes en un bucle.

- **input_uvc.so**, este *plugin* de entrada obtiene las imágenes de un dispositivo Linux-UVC V4L2 compatible, como por ejemplo la webcam usada en el proyecto (Logitech Quickcam Pro 9000). El código fuente de este *plugin* está basado en "*luvcview*". En contraste con la "*luvcview*" inicializa los nuevos comandos de Logitech como pan/tilt/focus sin la necesidad de una biblioteca auxiliar o de manipular archivos *XML*, etc. Este *plugin* es el más interesante, ya que puede transmitir imágenes de hasta 960x720 desde la webcam a una velocidad elevada (>=15 fps) sin carga de la *CPU* mencionable. Si no nos preocupa la carga de *CPU*, se puede incluso grabar imágenes grandes de 1600x1200 sin comprimir, realizar la compresión en software y transmitir a los clientes.

- **input_control.so**, este *plugin* de entrada sólo implementa la interfaz de control para realizar un giro horizontal/vertical de la webcam, dejando a la secuencia de vídeo disponible para otras aplicaciones, como por ejemplo para que otros interlocutores puedan controlar los giros de la webcam

mientras se realiza una videoconferencia por Skype[8] (que se encarga de realizar el stream de vídeo y audio).

Por otra parte encontramos dos *output-plugin* como son:

- **output_http.so,** este *plugin* es un servidor web completamente funcional de *HTTP* 1.0. Puede transmitir las imágenes obtenidas de un *input-plugin* a diversos clientes y se puede configurar con ciertos comandos.
- **output_file.so,** este *plugin* se utiliza para almacenar las imágenes *JPEG* del *plugin* de entrada en una carpeta especificada. Puede ser utilizado para tomar imágenes y simplemente almacenar, o para enviar a una cuenta FTP (*File Transfer Protocol*, «Protocolo de Transferencia de Archivos») mediante la ejecución de un comando después de guardar la imagen.

4.2.5 OpenCV

OpenCV[9] (*Open source Computer Vision library*) es una biblioteca libre de visión artificial originalmente desarrollada por Intel. Esta librería proporciona un alto nivel de funciones para el preprocesado de imágenes y permite a los programadores crear aplicaciones poderosas en el dominio de la visión digital. Se optó por utilizar esta biblioteca por ser una de las más potentes en el tratamiento de imágenes y constar de una gran funcionalidad. Esta elección se detallará en apartados posteriores.

Desde que apareció en 1999 se ha utilizado en infinidad de aplicaciones, desde sistemas de seguridad con detección de movimiento, hasta aplicativos de control de procesos donde se requiere reconocimiento de objetos. Esto se debe a que su publicación se da bajo licencia BSD (*Berkeley Software Distribution*), que permite que sea usada libremente para propósitos comerciales y de investigación con las condiciones en ella expresadas.

OpenCV es multiplataforma, existiendo versiones para *GNU*/Linux, Mac OS X y Windows. Contiene más de 500 funciones que implementan una gran variedad de herramientas para la interpretación de la imagen. Es compatible con Intel *Image Processing Library* (IPL) que implementa algunas operaciones en imágenes digitales. A pesar de primitivas como binarización, filtrado, estadísticas de la imagen, pirámides, etc. *OpenCV* es principalmente una librería que implementa algoritmos para las técnicas de:

- Calibración (Calibración de la Cámara)
- Detección de rasgos
- Para rastrear (Flujo Óptico)
- Análisis de la forma (Geometría, Contorno que Procesa)

[8] http://www.skype.com/intl/es-es/home
[9] http://opencv.org/

- Análisis del movimiento (Plantillas del Movimiento, Estimadores)
- Reconstrucción 3D (Transformación de vistas)
- Segmentación de objetos
- Reconocimiento (Histograma, etc.)
- Reconocimiento de objetos (Reconocimiento facial, etc.)
- Visión estéreo
- Visión robótica
- Etc.

El rasgo esencial de la librería, junto con funcionalidad y la calidad, es su desempeño. Los algoritmos están basados en estructuras de datos muy flexibles y están acoplados con estructuras *IPL*. Más de la mitad de las funciones han sido optimizadas aprovechando la arquitectura de Intel.

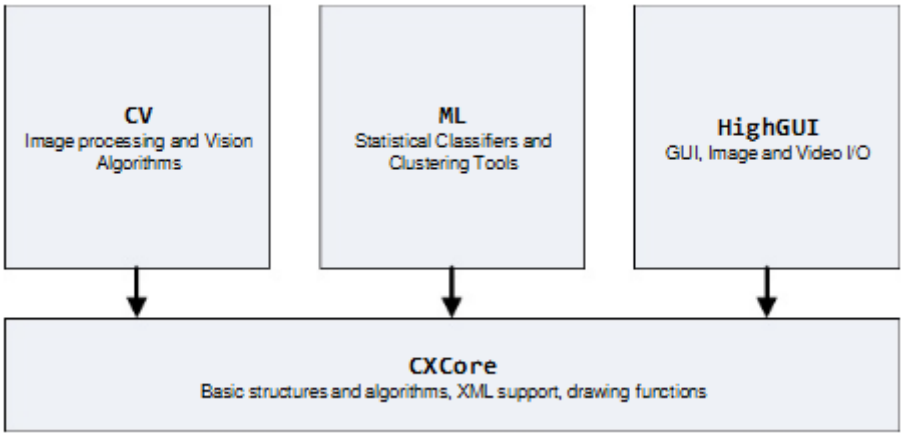

Figura 27 - Estructura de *OpenCV*

La estructura básica de *OpenCV* (Figura 27) se divide en cuatro componentes principales y cada uno de ellos contiene un cierto conjunto de funcionalidad:

- **CV,** contiene la funcionalidad para el procesamiento básico de imágenes y la manipulación, así como algoritmos de visión por computador.
- **ML,** contiene toda la funcionalidad de la máquina de aprendizaje que incluye la agrupación y clasificadores estadísticos.
- **HighGUI,** es una interfaz gráfica muy importante porque se necesita bajo *OpenCV* para visualizar imágenes y, además, contiene funciones para cargar y guardar imágenes y vídeos.

- **CXCore,** es un componente común entre todos los otros componentes, ya que contiene toda la base de datos y las estructuras de uso común de los otros componentes.

OpenCV usa la estructura *IplImage* para crear y manejar imágenes. Esta estructura tiene gran cantidad de campos, algunos de ellos son más importantes que otros. Por ejemplo, *Width* es la anchura de la imagen, *Height* es la altura, *Depth* es la profundidad en bits y *nChannels* el número de canales (uno para imágenes en escala de grises y tres para las imágenes en color).

Hay varios tipos de datos fundamentales como:

- *IplImage* (imagen de IPL), *CvMat* (matriz),
- Growable collections:
  - *CvSeq* (deque)
  - *CvSet*
  - *CvGraph*
- Tipos mixtos:
  - *CvHistogram* (histograma multi-dimensional)
- Tipos de datos auxiliares incluyen:
  - *CvPoint* (2d punto)
  - *CvSize* (anchura y altura)
  - *CvTermCriteria* (criterio de la terminación para los procesos iterativos)
  - IplConvKernel (núcleo de la circunvolución)
  - *CvMoments* (momentos espaciales)
  - etc.

Entre sus funciones podemos encontrar: *cvCanny* (aplica el algoritmo de Canny para la detección de los bordes en la imagen), *cvFindContours* (encuentra contornos en una imagen binaria), *CloneImage* (crea una copia de la imagen), etc.

La biblioteca pretende proporcionar un entorno de desarrollo fácil de utilizar y altamente eficiente. Esto se ha logrado realizando la programación en código C y C++ optimizados, aprovechando además las capacidades que proveen los procesadores multinúcleo. *OpenCV* puede además utilizar el sistema de primitivas de rendimiento integradas de Intel, un conjunto de rutinas de bajo nivel específicas para procesadores Intel.

4.2.6 ZeroConf. Bonjour. Avahi

Zeroconf[10] o *Zero Configuration Networking* es un conjunto de técnicas que permiten crear de forma automática una red *IP* sin configuración o servidores especiales. También conocida como *Automatic Private IP Addressing* o APIPA, permite a los usuarios sin conocimientos técnicos conectar ordenadores, impresoras de red y otros elementos y hacerlos funcionar. Sin *Zeroconf*, un usuario con conocimientos técnicos debe configurar servidores especiales, como DHCP (*Dynamic Host Configuration Protocol*, «protocolo de configuración dinámica de host») y DNS (*Domain Name System*, «sistema de nombres de dominio») o bien configurar cada ordenador de forma manual.

Zeroconf fue explotado por Stuart Cheshire, empleado en Apple Computer, durante su migración de AppleTalk a *IP*.

Actualmente *Zeroconf* soluciona tres problemas:

- Selecciona una dirección *IP* para los elementos de red.
- Descubre qué ordenador tiene un determinado nombre.
- Descubre dónde se encuentran los servicios, como el de impresión.

Además, *Zeroconf* se basa en dos protocolos:

- ***Multicast DNS*** **(mDNS),** es un protocolo que nos permite, dado el nombre de un dispositivo en una LAN (*Local Area Network*, «red de área local»), conocer su dirección *IP*. Internamente se encargará de mandar una serie de señales que el receptor correspondiente nos responderá, obteniendo así su dirección.

- ***DNS based Service Discovery*** **(DNS-SD**), es el protocolo que nos permite publicar un servicio en la red (por ejemplo el servicio ofrecido por una impresora) de manera que podamos pedir una lista de los proveedores de dicho servicio.

El *host* que ofrece el servicio publica los detalles sobre el servicio disponible: tipo, nombre, parámetros (opcionales) etc. El cliente buscará servicios de un tipo determinado y finalmente preguntará por la *IP* del servicio seleccionado.

Zeroconf es una iniciativa que nos propone una manera de hacer las cosas, no una implementación específica. Hay por tanto varias implementaciones de *Zeroconf*, como pueden ser Bonjour para Mac OS o Avahi para Linux que describiremos a continuación:

[10] http://www.zeroconf.org/

- **Bonjour**[11] (anteriormente **Rendezvous**), es un programa de Apple para la implementación de la especificación de la IETF[12] (*Internet Engineering Task Force*) del marco de trabajo *Zeroconf*. Bonjour usa paquetes estándar de *DNS* de una nueva forma, de modo que es otro servicio, pero se basa en una tecnología relativamente antigua: *DNS* sobre *IP*.

- **Avahi**[13]**,** es una implementación libre de *Zeroconf*, incluyendo un sistema de multidifusión *DNS / DNS-SD* de descubrimiento de servicios. Avahi permite a los programas publicar y descubrir servicios y servidores que se ejecutan en una red local sin una configuración específica.

4.2.7 Proceso Unificado

Para la elaboración del proyecto se ha elegido el Proceso Unificado como marco de desarrollo software, ya que es uno de los más extendidos y mejor documentados, además de haber demostrado su eficacia para satisfacer las necesidades del usuario final.

4.2.8 Metodología para la elicitación de requisitos

Para la realización de documentación de requisitos, se siguió la metodología propuesta por Durán Toro & Bernárdez Jiménez en 2002, desarrollada en el Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Sevilla.

Esta metodología se centra en la utilización de diagramas de casos de uso para representar los requisitos funcionales de nuestro sistema software. También destaca el uso de plantillas para describir los requisitos del proyecto.

4.2.9 Lenguaje Unificado de Modelado (UML)

En el apartado de notación del desarrollo de proyecto software se ha escogido el Lenguaje Unificado de Modelado (LUM o UML, por sus siglas en inglés, Unified Modeling Language), que es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad. Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema.

Figura 28 - UML

[11] http://www.apple.com/es/support/bonjour/
[12] http://www.ietf.org/
[13] http://avahi.org/

UML ofrece un estándar para describir un "plano" del sistema (modelo) incluyendo aspectos conceptuales tales como procesos de negocio, funciones del sistema y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y compuestos reciclados.

Es importante remarcar que *UML* es un "lenguaje de modelado" para especificar o para describir métodos o procesos. Se utiliza para definir un sistema, para detallar los artefactos en el sistema y para documentar y construir. En otras palabras, es el lenguaje en el que está descrito el modelo.

La razón de haber elegido *UML* para el proyecto es porque se trata de un lenguaje de modelado de objetos abierto y unificado, además de ser el estándar actual. De esta manera se cubren las distintas vistas del proceso de construcción del sistema a lo largo de las distintas etapas de desarrollo.

4.2.10 Visual Paradigm

Visual Paradigm es una herramienta *UML* profesional que soporta el ciclo de vida completo del desarrollo de software: análisis y diseño orientados a objetos, construcción, pruebas y despliegue.

El software de modelado *UML* ayuda a una más rápida construcción de aplicaciones de calidad, mejores y a un menor coste. Permite dibujar todos los tipos de diagramas de clases, código inverso, generar código desde diagramas y generar documentación. La herramienta *UML CASE* también proporciona abundantes tutoriales de *UML*, demostraciones interactivas de *UML* y proyectos *UML*.

Este programa se ha utilizado para la realización de la mayoría de los diagramas, sobre todo en los que aparecen en los anexos de la documentación del proyecto.

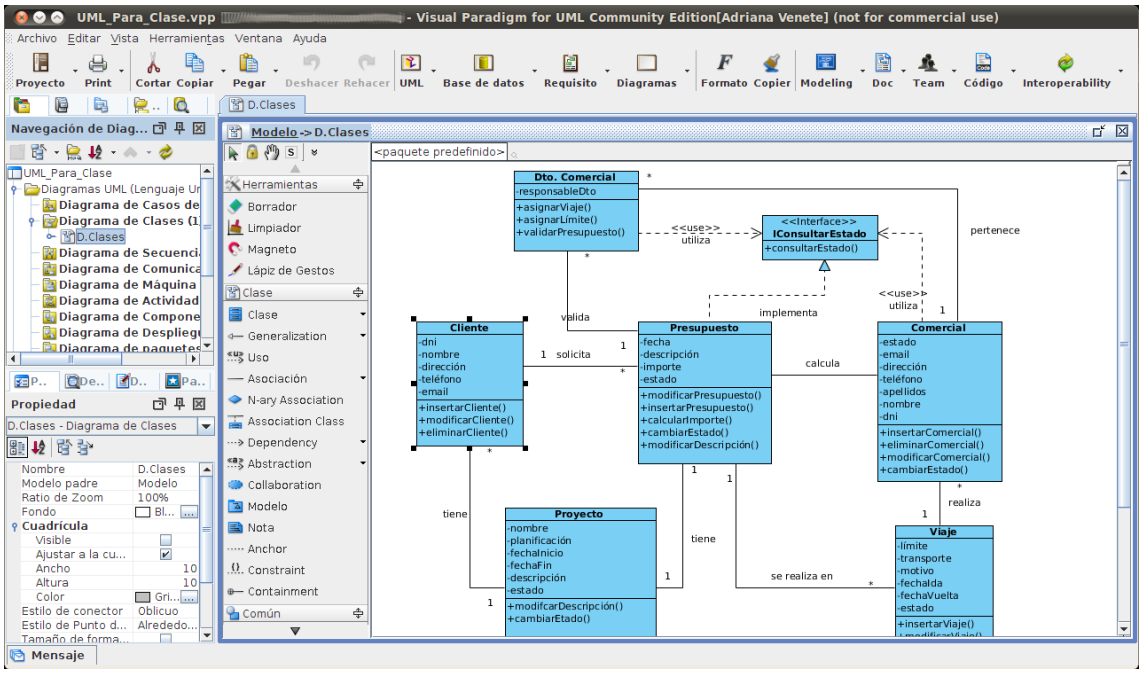

Figura 29 - Visual Paradigm

4.2.11 Microsoft Word

Microsoft Word es un software dedicado al procesamiento de textos. También podríamos definirlo como un programa interactivo que permite comprender y modificar de manera instantánea los textos que en él se visualizan. Fue creado por la empresa Microsoft, y actualmente viene integrado en la *suite* ofimática Microsoft Office. Hoy día, Microsoft Word es el programa estrella de Microsoft Office.

Originalmente fue desarrollado por Richard Brodie para el computador de IBM bajo el sistema operativo DOS en 1983. Ha llegado a ser el procesador de texto más popular del mundo.

Este programa se ha utilizado para la elaboración de la memoria y los anexos adjuntos en la documentación del proyecto.

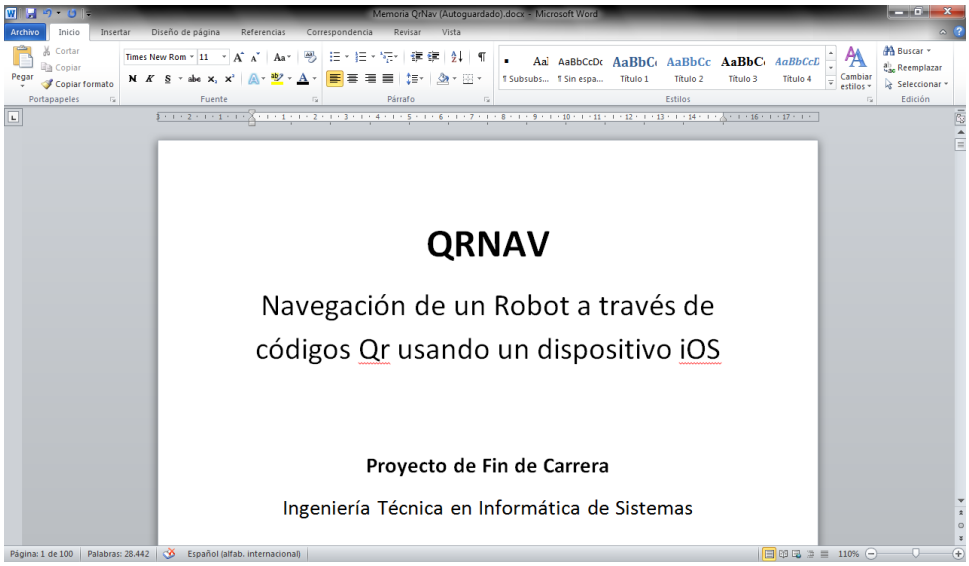

Figura 30 - Microsoft Word

4.2.12 iPhone 4

El iPhone 4 es la 4ª generación del dispositivo de telefonía móvil de Apple que fue lanzado al mercado en 2010.

Figura 31 - iPhone 4

Destaca la renovación estética frente al iPhone 3G y 3GS con una forma más poligonal y una parte un frontal y trasera planos realizados en vidrio sobre un chasis de acero. Destaca también el estreno del sistema operativo iOS 4, aunque actualmente ya dispone del iOS 5. Este dispositivo incluye multitasking, fondos de pantalla en pantalla de inicio, cámara de 5 MP, etc.

Esta herramienta ha sido utilizada para instalar y probar la aplicación cliente en un dispositivo físico real, en lugar del simulador iOS de XCode, así como para el desarrollo de pruebas y la búsqueda de diversos fallos.

4.2.13 Amigobot

El Amigobot de MobileRobots (Figura 32) es un robot de pequeño coste para proyectos de educación y colaboración. Cuenta con la misma arquitectura que plataformas más grandes, que funcionan con las librerías *ARIA* y SONARNL[14], pero esta plataforma simplificada está optimizada para su uso económico en las aulas.

El Amigobot llega totalmente montado y cuenta con 8 sensores de sonar. El Amigobot puede funcionar en modo conectado al ordenador, de manera inalámbrica con los accesorios de comunicación opcionales. La cubierta de plástico de alto impacto y marco de aluminio proporciona una plataforma diseñada para años de uso en las aulas.

[14] http://robots.mobilerobots.com/wiki/ARNL,_SONARNL_and_MOGS

Figura 32 - Amigobot

La plataforma base del Amigobot incluye el software que le permite:

- Ser manejado con teclas o *joystick*.
- Navegar automáticamente por sí mismo.
- Mostrar un mapa con las lecturas de sus sonar.
- Localizar un objeto en la trayectoria usando un sonar.
- Obtener y comunicar información sobre el sonar, el codificador del motor, controles de motor, los usuarios de E/S y la carga de la batería.
- Funcionar con programas en C o C++.
- Simular el comportamiento del AmiboBot mediante el simulador ofrecido por *ARIA*.

Este es el robot que ha sido proporcionado por el Departamento de Informática y Automática de la Universidad de Salamanca para poder llevar a cabo el proyecto.

4.2.14 MobileSim

MobileSim es el simulador ofrecido por *ARIA* para poder realizar pruebas con un robot virtual que imita el comportamiento real del Amigobot.

Este simulador ha sido usado para realizar pruebas antes de ejecutar el proyecto con el robot real.

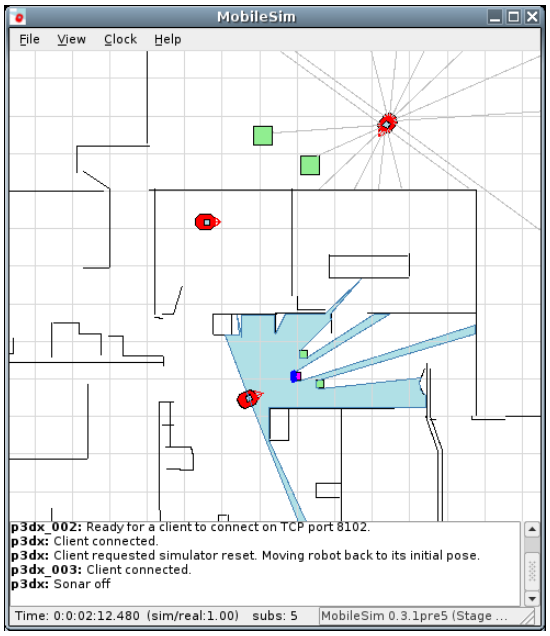

Figura 33 - Simulador AmigoBot

4.2.15 Webcam Logitech QuickCam Pro 9000

El éxito o no de un proyecto en el que intervenga un tratamiento de las imágenes por ordenador depende, en gran medida, de la calidad de la imagen sobre la que se trabaja.

Es por ello que en este proyecto se ha trabajado con una cámara de vídeo de contrastada válidez, suministrada por el Departamento de Informática y Automática de la Universidad Salamanca, la webcam Logitech[15] QuickCam Pro 9000.

Esta cámara es capaz de producir un vídeo fluido y natural e instantáneas de hasta 8 megapíxeles. Gracias a su enfoque automático de gama alta las imágenes son siempre nítidas, incluso en primeros planos (a 10 cm de la lente).

Esta cámara cuenta con las siguientes especificaciones:

- Óptica Zeiss ® con enfoque automático.
- Sensor nativo de alta resolución de 2 megapíxeles.
- Vídeo en alta definición (hasta 1600 x 1200*).
- Modo de pantalla panorámica de 720p (con sistema recomendado).
- Fotos de hasta 8 megapíxeles (mejoradas desde el sensor de 2 megapíxeles).
- Micrófono con tecnología Logitech RightSound.
- Vídeo de hasta 30 cuadros por segundo.
- Certificación USB 2.0 de alta velocidad.

[15] http://www.logitech.com/es-es

- Clip universal para monitores LCD (*Liquid Crystal Display*, «pantalla de cristal líquido») o portátiles

A continuación podemos ver en la Figura 34 la webcam utilizada para la obtención de imágenes en tiempo real proporcionada por la Universidad de Salamanca.

Figura 34 - Webcam Logitech QuickCam Pro 9000

4.2.16 DIA

DIA[16] es un editor de diagramas con las herramientas necesarias para crearlos o modificarlos sin apenas conocimientos (Figura 35).

Incluye herramientas de dibujo para introducir distintos elementos geométricos a nuestras composiciones, pudiendo editar sus propiedades y con un espacio cuadriculado para organizar nuestros diagramas y sistema de capas.

Permite abrir y exportar los dibujos realizados a los formatos más conocidos, además de tener su propio formato para editar el documento posteriormente.

Este programa se ha utilizado para la realización de los diagramas de flujo sobre los algoritmos realizados en el proyecto. El objetivo que se perseguía al incluirlos era mejorar la explicación y comprensión de dichos algoritmos.

[16] http://dia-installer.de/index.html.es

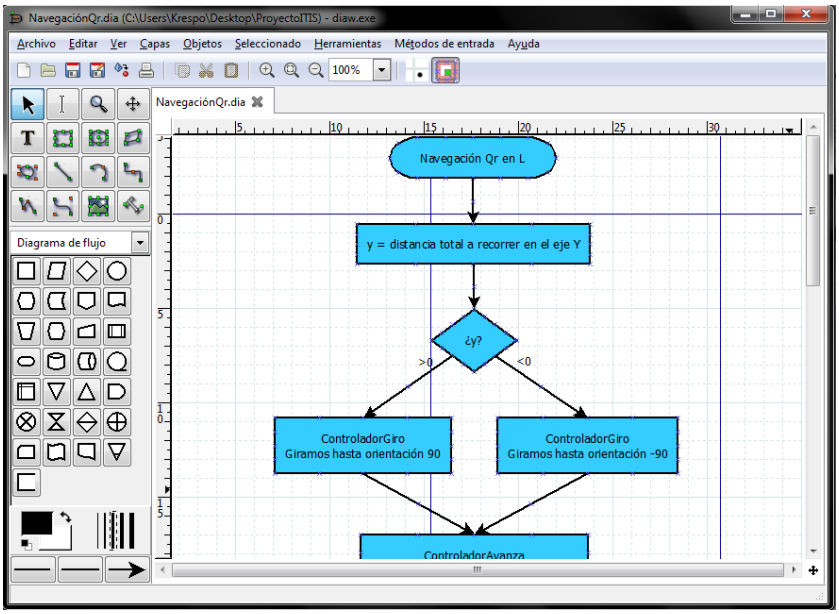

Figura 35 - DIA

4.2.17 Photoshop

Adobe Photoshop[17] es el nombre de uno de los programas más populares de Adobe Systems. Se trata esencialmente de una aplicación informática en forma de taller de pintura y fotografía que trabaja sobre un "lienzo" y que está destinado para la edición, retoque fotográfico y pintura a base de imágenes de mapa de bits (o gráficos rasterizados). Su capacidad de retoque y modificación de fotografías le ha dado el rubro de ser el programa de edición de imágenes más famoso del mundo.

Photoshop trabajaba con múltiples capas, donde se pueden aplicar toda una serie de efectos, textos, marcas, tratamientos, trazos, etc.

Photoshop se ha convertido en el estándar de facto en retoque fotográfico, pero también se usa extensivamente en multitud de disciplinas del campo del diseño y fotografía, como diseño web, composición de imágenes bitmap, estilismo digital, fotocomposición, edición y grafismos de vídeo y básicamente en cualquier actividad que requiera el tratamiento de imágenes digitales.

Entre las alternativas barajadas a este programa, existen algunos programas libres como GIMP[18], orientada a la edición fotográfica en general.

Elegir Photoshop como una herramienta más para el proyecto fue una decisión acertada, pues es el más popular del mercado y además tenía conociemientos previos sobre su uso.

[17] http://www.adobe.com/es/products/photoshop.html
[18] http://www.gimp.org.es/

Se ha usado para el desarrollo de la parte gráfica de la interfaz de usuario, botones, iconos, joystick, etc. y para la creación de los códigos QR que el robot decodifica tal y como se muestra en la Figura 36.

Figura 36 - Photoshop

4.2.18 Video For Linux 2

Video for Linux 2 [19](V4L2), es la segunda versión de la *API* de Video for Linux. Se trata de una interfaz de programación de aplicaciones (*API*) orientada a la captura de imágenes desde múltiples dispositivos, tales como tarjetas sintonizadoras de TV y webcams. Su principal objetivo es proveer de una interfaz común para una gran variedad de hardware, de manera tal que el código generado para una aplicación tenga un alto grado de portabilidad.

En 1999 Bill Dirks inició el desarrollo de *V4L2* para solucionar algunas deficiencias del *V4L* y para apoyar una amplia gama de dispositivos. La *API* se revisó de nuevo en el año 2002 antes de su inclusión en Linux 2.5/2.6 y se añadió un modo de compatibilidad para las aplicaciones *Video4Linux1*, aunque el soporte puede ser incompleto y se recomienda usar hardware *V4L2* en modo *V4L2*. Actualmente se sigue trabajando en mejoras y adiciones.

Algunas de las aplicaciones que tienen soporte para *V4L* son Skype, VLC y MPlayer, entre otras.

[19] http://linuxtv.org/downloads/v4l-dvb-apis/

Esta biblioteca se utilizó en un primer momento para obtener las imágenes de la Webcam en Debian y realizar el vídeo streaming, pero, posteriormente se desechó su uso debido al empleo de *MJPG-streamer* (aunque éste lo use internamente).

4.2.19 Zbar

Zbar[20] es un software de código abierto para la lectura de códigos de barras a partir de diversas fuentes, tales como vídeo streams, archivos de imágenes, etc. Es compatible con las simbologías (tipos de códigos de barras) más populares, incluyendo EAN-13/UPC-A, UPC-E, EAN-8, Code 128, Code 39, Código QR...

La implementación en capas facilita la lectura y decodificación de *bar codes* para cualquier aplicación: se puede usar independientemente con programas de línea de comandos o con interfaz de usuario.

Zbar se encuentra bajo la licencia *GNU LGPL* 2.1 para permitir el desarrollo de proyectos de código abierto y comerciales.

Algunas de sus características son:

- Es multiplataforma. Sirve para Linux / Unix, Windows,iPhone...
- Alta velocidad para escanear video stream en tiempo real
- No tiene límite de imágenes
- Adecuado para aplicaciones integradas que utilizan procesadores no muy potentes
- Componentes modulares que pueden ser utilizar juntos o separados
- Etc.

Para el desarrollo sobre iPhone tenemos *Zbar SDK* del iPhone, que es un paquete binario que está separado del resto de la biblioteca. Incluye todas las funciones y clases de Objective-C necesarias para poder decodificar un código QR de una imagen capturada con la cámara del iPhone o guardada en la biblioteca de imágenes.

Esta biblioteca fue elegida en un primer momento para realizar la decodificación de los códigos QR debido a la facilidad para integrar el *SDK* en XCode, la buena documentación de la que dispone y la gran variedad de funcionalidad. Posteriormente se desechó su uso ya que no proporcionaba la funcionalidad de decodificar una imagen contenida en una *UIImageView* como lo hace *Zxing*. En apartados posteriores se explicará detalladamente estos inconvenientes del uso de la biblioteca.

[20] http://zbar.sourceforge.net/

5. Aspectos relevantes del desarrollo

En esta sección se tratará de introducir aquellos aspectos más relevantes por su importancia dentro del proyecto y su dificultad en la concepción o en el desarrollo.

Antes de comenzar a explicar los aspectos más relevantes, es necesario especificar que la versión de iOS elegida para el desarrollo de la aplicación es la 5.1 (que en el momento del desarrollo era la última versión). La elección de esta versión se realizó teniendo en cuenta que se quería contar con los últimos avances para la programación iOS.

5.1 Arquitectura Cliente-Servidor

La arquitectura de la aplicación exige la separación del proyecto en tres partes (ver Figura 37):

- La aplicación QrNav, que puede ejecutarse en cualquier dispositivo iOS.
- El programa que controla al Amigobot (ServidorQrNav), es decir, que envía las órdenes al robot usando el *API* de *ARIA* y que está implementado en una máquina Debian.
- Programa *MJPG-Streamer*, servidor que transmite el flujo imágenes a través de *HTTP*.

Este esquema encaja perfectamente en el paradigma de la arquitectura de tipo Cliente-Servidor. El cliente se encarga de interactuar con el usuario a través de la pantalla, procesar los datos y comunicárselos al servidor. Además, también será un cliente que reciba las imágenes de la webcam conectada al Pc Debian. El ServidorQrNav recibirá los datos y, en función de los mismos, manejará al Amigobot. El programa *MJPG-Streamer* nos realiza la función de servidor para transmitir el flujo imágenes obtenidas de la webcam a través de *HTTP*.

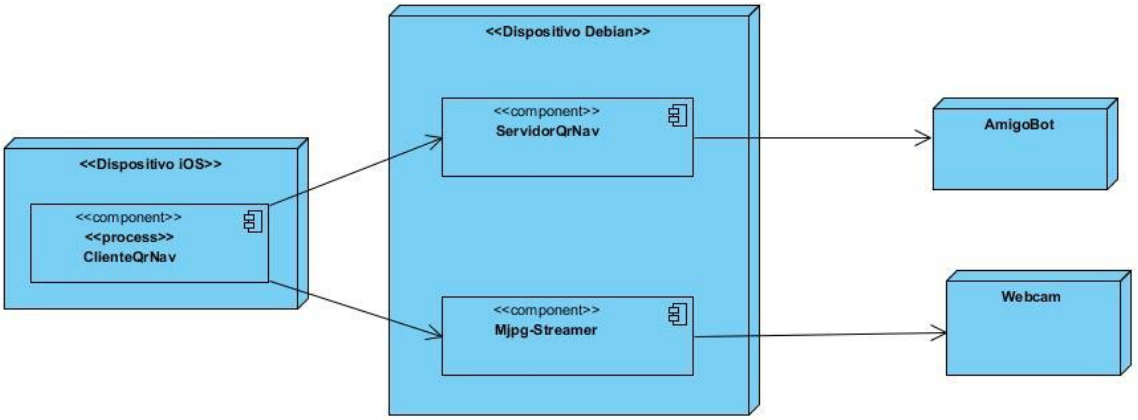

Figura 37 – Diagrama de Despliegue

5.2 Protocolo de comunicación

Este protocolo ha ido creciendo a medida que se ha ido avanzado en el proyecto, debido a la introducción de nuevas partes del proyecto, hecho que nos llevó a declarar nuevas variables.

Nuestro ClienteQrNav está continuamente comunicándose con nuestro ServidorQrNav. Esta comunicación sigue un protocolo.

El clienteQrNav envía 10 parámetros *float* a dicho servidor:

Parámetro	Descripción
Vel	Indica la velocidad del Amigobot
Wel	Indica la velocidad angular del Amigobot
QrFlag	Bandera que indica si se ha leído un QR con dirección de navegación
X	Coordenada X de la posición final de navegación
Y	Coordenada Y de la posición final de navegación
Th	Coordenada Th de la posición final de navegación
Emergency	Bandera que indica si se ha pulsado el botón de emergencia
ColocaFlag	Bandera que indica que si ha realizado el tratamiento de la imagen para la colocación
Pto	Punto medio del cuadrado Rojo
Área	Área del cuadrado Rojo

Tabla 5 - Parámetros protocolo de comunicación

El servidorQrNav sólo le envía un parámetro entero, `acabado`, al clienteQrNav y dependiendo de su valor:

- 0, el Amigobot está realizando el algoritmo de NavegaciónQr o ColocaciónQr.
- 1, indica al cliente que debe realizar el tratamiento de la imagen para la colocación.
- 2, es el valor normal que significa que el robot está en el modo Teleoperación.
- 3, indica al cliente que está delante de un código QR y debe intentar leerlo.
- 4, indica al cliente que el algoritmo de colocación ha fallado.

Estos son los parámetros que, tanto clienteQrNav como servidorQrNav, se transmiten continuamente a través de los *socket* conectados.

A continuación se explicará cómo actúa cada uno en función de los parámetros que recibe.

Comencemos por el clienteQrNav, que en función del valor de `acabado` realiza diversas operaciones. Si le llega `acabado` igual a 2, que es el valor normal, no hace nada. En cambio si le llega el valor:

- **1**, pone todos los parámetros a 0 (menos el de `emergency`) y activa una bandera interna del cliente llamada `colocaAmigo`, para que la clase *MandoViewController* la vea activada y realice el tratamiento de la imagen que tenga en ese momento para obtener el área y el punto medio. Los valores obtenidos los pondrá como parámetros y activará la bandera `ColocaciónFlag` para el siguiente envío.

- **3**, es similiar al anterior, pone todos los parámetros a 0 (menos el de `emergency`) y activa la bandera `LeerQr`, que la clase *MandoViewController* verá activada y realizará la decodificación de la imagen que tenga en ese momento para obtener la información del código QR. En caso de que sea un QR de navegación, los valores obtenidos (`X`, `Y`, `Th`) los pondrá como parámetros y activará la bandera `QrFlag` para el siguiente envío.

- **4**, pone todos los parámetros a 0 (menos el de `emergency`) y muestra una alerta de que no se detecta el cuadrado rojo, por lo que no puede colocar el robot.

Otras operaciones que el clienteQrNav realiza con respecto a este protocolo son:

- Cuando se pulsa el botón de emergencia se ponen a 0 todos los parámetros, menos la bandera de `emergency`, que se activa.

- Cuando movemos el *joystick* establecemos el valor de los parámetros `V` y `W` con dicho movimiento y ponemos todos los parámetros a 0 (menos el de `emergency`). Con esto conseguimos que si alguna de las banderas de `ColocaFlag` o `QrFlag` están activadas las desactivemos para cancelar dicho movimiento.

Seguimos con el ServidorQrNav que es el más complejo, pero antes de ver el algoritmo debemos saber la función de ciertas variables:

Variable	Descripción
Navegando	Indica si el hilo de navegación está ejecutando el algoritmo de NavegaciónQr. Su valor inicial es 0
Colocando	Indica si el hilo de colocación está ejecutando el algoritmo de Colocación. Su valor inicial es 0
Destino	Variable que indica si el Amigobot ha terminado de realizar el algoritmo de NavegaciónQr llegando a la posición indicada. 0 – No ha llegado, 1- Ha llegado. Valor inicial es 0
Fin	Variable que indica si el Amigobot ha terminado de realizar el algoritmo de Colocación llegando a su posición. 0 – No ha llegado, 1- Ha llegado. Valor inicial es 0
Colocaciones	Variable que indica el resultado de la ColocaciónQr del robot

Tabla 6 - Variables protocolo de comunicación

Una vez explicado las variables que intervienen vamos a sumergirnos en el algoritmo creado para el protocolo de comunicación:

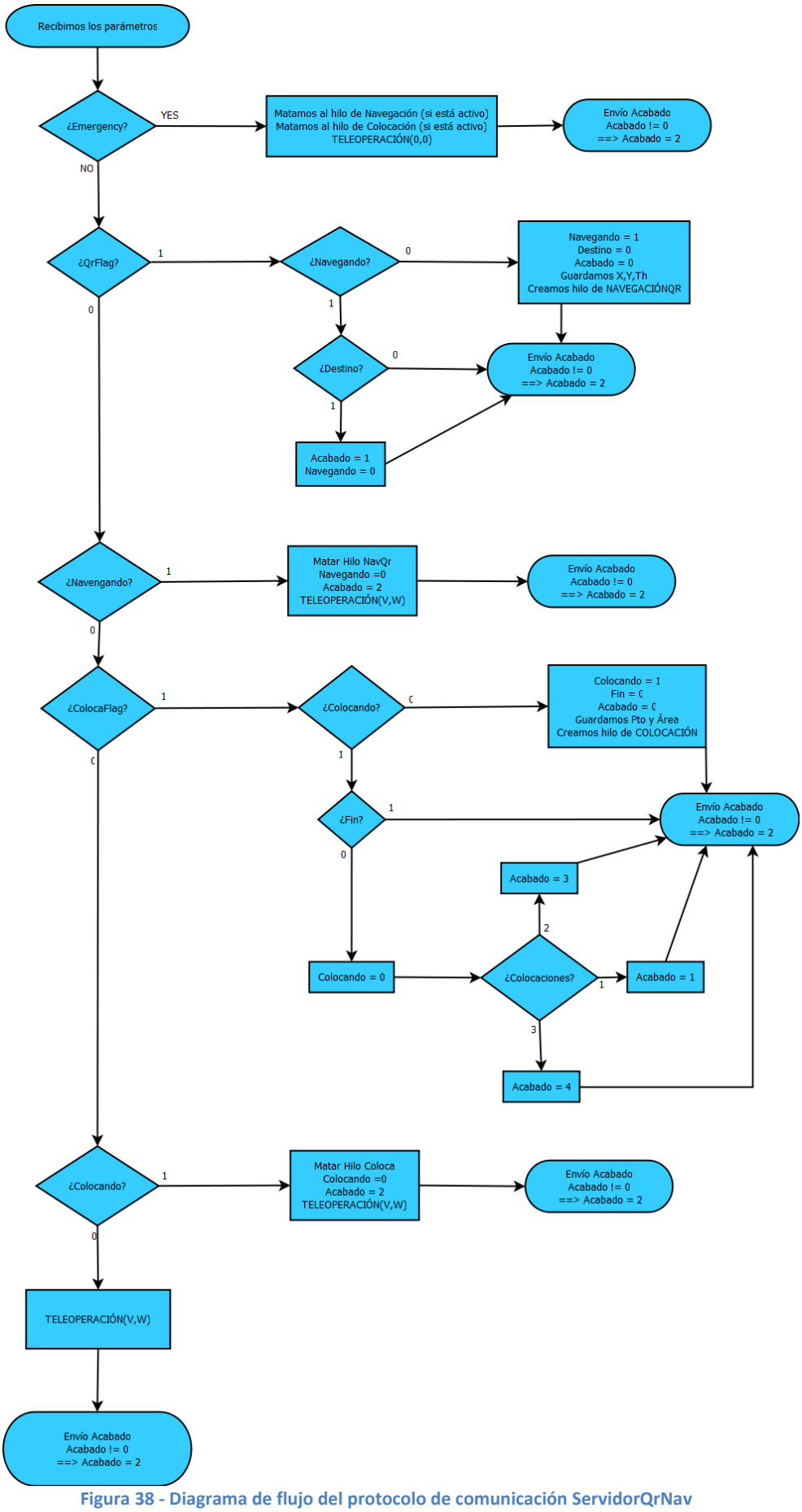

Figura 38 - Diagrama de flujo del protocolo de comunicación ServidorQrNav

En la Figura 38 podemos observar el funcionamiento del algoritmo que nos disponemos a comentar:

El algoritmo comprueba las banderas que le pasa el cliente como parámetro e incluye una funcionalidad, aparte de la NavegaciónQr, Colocación y Teleoperación, y es que si nuestro robot está colocando o navegando en cuanto el usuario mueva el *joystick*, cancelará dicho movimiento y pasará al modo de Teleoperación.

En primer lugar el algoritmo comprueba la Bandera de `Emergency`. Si está activa, mata los hilos creados para la NavegaciónQr y ColocaciónQr (pero sólo si están activos) y establece la `velocidad normal` y `angular` del robot a 0.

En caso de que no esté activa, el algoritmo pasa a comprobar la bandera de `QrFlag` para saber si tiene que comenzar una navegación:

- Si **`QrFlag`** no está activo, mira el valor de la variable `navegando`. Si es 1 quiere decir que está navegando, pero como `QrFlag` no está activo se produce una cancelación de la NavegaciónQr: se mata al hilo de Navegación, se pone `navegando` a 0, `acabado` a 2 y se realiza la teleoperación mandando al robot que se mueva con una velocidad `V` y velocidad angular `W`. Si es 0, pasa a comprobar la bandera **`ColocaFlag`:**

  - Si está inactiva, mira el valor de la variable `colocando`. Si es 1 quiere decir que está colocando, pero como `ColocaFlag` no está activo se produce una cancelación de la Colocación: se mata al hilo de Colocación, se pone `colocando` a 0, `acabado` a 2 y se realiza la teleoperación mandando al robot que se mueva con una velocidad `V` y velocidad angular `W`.

  - Si está activa, vuelve a mirar el valor de la variable `colocando`. Si es 0 quiere decir que no está colocando por lo que la pone a 1, `FIN` a 0, `acabado` a 0, guarda los valores de los parámetros `Area` y `Pto` y crea el hilo que realizará la Colocación. Si es 1, significa que el robot está colocando por lo que comprueba la variable `FIN` para ver si ha concluido, si es así, pone `colocando` a 0 y mira el valor de la variable `colocaciones` que indica el resultado de la ColocaciónQr del robot:

    - Si es 2, significa que no se ha realizado porque el robot esta colocado y pone `Acabado` a 3

    - Si es 1, significa que ha realizado la colocación y pone `Acabado` a 1.

- Si es 3, significa que alguno o ambos valores de colocación recibidos del QrNavCliente son 0, por lo que pone `Acabado` a 4.

❖ Si **`QrFlag`** está activo, vuelve a mirar el valor de la variable `navegando`. Si es 0 quiere decir que no está navegando, por lo que la pone a 1, el `destino` a 0, `acabado` a 0, guarda los valores de los parámetros `X,Y,Th` y crea el hilo que realizará la NavegaciónQr. Si es 1, significa que el robot está navegando por lo que comprueba la variable `Destino` para ver si ha concluido, si es así, pone `acabado` a 1 y `navegando` a 0.

Cuando termina el protocolo de comunicación, envía `acabado` al cliente y comprueba la variable `acabado` para restablecerla a 2 (si vale 1,3 ó 4). Esto es debido a que, cuando acabado tiene esos valores, significa que son peticiones que le realiza al cliente para que lleve a cabo alguna acción, por lo que sólo debe pedirlo una vez.

```
for(;;)
{
   recibidos=recv(socketServidorTCP2,&recibido,10*sizeof(float), 0);
   if(recibidos <= 0 )
  {
       close(socketServidorTCP2);
       exit (0);

  }else{

    if(recibido[6] == 1)
    {
       if(Navegando ==1)
              pthread_cancel(thread_navega);
       if(colocando ==1)
              pthread_cancel(thread_coloca);

       controladorAmigobot_.teleoperacion(0,0);
       exit(2);

    }else{//Si la bandera de emergencia no está activa

       if(recibido[2] == 0)
      {       if(Navegando == 1)
              {
                     pthread_cancel(thread_navega);
                     Navegando=0;
                     Acabado[0]=2;
                     controladorAmigobot_.teleoperacion(recibido[0],recibido[1]);

              }else if(Navegando == 0){
                     if(recibido[7]==0)
                    {       if(colocando==1)
                           {
                                  pthread_cancel(thread_coloca);
                                  colocando=0;
                                  Acabado[0]=2;
                                  controladorAmigobot_.teleoperacion
                                      (recibido[0] , recibido[1]);

                           }else if(colocando==0){
```

```
                                controladorAmigobot_.teleoperacion
                                   (recibido[0],recibido[1]);
                        }
                    }else if(recibido[7] == 1){
                        if(colocando == 0)
                        {
                                colocando = 1;
                                FIN=0;
                                Acabado[0]=0;
                                pto=recibido[8];
                                area=recibido[9];
                                if (pthread_create(&thread_coloca, NULL,
                                                    FuncionHilo2, this))
                                {
                                        cout << "Error al crear el hilo\n";
                                }

                                }else if(colocando==1 && FIN==1){
                                if(colocaciones == 2)
                                        Acabado[0]=3;
                                else if(colocaciones == 1)
                                        Acabado[0]=1;
                                else if(colocaciones == 3)
                                        Acabado[0]=4;

                                colocando=0;
                        }
                    }
                }
        }else if(recibido[2] == 1){
                if(Navegando == 0)
                {
                        Navegando = 1;
                        Destino=0;
                        Acabado[0]=0;
                        x=recibido[3];
                        y=recibido[4];
                        th=recibido[5];
                        if (pthread_create(&thread_navega, NULL, FuncionHilo, this))
                                cout << "Error al crear el hilo\n";

                }else if(Navegando==1 && Destino==1){
                        Acabado[0]=1;
                        Navegando=0;
                }
        }
    }
 }
send(socketServidorTCP2, Acabado ,1*sizeof(int), 0);
 if(Acabado[0] == 1 || Acabado[0]==3 || Acabado[0]==4 )
                        Acabado[0] = 2;
}
```

Tabla 7 - Código protocolo de Comunicación ServidorQrNav

Cuando el ServidorQrNav debe realizar una navegación o colocación, se crea un hilo para que realice tal tarea. En caso contrario el ServidorQrNav no enviaría acabado hasta no terminar con la navegación o colocación, por lo que el ClienteQrNav se quedaría bloqueado sin hacer nada esperando la respuesta del ServidorQrNav.

La comunicación entre el ClienteQrNav y ServidorQrNav debe ser continua y sin demoras.

5.3 Transmisión de Vídeo Streaming en tiempo real

Este aspecto ha sido el más costoso de elaborar y de llevar a la práctica, pues resultaba totalmente novedoso para mí.

A continuación se va a detallar tanto el desarrollo llevado a cabo en este apartado como los problemas encontrados en el mismo, además del estudio de sus posibles soluciones y de los nuevos errores presentes en las soluciones (que van a pasar a convertirse en nuevos problemas).

Se comenzó por intentar obtener las imágenes de la cámara, para ello se investigó y se comprobó que existía la biblioteca *VideoForLinux* que podría ser útil para este cometido. Tras examinarla brevemente, se descubrió la existencia de una versión más avanzada: *VideoForLinux2*.

El trabajo con esta biblioteca nos introdujo aspectos como que para acceder a las webcam se debe usar el archivo /dev/video0, que existen diferentes caminos para obtener las imágenes (la webcam con la que se trabaja en el proyecto sólo permite el mapeo de memoria para ello), que se pueden obtener las características de las webcam, qué driver utilizan (UVC), etc.

La solución a este aspecto fue implementar un driver con esta biblioteca y obtener los datos correspondientes a una imagen procedente de la webcam y almacenarlos en un archivo. Esto supuso la aparición de un nuevo problema, ya que al abrir este archivo se obtenía un error notificando que la imagen estaba dañada.

Mientras se intentaba buscar alguna solución a este problema, se encontró la biblioteca de tratamiento de imágenes *OpenCV*, que permite obtener las imágenes, almacenarlas, cambiar parámetros como la resolución de las imágenes, etc. Dichas imágenes se manejan en una variable llamada *IplImage*, por lo que optó por esta forma de solucionar el problema dejando a un lado la biblioteca anterior y comenzando a trabajar con ésta última.

Una vez solucionado el problema de obtener las imágenes de la webcam, se pasó al siguiente punto que era enviárselas a la aplicación. Pero una vez en este punto, ¿cómo realizar una comunicación entre Cliente y Servidor? Para responder a esta pregunta se pusieron en práctica los conocimientos adquiridos en la asignatura de Redes de ITIS (Ingeniería Técnica en Informática de Sistemas), usando para ello un *socket*. Una vez hallada una posible solución, se comenzó por implementar un *socket*, en el que el servidor obtenía las imágenes mediante *OpenCV* y, enviaba al cliente, el campo *ImageData* de *IplImage* que contenía los datos de la imagen en bruto.

En este punto surgió un nuevo problema ya que *OpenCV* no es totalmente compatible con iOS, algo relativamente normal, pues esta biblioteca tiene funciones

como *CvNamedWindow* (crea una ventana para mostrar imágenes), *CvShowImage* (muestra la imagen en una ventana), *CvSaveImage* (guarda una imagen como archivo), etc. que no son posibles de compatibilizar con iOS.

Investigando una posible solución a este aspecto, se encontró una forma de integrar la biblioteca *OpenCV* en el iPhone que no contenía el archivo de cabecera "*highgui.h*", que es el que agrupa las funciones antes mencionadas además de muchas otras dentro del mismo ámbito.

Tras un duro proceso se consiguió integrar esta biblioteca, recibiendo los datos en bruto que enviaba el servidor a la aplicación. Los datos se almancenaron en una variable *IplImage* y se encontraron en las páginas investigadas dos funciones de traducción para convertir una *IplImage* a *UIImageView* (que es la variable de iOS para contener imágenes) y viceversa.

Una vez realizado este proceso, se obtuvo que el servidor enviaba imágenes continuamente de resolución 160x120 a la aplicación, que las recibía, las almacenaba, las traducía y las mostraba en pantalla. Al hacerlo se pudo comprobar que funcionaba correctamente y que las imágenes captadas por la webcam se visualizaban en el iPhone.

En este punto del desarrollo parecía que nuestro objetivo de la trasmisión de imágenes se había cumplido. Debido a la baja resolución de las imágenes se dispuso a aumentar dicha resolución a 640x480, pero surgió un nuevo problema ya que durante la ejecución de la aplicación no había en ningún momento fluidez en la visualización de las imágenes.

Se intentó encontrar la razón del problema y sus posibles soluciones. Al comentarlo con los tutores se llegó a la conclusión de que el problema podía ser la demora en la obtención de las imágenes de *OpenCV* y la solución propuesta fue intentar obtener las imágenes mediante *VideoForLinux2*, que trabaja más rápido. Por lo que este problema nos llevaba de nuevo al inicio al volver a intentar obtener las imágenes de la webcam. Se regresó al driver construido anteriormente con esta biblioteca que nos daba el error de imagen dañada. Gracias a la colaboración de Carlos Fernández, investigador del Grupo de Robótica, que había trabajado con esta biblioteca, se comprendió que las imágenes procedentes de la webcam eran del tipo MJPG (*Motion JPEG*) y los datos que se obtenían eran las imágenes en bruto sin las cabeceras de Microsoft que debían ser incluidas.

Gracias a esta aportación se logró solucionar el problema obteniendo las imágenes de la cámara mediante el driver que se había implementado usando VideoForLinux2. Se modificó el servidor cambiando la manera de obtener las imágenes de la webcam de *OpenCV* al driver utilizando VFL2.

Aún así, se seguía viendo cómo las imágenes no tenían demasiada fluidez. En este momento, se realizó una tutoría con la profesora de Redes, Ángeles Moreno, que ofreció algunos consejos para intentar mejorarlo, como por ejemplo tocar los buffer tanto de

recepción como de envío del servidor y del cliente y buscar servidores de streaming ya existentes.

A pesar de seguir su primer consejo todo seguía igual. Por ello, tras buscar numerosa información sobre cómo funcionaban los servidores de streaming, se encontró que trabajaban con el protocolo RTP (*Real-time Transport Protocol*, «Protocolo de Transporte de Tiempo real») y alguna breve descripción de funcionamiento pero no se obtuvo mucha más mucha más información, y menos aún, sobre la forma de implementarlos.

Durante varias semanas se siguieron buscando nuevas formas de realizar la transmisión de vídeo streaming en tiempo real, llegando a encontrar diferentes programas que realizan la acción de servidor streaming, como VLC, pero necesitan recepcionar las imágenes con VLC por lo que esta solución no era viable. Hubo otros programas que ejercían de servidor pero necesitaban un archivo de vídeo ya grabado, es decir, se comportaban como servidor streaming de vídeo pero no en tiempo real (ya que el vídeo debía estar completo al comienzo de la transmisión).

Llegados hasta este punto se seguía necesitando un servidor streaming en tiempo real que enviara imágenes en *MJPG*. Tras mucho buscar se encontró un programa que podía resultar útil: *MJPG-Streamer*, que resultó ser nuestra solución al problema, ya que permite obtener, con el *plugin* de entrada *input_uvc.so*, las imágenes de la webcam conectada al pc (mediante *VideoForLinux2*) y con el *plugin* de salida *output_http.so*, que hace de servidor web, transmitir el flujo de imágenes a través de *HTTP* a cualquier cliente que lo solicite. Además este programa permite configurar el servidor con diversos parámetros, como por ejemplo la resolución de las imágenes, el número de imágenes por segundo del vídeo stream, la posibilidad de establecer usuario y contraseña para conectar con el servidor, etc.

En el proyecto este programa establece los *frames* del vídeo streaming a un tamaño de 640 x 480 píxeles y se transmiten a una velocidad de 15fps. En los anexos de esta documentación se explicará detalladamente cómo ejecutar este programa.

Una vez resuelto el problema del servidor en la transmión, únicamente faltaba conseguir un cliente. *MJPG-Streamer* lleva adjunto un programa cliente que recibe imágenes y las muestra pero está escrito en Pascal, por lo que se intentaron aprender las bases de este para entender mejor el programa y saber cómo se recibían las imágenes.

Se trató de construir un cliente *HTTP* en C y recibir una sola imagen, pero no se conseguió recibirla correctamente. Se llegó a la conclusión de intentar implementarlo en C teniendo presente que éste es más sencillo y Objetive-C es un superconjunto de C.

Mientras se trataba de solucionar el problema del cliente, se seguía buscando algún cliente *HTTP* ya implementado. En Github[21], se encontró un proyecto de código abierto

[21] https://github.com/

desarrollado por Hao Hu[22] para XCode que implementa un protocolo que actúa como cliente *HTTP* y recibe un flujo de imágenes[23].

Tras probar este protocolo se comprobó que era capaz de recibir las imágenes correctamente, únicamente se tenía que detallar la dirección de la que iba a recibirlas.

Este protocolo se llama `MJPEGClientDelegate` y para poder usarlo es necesario crear una instancia de la clase *MJPEGClient*, que es un cliente *HTTP* que recibe e interpreta el flujo de imágenes de *MJPG-Streamer*. Tal hecho se produce de forma transparente para el proyecto, pues este protocolo actúa como una caja negra del que sólo es necesario saber qué entradas y salidas tiene.

En el proyecto, la clase *MandoViewController* se convierte en el delegado de dicho protocolo y declara una instancia de *MJPEGClient*.

A continuación se describen las funciones necesarias para tratar con nuestra instancia de `MJPEGClient`:

- **`ClientConnectClicked`**, sirve para iniciar la instancia `client` de *MJPGClient* con la *URL* del servidor del que recibirá las imágenes (Tabla 8).

```
- (void) ClientConnectClicked
{
   Video = YES;

   if (client == nil)
   {
     client = [[MJPEGClient alloc] initWithURL:
                                @"http://10.42.43.1:8080/?action=stream"
                                delegate:self timeout:18.0];
         client.userName = @"";
         client.password = @"";
    }
}
```

Tabla 8 - Código ClientConnectClicked

- **`ClientStartClicked`**, es la función que se llama para indicarle a la instancia que comience a recibir imágenes.

- **`ClientStopClicked`**, sirve para pausar la recepción de imágenes hasta que se vuelva a pulsar el botón de Start.

- **`ClientReleaseClicked`**, sirve para liberar la instancia `client` (para ello es necesario parar la recepción de imágenes).

Este protocolo tiene dos funciones que deben ser implementadas y que serán llamadas al recibir una imagen:

[22] http://www.haohu.de
[23] https://github.com/horsson/mjpeg-iphone

- **`mjpegClient:didReceiveImage:`**, este método será llamado cada vez que se reciba una imagen correctamente. Se llama 15 veces por segundo ya que la transmisión se realiza a 15fps. En esta función es donde se realizan las operaciones necesarias según el parámetro recibido del servidor y se muestra la imagen en pantalla. Dichas operaciones serán detallas en el apartado "Gestión en MandoViewController" de la memoria.

- **`mjpegClient:didReceiveError:`**, es llamado cuando se produce un error al recibir alguna de las imágenes. Solo se ha usado para la depuración mostrando el error por consola.

De esta manera se solucionó el problema de la transmisión de vídeo en tiempo real entre la webcam conectada al Pc Debian y el iPhone.

5.4 Cliente: QrNav

La aplicación cliente se encarga de proporcionar al usuario una interfaz simple e intuitiva y sus funciones son:

- Conectar con el ServidorQrNav
- Controlar la teleoperación del Amigobot
- Gestionar las opciones para el mejor manejo del Amigobot
- Recibir las imágenes de la webcam
- Realizar un tratamiento de las imágenes cuando sea debido para colocar al robot en la posición correcta para leer
- Decodificar el código QR cuando se mande leer
- Mostrar al usuario el proceso que se está realizando
- Realizar un protocolo de comunicación entre Cliente y Servidor

El ClienteQrNav es el núcleo de nuestro proyecto, en otras palabras, es el que decide qué hacer y el que realiza todos los cálculos necesarios para, posteriormente, transmitírselo al servidor.

Nuestra aplicación realizará la función de dos clientes, uno que establecerá la conexión con el ServidorQrNav y realizará continuamente el protocolo de comunicación y, otro que realizará una petición para obtener continuamente el flujo de imágenes.

5.4.1 Patrones de diseño

Durante el desarrollo del diseño de la aplicación han surgido situaciones en las que nos hemos encontramos con un problema que era necesario resolver. Algunos de ellos,

se han resuelto con soluciones específicamente diseñadas y, otros, usando patrones de diseño.

5.4.1.1 MVC

El patrón *MVC* aparece de forma específica en el desarrollo para dispositivos iOS, dando la posibilidad de implementar de forma rápida y natural este patrón en cada una de nuestras aplicaciones.

En nuestra aplicación ha sido empleado en numerosas ocasiones, ya que cada vez que se crea una vista, ésta siempre lleva un controlador asociado a ella.

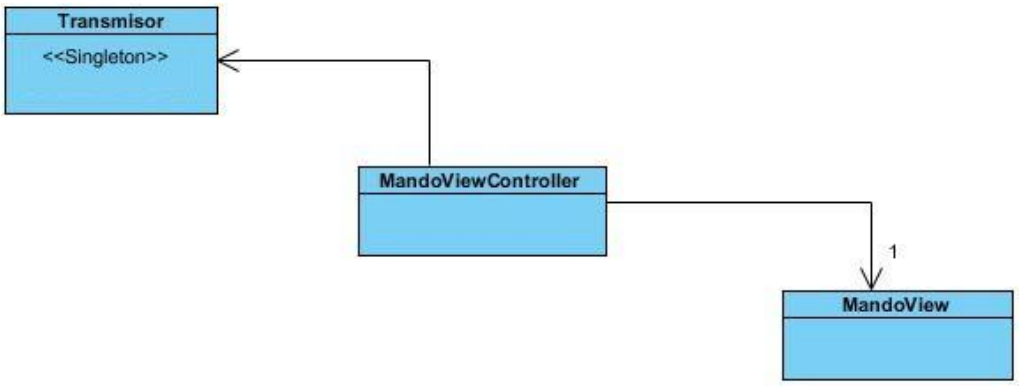

Figura 39 - MVC en QrNav

En esta Figura 39 se observa cómo circularían los datos en esta implementación del *MVC* desde el Modelo (*Transmisor*), hasta el controlador de la vista que le indicará a su vista los nuevos valores con los que modificar su aspecto en pantalla.

5.4.1.2 Singleton

En la aplicación QrNav existen varias clases que sólo deben tener una única instancia para su correcto funcionamiento. Una manera fácil de gestionar este problema es implementar el patrón *Singleton*.

En QrNav la clase *Transmisor* debe tener una única instancia, ya que es la clase que se encarga de la gestión del *socket* con el que se comunica el dispositivo iOS con el servidor. Como esta clase es referenciada por diferentes clases del sistema, todas ellas deben acceder a la misma instancia. La solución a este problema viene dada por el uso de este patrón (Tabla 9).

```
+(Transmisor *) getTransmisor
{
    static  Transmisor *inst = nil;
    @synchronized(self){
        if (!inst) {
            inst = [[self alloc] init];
        }
    }
    return inst;
}
```

Tabla 9 - Código Singleton QrNav

Con este patrón se consigue que nunca existan dos instancias de la clase *Transmisor*, es decir dos transmisores, que enviarían cada uno sus propios datos al servidor mediante un *socket* propio y diferente.

Otra clase que ha usado este patrón es *OpcionesMandos* que lo ha empleado para poder tener todas las opciones que el usuario ha elegido en una misma clase y que pueda ser accedida por las demás clases sin que cada una tenga sus propias opciones. Es por ello por lo que debe existir una única instancia de ella.

Uno de los inconvenientes de este patrón se produce en aplicaciones que usen varios hilos, ya que puede intentar obtener la instancia al mismo tiempo (pero en nuestra aplicación cliente no contamos con este problema).

5.4.1.3 Delegation

En Cocoa este patrón se usa en una gran cantidad de sitios. En nuestra aplicación hay varios patrones *Delegates*, como por ejemplo los *TextField* que tienen un protocolo *Delegate*, que nos permite controlar los eventos para gestionar nuestro *TextField*. Tanto es así que, cuando pulsamos sobre cualquier *TextField* o campo de texto, se genera el método `textFieldDidBeginEditing:` para indicar que se va a editar un campo de texto.

5.4.2 Conexión con el ServidorQrNav

El proyecto comprende dos aplicaciones diferentes que intercambian datos para controlar y hacer navegar un Amigobot usando un dispositivo iOS.

La manera de transmitir los datos necesarios es utilizando un *socket TCP* y, aunque Cocoa Touch y C++ tienen sus *API´*s para la creación de *sockets*, se ha elegido el *API* de *socket* de C puesto que:

- La asignatura de Redes de tercero de ITIS tiene una práctica en C que trata de un *socket* que comunica procesos. De esta manera se pusieron en práctica los conocimiento adquiridos en esta asignatura y, en concreto de esa práctica, llegando incluso a reutilizar partes de la misma.
- Tanto cliente como servidor están desarrollados en diferentes lenguajes: Objetive-C y C++. Ambos son superconjuntos de C y aceptan código escrito en

C dentro de sus códigos. Esto nos facilita el trabajo al poder utilizar una sola *API*, la de C, ya que usar las *API´s* de cada uno de estos *frameworks* supondría usar dos *API´s* diferentes y resultaría una tarea más complicada.

La implementación de los *socket* es una tarea más sencilla, pues el servidor se queda esperando en la `select` la petición de conexión de algún cliente. Cuando recibe una, la acepta y crea un nuevo proceso para que la atienda y dialogue con él. Este proceso será el que se encargue de realizar el protocolo de comunicación y controlar al Amigobot.

En el ClienteQrNav la clase *Transmisor* se encargará de la gestión total del *socket*. Esta clase será la encargada de convertir la *IP* y puerto en la correspondiente estructura y llamar a las funciones de la *API* de s*ockets* de C para establecer la conexión. También es la encargada de enviar y recibir mensajes y de cerrar la conexión.

Bonjour, obtención de la *IP* del servidor

Zeroconf nos ofrece la posibilidad de publicar servicios en nuestro servidor y recuperarlos en el cliente, de manera que el usuario no tenga que configurar su red. Una vez funcionando el servidor nuestro cliente podrá recuperar su servicio y mostrarlo en una *TableView*, de manera que en caso de haber varios servidores, pudiéramos elegir entre todos los disponibles.

Como no se sabe el número de servicios que se obtendrán, crearemos un array y lo llenaremos con los servicios que nos encuentre la aplicación. Para ello usaremos la clase *NSNetServiceBrowser* que será la encargada de buscarnos los servicios y, haciendo uso una vez más del patrón *Delegate*, configuraremos la clase *BonjourTableViewController* como su delegado.

En el momento en que se produzcan cambios en el número de servicios o en sus características, *NSNetServiceBrower* nos hará llegar los cambios mediante el paso del mensaje correspondiente (Tabla 10):

```
- (void)netServiceBrowser:(NSNetServiceBrowser *)netServiceBrowser
didFindService:(NSNetService *)netService moreComing:(BOOL)moreServicesComing
{
        [discoveredWebServices addObject: netService];
        [self.view reloadData];
}
```

Tabla 10 - Código netServiceBrowser:didFindService:moreComing:

Si el usuario selecciona uno de los servicios de la lista (Tabla 11) tendremos que conectar con ese servidor, para ello tenemos que obtener la dirección de dicho servidor pidiéndosela al `Browser`:

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
       // navigate to a web view for that service
       servicioSeleccionado = (NSNetService    *) [discoveredWebServices
                                          objectAtIndex: [indexPath row]];

       [servicioSeleccionado setDelegate:self];
       [servicioSeleccionado resolveWithTimeout: 0];

}
```

Tabla 11 - Código tableView:didSelectRowAtIndexPath:

Cuando el servicio de red realiza correctamente la resolución (obtención de los parámetros del servidor) envía el mensaje `netServiceDidResolveAddres:` a su delegado. En este método se rellenan los parámetros necesarios para conectar con la información del servidor y mandar al transmisor que conecte.

5.4.3 Gestión en MandoViewController

Esta clase es la encargada de administrar todo lo que ocurre en esta vista de los mandos. Entre sus funciones, que explicaremos a continuación, se encuentran:

- Mostrar las imágenes recibidas en pantalla
- Decodificar el código QR de la imagen cuando sea necesario
- Realizar el algoritmo de colocación para la obtención de los parámetros `area` y `Pto`.

Esta clase debe realizar las operaciones establecidas según el parámetro que recibe del servidorQrNav. Esta gestión se realiza en el método `mjpegCliente:didReceiveImage:` del protocolo `MJPEGClientDelegate`, ya que se ejecuta cada vez que se recibe una imagen correctamente.

A continuación se explica el proceso que realiza esta función:

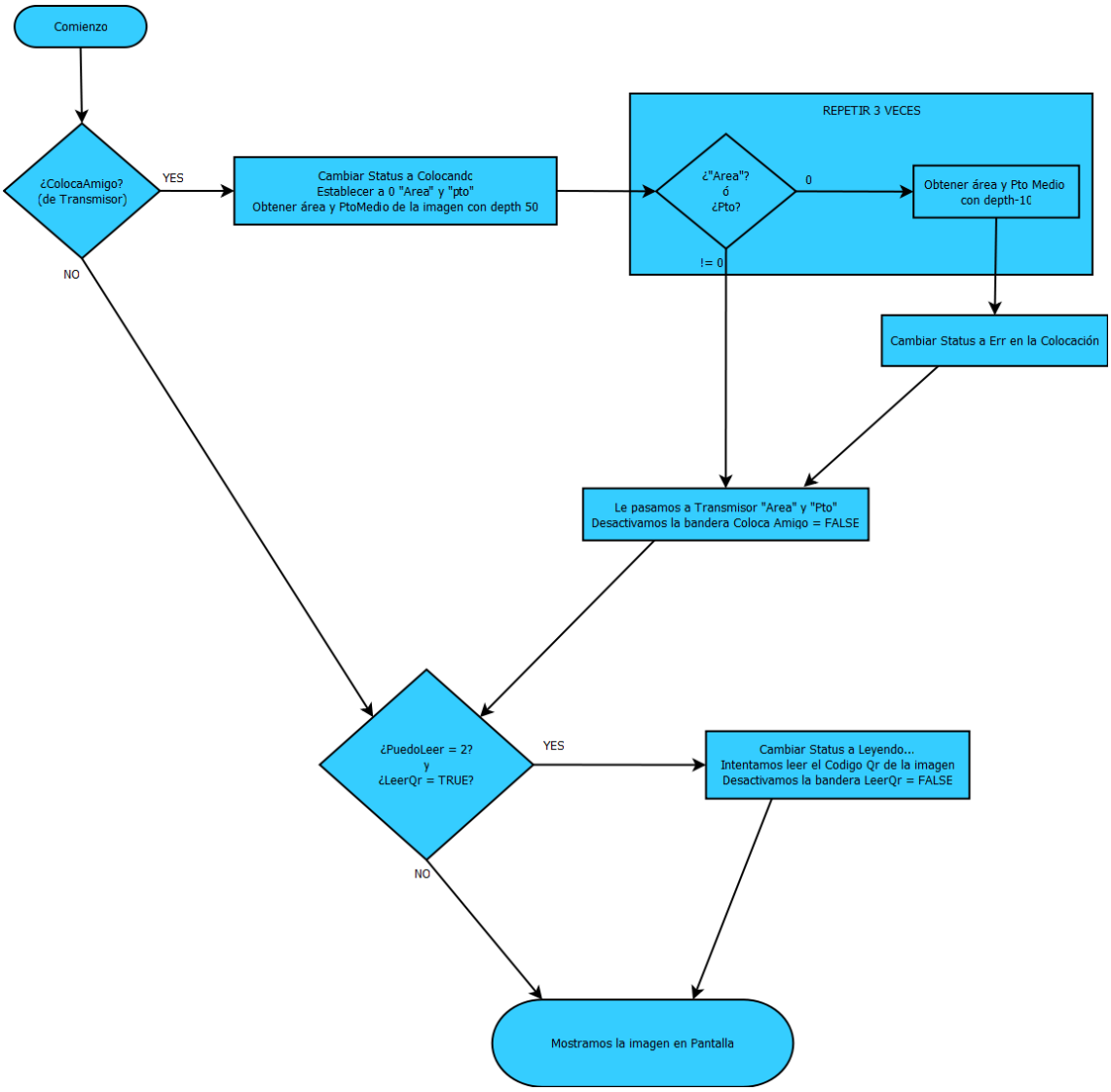

Figura 40 - Diagrama de flujo de control de operación en QrNav

A la hora de comentar el diagrama (Figura 40) comprobaremos, en primer lugar, la bandera `ColocaAmigo` (recordamos que esta bandera se activa cuando acabado es 1 porque significa que el robot ha realizado la navegación y se dispone a colocarlo) como podemos observar en la Tabla 12:

- Si está activa, cambiamos el `status` a "*colocando…*", ponemos las variables `area` y `pto` a 0 e intentamos obtener el área y punto medio con un umbral de valor 50. En caso de no conseguirlo, es decir `pto` o `area` sean 0, intentamos obtener de nuevo el área y punto medio pero con un umbral de valor 30. Si aún así no hay resultados lo intentamos nuevamente con 20 y con 10. Si `pto` o `area` siguen siendo 0, colocamos en el `status` "*Err de colocación…*".

Si alguno de los intentos de obtener el área y el punto resultaran satisfactorios, se saltarían los pasos restantes. Por último, pasamos a *Transmisor* el punto y el área obtenidos y desactivamos la bandera `ColocaAmigo`.

- Si no está activa pasamos al siguiente paso.

```
if( [transmisor colocaAmigo] ==TRUE)
{
    [status setText:@"Colocando..."];
    area=0;
    pto=0;
    [obtener ObtenerArea:image Area:&area PtoMed:&pto Depth:50];

    int depth=30;
    int i;
    for(i=0; i<3; i++)
    {
          if (area == 0 || pto==0)
          {
              sleep(1);
              [obtener ObtenerArea:image Area:&area PtoMed:&pto Depth:depth];
              depth = depth - 10;
          }else{
              break;
          }
    }
   [transmisor AlgoritColocaPto:pto Area:area];
   [transmisor setColocaAmigo:FALSE];

   if(area == 0 || pto==0)
   {
      [status setText:@"Error en la colocación..."];
   }
}
```

Tabla 12 - Código encargado de ordenar realizar cálculos de colocación

A continuación comprobamos que la variable `Acabado` sea 2 y que la bandera `LeerQr` (recordamos que esta bandera se activa cuando acabado es 3 porque significa que el robot ha realizado la navegación y está colocado o que el usuario ha pulsado el botón de leerQr) como se presenta en la Tabla 13:

- Si está activa, cambiamos el `status` a "*Leyendo...*", intentamos leer algún código QR en la imagen y desactivamos la bandera `LeerQr`.
- Si no está activa pasamos al siguiente paso.

```
    if([transmisor QrFlag] == 2 && [transmisor leerQr]== TRUE )
    {
        [status setText:@"Leyendo..."];
        [self SacarQr:image];
        [transmisor setLeerQr:FALSE];
    }
```

Tabla 13 - Código que ordena los cálculos para decodificar la imagen

Por último se muestra la imagen recibida en la pantalla.

5.4.3.1 Leer Código QR

En este apartado se detallará cómo se lleva a cabo la lectura de códigos QR.

Llegados a este punto, comencé a buscar alguna biblioteca y aplicaciones para iPhone con la que poder leer códigos QR.

Tras varias búsquedas se encontró la la biblioteca *Zbar* que es muy potente, completa y tiene un *SDK* para poder decodificar códigos QR en iPhone. A pesar de todas las ventajas que presentaba tenía un gran problema y era que estaba totalmente integrada en el iPhone, es decir, tenía clases para poder acceder a la biblioteca de imágenes del iPhone y seleccionar una de ellas para decodificarla o abrir la cámara para hacer una foto al código que quisiéramos decodificar, pero no nos ofrecía la funcionalidad que se necesitaba que era decodificar una imagen que teníamos almacenada en una variable de tipo *UIImageView*. Es por ello que se desechó esta biblioteca.

Posteriormente, se encontró la biblioteca *Zxing*, que decodifica códigos QR para varias plataformas. Esta biblioteca dispone de varios módulos que se actualizan de forma independiente, entre ellos para el iPhone, pero no se actualiza tan a menudo como la anterior ni posee una documentación tan elaborada. Posee las mismas funcionalidades que *Zbar* pero dispone de un protocolo `DecoderDelegate` para poder decodificar una imagen guardada en una variable *UIImageView*. Los inconvenientes de esta biblioteca son su difícil integración en XCode para poder usarla y que está escrita en C++, pero adaptada para poder hacer llamadas desde Objetive –C. Esto provocó un cambio en la extensión del archivo de código *MandoViewController.m* a *.mm* para que pueda aceptar el código escrito en C++.

Una vez comentado cómo se llegó hasta este punto, me dispongo a explicar su funcionamiento. En primer lugar se llama al método `SacarQr:`, para poder decodificar una imagen que le pasamos como parámetro. En esta función se inicializa el lector de códigos QR, `QrCodeReader`, así como sus propiedades y, después, manda decodificar la imagen.

Nuestro protocolo `DecodeDelegate` llamará a una función si consigue decodificar algún código QR en la imagen o, por el contrario, llamará a otra función distinta si no consigue leer el código QR o no lo hubiera.

En caso de que el código QR se lea con éxito, se llama al método `decoder:didDecodeImage:usingSubset:withResult:` que reproduce un sonido introducido en nuestro proyecto y guarda la cadena de texto obtenida del QR en una variable local. A continuación intentamos parsear la cadena leída del QR para saber si contiene una dirección con un `NSScanner`. Inicializamos el `Scanner` con la cadena y comprobamos si comienza por "*Goto*" (Tabla 14):

- Si comienza, se obtiene el valor de las coordenadas `x`, `y` y `th` de la cadena, se las pasa al transmisor para que se lo envíe al ServidorQrNav y cambia el `status` a "*Navegación Qr…*".

```
/* PARSEO DEL TEXTO CODIGO QR */
scanner = [NSScanner scannerWithString:Qrtext];
if([scanner scanString:@"Goto" intoString:NULL] == TRUE)
{
    float x=0;
    float y=0;
    float th=0;

    [scanner scanString:@"X:" intoString:NULL];
    [scanner scanFloat:&x];
    [scanner scanString:@"Y:" intoString:NULL];
    [scanner scanFloat:&y];
    [scanner scanString:@"O:" intoString:NULL];
    [scanner scanFloat:&th];


    [transmisor activaNavBidiX:x Y:y TH:th];
    [status setText:@"Navegación Qr..."];
}
```

Tabla 14 - Código parseo del QR

- Si no comienza por "*Goto*", es decir, si no se trata de una dirección, únicamente cambia el `Status` a "*Leido Qr…*".

Por último muestra en pantalla la cadena de texto obtenida del QR.

Si no hemos conseguido leer nada, se llama al método `decoder:failedToDecodeImage:usingSubset:reason:` (esta función únicamente muestra una alerta informando del error) y cambiamos el `status` a "*Error de lectura…*".

5.4.3.2 ColocaciónQr

Tras conseguir realizar la navegación se pudo comprobar que la odometría tenía fallos que debíamos disminuir.

En colaboración con los tutores decidimos que los códigos QR estuvieran enmarcados dentro de un cuadrado rojo intenso (R 255, G 0, B 0).

Pero... ¿Qué conseguíamos con esto? Una vez el Amigobot ha llegado a la posición que según su odometría coincide con la dirección a la que debía llegar, entra en juego este proceso que consiste en buscar en la imagen que tengamos en ese momento el cuadrado rojo que recubre al código QR para obtener su área y su punto medio en la imagen.

De esta manera averiguaríamos nuestra posición relativa puesto que si el área es demasiado grande significará que estamos demasiado cerca del código QR y tendríamos que retroceder, mientras que si el área es demasiado pequeña significaría que estamos demasiado lejos y nos tendríamos que acercar.

Además, si el punto medio se encuentra a la derecha de la imagen significa que el QR está a la derecha del robot por lo que giramos 90º a la derecha, avanzamos recto y volvemos a girar 90º a la izquierda. De esta manera conseguiríamos centrar un poco el

código en nuestro campo de visión. Análogamente ocurría si el punto medio estuviera a la izquierda.

Así conseguimos que el Amigobot lea siempre los QR con un tamaño, es decir, a una distancia y que lo lea relativamente centrado.

Debemos establecer unas medidas de referencia para que el cuadrado rojo siempre tenga el mismo tamaño, en caso contrario no podríamos determinar nada con el área y el punto medio. Las medidas que debe tener son las siguientes: 15cm de alto por 15 cm de Ancho y 1.3cm de grosor (Figura 41).

Figura 41 - QR con cuadrado rojo

En la aplicación clienteQrNav se realiza el proceso de obtención del área y del punto medio para que el robot lo coloque. Para ello se optó por la biblioteca *OpenCV* ya que se había trabajado con ella con anterioridad.

En primer lugar, se implementó el algoritmo en un programa aparte (en C++), en Linux, con *OpenCV* 2, para poder realizar diversas pruebas y posteriormente poder integrarlo en el proyecto. Una vez que el algoritmo fue probado, se comprobó que la versión modificada (sin *highgui.h*) compatible con iPhone era la *OpenCV* 1, por lo que hubo que traducir el algoritmo a esta versión.

Al integrar el algoritmo en el proyecto surgió otro problema: *MandoViewController* tiene la extensión *.mm* (debido a la biblioteca *Zxing*) y no es compatible con el archivo de cabecera "*opencv/cv.h*" de *OpenCV*. Por ello, hubo que declarar una clase adicional, *ObtenerController*, para que sea ella quien llame a las funciones de colocar.

Una vez resuelto este problema en la integración de la biblioteca, se implementó el algoritmo ayudándonos de los ejemplos de los que viene acompañada la biblioteca de *OpenCV*.

Cuando queremos obtener el área y el punto medio desde *MandoViewController* se llama a esta función (Tabla 15) de la clase *ObtenerController*:

```
-(void) ObtenerArea: (UIImage *)imagen Area:(int *)area PtoMed:(int *)pto
Depth:(int)depth
{
    ColocaController * coloca = [[ColocaController alloc]init];
    [coloca ObtenerArea:imagen Area:area XMin:&xMin XMax:&xMax Depth:depth];
    *pto = (xMin + xMax) / 2;

}
```

Tabla 15 - Código ObtenerArea:Area:XMin:XMax:Depth: de ObtenerController

Su cometido es pasarle, por referencia, las variables `area` y `pto` y, por valor, el umbral (`depth`) con el que queremos buscar el cuadrado rojo. En la función inicializamos una instancia de la clase *ColocaController* para buscar el cuadrado dentro de la imagen y obtener el área, la x mínima y x máxima del cuadrado y, a continuación, llamamos al método `ObtenerArea:Area:XMin:XMax:Depth:` de ColocaController (Tabla 16):

```
-(void)ObtenerArea:(UIImage *)img Area:(int *)area XMin:(int *)xMin
XMax:(int*)xMax Depth:(int)depth
{
  thresh = 50;
  imagen1= [self CreateIplImageFromUIImage:img];
  if(imagen1 ==NULL)
          NSLog(@"ERROR");
  [self buscarCuadradoRojoImg:imagen1 Area:area Xmin:xMin xMax:xMax Depth:depth];

}
```

Tabla 16 - Código ObtenerArea:Area:XMin:XMax:Depth: de ColocaController

En esta función se establece la variable `Thresh` a un valor de 50 (que será usada en las funciones posteriores) y transformamos la *UIImage* `(img)`, donde está almacenada la imagen en la que queremos buscar el el cuadrado rojo, al tipo *IplImage* (con el que trabaja *OpenCV*) mediante el método `CreateIplImageFromUIImage:` (que encontramos en la página tutorial[24] que nos guiaba sobre cómo intregrar *OpenCV* en XCode).

En la Tabla 17 y Tabla 18 se va a describir el proceso necesario para saber dónde está el cuadrado rojo. En primer lugar se llama a:

[24] http://niw.at/articles/2009/03/14/using-opencv-on-iphone/en

```
-(void) buscarCuadradoRojoImg:(IplImage *)imagen Area:(int *)area Xmin:(int
*)xMin xMax:(int *)xMax Depth:(int)depth
{
   IplImage* canales[3];
   CvMemStorage *storage = 0;
   storage = cvCreateMemStorage(0);

   for (int i=0; i<3; i++){
       canales[i] = cvCreateImage(cvGetSize(imagen), 8, 1);
   }

   cvSplit(imagen, canales[0], canales[1], canales[2], NULL);
   cvSub(canales[2], canales[0], canales[2],NULL);
   cvSub(canales[2], canales[1], canales[2],NULL);
   cvThreshold(canales[2], canales[2], depth, 255, CV_THRESH_BINARY);
   IplImage *canales2EnColor = cvCreateImage(cvGetSize(imagen), 8, 3);
   IplImage *cuadradosImagen = cvCreateImage(cvGetSize(imagen), 8, 3);
   cvCvtColor(canales[2], canales2EnColor, CV_GRAY2BGR);
   ...
```

Tabla 17 - Código buscarCuadradoRojoImg:Area:Xmin:xMax:

Separamos las imágenes en canales, restamos sobre el canal rojo el verde y el azul para no confundir el blanco con el rojo y a continuación umbralizamos el canal rojo, es decir, obtenemos dos niveles (binarios) del canal rojo que significa que todos los píxeles de color rojo de la imagen que estén dentro del intervalo del umbral [depth, 255] ponen a 1 su pixel y lo demás a 0.

A continuación creamos una nueva *IplImage* en color, `canales2EnColor`, a partir del canal rojo y otra llamada `cuadradosImagen`.

```
   ...

   [self drawSquaresImg:canales2EnColor Img2:cuadradosImagen Seq:
                        [self findSquares4:canales2EnColor MemSt:storage]];

   cvCvtColor(cuadradosImagen, canales[2], CV_BGR2GRAY);
   [self calcularAreaYEsquinasImg:canales[2] Area:area Xmin:xMin xMax:xMax];
}
```

Tabla 18 - Código buscarCuadradoRojoImg:Area:Xmin:xMax: (parte final)

Para finalizar esta función se llama a `drawSquaresImg:Img2:Seq:` que dibujará el cuadrado y el relleno de su área en la imagen. Pero antes debe llamar a `findSquares4:MemSt:` para obtener el *CvSeq* (ya que el método anterior lo necesita como parámetro). El último paso que se lleva a cabo es copiar `cuadradosImagen`, que tiene el cuadrado con su relleno, en `canales[2]` cambiando el color a escala de grises y llamar al método `calcularAreaYEsquinasImg:Area:Xmin:xMax:`, para calcular el área, la x mínima y la x máxima del cuadrado hallado en la imagen. Tanto las anteriores funciones como este último método se explicarán con mayor detalle a continuación.

En cuanto a los dos primeros métodos anteriormente mencionados, es necesario destacar que se ha utilizado como referencia un ejemplo adjunto a la biblioteca de *OpenCV* llamado "*squares.c*" que contiene dichos métodos. La función de este archivo es buscar cuadrados de cualquier color en una *IplImage* que contenga la imagen y dibujar los cuadrados en la imagen que se le pase como parámetro. Para conseguir nuestro objetivo se han tenido que modificar estos métodos. Con respecto a ellos podemos destacar:

- **`findSquares4:MemSt:`**, esta función busca cuadrados de cualquier color en una imagen. Recibe como argumentos una *IplImage*, `canales2EnColor`, donde buscará el cuadrado y un *CvMemStorage*, `storage`, que es una estructura que sirve para almacenar de forma dinámica el crecimiento de estructuras de datos tales como secuencias, contornos, gráficos, etc. Como esta función busca cuadrados de cualquier color hemos modificado una línea para que únicamente tenga en cuenta los cuadrados rojos. Para ello, se ha modificado el bucle que busca cuadrados en los 3 canales de la imagen (BGR) uno en cada iteracción, para que sólo busque en el último canal, es decir, en el canal rojo (R).

```
// find squares in every color plane of the image
for( c = 2; c < 3; c++ )
 {
```

Tabla 19 - Modificación de findSquares4:MemSt:

Esta función devuelve un *CvSeq* que contendrá los cuatro puntos del cuadrado hallado, es decir, los vértices.

- **`drawSquaresImg:Img2:Seq:`**, esta función según el ejemplo de *OpenCV* sólo dibuja el cuadrado en la *IplImage* que se le pase como parámetro. Para conseguir que dibuje el cuadrado (encontrado en el método anterior) y rellene su área de color blanco se ha modificado este método añadiendo la siguiente función:

```
cvFillConvexPoly(cuadradosImagen, rect, count, CV_RGB(0,255,0),8,0);
```

Tabla 20 - Función cvFillConvexPoly

  Recibe como parámetros una *IplImage* que tiene almacenada una imagen en la que se encuentra el cuadrado, `canales2EnColor`, y un *CvSeq* devuelto por la función anterior (que contiene los puntos del cuadrado hallado). Además hemos añadido otro parámetro, una *IplImage*, `cuadradosImagen`, para que sea en ella sobre la que se dibuje el cuadrado y se rellene su área.

Una vez llegados en este punto ya tenemos una *IplImage* con una imagen que tiene dibujado el cuadrado con su relleno, ahora sólo tenemos que calcular cuál es su área, su x mínima y su x máxima. Para ello he creado el método (Tabla 21) antes citado:

```
-(void) calcularAreaYEsquinasImg:(IplImage*)imagen Area:(int *)area Xmin:(int
*)xMin xMax:(int *)xMax
{
    int xMinEncontrada = 0;
    int xAnterior = 0;
    int hayBlancoEnColumna = 0;
    int valorPixel = 0;

    for(int x=0; x<imagen->width; x++){
        hayBlancoEnColumna = 0;
        for(int y=0;y<imagen->height; y++){
            valorPixel = ((uchar*)(imagen->imageData+imagen->widthStep*y))[x];
                if (valorPixel != 0){
                        if (xMinEncontrada == 0){
                                *xMin = x;
                                xMinEncontrada = 1;
                        }
                        xAnterior = x;
                        *area = *area + 1;
                        hayBlancoEnColumna = 1;
                }
        }
        if (xMinEncontrada == 1){
                if (hayBlancoEnColumna == 0){
                        *xMax = xAnterior;
                }
        }
    }
}
```

Tabla 21 - Código calcularAreaYEsquinasImg:Area:Xmin:xMax:

Este método busca píxel a píxel cuál es el punto más a la izquierda del cuadrado (xmin) y cuál es el que está más a la derecha (xmax) y calcula su área contando el número de píxeles que tiene el cuadrado. Este método recorre la imagen columna a columna puesto que si el cuadrado está inclinado, por ejemplo hacia la derecha, la xmin no coincide con el vértice superior izquierdo como cuando el cuadrado está recto. Como la imagen está en escala de grises, va comprobando si el píxel es blanco (que significa que forma parte del cuadrado). El primer píxel que encuentre será la `Xmin` del cuadrado y cada vez que localice un píxel del cuadrado suma uno al área. Para hallar la xmax va guardando el valor de x del último píxel encontrado del cuadrado y, cuando recorre una columna entera en la que no encuentre ninguno, guarda el último píxel hallado como `Xmax`.

Así es como conseguimos obtener el área y el punto medio `((xmin+ xmax)/2)` de la imagen que recibimos de la webcam.

En este proceso pueden aparecer errores por causa de la luz que dan lugar a problemas de saturación, como por ejemplo, que la luz se refleje en el código QR y en el cuadrado rojo y el dispositivo no lo detecte.

Para ello, hicimos varias pruebas, por ejemplo, el día 21 de junio de 2012 a las 13:05 en el laboratorio de robótica con ambas luces de sala encendidas y con día soleado, que provocaba un contraluz debido a la ventana. En las pruebas se intentaba en

intentar leer el código QR bajo estas condiciones y desde una posición en la que el Amigobot ya había fallado previamente.

Umbral	Resultados luz apagada	Resultados luz encendida
50	0/10	0/10
30	0/10	3/10
20	0/10	6/10
10	0/10	8/10

Tabla 22 - Pruebas de búsqueda del cuadrado rojo

5.4.4 Movimiento y gestión de los eventos Touch

En la aplicación QrNav, más concretamente en la pestaña de mandos, nos encontramos un *joystick* para poder teledirigir al Amigobot.

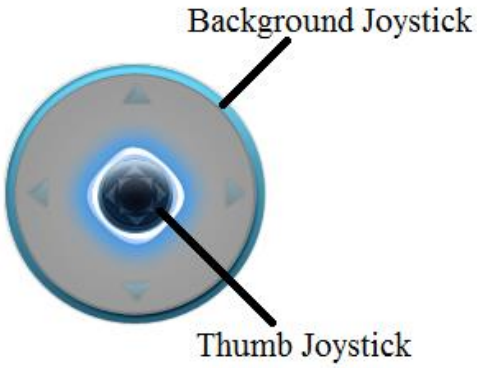

Figura 42 - Joystick usado en QrNav

El funcionamiento del *joystick* es bastante sencillo, sólo hace falta pulsar con un dedo encima del *thumb* del *joystick*, arrastrar hacia la dirección que queramos que el robot se desplace y, una vez queramos dejar de movernos, sólo tenemos que soltar el *thumb* del *joystick* (que volverá a su sitio inicial).

Este hecho genera tres interacciones por parte del usuario:

- Tocar la pantalla.
- Arrastrar el dedo por la pantalla.
- Dejar de tocar la pantalla.

Cada interacción genera su propio evento (Tabla 23) en la vista en la que se produce, que es la encargada de gestionarlos.

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
```

Tabla 23 - Eventos de pulsación

Estas funciones son bastante sencillas ya que sus propios nombres nos indican su función:

- TouchesBegan, se genera cuando el usuario toca la pantalla.
- TouchesMoved, se genera cuando el usuario mueve el dedo por la pantalla.
- TouchesEnded, se genera al levantar el dedo de la pantalla.

Estas funciones tienen un parámetro muy relevante, se trata de un `Set` que es similar a un array de instancias *UITouch*. Cada *UITouch* representa una pulsación, arrastre o finalización de pulsación, dependiendo de cuál de los tres eventos nos haya llegado.

Todos los cálculos que se realizan en estos eventos se generan a partir de las coordenadas en las que se encuentra el *joystick* en la pantalla. Estos cálculos se pueden apreciar en la Tabla 24:

```
/*
  Joystick:

         Tamaño:     50x50
         Posición:
                          x:371 + 25 = 396 centro
                          y:183 + 25 = 208 centro
         Intervalo de pulsación permitida:
                          x:[371 , 421]
                          y:[183 , 233]

         Desplazamiento en:      eje X Y
         Max Desplazamiento:
                          centro +- 46 :  x: [350 , 442]
                          centro +- 46 :  y: [162 , 254]
*/
```

Tabla 24 - Cálculo de coordenadas

En primer lugar, el usuario realizará una pulsación en la pantalla y se generará el evento `touchesBegan`. El paso siguiente será comprobar si dicha pulsación se ha realizado sobre el *thumb* del *joystick* y posteriormente se comprobará si ya hay una pulsación activa sobre éste. Es decir, para poder manejar al Amigobot debemos pulsar encima del *joystick* y luego arrastrar, si mientras tanto se produce otra pulsación en dicho lugar, la aplicación ignorará esa pulsación.

Cuando se pulsa sobre el *thumb* del *joystick* (Tabla 25) se guarda una referencia a ese *UITouch*, se coloca el *thumb* del *joystick* en el lugar de la pulsación y le pasamos a *MandoViewController* el punto de pulsación.

```
for(UITouch *touch in event.allTouches)
{
    location = [touch locationInView:self];
     if (location.x > 371 && location.x < 421 && location.y > 183 &&
                        location.y < 233 && touchJoystick == nil )
     {
        touchJoystick = touch;
       [joystickThumbImageView setPosition: location];
       [mandoViewController MovimientoEjeX: location.x EjeY: location.y];
     }
}
```

Tabla 25 - Código TouchesBegan

En segundo lugar, el usuario arrastrará el *thumb* del *joystick* hacia la dirección a la que quiera mover el Amigobot, por lo que nos llegarán eventos `TouchesMoved`. Como hemos guardado en `TouchBegan` la referencia a nuestro `Touch`, únicamente comprobamos si alguno de los `touchsMoved` que llegan es de ese `Touch` en concreto. Si se trata del mismo `Touch`, obtenemos la localización del arrastre del evento y se comprueba si dicha localización se encuentra dentro de los límites establecidos para el *thumb* del *joystick*. Si se encuentra dentro, simplemente colocamos en dicha localización el *thumb* del *joystick*. En caso contrario, el *thumb* del *joystick* se quedará en el límite (siguiendo el arrastre siempre dentro de dicho límite) como vemos en la Figura 43. Por último, le pasamos el punto de arrastre a *MandoViewController*.

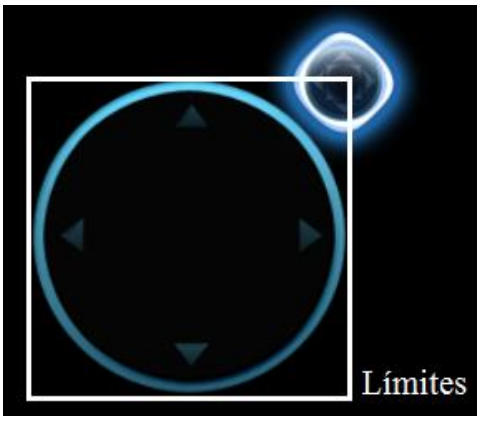

Figura 43 - Pulsación hasta los límites del joystick

Por último, el usuario soltará el *thumb* del *joystick* levantando el dedo de la pantalla y nos llegará `TouchesEnded`. Aquí solamente tenemos que comprobar de nuevo si es el `Touch` que tenemos guardado y ha finalizado. En ese caso, eliminamos nuestra referencia a él para que se pueda volver a usar el *joystick*, colocamos el *thumb* del *joystick* en su posición inicial y le pasamos a *MandoViewController* el punto de pulsación.

A continuación se pretende mostrar cómo gestiona *MandosViewController* las coordenadas que se le pasan desde eventos `Touch` y traducirlas a `velocidad` y `velocidad angular` para que se mueva el robot.

Las pulsaciones sólo pueden ir, en el eje y, desde la coordenada 350 a 442 y, en el eje x, de 162 a 254. Ambas tienen un intervalo de 46 positivos y 46 negativos. Es decir:

EJE X			**EJE Y**		
350	396	442	162	208	254
-46	0	46	46	0	-46
Velocidad de Giro			**Velocidad**		
Máxima a la izquierda	Nula	Máxima a la derecha	Máxima hacia adelante	Nula	Máxima hacia atrás

Tabla 26 - Coordenadas a intervalos

Cuando la vista, *MandosView*, le manda una coordenada (`x, y`) lo primero que hace es cambiar el `status` a "*Teleoperación…*". A continuación, se traducen las coordenadas a valores dentro del intervalo [-46,46] según la Tabla 24. Después se obtiene la `sensibilidad` y la `velocidad máxima` tanto `normal` como `angular` establecidas en opciones. Ahora, simplemente, si la velocidad está dentro del intervalo de sensibilidad [-Sen,+Sen] establece `velocidad` 0 a ambas velocidades. Si está fuera del intervalo, calcula la `velocidad normal` y `angular` en función de la `velocidad máxima` establecida en las opciones como vemos en la Tabla 27:

```
velocidad = ( ( VelMax * valor) / 46 );
velocidadAngular = ( ( WelMax * valorAngular ) / 46 );
```

Tabla 27 - Código cálculo de las velocidades

En caso de que el valor exceda los límites del intervalo [-46,46], se establecerá la `velocidad máxima` hacia adelante o atrás en el caso de la velocidad normal y, hacia la izquierda o la derecha, en el caso de la velocidad angular.

Una vez obtenida una `velocidad normal` y `angular` y, en función de las pulsaciones en las pantallas, se la pasamos al *Transmisor* para que se las envíe al servidor.

5.4.5 Gestión de opciones

La aplicación QrNav presenta un sistema de opciones persistentes, en las que se podrá configurar el control del Amigobot a gusto del usuario y algún aspecto gráfico de la pestaña de mandos.

La clase *OpcionesMandos* está implementada con el patrón *Singleton* para que cualquier otra clase que necesite las opciones pueda obtenerlas. Esta clase es la que tiene los valores de las opciones.

La Clase *OpcionesViewController* es el controlador de la vista de opciones y se encarga de controlar todos los cambios en las opciones y guardarlos en la base de datos y en la clase *OpcionesMandos*. Se guardan en ambos lados para que, mientras dure la ejecución, no tengamos que hacer peticiones continuamente para obtener alguna opción a la base de datos.

5.4.5.1 Gestión del acceso a la capa persistente

Para mejorar el sistema de opciones, en cuanto al manejo del Amigobot y en algún aspecto gráfico, se decidió que el sistema fuera persistente para mejorar la comodidad de la aplicación y para que el usuario en su dispositivo iOS no tuviera que cambiar las opciones cada vez que ejecute la aplicación.

Cuando en una aplicación iOS necesitemos almacenar información que se conserve entre una y otra ejecución, contamos con las siguientes opciones:

- **NSUserDefaults:** podemos almacenar simples preferencias de usuario utilizando tipos de datos como *NSString* o *NSInteger*.
- **Ficheros .plist:** son ficheros de tipo *XML* donde poder almacenar medianas cantidades de información. Esta información podrían ser objetos serializados, una opción muy a tener en cuenta en cuanto a sencillez-posibilidades.
- **SQL Lite:** es un motor de base de datos que ofrece nuestro dispositivo y sobre el que podemos construir modelos completos de datos.
- **Core Data:** es un *framework* que nos facilita Apple y que por debajo realmente utiliza SQL Lite. Contamos con una serie de herramientas que nos facilita tanto la creación del modelo, como su gestión posterior desde código.

Para guardar unas opciones tan pequeñas como son las de la aplicación QrNav podríamos haber utilizado cualquier método sencillo. Sin embargo, se optó por Core Data como método de persistencia pese a que pudiera ser una medida demasiado compleja para nuestro caso. La razón por la que se seleccionó este método fue su aprovechamiento posterior para otras posibles aplicaciones.

Core Data nos ofrece una *API* que recubre *SQL* de forma compleja y que se ocupará por nosotros de los detalles de más bajo nivel. El trabajo con este *framework* nos facilita la labor, debido a que se trabaja a nivel objetual y con instancias de clases diseñadas usando una herramienta de XCode.

5.4.5.1.1 Estructura del acceso a la capa persistente

El *framework* CoreData está basado en la *Data Stack* que será la que nos permita ejecutar todas las acciones. La configuración funcional mínima requiere una serie de objetos ya comentados en apartados anteriores de la presente memoria.

Todos estos objetos hay que inicializarlos antes de poder usar el *framework* correctamente. Para ello se creó la clase *CoreDataManager*, que centraliza toda la gestión de la capa persistente y nos ofrece la posibilidad de iniciar el *Stack*, pero también de insertar y modificar opciones en la capa persistente.

La desventaja es que la clase queda constituida con una funcionalidad mayor y un mayor número de métodos a implementar y, la ventaja, es que todo queda centralizado de manera que cualquier cambio se realizará en la misma clase.

La clase *OpcionesViewController* está relacionada con *CoreDataManager*, de esta forma el acceso a la capa persistente queda totalmente centralizado en estas dos clases, una que se encargará de dar las órdenes necesarias y la otra de cumplirlas.

En cierto modo se hace un uso de un patrón *Delegate*, pues la clase *OpcionesViewController* necesita un acceso a la capa persistente, pero no es su principal función ya que ésta es la de controlar su vista y los cambios que en ella se produzcan. Por tanto lo delega en la clase *CoreDataManager*, usándola para hacer todo el "trabajo sucio" y simplemente dedicándose a dar las órdenes necesarias.

CoreDataManager

CoreDataManager es un Controlador Core Data, que es la clase que manejará todos los aspectos relacionados con el acceso persistente y la encargada de tener los métodos necesarios, tanto para inicializar el *Data Stack*, como para realizar las modificaciones en la base de datos (según lo requieran los otros elementos de la aplicación).

Una vez incializada nuestra Data Stack podemos realizar el acceso a la base de datos, por ejemplo, para crear las opciones de la clase *Opciones* que habremos creado en nuestro Managed Object Model y que XCode nos habrá generado como clase mediante la herramienta comentada con anterioridad y que podemos obsevar en la Figura 18.

Para ello *AppDelegate* al comenzar la aplicación ejecuta el método `crearOpciones`. En primer lugar, realizamos una búsqueda de las opciones de nuestro programa en la base de datos usando *NSFetchRequest* (Tabla 28), que se encargará de cargarnos en un mutable array todas las entidades de tipo *Opciones* y podríamos ir accediendo a ellas mediantes índices (hecho que no ocurrirá en la aplicación, ya que sólo se creará una entidad de opciones). Otra ventaja de *NSFetchRequest* es que realiza un consumo optimizado de memoria cargando los usuarios según sea necesario.

```
NSError *error = nil;
NSFetchRequest *request = [[[NSFetchRequest alloc] init] autorelease];
NSEntityDescription *entity = [NSEntityDescription entityForName:@"Opciones"
                               inManagedObjectContext:self.managedObjectContext];
[request setEntity:entity];
NSMutableArray *mutableFetchResults = [[_managedObjectContext
                                    executeFetchRequest:request error:&error]
                                     mutableCopy];
```

Tabla 28 - Código búsqueda mediante NSFecthRequest

Tras realizar la búsqueda, si el resultado es 0 significa que es la primera ejecución de la aplicación, por lo que crearemos un nuevo objeto de la clase *Opciones* y lo insertaremos en nuestro *Managed Object Context*. No se hace referencia directamente a la clase *Opciones*, sino que se usa una clase genérica *NSManagedObject*. Después insertaremos valores para las claves (*Key*), que son los atributos que definimos en

nuestro *Managed Object Model* y, por último, guardamos el contexto en la base de datos (Tabla 29).

```
if ([mutableFetchResults count] == 0)
{
   NSManagedObject *newManagedObject = [NSEntityDescription
                                        insertNewObjectForEntityForName:@"Opciones"
                                      inManagedObjectContext:_managedObjectContext];

   [newManagedObject setValue:[NSNumber numberWithBool:YES] forKey:@"panelVideo"];
   [newManagedObject setValue:[NSNumber numberWithInt:5] forKey:@"sensibilidad"];
   [newManagedObject setValue:[NSNumber numberWithInt:300] forKey:@"velMax"];
   [newManagedObject setValue:[NSNumber numberWithInt:40] forKey:@"welMax"];

   [self saveContext];
}
```

Tabla 29 - Código creación de las opciones

Para obtener las opciones de nuestro programa se llama al método `ObtenerOpciones`, que es similar al anterior. Lo primero sería realizar de nuevo la búsqueda mediante *NSFetchRequest*. Una vez que ya tenemos el array con nuestras opciones dentro, creamos un puntero de la clase *Opciones* (`opt`) y hacemos que apunte al elemento 0 del array (nuestras opciones). De esta manera, ya podemos acceder a nuestras opciones. A continuación, le pasaremos dichas opciones a la clase *OpcionesMandos* para que las demás clases puedan acceder a ellas sin tener que realizar una consulta a la base de datos como vemos en la Tabla 30.

```
Opciones * opt = [mutableFetchResults objectAtIndex: 0];
OpcionesMandos *optMandos = [OpcionesMandos getOpciones];

optMandos.sensibilidad =[opt.sensibilidad integerValue];
optMandos.VelMax =[opt.velMax integerValue];
optMandos.VelAngular = [opt.welMax integerValue];
optMandos.PanelVideo =[opt.panelVideo boolValue];
```

Tabla 30 - Código obtener las opciones

El proceso a seguir para modificar cualquier opción es muy similar, basta con llamar al método del parámetro de opciones que queremos modificar, por ejemplo, `modificarOpcionValorDePanelvideo:`. Para ello, realizaremos la búsqueda de nuevo, apuntaremos con `opt` al elemento 0 del array, cambiaremos el valor de la opción (clave) que queramos modificar por el parámetro recibido en la función mediante `setValue:forKey:` y guardaremos el contexto.

5.4.5.2 Gestión de UITextField y ScrollView

Uno de los principales problemas que surgen a la hora de trabajar con el teclado en iOS es que su aparición oculte el campo que estábamos pretendiendo editar. Ésto es algo que sucede habitualmente con los campos de tipo *UITextField*.

Figura 44 - Problema con los UITextField y ScrollView

El tamaño de la ventana vendrá dado por las propias dimensiones del *UIScrollView*, es decir, su propiedad frame. Por su parte, las dimensiones que tendrán los contenidos se definen mediante la propiedad `contentSize` que admite un valor de tipo *CGSize*.

Una de las propiedades de *UIScrollView*, que nos va a resultar especialmente útil para solventar el problema que aquí se nos presenta, es la propiedad `contentInset`. Esta propiedad establece un espacio vacío entre los bordes físicos del *scroll view* y el contenido propiamente dicho. Suele resultar de gran utilidad cuando el *scroll view* extiende sus contenidos por debajo de algún otro control (barra de navegación, barra de herramientas, el propio teclado…) y nos interesa que dichos contenidos no se solapen por debajo de dicho control.

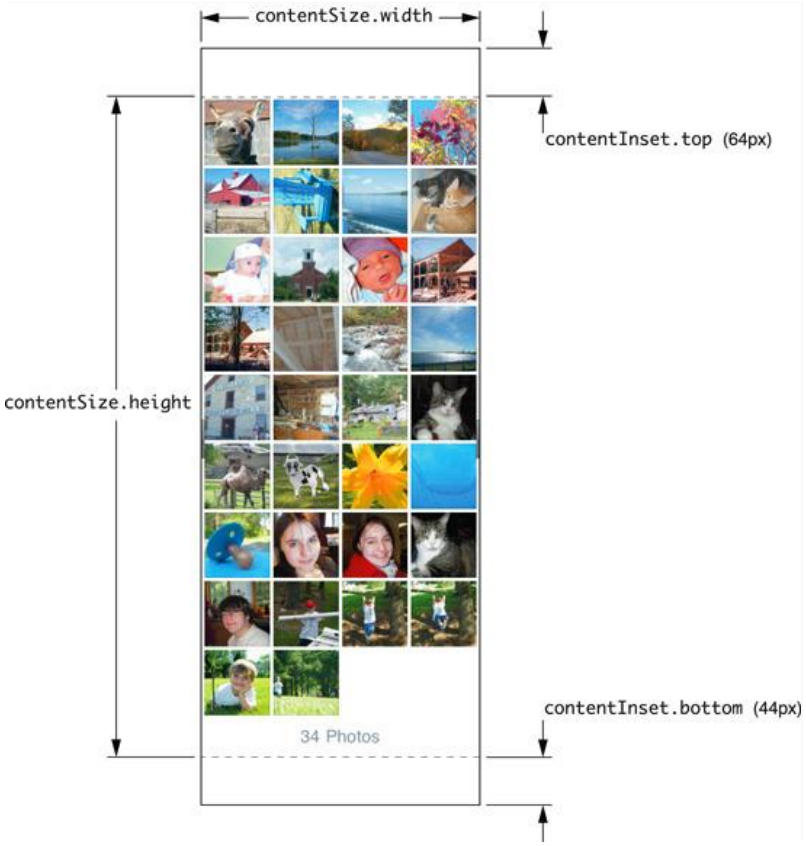

Figura 45 - Propiedades de un ScrollView

Habitualmente, la *view* principal de un *ViewController* suele ser un objeto de la clase *UIView*. Sin embargo, cuando vayamos a trabajar con un *UIScrollView* será el propio objeto *UIScrollView* el que se asigne como la *view* principal del *ViewController* y los elementos que componen la interfaz se agreguen dentro del propio *scroll view*. Al fin y al cabo, un *UIScrollView* no deja de ser una subclase de *UIView*.

Para solucionarlo, lo primero que hacemos es suscribirnos al Centro de Notificaciones (Tabla 31) del sistema mediante la clase *NSNotificationCenter* en el método `viewDidLoad` de *OpcionesViewController*, que nos avisará cuando el teclado se muestre en pantalla y cuando desaparezca llamando a las funciones `apareceElTeclado:` y `desaparece ElTeclado:`.

```
[[NSNotificationCenter defaultCenter] addObserver:self
                                 selector:@selector(apareceElTeclado:)
                                  name:UIKeyboardWillShowNotification
                                        object:nil];

 [[NSNotificationCenter defaultCenter] addObserver:self
                                  selector:@selector(desapareceElTeclado:)
                                   name:UIKeyboardWillHideNotification
                                         object:nil];
```

Tabla 31 - Código subcripción al centro de notificaciones

Posteriormente, nos aprovecharemos de la propiedad *delegate* que ponen a nuestra disposición los objetos de la clase *UITextField* para hacer uso de los métodos del

protocolo `UITextFieldDelegate`: `textFieldDidBeginEditing:` y `textFieldDidEndEditing:`.

Cuando aparezca el teclado en pantalla, se ejecutará el método `apareceElTeclado:` que nos permite acceder al objeto *NSDictionary* que se adjunta a todo método que actúe como encargado de responder a notificaciones. La notificación *UIKeyboardWillShowNotification* envía un *NSDictionary* entre cuyos objetos hay uno cuya clave viene definida en la constante del sistema *UIKeyboardFrameBeginUserInfoKey* y que contiene un objeto *NSValue*, del tipo *CGRect*, con las dimensiones del teclado dentro de su valor `size`.

Una vez que sabemos las dimensiones del teclado, añadimos un `inset` o relleno inferior del mismo tamaño que la altura de nuestro teclado a nuestro *scroll view*. Para ello haremos uso de una estructura de tipo *UIEdgeInsets* en la cual debemos indicar el `inset` (*padding* o relleno) para la parte superior, izquierda, inferior y derecha en ese mismo orden. En nuestro caso, el único lado al que debemos agregarle relleno es a la parte inferior del *scroll view*, ya que el teclado en iOS aparece siempre por la parte de inferior de la pantalla. Al agregar el `inset` conseguimos que nuestro *scroll view* haga su función de *scrolling* pero, además, como el `inset` añadido es justo de la misma altura que la del teclado de iOS, nunca llegaremos a ver ese vacío (ya que queda justo debajo).

Al agregar un relleno a nuestro *scroll view* nos encontraremos con un problema, y es que los indicadores de *scroll* seguirán comportándose de la misma forma que cuando no habíamos añadido el inset. Para evitar este problema asignamos el mismo `inset` al indicador de *scroll* mediante la propiedad `scrollIndicatorInset`, haciendo uso de la misma estructura *UIEdgeInsets* (anteriormente asignada a la propiedad `contentInset`).

Pata terminar, hacemos uso del método `scrollRectToVisible:animated:` (de la clase *UIScrollView*) que, como su nombre indica, hace que el *scroll view* se desplace hasta un determinado rectángulo (en nuestro caso, el rectángulo que define el *UITextField* activo), es decir, su propiedad *frame*. Si el control no estuviese oculto por el teclado, es decir, fuese visible en la ventana del *scroll view*, el método `scrollRectToVisible:animated:` no surtiría ningún efecto.

```
- (void) apareceElTeclado:(NSNotification *)laNotificacion
{
NSDictionary *infoNotificacion = [laNotificacion userInfo];
CGSize tamanioTeclado = [[infoNotificacion
          objectForKey:UIKeyboardFrameBeginUserInfoKey]CGRectValue].size;
UIEdgeInsets edgeInsets = UIEdgeInsetsMake(0, 0,tamanioTeclado.width-25, 0);
[[self scrollView] setContentInset:edgeInsets];
[[self scrollView] setScrollIndicatorInsets:edgeInsets];

[[self scrollView] scrollRectToVisible:[self campoActivo].frame animated:YES];
}
```

Tabla 32 - Código apareceElTeclado:

Para hacer desaparecer el teclado, asociaremos un identificador de gestos a nuestro *scroll view* de forma que cuando el usuario pulse sobre él, el teclado desaparezca. Para ello, en primer lugar, creamos el identificador de gestos (*UITalpGestureRecognizer*) y lo asociamos a nuestro scroll view en el método `viewDidLoad` de nuestro view controller. Al pulsar sobre cualquier parte del *scroll view* se llamará al método `scrollViewPulsado:` que ocultará el teclado llamando a `endEditing`.

Cuando esto ocurra, ¿cómo restauramos el tamaño de la ventana de nuestro *scroll view*? Para ello, se llamará al método `desapareceElTeclado:` que hemos configurado como método de respuesta a la notificación *UIKeyboardDidHideNotification*. En este método, únicamente se devolverá al *scroll view* sus características iniciales, es decir, se ponen los `insets` de nuevo a *CGSizeZero*.

5.5 ServidorQrNav

El servidorQrNav es el encargado de realizar parte del protocolo de comunicación por el que recibirá la información que le transmite el ClienteQrNav y la usará para el control del Amigobot.

5.5.1 Estructura

El servidor se compone de dos clases y de la función *main*:

- La función *main*, es la función con la que comienza el ServidorQrNav. Su tarea es recibir los parámetros con los que se ha ejecutado el programa, crear una instancia de la clase *Receptor* y llamar al método `empezarRecepcion` pasándole los parámetros recibidos en el *main*.
- La clase *Receptor* es la encargada de manejar los aspectos de la conexión y de comunicarse continuamente con el cliente:
  - Publicación del Servicio
  - Esperar la llegada de peticiones de conexión de los clientes
  - Aceptar las peticiones de conexión a los clientes

- Manejar la comunicación creando un proceso hijo que realizará el protocolo de comunicación con la aplicación
- Cerrar la conexión.

❖ La clase *ControladorAmigobot* es la que se comunica con el robot:

- Realiza la teleoperación del robot a partir de los parámetros recibidos en el *Receptor*.
- Realiza la NavegaciónQr mediante el controlador de avance y el controlador de giro, según los parámetros recibidos del *Receptor*.
- Realiza la Colocación del Amigobot en función de los parámetros recibidos del *Receptor*.

5.5.2 Publicación del servicio. Avahi

Para conectar nuestro dispositivo iOS con el ordenador que controlará el Amigobot, se ha optado (como ya he explicado anteriormente) por usar *Zeroconf*, en concreto para el servidor, la implementación Avahi.

El funcionamiento de *Zeroconf* en cuanto a los servicios, es el siguiente:

❖ Un servidor publica un servicio.

❖ Un cliente busca un tipo determinado de servicios.

❖ El cliente elige un servicio y obtiene su *IP* y su Puerto.

Por lo tanto, para hacer que nuestro dispositivo iOS sea capaz de ver el ServidorQrNav, tendremos que publicar un servicio que represente a este servidor.

En Avahi hay varios métodos para publicar un servicio. Uno de ellos consiste en crear en la carpeta: */etc/avahi/services* un archivo del tipo *.service*, este archivo será un archivo tipo *NXML* que contendrá los datos del servicio.

Los archivo tipo *service*, tienen la siguiente forma:

```
<service-group>
        <name replace-wildcards="yes">Amigobot</name>
        <service protocol="any">
                <type>_http._tcp</type>
                <port>2004</port>
        </service>
</service-group>
```

Tabla 33 - Estructura archivos.service

Una vez creado en la carpeta un archivo *.service* con la forma que se presenta en la Tabla 33, tendríamos un servicio del tipo *http/tcp-ip* con el nombre "*Amigobot"* y si,

desde nuestro dispositivo iOS accedemos a la lista de servicios, nos aparecerá y nos permitirá conectar con nuestro ServidorQrNav.

Avahi realizará el resto del trabajo por nosotros, ya que su *daemon* presente en el sistema se encargará de publicar el servicio y de responder a las peticiones que recaigan sobre él.

5.5.3 Conexión con el cliente

La conexión con el ClienteQrNav se realiza usando un *socket TCP* y la *API* de C (de *sockets*).

El servidor realiza todas las funciones previas para crear el *socket*, asociarlo con una dirección (`bind`), etc. Después, nuestro servidor se queda esperando la petición de algún cliente en la función `select`. Cuando algún cliente realiza una petición de conexión, el servidor crea un nuevo *socket* y un proceso hijo que será el que se comunique con el cliente a través del *socket* recién creado.

5.5.4 Conexión con el Amigobot

El ServidorQrNav conecta con el robot Amigobot en la clase *Receptor* antes de comenzar el diálogo con el cliente. Esta clase llama al método `conectar` del *ControladorAmigoBot* para que conecte con él como vemos en la Tabla 34.

```
Aria::init();
ArSimpleConnector connector(&argc, argv);
connector.connectRobot(&robot))
robot.runAsync(true);
robot.enableMotors();
robot.disableSonar();
```

Tabla 34 - Conexión con ARIA y el AmigoBot

Primero iniciamos *ARIA* y para ello usamos el método de clase `init`. Después conectamos con el robot usando un `ARSimpleConnector` (este conector necesita los parámetros que recibe el *main*) y, por último llamamos a los métodos necesarios para usar el robot y desactivamos los sonar, puesto que no se van a utilizar y así consumiremos menos batería. *ARIA* está creada para configurar directamente desde la línea terminal el tipo de robot que queremos manejar y cómo conectar con él, además de otras opciones disponibles. Por ello, esta clase acepta directamente los parámetros de ejecución del programa. Dicha ejecución sería la siguiente para conectar con el Amigobot:

```
./ServidorQrNav -rp /dev/ttyUSB0
```

Tabla 35 - Comando para lanzar el ServidorQrNav conectado al Amigobot

Y para hacerlo con el simulador:

```
./ServidorQrNav
```

Tabla 36 - Comando para lanzar el ServidorQrNav conectado al simulador

Si no hemos realizado ninguna configuración en los puertos USB, el Amigobot se conecta automáticamente a *ttyUSB0*. Si por el contrario el usuario ha realizado algún cambio en la configuración, este puerto puede variar. Estos parámetros indican al conector que debe intentar conectar con el Amigobot en el puerto USB0.

5.5.5 Control o teleoperación del Amigobot

Cuando el protocolo de comunicación nos indica que tenemos que teleoperar con una velocidad `V` y una velocidad angular `W` se llama a:

```
void ControladorAmigobot::teleoperacion(float velocidad, float
                                                velocidadAngular)
{
       if(velocidad >= 0)
       {  robot.setRotVel(-1*velocidadAngular);
          robot.setVel(velocidad);
       }else{
          robot.setRotVel(velocidadAngular);
          robot.setVel(velocidad);
       }
}
```

Tabla 37 - Código ControladorAmigobot::teleoperación

Como se puede observar en la Tabla 37, tenemos dos funciones principales: `setVel`, que nos permite establecer la velocidad a la que queremos que se mueva el Amigobot y `setRotVel`, que le indica al robot la velocidad normal o de giro a la que queremos que se mueva.

La comprobación de si la velocidad es positiva o negativa se realiza para que, cuando el robot se mueva marcha atrás y le indiquemos un giro, éste realizará dicho giro con la parte trasera hacia el lado que le estamos indicando, en lugar de hacerlo con la parte delantera que sería lo normal.

5.5.6 NavegaciónQR

Se trata de una de las partes centrales del proyecto. Consiste en que cuando se recibe la bandera de navegación activada con las coordenadas de la dirección (x, y,`th`), el Amigobot se dirija hacia dicha dirección sin que el usuario tenga que realizar ninguna interacción con el dispositivo iOS. Este movimiento se realiza en forma de L y para ello se utiliza la odometría de las ruedas para el control de las coordenadas.

Para la realización de este algoritmo se llevaron a cabo gran cantidad de pruebas en las que se utilizaron navegaciones diferentes, cambiando de cuadrantes y de coordenadas y desde una posición inicial (0, 0, 0) o en la que previamente se había realizado una teleoperación para desplazar al robot del lugar de comienzo, etc.

Este algoritmo es llevado a cabo por un hilo creado por el `Receptor` para que no se pare la comunicación entre cliente y servidor hasta que termine de realizar la navegación QR. El hilo ejecuta esta función es el que podemos observar en la Tabla 38:

```
 void Receptor::NavQr(void)
{
	pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
	pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
	controladorAmigobot_.navegacionQr(x , y, th);
	 Destino=1;
	sleep(1);
}
```

Tabla 38 - Código Receptor::NavQr

En primer lugar hacemos que el hilo se pueda cancelar en cualquier momento y se llama a la función `NavegaciónQr` de la clase *ControladorAmigobot*. A continuación ponemos `Destino` a 1 indicando que hemos llegado a nuestro destino y esperamos un segundo para que la webcam tenga una imagen enfocada y nítida.

Este hilo ejecuta la función del algoritmo de navegaciónQr (Figura 46) que consiste en:

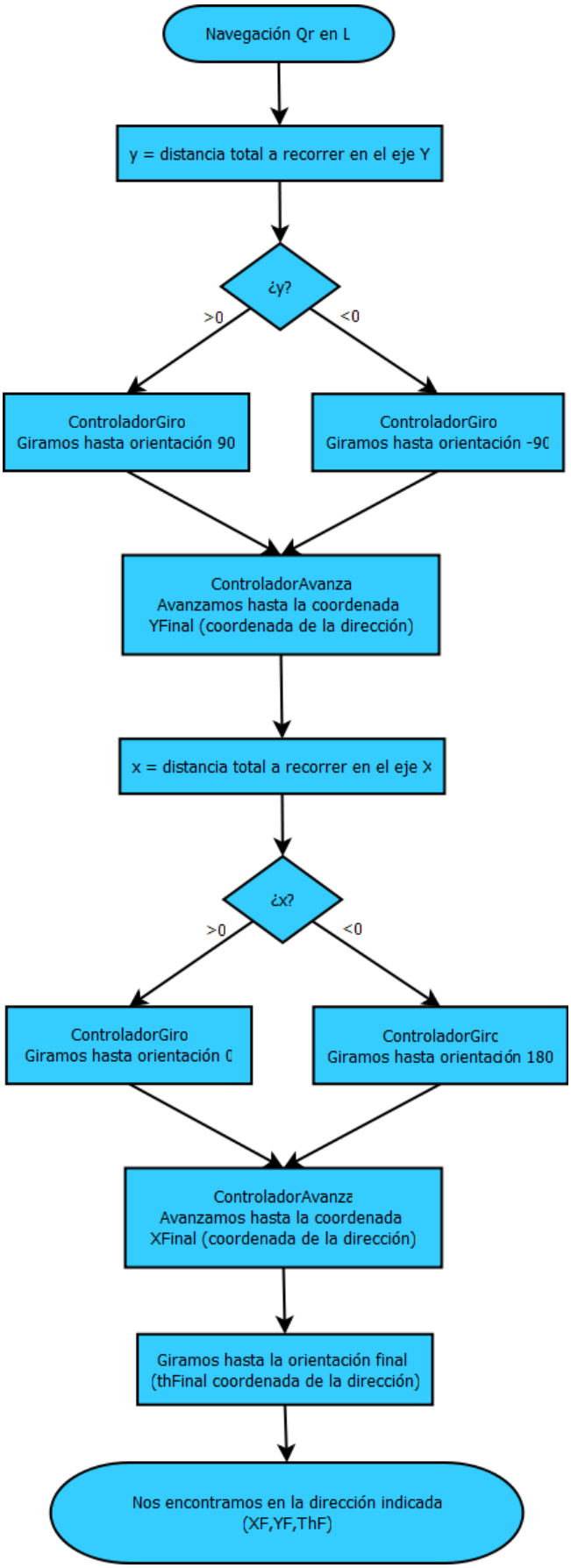

Figura 46 - Diagrama de flujo de la NavegaciónQr ServidorQrNav

A la hora de explicar el funcionamiento vemos cómo el movimiento en L se realiza primero en el eje de las Y del robot (eje horizontal) y posteriormente en el eje de las X (eje vertical).

- 1º Paso. Consiste en saber hacia dónde girar para desplazarnos en el eje Y. Para ello contamos con la información de dos variables `Yc`, que es la coordenada Y de la posición actual del robot, y la `Yf`, que es la coordenada Y a la que debemos llegar. Con estas dos variables podemos saber si debemos desplazarnos hacia la izquierda en el eje Y de la posición en la que nos encontramos, o hacia la derecha en este mismo eje. Debemos restar `Yf-Yc`, si el resultado es positivo debemos ir hacia la izquierda por lo que le mandaremos al *controladorGiro* que gire hasta la posición `Th`= 90. Si por el contrario es negativo sería a la derecha y girar hasta la posición `Th`= -90.

- 2º Paso. Avanzar en el eje de la Y. Simplemente ordenamos al *controladorAvanza* que se dirija hasta la coordenada `Yf` del eje Y.

- 3º Paso. Consiste en saber hacia dónde girar pero ahora en el eje X. Para ello, igual que en el paso uno, restamos `Xf-Xc`. Si es positivo debemos ir hacia la derecha por lo que le indicamos al *controladorGiro* que gire hasta la posición 0 y, si es negativo, a 180.

- 4º Paso. Avanzar en el eje de la X. Repetimos el proceso de ordenar al `controladorAvanza` dirigirse hasta la coordenada `Xf` del eje X.

- 5º Paso. Una vez el robot se encuentra en la coordenada (x, y) de la dirección final, sólo nos queda colocarnos en la orientación final. Para ello mandamos al *controladorGiro* que gire hasta la orientación `Thf`.

Para una mejor comprensión, podemos mirar el código en la Tabla 39.

```
void ControladorAmigobot::navegacionQr(float Xf, float Yf, float Thf)
{
      float Xc,Yc,Thc;
      float x,y,th;

      Yc  = robot.getY();
      Thc = robot.getTh();

     y=Yf-Yc;

     if(y > 0) //Girar a la Izq
     {
        ControladorAmigobot::ControladorGiro(90);
     }
     else if(y<0) //Girar a la Derecha
     {
        ControladorAmigobot::ControladorGiro(-90);
     }

     ControladorAmigobot::ControladorAvanza(Yf,1);

     Xc  = robot.getX();
     x=Xf-Xc;

     if(x > 0) //Girar a la Derecha
     {
        ControladorAmigobot::ControladorGiro(0);
     }
     else if(x < 0) //Girar a la Izq
     {
        ControladorAmigobot::ControladorGiro(180);
     }

     ControladorAmigobot::ControladorAvanza(Xf,0);

     ControladorAmigobot::ControladorGiro(Thf);
}
```

Tabla 39 - Código Controlador::navegaciónQr

5.5.6.1 Controladores de Avance y Giro

En este apartado se desarrollan los controladores de giro y de avance implementados para realizar la navegación.

En principio no contábamos con controladores. Ésto daba lugar a que el Amigobot acelerara bruscamente y, además, añadía otro problema a la odometría: al establecer velocidad normal cero para detener al robot (al haber llegado a la coordenada de navegación), éste se desplazaba unos centímetros más debido a la inercia del movimiento.

Esto nos llevo a implementar un controlador de avance y de giro. Ambos controladores son controladores lineales que constan de 3 fases:

- **Fase de aceleración**, que consiste en acelerar hasta la velocidad máxima desde una velocidad inicial de cero.

- **Fase de velocidad constante**, en esta fase nos desplazamos a una velocidad máxima constantemente.
- **Fase de deceleración**, vamos decelerando desde la velocidad máxima hasta llegar a cero.

Estas fases ocupan un tercio de la distancia total.

Comenzaremos explicando el *controladorAvanza* (Figura 47), que es el que maneja el desplazamiento en línea recta del Amigobot hasta una coordenada de un eje que recibe como parámetro.

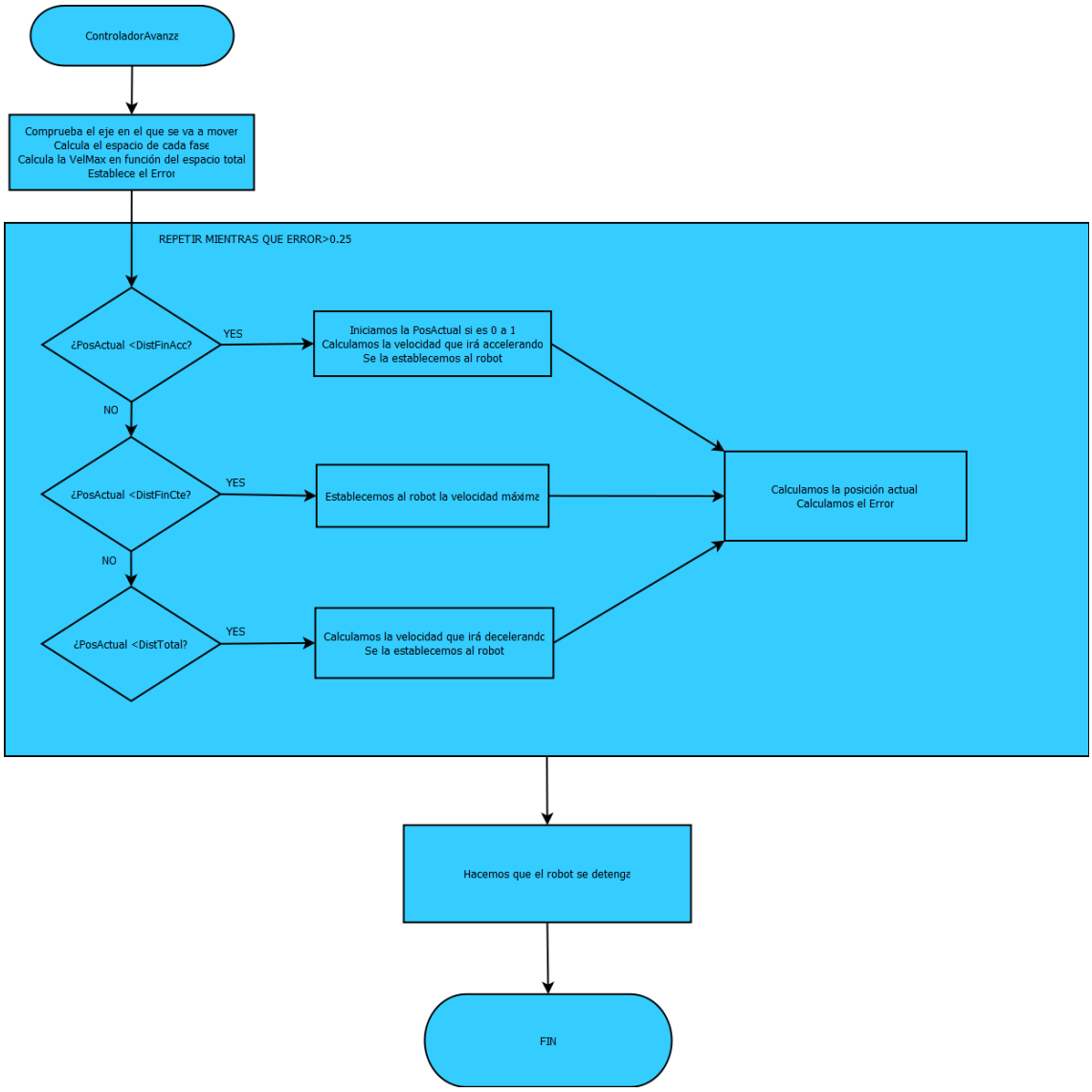

Figura 47 – Diagrama de flujo Controlador de Avance ServidorQrNav

Este algoritmo comprueba primero el parámetro `XY` para saber en qué eje va a moverse. Después, calcula la distancia total que tiene que recorrer restando en valor

absoluto la `XYFinal` a `XYInicial`. Una vez sabemos la distancia total calculamos el punto de la distancia total en el que cada una de las fases concluyen:

- **DistanciaFinAceleración** (`DistFinAcc`), que es el punto en el que finaliza la fase de aceleración. Es 1/3 de la distancia total.
- **DistanciaFinConstante** (`DistFinCte`), punto en el que concluye la fase de velocidad constante. Son 2/3 de la total.
- **DistanciaTotal** (`DistTotal`), es el punto en el que el robot tiene que pararse, es decir, donde acaba la fase de deceleración.

A continuación, establecemos la posición actual (`PosActual`) a 0 y calculamos el error (`Err`) restando la distancia total a la posición actual. Como es el comienzo, el error será igual a la distancia total.

Ahora establecemos la velocidad máxima (`Vmax`) en función de la distancia total (`DistTotal`):

- Si ésta es menor de 200, la velocidad máxima se calcula mediante una proporción, siendo la máxima velocidad con la que podemos avanzar siempre de 200. Por lo que:
  - `Vmax = (100*DistTotal/200)`

  Con esta manera de calcular la velocidad, podría ocurrir que obtuviéramos una velocidad muy pequeña haciendo que el robot tardara bastante tiempo en avanzar el espacio para dicha velocidad. Por ello se introdujo una velocidad máxima "mínima", que será el valor más pequeño de la velocidad máxima que podemos obtener (en este controlador es de 5).
- Si es mayor, establecemos la velocidad máxima a 200.

Bucle

Realizaremos un bucle que seguirá ejecutándose hasta que el error (variable que controla la distancia restante) sea más pequeño que 0.25. En este bucle, en el que podemos ver las tres fases bien diferenciadas, comprobaremos en cuál de ellas nos situamos:

- `PosActual < DistFinAcc`, para saber si estamos en la fase de aceleración.
- `PosActual < DistFinCte`, para saber si estamos en la fase de velocidad constante.
- `PosActual < DistTotal`, para saber si estamos en la fase de deceleración.

Dependiendo de la fase en la que nos encontremos, realizaremos unos cálculos u otros:

- Fase de aceleración. Iniciamos la posición actual (si es 0 la ponemos a 1, de lo contrario siempre estaría en 0) y calculamos la velocidad de aceleración para esa `PostActual` en proporción a la distancia de aceleración :
  - `Vel= (Vmax *PostActual) / DistFinAcc`

Si la velocidad resultante es menor que la `Vmax` entre 2, cambiamos dicha velocidad (`Vel`) a `Vmax` entre 2 para agilizar la aceleración y que no sea haga tan lenta. Por último, establecemos la velocidad al robot.

- Fase de velocidad máxima constante. Establecemos al robot la velocidad máxima.
- Fase de deceleración. Para calcular la velocidad de deceleración primero calculamos cuánto hemos avanzado en la última fase:
  - `a = PosActual – DisFinCte`

Ahora calculamos la distancia de esta fase, dividiendo la distancia total entre 3. Con estos datos podemos calcular la velocidad en proporción a la distancia:

- `Vel= (Vmax *a) / Distancia`

Esta velocidad resultante es la velocidad de aceleración para esa distancia. Para obtener la de deceleración basta con:

- `Vel = Vmax – Vel`

Si la velocidad resultante es menor que 15, la establecemos a 15 para agilizar la deceleración y que no sea haga tan lenta. Por último, establecemos la velocidad al robot.

Para finalizar, se calcula la `PosActual` en cada iteración del bucle, restando la coordenada `x` o `y` actual (dependiendo del eje en que te muevas) a la posición inicial (`PosI`).

Y el error es igual a la distancia total menos la posición actual.

Una vez finalizado el bucle habremos llegado a la `X` o `Y` final, por lo que detendremos al robot estableciendo la velocidad al valor 0 y esperaremos un segundo para que la webcam tenga una imagen enfocada y nítida.

Podemos tomar como aclaración la Tabla 40 que muestra todo lo que se ha explicado en el código.

```
void ControladorAmigobot::ControladorAvanza(float Ptof, int XY)
{
   float PosI, PosF, DistTotal, DistFinAcc, DistFinCte, PosActual, Err, Vel,
         a,f,Vmax;
```

```
   if(XY == 0)
      PosI = robot.getX();
   else if(XY == 1)
      PosI = robot.getY();

   PosF = Ptof;
   DistTotal = abs(PosF -PosI);
   DistFinAcc = DistTotal / 3;
   DistFinCte = (DistTotal * 2) / 3;
   PosActual = 0;
   Err = DistTotal - PosActual;

   /* Velocidad */
   if(DistTotal < 200)
   {
       Vmax= (100*DistTotal) / 200;
       if(Vmax<5)
             Vmax=5;
   }else{
       Vmax=100;
   }

   while( Err > 0.25 )
   {
       if( PosActual < DistFinAcc )
       {
            if(PosActual == 0)
                     PosActual =1;
            Vel = (Vmax * PosActual) / DistFinAcc;
             if(Vel < (Vmax/2))
                     Vel = Vmax/2;
           robot.setVel(Vel);

       }else if ( PosActual < DistFinCte) {

           Vel = Vmax;
           robot.setVel(Vel);

       }else if ( PosActual < DistTotal ) {

            a= PosActual - DistFinCte;
            f= DistTotal / 3;
            Vel = (Vmax * a) / f;
            Vel = Vmax - Vel;
            if (Vel < 15)
                    Vel =15;
            robot.setVel(Vel);
       }

       if(XY == 0)
            PosActual = abs(robot.getX() - PosI);
       else if(XY == 1)
            PosActual = abs(robot.getY() - PosI);
       Err = DistTotal - PosActual;

  }
  robot.setVel(0);
  sleep(1);

}
```

Tabla 40 - Código ControladorAmigobot::ControladorAvanza

A continuación se detalla con una mayor precisión el *controladorGiro* (Figura 48) que resulta más complejo debido a que cambia de positivo a negativo al llegar a 180º y no es, por tanto progresivo, aunque su estructura sea la misma que la del *controladorAvanza*. La función que realiza este controlador es la de girar hasta la orientación que recibe como parámetro y decidir si debe girar hacia la derecha o la izquierda para llegar hasta ella.

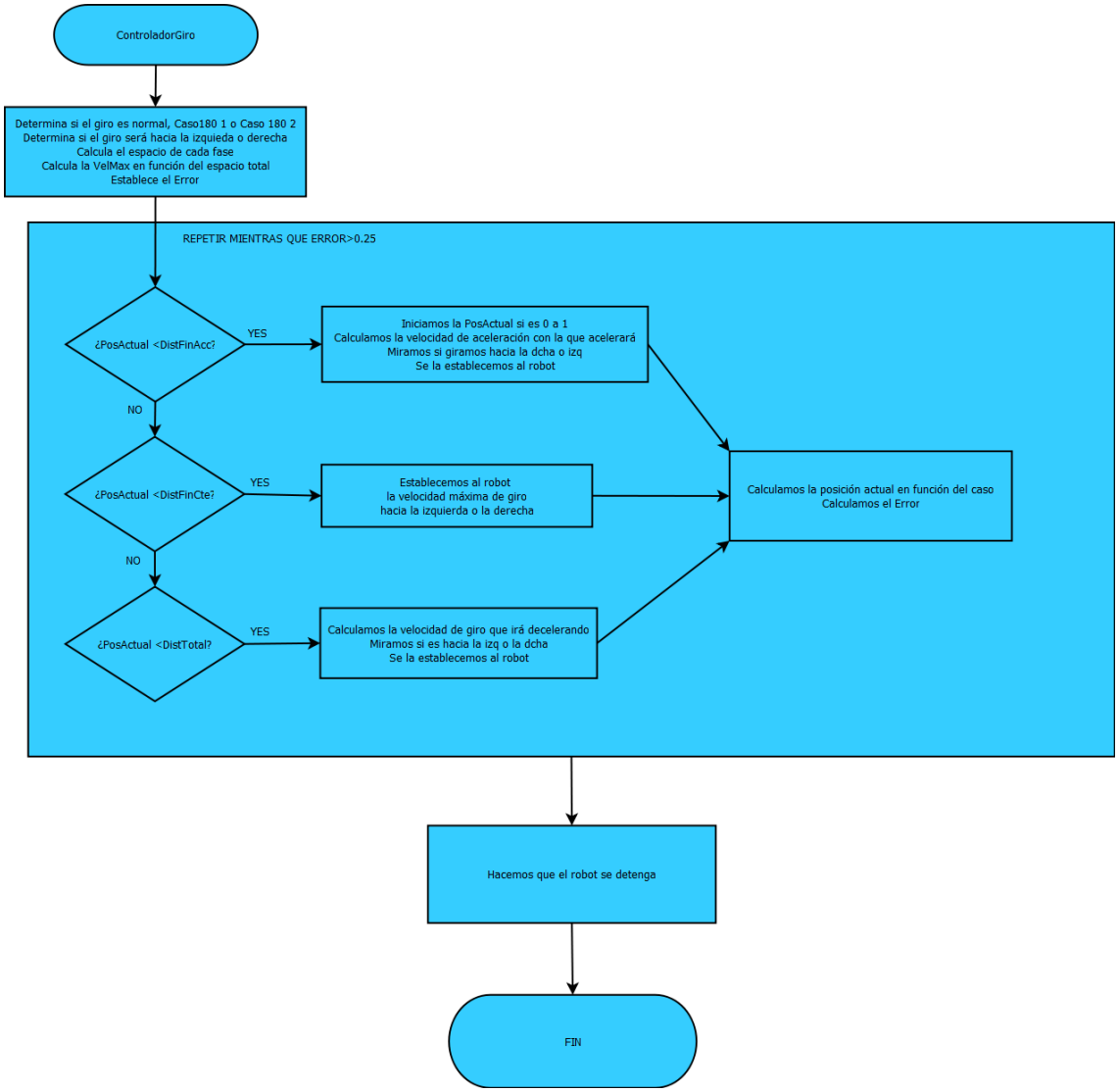

Figura 48 - Diagrama de flujo Controlador Giro ServidorQrNav

Como podemos comprobar el controlador es bastante similar sólo cambian los cálculos a realizar debido al problema que surge cuando el giro pasa por 180º, ya que pasa de positivo a negativo (179 a 180 a -179) o viceversa.

En este controlador se comprueba si el giro entorna dificultad al pasar por 180º o no:

- Si la posición inicial es menor o igual que -89 y la posición final es mayor o igual que 89, estamos en el `Caso 180 1` que va desde el cuadrante negativo [-90,180] al cuadrante [90,180]. Calculamos la distancia total

mediante lo que nos queda para llegar a 180 más la distancia desde 180 a la posición final:

- `DistTotal = (180 – abs(PosI)) + (180 – PosF)`

Y establecemos que el giro será hacia la derecha (control=0)

- Si la posición inicial es mayor o igual que 89 y la posición final es menor o igual que -89, estamos en el Caso 180 2 que va desde el cuadrante [90,180] al cuadrante [-90,180]. Calculamos la distancia total mediante lo que nos queda para llegar a 180 más la distancia desde 180 a la posición final:

  - `DistTotal = (180 – PosI) + (180 – abs(PosF))`

  Y establecemos que el giro será hacia la izquierda (`control`=1)

- `Caso normal`, no hay problemas en el giro por lo que calculamos la distancia total mediante:

  - `DistTotal = abs(PosF – PosI)`

  Y debemos mirar hacia dónde es el giro, dependiendo de si la posI es mayor (hacia la derecha) o menor (hacia la izquierda) que la posF.

Una vez sabemos la distancia total calculamos hasta dónde llega cada fase, establecemos la posición actual y calculamos el error de la misma manera que en el *controladorAvanza*.

A continuación establecemos la velocidad máxima de giro en función de la distancia total, igual que en el *controladorAvanza*:

- Si es menor de 45:

  - `Vmax = (15*DistanciaTotal/45)`

  En este controlador también tenemos un valor mínimo para la velocidad máxima por el mismo problema descrito en el controlador anterior, pero en este caso es de 2.

- Si es mayor, establecemos la velocidad máxima de giro a 15.

Bucle

Al igual que la estructura, el bucle también es igual. Se realiza de la misma manera hasta que el error sea menor que 0.25, al igual que las comprobaciones para saber en qué fase estamos (que terminan siendo exactamente las mismas).

Dependiendo de la fase en la que nos situemos se realizarán una serie de cálculos:

- **Fase de aceleración**. Realiza todos los cálculos ya mencionados en el *controladorAvanza*, salvo que la Velocidad mínima de aceleración de giro

es 5 y que, en función de si el giro es hacia la izquierda o derecha, establece una velocidad positiva o negativa al robot respectivamente.

- **Fase de velocidad máxima de giro constante**. Establecemos al robot la velocidad máxima teniendo en cuenta si es a la izquierda o a la derecha.

- **Fase de deceleración.** En esta fase, los cálculos también son los mismos que el *controladorAvanza* pero cambiando la velocidad mínima de giro a 0.6 y estableciéndola en función del `control`.

Por último, para calcular la `PosActual` en cada iteración del bucle, debemos tener en cuenta que ésta es diferente dependiendo del caso en el que se encuentre:

- `Caso Normal`, basta con restar la coordenada th actual a la orientación inicial.

- `Caso 180 1`, si la orientación `th` actual:

  - Es negativa, significa que aún no hemos pasado por 180º. Calculamos la distancia que nos falta para llegar a 180 más la que haya de 180 hasta la posición final. Esto sería la distancia que nos queda para llegar a la orientación final y, para calcular la distancia recorrida, basta con restársela en valor absoluto a la posición actual:

    - `PosActual=(180-PosF)+(180-abs(robot.getTh()));`

    - `PosActual = abs(DistTotal -PosActual);`

  - Es positiva, significa que hemos pasado por 180º. Por tanto habrá que calcular la distancia para llegar a la orientación final más la distancia desde la posición inicial hasta 180:

    - `PosActual=(180-robot.getTh())+(180-abs(PosI));`

- `Caso 180 2`, es exactamente igual que el caso 1 pero cambiando el lugar donde tenemos que poner el valor absoluto e intercambiando el positivo por el negativo:

  - `Th` positiva:

    - `PosActual=(180-abs(PosF))+(180-robot.getTh()`);

    - `PosActual = abs(DistTotal -PosActual);`

  - `Th` negativa:

- `PosActual=(180-PosI)+(180-abs(robot.getTh()));`

El error es igual a la distancia total menos la posición actual.

Que el bucle finalice significa que hemos llegado a la `Th` final, por lo que detendremos al robot estableciendo la velocidad de giro 0 y esperaremos un segundo para que la webcam consiga una imagen bien enfocada y nítida.

Para una mayor aclaración podemos ver todo lo explicado en el código en la Tabla 41.

```
void ControladorAmigobot::ControladorGiro(float OrientacionF)
{
	float PosI, PosF, DistTotal, DistFinAcc, DistFinCte, PosActual, Err,
		Vel, a,f,Control, Caso180,Vmax;

	Caso180=0;
	PosI = robot.getTh();
	PosF = OrientacionF;

	//Los 89 es como si fueran 90 xk se para siempre en +-89.91
	if(PosI <= -89)
	{	if(PosF >= 89)
		{	DistTotal = (180 - abs(PosI)) + (180 -PosF);
			Caso180=1;
			Control=0;
		}else{
			DistTotal = abs(PosF -PosI);
		}
	}else if(PosI >= 89){
		if(PosF <= -89)
		{	DistTotal = (180 - PosI) + (180 - abs(PosF));
			Caso180=2;
			Control=1;
		}else{
			DistTotal = abs(PosF -PosI);
		}
	}else{
		DistTotal = abs(PosF -PosI);
	}

	DistFinAcc = DistTotal / 3;
	DistFinCte = (DistTotal * 2) / 3;
	PosActual = 0;
	Err = DistTotal - PosActual;

	/* Velocidad */
	if(DistTotal < 45)
	{	Vmax= (15 * DistTotal) / 45;
		if(Vmax<2)
		     Vmax=2;
	}else{
		Vmax=15;
	}

	if(Caso180 == 0)
```

```
	{	if(PosI < PosF)
			Control = 1;
		else
			Control= 0;
	}

	while( Err > 0.25 )
	{
		if( PosActual < DistFinAcc )
		{	if(PosActual == 0)
				PosActual =1;
			Vel = (Vmax * PosActual) / DistFinAcc;
			if(Vel < 5)
				Vel =5;
			if(Control == 0)
			{	Vel = Vel * (-1);
				robot.setRotVel(Vel);
			}
			else if (Control == 1)
				robot.setRotVel(Vel);

		}else if ( PosActual < DistFinCte ) {
			Vel = Vmax;
			if(Control == 0)
			{	Vel = Vel * (-1);
				robot.setRotVel(Vel);
			}
			else if (Control == 1)
				robot.setRotVel(Vel);

		}else if ( PosActual < DistTotal ) {
			a= PosActual - (DistFinCte);
			f= DistTotal / 3;
			Vel = (Vmax * a) / f;
			Vel = Vmax - Vel;
			if(Control == 0)
			{	Vel = Vel * (-1);
				if (Vel > (-0.6))
					Vel =-0.6;
				robot.setRotVel(Vel);
			}
			else if (Control == 1)
			{	if (Vel < 0.6)
					Vel =0.6;
				robot.setRotVel(Vel);
			}
		}

		if(Caso180 == 0)
		{	PosActual = abs(robot.getTh()- PosI);

		}else if(Caso180 == 1){
			if(robot.getTh() < 0)
			{	PosActual = (180-PosF)+(180 - abs(robot.getTh()));
				PosActual = abs(DistTotal -PosActual);
			}else if(robot.getTh() > 0){
				PosActual = (180-robot.getTh() + 180 - abs(PosI));
			}

		}else if(Caso180 == 2){
			if(robot.getTh() > 0)
			{	PosActual = (180-abs(PosF)) + (180-robot.getTh());
```

```
                        PosActual = abs(DistTotal -PosActual);
                    }else if(robot.getTh() < 0){
                        PosActual = (180-PosI) + (180 -abs(robot.getTh()));
                    }
                }
                Err = DistTotal - PosActual;
        }
        robot.setRotVel(0);
        sleep(1);
}
```

Tabla 41 - Código ControladorAmigobot::ControladorGiro

Errores

A continuación se presentan algunos de los errores que pueden producirse durante este proceso. La mayoría de ellos se consideran que tienen un nivel de gravedad elevado:

- El suelo de la universidad y de muchos lugares no es una superficie uniforme, sino que presentan irregularidades. En el caso de la universidad encontramos un suelo embaldosado y muchas veces estas baldosas no están al mismo nivel sino que tienen superficies entrantes o salientes en el terreno. Esto provoca fallos en la odometría ya que, cuando se encuentra un saliente, el Amigobot contará más espacio del que existe en una superficie uniforme y, si es un entrante, contará menos espacio. Otro fallo que se puede producir es que al pasar el Amigobot un saliente caiga bruscamente de una baldosa a otra.

  Esto provoca que la dirección a la que llega el amigobot no se corresponda con la posición real a la que le hemos mandado, a la vez que hace que el movimiento no sea una L exacta.

  Estos fallos son muy graves para la navegación, pero aún más para los giros ya que si gira mal, por ejemplo, en lugar de girar hasta 90º gire hasta 93º en la realidad, provoca que la trayectoria del Amigobot se vaya curvando levemente hacia abajo.

- Los suelos pueden provocar otro tipo de fallo y es que, en superficies lisas, las ruedas pueden llegar a patinar haciendo que el espacio patinado no cuente en la odometría.

- Cuando movemos al Amigobot en línea recta estableciendo únicamente una velocidad lineal positiva, podemos comprobar cómo el Amigobot se va desviando hacia la izquierda. Esto se produce debido a las limitaciones del Amigobot, que no es capaz de establecer la misma potencia en cada rueda al mismo tiempo. Este error se podría solventar en un futuro usando un robot más técnico y preparado.

5.5.7 Colocación del Amigobot

Para minimizar muchos de los errores anteriormente comentados, se introdujo esta nueva parte del proyecto.

En este apartado se explica cómo el Amigobot se coloca en una posición adecuada para leer el código QR, en función del área y del punto medio del cuadrado rojo detectado por el ClienteQrNav.

Para establecer una posición común para leer todos los códigos QR se realizó un estudio tomando el cuadrado rojo con las medidas establecidas (mencionadas en esta memoria). En primer lugar se llevaron a cabo unas pruebas para saber cuál era la distancia máxima a la que la webcam leía un código QR.

Distancia	
20 cm	Lee el código QR. Pero ocupa casi todo el frame
25 cm	Lee el código QR. Sigue siendo muy grande
30 cm	Lee el código QR. El código es algo más pequeño
35 cm	Lee el código QR.
40 cm	Lee el código QR.
45 cm	Lee el código QR.
50 cm	Lee el código QR.
55 cm	Lee el código QR.
60 cm	Lee el código QR.
65 cm	Lee el código QR.
70 cm	Lee el código QR.
75 cm	Comienza a fallar la lectura. Solo lee a veces (4/5).
80 cm	Fallo de lectura (2/5)
85 cm	Falla bastante la lectura (1/10)
90 cm	No consigue leer nada
100 cm	No consigue leer nada

Tabla 42 - Pruebas para la distancia a la que leer un códigoQR

Una vez concluidas las pruebas, se decidió que el Amigobot debía leer los códigos QR a una distancia de entre 40 y 50 centímetros. Para saber cuál era el área del cuadrado rojo a estas distancias, se empleo el clienteQrNav, poniendo la webcam a la distancia indicada. El resultado fue que el AmigoBot debía leer el código QR cuando estuviera a una distancia en la que el área de dicho código estuviera dentro del intervalo [40.000 - 25.000], es decir, entre 50 y 40 centímetros.

Para calcular el intervalo en el que debía encontrarse el punto medio se fueron probando varias ejecuciones. Sabiendo que la imagen en la que se encuentra el código QR cuenta con 640 píxeles de ancho, se dispuso que el punto medio del cuadrado debía estar entre [400 – 240], siendo el medio exacto 320.

Este algoritmo es llevado a cabo por un hilo creado en la clase *Receptor*, para que no se pare la comunicación entre cliente y servidor hasta que termine de colocarlo. Este hilo ejecuta la siguiente función (Tabla 43):

```
void Receptor::AlgoritColoca(void)
{
	pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
	pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
	colocaciones = controladorAmigobot_.colocacionQr(area,pto);
	FIN=1;
	sleep(1);
}
```

Tabla 43 - Código Receptor::AlgoritColoca

En primer lugar hacemos que el hilo se pueda cancelar en cualquier momento y se llama a la función `ColocaciónQr`, de la clase *controladorAmigobot*, que nos devolverá un *int* que guardaremos en `colocaciones`. A continuación ponemos `FIN` a 1 indicando que hemos acabado de colocarlo y esperamos un segundo para que webcam consiga una imagen enfocada y nítida

Este hilo ejecuta la función del algoritmo de ColocaciónQr que consta de (Figura 49):

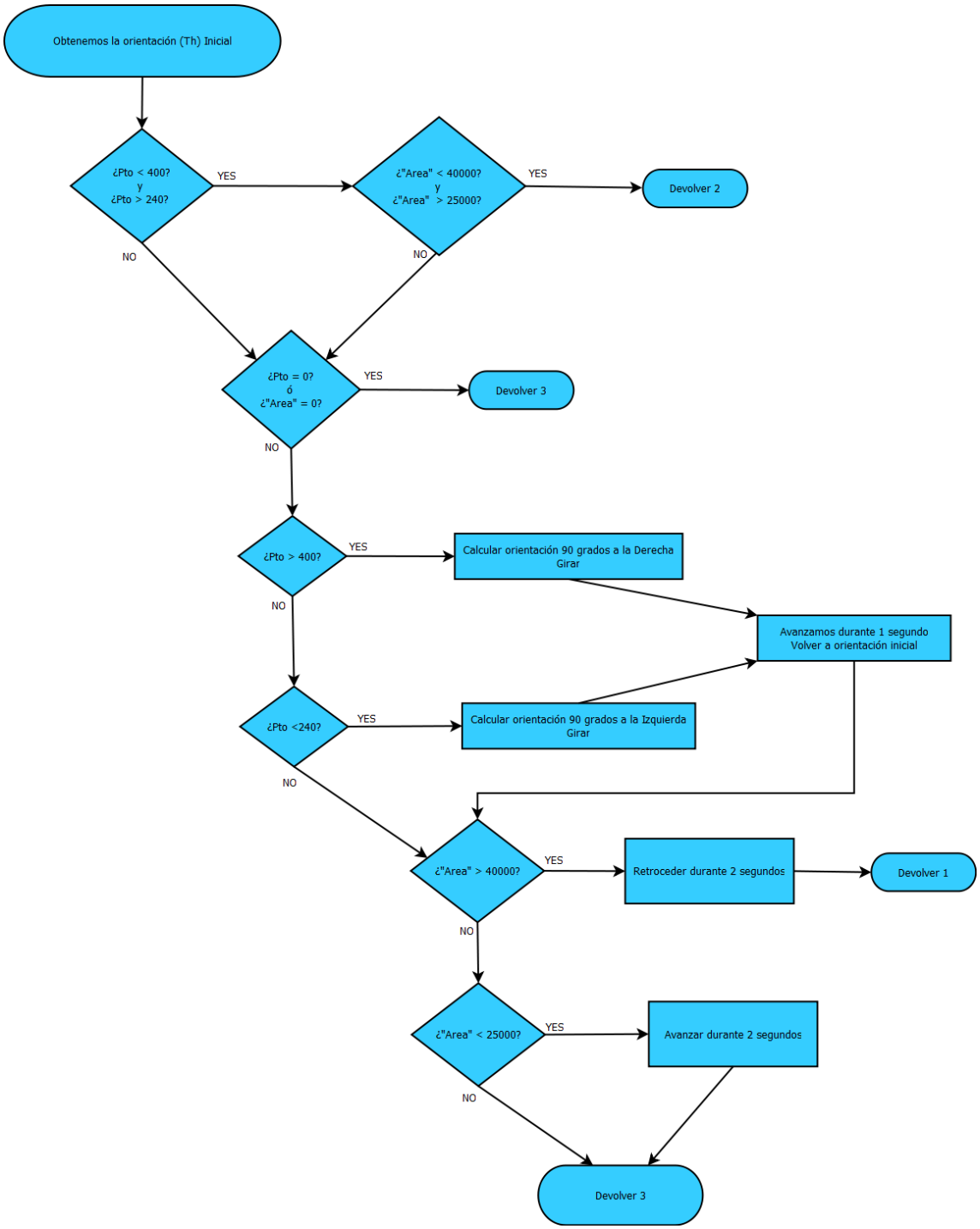

Figura 49 - Diagrama de flujo ColocaciónQr ServidorQrNav

En primer lugar comprueba si, tanto `pto` como `area`, están dentro de los intervalos permitidos y si es así devuelve el valor 2. En cambio, si cualquiera de los dos parámetros es 0 devuelve 3.

Después, se comprueba si el cuadrado rojo se encuentra a la derecha (>400) o a la izquierda (< 240) del centro de la imagen.

- Si está a la derecha deberá calcular la orientación (`Th`) hasta la que debe girar, que es la orientación en la que se encuentra menos 90 grados. Aquí podemos encontramos con dos casos:

- Si el robot se encuentra en una orientación inicial situada entre -90 y 180, debemos calcular cuántos grados le quedan para llegar a 180, restárselo a 90 (para saber cuántos grados le quedan para hacer su giro de 90 grados) y restárselo a 180 para obtener la posición final.

- Si su posición final no está comprendida en ese intervalo solo deberá restar 90 grados para obtenerla.

❖ Si está a la izquierda el procedimiento sería exactamente el mismo, salvo que la orientación hasta la que debe girar es su orientación inicial más 90 grados y el caso conflictivo sería en cuadrante 90 y 180.

Una vez hemos girado, avanzaremos durante un segundo y justo después volveremos a nuestra orientación inicial.

Una vez resuelto el problema del punto medio pasamos al área, comprobando si el `area` es:

❖ Mayor que 40000, que significa que estamos muy cerca por lo que retrocederemos.

❖ Menor que 25000, por el contrario estaremos lejos del código y tendremos que avanzar.

Una vez lo hemos colocado devolvemos 1.

Para una mayor aclaración podemos ver todo lo explicado en el código (Tabla 44).

```
int ControladorAmigobot::colocacionQr(float area, float pto)
{
   float thI = robot.getTh();
   float h,j,thf;

    if(pto < 400 && pto >240)
   {   if(area < 40000 && area >25000)
                return 2;
   }

   if( pto == 0)
       return 3;
   if( area == 0)
       return 3;

   if(pto>400)//Girar a la Dcha
   {
       if(thI < -90)
       {      h= 180 - abs(thI);
              j=90-h;
              thf= 180-j;
              ControladorAmigobot::ControladorGiro(thf);
       }else{
              thf = thI - 90;
              ControladorAmigobot::ControladorGiro(thf);
       }
```

```
        // Ahora avanzamos.
        robot.setVel(50);
        sleep(1);
        robot.setVel(0);

        // Volvemos a la posición original.
        ControladorAmigobot::ControladorGiro(thI);

    }else if(pto<240)//Girar a la Izq
    {
        if(thI > 90)
        {       h= 180 - thI;
                j=90-h;
                thf= -180 + j;
                ControladorAmigobot::ControladorGiro(thf);
        }else{
                thf = thI + 90;
                ControladorAmigobot::ControladorGiro(thf);
        }

        // Ahora avanzamos.
        robot.setVel(50);
        sleep(1);
        robot.setVel(0);

        // Volvemos a la posición original.
        ControladorAmigobot::ControladorGiro(thI);
    }


    //Area
    if( area > 40000) //Retrocedo
    {   robot.setVel(-50);
        sleep(2);
        robot.setVel(0);

    }else if( area < 25000){ // Avanzo
        robot.setVel(+50);
        sleep(2);
        robot.setVel(0);
     }

    return 1;
}
```

Tabla 44 - Código ControladorAmigobot::colocacionQr

6. Trabajos relacionados

El presente proyecto es una continuación del proyecto “TAI: Teleoperación de un Amigobot usando dispositivos iOS” de Francisco Javier Esteban Vicente de la Universidad de Salamanca, el cual consiste en una aplicación cliente que se comunica con un servidor conectado al Amigobot para teleoperarlo mediante inclinación o dos botones.

Este proyecto ha servido de base pudiendo reutilizar la conexión mediante un servicio entre el cliente y el servidor y parte de la estructura. Los demás aspectos no han podido ser reutilizados para el presente proyecto, por lo que han sido creados de nuevo y se han introducido otras partes totalmente novedosas.

Sobre esta base se ha realizado el proyecto “QrNav: Navegación de un Robot a través de códigos QR usando un dispositivo iOS” que, como se ha detallado en esta memoria y en los anexos adjuntos de la documentación, permite obtener las imágenes de una webcam conectada al amigobot y manejarlo para dirigirlo hacia un código QR que contenga una dirección. Éste realizará una navegación en forma de L gracias a la odometría, reduciendo los posibles errores causados por ésta mediante la detección de un cuadrado rojo alrededor del QR.

7. Descripción funcional de la aplicación

A continuación se realizará una descripción global del proyecto y se explicarán brevemente cada una de las partes que conforman la interfaz de usuario de la aplicación QrNav. Los servidores ServidorQrNav y *MJPG-Streamer* no tienen interfaz de usuario, por lo que no serán detallados aquí.

7.1 Introducción

En este apartado se realizará una descripción global del proyecto desde un punto de vista tanto de software como de hardware.

En primer lugar, podemos observar en la Figura 50 cómo está estructurado el proyecto desde una vista de hardware.

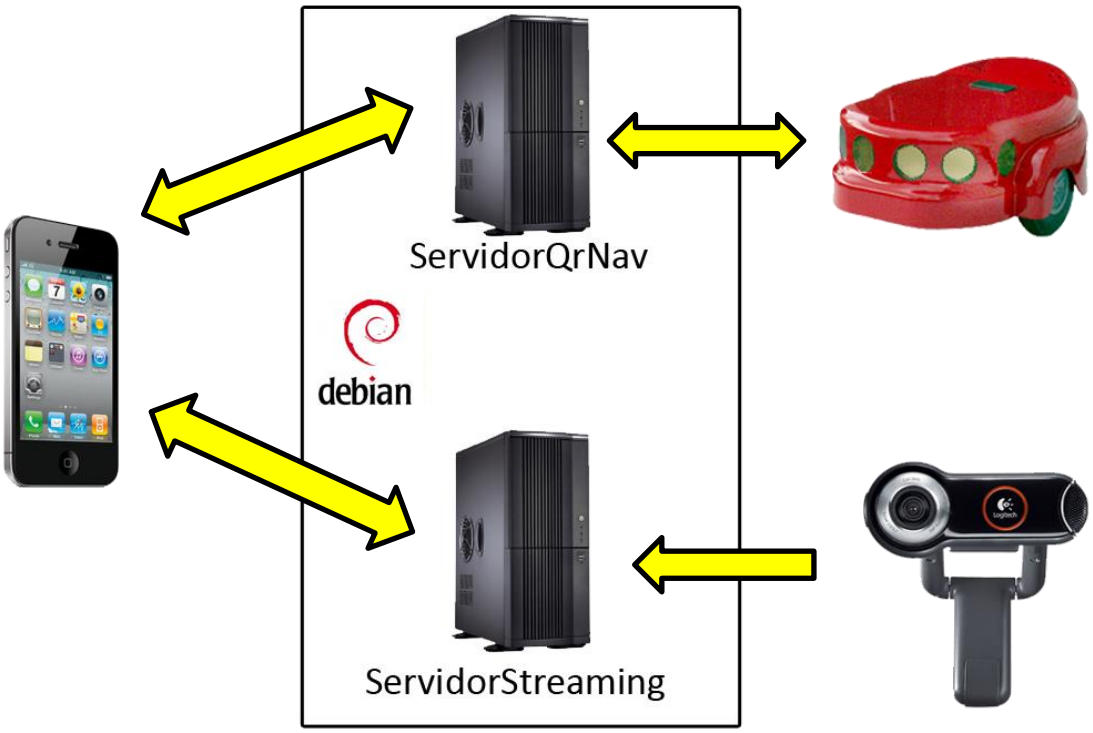

Figura 50 - Estructura Hardware

Como se puede apreciar, el hardware con el que cuenta el proyecto es un iPhone 4 en el que se ejecutará la aplicación QrNav y una máquina Debian en la que se ejecutarán dos servidores: el ServidorQrNav, que se comunica con el robot Amigobot y el ServidorStreaming, que se relaciona con una webcam conectada a la máquina Debian.

En segundo lugar se detalla la estructura del proyecto desde un punto de vista de software.

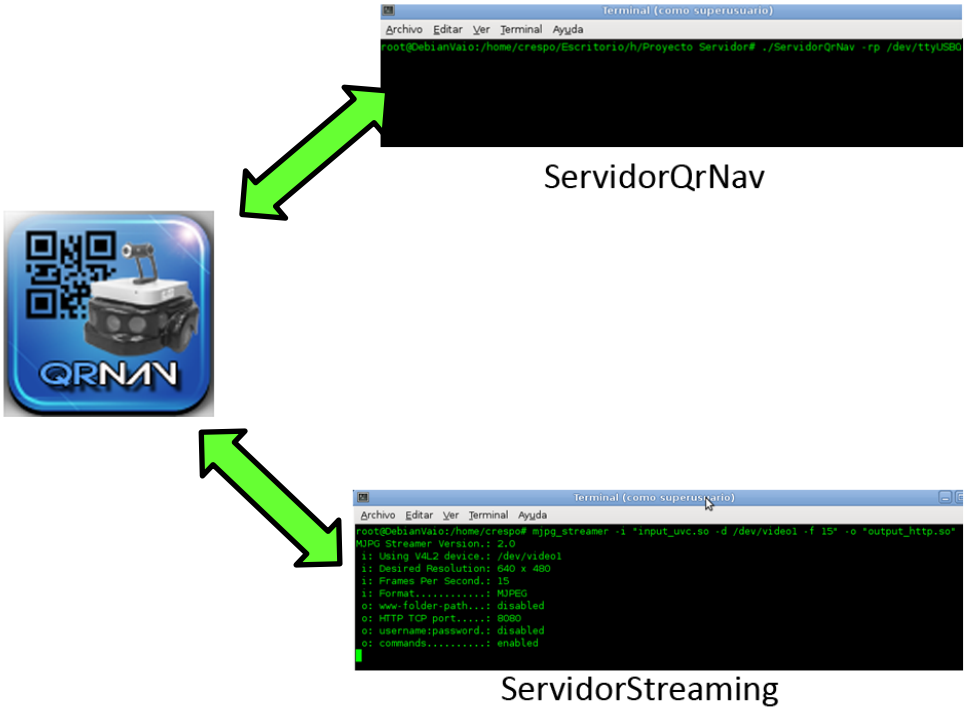

Figura 51 - Estructura Software

Como podemos observar en la Figura 51, el proyecto se divide en tres programas:

- **QrNav**, una aplicación para dispositivos iOS que sirve de interfaz para el usuario y que deberá realizar diversas operaciones. Es el encargado de decidir qué proceso debe ejecutarse en cada momento. Se comunica con el ServidorQrNav enviándole las tareas que debe realizar con el robot y con el servidor streaming (*MJPG-Streamer*) recibiendo las imágenes de la webcam.

- **ServidorQrNav**, servidor que es el encargado de controlar al robot. Se comunica con la aplicación QrNav recibiendo las tareas que tiene que realizar con el robot y enviándole el resultado de dichas tareas.

- **Servidor Streaming**, servidor que obtiene las imágenes de la webcam y se las transmite a la aplicación QrNav a través de *HTTP*.

En la Figura 52 se observa este paso de mensajes antes descrito:

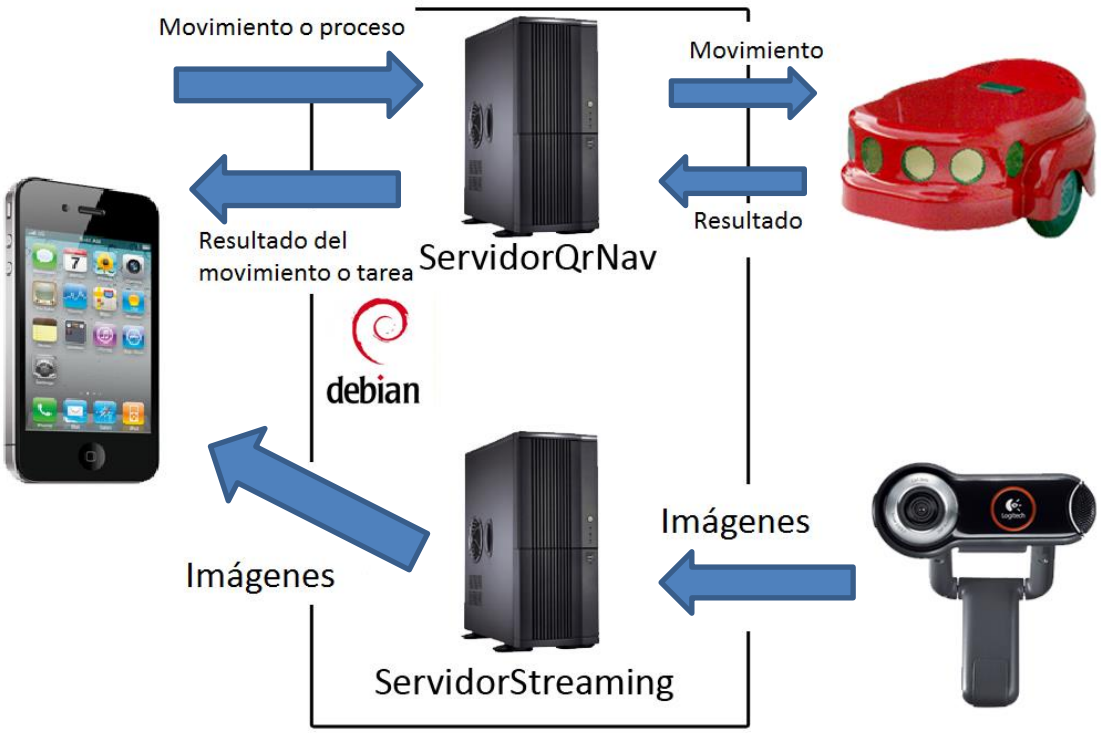

Figura 52 - Estructura paso de mensajes

La aplicación QrNav comienza con una imagen por defecto (Figura 53) que se muestra mientras el dispositivo iOS carga la interfaz de usuario.

Figura 53 - Pantalla default QrNav

Esta aplicación se ejecuta en horizontal para un mejor aprovechamiento del espacio en la pestaña de mandos. Una vez cargada la interfaz, nos saltará una alerta dándonos la bienvenida.

Esta app consta de tres secciones dividas en:

- Bonjour Connect

- Mandos
- Opciones

A cada una de estas secciones se accede pulsando su correspondiente pestaña llamada ítem en el *TabBar*, que es la barra que se encuentra en la parte inferior de la pantalla del dispositivo iOS.

Al principio de la ejecución, la pestaña seleccionada será la de Bonjour Connect.

7.2 Boujour Connect

Esta parte de la aplicación será la que nos permita conectar con el servidor (que controla al Amigobot) usando Bonjour. Consiste en una *TableView* que nos mostrará los servidores disponibles para que podamos elegir el deseado.

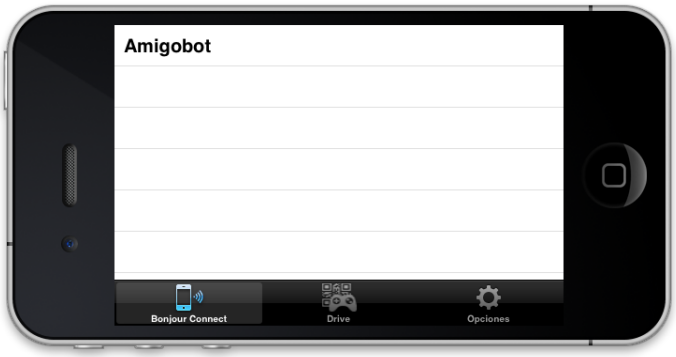

Figura 54 - Pestaña Bonjour Connect con servidor disponible

Cuando pulsemos sobre el nombre de uno de los servicios, la aplicación resolverá en tiempo de ejecución la dirección del servidor y se conectará con él. La aplicación nos indicará que estamos conectados mediante una alerta.

Una vez conectados podremos pasar a la pestaña de mandos para controlar al robot.

7.3 Mandos

Es la parte central de la aplicación. En esta sección podemos:

- Visualizar lo que el robot ve a través de la webcam
- Comenzar la recepción de vídeo o pausarla
- Manejar al AmigoBot a través del joystick
- Mandar a la aplicación leer un código QR
- Saber qué tarea está realizando el Amigobot
- Visualizar el texto contenido en un código QR

- Cancelar la comunicación en caso de emergencia

Cuando el usuario pulse encima de una pestaña se generará un evento pidiendo permiso para abrir la pantalla, que llegará al delegado del *TabBar* decidiendo éste si se permite o no mostrar la vista.

Esta pestaña está inhabilitada inicialmente, es decir, aunque el usuario pulse encima no mostrará la interfaz hasta que se haya conectado con el ServidorQrNav.

Figura 55 - Pestaña de Mandos inicial

Nada más comenzar nos encontraremos con la pantalla de la Figura 55. Para comenzar a recibir las imágenes debemos pulsar sobre el botón de Play y, en cualquier momento, podremos pausar la recepción mediante Stop.

Figura 56 - Pestaña de Mandos en funcionamiento

Esta pantalla (Figura 56) de la aplicación nos ofrece una forma de manejar el Amigobot mediante el *joystick* situado en la parte inferior derecha. Para mover al Amigobot debemos pulsar con un dedo encima del *thumb* del *joystick* y deslizarlo hacia el lado que queremos que se mueva. Como ya se explicó en el apartado 5, la aplicación traducirá las coordenadas del *joystick* a un valor dentro del intervalo de velocidad y se enviarán al ServidorQrNav que las establecerá en el robot.

Para leer un código QR que se encuentre en una imagen captada por la webcam conectada al Amigobot (recibida del servidor streaming), tenemos que pulsar el botón con forma de QR situado en la esquina superior izquierda. Esto hará que el dispositivo iOS intente decodificar el código QR de la imagen y, en caso de éxito, se mostrará en la

pantalla el texto de dicho código (debajo de *Status*). Si además se trata de un QR con una dirección, la aplicación obtendrá las coordenadas de la cadena de texto y las transmitirá al ServidorQrNav para que el robot realice una navegación hasta la dirección indicada. Cuando el robot llegue a la posición de navegación comenzará la colocación para realizar una lectura de otro QR desde una posición centrada Ambos procesos, tanto navegación como colocación, podemos cancelarlos simplemente moviendo el *joystick*.

La aplicación nos indicará en todo momento qué proceso está realizando mediante el *Status*, como por ejemplo: "*Teleoperación…*", "*Colocando…*", "*Navegación Qr…*", Etc.

En caso de que el Amigobot se descontrole o cualquier otra situación anómala, como por ejemplo que no responda, debemos pulsar el botón de emergencia (situado en la parte de arriba) que notificará al ServidorQrNav que debe detener completamente al Amigobot estableciendo el valor cero a ambas velocidades (avance y giro). Ésto sólo ocurrirá si en el momento de la pulsación sigue existiendo conexión con el ServidorQrNav y de éste con el robot. En caso contrario debemos recurrir a la detención física del robot.

7.4 Opciones

La vista de opciones nos dará la posibilidad de personalizar nuestra aplicación para facilitarnos el manejo del Amigobot a nuestro gusto. A esta pestaña (Figura 57) se puede acceder en cualquier momento.

Figura 57 - Pestaña Opciones

Las opciones que nos permiten modificar la aplicación son:

- **Panel de vídeo**, un aspecto visual que nos permite decidir mediante un *switch* o interruptor si queremos visualizar dicho panel en la pantalla de mandos.

- **Sensibilidad**, un valor entero que establecerá cuánto debemos mover el *joystick* para que comience a contar como teleoperación. Para modificarlo sólo debemos mover el *Slider* hacia izquierda o derecha.

- **Velocidad Máxima de Avance**, nos permite introducir un valor entre [500 – 50] que será la velocidad máxima de avance a la que podremos mover el

AmigoBot con el *joystick*. Si introducimos un valor por encima del rango se nos establecerá el valor máximo y, si es por debajo, fijará el valor mínimo.

- **Velocidad Máxima de Giro**, nos permite introducir un valor entre [50 – 10] que será la velocidad máxima de giro a la que podremos mover el AmigoBot con el *joystick*. Si introducimos un valor por encima del rango se nos establecerá el valor máximo y, si es por debajo, fijará el valor mínimo.

Para esconder el teclado cuando hayamos finalizado de introducir la velocidad, sólo debemos realizar un toque en el resto de la pantalla.

Dado que la pantalla no tiene mucho espacio al ser horizontal y tener el *TabBar*, se ha introducido un *scroll* para poder navegar en vertical y poder visualizar todas las opciones (Figura 58). La aplicación se encargará de que el teclado nunca oculte el *TextField* seleccionado para introduir el valor, cambiando para ello el tamaño del *ScrollView*.

Figura 58 - ScrollView de la pestaña de opciones

La aplicación guardará el valor de los ajustes en la base de datos cada vez que salga de la pestaña de opciones.

Por último, en esta pestaña también tenemos un botón de información que únicamente muestra una ventana modal informativa (Figura 59).

Figura 59 - Ventana modal informativa

8. Conclusiones

Después del trabajo realizado para la consecución del Trabajo Fin de Grado, se han obtenido una serie de conclusiones que, a continuación, me dispongo a valorar.

En primer lugar, se consideran superados los objetivos técnicos y de documentación del software a construir, al igual que los objetivos personales que se marcaron al comienzo del proyecto. La construcción del proyecto, así como la realización de toda la documentación, me ha servido para afianzar y practicar los conceptos teóricos adquiridos a lo largo de la carrera.

Cabe destacar, que se han comprendido las fases que deben seguirse a la hora de desarrollar un producto software, así como la importancia que tienen las fases previas de análisis y diseño, ya que van a marcar de manera muy determinante la calidad del producto final, así como ayudan a reducir el tiempo total de desarrollo. Lo considero un punto importante en el aprendizaje, ya que, hasta ahora, parecía que el desarrollo del software sólo consistía en escribir código sin más.

A pesar de considerar que el presente proyecto posee una dificultad elevada, al haber utilizado áreas y ámbitos con los que no había tratado previamente (como puede ser Objetive-C, tratamiento de imágenes, robótica, etc) considero que trabajar sobre él me ha resultado beneficioso, puesto que me ha aportado bastantes conocimientos sobre las áreas anteriormente mencionadas. Además, considero que a nivel personal ha sido un motivo de superación puesto que me enfrentaba a algo desconocido pero que he conseguido resolver satisfactoriamente.

También quiero destacar que en mi opinión, la planificación del tiempo para desarrollar el proyecto es esencial, dicho de otro modo, se deben marcar unos objetivos a cumplir dentro de unas fechas concretas y tener ciertos aspectos implementados o partes de la memoria realizadas.

En cuanto a la programación sobre dispositivos iOS, mencionaré varios puntos:

- Aprender Objetive-C no ha resultado demasiado costoso, puesto que está basado en C y disponía de conocimientos previos sobre el mismo. Lo único que desctaca de éste (aparte del paradigma de orientación a objetos) es el cambio de la sintaxis para declarar y llamar a los métodos pero, cuando te acostumbras, resulta más fácil.

- Sobre XCode 4.2, comentar que es el mejor *IDE* que he utilizado pues es muy potente. También me han resultado fascinantes algunas herramientas que en versiones anteriores no estaban disponibles como los *Storyboard* ya que facilitan la creación de interfaces gráficas. La tarea más ardua de todas ha sido integrar las diferentes bibliotecas con las que trabaja la aplicación QrNav en XCode (*Zxing* y *OpenCV*).

- En relación a Cocoa Touch, me ha parecido que es un *framework* muy completo y que se mejora en cada nueva versión iOS disponible. Apple ha

sido capaz de crear un marco de trabajo atractivo que proporciona muchas facilidades a la hora de crear aplicaciones.

La tarea más costosa y en la que más tiempo se empleó fue, sin lugar a dudas, la transmisión de vídeo streaming en tiempo real, puesto que fue la parte a la que más tiempo dediqué y de la que no existe apenas documentación. Los únicos conocimientos con los que contaba eran los adquiridos en la asignatura de Redes y, a partir de ahí, fui madurando soluciones hasta encontrar una totalmente válida.

El uso de la biblioteca *Zxing* no fue muy complicado ya que encontré la funcionalidad que necesitaba y se adaptó perfectamente a la estructura del proyecto. En cambio, con *OpenCV*, buscar un cuadrado en una imagen resultó algo más complejo (debido también a la propia complejidad de la tarea) y nunca había realizado ningún tipo de tratamiento de imágenes.

La utilización de C++, al haber cursado programación orientada a objetos, me resultó más asequible. El único problema es que usar g++ como compilador y GEdit como editor de textos, hace que la programación resulte aún más pesada (sobre todo si se realiza la comparación con la forma de programar en iOS con XCode), aunque se podría haber usado algún *IDE* como Eclipse. Con respecto a *ARIA*, comentar que ha resultado una biblioteca bastante potente aunque, por otra parte, su documentación deja bastante que desear.

El desarrollo de algoritmos me ha resultado una tarea bastante entretenida, pues se debe pensar en todas las posibilidades, en qué hacer cuando se recibe un parámetro u otro, en cómo realizar los controladores con sus problemas ya existentes, en la navegación en forma de L, etc. El desarrollo de estas tareas no ha sido rápido sino que ha conllevado numerosas pruebas que han demostrado fallos, situaciones que no se habían contemplado, variables sueltas…, pudiendo existir algún caso aislado que nunca se hubiera tenido en cuenta.

Resaltar también que se ha intentado trabajar mucho la una interfaz gráfica, cuidando multitud de aspectos y creando los botones, icono, pantalla default y demás componentes a medida con el programa Photoshop, puesto que es un aspecto importante de cara a los usuarios de la aplicación.

Por último, con respecto al pequeñisimo mundo de la robótica en el que me he sumergido, quiero comentar que me ha resultado fascinante e interesante y que cada vez que pruebo el proyecto me resulta gratificante ver lo que he conseguido realizar con un Smartphone y un robot. Sin duda, acerté en la elección de un proyecto que abarcara todos estos temas pese a que su dificultad pudiera ser más elevada que en otros ámbitos, aunque también dependerá en gran medida de la persona que lo realice.

8.1 Líneas futuras

En este apartado se van a describir diversas maneras de continuar y ampliar el proyecto. Éste podría ser la base de numerosas utilidades y aplicaciones y se abrirían, de este modo, un amplio conjunto de posibles mejoras:

- Mejorar los errores que tiene la odometría con un robot más potente y ruedas de mayor calidad e incluyendo nuevas partes en el proyecto (como la ya incluida ColocaciónQr) para tener más puntos de referencia en relación a la posición en la que nos encontramos.
- Corregir los errores de tratamiento de imágenes provocados por la luz ambiental.
- Cambiar los controladores lineales a controladores PID (propocional integral derivativo) comúnmente usados en el área de robótica.
- Añadir una nueva funcionalidad en la que el usuario pueda elegir entre dos direcciones que lea de un código QR, es decir, al decodificar un código QR el usuario podrá seleccionar la dirección que quiere seguir. Por ejemplo, en un museo, para ir hasta una escultura u otra.
- Añadir otra funcionalidad: que el contenido del código QR sea reproducido en voz por el robot o dispositivo móvil para personas invidentes y que, mientras realice la navegación, suene un breve pitido para que estas personas puedan seguirlo.
- Crear una pequeña interfaz gráfica para entornos Linux que lanzara al ServidorQrNav y al programa *MJPG-Streamer*, en lugar de tener que hacerlo desde consola y de forma separada.
- Añadir más opciones a la aplicación para hacer que la teleoperación se adapte mejor al usuario y sobre más aspectos gráficos de la pestaña de Mandos.
- Optimizar la aplicación para consumir menos RAM (*random-access memory*, en español, memoria de acceso aleatorio) CPU y batería. Estos tres componentes son críticos en el rendimiento de cualquier sistema móvil y es importante cuidarlos.
- Internacionalización de la aplicación para que dispusiera de varios idiomas.

9. Glosario

API: (Application Programming Interface - Interfaz de Programación de Aplicaciones). Se trata de un grupo de rutinas (conformando una interfaz) que provee un sistema operativo, una aplicación o una biblioteca y que definen cómo invocar desde un programa un servicio que éstos prestan. En otras palabras, una API representa un interfaz de comunicación entre componentes software.

Aplicación: En informática, una aplicación es un tipo de programa informático diseñado como herramienta para permitir a un usuario realizar uno o diversos tipos de trabajo. Esto es lo que la diferencia principalmente de otros tipos de programas como los sistemas operativos, que hacen funcionar al ordenador o las utilidades, que realizan tareas de mantenimiento o de uso general.

Bar Code: Representación óptica, legible por máquinas, de los datos relativos al objeto al que está unido.

Base de Datos: Conjunto de datos pertenecientes a un mismo contexto y almacenados sistemáticamente para su posterior uso.

C++: Lenguaje de programación diseñado a mediados de los años 1980 por Bjarne Stroustrup. Extiende el exitoso lenguaje de programación C con mecanismos que permitan la manipulación de objetos. Por tanto, desde el punto de vista de los lenguajes orientados a objetos, el C++ es un lenguaje híbrido.

Cocoa: Conjunto de frameworks orientados a objetos creados por Apple que permiten el desarrollo de aplicaciones nativas para Mac OS X en el lenguaje orientado a objetos Objective C.

Cocoa Touch: API para la creación de programas para dispositivos iOS (iPad, iPhone y iPod Touch) creado por Apple Inc. Cocoa Touch proporciona una capa de abstracción al sistema operativo iOS. Está basado en Cocoa.

Controlador: Componente de la arquitectura MVC o del patrón de diseño MVC, responde a eventos, usualmente acciones del usuario, e invoca peticiones al modelo y, probablemente, a la vista.

Core Animation: Es uno de los APIs contenidos en Cocoa y Cocoa Touch, en concreto un API de visualización. Proporciona una vía para que los desarrolladores puedan producir interfaces de usuario animadas.

Core Data: Es uno de los APIs contenidos en Cocoa y Cocoa Touch, en concreto un API que nos permite el acceso a la capa persistente. Nos proporciona una interfaz Orientada a Objetos como recubrimiento de una base de datos relacional subyacente, de modo que podamos trabajar con nuestra base de datos mediante el paradigma orientado a objetos.

Debian: Sistema operativo de libre distribución que utiliza el núcleo Linux. La mayor parte de sus herramientas básicas vienen del Proyecto GNU. Debian destaca por ser un sistema robusto y con una interfaz agradable al usuario.

Delegate: Para evitar las relaciones del tipo es-un-rol-ejecutado-por, que son mal modeladas con herencia tenemos el patrón delegate, mediante el cual un objeto delegará parte de sus funciones en otro.

Dispositivos iOS: Dispositivos creados por Apple que incorporan el sistema operativo iOS, actualmente existen 3 modelos: iPhone iPad e iPod Touch.

Encoder: es un sensor electro-opto-mecánico que unido a un eje, proporciona información de la posición angular. Su fin, es actuar como un dispositivo de realimentación en sistemas de control integrado.

FPS: Sigla utilizada para referirse a los frame por segundo.

Frame: Representa una imagen concreta dentro del conjunto de ellas que conforman un vídeo.

Framework: Estructura conceptual y tecnológica de soporte definido, normalmente con artefactos o módulos de software concretos, con base a la cual otro proyecto de software puede ser más fácilmente organizado y desarrollado. Típicamente, puede incluir soporte de programas, bibliotecas y un lenguaje interpretado, entre otras herramientas, para así ayudar a desarrollar y unir los diferentes componentes de un proyecto.

GNU/Linux: Uno de los términos empleados para referirse a la combinación del núcleo o kernel libre similar a Unix denominado Linux, que es usado con herramientas de sistema GNU. Su desarrollo es uno de los ejemplos más prominentes de software libre; todo su código fuente puede ser utilizado, modificado y redistribuido libremente por cualquiera bajo los términos de la GPL.

Hardware: Corresponde a todas las partes tangibles de un sistema informático.

HTML: Siglas de HyperText Markup Language, es el lenguaje de marcado predominante para la elaboración de páginas web. Es usado para describir la estructura y el contenido en forma de texto, así como para complementar el texto con objetos tales como imágenes.

IDE: Entorno de programación que ha sido empaquetado como un programa de aplicación, es decir, consiste en un editor de código, un compilador, un depurador y un constructor de interfaz gráfica (GUI).

Ingeniería del Software: Es el área de la ingeniería que ofrece métodos y técnicas para desarrollar y mantener software. Trata con áreas muy diversas de la informática y de las ciencias de la computación, tales como construcción de compiladores, sistemas operativos, o desarrollos Intranet/Internet, abordando todas las fases del ciclo de vida del desarrollo de cualquier tipo de sistemas de información.

Interface Builder: Aplicación del entorno de programación para Mac OS X e iPhone OS usada para hacer interfaces gráficas de usuario. Es una aplicación muy potente con opciones como el uso inteligente de las guías para colocar componentes.

Interfaz Grafica de Usuario: Conocida también como GUI es un programa informático (o parte de uno) que actúa de interfaz de usuario, utilizando un conjunto de imágenes y objetos gráficos para representar la información y acciones disponibles en la interfaz. Su principal uso consiste en proporcionar un entorno visual sencillo para permitir la comunicación entre el usuario y la aplicación.

MVC: Patrón de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario y la lógica de negocio en tres componentes distintos: Model, Vista y Controlador.

Obejective C: Lenguaje de programación orientado a objetos creado como un superconjunto de C pero que implementase un modelo de objetos parecido al de Smalltalk. Originalmente fue creado por Brad Cox y la corporación StepStone en 1980.

OSI: Modelo de red descriptivo creado por la Organización Internacional para la Estandarización en el año 1984. Es decir, es un marco de referencia para la definición de arquitecturas de interconexión de sistemas de comunicaciones.

Pantalla Táctil: Pantalla que mediante un toque directo sobre su superficie permite la entrada de datos y órdenes al dispositivo y, a su vez, muestra los resultados introducidos previamente.

Patrones de Diseño: Es una solución a un problema de diseño. Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser reutilizable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.

Plugin: Es un módulo de hardware o software que añade una característica o un servicio específico a un sistema más grande.

Proceso Unificado: Marco de desarrollo de software que se caracteriza por estar dirigido por casos de uso, centrado en la arquitectura y por ser iterativo e incremental. El refinamiento más conocido y documentado del Proceso Unificado es el Proceso Unificado de Rational o simplemente RUP.

Scroll: Tipo de movimiento que aparce en dispositivos informáticos para desplazarse vertical u horizontalmente por la pantalla.

Singleton: Patrón de diseño orientado a restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.

Sockets: Concepto abstracto por el cual dos programas (posiblemente situados en computadoras distintas) pueden intercambiar cualquier flujo de datos, generalmente de manera fiable y ordenada.

Software: Equipamiento lógico o soporte lógico de un sistema informático; comprende el conjunto de los componentes lógicos necesarios que hacen posible la realización de tareas específicas, en contraposición a los componentes físicos, que son llamados hardware.

SQL: Lenguaje declarativo de acceso a bases de datos relacionales que permite especificar diversos tipos de operaciones en estas.

TabBar: Uno de los Idiomas principales de Cocoa Touch que nos permite la separación de las distintas partes de una aplicación mediante una barra horizontal con iconos, situada en la parte inferior de la aplicación. Cada una de las secciones tendrá un icono en la TabBar, pulsando en él accedemos a dicha parte de la aplicación.

TableView: Uno de los Idiomas principales de Cocoa Touch que nos permite mostrar al usuario una serie de elementos comunes en forma de una lista unidimensional para que el usuario pueda elegir el que quiera.

Wi-Fi: Mecanismo de conexión de dispositivos electrónicos de forma inalámbrica.

XCode: Entorno de desarrollo integrado (IDE, en sus siglas en inglés) de Apple. Se suministra gratuitamente junto con Mac OS X. XCode trabaja conjuntamente con Interface Builder, una herencia de NeXT, una herramienta gráfica para la creación de interfaces de usuario.

XML: Metalenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium (W3C). Es una simplificación y adaptación del SGML y permite definir la gramática de lenguajes específicos.

Zeroconf: Conjunto de técnicas que permiten crear de forma automática una red IP sin configuración o servidores especiales.

10. Bibliografía

10.1 Libros (Medios Impresos)

- **Programación: Desarrollo de aplicaciones para IPhone & IPad.**

  Autores: Joe Conway, Aaron Hillegass. Ediciones Anaya Multimedia. 2011.

- **Objetive C: Curso práctico para Programadores Mac Os X, IPhone y Ipad.**

  Autor: Fernando López Hernández. Grupo RC. 2012

- **Programación: IOS 5.**

  Autores: Rob Napier, Mugunth Kumar. Ediciones Anaya Multimedia. 2012.

- **Desarrollo de aplicaciones para IOS 5.**

  Autor: Wei – Meng Lee. Ediciones Anaya Multimedia. 2012.

- **Beginning Mac Programming:** Develop with Objective-C and Cocoa

  Autor: Tim Isted

- **Cocoa Programming:** A Quick-Start Guide for Developers

  Autor: Daniel H Steinberg

- **IPhone SDK Development**

  Autores: Bill Dudney, Chris Adamson

- **iPad Programming:** A Quick-Start Guide for iPhone Developers.

  Autores: Daniel h Steinberg, Eric T Freeman

- **ARIA Documentation**

  Autor: Mobile Robotics

- **Apuntes de la asignatura Ingeniería del Software**, del 3º curso de Ingeniería Técnica en Informática de Sistemas en la Facultad de Ciencias, de la Universidad de Salamanca.

  Autor: Francisco José García Peñalvo

- **Apuntes de la asignatura Redes**, del 3º curso de Ingeniería Técnica en Informática de Sistemas en la Facultad de Ciencias, de la Universidad de Salamanca.

  Autor: Ángeles Mª Moreno Montero

❖ **Guía de Realización y documentación** (Versión 1.52) Ingeniería Técnica en Informática.

Autores: Francisco José García Peñalvo, Jesús Mudes Raedos, Mario Gerardo Piattini Velthuis, José Rafael García-Bermejo Giner, María N. Moreno García

10.2 Sitios Web (Medios Digitales)

❖ **Stack Overflow:**

- http://stackoverflow.com/questions/1738828/how-to-use-learn-video4linux2-on-screen-display-output-apis
- http://stackoverflow.com/questions/14287/increasing-camera-capture-resolution-in-opencv
- http://stackoverflow.com/questions/7882781/ios-how-to-decode-image-with-zxing
- http://stackoverflow.com/questions/5803352/didselectviewcontroller-not-getting-called-when-in-more-section
- http://stackoverflow.com/questions/5396134/using-nsscanner-to-parse-a-string
- http://stackoverflow.com/questions/3692237/can-you-cast-an-nsstring-to-an-nsinterger-if-the-string-contains-numbers-only
- http://stackoverflow.com/questions/8738485/creating-a-tabbarcontrollerdelegate-in-a-storyboard
- http://stackoverflow.com/questions/4199758/return-to-top-of-uinavigationcontroller-stack-when-clicking-a-tab
- http://stackoverflow.com/questions/3555906/how-to-convert-nsnumber-to-int-in-objective-c
- http://stackoverflow.com/questions/8465769/convert-storyboard-from-iphone-to-ipad
- http://stackoverflow.com/questions/6308382/start-a-new-line-paragraph-on-a-label-or-textview
- http://stackoverflow.com/questions/5266350/xcode-4-archive-is-greyed-out

- **Video for Linux Two API Specification:**
  - http://v4l2spec.bytesex.org/spec/capture-example.html
  - http://v4l2spec.bytesex.org/spec/book1.htm
- **LWN.net:**
  - http://lwn.net/Articles/204545/
- **API Video for Linux:**
  - http://profesores.elo.utfsm.cl/~agv/elo330/2s03/projects/Video4Linux/Video4Linux.html
  - http://profesores.elo.utfsm.cl/~agv/elo330/2s03/projects/Video4Linux/
- **Antonym:**
  - http://antonym.org/libfg/
- **Linux UVC driver and tolos:**
  - http://www.ideasonboard.org/uvc/
- **Usuario Debian Blog:**
  - http://usuariodebian.blogspot.com.es/2008/01/webcam-usb-en-debian-drivers-gspca-y.html
- **Oracle Blog:**
  - https://blogs.oracle.com/madlab/entry/getting_started_with_video4linux2_usbvc
  - https://blogs.oracle.com/madlab/entry/using_the_video4linux2_usbvc_vidioc
- **Tux Brain:**
  - http://www.tuxbrain.com/content/c%C3%B3digos-de-barras-webcam-cualquier-aplicacion
- **V4L2 interface for Camera Capture:**
  - http://old.nabble.com/V4L2-interface-for-Camera-Capture-td32629203.html
- **Noway:**
  - http://www.noway.es/index.php?q=LogitechPro9000

- **Tecnoyo:**
  - http://www.tecnoyo.com/blog/?p=47
- **Devwo:**
  - http://es.devwo.com/soft/sort014/sort032/sort0150/list150_1.html
- **Procesamiento Digital de Imágenes Blog:**
  - http://up-procesamientodigitaldeimag.blogspot.com.es/2011/06/captura-de-imagen-con-opencv.html
- **API:**
  - http://trac.koka-in.org/libdecodeqr/wiki/ApiReference
- **OF forum:**
  - http://forum.openframeworks.cc/index.php?topic=1547.0
- **Así somos Linux Blog:**
  - http://www.clopezsandez.com/2012/01/fedora-y-qr-en-terminal.html
- **DsynFLO:**
  - http://dsynflo.blogspot.com.es/2010/06/libdecodeqr-libdecodeqr-is-cc-library.html
- **Scribd:**
  - http://es.scribd.com/doc/76878232/21/Table-2-Libdecodeqr-failed-statuses
  - http://es.scribd.com/doc/76878232/7/LIBDECODEQR
  - http://es.scribd.com/doc/74984011/The-iOS-5-Developer-s-Cookbook-Core-Concepts-and-Essential-Recipes-for-iOS-Programmers
- **Libqrencode:**
  - http://fukuchi.org/works/qrencode/index.html.en
  - http://fukuchi.org/works/qrencode/manual/qrencode_8h.html
- **LinuxHispano:**
  - http://www.linuxhispano.net/2011/03/30/leer-codigos-qr-en-ubuntu/

- **Elotrolado.net:**
  - http://www.elotrolado.net/hilo_leer-codigos-de-barras-qr-desde-la-webcam-en-linux_1414676
- **CHW:**
  - http://www.chw.net/foro/webmasters-f91/915637-a-jugar-con-codigos-qr-lector-web-usando-webcam.html
  - http://www.chw.net/foro/webmasters-f91/915637-a-jugar-con-codigos-qr-lector-web-usando-webcam.html
- **TiroKart Blog:**
  - http://tirokartblog.wordpress.com/2010/09/08/qr-codes-decoded-on-the-beagleboard/
- **Saving YUVY image from V4L2 buffer to file:**
  - http://www.spinics.net/lists/vfl/msg42235.html
- **Saving YUVY image from V4L2 buffer to file:**
  - http://comments.gmane.org/gmane.comp.video.video4linux/43838
  - http://permalink.gmane.org/gmane.comp.video.video4linux/43850
- **Nashruddin.com**:
  - http://nashruddin.com/OpenCV_Face_Detection
  - http://nashruddin.com/Streaming_OpenCV_Videos_Over_the_Network
- **Computer Vision Talks:**
  - http://computer-vision-talks.com/2011/01/using-opencv-in-objective-c-code/
  - http://computer-vision-talks.com/2010/12/building-opencv-for-ios/
- **Yoshimasa Niwa Blog:**
  - http://niw.at/articles/2009/03/14/using-opencv-on-iphone/en
- **Rajavo Blog:**
  - http://rajavo.blogspot.com.es/2011/03/opencv-en-el-iphone.html

- **Cloud District**
  - http://clouddistrict.com/articulo/tutorial-reconocimiento-de-patrones-con-opencv-en-iphone-i/
  - http://clouddistrict.com/articulo/tutorial-reconocimiento-de-patrones-con-opencv-en-iphone-iii/
- **Instalar OpenCV 2.1.0 en GNU/Linux Ubuntu 10.04:**
  - http://gibup.files.wordpress.com/2010/06/tutorial-instalacion-opencv-2-1-0.pdf
- **Sourceforge:**
  - http://sourceforge.net/
  - http://sourceforge.net/projects/opencvlibrary/?source=directory
  - http://sourceforge.net/projects/opencvlibrary/files/opencv-unix/
  - http://sourceforge.net/projects/opencvlibrary/files/opencv-unix/2.3.1/OpenCV-2.3.1a.tar.bz2/download
  - http://sourceforge.net/apps/mediawiki/zbar/index.php?title=HOWTO:_Scan_images_using_the_API#C.2B.2B
  - http://sourceforge.net/projects/mjpg-streamer/
  - http://sourceforge.net/projects/zbar/forums/forum/1072195/topic/4766667
  - http://sourceforge.net/apps/mediawiki/mjpg-streamer/index.php?title=Main_Page
- **CV Reference Manual:**
  - http://www.seas.upenn.edu/~bensapp/opencvdocs/ref/opencvref_cv.htm
- **Universidad Politécnica de Valencia:**
  - http://web-sisop.disca.upv.es/~imd/cursosAnteriors/2k8-2k9/videos/trabajandoConOpenCV/secciones/Video.html
  - http://web-sisop.disca.upv.es/~sdv/captura.html
  - http://web-sisop.disca.upv.es/~sdv/
  - http://web-sisop.disca.upv.es/imd/cursosAnteriors/2k3-2k4/copiaTreballs/serdelal/trabajoIMD.xml

- **Robótica y visión artificial Blog:**
  - http://asysver.blogspot.com.es/2010/09/solucion-reto-ii-reconocer-una-peloto-y.html
- **Raywenderlich:**
  - http://www.raywenderlich.com/3932/how-to-create-a-socket-based-iphone-app-and-server
  - http://www.raywenderlich.com/5138/beginning-storyboards-in-ios-5-part-1
  - http://www.raywenderlich.com/934/core-data-on-ios-5-tutorial-getting-started
- **Programming with OpenCV:**
  - http://www.cs.iit.edu/~agam/cs512/lect-notes/opencv-intro/opencv-intro.html
- **HighGUI Reference:**
  - http://www.ai.rug.nl/vakinformatie/pas/content/Highgui/opencvref_highgui.htm
- **OpenCV captura WebCam:**
  - http://www.openvov.com/blog/?p=145
- **Microsoft msdn:**
  - http://social.msdn.microsoft.com/Forums/es-ES/vcppes/thread/4a11f2db-c29c-4d96-8dda-408d82ab374c
  - http://msdn.microsoft.com/es-es/library/bb972240.aspx
- **OpenCV 2.1**:
  - http://opencv.willowgarage.com/documentation/cpp/index.html
  - http://opencv.willowgarage.com/documentation/c/index.html
- **Robótica Blog:**
  - http://blogrobotica.linaresdigital.com/2011/02/filtro-de-deteccion-de-bordes-canny.html
- **Tracking de personas a partir de visión artificial:**
  - http://earchivo.uc3m.es/bitstream/10016/10118/4/PFC_Javier_Yanez_Garcia.pdf

- **Giingo:**
  - http://giingo.org/index.php?post/2007/01/11/230-linux-en-un-mac-mini-intel-core-duo
- **Blog migueldiazrubio:**
  - http://www.migueldiazrubio.com/2012/01/03/desarrollo-ios-primeros-pasos-con-storyboard/
  - http://www.migueldiazrubio.com/2010/10/27/desarrollo-ios-primeros-pasos-con-core-data/
- **Penguintosh Blog:**
  - http://penguintosh.com/tag/builder/
- **Taringa:**
  - http://www.taringa.net/posts/linux/5940418/Instalar-wifi-en-Debian-_driver-y-gestor-de-red-wifi_.html
- **Wiki.debian.org:**
  - http://wiki.debian.org/iwlwifi?action=show&redirect=iwlagn
- **Live555:**
  - http://www.live555.com/liveMedia/
- **Streaming con VLC en Ubuntu 9.10:**
  - http://miblockdenotix.wordpress.com/2009/11/11/streaming-con-vlc-en-ubuntu-9-10/
- **Code.google:**
  - http://code.google.com/p/aradia/source/browse/trunk#trunk%253Fstate%253Dclosed
  - http://code.google.com/p/zxing/
- **Aplicaciones sobre sctp:**
  - http://earchivo.uc3m.es/bitstream/10016/7376/1/PFC_Jorge_Lopez_Aragoneses_Presentacion.pdf
- **Read.pudn:**
  - http://read.pudn.com/downloads118/sourcecode/multimedia/streaming/503792/RTSP/Client/rtsp.c__.htm

- **Streaming Video with RTSP and RTP:**
  - http://www.csee.umbc.edu/~pmundur/courses/CMSC691C/lab5-kurose-ross.html
- **Sistema Distribuido – Cliente/Servidor para la reproducción de video:**
  - http://msdarkici.wordpress.com/2010/10/14/c-desarrollo-de-sistema-distribuido-%E2%80%93-clienteservidor-para-la-reproduccion-de-video-iii/
- **Github:**
  - https://github.com/dropcam/dropcam_for_iphone
  - https://github.com/horsson/mjpeg-iphone
- **Control remoto (web) de un robot Aibo:**
  - http://www.recercat.net/bitstream/handle/2072/9056/PFCCa%C3%B1etePoyatos.pdf?sequence=1
- **ZBar bar:**
  - http://zbar.sourceforge.net/iphone/index.html
  - http://zbar.sourceforge.net/api/zbar_8h.html#70d23bf7c2ee5e60fa0f19d4f66dc1bb
- **Zbarcam:**
  - http://linux.die.net/man/1/zbarcam
- **Código QR: Codificando y Decodificando en Linux:**
  - http://hvivani.com.ar/2011/09/28/codigo-qr-codificando-y-decodificando-en-linux/
- **Streaming a video using a program in C/C++:**
  - http://systemsdaemon.blogspot.com.es/2011/03/http-streaming-video-using-program-in-c.html
- **MJPEG Stream Client:**
  - http://forum.wiibrew.org/read.php?11,64272
- **Universidad de Córdoba:**
  - www.uco.es/~i62gicaj/RTP.pdf

- **C-Pascal:**
  - http://www.algoritmia.net/articles.php?id=52
- **De Pascal a C++:**
  - http://www.cefce.com.ar/gecop/archivos/de_pascal_a_cpp.html
- **Freepascal:**
  - http://www.freepascal.org/docs-html/rtl/objects/tmemorystream.html
- **Habrahabr:**
  - http://habrahabr.ru/post/115808/
- **Install ZXing in XCode 4:**
  - http://yannickloriot.com/2011/04/how-to-install-zxing-in-xcode-4/
- **Opengroup:**
  - http://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html
- **A Custom Tab Bar Controller:**
  - http://padlamp.blogspot.com.es/2012/03/from-top-down-custom-tab-bar-controller.html
- **Iphonedevsdk:**
  - http://www.iphonedevsdk.com/forum/iphone-sdk-development/3122-didselectviewcontroller-tab-bar-tutorial.html
- **Glyphish:**
  - http://glyphish.com/
- **Iphonedroid:**
  - http://www.iphonedroid.com/
- **Untitled.es:**
  - http://untitled.es/uiswitch-con-ios/
  - http://untitled.es/uislider-con-ios/
  - http://untitled.es/enviar-email-con-ios/
  - http://untitled.es/web-view-con-ios/
  - http://untitled.es/enviar-tweet-con-ios/

- **PunteroaVoid Blog:**
  - http://www.punteroavoid.com/blog/2012/03/01/como-evitar-que-el-teclado-de-ios-oculte-campos-de-texto/
- **Maniacdev:**
  - http://maniacdev.com/2012/03/tutorial-getting-started-with-core-data-in-ios-5-using-xcode-storyboards/
- **Zbutton:**
  - http://zbutton.wordpress.com/2011/03/06/desmitificando-core-data-para-ios/
- **Core Data en iPhone:**
  - http://www.zenbrains.com/blog/2010/04/core-data-en-iphone/
- **Desarrollo En Mac:**
  - http://www.desarrolloenmac.com/2011/02/10/anyadiendo-core-data/
- **Wikipedia:**
  - http://es.wikipedia.org/wiki/Video4Linux
  - http://es.wikipedia.org/wiki/Apple
  - http://es.wikipedia.org/wiki/IOS_(sistema_operativo)
  - http://es.wikipedia.org/wiki/IPad
  - http://es.wikipedia.org/wiki/IPod_Touch
  - http://es.wikipedia.org/wiki/IPhone
  - http://es.wikipedia.org/wiki/Proceso_Unificado
  - http://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o
  - http://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o#Relaci.C3.B3n_de_principales_patrones_GoF_.28Gang_Of_Four.29
  - http://es.wikipedia.org/wiki/Singleton
  - http://es.wikipedia.org/wiki/Singleton#Objective_C
  - http://es.wikipedia.org/wiki/Modelo_Vista_Controlador
  - http://es.wikipedia.org/wiki/Delegation_(patr%C3%B3n_de_dise%C3%B1o)

- http://es.wikipedia.org/wiki/Cliente-servidor
- http://es.wikipedia.org/wiki/Socket_de_Internet
- http://es.wikipedia.org/wiki/Odometr%C3%ADa
- http://es.wikipedia.org/wiki/OpenCV
- http://es.wikipedia.org/wiki/C%C3%B3digo_QR
- http://es.wikipedia.org/wiki/Bonjour_(software)
- http://es.wikipedia.org/wiki/Zeroconf#Detecci.C3.B3n_de_servicios
- http://en.wikipedia.org/wiki/Multicast_DNS
- http://en.wikipedia.org/wiki/Avahi_%28software%29
- http://es.wikipedia.org/wiki/Objective-C
- http://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n
- http://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos
- http://es.wikipedia.org/wiki/C%2B%2B
- http://es.wikipedia.org/wiki/Xcode
- http://es.wikipedia.org/wiki/Microsoft_Word
- http://es.wikipedia.org/wiki/Photoshop
- http://es.wikipedia.org/wiki/IPhone_SDK
- http://es.wikipedia.org/wiki/Iphone#iPhone_4
- http://es.wikipedia.org/wiki/Lenguaje_Unificado_de_Modelado

❖ **Proceso Unificado de Desarrollo de Software:**

- http://www.rodolfoquispe.org/blog/que-es-el-proceso-unificado-de-desarrollo-de-software.php
- http://antares.itmorelia.edu.mx/~jcolivares/courses/pm10a/rup.pdf

❖ **4+1 Vistas:**

- http://synergix.wordpress.com/2008/07/31/las-4-mas-1-vistas/

❖ **Vista de casos de uso:**

- http://clases3gingsof.wetpaint.com/page/Vista+de+casos+de+uso

- **Vistas de la Arquitectura del Software:**
  - http://cic.puj.edu.co/wiki/lib/exe/fetch.php?media=materias:modelo4_1.pdf
- **MCV:**
  - http://cuentacontrolx.wix.com/mvc
- **Patrones de Diseño:**
  - http://jms32.eresmas.net/tacticos/programacion/documentacion/logica/patrones/patrones0101.html
- **Odometría**:
  - http://optimus.meleeisland.net/links/04_odometry.html
- **Diferencias entre Código QR y Código BIDI:**
  - http://www.codigo-qr.es/diferencias-entre-codigo-qr-y-codigo-bidi/
- **QR. Definición y estructura:**
  - http://nuevastecsomamfyc.wordpress.com/2012/05/08/que-es-un-codigo-qr-definicion-y-estructura/
- **Estructura Código Qr:**
  - http://codigo-qr.blogspot.com.es/2011/10/qr-estructura-y-explicacion-de-un.html
- **Mobilerobot**
  - http://robots.mobilerobots.com/wiki/ARIA
  - http://www.mobilerobots.com/ResearchRobots/AmigoBot.aspx
- **Developer Apple:**
  - https://developer.apple.com/technologies/ios/cocoa-touch.html
  - http://developer.apple.com/library/ios/#documentation/uikit/reference/UIKit_Framework/_index.html
  - https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/ObjC_classic/_index.html
  - https://developer.apple.com/technologies/ios/
  - http://developer.apple.com/library/ios/#documentation/uikit/reference/uiview_class/uiview/uiview.html

- http://developer.apple.com/library/ios/#documentation/UIKit/Reference/UIViewController_Class/Reference/Reference.html
- http://developer.apple.com/library/ios/#DOCUMENTATION/UIKit/Reference/UITableView_Class/Reference/Reference.html
- http://developer.apple.com/library/ios/#DOCUMENTATION/UIKit/Reference/UITabBar_Class/Reference/Reference.html
- https://developer.apple.com/library/mac/#documentation/DataManagement/Devpedia-CoreData/coreDataStack.html
- http://developer.apple.com/library/ios/#documentation/DataManagement/Conceptual/iPhoneCoreData01/Articles/01_StartingOut.html
- https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/LoadingResources/CocoaNibs/CocoaNibs.html
- http://developer.apple.com/library/mac/#documentation/General/Conceptual/Devpedia-CocoaApp/Storyboard.html

❖ **Core Data (1):**

- http://www.microedition.biz/blog/?p=549