UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

DISEÑO DE UNA INTERFAZ DE USUARIO AVANZADA PARA SOFTWARE DE CONTROL AÉREO MEDIANTE UNA CÁMARA RGBD

TRABAJO FINAL DEL

Grado en Ingeniería Aeroespacial

REALIZADO POR

D. Álvaro Edo Martínez

TUTORIZADO POR

Dr. D. Àngel Rodas Jordà

FECHA: **València, septiembre, 2019**

RESUMEN

El creciente aumento del tráfico aéreo a nivel global presenta un reto: optimizar hasta las tareas más básicas relacionadas con el control de tráfico aéreo (ATM), para mejorar la capacidad del espacio aéreo y la seguridad del mismo. Es en este contexto donde surge la idea del presente trabajo final de grado: mejorar la comunicación humano-máquina aplicando un método de entrada alternativo e innovador para controlar una aplicación avanzada de radar ATM.

El presente trabajo final de grado hace uso de una cámara RGBD (*LEAP Motion* ™) que ofrece un método de entrada alternativo al ratón basado en la obtención de un mapa de profundidad de la escena a partir del cuál se extrae un modelo tridimensional de las manos del usuario. Con ayuda del mismo, se diseñará una interfaz que facilite la interacción con un *software* de control aéreo; lo que permitirá introducir distintas interpretaciones gestuales avanzadas de las intenciones del usuario.

El trabajo se desarrollará en el lenguaje de programación *Java* ™ y, junto al diseño del *software* de la interfaz, ofrecerá un banco de pruebas que mezcle la información de los vuelos obtenidos del transpondedor instalado en la Escuela (*servantena*) con vuelos sintéticos.

Palabras clave: GUI; Control aéreo; ATM; Cámara RGBD; LEAP Motion; Java

RESUM

El creixent augment del tràfic aeri a nivell global presenta un repte: optimitzar fins i tot les tasques més bàsiques relacionades amb el control de tràfic aeri (ATM), per millorar la capacitat de l'espai aeri i la seguretat del mateix. És en aquest context on sorgeix la idea del present treball fi de grau: millorar la comunicació humà-màquina aplicant un mètode d'entrada alternatiu i innovador per tal de controlar una aplicació avançada de radar ATM.

El present treball fi de grau fa servir una càmera RGBD (*LEAP Motion* ™) que ofereix un mètode d'entrada alternatiu al ratolí basat en la obtenció d'un mapa de profunditat de l'escena a partir del qual s'extrau un model tridimensional de les mans de l'usuari. Amb l'ajuda del mateix, es dissenyarà una interfície que facilite la interacció amb un *software* de control aeri; cosa que permetrà introduir diverses interpretacions gestuals avançades de les intencions de l'usuari.

El treball es desenvoluparà en el llenguatge de programació *Java* ™ i, junt al disseny del *software* de la interfície, oferirà un banc de proves que mescle la informació dels vols obtinguts del transponedor instal·lat a l'Escola (*servantena*) amb vols sintètics.

Paraules clau: GUI; Control aeri; ATM; Càmera RGBD; LEAP Motion; Java

ABSTRACT

The increasing growth of air traffic at a global scale presents a challenge: optimising even the most basic tasks related to air traffic management (ATM), in order to improve the capacity of the global air space and its security. It is in this context where the idea of this final degree project arises: to improve the human-machine communication by applying an alternative and innovative input method to control an advanced ATM radar application.

This final degree project makes use of an RGBD camera (*LEAP Motion* ™) which offers an input method alternative to the mouse based in the acquisition of a depth map of the scene from which a three-dimensional model of the user's hands is extracted. With its help, an interface which eases the interaction with an air traffic management software will be designed; in order to introduce several advanced gestural interpretations of the intentions of the user.

This project will be developed in *Java* ™ programming language and, together with the interface design, will offer a test environment which uses information from the flights obtained by the transponder installed at the School (*servantena*), with artificial flights.

Keywords: GUI; Air traffic management; ATM; RGBD camera; LEAP Motion; Java

TABLA DE CONTENIDOS

TABLA DE FIGURAS

1. INTRODUCCIÓN

Según estimaciones de *Boeing* ®, en los próximos 20 años la flota global de aeronaves en servicio se verá duplicada[1]. Es un hecho evidente que el sector de la aviación civil está experimentando un crecimiento muy notable, y cada vez es más difícil evitar que el espacio aéreo se congestione, especialmente en zonas como Europa, con una alta densidad de población. Por tanto, cada vez es mas necesario (e incluso urgente) buscar soluciones que nos permitan optimizar hasta las tareas más simples en el campo de ATM, para evitar retrasos innecesarios.

Son muchos los ámbitos del control de tráfico aéreo que podrían ser estudiados, y para los cuales se podrían presentar posibles mejoras y optimizaciones; pero este proyecto se centrará en uno de los aspectos más elementales: la interacción humano-máquina. En su día a día y teniendo en cuenta la importancia que cobran actualmente de los equipos informáticos para realizar tareas de forma rápida y óptima, un controlador de tráfico aéreo depende completamente de dichos equipos para realizar su trabajo; y es crucial que su manejo sea tan óptimo como sea posible.

1.1. Motivación y justificación

De este afán de optimizar e innovar hasta en las tareas más básicas, como la interacción del controlador con sus equipos, es de donde surge este proyecto. En su día a día, un controlador utiliza los equipos informáticos necesarios para desempeñar sus funciones con dos dispositivos convencionales: teclado y ratón. Es llamativo que, con el nivel de desarrollo al que se ha llegado en el campo de la informática, nunca se haya propuesto un sistema alternativo haciendo uso de las nuevas tecnologías que hay disponibles; ya que, a pesar de que prácticamente cualquier persona con acceso a un ordenador sabe utilizarlos, hay nuevos sistemas que permitirían controlarlo de forma más eficiente.

En el presente proyecto, se explora la vía de la tecnología de control por gestos, ya que se trata de una de las formas más elementales de expresión que tiene una persona; y también, debido a que se trata de una perspectiva poco común en el campo de ATM. Mediante esta vía, también se satisface la intención de innovar y buscar sistemas diferentes que permitan hacer más eficiente el control de tráfico aéreo, que es de importancia vital dado el notable crecimiento del tráfico aéreo a nivel mundial.

Por tanto, se podría establecer como motivación principal proponer un sistema alternativo y más eficiente de manejar los equipos y sistemas que utiliza un controlador aéreo a diario, ya que la combinación teclado-ratón se empieza a encontrar obsoleta con la aparición de nuevas tecnologías. Se podrían valorar diferentes tecnologías (por ejemplo, pantallas táctiles, dispositivos de seguimiento de retina, etc.) pero por razones económicas, y, sobre todo, al ser una tecnología

[1] Agencia EFE. (11 jun. 2015). "Boeing prevé una demanda de 38.050 nuevos aviones en 20 años por valor de 5,6 billones de dólares". Efe.com

que no es muy habitual encontrar en el campo de ATM, se ha elegido explorar el terreno del control por gestos.

Cabe destacar que este proyecto no tiene como meta obtener dicho sistema de manejo de forma completa, ya que la precisión y fiabilidad que nos proporcionan los dispositivos que podemos encontrar a nivel de consumidor (el dispositivo *LEAP Motion* ™, en concreto) no se corresponden con las que serían exigibles en el campo de ATM, a pesar de tener un nivel de precisión bueno. Sin embargo, ofrece posibilidades innumerables (entre ellas, en esta Universidad se propuso un sistema de guiado de drones utilizando este dispositivo), y se puede utilizar para ilustrar cómo sería una solución innovadora y alternativa al teclado-ratón en el control de tráfico aéreo, para permitir una interacción más optimizada entre el controlador y sus equipos.

En definitiva, se tratará de explorar una opción muy interesante como el control por gestos y proponer un prototipo, en líneas generales, de cómo sería la aplicación de esta nueva tecnología al campo de ATM. A partir de dicho prototipo, se podría desarrollar en un futuro un hardware que cumpliera las especificaciones necesarias y que superara las certificaciones correspondientes al sector, y pudiera ser utilizado en situaciones reales; y un software adaptado al mismo y con todas las opciones y herramientas que necesitaría un controlador ATM.

1.2. Objetivos

Como se ha comentado anteriormente, la meta final del presente proyecto será la obtención de una aplicación de muestra con la que se visualizarán las distintas aeronaves cercanas en un mapa de radar y que incluya un modo con aviones ficticios para ilustrar cómo se controlaría el trafico con dicha aplicación; todo ello controlado íntegramente por gestos mediante el dispositivo *LEAP Motion* ™ y sin mayor ayuda que las manos del usuario.

Este objetivo podrá ser dividido en cuatro hitos o subobjetivos. En primer lugar, adquirir conocimientos profundos sobre la tecnología de control por gestos y la extracción de información de una cámara RGBD; ya que conocer la tecnología que se está tratando será imprescindible para realizar una aplicación intuitiva y fiable que cualquier usuario pueda utilizar.

En segundo lugar, utilizar los conocimientos obtenidos para desarrollar a bajo nivel un sistema de control por gestos completo que pueda ser aplicado a una interfaz de usuario avanzada, hito que podría ser considerado como uno de los más relevantes, al ser el que proporciona el punto innovador al presente proyecto.

Como tercer hito, que sería el más puramente aeroespacial, se podría establecer la consecución de un sistema de control de tráfico aéreo obteniendo una aplicación radar básica que representara los distintos tráficos aéreos obtenidos desde *servantena*.

Finalmente, y como cuarto hito o subobjetivo, se conseguiría combinar el sistema de control por gestos con la aplicación de control de tráfico aéreo para así completar la meta de este proyecto, añadiéndole nuevas funciones (como por ejemplo, mostrar tráficos ficticios para demostrar el funcionamiento completo de la aplicación).

2. MATERIALES Y MÉTODOS

Para el desarrollo del presente proyecto, se han requerido varios recursos. Dada la naturaleza puramente informática del mismo, podemos dividir dichos materiales y métodos en *hardware* y *software*.

2.1. Hardware

En el terreno del *hardware* serán necesarios únicamente tres elementos: un equipo informático para desarrollar y ejecutar la aplicación, una cámara RGBD para obtener los gestos y posiciones de las manos y una antena ADS-B para recibir los tráficos aéreos de la zona

2.1.1. Equipo informático

Tanto para el desarrollo de la aplicación como para su ejecución, es necesario un ordenador. Al tener que estar procesándose imágenes, reconociendo gestos, obteniendo datos de los aviones cercanos, etc., se requiere un mínimo de potencia. No sería estrictamente necesario tener un equipo muy potente, pero ayudaría en su correcta ejecución y evitaría que se bloqueara.

El equipo utilizado en nuestro caso ha sido un *Apple MacBook Pro* ® de 13 pulgadas con *macOS* ® *Mojave* ™ (10.14.x); sin embargo, la aplicación no está limitada al sistema *macOS* ®, ya que es multiplataforma y se podría ejecutar perfectamente en *Microsoft Windows* ® y en sistemas basados en *Linux*.

Figura 1: Equipo utilizado (Fuente: ifixit.com)

2.1.2. Dispositivo LEAP Motion ™

El principal componente a nivel de *hardware* del que constará este proyecto será la cámara RGBD que se utilizará para reconocer los distintos gestos y para controlar la aplicación. Una vez abierta, la aplicación será controlada íntegramente utilizando este dispositivo, sin que sea necesario que el usuario toque el equipo ni utilice teclado, ratón u otros dispositivos.

Este dispositivo fue desarrollado por la compañía homónima fundada en 2010 por *David Holz* y *Michael Buckwald*, que se creó para desarrollar la idea que el primero tuvo en 2008[2] y que resultó en el anuncio del propio dispositivo en 2012[3].

Figura 2: Dispositivo *LEAP Motion* ™ (Fuente: leapmotion.com)

Se trata de una pequeña cámara conectada al ordenador mediante USB 3.0, que obtiene imágenes y distancias (utilizando infrarrojos), que son tratadas en tiempo real para obtener las coordenadas en tres dimensiones de distintos puntos de las manos, así como distintos gestos muy básicos.

Dado que la naturaleza de este proyecto es solamente la implementación de este sistema y no el propio reconocimiento de imágenes y distancias, este dispositivo será tratado como una "caja negra"; es decir, utilizando las distintas funciones de librería proporcionadas por el fabricante, se obtendrán datos en tiempo real sobre las manos del usuario, que serán tratados por la propia aplicación.

2.1.3. Antena ADS-B

Para representar las aeronaves cercanas, la aplicación recibirá datos del servidor de la antena ADS-B (Sistema Automático Dependiente de Vigilancia – Difusión) ubicada en la Escuela Técnica Superior de la Ingeniería del Diseño (ETSID), en la *Universitat Politècnica de València* (*servantena.etsid.upv.es*, puerto 30002).

Este servidor envía una serie de paquetes con la información sobre los tráficos que está recibiendo, parámetros como coordenadas, *callsign*, código identificador hexadecimal, altitud, velocidad, etc. Al igual que con el dispositivo *LEAP Motion* ™, se tratará como una caja negra, ya que el objeto de este proyecto no es obtener datos del servidor, si no que será algo complementario para conseguir un funcionamiento mejor y más realista de la aplicación.

Cabe destacar que la tecnología ADS-B requiere de la colaboración de la aeronave (emite su posición, etc. en todo momento), así como de la disponibilidad de sistemas GNSS (Sistemas Globales de Navegación por Satélite); por tanto, no seria una forma fiable de controlar el tráfico aéreo. Además, dada la ubicación de la antena y debido a que no se dispone de ninguna más, solo se recibirán tráficos de la zona del este de España. Sin embargo y a pesar de estas condiciones, este sistema será más que suficiente para poder obtener la aplicación de ejemplo

[2] Richardson, Nicole Marie. (28 may. 2013). "One Giant Leap for Mankind". Inc.com
[3] Leap Motion, Inc. (s.f.). "Leap Motion - Información". Facebook.com

de la tecnología de control por gestos que se pretende conseguir en este proyecto, para indicar el camino a seguir en la obtención de un hipotético sistema que tuviera los requisitos necesarios para poder ser utilizado en casos reales.

2.2. Software

A nivel de *software*, que será la parte más relevante del proyecto, contaremos principalmente con dos factores: el lenguaje de programación a utilizar y el entorno de desarrollo. Además, se utilizarán otros recursos, como mapas, iconos y fuentes, que serán discutidos en el punto 6.4 (*Proceso de desarrollo: Aplicación de radar*), donde se comentarán aspectos de la aplicación final tales como el diseño.

2.2.1. Lenguaje de programación: Java ™

El lenguaje a utilizar será *Java* ™ debido a varios factores. En primer lugar, esto permitiría que la aplicación sea multiplataforma, cosa que con el resto de lenguajes es más complicada. En segundo lugar, el fabricante del dispositivo proporciona unas librerías para obtener datos de este, y *Java* ™ es uno de los lenguajes para los que hay librerías disponibles. Y finalmente, existe una gran cantidad de bibliografía y código disponible de la Universidad en el que basarse y con el que resolver dudas.

Java ™ es un lenguaje de alto nivel y propósito general, orientado a objetos y compatible con la mayoría de dispositivos actuales (más de 3 billones de dispositivos en el mundo utilizan *Java*). Fue desarrollado por *Sun Microsystems* ® en 1995[4], compañía que fue posteriormente adquirida por *Oracle* ® en 2010[5].

2.2.2. Entorno de desarrollo: Apache NetBeans 11.1

A la hora de escribir código, es de vital importancia dónde se haga, ya que unos entornos facilitan la programación más que otros. Para este proyecto se ha optado por *NetBeans*; principalmente porque es de *software* abierto; y, además, por su potente generador de código y diseñador de interfaces gráficas, sus innumerables opciones y funciones y su uso intuitivo.

Figura 3: Logotipo de *Apache NetBeans 11.1* (Fuente: netbeans.apache.org)

[4] Oracle. (s.f.). "The History of Java ™ Technology". Oracle.com
[5] Oracle. (s.f.). "Oracle and Sun Microsystems". Oracle.com

Al inicio del proyecto, se empezó utilizando *NetBeans* 8.2, la versión estable más reciente, pero posteriormente se presentó *NetBeans 11.1*, esta vez como parte de *Apache*, una fundación que apoya y promueve los proyectos de *software* abierto.

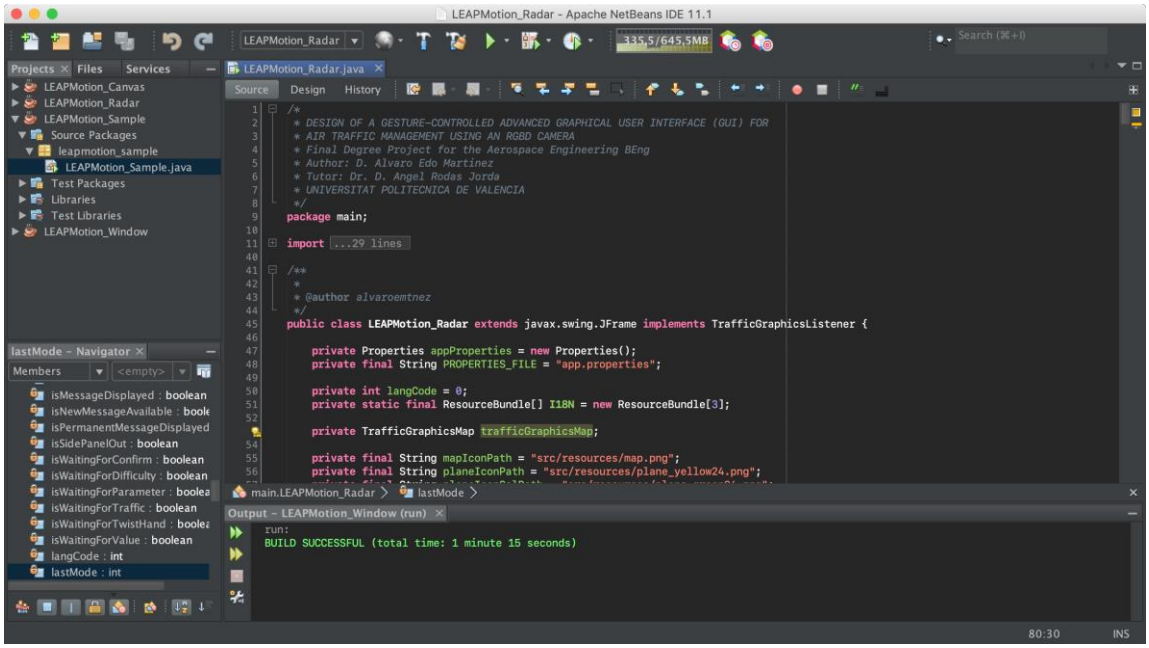


Figura 4: Captura de pantalla del entorno de desarrollo

NetBeans también fue desarrollado por *Sun Microsystems* ®, pero convertido a *software* abierto en el año 2000. En el año 2016 fue donado íntegramente a la fundación *Apache* por *Oracle* ® (nuevo propietario tras la adquisición de *Sun Microsystems* ®). Hasta la fecha, el IDE (Entorno de Desarrollo Integrado) *NetBeans* cuenta con más de 18 millones de descargas y más de 800 000 desarrolladores registrados[6].

[6] Apache Software Foundation. (s.f.). "About Apache NetBeans". Apache.org

3. ESTADO DEL ARTE

Para saber a ciencia cierta cómo encarar el proyecto, es necesario conocer el nivel de desarrollo actual que tienen las tecnologías que se van a tratar. Para el presente proyecto, son relevantes dos de ellas: el *software* de control de tráfico aéreo usado actualmente y las tecnologías de reconocimiento de gestos.

También se debe tener en cuenta que en el campo de ATM se deben tener unas exigencias y requisitos de fiabilidad determinados, ya que un correcto funcionamiento de los distintos sistemas es crucial para el mantenimiento de la seguridad aérea.

3.1. Software ATM actual

En España, el sistema utilizado para el control del tráfico aéreo es SACTA ® (Sistema Automatizado de Control de Tránsito Aéreo), desarrollado por *ENAIRE* ®. Integra todos los centros de control de ruta, aproximación y aeródromo españoles, de forma que la gestión se realiza de una manera coordinada, proporcionando funciones como la ayuda a la planificación del tráfico y procesamiento de información relativa a planes de vuelo, procesamiento de información radar y asociación de la misma con planes de vuelo, presentación de información meteorológica, alertas sobre desviaciones de aeronaves, etc[7].

Figura 5: Vista radar del SACTA (Fuente: enaire.es)

Tanto el *software* como el equipamiento *hardware* (este último a cargo de *INDRA* ®) se encuentran perfectamente compenetrados y trabajan de acuerdo con los requisitos exigibles en ATM. Se trata, por tanto, de una tecnología muy desarrollada y fiable, utilizada en todos los centros gestionados por *ENAIRE* ®.

[7] ENAIRE. (s.f.). "SACTA". Enaire.es

3.2. Tecnologías de reconocimiento de gestos

El control por gestos ha sido tradicionalmente utilizado en el cine y la televisión para mostrar mundos "futuristas". Hoy en día, sin embargo, es una realidad y podría tener múltiples aplicaciones. Compañías como *Intel* ® y *Microsoft* ® han presentado distintos usos de este tipo de tecnología, e incluso se ha llegado a utilizar en la industria automovilística por fabricantes como *BMW* ®, que han integrado el control por gestos al navegador y equipo multimedia.

Para el reconocimiento de gestos, se precisa de un dispositivo que cree una imagen en tres dimensiones del entorno, que posteriormente pueda ser procesada para extraer de la misma en tiempo real los gestos correspondientes. Esto se puede realizar de dos formas: con ultrasonidos o con infrarrojos. *Elliptic Labs* ®, por ejemplo, creó un sistema de reconocimiento de gestos utilizando señales de ultrasonidos para reconocerlos (al estilo de los murciélagos). Respecto a los infrarrojos, destaca el *Kinect for Xbox 360* ®, una cámara RGBD conectada a la propia videoconsola para crear una imagen en tres dimensiones del entorno, e interaccionar así con la misma[8].

El control por reconocimiento de gestos se presenta como una forma revolucionaria de interacción con los dispositivos. Es en este contexto en el que surgen dispositivos como el *LEAP Motion* ™, que será utilizado para el presente proyecto. Se trata, como se ha comentado anteriormente, de una pequeña cámara RGBD que, mediante la proyección de multitud de puntos infrarrojos y combinándolos con imágenes, extrae las posiciones de varios puntos de la mano. Procesando dichos puntos se pueden reconocer gestos y realizar un control eficiente de una aplicación o juego.

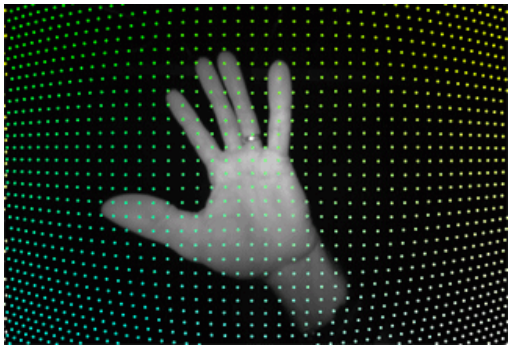


Figura 6: Imagen sin procesar del dispositivo *LEAP Motion* ™ con puntos de calibración superpuestos (Fuente: leapmotion.com)

A nivel de usuario, existen multitud de juegos compatibles con el dispositivo, así como *software* para utilizarlo como señalador (a modo de sustituto del ratón o *trackpad*) y para otras aplicaciones profesionales. Además, el fabricante proporciona una serie de librerías y una API (Interfaz de Programación de Aplicaciones) con la que desarrollar cualquier tipo de *software* valiéndose de los datos obtenidos por el dispositivo a modo de "caja negra".

[8] Wendorf, Marcia. (31 mar. 2019). "How Gesture Recognition Will Change Our Relationship With Tech Devices". Interestingengineering.com

4. DISPOSITIVO LEAP MOTION ™

La base de este proyecto será, sin ningún tipo de dudas, el dispositivo *LEAP Motion* ™, ya que será el nexo entre el usuario y la propia aplicación y el componente principal del sistema de control por gestos. Extraerá toda la información necesaria a tiempo real sobre las manos del operador y permitirá que este interactúe con la interfaz. Es por ello que es importante tener una sólida idea de su funcionamiento y particularidades; ya que, a pesar de que será tratado como una caja negra, conocerlo correctamente asegurará su correcta implementación.

4.1. Descripción técnica

A nivel de *hardware*, el dispositivo *LEAP Motion* ™ cuenta con dos cámaras con objetivo gran angular y tres LEDs infrarrojos, que emiten luz infrarroja con una longitud de onda de 850 nm (es decir, fuera del espectro visible) que será recogida por las dos cámaras[9].

Figura 7: Estructura interna del dispositivo *LEAP Motion* ™ (Fuente: leapmotion.com)

El rango de visión proporcionado por las dos cámaras es bastante amplio, gracias a los objetivos a los que se encuentran acopladas; formando un ángulo de visión del controlador de 150º longitudinalmente respecto a las cámaras, y 120º perpendicularmente a las mismas.

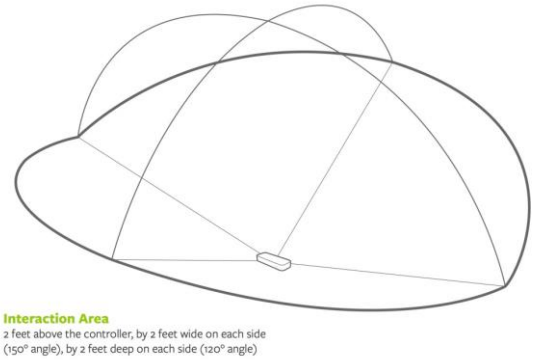


Figura 8: Área de interacción del dispositivo *LEAP Motion* ™ (Fuente: leapmotion.com)

[9] Colgan, Alex. (9 ago. 2014). "How Does the Leap Motion Controller Work?". Leapmotion.com

Como se puede comprobar, a nivel técnico y respecto al *hardware*, no se trata de un sistema complejo; sin embargo, es en el *software* donde se despliega toda la complejidad de este sistema de visión artificial.

4.2. Principio de funcionamiento

Desde el punto de vista del *software*, se utilizan ambas cámaras para crear un sistema de visión estereoscópica (es decir, imitando los ojos humanos, se combinan las imágenes de ambas cámaras para obtener un mapa de profundidad).

Los LEDs emitirán infinidad de puntos de calibración (como se ha mostrado previamente en la figura 6) que serán recogidos por las cámaras, y combinando las imágenes de ambas, se conseguirá la distancia desde cada uno de los puntos a las mismas. Conociendo la distancia a la que se encuentran todos los puntos diferenciales, se creará el mapa de profundidad, que podrá ser posteriormente interpretado.

Para discernir la luz emitida por los LEDs de la recibida proveniente de otras fuentes de interferencia, la primera será emitida a pulsos. En cada pulso, las cámaras recibirán la luz de los LEDs y, valiéndose de la imagen anterior al pulso, compensarán la iluminación infrarroja externa.

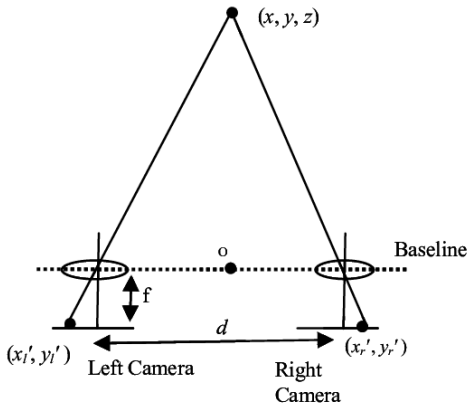


Figura 9: Esquema de funcionamiento de la visión estereoscópica (Fuente: researchgate.net)

Como se puede observar en la figura 9, con parámetros como la distancia entre cámaras, y la distancia focal de las mismas, se pueden obtener las coordenadas de un determinado punto visible por ambas (tras ejecutar varias transformaciones matemáticas). Esto se realizaría en el rango de los infrarrojos (con la imagen obtenida tras compensar las fuentes de interferencia) y para cada uno de los puntos diferenciales, con lo que se conseguiría una imagen en tres dimensiones del entorno del dispositivo (definido en la figura 8).

Una vez conseguido esto, se aplicarán técnicas de visión artificial y reconocimiento de imágenes para obtener la mano o manos incluidas en el rango de visión. Toda esta información podrá ser extraída del dispositivo gracias a la API (Interfaz de Programación de Aplicaciones) proporcionada por el fabricante, que con una serie de métodos y funciones permitirá extraer todo tipo de magnitudes reales sobre el estado en tiempo real de las manos del usuario.

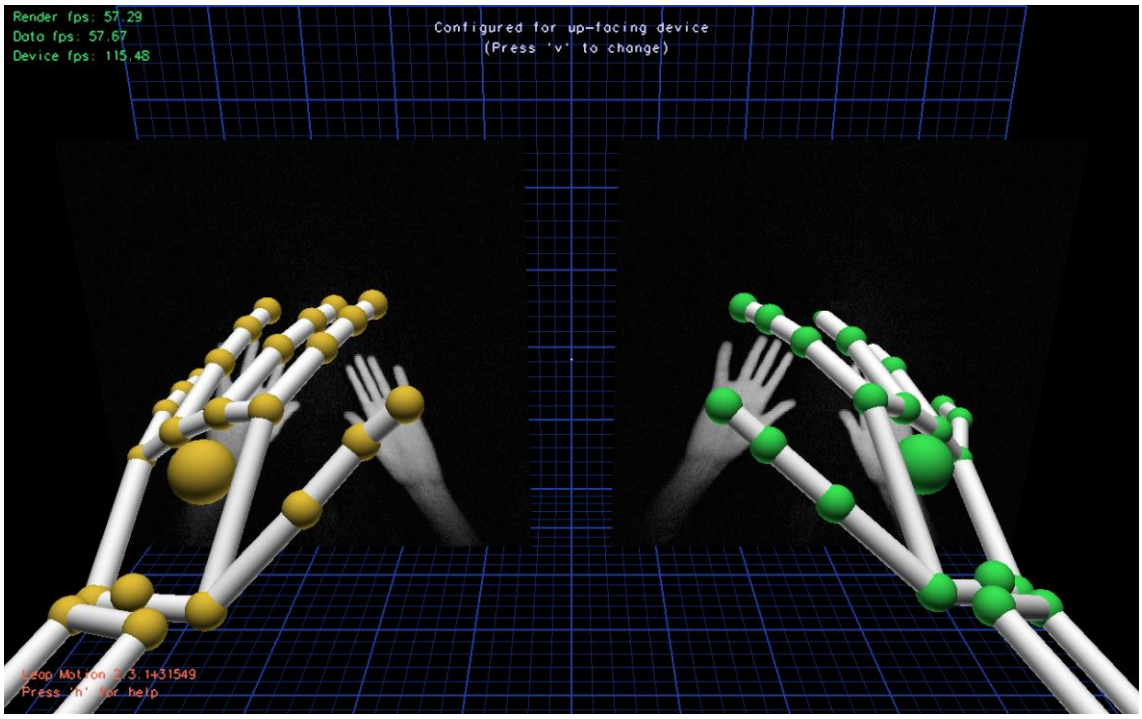


Figura 10: Captura del visualizador proporcionado por el fabricante

El dispositivo generará, a nivel último, una serie de datos relativos a las posiciones y movimiento de multitud de puntos de la mano. En la figura 10 podemos observar una representación en tres dimensiones del estado actual de las manos del usuario, junto con las imágenes (tras la representación) sin tratar obtenidas por los sensores de ambas cámaras.

5. IMPLEMENTACIÓN

Una vez conocidos tanto la idea que se desea poner en práctica y sus objetivos, como el estado actual de las tecnologías utilizadas y su funcionamiento, será necesario trazar un camino a seguir para el desarrollo del proyecto. Cabe destacar que la tecnología de reconocimiento de gestos (como se ha explicado en los puntos anteriores) es compleja y no existe demasiada bibliografía sobre ella; por tanto, el aprendizaje sobre la obtención y tratamiento de la información proporcionada por el dispositivo *LEAP Motion* ™ será una parte importante a cumplir (consiguiendo el primero de los subobjetivos).

A continuación, habiendo conseguido tratar la información proporcionada por el controlador, se establecerá, en líneas muy generales, un sistema de control por gestos que funcione de forma fiable (cumpliendo así el segundo de los subobjetivos). Entonces, se procederá a familiarizarse y hacer funcionar una aplicación básica de radar, que no realice ninguna función pero que implemente algún tipo de "colaboración" con el dispositivo *LEAP Motion* ™ (como, por ejemplo, seleccionar tráficos). Con ello se cumplimentará el tercer subobjetivo, se conseguirá obtener datos sobre los tráficos aéreos circundantes y representarlos en un mapa básico.

Finalmente, se utilizará todo lo aprendido (tanto sobre control por gestos y el dispositivo *LEAP Motion* ™, como sobre recepción de datos de *servantena* y representación de tráficos aéreos en un mapa) para desarrollar la aplicación final, con todas sus funciones.

De esta forma, se establecerán cuatro fases en el desarrollo del proyecto. En primer lugar, se pondrá en funcionamiento el dispositivo y se observará su comportamiento, para familiarizarse con el mismo. Para ello se buscará algún tipo de aplicación *HelloWorld* que pueda establecerse como punto de partida.

En segundo lugar, se desarrollará una aplicación básica que extraiga datos del dispositivo y los muestre por pantalla, cosa que será interesante para aprender cómo obtener información y tratarla de forma básica, así como comprender cómo se comunica el dispositivo con *Java* ™.

En tercer lugar, se obtenerá una aplicación más avanzada que utilizará gestos y dará una respuesta gráfica en tiempo real sobre la información que obtiene el dispositivo. Con ello se conseguirá un conocimiento profundo sobre el tratamiento de los datos extraídos del controlador, así como sobre la tecnología de control por gestos y la monitorización de los mismos a bajo nivel.

Finalmente, se procederá a obtener la aplicación de radar por fases: primero obteniendo un mapa radar plenamente operativo, y después aplicándole la tecnología de control por gestos para conseguir la aplicación final. Se tratará de la parte más compleja y extensa, ya que consistirá en juntar los conocimientos obtenidos sobre tecnología de control por gestos, con aquellos púramente aeroespaciales (como, por ejemplo, sobre ADS-B) y con otros sobre sistemas de computadores y programación (al tener que comunicarse con un servidor para obtener datos).

El resultado final consistirá en la aplicación en sí, que se valdrá de recursos como dispositivo *LEAP Motion* ™ y la antena ADS-B instalada en la Escuela para mostrar los tráficos aéreos circundantes, junto con otros ficticios sobre los cuales se podrá operar. Todo ello

controlado íntegramente con el sistema de control por gestos desarrollado. En la figura 11 se muestra un diagrama detallado del sistema completo que implementará la meta de este proyecto.

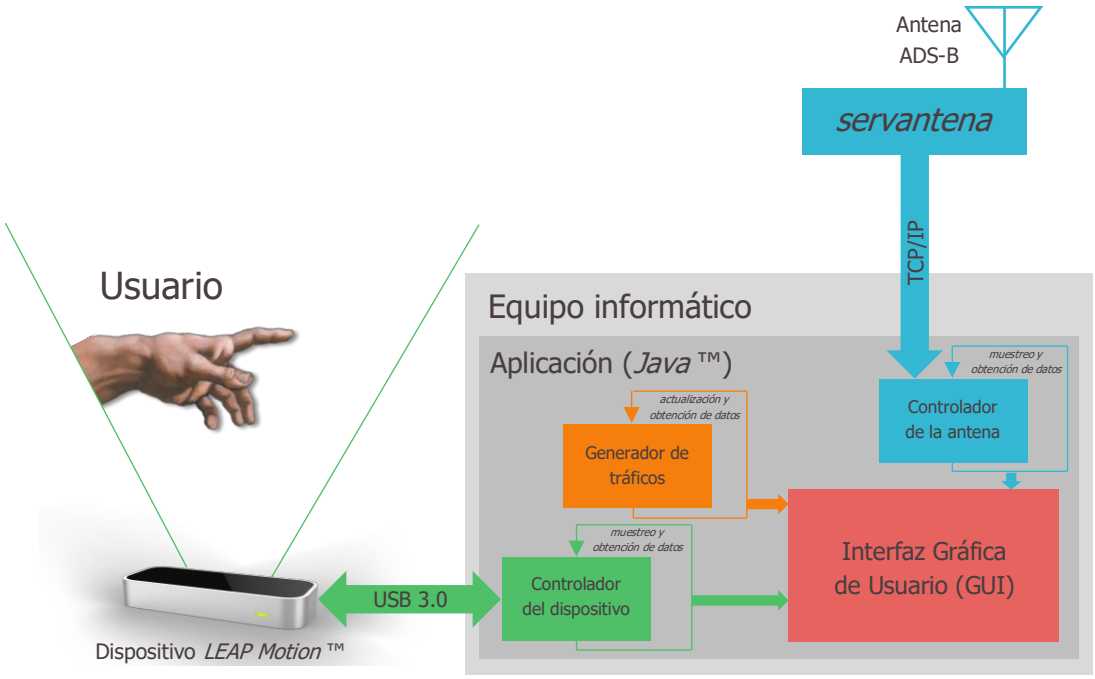


Figura 11: Diagrama de funcionamiento de la aplicación

Y su proceso de implementación podría definirse según el diagrama de *Gantt* mostrado en la figura 12.

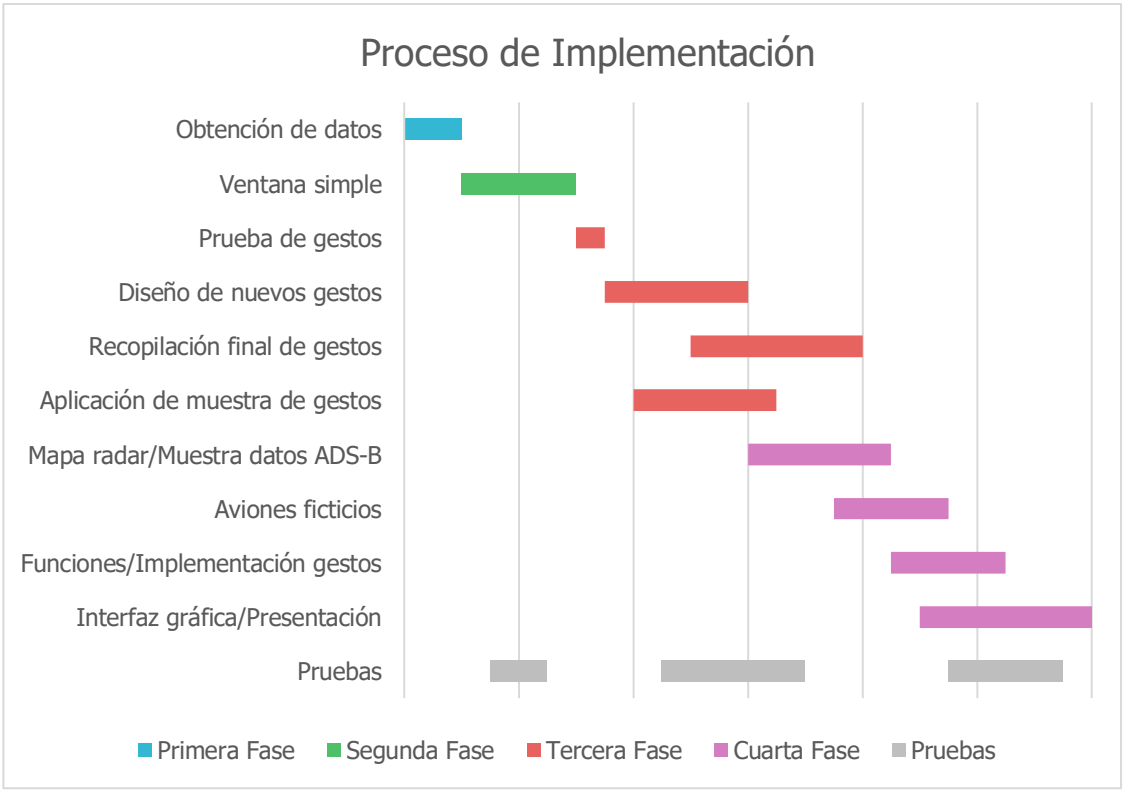


Figura 12: Diagrama de *Gantt* del proceso de implementación

6. PROCESO DE DESARROLLO

Como se ha explicado anteriormente, dada la complejidad que conlleva crear todo un sistema de gestos e implementarlo a una aplicación radar, el proceso de desarrollo se podría dividir en cuatro fases: la puesta en funcionamiento del dispositivo, la obtención de datos del dispositivo, el tratamiento de dichos datos (donde se establecerán los gestos y un sistema para apuntar) y finalmente, el traslado del sistema de control por gestos desarrollado a una aplicación de radar ATM.

6.1. Puesta en funcionamiento: aplicación de muestra

Teniendo en cuenta que el dispositivo *LEAP Motion* ™ es bastante complejo y novedoso, y no existe demasiada bibliografía más allá de la documentación proporcionada por el fabricante, el punto de partida obvio será la aplicación de muestra incluida en el SDK (kit de desarrollo de *software*). Esta simple aplicación extrae multitud de datos del dispositivo y los muestra en consola, y es de especial utilidad a la hora de aprender las distintas funciones de la API (interfaz de programación de aplicaciones) y formas de obtener información.

A nivel interno, la aplicación consta de una única clase que se extiende de la clase *Listener* de *Java* ™. Esta clase cuenta con varios métodos: **`onInit`**, **`onConnect`**, **`onDisconnect`**, **`onExit`** y **`onFrame`**. Los cuatro primeros corresponden, respectivamente, a cuando se inicia la aplicación (que muestra en consola el literal *"Initialized"* y *"Press Enter to quit…"*), cuando se conecta el dispositivo (que muestra en consola el literal *"Connected"*), cuando se desconecta el dispositivo (que muestra en consola el literal *"Disconnected"*) y cuando se cierra la aplicación (que muestra en consola el literal *"Exited"* y se lanza al pulsar la tecla *Enter*). El último de todos, **`onFrame`**, se ejecuta cada vez que se dispone de nuevos datos (cosa que ocurre, aproximadamente, cada 10-40 ms). En este último método donde se extraen los datos y se muestran en consola.

```
@Override
public void onFrame(Controller controller) {
    // Get the most recent frame and report some basic information
    Frame frame = controller.frame();
    System.out.println("Frame id: " + frame.id()
                       + ", timestamp: " + frame.timestamp()
                       + ", hands: " + frame.hands().count()
                       + ", fingers: " + frame.fingers().count()
                       + ", tools: " + frame.tools().count()
                       + ", gestures " + frame.gestures().count());

    //Get hands
    for(Hand hand : frame.hands()) {
        String handType = hand.isLeft() ? "Left hand" : "Right hand";
        System.out.println("  " + handType + ", id: " + hand.id()
                           + ", palm position: " + hand.palmPosition());
```

Figura 13: Fragmento del método **`onFrame`** de la aplicación de muestra

Como se observa en la figura 13, los datos se extraen de la clase **`Frame`**, que contiene todos los datos (incluidos en clases que se extraen de ella, tales como **`Hands`**, **`Fingers`**, **`Gestures`**, etc.) que se han obtenido internamente para el *frame* correspondiente, y se representan valiéndose de la función **`System.out.println`**. En la figura 14 se muestra el inicio de la aplicación y su comportamiento cuando no se introduce ninguna mano; y en la figura 15, los datos mostrados al introducir una mano.

```
run:
Initialized
Press Enter to quit...
Connected
Frame id: 5618, timestamp: 37586437309, hands: 0, fingers: 0, tools: 0, gestures 0
Frame id: 5619, timestamp: 37586529619, hands: 0, fingers: 0, tools: 0, gestures 0
Frame id: 5620, timestamp: 37586575689, hands: 0, fingers: 0, tools: 0, gestures 0
```

Figura 14: Inicio de la aplicación *Sample* por consola

```
Frame id: 5717, timestamp: 37590440871, hands: 1, fingers: 5, tools: 0, gestures 0
  Right hand, id: 4, palm position: (101.214, 127.222, 24.3629)
  pitch: 19.50072625023625 degrees, roll: -30.239129172762787 degrees, yaw: 2.571024503131254 degrees
  Arm direction: (-0.0116503, 0.209656, -0.977706), wrist position: (107.282, 110.337, 72.443), elbow position: (110.233, 57.2344, 320.079)
    TYPE_THUMB, id: 40, length: 48.88958mm, width: 18.99622mm
      TYPE_METACARPAL bone, start: (78.8431, 113.354, 75.643), end: (78.8431, 113.354, 75.643), direction: (0, 0, 0)
      TYPE_PROXIMAL bone, start: (78.8431, 113.354, 75.643), end: (60.901, 120.756, 33.0279), direction: (0.383158, -0.158079, 0.910055)
      TYPE_INTERMEDIATE bone, start: (60.901, 120.756, 33.0279), end: (43.6806, 131.423, 8.27527), direction: (0.538397, -0.333493, 0.773894)
      TYPE_DISTAL bone, start: (43.6806, 131.423, 8.27527), end: (27.9973, 144.242, -0.192359), direction: (0.714353, -0.583904, 0.385689)
    TYPE_INDEX, id: 41, length: 55.16644mm, width: 18.145187mm
      TYPE_METACARPAL bone, start: (94.6797, 128.248, 70.9702), end: (82.5758, 147.6, 5.83961), direction: (0.175382, -0.280406, 0.943723)
      TYPE_PROXIMAL bone, start: (82.5758, 147.6, 5.83961), end: (73.3902, 163.916, -29.8491), direction: (0.227916, -0.404847, 0.885524)
      TYPE_INTERMEDIATE bone, start: (73.3902, 163.916, -29.8491), end: (63.7258, 163.167, -50.3465), direction: (0.426235, 0.0330225, 0.90401)
      TYPE_DISTAL bone, start: (63.7258, 163.167, -50.3465), end: (55.4382, 156.854, -62.5262), direction: (0.517084, 0.393906, 0.759909)
    TYPE_MIDDLE, id: 42, length: 62.857727mm, width: 17.820986mm
      TYPE_METACARPAL bone, start: (105.275, 125.211, 67.5133), end: (101.671, 139.715, 3.79436), direction: (0.0550628, -0.221622, 0.973577)
      TYPE_PROXIMAL bone, start: (101.671, 139.715, 3.79436), end: (103.42, 146.433, -40.8857), direction: (-0.0386803, -0.148564, 0.988146)
      TYPE_INTERMEDIATE bone, start: (103.42, 146.433, -40.8857), end: (97.1079, 140.533, -66.1236), direction: (0.236612, 0.22116, 0.946099)
      TYPE_DISTAL bone, start: (97.1079, 140.533, -66.1236), end: (89.6332, 131.93, -79.5724), direction: (0.424014, 0.48805, 0.762902)
    TYPE_RING, id: 43, length: 60.439384mm, width: 16.957798mm
      TYPE_METACARPAL bone, start: (114.876, 119.414, 64.8603), end: (119.397, 128.189, 6.93364), direction: (-0.0769402, -0.149319, 0.985791)
      TYPE_PROXIMAL bone, start: (119.397, 128.189, 6.93364), end: (126.319, 132.026, -34.2255), direction: (-0.165166, -0.0915496, 0.982008)
      TYPE_INTERMEDIATE bone, start: (126.319, 132.026, -34.2255), end: (123.724, 126.896, -59.5684), direction: (0.0998877, 0.197393, 0.975222)
      TYPE_DISTAL bone, start: (123.724, 126.896, -59.5684), end: (118.38, 119.695, -74.6279), direction: (0.304899, 0.410856, 0.859206)
    TYPE_PINKY, id: 44, length: 47.383358mm, width: 15.063242mm
      TYPE_METACARPAL bone, start: (121.206, 107.757, 63.3467), end: (133.032, 113.905, 10.6099), direction: (-0.217409, -0.113014, 0.969516)
      TYPE_PROXIMAL bone, start: (133.032, 113.905, 10.6099), end: (149.161, 115.236, -18.3439), direction: (-0.486262, -0.0401401, 0.872891)
      TYPE_INTERMEDIATE bone, start: (149.161, 115.236, -18.3439), end: (152.762, 111.734, -35.9907), direction: (-0.196271, 0.190858, 0.961796)
      TYPE_DISTAL bone, start: (152.762, 111.734, -35.9907), end: (151.56, 105.807, -50.9867), direction: (0.0743463, 0.366559, 0.92742)
```

Figura 15: Representación de un *frame* con una mano introducida

Como se puede observar, se reconoce que es la mano derecha y se muestra la posición de su palma, así como multitud de otros valores sobre los dedos, el brazo, la muñeca, etc.

Esta aplicación es importante para familiarizarse con el dispositivo y conocer de forma práctica cómo extraer datos del mismo.

6.2. Obtención de datos: aplicación preliminar

Una vez se ha puesto en funcionamiento el dispositivo y se conoce como obtener datos del mismo, se procederá a trasladarlos a una interfaz gráfica de usuario (GUI) básica, que muestre algunos de los datos más relevantes a modo ilustrativo y familiarizarse con su obtención y tratamiento.

Para la realización de la aplicación se cuenta con el potente diseñador de interfaces y generador de código que ofrece el entorno *NetBeans*, así como con la documentación proporcionada por el fabricante del dispositivo. A nivel interno, se cuenta con dos clases: una que extenderá de la clase *JFrame* y donde se diseñará la propia interfaz, y otra que extenderá de un *Thread* (que es un hilo de ejecución de un programa, pudiendo ejecutarse varios de forma concurrente, como ocurrirá en la aplicación final), y obtendrá periódicamente (concretamente, cada 500 ms) datos del controlador y los representará en la interfaz llamando a métodos del *JFrame* que, a su vez, cambiaran el texto de las etiquetas. Además, se utilizará el método `isConnected()` perteneciente a la clase `Controller` (la que rige el comportamiento del dispositivo y de la que se extrae toda la información necesaria) para comprobar si el dispositivo está conectado, y si no lo estuviera, solamente se mostrará el literal *"Device is not connected"* centrado en la ventana.

Figura 16: Captura de la aplicación *LEAPMotion_Window* cuando el controlador no está conectado

Cuando el controlador está conectado, se muestra la interfaz con los datos más básicos: el número de manos introducidas y el recuento total de dedos (tanto extendidos como retraídos), el número de gestos de librería (de serie son cuatro: *CIRCLE*, *SWIPE*, *SCREEN TAP* y *KEY TAP*) detectados y sus nombres, si la mano introducida es derecha o izquierda (iluminando un indicador), los ángulos que forma la mano y la posición de la palma.

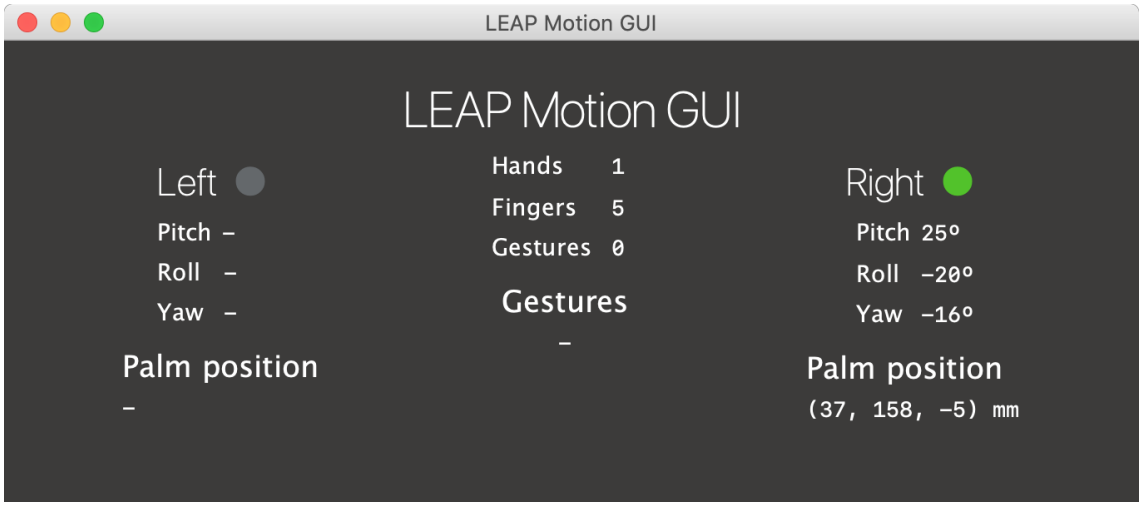


Figura 17: Captura de la aplicación *LEAPMotion_Window* cuando hay una mano insertada

A izquierda y derecha se agruparán los datos individuales de cada mano (ambas podrán ser introducidas simultáneamente) y cuando no haya ninguna mano insertada, se mostrará el indicador en color gris y el literal "-" en el lugar de los valores.

Cabe destacar que hay multitud de datos que se podrían representar de igual forma, pero como el objeto de este punto es simplemente extraerlos y mostrarlos, sin importar cuáles sean, se han elegido los que se pueden observar de forma más notable, para comprobar de forma práctica el funcionamiento del dispositivo.

Figura 18: Captura de la aplicación *LEAPMotion_Window* cuando las dos manos están insertadas y se detecta el gesto *CIRCLE*

6.3. Tratamiento de datos: aplicación con sistema de apuntamiento y detección de gestos

Tras obtener la gran cantidad de datos e información que proporciona el controlador, surge una pregunta: ¿qué se puede hacer con dicha información? Por tanto, el siguiente paso en el desarrollo de la aplicación será obtener formas de tratar los datos para reconocer gestos y tener un sistema fiable para controlar una interfaz gráfica.

Dada la naturaleza de la aplicación final, donde se mostrarán diversos tráficos aéreos y se deberá operar sobre ellos, es imprescindible un sistema fiable de apuntamiento y selección, que nos permita poder elegir uno de los tráficos de forma precisa. En primer lugar, se barajó la posibilidad de utilizar la posición de la punta del dedo índice (obtenida a partir de la clase `Finger`) y a partir de ella, utilizar los valores de *X* y de *Y* para trasladarlos a la posición en dos dimensiones en la pantalla (ya que el plano *XY* del dispositivo es paralelo a la pantalla, véase figura 19). La posición en *Z* determinaría si se está pulsando o no, al traspasar un determinado valor.

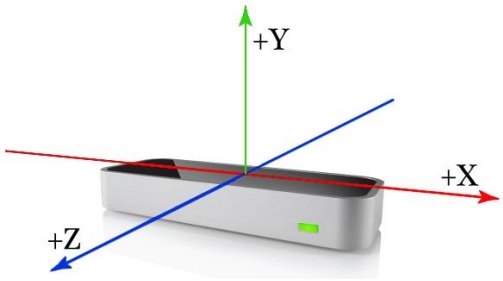


Figura 19: Sistema de referencia del dispositivo *LEAP Motion*™

Sin embargo, se pudo comprobar que, a nivel de usuario, era bastante incómodo, ya que no se estaba señalando, si no moviendo toda la mano. Además, la falta de respuesta por parte de la aplicación respecto a si se estaba cerca o no del umbral de pulsado en *Z* hacía que no fuera nada intuitiva, pulsándose sin querer numerosas veces. Todo esto, sumado a la inestabilidad que

daba el dispositivo, hacía que se tratara de un sistema muy poco eficiente; por tanto, se decidió innovar y rehacerlo.

Para empezar, se debería proporcionar una respuesta por parte de la interfaz sobre la posición del dedo en el eje *Z*, para que el pulsado sea más cómodo e intuitivo, lo cual se resuelve con aumentar el tamaño del cursor al alejarse y disminuirlo al acercarse. Y respecto al sistema de apuntado, se decide introducir un nuevo elemento a tener en cuenta además de la posición del dedo: su dirección.

De forma paralela, el antiguo *Thread* de la aplicación básica anterior, donde en cada recorrido se escribían todos los datos, decide mejorarse y crear una especie de "máquina de estados"; es decir, en cada recorrido (cuya frecuencia ahora es de 20 ms para asegurar una mayor fluidez de la aplicación) primero se obtienen unos parámetros muy básicos para determinar si el usuario se encuentra señalando, haciendo un gesto, etc. y después se actúa de acuerdo al estado en el que se encuentre el *Thread*. Si hay solamente un dedo extendido, se considera que el usuario está señalando, si hay varios y la mano se encuentra vertical, se considera que se está realizando un *swipe* (desplazamiento lateral), y si se encuentra horizontal, se reserva otro gesto similar al *swipe* pero en horizontal, al que se le podrán asignar funciones en el futuro como, por ejemplo, cambiar valores.

Cuando se está señalando, el *Thread* llama al método `trackFinger` de la clase principal de la aplicación (la que extiende de *JFrame*, como ocurría en la aplicación anterior). Dicho método, recibe la posición de la punta del dedo y su dirección, y valiéndose de una transformación trigonométrica y utilizando parámetros como el tamaño de pantalla y la distancia del controlador a la pantalla, obtiene el píxel de la pantalla al que se está apuntando, y lo traslada a la clase *CPaint* para representar el cursor.

```
public void trackFinger(Vector dir, Vector pos) {
    double x, y, squareSize;
    boolean pressed = false;

    x = canvasPanel.getSize().getWidth() * (11.069 * SCREEN_SIZE + pos.getX()
            - (CONTROLLER_POS + pos.getZ()) * dir.getX() / dir.getZ()) / (22.138 * SCREEN_SIZE);
    y = canvasPanel.getSize().getHeight() * (1 - (pos.getY() - (CONTROLLER_POS
            + pos.getZ()) * dir.getY() / dir.getZ()) / (12.4526 * SCREEN_SIZE));
    squareSize = (20 + pos.getZ() / 4) * (canvasPanel.getSize().getWidth() / (22.138 * SCREEN_SIZE)) / 4.3473;

    if (squareSize <= 10) {
        squareSize = 10;
        pressed = true;
    } else if (squareSize > 30) {
        squareSize = 30;
    }

    if (x < 0) {
        x = 0;
    } else if (x > canvasPanel.getSize().getWidth() - (int) squareSize) {
        x = canvasPanel.getSize().getWidth() - (int) squareSize;
    }

    if (y < 0) {
        y = 0;
    } else if (y > canvasPanel.getSize().getHeight() - (int) squareSize) {
        y = canvasPanel.getSize().getHeight() - (int) squareSize;
    }

    ((CPaint) canvasPanel).dispCursor((int) x, (int) y, (int) squareSize, pressed);
    ((CPaint) canvasPanel).repaint();
}
```

Figura 20: Fragmento del código con el método `trackFinger`

Para obtener el sistema de apuntamiento, obtendremos primero la coordenada *X* del punto al que se está señalando. Para ello, será necesario conocer la dimensión horizontal de la pantalla y la distancia del controlador a la misma, ambas en milímetros.

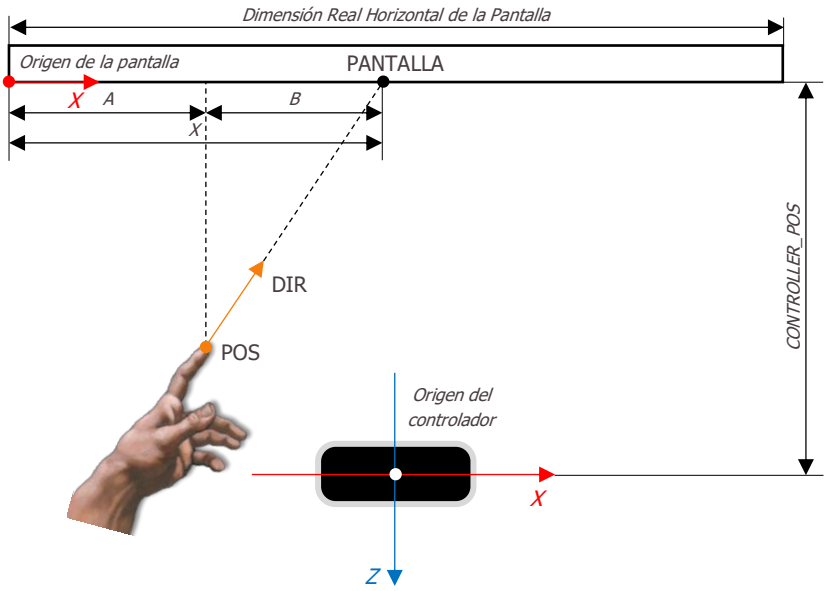
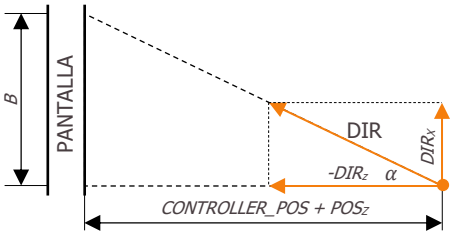


Figura 21: Esquema para la obtención de la coordenada *X* de la pantalla a la que se está señalando

Como se observa en la figura 21, la coordenada *X* se podrá obtener fácilmente sumando los parámetros *A* y *B*. El primero se obtendrá de forma inmediata sumando la coordenada *X* de la posición del dedo (recordemos que en el esquema sería negativa) a la mitad de la dimensión real horizontal de la pantalla; y el segundo se podrá obtener tal y como se muestra en la figura 22 (recordemos, una vez más, que en el caso particular que se muestra en las figuras, el valor *POS_Z* será negativo).



$$\frac{B}{CONTROLLER\_POS + POS_Z} = \tan\alpha$$

$$\tan\alpha = \frac{DIR_X}{-DIR_Z}$$

$$B = -\frac{DIR_X}{DIR_Z} \cdot (CONTROLLER\_POS + POS_Z)$$

Figura 22: Cálculo del valor *B* para la coordenada horizontal de la pantalla

Finalmente, las dimensiones reales de la pantalla se calcularán a partir del tamaño de pantalla en pulgadas introducido en los ajustes de la futura aplicación (y con la premisa de que el aspecto será de 16:9, estándar actual para monitores) de la forma descrita en la figura 23.

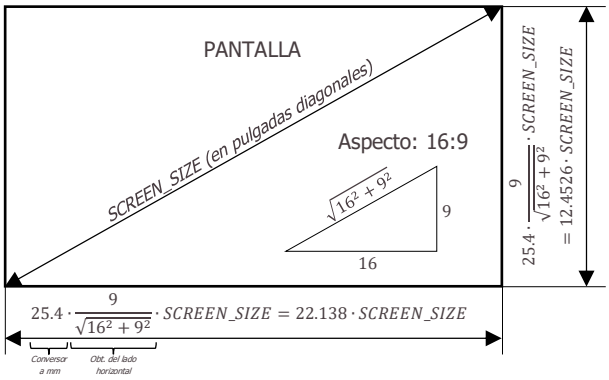


Figura 23: Cálculo de las dimensiones reales de la pantalla a partir de su tamaño en pulgadas diagonales

Por tanto, la coordenada X sobre la pantalla a la que se estará señalando será:

$$X = 22.138 \cdot SCREEN\_SIZE/2 + POS_X - \frac{DIR_X}{DIR_Z} \cdot (CONTROLLER\_POS + POS_Z)$$

Teniendo en cuenta que el valor de *X* obtenido sumando *A* y *B* estará en milímetros, podremos obtenerlo en píxeles de forma inmediata sabiendo que el tamaño horizontal de la pantalla en milímetros tendrá el número de píxeles determinado por el método **`getWidth()`** del panel donde se esté apuntando (dado que la aplicación se ejecutará a pantalla completa). Por tanto, el valor descrito en la expresión anterior deberá ser multiplicado por el ancho en píxeles y dividido por la dimensión horizontal real de la pantalla (ver expresión en el fragmento de código de la figura 20).

El valor de *Y* se obtendrá de forma similar, siguiendo el esquema de la figura 24.

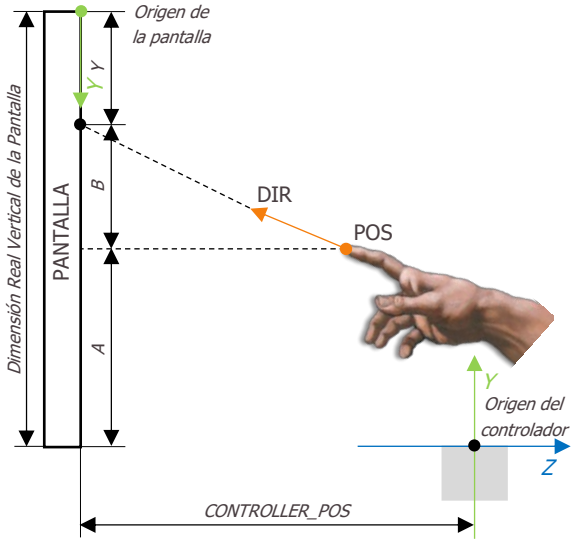


Figura 24: Esquema para la obtención de la coordenada Y de la pantalla a la que se está señalando

El valor *A* se obtendrá de forma inmediata, ya que será la componente *Y* de la posición de la punta del dedo, que será siempre positiva. El valor de *B*, sin embargo, se deberá obtener de la forma descrita en la figura 25.

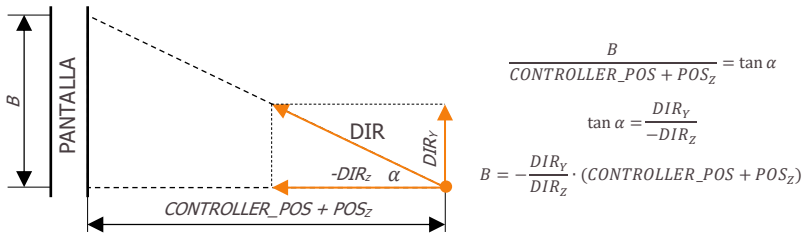


Figura 25: Cálculo del valor B para la coordenada vertical de la pantalla

Por tanto, la coordenada Y sobre la pantalla a la que se estará señalando será:

$$Y = 12.4526 \cdot SCREENSIZE - (POS_Y - \frac{DIR_Y}{DIR_Z} \cdot (CONTROLLER_{POS} + POS_Z))$$

De forma análoga a con la coordenada horizontal, esta magnitud real se podrá trasladar a píxeles multiplicándola por el alto de píxeles del panel (utilizando el método `getHeight()`) y dividiéndola entre la magnitud real vertical de la pantalla descrita en la figura 23, obteniendo así la expresión dada en el fragmento de código de la figura 20.

Para el tamaño del cursor (definido en el fragmento de código de la figura 20 como `squareSize`), se obtendrá empíricamente probando distintos tamaños que dependerán de la coordenada *Z* de la posición del dedo, y se hará independiente de la resolución de la pantalla (en pro de hacer que la aplicación sea completamente multiplataforma) haciéndolo depender de la relación píxeles-magnitud real utilizada anteriormente. También será necesario introducir varias condiciones para limitar el rango de coordenadas y conseguir que si se apunta fuera de la pantalla, se muestre el cursor en el lado más cercano al lugar donde se está apuntando (cosa que se consigue simplemente haciendo que, si la coordenada excede el rango o es negativa, se considere el máximo correspondiente o 0, respectivamente, como se realiza en el fragmento de código de la figura 20).

Con todo esto, se obtiene un sistema de apuntamiento muy intuitivo y estable, que permitirá seleccionar los aviones de forma precisa, pero aún será necesario desarrollar dicho sistema de selección. Dada la naturaleza crítica de las aplicaciones ATM, es necesario que el sistema de selección sea igualmente estable y fiable, y tras comprobar empíricamente que el sistema de pulsar para seleccionar no es nada estable, se decide encontrar uno más fiable e innovador.

Este nuevo sistema consistirá en que, cuando se "pulsa" (es decir, se rebasa cierto umbral en la coordenada *Z* de la posición de la punta del dedo) el cursor se vuelve naranja, y se empiezan a escribir las posiciones en un *array* de puntos. Al mismo tiempo, se estará trazando una polilínea con dichos puntos, de forma que el usuario tendrá la respuesta visual de que está trazando una línea curva irregular a medida que va apuntando. Cuando esa polilínea se cierre (es decir, en tiempo real se estará comprobando si alguno de los distintos segmentos infinitesimales que

componen la polilínea se cruza con otro), el cursor se pondrá en verde (indicando que la selección se ha completeado) y se considerará el polígono cerrado compuesto por la multitud de segmentos que componían la polilínea, y se seleccionará lo que se encuentre dentro de dicho polígono. Desde el punto de vista del usuario, al pulsar se empezará a trazar una línea, que cuando se cierre (a modo de "lazo") seleccionará lo que se encuentre dentro de ella (véase figura 26).

Figura 26: Sistema de selección por lazo

Con esto conseguimos mejorar notablemente la precisión, ya que al "pulsar" es muy complicado variar la posición en *Z* del dedo manteniendo estáticos el resto de parámetros, y en la práctica, nunca se seleccionaba lo que se quería. También cabe destacar que, dada la congestión habitual del espacio aéreo, los tráficos se encontrarán muy juntos y será posible que dentro del "lazo" sean incluidos varios; pero este problema será resuelto con un simple selector entre los tráficos que se incluyan dentro del "lazo", que será desarrollado en la aplicación final.

Respecto a los gestos, aquellos que el fabricante incluye en la librería no tenían un funcionamiento adecuado (además de que no podían ser controlados a muy bajo nivel, dado que las librerías y funciones proporcionadas por el fabricante están compiladas y no se pueden modificar). Por tanto, se decidió crear gestos propios, que consistirán, básicamente, en movimientos de la mano con la palma abierta; tanto en horizontal (*swipe*), como en vertical (provisionalmente llamado *palm*).

Los gestos *swipe* podrán ser utilizados en la aplicación final para extraer, por ejemplo, un menú lateral; y los *palm*, para selecciones y cambio de valores, subiendo y bajando la mano para cambiar valores o desplazándola lateralmente para seleccionar entre distintas opciones mostradas en pantalla. Al igual que en el sistema anterior de selección se confirmaba la misma cerrando el "lazo", será necesario un sistema de confirmación del valor u opción elegidos; que, por ejemplo, podría ser girar la mano, cosa que es bastante intuitiva.

Sin embargo, dado que dichos gestos deben ser pulidos y perfeccionados sobre la interfaz final (ya que dependen de circunstancias concretas como cierto valor a cambiar, una serie de opciones que se deben mostrar en pantalla, etc), en esta aplicación se dejarán preparados y se mostrará visualmente, a nivel de prueba, el gesto *swipe* (que se utilizará para cambiar el color del fondo de la pantalla donde se está probando el sistema anterior de selección). La respuesta visual de dicho gesto será mostrar una barra vertical blanca con la posición cuando se inició el gesto y una azul con la posición actual (cogiendo simplemente la coordenada *X* actual de la palma de la mano y convirtiéndola a píxeles con una sencilla regla de tres, ya que, al ser meramente demostrativo, no se requiere un elaborado sistema como para el apuntamiento).

Figura 27: Respuesta visual del gesto *swipe*

Al rebasar un umbral (que en esta aplicación será arbitrario) cambiará el color del fondo a blanco (desplazando a la izquierda) o negro (desplazando a la derecha).

6.4. Aplicación final de radar

Una vez está preparado el avanzado sistema de apuntamiento y definidos los gestos básicos, se puede proceder al desarrollo de la aplicación final. En este punto se procederá a proporcionar una descripción funcional de la aplicación en términos generales, centrándose en aspectos básicos como su diseño y características; mientras que en el punto 7 (*Estructura interna de la aplicación*) se proporcionará una visión detallada y pormenorizada del código, y en el apéndice, una descripción detallada de su funcionalidad desde el punto de vista del usuario. Todo ello dará una idea completa de la aplicación y su funcionamiento.

6.4.1. Diseño general de la aplicación

La idea principal para el presente proyecto será conseguir una aplicación con una interfaz que no solamente sea funcional, si no también amigable y agradable de utilizar, ya que es el único canal de comunicación entre el usuario y la propia aplicación. Por tanto, su diseño a nivel gráfico será un punto crucial en el desarrollo de la misma y será cuidado hasta el más mínimo detalle.

Siguiendo las tendencias actuales en el desarrollo de aplicaciones y sistemas operativos, se utilizará una interfaz con paneles translúcidos superpuestos sobre un fondo que siempre será el mapa radar, y será intuible a través de dichos paneles. Habrá dos principales: uno lateral que se extraerá con un *swipe* con los modos y ajustes de la aplicación, y uno que saldrá del lado inferior cuando se necesiten mostrar datos. Todo lo demás se encontrará flotando sobre el mapa (elementos como un reloj en la parte superior derecha, los indicadores de estado del servidor y el dispositivo en la parte inferior derecha, o mensajes puntuales sobre la aplicación y su manejo en la parte inferior central, que ayudarán al usuario).

El mapa será en tonos grises y azules oscuros, facilitando la visibilidad de los tráficos sobre el mismo, pero sin renunciar al diseño (ya que incluso en las aplicaciones más avanzadas, solo se muestra el contorno de la costa y fronteras sobre fondo negro). Se obtendrá de *Google Maps* ©, utilizando una herramienta proporcionada por *Google* ® para modificar los colores del mapa, y una vez conseguido, se guardará en formato imagen (ya que solo será necesaria el área donde la antena recibe datos, no será necesario un mapa que se pueda mover).

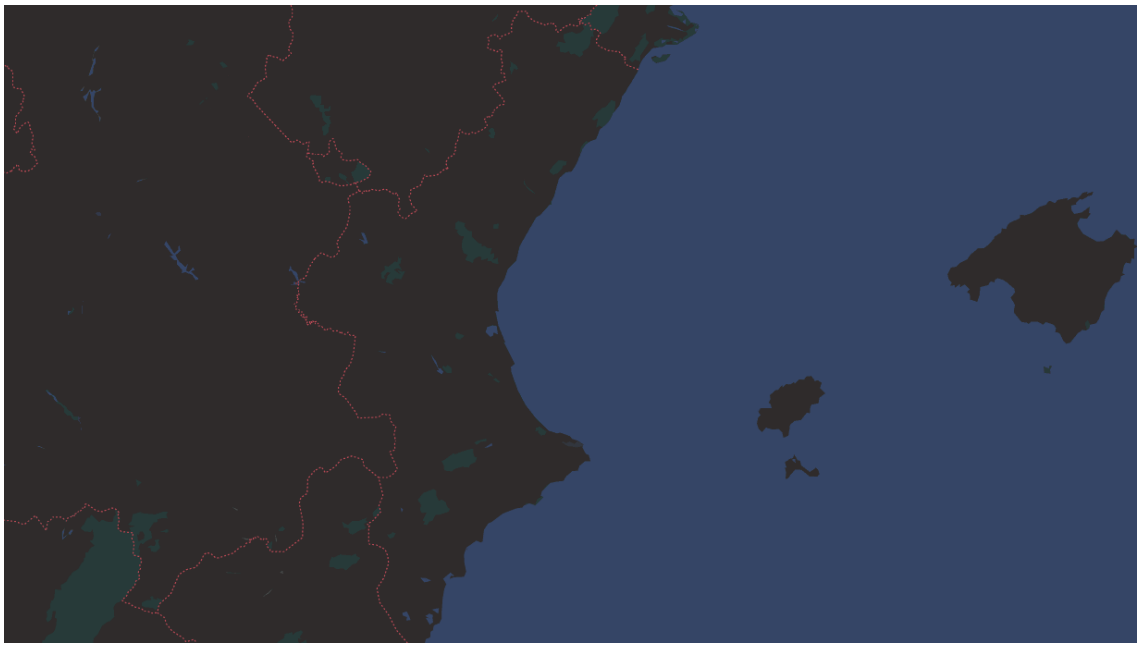


Figura 28: Mapa utilizado en la aplicación

Respecto a las fuentes utilizadas, serán la *San Francisco Pro Display Ultrathin* para títulos y letreros, la *San Francisco Pro Text Regular* para textos y etiquetas, y la *San Francisco Mono Medium* para valores numéricos. Todas ellas son fuentes de los sistemas operativos y aplicaciones desarrollados por *Apple* ®, y están disponibles para su libre uso en la plataforma *Apple Developer* ©[10]. Siguiendo la idea de conseguir una aplicación multiplataforma, serán incluidas en la aplicación como recursos, para que la experiencia del usuario sea la misma independientemente del equipo que utilice.

Los iconos de los tráficos aéreos serán una imagen del icono de avión de la fuente *San Francisco*, editada en *Adobe Photoshop* © para conseguir los seis colores que serán necesarios: amarillo como color básico, verde para indicar selección, morado para selección cuando se calculen distancias, rojo para cuando se encuentre en emergencia, naranja para alerta y azul para cuando se encuentre en tierra. Cuando haya riesgo de colisión, se alternará entre amarillo y rojo en intervalos de 1000 ms.

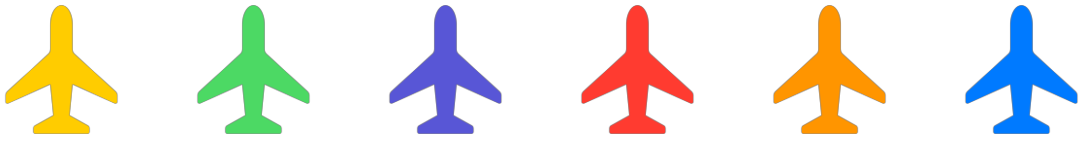


Figura 29: Iconos de los tráficos (de izquierda a derecha) normal, selección, cálculo de distancias, emergencia, alerta y en tierra

Con todo lo anteriormente explicado, se tratará de conseguir una interfaz tan funcional como agradable de utilizar, cuidando los detalles y aprovechando multitud de recursos gratuitos que ofrecen distintas compañías para ayudar a la comunidad de desarrolladores.

[10] Disponible en "Fonts for Apple Platforms - Apple Developer".

6.4.2. Características y funciones principales de la aplicación

A nivel funcional, la aplicación contará con dos modos: Radar y Entrenar. El primero se valdrá de los datos recibidos de *servantena* para representar los tráficos circundantes y obtener información de ellos; mientras que el segundo creará tráficos sintéticos de forma periódica y permitirá operar sobre ellos (modificar sus trayectorias, altitudes y velocidades), con ayudas como las alertas de riesgo de colisión. Habrá una tercera opción que valdrá para ambos modos, Distancias, que permitirá seleccionar dos tráficos y mostrará la distancia entre ambos.

La aplicación siempre se iniciará en el modo Radar, ya que el modo Entrenar requerirá la completa concentración del operador, y el primero de ambos será más relevante a nivel demostrativo (ya que mostrará tráficos reales).

Sobre el mapa de la figura 28 se mostrarán los tráficos circundantes junto con su *callsign*. Además, se mostrará un reloj en la parte superior derecha con la hora UTC (Tiempo Universal Coordinado), también llamada hora *Zulu*, utilizada en el ámbito ATM. En la parte inferior derecha se mostrará el estado del servidor y del dispositivo (verde si están conectados, rojo si están desconectados).

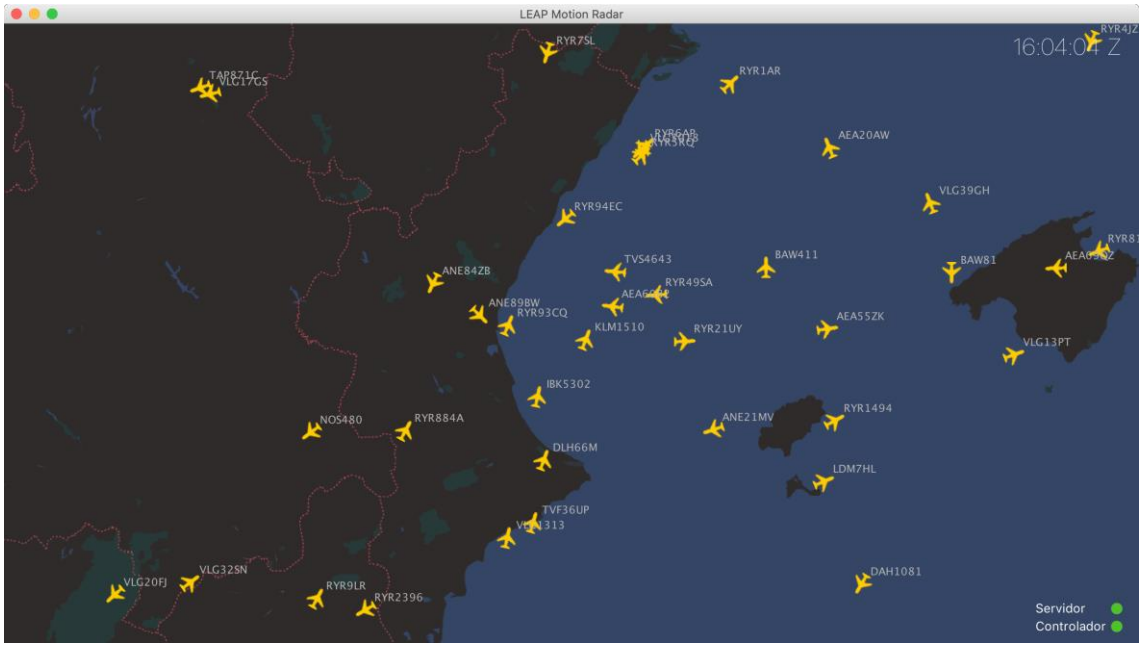


Figura 30: Vista inicial de la aplicación

El usuario podrá entonces seleccionar los tráficos que desee para obtener información sobre ellos, utilizando el sistema de selección por "lazo" desarrollado para este proyecto. Al haber multitud de tráficos, si se seleccionaran varios, se mostrarían en un selector, para obtener el deseado; hasta un máximo de cinco tráficos. Si se seleccionaran más de cinco (cosa que es improbable a no ser que haya algún despiste), se mostrará el mensaje "Demasiados tráficos seleccionados, por favor seleccione de 1 a 5 tráficos".

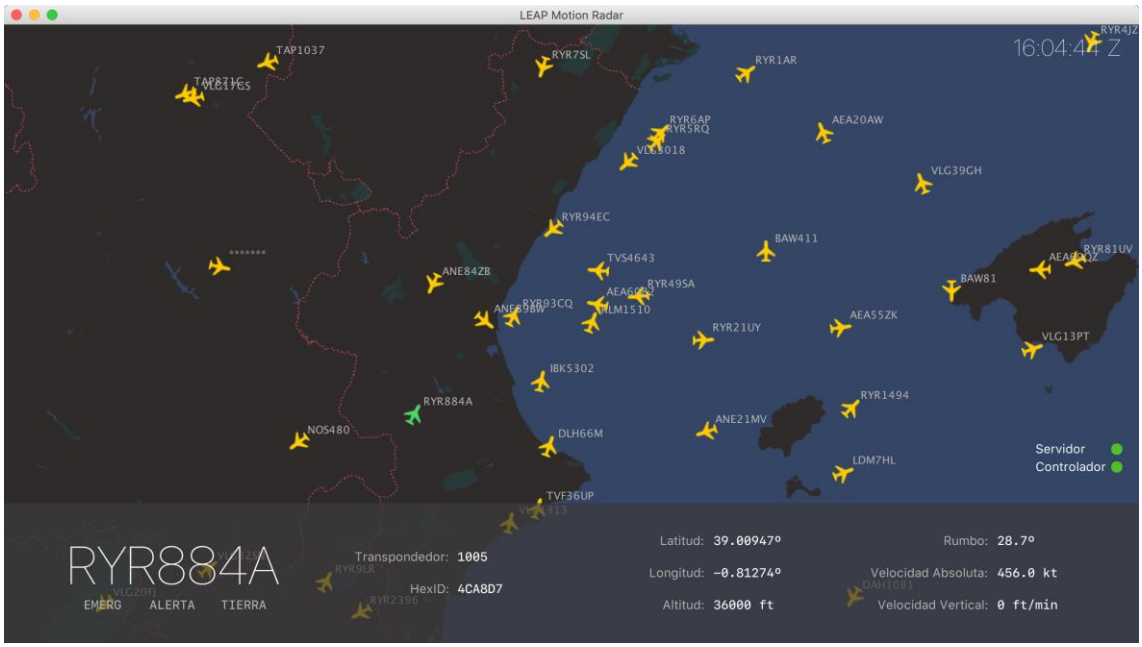


Figura 31: Vista de la aplicación con un tráfico seleccionado

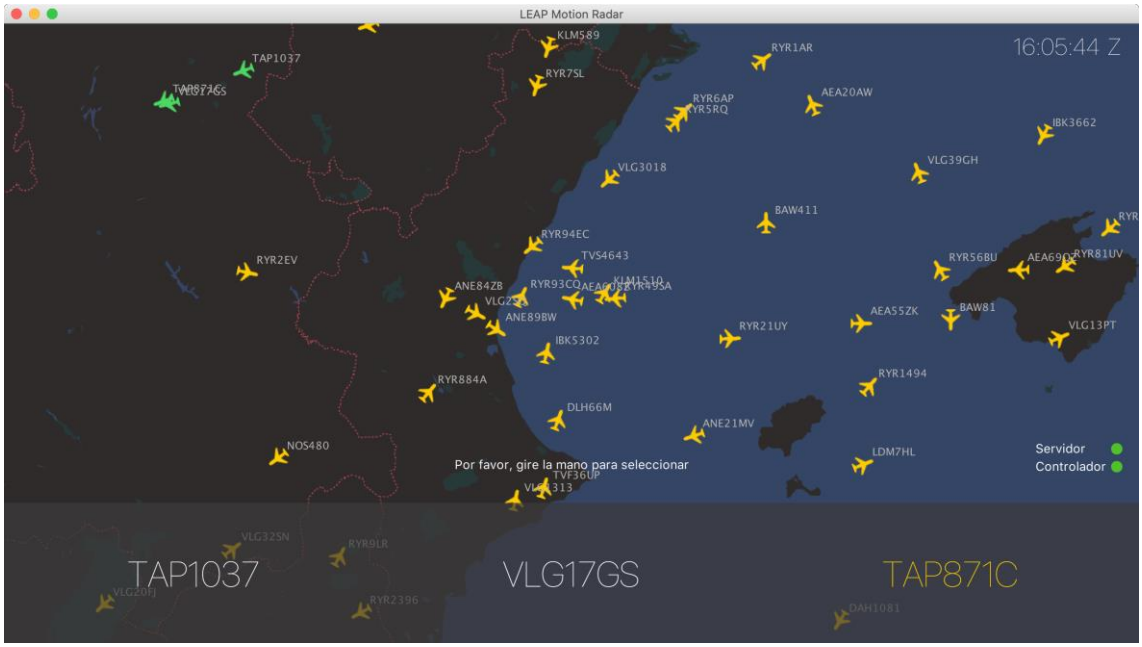


Figura 32: Vista de la aplicación mostrando el selector para tres tráficos

Como se muestra en la figura 31, para el tráfico seleccionado se mostrará si se encuentra en emergencia, alerta o en tierra, su código transpondedor, su identificador hexadecimal, su ubicación (latitud, longitud y altitud) y parámetros sobre su movimiento (rumbo o *track*, velocidad absoluta relativa al suelo, y velocidad vertical).

Con el panel fuera, se podrá seleccionar otro tráfico o esconderlo desplazando la mano hacia abajo con la palma abierta, que proporcionará un efecto de estar "hundiendo" el propio panel.

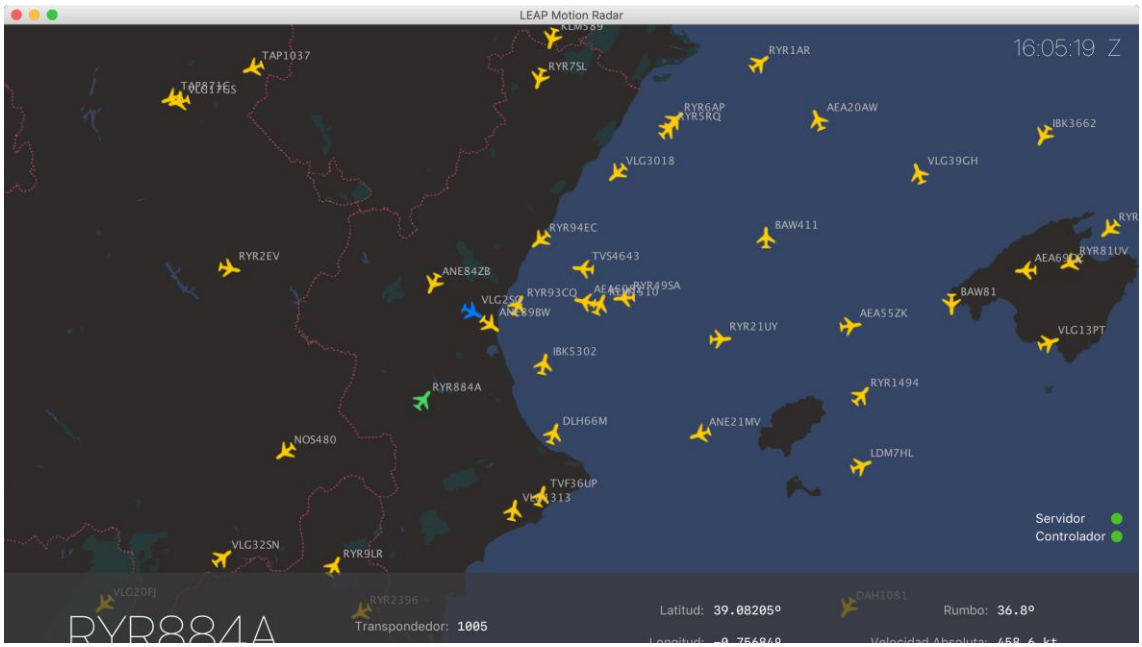


Figura 33: Panel de información siendo ocultado por el usuario

Al hacer el gesto *swipe* hacia la izquierda, se mostrará el panel lateral con los modos y ajustes (haciendo el mismo efecto de estar desplazándose desde el borde derecho hasta aparecer completamente). Este panel mostrará seis opciones: la primera seleccionará el modo Radar, la segunda el modo Entrenar, la tercera entrará en el asistente de cálculo de distancias y las otras tres serán relativas al punto 6.4.3 (*Ajustes y calibración*).

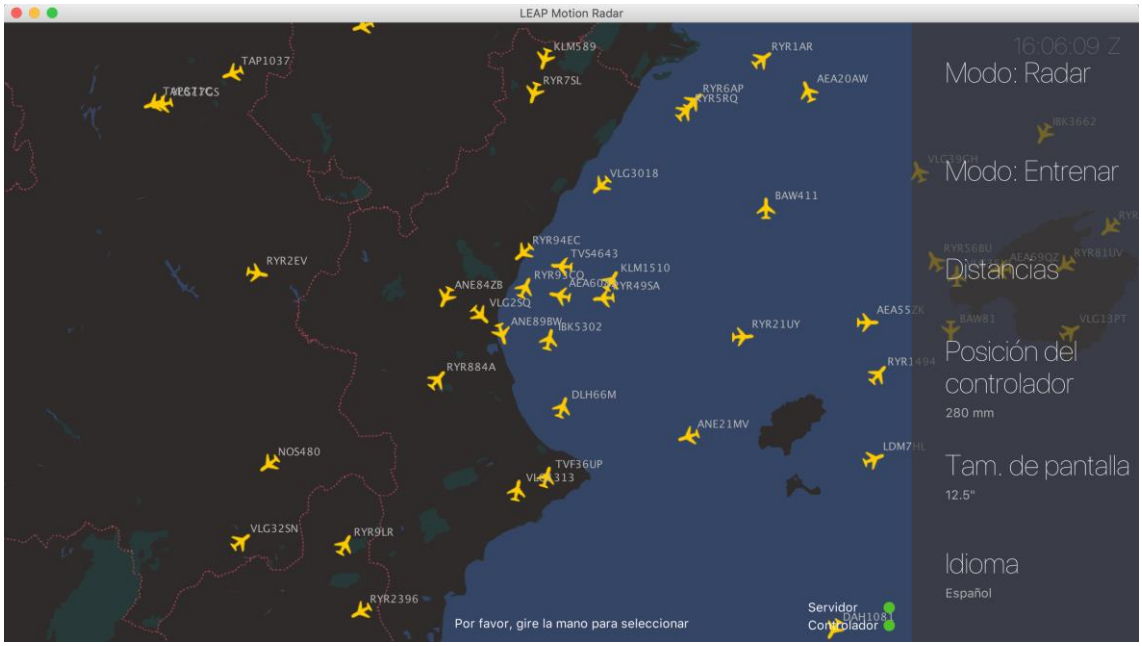


Figura 34: Vista de la aplicación con el panel lateral mostrándose

Para seleccionar entre las distintas opciones, bastará con desplazar la mano verticalmente hasta llegar a la deseada, y girar la mano hasta que la palma esté hacia arriba para confirmar la selección. Como se observa en la figura 34, cuando es necesario en la aplicación, se muestran

instrucciones en la parte inferior central, para que la aplicación sea completamente intuitiva y accesible a cualquier usuario.

Al entrar en el modo Entrenar se mostrará un panel para seleccionar la dificultad de 1 a 5 (a mayor dificultad, menor periodo entre apariciones de tráficos sintéticos, como se muestra en la figura 35).

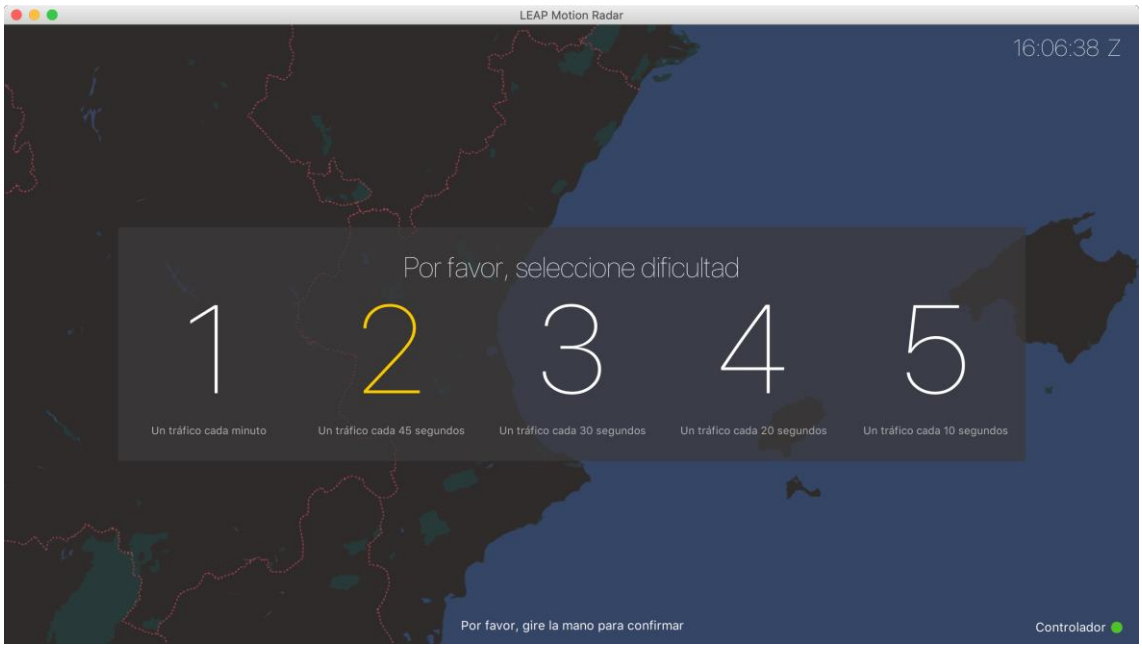


Figura 35: Vista del selector de dificultad

Al haber seleccionado el modo en el panel lateral, la mano del usuario estará hacia arriba, y por tanto para confirmar la dificultad se deberá girar hacia abajo. En toda la aplicación, el modo de selección será el mismo: girar la mano hacia arriba si se tiene hacia abajo y viceversa; y a medida que se gire la mano, el color de la opción o valor que se está seleccionando pasará de amarillo a verde (o rojo si es una opción peligrosa, como salir del modo Entrenar, con lo que no se podría volver al punto de la simulación en el que se está).

Una vez seleccionada la dificultad, se empezarán a mostrar tráficos en el intervalo de tiempo correspondiente. Además, se mostrará un mensaje en la parte inferior central confirmando la dificultad seleccionada ("Nivel X de dificultad seleccionado"). Dichos tráficos se mostrarán de forma idéntica a los reales, pero con el *callsign* de estilo "TRFXXXX"

Si se vuelve a sacar el panel lateral y se selecciona el modo radar, se preguntará si se desea salir; y si se selecciona el modo entrenar, si se desea reiniciar (excepto si se viene del modo Distancias, donde se considerará que simplemente se desea volver al selector de tráficos y no reiniciar la simulación).

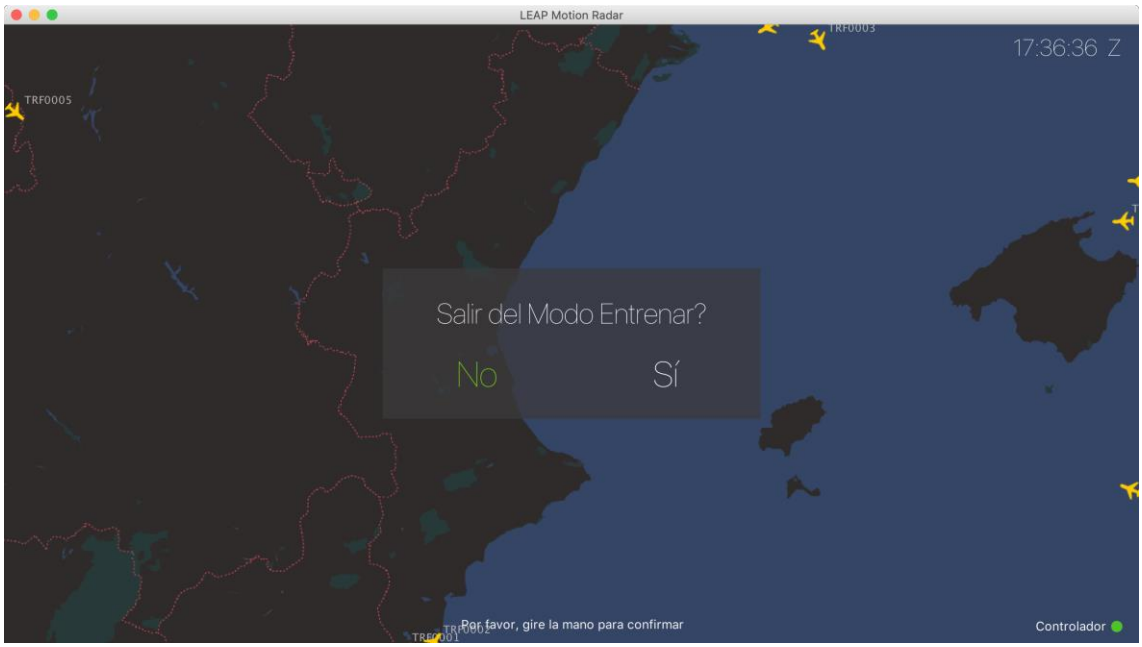


Figura 36: Mensaje de confirmación de salida del modo Entrenar

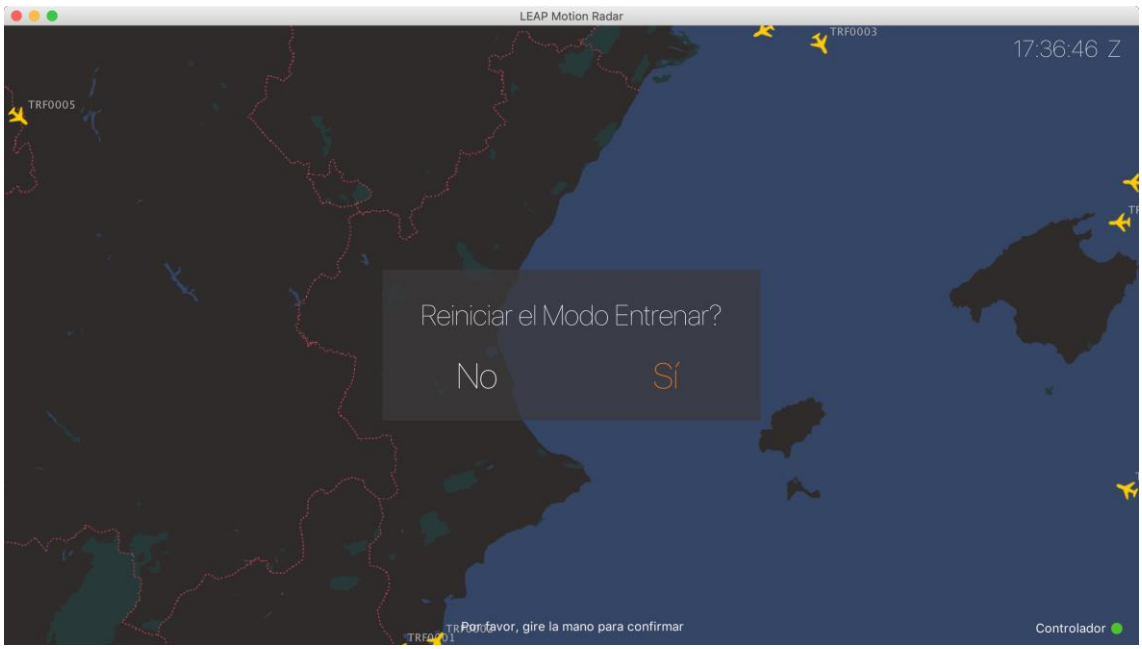


Figura 37: Mensaje de confirmación de reinicio del modo Entrenar

Al igual que con los tráficos reales, se podrá obtener información de un determinado vuelo seleccionándolo; con la novedad de que cuando esté el panel fuera, se podrá seleccionar uno de los tres parámetros que rigen su movimiento (rumbo, velocidad absoluta y altitud) y modificarlo desplazando la mano verticalmente (y girándola para confirmar).

Los tráficos sintéticos se desplazarán de un punto aleatorio de los lados de la pantalla, a otro punto aleatorio de otro lado aleatorio, y tendrán una velocidad aleatoria desde los 350 a los 500 nudos y una altitud aleatoria de los 30000 a los 45000 pies de altitud. También tendrán un 1% de probabilidades de aparecer en estado de emergencia.

A tiempo real se estará comprobando si hay riesgo de colisión entre alguno de los tráficos (basándose en sus trayectorias y velocidades y calculando la distancia mínima a la que se encontrarán), y si lo hubiera, se mostrarán parpadeando en color rojo y aparecerá un mensaje indicando los tráficos en peligro y el tiempo estimado para su colisión.

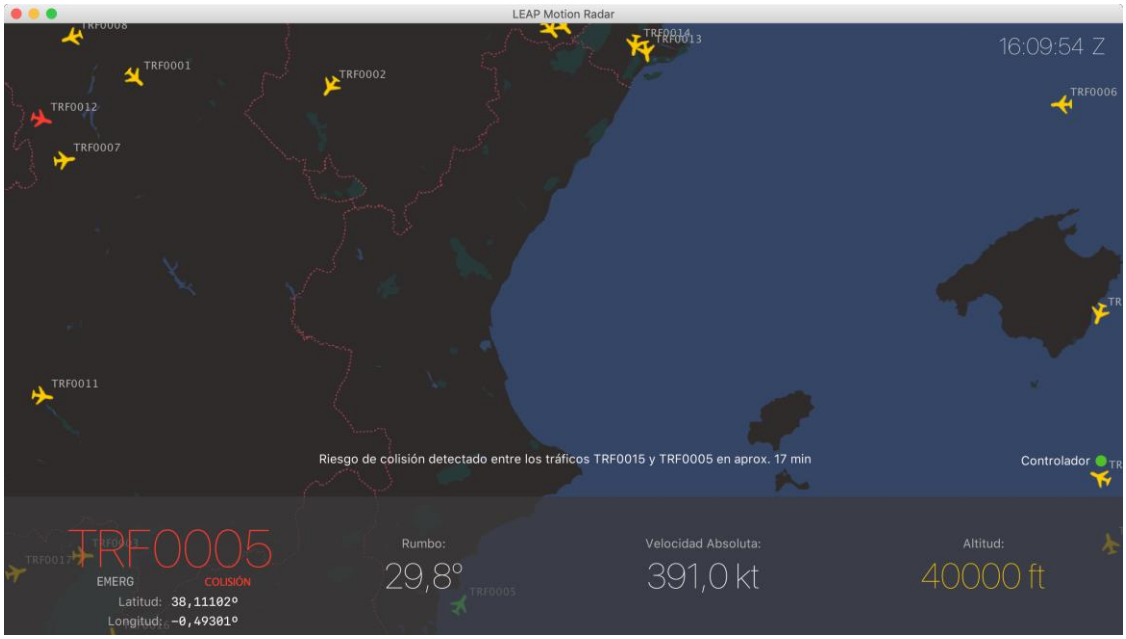


Figura 38: Vista del panel de información de un tráfico sintético en riesgo de colisión, con el usuario cambiando la altitud del mismo

Al entrar en el asistente de cálculo de distancias (mediante el panel lateral de opciones para cualquiera de los modos) se pedirá al usuario que seleccione el primer tráfico. Una vez elegido (con el método de selección por "lazo" y el selector si hubiera varios), se marcará el tráfico seleccionado en morado, se trazará una línea morada que unirá el cursor y el tráfico en cuestión, y se pedirá que se seleccione el segundo, tal y como se muestra en la figura 39.

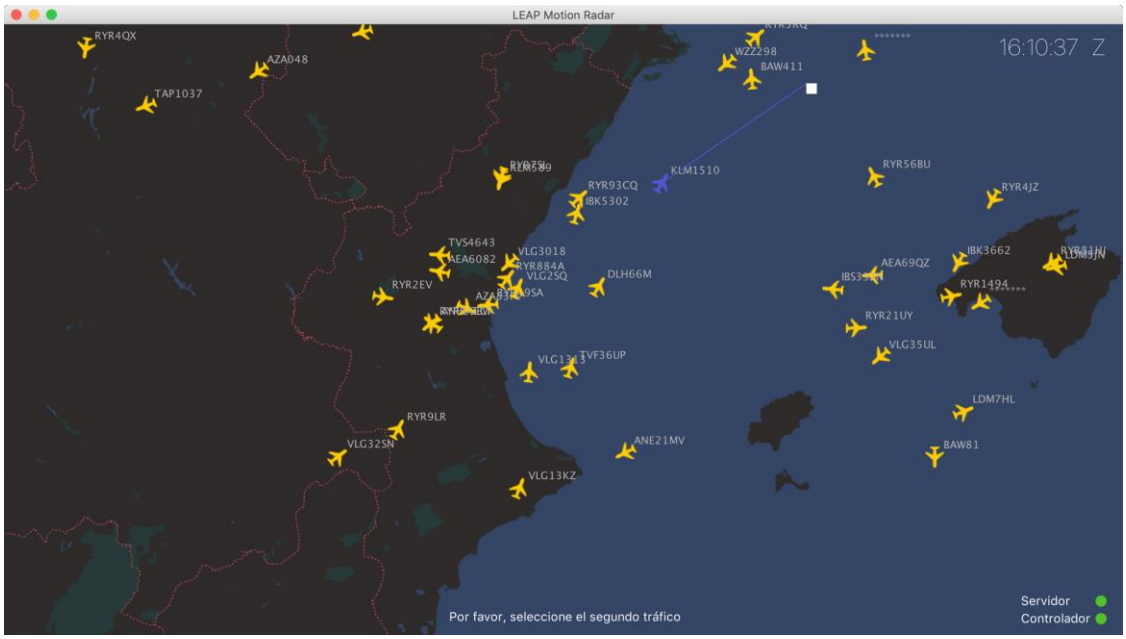


Figura 39: Vista de la aplicación con el asistente de cálculo de distancias

Una vez seleccionado el segundo, la línea morada unirá ambos tráficos (representando su distancia) y se mostrará un panel con las posiciones de ambos tráficos y su separación en kilómetros y millas náuticas, como se muestra en la figura 40. Además, serán actualizadas cada 500 ms.

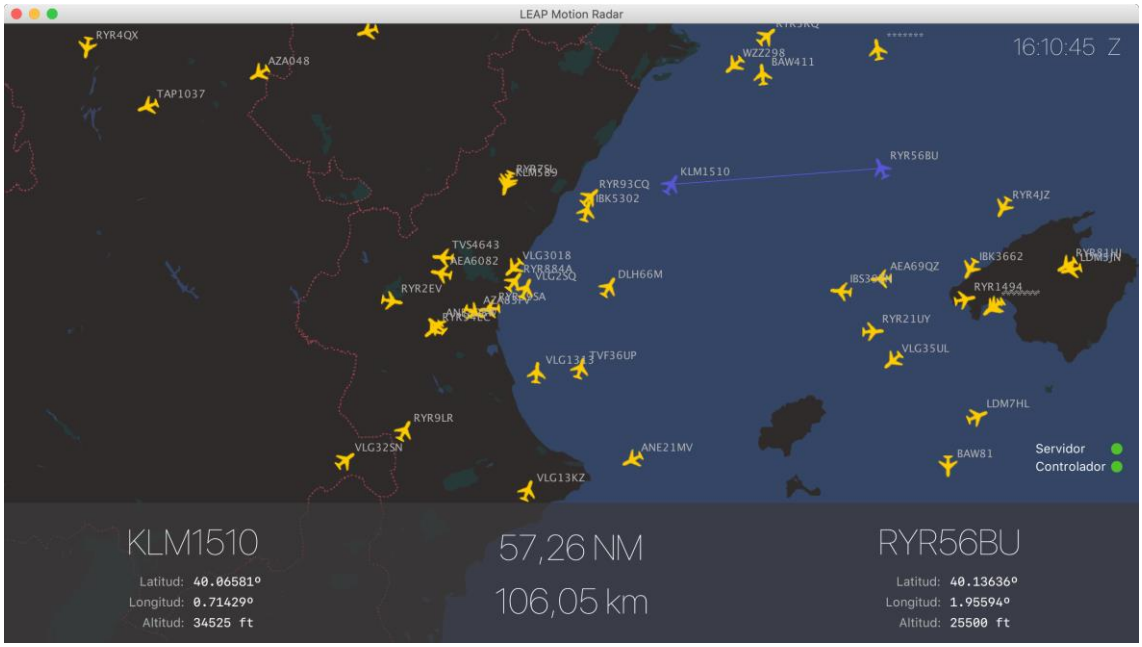


Figura 40: Vista del panel de distancias

Con todo ello, contaremos con una aplicación que permitirá representar tráficos reales y calcular distancias entre ellos; y tráficos sintéticos, pudiendo operar sobre ellos, calcular distancias y recibir alertas cuando haya riesgo de colisión.

6.4.3. Ajustes y calibración

Para la aplicación anteriormente descrita, se supone un funcionamiento fiable y calibrado del dispositivo *LEAP Motion* ™; cosa que, al tratar de hacer una aplicación multiplataforma, es complicada. Dependiendo del equipo utilizado (concretamente, de su pantalla), la transformación realizada por el sistema de apuntamiento será distinta, ya que esta depende del tamaño de pantalla y la distancia entre el controlador y la misma (ver punto 6.3).

Para ello, se incluirán dos opciones en el panel lateral con estos dos ajustes (véase figura 34), que permitirán que el dispositivo se encuentre perfectamente calibrado para el equipo con el que se use. Estos ajustes serán guardados en un archivo *app.properties* y recuperados al volver a iniciar la aplicación, para que el usuario no tenga que introducir los datos en cada uso. Simplemente en el primer uso deberá medir la distancia que separa la pantalla y el dispositivo *LEAP Motion* ™ e introducirla junto con el tamaño de pantalla.

Desde el código se puede elegir cualquier valor para ambos ajustes, pero por tratarse de los más comunes, se ha elegido un intervalo para el tamaño de pantalla de las 10.5 a las 17.0 pulgadas diagonales, y de los 200 a los 400 milímetros para la posición del controlador.

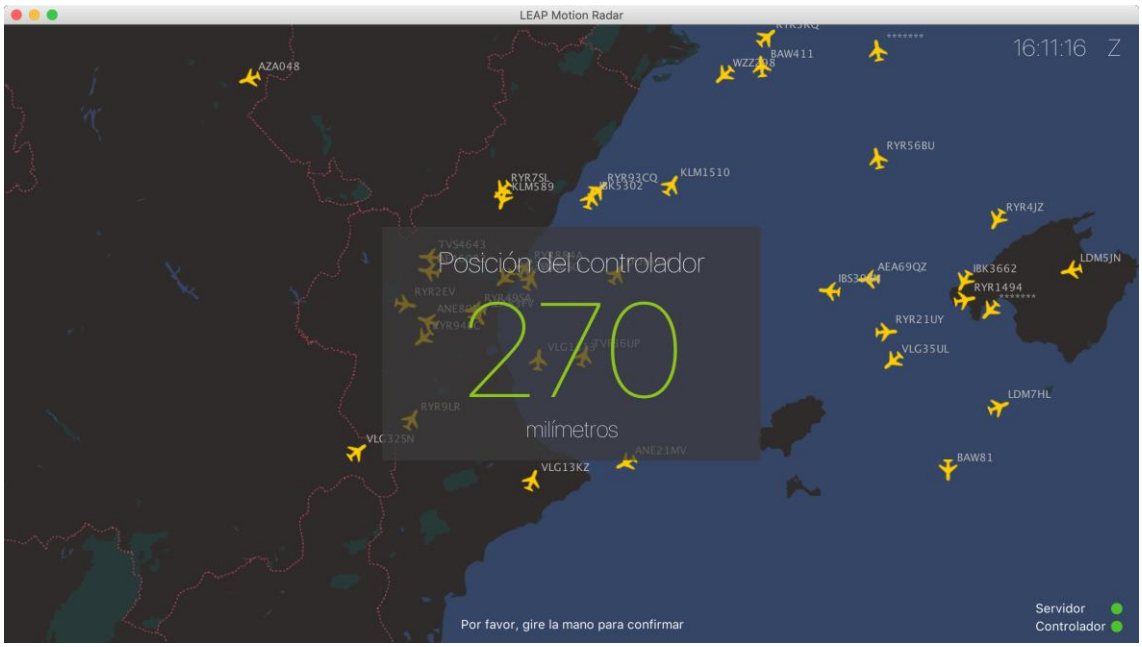


Figura 41: Vista del selector de la posición del controlador

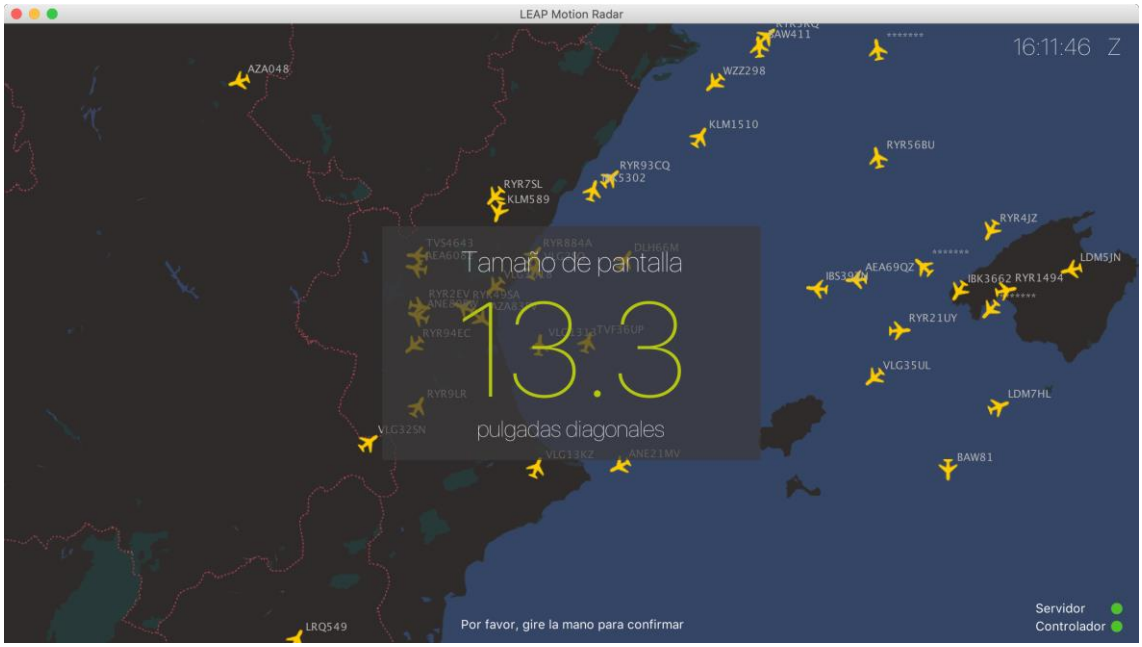


Figura 42: Vista del selector del tamaño de pantalla

Una vez cambiado el valor en cuestión, se escribirá a la variable correspondiente para ser utilizado en las fórmulas trigonométricas desarrolladas en el punto 6.3, para asegurar que el apuntamiento sea óptimo y el dispositivo *LEAP Motion* ™ sea transparente al usuario; centrándose este solo en interactuar con la interfaz sin tener en cuenta el dispositivo.

Para saber los ajustes que se han seleccionado sin entrar en el selector (cosa que podría hacer que el usuario los olvidara por error), se mostrarán en pequeño bajo cada ajuste en el panel lateral, como se observa en la figura 34.

6.4.4. Internacionalización

Finalmente, aunque la aplicación fue inicialmente desarrollada íntegramente en inglés, se realiza una internacionalización de todas las *strings* del código valiéndose del asistente proporcionado por *NetBeans*, para que esté disponible también en castellano y valenciano/catalán. Además, en el menú principal se incluye un ajuste (tras los otros dos: tamaño de pantalla y posición del controlador) que llevará a un selector de idioma (véase figura 43).

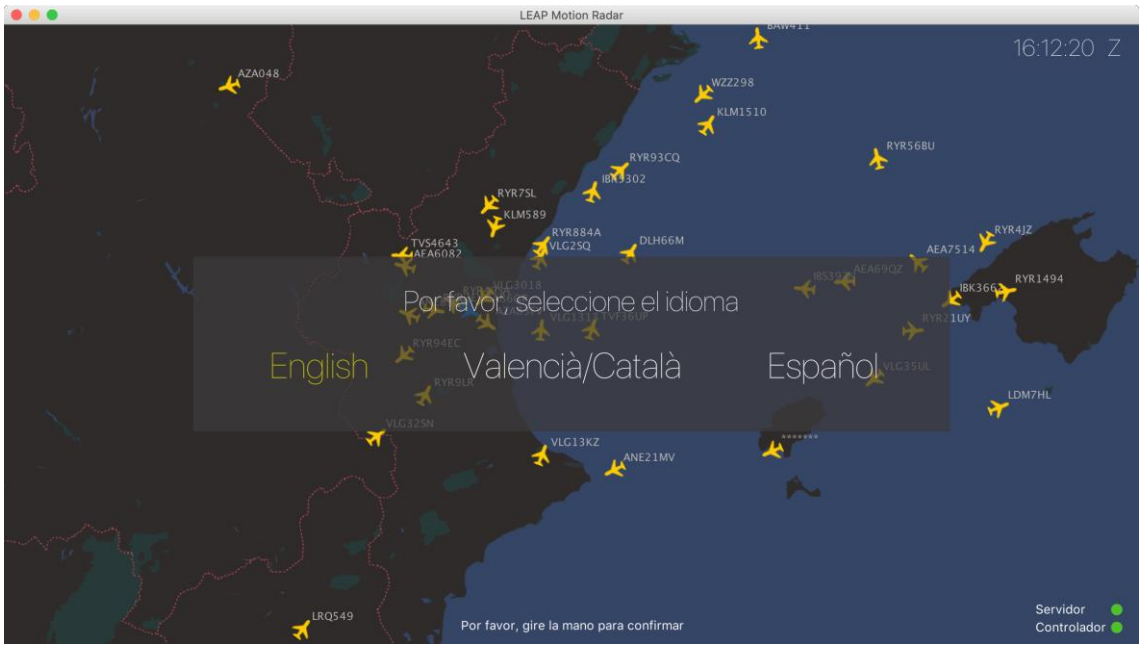


Figura 43: Vista del selector de idioma con los tres idiomas disponibles para la aplicación

Cuando se seleccione un idoma distinto al actual, se mostrará un mensaje de confirmación en la parte inferior central y se cambiará el texto de todas las etiquetas.

De forma análoga a los ajustes sobre la calibración del dispositivo, la selección del usuario será guardada en el archivo *app.properties* para que no sea necesario seleccionarla en cada uso.

7. ESTRUCTURA INTERNA DE LA APLICACIÓN

Seguidamente, se procederá a dar una explicación técnica exhaustiva del funcionamiento interno del código de la aplicación.

7.1. Diagrama de clases

A continuación, se muestra un diagrama detallado de las principales clases que componen la aplicación y sus relaciones entre ellas, así como de las variables y métodos más relevantes.

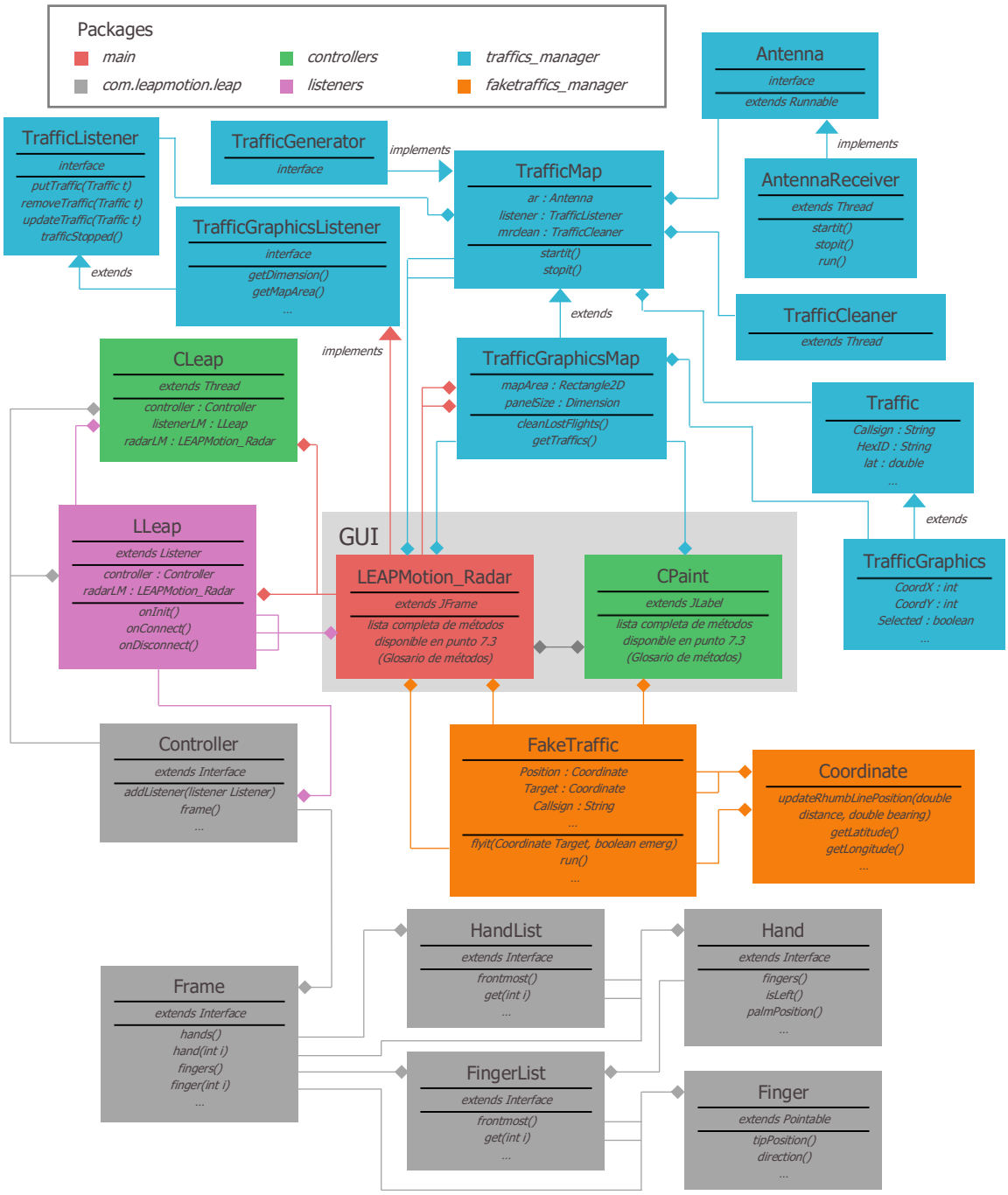


Figura 44: Diagrama de clases de la aplicación

Como se puede observar, contamos con seis paquetes (además de uno llamado *resources* que contendrá archivos esenciales como las fuentes, la imagen del mapa, los iconos de los aviones y los archivos de internacionalización): *main*, donde se encontrará *LEAPMotion_Radar*, el *JFrame* que regirá la aplicación; *controllers*, donde se encontrarán dos clases que controlarán aspectos básicos de la aplicación; *listeners*, donde encontraremos una clase tipo *listener* para acciones esenciales del dispositivo; *traffics_manager*, donde encontraremos todas las clases encargadas de gestionar la obtención de datos de *servantena*; *faketraffics_manager*, con las clases relativas a la gestión de tráficos sintéticos; y *com.leapmotion.leap*, con las clases proporcionadas por el fabricante del dispositivo *LEAP Motion* ™ (todas ellas compiladas e incluidas en la librería *LeapJava.jar*), esenciales para la obtención de datos sobre las manos del usuario y el funcionamiento de la aplicación.

7.2. Funcionamiento interno

En el primero de los paquetes comentados, encontramos la clase principal que gestionará la interfaz gráfica y la aplicación, en general. A pesar de delegar funciones a otras clases (a las controladoras, que se comentarán a continuación), sirve como nexo para todas las clases de la aplicación; gestiona todo el flujo de datos entre ellas y rige el comportamiento completo de la aplicación. Para ello, cuenta con multitud de métodos (que serán explicados detalladamente en el punto 7.3) y variables.

En el segundo de los distintos grupos, encontramos los controladores: dos clases en las que la anterior delega aspectos sobre el control de la aplicación. La primera de ellas, *CLeap*, rige todo lo relativo al dispositivo *LEAP Motion* ™; obtiene datos del dispositivo, reconoce el estado en el que se encuentra el usuario (por ejemplo: señalando, extrayendo el panel de ajustes, cerrando el panel de información, cambiando un valor, etc.) y dirige los datos necesarios a los métodos correspondientes de la clase *LEAPMotion_Radar* para que esta los gestione de forma adecuada. La segunda de ambas, *CPaint*, se corresponde al mapa radar y controla todo que se representa sobre él (sean tráficos reales o sintéticos, el cursor, el lazo de selección, etc.). Además, gestiona aspectos relativos a los tráficos como obtener cuáles han sido seleccionados o controlar si hay riesgo de colisión en el modo con tráficos sintéticos.

Respecto al paquete de *listeners*, solo encontramos una clase: *LLeap* (que extenderá de la clase *Listener* de la librería proporcionada por el fabricante del dispositivo). Se trata de una variación del *Listener* proporcionado por el fabricante y comentado previamente en el punto 6.1, de la que solo se han conservado los métodos que gestionan la conexión y desconexión del dispositivo. Cuando el dispositivo es conectado o desconectado, se lanza el método correspondiente, que a su vez ejecutan un método de *LEAPMotion_Radar* para notificar al usuario de la conexión o desconexión (mostrando un mensaje y cambiando el color del indicador de estado).

En el tercero de los paquetes, *traffics_manager*, encontramos todas las clases encargadas de la obtención y tratamiento de datos de *servantena*, para dirigirlos a *LEAPMotion_Radar* y *CPaint* de forma inteligible y gestionable. La fuente de todo este código es el material de la asignatura de Sistemas de Gestión de Vuelo por Computador, del Máster Universitario en Ingeniería Aeronáutica, donde se utilizan para trasladar los tráficos a un mapa muy básico (cosa

que, en este proyecto, ha sido llevada a una interfaz avanzada). Su clase principal sería *TrafficGraphicsMap*, un *Map* donde encontraremos todos los tráficos que recibe la antena con sus distintos parámetros separados. A nivel básico, se podría decir que es un "saco" donde se irán metiendo los tráficos que reciba la antena, y sacando cuando desaparezcan; y la GUI utilizará el método `getTraffics()` para obtenerlos cuando sea necesario. Dicha clase extenderá de otra llamada *TrafficMap* (que realiza las funciones básicas de obtención de datos), a la cual añadirá su tratamiento para representarlos en una interfaz. Los distintos tráficos serán del tipo *TrafficGraphics* (que al igual que antes, extiende de *Traffic* y le añade parámetros de representación gráfica, como las coordenadas *X* e *Y* sobre la pantalla a partir del tamaño del mapa y su rango de coordenadas), que contendrá información básica como las coordenadas del tráfico, su *Callsign*, etc. *TrafficMap* implementará un generador de tráficos (la clase *TrafficGenerator*) y se valdrá de otras clases para desempeñar sus tareas; por ejemplo, de *TrafficCleaner* para eliminar los tráficos perdidos, de *TrafficListener* para funciones como añadir, quitar o actualizar tráficos cuando sea oportuno, y *Antenna* para recibir datos del servidor. Finalmente, se contará con *AntennaReceiver*, la implementación de *Antenna* para la recepción de datos de un servidor (ya que esta se podría sustituir por otra que no ha sido usada, *AntennaEmulator*, que iría generando tráficos a partir de un archivo de recepción de datos ADS-B, que no será necesaria en el presente proyecto, ya que se cuenta con un potente generador de tráficos sintéticos), y con *TrafficGraphicsListener*, una *interface* que será implementada por *LEAPMotion_Radar* para permitir el funcionamiento de todo este complejo sistema de clases.

También se contará con las dos clases del paquete *faketraffics_manager*: *FakeTraffic* y *Coordinate*. La primera de ellas será un *Thread* que regirá el comportamiento de un avión ficticio, actualizando, en cada iteración, su posición. En su constructor se le definirá una posición inicial, rumbo y velocidad; y al ejecutar el método `flyit(Coordinate target, boolean emerg)` se pondrá en marcha y se le definirá el destino `target` al que se acudirá, así como si se encontrará en emergencia o no. Las coordenadas (tanto de posición como de destino) serán del tipo *Coordinate*, definido por la segunda de las clases comentadas. Esta clase dispone de un método `updateRhumbLinePosition`, el cual, introduciendo una distancia y un rumbo, actualizará dicha coordenada a la nueva posición (utilizando distancias loxodrómicas). Por tanto, en *LEAPMotion_Radar* se creará un *array* de *FakeTraffic* que contendrá los distintos vuelos ficticios, cuyos *Thread* irán actualizando las respectivas posiciones de acuerdo a sus rumbos y velocidades determinadas.

Finalmente, incluidas en el paquete *com.leapmotion.leap*, junto con muchas otras clases que no son necesarias en el presente proyecto, contaremos con una serie de clases que nos permitirán obtener datos del dispositivo. Cabe destacar que, al haber sido proporcionadas por el fabricante y no tratarse de *software* abierto, se encuentran compiladas y no se conoce el interior de ellas; solo sus distintos métodos. Se procederá a comentarlas en líneas generales, ya que su funcionamiento detallado se encuentra disponible en la documentación proporcionada por el fabricante[11]. Utilizaremos seis de las clases proporcionadas: *Controller*, *Frame*, *HandList*, *FingerList*, *Hand* y *Finger*. La primera de ellas, *Controller*, regirá la obtención de datos del dispositivo y será el único canal de comunicación entre la aplicación y el mismo. A dicho controlador, se le añadirá el *listener* anteriormente mencionado del que se obtendrán los eventos de conexión y desconexión. Además, con el método `frame()` obendremos una clase *Frame*, que

[11] Disponible en "Leap Motion Java SDK v3.2 Documentation".

contendrá toda la información extraída del dispositivo en ese instante concreto de tiempo; es decir, la clase *CLeap* comentada anteriormente, irá obteniendo datos de clases *Frame* obtenidas en cada iteración. De dicho *Frame*, a su vez, podremos obtener las otras cuatro clases: *HandList*, que será una recopilación de todas las manos detectadas por el dispositivo; *FingerList*, de forma análoga, pero con dedos; *Hand*, que será una mano concreta; y *Finger*, que será un dedo en concreto. En este proyecto, se utilizará para obtener estas dos últimas en ambos casos el método `frontmost()`, que devolverá, en cada caso, la mano o dedo más avanzado hacia la pantalla (es decir, con menor valor de *Z*). La extracción de datos de cada mano o dedo se realizará con distintos métodos (por ejemplo, utilizando `fingers()` en *Hand* para obtener el *FingerList* con todos los dedos de dicha mano, `palmPosition()` para la posición de dicha mano en forma de vector; o `direction()` para la dirección de un *Finger* concreto). Todas estas clases extenderán de otra proporcionada por el fabricante, llamada *Interface* (excepto *Finger*, que extenderá de la clase *Pointable*, que a su vez extenderá de *Interface*).

Como es obvio, se utilizarán muchas otras clases y librerías de apoyo para realizar el tratamiento de datos y operaciones matemáticas, como la clase *Math* y la librería *JavaX.VecMath* de *Java* ™ o la clase *Vector* de la librería del dispositivo.

7.3. Glosario de métodos por clase

Una vez se ha mostrado una visión general del funcionamiento interno del código que rige la aplicación, se procederá a dar una más detallada y pormenorizada explicando y comentando los métodos más relevantes de cada una de las clases anteriormente expuestas.

LEAPMotion_Radar

- `chooseConfirm(double palmPos, double palmRoll)`: Gestionará los mensajes de confirmación de salida y reinicio del modo Entrenar. Será ejecutado desde *CLeap* cuando el booleano `isWaitingForConfirm` sea *true* (que se habrá cambiado al intentar salir de dicho modo en los casos dispuestos en el punto 6.4.2), y se le enviará la posición en *X* de la palma de la mano y su *roll*. Con estos datos, el método obtendrá, en función de la posición de la mano, si el usuario está seleccionando "Sí" o "No"; y asignará un color entre amarillo y verde para el *no* y entre amarillo y rojo para el "Sí" (indicando así al usuario que esta última opción será peligrosa, ya que reiniciaría el modo). Esta asignación se realizará haciendo que los valores RGB de la variable `selectionColor` dependan de `palmRoll` (dentro de un máximo y mínimo), y cambiando el booleano `confirmed` a *true* cuando se alcance el *roll* correspondiente a la confirmación.

  Una vez la selección haya sido confirmada, se cambiará el booleano `isWaitingForConfirm` a *false* (con lo que no se seguirá entrando en el método desde *CLeap*), se volverá al estado anterior si se ha seleccionado "No", se volverá al modo Radar si se ha confirmado la salida o se sacará el panel de selección de dificultad si se ha confirmado el reinicio.

```
if (palmPosition < 0) {
    currentConfirm = 1;
    Color selectionColor;
    if (palmRoll > -0.7853 && palmRoll < 0.7853) {
        selectionColor = new Color(82, 194, 43);
        confirmed = true;
    } else {
        selectionColor = new Color((int) (73.4204 * Math.abs(palmRoll) + 24.3429),
                (int) (4.2440 * Math.abs(palmRoll) + 190.6672),
                (int) (-17.8246 * Math.abs(palmRoll) + 56.9977));
    }
    noLabel.setForeground(selectionColor);
    yesLabel.setForeground(new Color(255, 255, 255));
} else if (palmPosition > 0) {
    currentConfirm = 2;
    Color selectionColor;
    if (palmRoll > -0.7853 && palmRoll < 0.7853) {
        selectionColor = new Color(82, 194, 43);
        confirmed = true;
    } else {
        selectionColor = new Color(255, (int) (61.5373 * Math.abs(palmRoll) + 10.6747),
                (int) (-19.9466 * Math.abs(palmRoll) + 63.6641));
    }
    noLabel.setForeground(new Color(255, 255, 255));
    yesLabel.setForeground(selectionColor);
}
```

Figura 45: Fragmento de código relativo a la selección y asignación de color en el método `chooseConfirm`

- `chooseDifficulty(double palmPos, double palmRoll)`: Gestionará la selección de dificultad del inicio del modo Entrenar. Será ejecutado desde *CLeap* cuando el booleano `isWaitingForDifficulty` sea *true* (que se habrá cambiado al iniciar o reiniciar dicho modo en los casos dispuestos en el punto 6.4.2), y se le enviará la posición en *X* de la palma de la mano y su *roll*. Con estos datos, el método obtendrá, en función de la posición de la mano, qué dificultad (del 1 al 5) está seleccionando el usuario; y asignará un color entre amarillo y verde para dar una respuesta gráfica al usuario sobre la confirmación de dicha dificultad. Esta asignación se realizará haciendo que los valores RGB de la variable `selectionColor` dependan de `palmRoll` (dentro de un máximo y mínimo), y cambiando el booleano `confirmed` a *true* cuando se alcance el *roll* correspondiente a la confirmación.

  Una vez la selección haya sido confirmada, se cambiará el booleano `isWaitingForDifficulty` a *false* (con lo que no se seguirá entrando en el método desde *CLeap*), se programará un *TimerEvent* periódico (con el tiempo correspondiente a la selección, véase figura 46) del tipo `CreateFakeTraffic`, y se esconderá el panel de selección de dificultad.

```
if (confirmedValue) {
    difficultyPanel.setVisible(false);
    isWaitingForDifficulty = false;
    hideMessage();
    displayMessage(I18N[langCode].getString("LEVEL ") + String.valueOf(currentDifficulty)
            + I18N[langCode].getString(" OF DIFFICULTY SELECTED"), 5000);
    switch (currentDifficulty) {
        case 1:
            delayFakeTraffics = 60000;
            break;
        case 2:
            delayFakeTraffics = 45000;
            break;
        case 3:
            delayFakeTraffics = 30000;
            break;
        case 4:
            delayFakeTraffics = 20000;
            break;
        case 5:
            delayFakeTraffics = 10000;
            break;
    }
    isHolding = true;
    fakeTrafficTimer = new Timer();
    fakeTrafficTimer.schedule(new CreateFakeTraffic(this), 2000, delayFakeTraffics);
}
```

Figura 46: Fragmento de código relativo a la confirmación de un determinado nivel de dificultad y la creación periódica de tráficos sintéticos

- **`chooseLanguage(double palmPos, double palmRoll)`**: Gestionará la selección de idioma. Será ejecutado desde *CLeap* cuando el booleano **`isWaitingForLanguage`** sea *true* (que se habrá cambiado al acceder a la opción "Idioma" del panel lateral), y se le enviará la posición en *X* de la palma de la mano y su *roll*. Con estos datos, el método obtendrá, en función de la posición de la mano, qué lenguaje (entre los tres disponibles) está seleccionando el usuario; y asignará un color entre amarillo y verde para dar una respuesta gráfica al usuario sobre la confirmación de dicha dificultad. Esta asignación se realizará haciendo que los valores RGB de la variable **`selectionColor`** dependan de **`palmRoll`** (dentro de un máximo y mínimo), y cambiando el booleano **`confirmed`** a *true* cuando se alcance el *roll* correspondiente a la confirmación.

  Una vez la selección haya sido confirmada, se cambiará el booleano **`isWaitingForLanguage`** a *false* (con lo que no se seguirá entrando en el método desde *CLeap*), se cambiará el valor de **`langCode`** (que determina en qué idioma se obtienen las *strings*) todas las *strings* del todos las etiquetas para adaptar la aplicación al idioma escogido, y se esconderá el panel de selección de idioma.

```
if (palmPosition < -22.138 * screenSize / 6) {
    currentLanguage = 0;
    enGBLabel.setForeground(selectionColor);
    esESLabel.setForeground(new Color(255, 255, 255));
    caESLabel.setForeground(new Color(255, 255, 255));
} else if (palmPosition > 22.138 * screenSize / 6) {
    currentLanguage = 2;
    enGBLabel.setForeground(new Color(255, 255, 255));
    esESLabel.setForeground(selectionColor);
    caESLabel.setForeground(new Color(255, 255, 255));
} else {
    currentLanguage = 1;
    enGBLabel.setForeground(new Color(255, 255, 255));
    esESLabel.setForeground(new Color(255, 255, 255));
    caESLabel.setForeground(selectionColor);
}
```

Figura 47: Fragmento de código con la selección de uno de los idiomas de acuerdo a la posición de la palma

- **`chooseParameter(double palmPos, double palmRoll)`**: Gestionará la selección de los distintos parámetros que rigen un determinado vuelo sintético (rumbo, velocidad absoluta y altitud). Será ejecutado desde *CLeap* cuando el booleano **`isWaitingForParameter`** sea *true* (que se habrá cambiado al seleccionar un tráfico sintético), y se le enviará la posición en *X* de la palma de la mano y su *roll*. Con estos datos, el método obtendrá, en función de la posición de la mano, qué valor (entre los tres disponibles, y añadiendo un espacio extra por si no se quiere cambiar ningún valor) está seleccionando el usuario; y asignará un color entre amarillo y verde para dar una respuesta gráfica al usuario sobre la confirmación de dicha dificultad. Esta asignación se realizará haciendo que los valores RGB de la variable **`selectionColor`** dependan de **`palmRoll`** (dentro de un máximo y mínimo), y cambiando el booleano **`confirmed`** a *true* cuando se alcance el *roll* correspondiente a la confirmación.

  Una vez la selección haya sido confirmada, se cambiará el booleano **`isWaitingForParameter`** a *false* (con lo que no se seguirá entrando en el método desde *CLeap*), el booleano **`isWaitingForValue`** a *true* (indicando que se va a cambiar un valor) y se procederá a cambiar dicho valor seleccionado (o se cerrará la etiqueta si el usuario no ha realizado una confirmación sobre uno de los tres parámetros, si no sobre el *callsign* mostrado en la etiqueta del tráfico, mostrada anteriormente en la figura 38).

- `chooseSettings(double palmPos, double palmRoll)`: Gestionará la selección de las distintas opciones del panel lateral. Será ejecutado desde *CLeap* cuando el booleano `isSidePanelOut` sea *true* (que se habrá cambiado al sacar el panel con el gesto *swipe*), y se le enviará la posición en *Y* de la palma de la mano y su *roll*. Con estos datos, el método obtendrá, en función de la posición de la mano, qué opción del panel está seleccionando el usuario; y asignará un color entre amarillo y verde para dar una respuesta gráfica al usuario sobre la confirmación de dicha dificultad. Esta asignación se realizará haciendo que los valores RGB de la variable `selectionColor` dependan de `palmRoll` (dentro de un máximo y mínimo), y cambiando el booleano `confirmed` a *true* cuando se alcance el *roll* correspondiente a la confirmación.

  Una vez la selección haya sido confirmada, se esconderá el panel, se cambiará el booleano `isWaitingSidePanelOut` a *false* (con lo que no se seguirá entrando en el método desde *CLeap*), y se procederá a entrar en la opción elegida. Si se ha seleccionado "Modo: Radar" viniendo desde el propio modo o desde el asistente de cálculo de distancias, se accederá a dicho modo sin más; en cambio, si se procede del modo Entrenar, se accederá al selector de confirmación de salida de dicho modo (cambiando el booleano `isWaitingForConfirm` a *true*). Si se selecciona "Modo: Entrenar", se accederá al selector de dificultad del mismo (cambiando el booleano `isWaitingForDifficulty` a *true*), al selector de confirmación de reinicio si ya se estaba en el propio modo, o se saldrá del asistente de cálculo de Distancias. Si se selecciona "Distancias", se accederá a dicho asistente o se reiniciará si ya se encontraba dentro del mismo (lanzando el método `distancesManager(1)`). Para las dos opciones de calibración, se cambiará el booleano `isWaitingForValue` a *true*; y para "Idioma", el booleano `isWaitingForLanguage` (mostrando el panel correspondiente a el cambio de cada uno).

- `chooseTraffic(TrafficGraphics[] selectedTraffics, FakeTraffic[] selectedFakeTraffics, int selectedTrafficsCount)`: Gestionará la selección de los distintos tráficos que hayan sido incluidos en el "lazo" de selección. Será ejecutado desde *CPaint* cuando se hayan detectado uno o varios tráficos seleccionados, y se le enviará un *array* con dichos tráficos (sean reales o sintéticos) y un entero con el número total de tráficos. Con estos datos, el método extrerá el panel de información y representará los distintos *callsigns* en el mismo, para que posteriormente el usuario seleccione uno de ellos.

  Una vez hayan sido representados, se cambiará el booleano `isWaitingForTraffic` a *true*, indicando que se está esperando a elegir un tráfico; y el booleano `isInfoPanelOut` a *true*, indicando que se el panel de información se ha extraído. Si solo hay un tráfico, se ejecutará el método `showInfo(true, 0, 0)`, representando directamente el único tráfico (no será necesario utilizar el sistema de selección del método `showInfo`).

- `chooseValue(double palmPos, double palmRoll)`: Gestionará la selección de un valor y será utilizado para cambiar el tamaño de pantalla, posición del controlador o parámetros de un avión sintético. Será ejecutado desde *CLeap* cuando el booleano `isWaitingForValue` sea *true* (que será cambiado al requerir la entrada de uno de los valores anteriormente expuestos), y se le enviará la posición en *Y* de la palma de la mano y su *roll*. Con estos datos, el método obtendrá, en función de la posición de la mano, qué

valor (de acuerdo a las escalas correspondientes explicadas para cada caso concreto en los puntos 6.4.2 y 6.4.3) está seleccionando el usuario; y asignará un color entre amarillo y verde para dar una respuesta gráfica al usuario sobre la confirmación de dicha dificultad. Esta asignación se realizará haciendo que los valores RGB de la variable **`selectionColor`** dependan de **`palmRoll`** (dentro de un máximo y mínimo), y cambiando el booleano **`confirmed`** a *true* cuando se alcance el *roll* correspondiente a la confirmación.

Una vez la selección haya sido confirmada, se cambiará el booleano **`isWaitingForValue`** a *false* (con lo que no se seguirá entrando en el método desde *CLeap*) y se cambiará la variable que contenga el valor correspondiente, mostrando un mensaje de confirmación con el valor elegido en la parte inferior de la ventana. En el caso haber cambiado uno de los parámetros de un tráfico sintético, se cerrará el panel de información y se cambiará el booleano **`isInfoPanelOut`** a *false*.

```
case 4:
    value = 0.8 * palmPosition + 120;
    if (value > 400) {
        value = 400;
    } else if (value < 200) {
        value = 200;
    }
    valueLabel.setForeground(selectionColor);
    valueLabel.setText(String.valueOf(Math.round(value / 10) * 10));
    break;
case 5:
    value = 0.026 * palmPosition + 7.9;
    if (value > 17) {
        value = 17;
    } else if (value < 10.5) {
        value = 10.5;
    }
    valueLabel.setForeground(selectionColor);
    valueLabel.setText(String.valueOf(Math.round(value * 10.0) / 10.0));
    break;
```

Figura 48: Fragmento de código con la obtención de los valores de posición del controlador (caso 4) y del tamaño de pantalla (caso 5)

- **`connectedDevice(boolean connected)`**: Gestionará el cambio del indicador de estado del controlador (a verde cuando se ha conectado y a rojo cuando se ha desconectado). Se ejecuta desde *LLeap* desde los métodos correspondientes que se lanzan automáticamente en dichos eventos.

- **`displayMessage(String message, long milliseconds)`**: Gestionará el mostrado de mensajes para toda la aplicación. Podrá ser llamada desde cualquier punto donde se quiera mostrar un mensaje en la parte inferior central. Dicho mensaje será del tipo *String* y se mantendrá mostrado el tiempo definido por **`milliseconds`** (siendo permanente cuando esta variable es 0), mediante un *TimerEvent* (**`HideMessage()`**)que lo ocultará. En ella se cambiarán los booleanos **`isMessageDisplayed`** (para indicar que hay un mensaje mostrándose), i**`sNewMessageAvailable`** (para indicar que ha aparecido un mensaje nuevo y se debe ignorar el *TimerEvent* que ocultaba el anterior y **`isPermanentMessageDisplayed`** (para indicar que el mensaje mostrado es permanente y si se lanzara uno nuevo temporal, se debería volver al permanente al ocultar el nuevo).

```
public void displayMessage(String message, long milliseconds) {
    displayMessage.setText(message);
    displayMessage.setVisible(true);

    if (milliseconds == 0) { // If milliseconds == 0, the message is permanent
        isPermanentMessageDisplayed = true;
        permanentMessage = message;
    } else {
        if (isMessageDisplayed) {
            isNewMessageAvailable = true;
        }
        Timer timer = new Timer();
        timer.schedule(new HideMessage(), milliseconds);
    }
    isMessageDisplayed = true;
}
```

Figura 49: Fragmento del código con el método **displayMessage**

- **distancesManager(int step)**: Gestionará el cálculo de distancias, y se lanzará desde el método **chooseSettings** cuando se entre en "Distancias". Constará de tres pasos (definidos por el *input* **step**): el primero mostrará un mensaje pidiendo que se seleccione el primero de ambos tráficos, el segundo pedirá que se seleccione el segundo tráfico y lanzará el método **setShowDistLine** de *CPaint* para mostrar la línea de distancias entre el primer tráfico y el cursor, y el tercero mostrará el panel de información, calculará las distancias valiéndose del método **loxoDistance**, y programará un *TimerEvent* (**UpdateDistances()**) para actualizarlas cada 500 ms.

- **formComponentResized(ComponentEvent evt)**: Gestionará el reescalado de la ventana. Será lanzada automáticamente cuando el usuario modifique el tamaño de la ventana, y mantendrá el aspecto 16:9 de la misma. Modificará la posición y tamaño de absolutamente todos los componentes de la interfaz, para mantenerlos en el sitio proporcional que les corresponda (por lo que será muy extensa).

- **formWindowClosing(WindowEvent evt)**: Gestionará el cerrado de la aplicación. Será lanzada automáticamente cuando el usuario cierre la ventana, y detendrá el *TrafficGraphicsMap*, para cerrar correctamente la conexión con *servantena*.

- **getDimension()**: Devolverá el tamaño de la pantalla en píxeles, y será utilizado por el *TrafficGraphicsMap* y por *FakeTraffic* para obtener las coordenadas en píxeles sobre la ventana donde haya que representar un determinado tráfico.

- **getFakeTraffics()**: Devolvera un *array* de *FakeTraffic*, y será utilizado por *CPaint* para obtener los tráficos sintéticos a representar.

- **getFakeTrafficsCount()**: Devolverá un entero con el número total de tráficos sintéticos creados, y será utilizado por *CPaint* para evitar excederse del número total de tráficos incluidos en el *array* obtenido por el método anterior (para así evitar una *NullPointerException*).

- **getLanguage()**: Devolverá un entero indicando el código de idioma seleccionado por el usuario (0 para inglés, 1 para valenciano/catalán y 2 para español), que será utilizado en el resto de clases para extraer las *strings* del paquete de idioma correspondiente.

- `getMapArea()`: Devolverá un *Rectangle2D* con las coordenadas de los extremos del mapa, y será utilizado por el *TrafficGraphicsMap* y por *FakeTraffic* para obtener las coordenadas en píxeles sobre la ventana donde haya que representar un determinado tráfico.

- `hideCursor()`: Ejecutará el método homónimo de *CPaint* (ya que desde *CLeap* no se pueden ejecutar métodos de *CPaint*).

- `hideInfo(double palmPosition, boolean lastModeIsNotPalmDown)`: Gestionará el escondido del panel de información. Será ejecutado desde *CLeap* cuando el booleano `isInfoPanelOut` sea *true* y la palma se encuentre abierta y encarada hacia abajo, y se le enviará la posición en *Y* de de la misma. El método calculará la diferencia entre la primera posición recibida (cuando `lastModeIsNotPalmDown` era *true*) y la actual, y modificará la posición del panel respecto a ellas. Con ello, el usuario tendrá la sensación de estar moviendo el panel hacia abajo. Cuando se oculte completamente, se cambiará el booleano `isInfoPanelOut` a *false*, para evitar que se siga entrando en el método al abrir la mano.

- `hideMessage()`: Esconderá cualquier mensaje que se esté mostrando en la parte inferior central de la ventana en cualquier momento, sin necesidad de esperar al *TimerEvent* que los oculta.

- `initComponents()`: Iniciará todos los componentes de la GUI. Todo el código en este método está generado automáticamente por el entorno *NetBeans*, y será ejecutado automáticamente al inicio de la aplicación.

- `isInfoPanelOut()`: Devolverá un booleano indicando si el panel de información se encuentra visible. Será utilizado por *CLeap* para activar el gesto con la palma hacia abajo, que ocultaría dicho panel.

- `isSidePanelOut()`: Devolverá un booleano indicando si el panel de opciones se encuentra visible. Será utilizado por *CLeap* para bloquear todos los gestos y habilitar únicamente el que permitirá seleccionar una opción (el único necesario cuando el panel de opciones se encuentra extraído).

- `isWaitingForConfirm()`: Devolverá un booleano indicando si la interfaz se encuentra esperando una confirmación. Será utilizado por *CLeap* para ejecutar el método `chooseConfirm` cuando sea necesario.

- `isWaitingForDifficulty()`: Devolverá un booleano indicando si la interfaz se encuentra esperando a que el usuario seleccione un nivel de dificultad del modo Entrenar. Será utilizado por *CLeap* para ejecutar el método `chooseDifficulty` cuando sea necesario.

- `isWaitingForLanguage()`: Devolverá un booleano indicando si la interfaz se encuentra esperando a que el usuario seleccione un idioma. Será utilizado por *CLeap* para ejecutar el método `chooseLanguage` cuando sea necesario.

- **`isWaitingForParameter()`**: Devolverá un booleano indicando si la interfaz se encuentra esperando a que el usuario seleccione uno de los tres parámetros de un vuelo sintético para modificarlo. Será utilizado por *CLeap* para ejecutar el método **`chooseParameter`** cuando sea necesario.

- **`isWaitingForTraffic()`**: Devolverá un booleano indicando si la interfaz se encuentra esperando la selección de un tráfico. Será utilizado por *CLeap* para ejecutar el método **`chooseTraffic`** cuando sea necesario.

- **`isWaitingForTwistHand()`**: Devolverá un booleano indicando si la interfaz se encuentra esperando a que el usuario coloque la mano en la posición adecuada antes de hacer una confirmación. Será utilizado por *CLeap* para evitar que si se tiene la mano en posición de confirmación antes de mostrar un valor, se confirme el primero que salga sin que el usuario pueda elegir.

- **`isWaitingForValue()`**: Devolverá un booleano indicando si la interfaz se encuentra esperando la selección de un valor. Será utilizado por *CLeap* para ejecutar el método **`chooseValue`** cuando sea necesario.

- **`loxoDistance(double lat1, double lon1, double alt1, double lat2, double lon2, double alt2)`**: Devolverá la distancia en metros entre dos tráficos definidos por las coordenadas y altitudes introducidas. Para ello, se valdrá de las fórmulas de cálculo de distancias loxodrómicas, implementadas como se muestra en la figura 50.

```
lon1 = lon1 * (Math.PI / 180);
lon2 = lon2 * (Math.PI / 180);
lat1 = lat1 * (Math.PI / 180);
lat2 = lat2 * (Math.PI / 180);

dlat = lat2 - lat1;
dlon = lon2 - lon1;
dalt = alt2 - alt1;
dPhi = Math.log(Math.tan(Math.PI / 4 + lat2 / 2) / Math.tan(Math.PI / 4 + lat1 / 2));

if (Double.isFinite(dlat / dPhi)) {
    q = dlat / dPhi;
} else {
    q = Math.cos(lat1);
}

if (Math.abs(dlon) > Math.PI) {
    if (dlon > 0) {
        dlon = -(2 * Math.PI - dlon);
    } else {
        dlon = 2 * Math.PI + dlon;
    }
}
dist = Math.sqrt(Math.pow(dlat, 2) + Math.pow(q, 2) * Math.pow(dlon, 2)) * earthRadius;
return Math.sqrt(Math.pow(dist, 2) + Math.pow(dalt, 2));
```

Figura 50: Fragmento del código con la implementación del cálculo de la distancia loxodrómica entre dos puntos, teniendo en cuenta la diferencia de altitudes

- **`showInfo(boolean oneTraffic, double palmPosition, double palmRoll)`**: Gestionará la selección de los distintos tráficos incluidos en el "lazo" de selección (si solo se ha incluido uno, determinado por el booleano **`oneTraffic`**, se pasará directamente a su representación). Será ejecutado desde *CLeap* cuando el booleano **`isWaitingForTraffic`** sea *true* (que se habrá cambiado al presentar los tráficos en el panel de información, por el método **`chooseTraffic`**), y se le enviará la posición en *X* de la palma de la mano y su *roll*. Con estos datos, el método obtendrá, en función de la posición de la mano, qué tráfico está seleccionando el usuario; y asignará

un color entre amarillo y verde para dar una respuesta gráfica al usuario sobre la confirmación de dicho tráfico. Esta asignación se realizará haciendo que los valores RGB de la variable `selectionColor` dependan de `palmRoll` (dentro de un máximo y mínimo), y cambiando el booleano `confirmed` a *true* cuando se alcance el *roll* correspondiente a la confirmación.

Una vez la selección haya sido confirmada, se cambiará el booleano `isWaitingForTraffic` a false (con lo que no se seguirá entrando en el método desde *CLeap*), se mostrará la información del tráfico en cuestión (con la disposición adecuada para cada tipo de tráfico, real o sintético, como se expone en el punto 6.4.2), yse programará un *TimerEvent* periódico (`UpdateLabels()`) para actualizar los datos cada 500 ms.

- `toggleInfoPanel()`: Esconderá el panel de información cuando se finalice el gesto de arrastrar la palma hacia abajo.

- `toggleSidePanel()`: Mostrará el panel de información cuando se finalice el gesto de *swipe* usado para extraerlo.

- `trackFinger(Vector dir, Vector pos)`: Se tratará del código del sistema de apuntamiento presentado en el punto 6.3 (figura 20), que recibirá la dirección y posición de la punta del dedo y representará el cursor en la pantalla. Será ejecutado desde *CLeap* cuando se tenga el dedo extendido.

- `trackSwipe(double palmPosition, boolean lastModeIsNotSwipe)`: Gestionará el gesto *swipe* que extraerá el panel de opciones. Será ejecutado desde *CLeap* cuando la mano se encuentre extendida y en posición vertical (encarada hacia derecha o izquierda). Recibirá la coordenada *X* de la posición de la mano y un booleano indicando si se acaba de entrar en el modo *swipe*. Al entrar, se guarda el primer valor de la posición y posteriormente se modifica la posición del panel de opciones de acuerdo con la diferencia entre el valor inicial y el actual (con lo que el usuario tiene la sensación de estar arrastrando el panel con su mano). Cuando se sale del modo *swipe*, *CLeap* lanzará el método `toggleSidePanel()` y si este se encuentra tras cierto umbral, se mantendrá extraído y se ejecutará `chooseSettings`.

- `waitTwistHand(double palmRoll)`: Gestionará la espera hasta que el usuario coloque la mano en la posición previa a la selección (para evitar que se seleccione el primer valor u opción sin que el usuario tenga respuesta). Será ejecutado por *CLeap* cuando el booleano `isWaitingForTwistHand` sea *true*.

CLeap

Tal y como sucedía en la aplicación del punto 6.3, *CLeap* será un *Thread* (hilo de ejecución de la aplicación, que se ejecutará concurrentemente junto a otros y la propia aplicación) que tomará la forma de una especie de "máquina de estados". Constará de dos bloques principales: uno donde se reconocerá la acción o estado de control que esté ejerciendo el usuario, y otro donde se ejecutará la acción pertinente para responder al control del usuario.

En concreto, se contará con catorce estados:

- `MODE_CHOOSECONFIRM`: Será utilizado cuando se esté esperando una confirmación (estado determinado por el método `isWaitingForConfirm()` de *LEAPMotion_Radar*). Ejecutará el método `chooseConfirm` de *LEAPMotion_Radar*.
- `MODE_CHOOSEDIFFICULTY`: Será utilizado cuando se esté esperando una selección de dificultad (estado determinado por el método `isWaitingForDifficulty()` de *LEAPMotion_Radar*). Ejecutará el método `chooseDifficulty` de *LEAPMotion_Radar*.
- `MODE_CHOOSELANGUAGE`: Será utilizado cuando se esté esperando una selección de idioma (estado determinado por el método `isWaitingForLanguage()` de *LEAPMotion_Radar*). Ejecutará el método `chooseLanguage` de *LEAPMotion_Radar*.
- `MODE_CHOOSEPARAMETER`: Será utilizado cuando se esté esperando una selección de parámetro de un vuelo sintético (estado determinado por el método `isWaitingForParameter()` de *LEAPMotion_Radar*). Ejecutará el método `chooseParameter` de *LEAPMotion_Radar*.
- `MODE_CHOOSESETTINGS`: Será utilizado cuando se esté esperando una selección del panel de opciones (estado determinado por el método `isSidePanelOut()` de *LEAPMotion_Radar*). Ejecutará el método `chooseSettings` de *LEAPMotion_Radar*.
- `MODE_CHOOSETRAFFIC`: Será utilizado cuando se esté esperando una selección de tráficos entre los incluidos en el "lazo" (estado determinado por el método `isWaitingForTraffic()` de *LEAPMotion_Radar*). Ejecutará el método `showInfo` de *LEAPMotion_Radar*.
- `MODE_CHOOSEVALUE`: Será utilizado cuando se esté esperando una selección de un valor (estado determinado por el método `isWaitingForValue()` de *LEAPMotion_Radar*). Ejecutará el método `chooseValue` de *LEAPMotion_Radar*.
- `MODE_NULL`: Se trata del estado nulo. Será utilizado cuando no se haya detectado ningún estado reconocible (por ejemplo, introduciendo tres manos) o haya habido algún error. No ejecutará ninguna acción.
- `MODE_PALMUP`: Será utilizado cuando se esté realizando un gesto desplazando la mano con la palma hacia arriba. Ha sido creado, pero actualmente no es necesario para la aplicación presente, pudiendo implementársele alguna función en lanzamientos futuros.
- `MODE_SWIPE`: Será utilizado cuando se esté realizando un gesto desplazando la mano con la palma hacia derecha o izquierda (el gesto *swipe*), para extraer el panel de opciones mediante el método `trackSwipe` de *LEAPMotion_Radar*.

- `MODE_TRACKING`: Será utilizado cuando se tenga extendido un dedo y se esté señalando con el mismo. Ejecutará el método trackFinger de *LEAPMotion_Radar*, que gestionará el sistema de apuntamiento.
- `MODE_TWISTHAND`: Será utilizado cuando se esté esperando a que el usuario coloque la mano en la posición adecuada (estado determinado por el método `isWaitingForTwistHand()` de *LEAPMotion_Radar*). Ejecutará el método `waitTwistHand` de *LEAPMotion_Radar*.
- `MODE_ZOOM`: Será utilizado cuando se esté realizando un gesto de "ampliación", con ambas manos con las palmas extendidas. Ha sido creado, pero actualmente no es necesario para la aplicación presente, pudiendo implementársele alguna función en lanzamientos futuros (como por ejemplo, ampliar o reducir un mapa si se dispusiese de varias antenas).

CPaint

- `checkCollisionRisk()`: Gestionará el sistema de detección de colisiones. Será ejecutado por *LEAPMotion_Radar* cada vez que se cree un tráfico sintético o se modifique la trayectoria de alguno de los existentes, y comprobará que la estimación de distancia mínima entre los tráficos (comparando las trayectorias de dos en dos) siempre se encuentra por encima de un umbral (que se ha definido en 1000 metros). Si fuera menor, se marcaría los tráficos con el booleano `CollisionRisk`.

```
Vector3d normc = new Vector3d();

Vector3d v1 = new Vector3d();
Vector3d v2 = new Vector3d();

Vector3d pca1 = new Vector3d();
Vector3d pca2 = new Vector3d();
Vector3d minDist = new Vector3d();

v1.scale(fakeTraffics[i].GroundSpeed * 1.852 / 3.6, normp1);
v2.scale(fakeTraffics[j].GroundSpeed * 1.852 / 3.6, normp2);

r.sub(pos1, pos2);
c.sub(v1, v2);

normc.normalize(c);

Vector3d cross_r_normc = new Vector3d();

cross_r_normc.cross(r, normc);
rm.cross(normc, cross_r_normc);

double tau = -r.dot(c) / c.dot(c);

if (tau > 0) {
    v1.scale(tau);
    v2.scale(tau);
    pca1.add(pos1, v1);
    pca2.add(pos2, v2);
    minDist.sub(pca2, pca1);
    if (minDist.length() < collisionRiskThreshold) {
        fakeTraffics[i].CollisionRisk = true;
        fakeTraffics[j].CollisionRisk = true;
        radarLM.displayMessage(I18N[radarLM.getLanguage()].getString("COLLISION RISK DETECTED BETWEEN TRAFFICS ") +
                fakeTraffics[i].Callsign + I18N[radarLM.getLanguage()].getString(" AND ") + fakeTraffics[j].Callsign +
                I18N[radarLM.getLanguage()].getString(" IN APROX. ") + String.valueOf(Math.round(tau / 60)) + " min", 5000);
    }
}
```

Figura 51: Algoritmo de detección de riesgo de colisiones (donde `pos1` y `pos2` son los vectores de posición de ambos tráficos; y `normp1` y `normp2`, los vectores unitarios de dirección)

- `checkSelectedTraffics()`: Gestionará el método de selección por "lazo". Será ejecutado cuando se complete el polígono de selección, y comprobará uno a uno todos los tráficos disponibles, para obtener cuáles se encuentran dentro. Los que se encuentren

dentro serán incluidos en un *array* de tráficos (reales o sintéticos) que será utilizado por *LEAPMotion_Radar* para los pasos posteriores del proceso de selección.

```
TrafficGraphics[] selectedTraffics = new TrafficGraphics[5];
TrafficGraphics[] traffics = trafficGraphicsMap.getTraffics();
for (TrafficGraphics traffic : traffics) {
    traffic.Selected = selectionShape.contains(traffic.CoordX, traffic.CoordY);
    if (traffic.Selected) {
        if (count < 5) {
            selectedTraffics[count] = traffic;
        }
        count++;
    }
}
if (count > 0 && count <= 5) {
    radarLM.chooseTraffic(selectedTraffics, null, count);
} else if (count == 0) {
    radarLM.displayMessage(I18N[radarLM.getLanguage()].getString("NO TRAFFICS SELECTED"), 3000);
} else if (count > 5) {
    radarLM.displayMessage(I18N[radarLM.getLanguage()].getString("TOO MANY TRAFFICS SELECTED, PLEA
    deselectAllTraffics();
}
```

Figura 52: Fragmento del código con el proceso de comprobación de los tráficos (reales, para este caso) seleccionados, donde **`selectionShape`** es el polígono determinado por el "lazo"

- **`deselectAllTraffics()`**: Reseteará todos los tráficos respecto al booleano **`Selected`**, que determina cuando un tráfico está siendo seleccionado y debe representarse en verde.

- **`dispCursor(int x, int y, int squareSize, boolean pressed)`**: Gestionará el mostrado del cursor. Será lanzado por el método **`trackFinger`** de *LEAPMotion_Radar*, del que recibirá las coordenadas calculadas de su posición, el tamaño del cursor, y si está "pulsado". Si estuviera pulsado, se empezaría a trazar la figura de selección, y este mismo método comprobaría en cada iteración si dicho "lazo" se intersecta. Cuando se completará, se lanzaría el método **`checkSelectedTraffics()`**.

```
if (pressed && !selectionCompleted) {
    if (totalSelectionPoints < 150 && totalSelectionPoints != 0) {
        Line2D line = new Line2D.Double(selectionPoints[0][totalSelectionPoints - 1],
                selectionPoints[1][totalSelectionPoints - 1], x, y);
        for (int i = 1; i < totalSelectionPoints - 10; i++) {
            if (line.intersectsLine(selectionPoints[0][i - 1], selectionPoints[1][i - 1], selectionPoints[0][i], selectionPoints[1][i])
                    && selectionPoints[0][i] != selectionPoints[0][i - 1] && selectionPoints[1][i] != selectionPoints[1][i - 1]) {
                checkSelectedTraffics(new Polygon(Arrays.copyOfRange(selectionPoints[0], i, totalSelectionPoints),
                        Arrays.copyOfRange(selectionPoints[1], i, totalSelectionPoints), totalSelectionPoints - i));
                selectionCompleted = true;
                break;
            }
        }
    } else if (totalSelectionPoints >= 150) {
        totalSelectionPoints = 0;
    }
    selectionPoints[0][totalSelectionPoints] = x;
    selectionPoints[1][totalSelectionPoints] = y;
    totalSelectionPoints++;
} else if (!pressed) {
    totalSelectionPoints = 0;
    selectionCompleted = false;
}
```

Figura 53: Fragmento del código con la generación del polígono de selección y su comprobación

- **`hideCursor()`**: Esconderá el cursor. Será utilizado cuando haya algún panel o menú extraído que sea incompatible con la selección.

- **`paintComponent(Graphics g)`**: Será utilizado interna y automáticamente por la aplicación para representar los elementos sobre el mapa. Llamará a los métodos **`paintFlightData`** y **`paintFakeFlightData`** para representar, respectivamente, los tráficos reales y sintéticos.

- **`paintFakeFlightData(Graphics2D g)`**: Representará los tráficos sintéticos disponibles. Los obtendrá del *array* correspondiente, ejecutará transformaciones sobre el

icono del tráfico para que se muestren de acuerdo a sus rumbos, y los representará sobre el mapa. También elegirá el color correspondiente a su estado (normal, emergencia, riesgo de colisión, seleccionado y seleccionado para distancias).

- `paintFlightData(Graphics2D g)`: Representará los tráficos reales disponibles. Los obtendrá del método `getTraffics()` de *TrafficGraphicsMap*, ejecutará transformaciones sobre el icono del tráfico para que se muestren de acuerdo a sus rumbos, y los representará sobre el mapa. También elegirá el color correspondiente a su estado (normal, emergencia, alerta, en tierra, seleccionado y seleccionado para distancias).

- `removeAllCalcDist()`: Reseteará todos los tráficos respecto al booleano `CalcDist`, que determina cuando un tráfico está siendo utilizado para calcular distancias y debe representarse en morado.

- `setPaintRealTraffics(boolean paintRealTraffics)`: Se utilizará por *LEAPMotion_Radar* para determinar si se deben representar los tráficos reales (*true*) o los sintéticos (*false*), dependiendo del modo en el que se encuentre la aplicación.

- `setShowDistLine(int showDistLine, TrafficGraphics[] distanceTraffics, FakeTraffic[] distanceFakeTraffics)`: Se utilizará para determinar si se debe representar la línea de distancias entre un tráfico seleccionado y el cursor (`showDistLine` = 1), entre dos tráficos seleccionados (`showDistLine` = 2) o no se debe mostrar (`showDistLine` = 0). Los otros dos *inputs* del método recibirán, respectivamente, los tráficos reales o sintéticos para los que se va a calcular la distancia.

- `setTrafficGenerator(TrafficGraphicsMap trafficGraphicsMap)`: Será utilizado para recibir de *LEAPMotion_Radar* el generador de tráficos reales.

LLeap

- `onConnect(Controller controller)`: Gestionará la conexión del dispositivo. Será ejecutado automáticamente cuando se conecte el dispositivo, mostrará un mensaje con el método `displayMessage` y ejecutará el método `connectedDevice(true)` de *LEAPMotion_Radar*.

- `onDisconnect(Controller controller)`: Gestionará la desconexión del dispositivo. Será ejecutado automáticamente cuando se desconecte el dispositivo, mostrará un mensaje con el método `displayMessage` y ejecutará el método `connectedDevice(false)` de *LEAPMotion_Radar*.

TrafficMap

- `startit()`: Iniciará la obtención de datos desde *servantena*.

- `stopit()`: Cerrará correctamente la conexión con *servantena*.

TrafficGraphicsMap

- `getTraffics()`: Devolverá un *array* con todos los tráficos recibidos desde *servantena*, siendo cada uno de ellos determinado por la clase *TrafficGraphics*, y conteniendo toda la información correspondiente al mismo.

FakeTraffic

- `flyit(Coordinate Target, boolean emerg)`: Iniciará la simulación del tráfico en cuestión hacia las coordenadas `Target`. Si `emerg` es *true*, se encontrará en estado de emergencia.
- `getPosition()`: Devolverá las coordenadas de la posición actual simulada del tráfico.
- `getSpeed()`: Devolverá la velocidad absoluta relativa al suelo del tráfico en nudos.
- `getTrack360()`: Devolverá el rumbo del tráfico de 0 a 360 grados respecto al norte geográfico.
- `newDestinationFromTrack(double newTrack)`: Será utilizado para cambiar el rumbo del tráfico, estableciendo un nuevo destino a partir del nuevo rumbo.

```
public void newDestinationFromTrack(double newTrack) {
    Target = Position.getRhumbLineDestination(Position.getRhumbLineDistance(Target), newTrack);
    Track = newTrack;
}
```

Figura 54: Fragmento de código relativo al cambio de rumbo de un tráfico sintético, recalculando el nuevo destino con el método `getRhumbLineDestination` de *Coordinate*

- `setAltitude(double Altitude)`: Será utilizado para cambiar la altitud en pies del tráfico simulado.
- `setSpeed(double GroundSpeed)`: Será utilizado para cambiar la velocidad absoluta relativa al suelo en nudos del tráfico simulado.

Coordinate

- `getAltitude()`: Devolverá la altitud en pies de la coordenada.
- `getLatitude()`: Devolverá la latitud en grados de la coordenada.
- `getLongitude()`: Devolverá la longitud en grados de la coordenada.
- `getRhumbLineDestination(double distance, double bearing)`: Devolverá las coordenadas del punto ubicado a una distancia `distance` y un rumbo `bearing` de las coordenadas en cuestión, utilizando la distancia loxodrómica.
- `updateRhumbLinePosition(double distance, double bearing)`: Actualizará la posición de las coordenadas en cuestión a las ubicadas a una distancia

`distance` y un rumbo `bearing` de las mismas, utilizando la distancia loxodrómica. Será utilizado por los distintos *FakeTraffic* para "moverse" durante la simulación.

Controller

- `addListener(Listener listener)`: Será utilizado para añadir al controlador el *Listener* del dispositivo (en esta aplicación, *LLeap*).
- `frame()`: Devolverá un *Frame* con los datos actuales obtenidos por el dispositivo.

Frame

- `hands()`: Devolverá una *HandList* con la lista de manos detectadas por el controlador.
- `hand(int i)`: Devolverá una *Hand* con la mano número `i`.
- `fingers()`: Devolverá una *FingerList* con la lista de dedos detectados por el controlador.
- `finger(int i)`: Devolverá un *Finger* con el dedo número `i`.

HandList

- `frontmost()`: Devolverá una *Hand* con la mano más avanzada hacia la pantalla (con menor coordenada de posición *Z*).
- `get(int i)`: Devolverá una *Hand* con la mano número `i`.

FingerList

- `count()`: Devolverá el número total de dedos de la *FingerList*. Será utilizado para contar los dedos que se encuentren extendidos.
- `extended()`: Devolverá otra *FingerList* con la lista de dedos extendidos.
- `frontmost()`: Devolverá un *Finger* con el dedo más avanzado hacia la pantalla (con menor coordenada de posición *Z*).
- `get(int i)`: Devolverá un *Finger* con el dedo número `i`.

Hand

- `fingers()`: Devolverá una *FingerList* con la lista de dedos de la mano en cuestión.
- `isLeft()`: Devolverá un booleano indicando si la mano es izquierda.

- `isRight()`: Devolverá un boooleano indicando si la mano es derecha.
- `palmNormal()`: Devolverá el vector dirección normal al plano determinado por la palma de la mano en cuestión.
- `palmPosition()`: Devolverá el vector posición del centro de la palma de la mano en cuestión.

Finger

- `direction()`: Devolverá el vector dirección del dedo en cuestión.
- `tipPosition()`: Devolverá el vector posición de la punta del dedo en cuestión.

8. PRUEBAS Y RESULTADOS

Durante todo el proceso de desarrollo del proyecto, se han ejecutado multitud de pruebas a la par de su avance. Al tratarse de un sistema relativamente portátil, no se requería un banco o zona de pruebas con condiciones especialmente concretas, pudiendo realizarse estas en cualquier lugar con una iluminación correcta (para evitar posibles distorsiones en las cámaras infrarrojas).

Inicialmente, se probó que fuera posible comunicar el entorno *NetBeans* con el dispositivo *LEAP Motion* ™ mediante la aplicación de muestra proporcionada por el fabricante. Tras varios intentos, se consiguió obtener satisfactoriamente datos por consola. Posteriormente, disponiendo de la ventana con datos desarrollada, se pudieron realizar pruebas más efectivas, comparando las magnitudes reales de posición de la mano respecto al dispositivo (medidas aproximadamente) con las que estaba mostrando el dispositivo en la interfaz.

El grueso de pruebas empezaría a partir del desarrollo de la aplicación de gestos. A partir de ella, se requería un cierto nivel de precisión (para señalar correctamente o realizar movimientos y gestos de forma precisa) y se empezó a observar que cuando la iluminación no era óptima (por ejemplo, cuando se tenía un exceso de luz de tubos fluorescentes o el dispositivo recibía luz solar directa) las medidas llegaban con ruido, y la posición del cursor era muy inestable. Por tanto, se decidió ejecutar las pruebas a baja iluminación y con el equipo informático conectado a la luz (ya que así mejoraba su rendimiento y potencia y el muestreo era más preciso).

Figura 55: Banco de pruebas de la aplicación de gestos y de radar (equipo conectado a la corriente, e iluminación tenue pero visible)

Finalmente, con la aplicación radar, se procedería de forma idéntica: creando un entorno favorable para el correcto funcionamiento del dispositivo con el menor nivel de interferencias posible. Para esta parte sería requisito indispensable disponer de una conexión estable a Internet, necesaria para la correcta recepción de datos desde *servantena*. También se observaron problemas de caídas puntuales del servidor, especialmente en días de lluvia, que fueron solucionadas con un simple *reseteo* del mismo.

Además, una vez se tuvo la aplicación final completamente desarrollada y operativa, se realizaron las pruebas finales, que fueron grabadas en vídeo (tanto desde una cámara, como

grabando la pantalla en el propio equipo) para la creación de una demostración funcional y operativa completa de la misma.

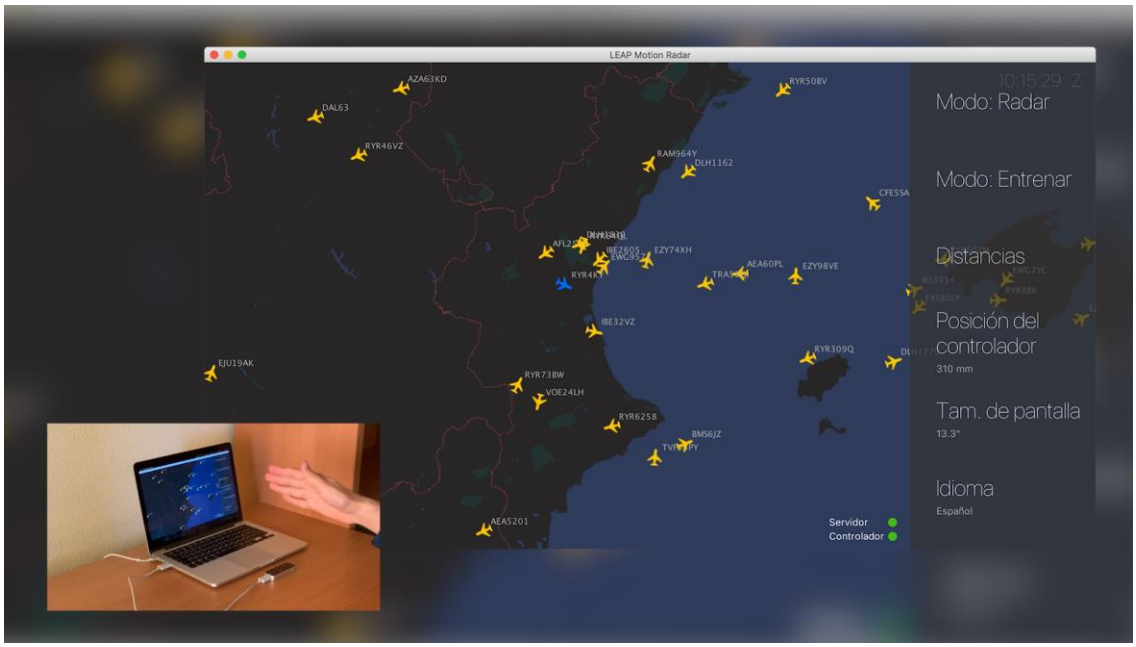


Figura 56: Fotograma del vídeo demostrativo grabado durante las pruebas finales de la aplicación de radar

En cuanto a los resultados obtenidos, se podría considerar que han sido plenamente satisfactorios, ya que se ha podido implementar la idea inicial en una aplicación plenamente funcional y operativa con las características que se deseaban. Durante este desarrollo, se han ido cumplimentando los distintos hitos o subobjetivos presentados en el apartado 1.2 (*Objetivos*) y puestos sobre el plano de la realidad en el 5 (*Implementación*); consiguiendo paso por paso un mejor conocimiento de la tecnología de control de gestos y su adaptación al terreno del control de tráfico aéreo, hasta conseguir como resultado la aplicación que demuestra que estas tecnologías pueden ser perfectamente implementadas para tales fines.

9. CONCLUSIONES

La conclusión principal que se puede inferir del presente proyecto se obtiene de forma inmediata: es posible conseguir un sistema de entrada alternativo al teclado y el ratón para el manejo de aplicaciones de control aéreo. La tecnología de control por gestos ofrece infinidad de posibilidades, solo unas pocas de todas ellas han sido utilizadas en este proyecto; y si se llegara a un completo desarrollo, sería posible obtener un sistema de interacción con los equipos informáticos mucho más inmediato, eficiente e intuitivo que cualquiera de los que es posible encontrar hoy en día.

El rol más relevante para la consecución de los objetivos del proyecto ha sido llevado por el dispositivo controlador *LEAP Motion* ™, con el que se ha podido demostrar el funcionamiento de la tecnología de control por gestos. La potencia, precisión y versatilidad del mismo ha resultado ser mucho mayor de lo que se pensaba inicialmente, cuando la idea era obtener una aplicación de radar básica con la que realizar algunos pocos gestos, pero con el que se ha podido conseguir un control completo y absoluto de la misma utilizando una gran variedad de gestos desarrollados íntegramente para ella y controlados a bajo nivel.

Con todo ello, y añadiéndole una interfaz con un diseño agradable y amigable para el usuario y un simulador de control de tráfico aéreo, se ha conseguido la aplicación final; que, al no tener ni siquiera implementado ningún tipo de código para obtener datos del teclado o del ratón y estar íntegramente controlada por gestos, traza el camino a seguir para obtener una aplicación completa utilizable en el ámbito de la realidad en control de tráfico aéreo.

10. DESARROLLOS FUTUROS

Retomando el punto anterior, la aplicación demostrativa ha sido conseguida, y el camino a seguir para la obtención de una aplicación utilizable en el mundo real ha sido indicado. Por tanto, para posibles continuaciones de este proyecto, podremos hablar de dos casos: mejoras aplicables a esta aplicación concreta, y aplicabilidad de la misma al mundo real.

En el primero de ambos terrenos, se ha comprobado que las posibilidades del dispositivo son numerosas; y, siguiendo en esa línea, sería posible profundizar en el tema del control por gestos. Tal y como se exponía en el punto 6.4.3 (*Glosario de funciones*), en el controlador del dispositivo (*CLeap*) se habían reservado algunos gestos que no eran utilizados, como el destinado al *zoom*. Por tanto, generar nuevos gestos e implementarlos a la aplicación sería posible (por ejemplo, incluyendo un mapa reescalable que se pudiera mover y ampliar, y que recibiera tráficos de una zona más amplia).

Respecto a la aplicabilidad de esta idea al mundo real, es indudablemente posible, pero con algunos *peros*. Con la precisión y robustez proporcionada por el dispositivo *LEAP Motion* ™ es posible realizar multitud de tareas de forma fiable, pero como cualquier tecnología nueva, su buen funcionamiento no puede asegurarse al 100%. Ofrece multitud de posibilidades y su ámbito de uso no se limita a actividades lúdicas, puede tener aplicaciones profesionales; pero el terreno del control de tráfico aéreo requiere una robustez y precisión que, hoy por hoy, no puede ser ofrecida por este dispositivo. Por tanto, una hipotética aplicación de la tecnología de control por gestos al ámbito aeronáutico pasaría por el desarrollo de un *hardware* muy robusto y preciso y plenamente certificado para operar de forma fiable en un sector tan crítico.

A pesar de todo ello, y como se ha indicado anteriormente, el camino a seguir está indicado; y es solo cuestión de desarrollar estas tecnologías hasta un punto en el que sean aplicables a cualquier ámbito, por sensible y crítico que sea.

11. PRESUPUESTO

Para conocer el coste total presupuestado para el proyecto, podremos dividir el cálculo del mismo en tres apartados concretos: materiales (*hardware*), licencias de *software*, y mano de obra.

En primer lugar, el coste total de los materiales se podría calcular con la relación mostrada en la figura 57.

Materiales	Cantidad (uds.)	Coste unitario (€/ud.)	Coste total (€)
Apple MacBook Pro ® de 13 pulgadas (incl. descuento para Educación)	1	1 054.00	1 054.00
Controlador *LEAP Motion* ™	1	109.46	109.46
Servidor Antena ADS-B	1	0.00	0.00
		TOTAL	**1 163.46**

Figura 57: Relación de materiales utilizados y sus precios

Cabe destacar que, como se comentó previamente, en el punto 2.1.1 (*Materiales y métodos: Equipo informático*), al tratarse de una aplicación multiplataforma, se podrían utilizar multitud de equipos de precios muy distintos; en este presupuesto se listará el utilizado por preferencia personal. Respecto al servidor de la antena ADS-B, no tendrá ningún coste, ya que se encuentra en abierto en *servantena.etsid.upv.es*.

Respecto al coste presupuestado para las licencias de *software*, se podrían resumir de acuerdo a la figura 58.

Licencia	Cantidad (uds.)	Coste unitario (€/ud.)	Coste total (€)
macOS ® *Mojave* ™ (10.14.x)	1	0.00	0.00
Java ™ SDK Version 8	1	0.00	0.00
LEAP Motion ™ *Java* ™ SDK v3.2	1	0.00	0.00
Apache NetBeans 11.1	1	0.00	0.00
Microsoft Office 365 ® (licencia personal de 1 año)	1	69.00	69.00
		TOTAL	**69.00**

Figura 58: Relación de licencias de *software* utilizadas y sus precios

Como se puede observar, la mayoría de licencias son gratuitas (ya que se utiliza *software* abierto o proporcionado gratuitamente para su libre uso); con la excepción de la licencia de *Microsoft Office 365* ®, para la que será necesaria la mínima licencia disponible (es decir, la personal de 1 año).

Finalmente, la mano de obra, tanto del alumno como del profesor tutor, se podrá resumir de acuerdo con la figura 59.

Concepto	Cantidad (h)	Coste unitario (€/h)	Coste total (€)
Obtención de datos	10	12.00	120.00
Ventana simple	30	12.00	360.00
Prueba de gestos	10	12.00	120.00
Diseño de nuevos gestos	40	12.00	480.00
Recopilación final de gestos	25	12.00	300.00
Aplicación de muestra de gestos	40	12.00	480.00
Mapa radar/Muestra datos ADS-B	15	12.00	180.00
Aviones ficticios	30	12.00	360.00
Funciones/Implementación gestos	40	12.00	480.00
Interfaz gráfica/Presentación	50	12.00	600.00
Pruebas	20	12.00	240.00
Redacción	50	12.00	600.00
Tutorías (Profesor Tutor)	20	30.00	600.00
		TOTAL	**4 920.00**

Figura 59: Relación de horas dedicadas y precios de mano de obra

Para la estimación de horas dedicadas por el alumno, se ha considerado que el proyecto debe realizarse en un total de 360 horas (es decir, 12 créditos ECTS, a 30 horas de trabajo por crédito), y se han distribuido de acuerdo a las estimaciones presentadas en el diagrama de *Gantt* de la figura 12 del punto 5 (*Implementación*); y para las del Profesor Tutor, unas 10 tutorías (a dos horas por tutoría).

Por tanto, el presupuesto final, contando los tres bloques anteriormente descritos, será el estipulado en la figura 60.

Concepto	Coste total (€)
Materiales	1 163.46
Licencias de *software*	69.00
Mano de obra	4 920.00
TOTAL (incl. IVA 21%)	**6 152.46**

Figura 60: Cantidad total presupuestada para la realización del proyecto

El coste total presupuestado para la realización del proyecto ascendería a SEIS MIL CIENTO CINCUENTA Y DOS EUROS CON SESENTA CÉNTIMOS, incluyendo el IVA del 21% por valor de 1067.78 €.

12. BIBLIOGRAFÍA

Documentación

- Documentación SDK (kit de desarrollo de *software*) para *LEAP Motion* ™ en lenguaje *Java* ™, versión 3.2 ("Leap Motion Java SDK v3.2 Documentation".)
- Foros de programación de Internet para resolución de problemas y adquisición de habilidades (principalmente *Stack Overflow* y *CodeRanch*)

Material didáctico de la Universidad

- Archivos *MATLAB* ® de la asignatura de Transporte, Navegación y Circulación Aérea, del Grado en Ingeniería Aeroespacial, sobre cálculo de distancias loxodrómicas y obtención de datos de *servantena*
- Material didáctico de la asignatura de Sistemas de Gestión de Vuelo por Computador, del Máster Universitario en Ingeniería Aeronáutica
- Material didáctico de las tutorías referentes a este proyecto impartidas por el Prof. Dr. D. Àngel Rodas Jordà

Artículos y documentos de Internet

- Agencia EFE. (11 jun. 2015). "Boeing prevé una demanda de 38.050 nuevos aviones en 20 años por valor de 5,6 billones de dólares." Efe.com
- Richardson, Nicole Marie. (28 may. 2013). "One Giant Leap for Mankind". Inc.com
- Leap Motion, Inc. (s.f.). "Leap Motion - Información". Facebook.com
- Oracle. (s.f.). "The History of Java ™ Technology". Oracle.com
- Oracle. (s.f.). "Oracle and Sun Microsystems". Oracle.com
- Apache Software Foundation. (s.f.). "About Apache NetBeans". Apache.org
- ENAIRE. (s.f.). "SACTA". Enaire.es
- Wendorf, Marcia. (31 mar. 2019). "How Gesture Recognition Will Change Our Relationship With Tech Devices". Interestingengineering.com
- Colgan, Alex. (9 ago. 2014). "How Does the Leap Motion Controller Work?". Leapmotion.com

APÉNDICE: MANUAL DEL USUARIO

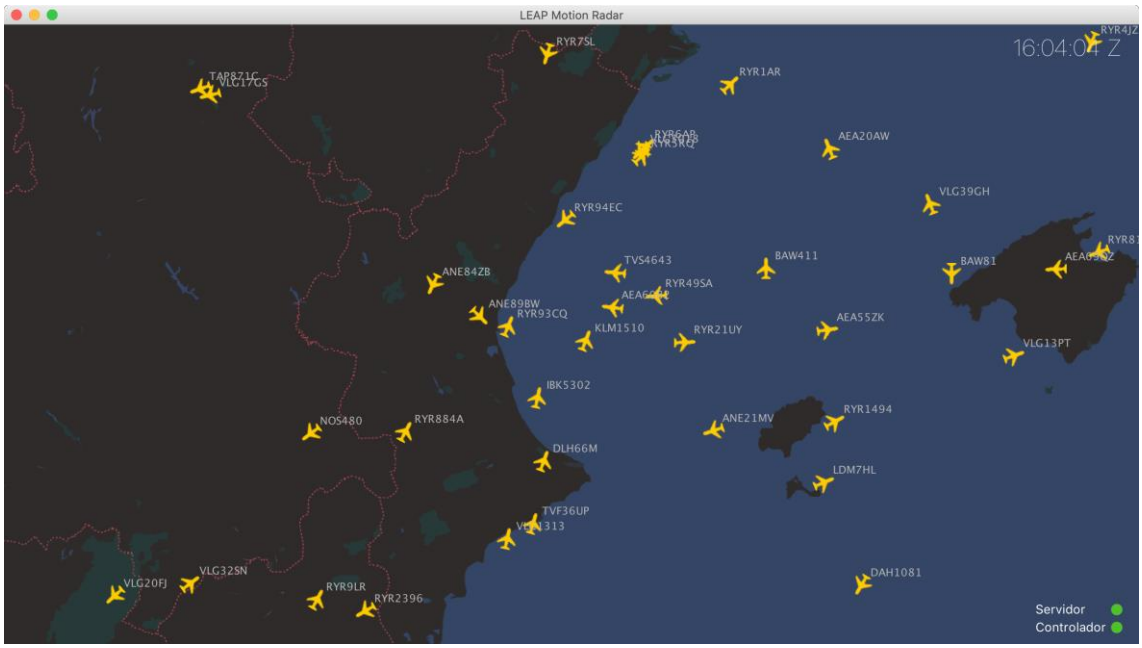
Bienvenido/a al manual de usuario de la aplicación *ATC Radar for LEAP Motion* ™. En las siguientes páginas encontrará una sencilla guía que le ayudará a conseguir un manejo adecuado de la aplicación, así como consejos para utilizarla.

Primeros Pasos

Para iniciarse en el manejo de la aplicación, se presentará a continuación una guía rápida de su puesta en marcha y configuración. No le llevará más de diez minutos. Para comenzar, necesitará los siguientes ítems:

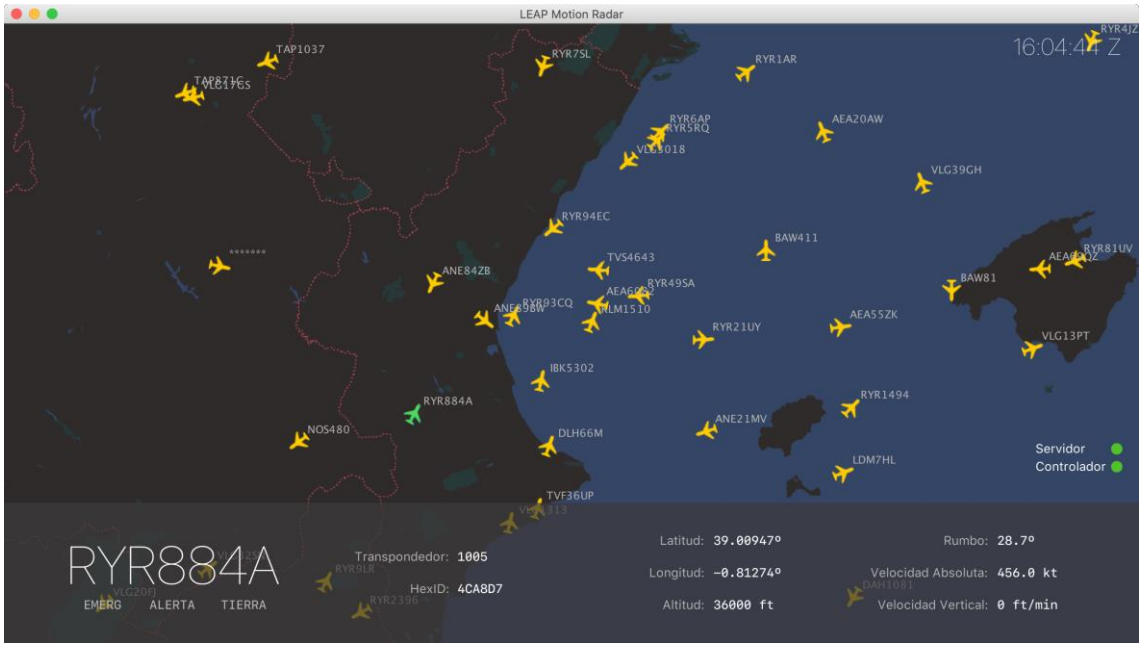
- Equipo informático con la aplicación correctamente instalada (se recomienda un procesador *Intel* ® *Core* ™ i5 como mínimo y 4 GB de memoria RAM, y una pantalla de 10.5 a 17.0 pulgadas)
- Dispositivo *LEAP Motion* ™ (*plug-and-play*, no se requiere instalación)
- Conexión a Internet (necesaria para representar los tráficos aéreos)

Una vez disponga de ellos, conecte el dispositivo al equipo, colóquelo entre el equipo y usted (con la luz verde encarada hacia usted y la parte negra brillante hacia arriba) y ejecute la aplicación. Se mostrará la ventana principal de la misma, donde podrá ver los tráficos reales representados sobre el mapa radar:



En la esquina inferior derecha podrá observar el estado del servidor y del dispositivo; y en la parte inferior central, se mostrarán mensajes puntuales que le ayudarán en todo momento utilizando la aplicación.

Extendiendo su dedo índice, pruebe a trazar un círculo alrededor de algún tráfico aéreo. Si lo selecciona correctamente, podrá ver desplegado un panel con la información detallada relativa al tráfico seleccionado, actualizada en tiempo real.



A continuación, pruebe a extender todos los dedos de la mano y realizar un gesto moviendo la mano hacia abajo con la palma abierta hacia abajo, y podrá observar como el panel se desplaza con su mano, hasta desaparecer completamente.

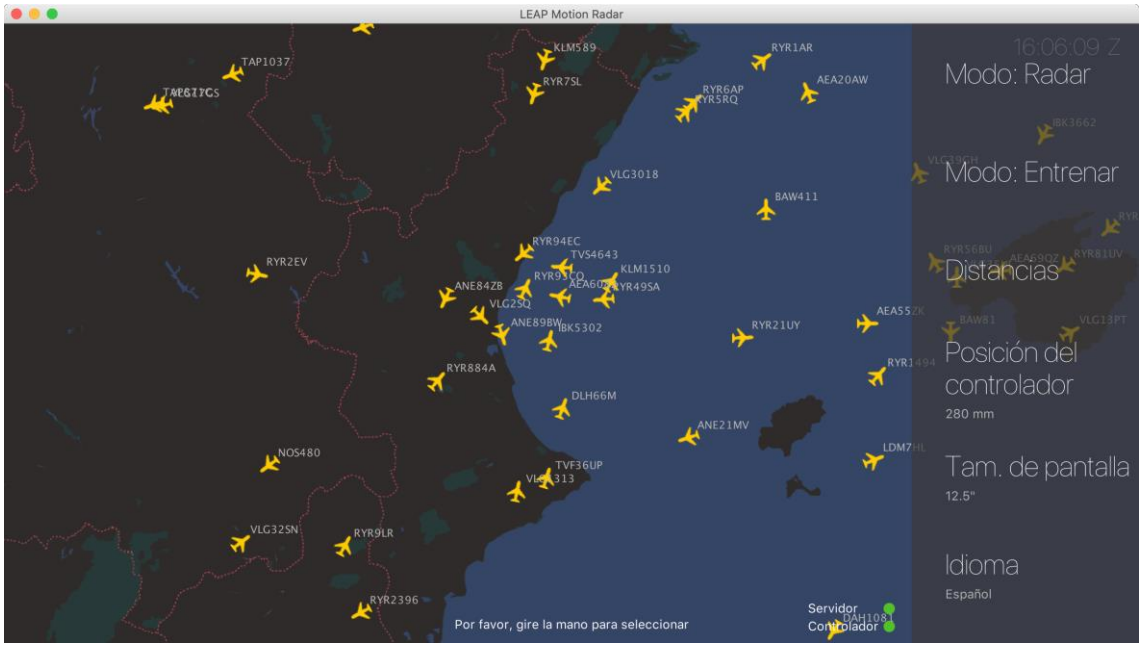
Repita esto cuantas veces desee, con cualquiera de los tráficos disponibles.

Si dentro de el "lazo" de selección se detectan varios tráficos, aparecerá un selector con todos ellos. Con la palma de la mano abierta boca abajo, desplácela horizontalmente hasta alcanzar el tráfico deseado, y gire la mano hasta que la palma quede hacia arriba para seleccionarlo. Para ayudarle en este proceso, el *callsign* del tráfico seleccionado se iluminará en color amarillo; y, a medida que gire la mano, este color se irá tornando verde, hasta completar la selección. Este sistema admite un máximo de cinco tráficos; si se detectaran más, podrá ver un mensaje indicándolo y pidiendo que vuelva a realizar la selección.

Con todo lo anterior, y una vez se haya familiarizado con el manejo, estará listo para utilizar la aplicación.

Menús y opciones

Todos los modos, opciones y ajustes de la aplicación se encuentran centralizados en el panel lateral de opciones. Para extraerlo, extienda su mano con la palma abierta y encarada hacia la izquierda (o hacia la derecha, si utiliza la aplicación con la mano izquierda), y desplácela hacia la izquierda (lo que conoceremos como gesto *swipe*). Podrá ver aparecer, a medida que desplace la mano, un panel lateral con seis etiquetas, desde el cual podrá controlar en cualquier momento todos los aspectos de la aplicación.



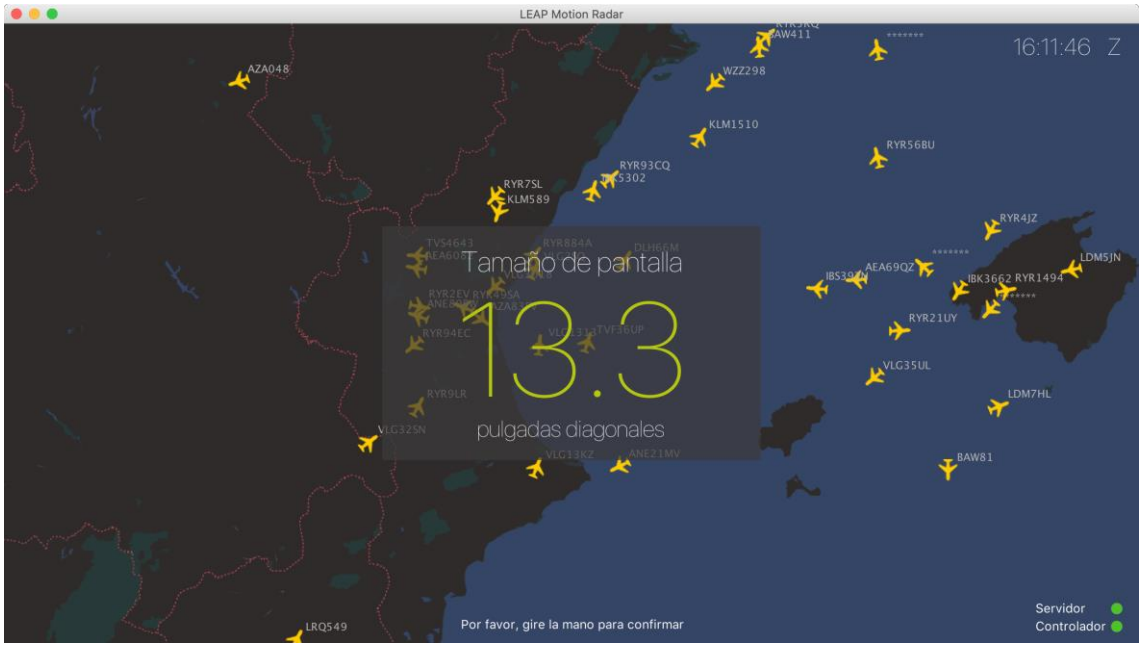
A continuación, manteniendo la mano extendida y con la palma hacia abajo, podrá desplazarse entre las distintas opciones. Para seleccionar una de ellas, bastará con girar la mano hasta encarar la palma hacia arriba (en toda la aplicación, el sistema de selección será el mismo, y cuando sea necesario girar la mano para confirmar, aparecerá un mensaje en la parte inferior indicándolo).

Se mostrarán seis elementos: los dos primeros serán los modos de la aplicación, *Radar* y *Entrenar*; el tercero, *"Distancias"*, accederá al asistente de cálculo de distancias en cualquiera de los dos modos; y finalmente, los tres últimos serán los ajustes de la aplicación, tanto de calibración del dispositivo como del idioma de la aplicación.

Calibración del dispositivo y selección de idioma

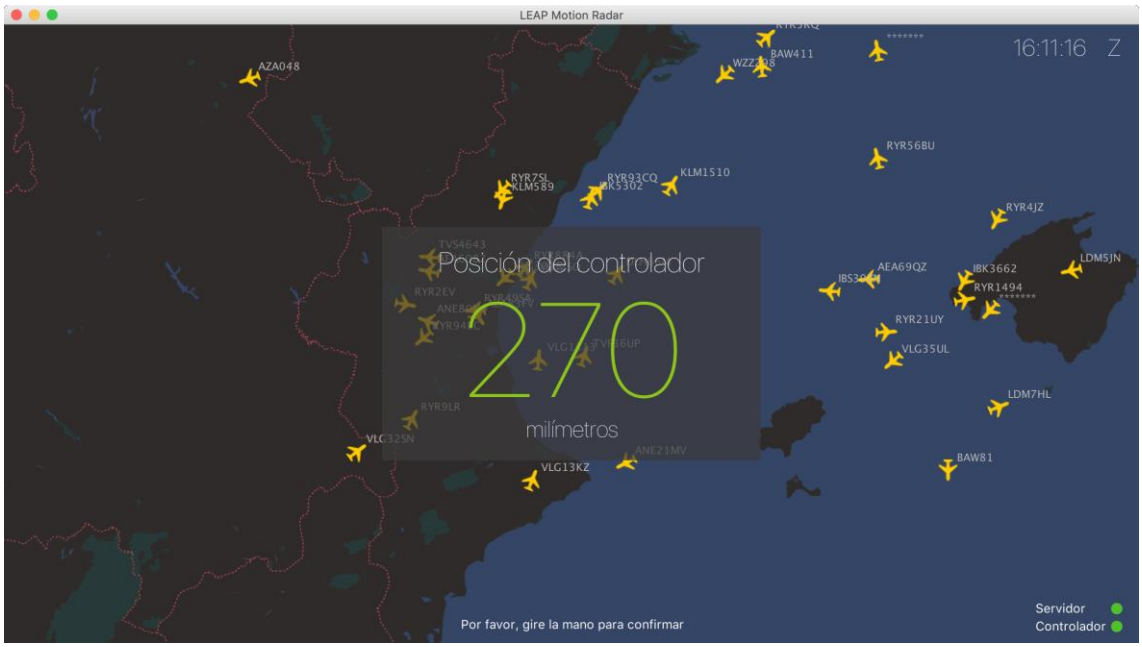
Como probablemente habrá podido observar en su primer uso de la aplicación, el cursor no coincide exactamente con el punto al que se está señalando. Esto es debido a que el dispositivo debe ser calibrado en relación al equipo en el que se está utilizando. Para realizar esto, deberá, en primer lugar, obtener el tamaño en pulgadas diagonales del monitor que está utilizando (información que puede conseguir en las instrucciones del mismo, o en la información del equipo, si está utilizando un ordenador portátil). Para que el funcionamiento sea óptimo, asegúrese de que la parte inferior de la pantalla esté a la misma altura que la parte superior del dispositivo. Posteriormente, mida la distancia que separa el dispositivo de la parte inferior de la pantalla en horizontal (en milímetros). Tenga en cuenta que esta aplicación soporta tamaños de pantalla de 10.5 a 17.0 pulgadas, y el dispositivo deberá estar situado con una separación de 200 a 400 milímetros.

Una vez tenga claros estos dos parámetros, acceda al panel de opciones y seleccione *"Tam. de pantalla"*. Gire la mano para confirmar, y manteniendo la palma hacia arriba, suba o baje la mano hasta llegar a su tamaño de pantalla.



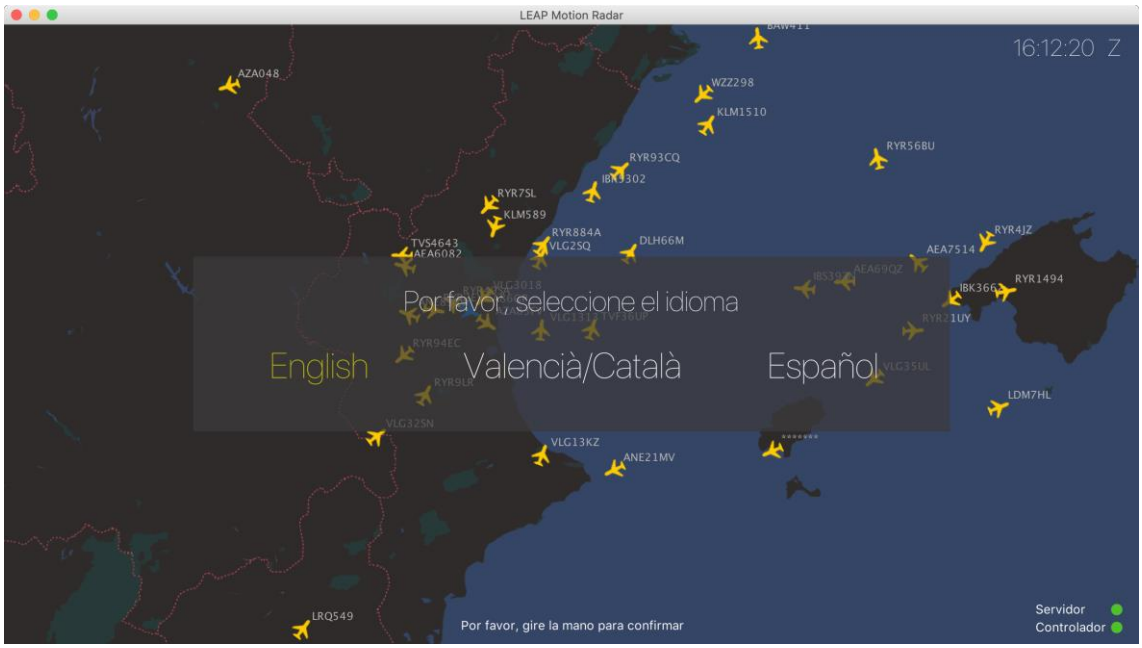
Para confirmar el valor, gire la mano nuevamente.

Repita el mismo proceso para introducir la posición del controlador en milímetros, seleccionando *"Posición del controlador"* en el menú de opciones. Aparecerá un selector similar al anterior para introducir la distancia en cuestión. De forma análoga, desplace la mano verticalmente hasta el valor deseado y confírmelo girando la mano.



Solo necesitará realizar la calibración una vez, ya que será guardada para los posteriores usos de la aplicación. Sin embargo, si cambia de pantalla o equipo, deberá introducir nuevamente sus características (si estas fueran distintas).

Si selecciona la opción *"Idioma"* en el panel de opciones, podrá acceder al selector del idioma de la aplicación. Desplazando horizontalmente la mano y girándola, podrá seleccionar el deseado.



Actualmente, la aplicación dispone de tres idiomas: inglés, valenciano/catalán y español.

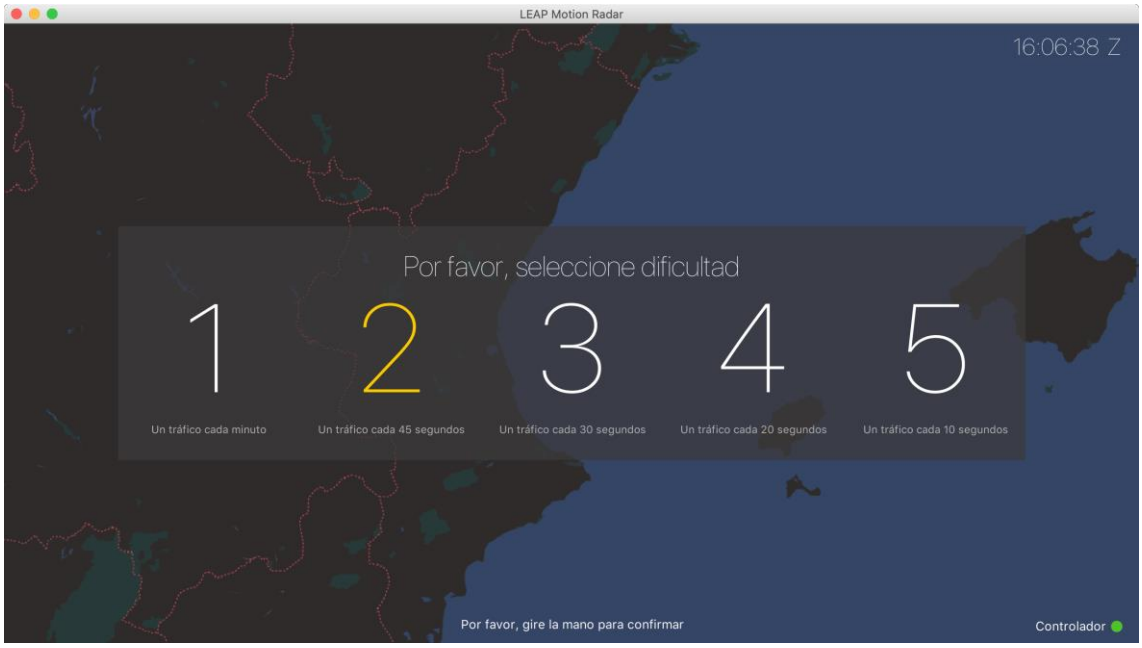
Una vez haya calibrado el dispositivo y seleccionado el idioma, podrá realizar un uso completo de la aplicación.

Modo Radar

El modo Radar es el predeterminado, en el que se inicia la aplicación. Sin embargo, se podrá acceder al mismo desde el panel lateral de opciones, seleccionando *"Modo: Radar"*. Su uso es muy sencillo: tal y como se ha explicado en el apartado Primeros Pasos, podrá seleccionar tráficos y mostrar su información. Si se seleccionaran varios, podrá elegir uno entre ellos; y con el panel de información mostrándose, podrá cerrarlo desplazando hacia abajo con la palma, o seleccionar otro/s tráficos sin necesidad de cerrarlo.

Modo Entrenar

La aplicación cuenta con un potente generador de tráficos sintéticos, para poder simular un uso real dirigiendo tráficos aéreos. Para acceder al mismo, seleccione *"Modo: Entrenar"* en el panel de opciones. A continuación, podrá observar que aparece un selector con los modos de dificultad en los que se podrá ejecutar el simulador.

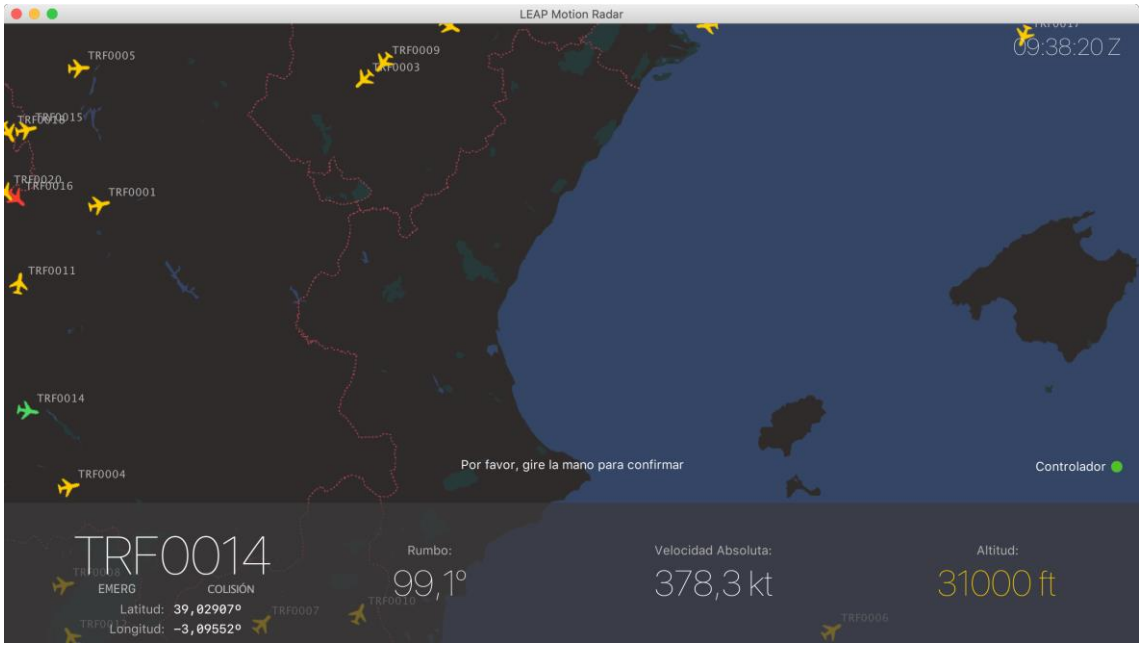


Se dispone de cinco modos:

- Modo 1 de dificultad, que mostrará un tráfico cada minuto
- Modo 2 de dificultad, que mostrará un tráfico cada 45 segundos
- Modo 3 de dificultad, que mostrará un tráfico cada 30 segundos
- Modo 4 de dificultad, que mostrará un tráfico cada 20 segundos
- Modo 5 de dificultad, que mostrará un tráfico cada 10 segundos

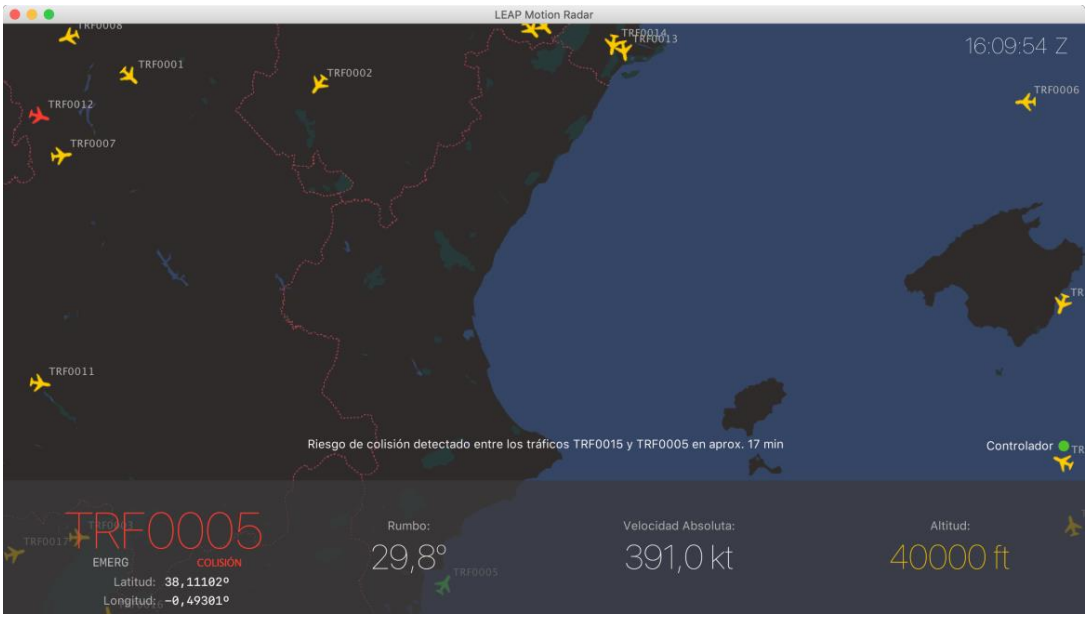
Además, los tráficos tendrán un 1% de probabilidades de ir en estado de emergencia.

Una vez seleccionado el modo, irán apareciendo vuelos aleatorios en el tiempo especificado, y el usuario podrá dirigirlos modificando sus parámetros de rumbo (en grados), velocidad absoluta relativa al suelo (en nudos), y altitud (en pies). Para cambiar los valores, bastará con seleccionar uno de los tráficos (con el mismo sistema que en el modo *Radar*) y aparecerá el panel de información con datos básicos sobre el tráfico y los tres parámetros que rigen su trayectoria (estando estos cuatro elementos colocados en dicho orden de izquierda a derecha).



Desplazando la mano horizontalmente con la palma hacia abajo, se podrá seleccionar entre los tres valores. Confirmando la selección sobre uno de los valores, se podrá subir y bajar la mano para cambiarlo (teniendo que girarla una vez más para confirmar el nuevo valor); y si se realiza el gesto de confirmación sobre la etiqueta del *callsign*, no se variará ningún valor y se cerrará el panel de información.

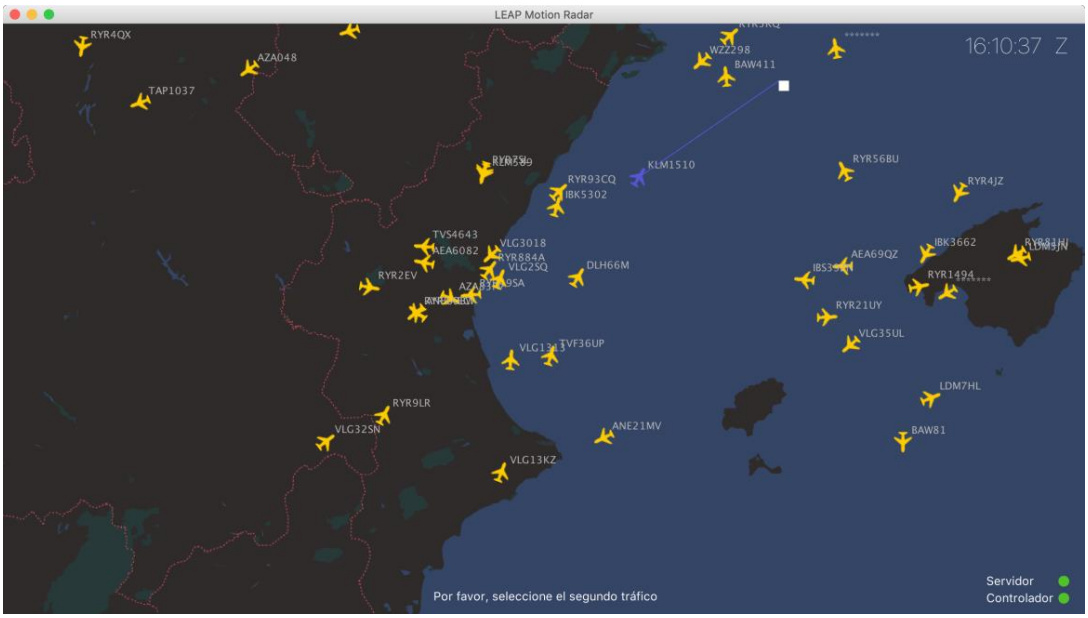
En tiempo real durante la simulación, la aplicación comprobará el riesgo de colisión entre los distintos tráficos (con operaciones matemáticas basándose en la trayectoria de los mismos). Cuando se detecte un riesgo de colisión, se mostrará un mensaje con los tráficos involucrados y el tiempo hasta que ocurra. Si el tráfico seleccionado tiene riesgo de colisión (o se encuentra en estado de emergencia), el *callsign* se mostrará en color rojo, y su icono parpadeará alternando el color rojo y el amarillo.



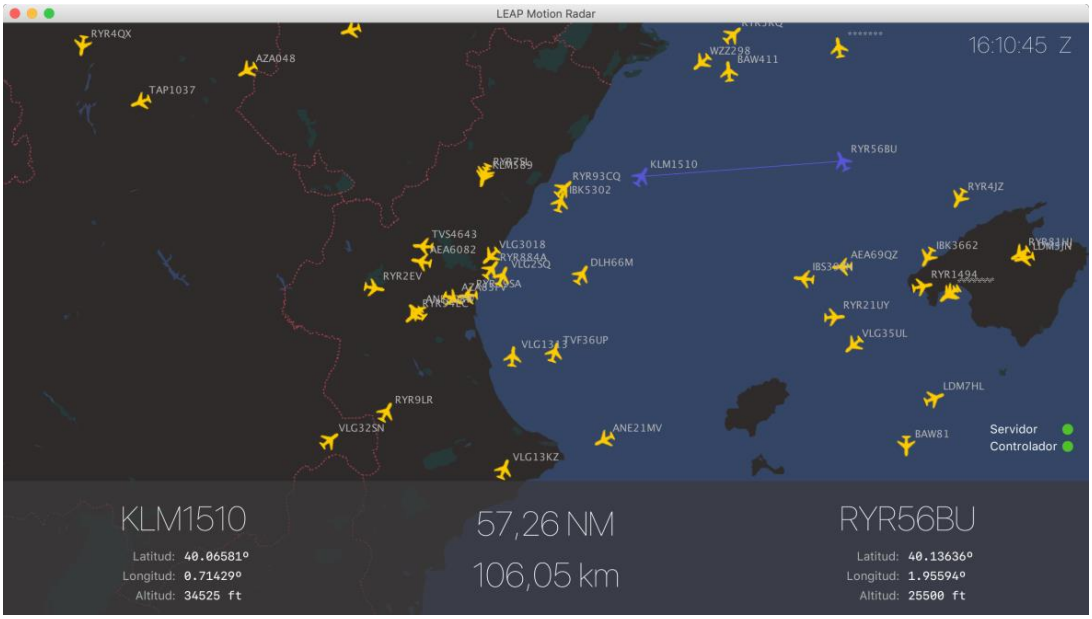
El objetivo de la simulación será aguantar el mayor tiempo posible controlando todos los tráficos (hasta un máximo de 50) y evitando riesgos de colisión; mejorando así su tiempo de respuesta ante alertas.

Asistente de cálculo de distancias

Para ayudar durante el uso de ambos modos, la aplicación cuenta con un asistente que permitirá calcular distancias entre dos tráficos cualquiera de los que se estén mostrando. Para acceder al asistente, bastará con seleccionar *"Distancias"* en el panel de opciones. A continuación, se mostrará un mensaje en la parte inferior pidiendo que se seleccione el primer tráfico. Una vez seleccionado (utilizando el sistema habitual), el icono de dicho tráfico se pondrá de color morado, y se pedirá que se seleccione el segundo.



Cuando se tengan ambos tráficos seleccionados, se mostrará una línea uniéndolos y un panel con información básica de ambos tráficos y su distancia en millas náuticas y kilómetros.



Las distancias y los datos se irán actualizando en tiempo real.

Para obtener la distancia entre otros dos tráficos, simplemente seleccione uno nuevamente y se le pedirá que seleccione el segundo. Para cerrar el panel, proceda de la misma forma que con el de información sobre un tráfico (haciendo un gesto hacia abajo), y para salir del asistente, vuelva a seleccionar el modo correspondiente (*Radar* o *Entrenar*) en el panel de opciones.

Este asistente estará disponible y será idéntico para ambos modos.

Especificaciones técnicas

Requisitos de *hardware*

- Monitor o pantalla de 10.5 a 17.0 pulgadas diagonales
- Puerto USB 3.0 o 2.0

Requisitos de *software*

- *Java ™ Runtime Environment Version 8* o superior instalado

Sistemas operativos admitidos

- Windows ® 7 o superior
- macOS ® 10.7 o superior
- Ubuntu Linux 12.04 LTS o superior

Historial de versiones

Versión 1.0

- Versión inicial de lanzamiento de la aplicación

Sobre *ATC Radar for LEAP Motion* ™

Esta aplicación ha sido desarrollada en la *Universitat Politècnica de València* en el marco de un Trabajo Final de Grado para el Grado en Ingeniería Aeroespacial.

LEAP Motion ™ es una marca registrada propiedad de *Ultrahaptics Ltd.*