PLATAFORMA PARA LA INTERACCIÓN CON DISPOSITIVOS DE ELECTROENCEFALOGRAFÍA.

UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

INGENIERÍA
EN INFORMÁTICA

PROYECTO FIN DE CARRERA

Plataforma para la interacción con dispositivos de Electroencefalografía.

Eduardo Aguilar Gallego

Septiembre, 2013

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

Departamento de Tecnologías y Sistemas de Información

PROYECTO FIN DE CARRERA

Plataforma para la interacción con dispositivos de Electroencefalografía.

Autor: Eduardo Aguilar Gallego

Director: Dr. Carlos González Morcillo

Septiembre, 2013

Eduardo Aguilar Gallego

Ciudad Real – Spain

E-mail: Eduardo.Aguilar@alu.uclm.es
Teléfono: 635 45 32 60

TRIBUNAL:

Presidente:

Secretario:

Vocal:

FECHA DE DEFENSA:

CALIFICACIÓN:

PRESIDENTE	**SECRETARIO**	**VOCAL**
Fdo.:	Fdo.:	Fdo.:

Resumen

Desde sus inicios, los mecanismos de Interacción Persona-Ordenador han ido evolucionando, buscando cada vez más un software «a medida del usuario», más simple de utilizar y que se adecue a sus necesidades. Esta forma de desarrollo ha acuñado el término Natural User Interface (NUI), o Interfaces de Usuario Naturales, que se adaptan al usuario y se vuelven «invisibles» a él, de una forma natural. Unos de los paradigmas más novedosos son los Interfaces Cerebrales, y como caso particular de estudio los basados en Electroencefalografía (EEG). En la EEG, la actividad cerebral del usuario se captura mediante un dispositivo y se envía al computador, donde se procesa y es asociada a acciones y procedimientos, de forma que puede controlarse el computador directamente con la actividad cerebral.

Este tipo de novedosos interfaces sufren de la falta de estandarización y soporte sencillo para el desarrollador de aplicaciones. Muchos de ellos son privativos, y el software incluido también lo es, de forma que es necesario utilizar una API cerrada que no puede ser estudiada o adaptada a necesidades concretas, además de limitar la compatibilidad entre dispositivos.

El presente Proyecto Fin de Carrera surge como solución a estos problemas, con el objetivo de crear una plataforma bajo sistemas GNU/Linux que permita la integración de diversos dispositivos de EEG y abstraiga al programador de los detalles concretos de implementación de los dispositivos, ofreciendo funcionalidad completa y de forma que todos se utilicen de forma estándar.

La Plataforma implementará un módulo de comunicación a bajo nivel, el tratamiento de las ondas, con capacidad de adaptación a diversos entornos de explotación, homogenización de dispositivos y capacidad de representación gráfica de valores y ondas, formando una solución completa para el desarrollador de interfaces naturales de electroencefalografía.

Para finalizar, se desarrollará a modo de demostración de las capacidades de la plataforma un juego de concentración y relajación que utilice de forma simultánea dos dispositivos de EEG y un controlador Arduino.

Abstract

Human-Computer Interaction mechanics have been evolving from their beginning, finding a simplier user-based software to fulfill his needs. A new term has grown, Natural User Interface (NUI), user-adapted interfaces that turn themselves invisible to the user in a natural way. One of newest paradigms are Computer-Brain Interfaces, with Electroencephalography-based interfaces as a particular study case. In Electroencephalography (EEG), the user neural activity is captured by EEG devices and sent to computer, where is processed and associated to actions and procedures, controlling computer directly by neural activity.

Such novel interfaces suffer from the lack of standardization and simple support to the application developer. Many of them are privative, with privative software too. The developer must use a privative API which cannot be studied or adapted to his own needs, besides limiting compatibility between devices.

The current Final Project arises to solve these problems. It aims at creating a platform under GNU/Linux that allows the integration of various EEG devices and abstracts the developer from the specifics of implementation, offering complete functionality so all devices are used as a standard.

Platform will provide a low-level communication module, neural wave processing, ability to adapt itself to different operating environments, device homogenization capacity and graphical representation of values and waves, getting a complete solution to develop EEG-based natural user interfaces.

Finally, an attention and meditation game will be developed to show the platform features and capabilities. This game will use two EEG devices at the same time and an Arduino controller.

Índice general

Índice de cuadros

Índice de figuras

Índice de listados

Listado de acrónimos

GNU	GNU is Not Unix
GCC	GNU C Compiler
GDB	GNU Debugger
OO	Orientación a Objetos
EEG	Electroencefalografía
PFC	Proyecto Fin de Carrera
NUI	Natural User Interface
IA	Inteligencia Artificial
RNA	Red Neuronal Artificial
IPC	Interacción Persona-Computador
FFT	Fast Fourier Transform

Agradecimientos

En primer lugar, quiero dar las gracias a mis padres y a mi hermano, por el enorme apoyo que me han dado todos estos años, sobre todo en los momentos más difíciles, sin desistir. A Alba, por su amor y apoyo en todo momento, animándome a no rendirme. Y a mi familia en general, siempre dispuesta a ayudarme.

Quiero dedicar un sincero agradecimiento a José Luís Seldas por la enorme ayuda ofrecida desinteresadamente con el circuito electrónico, y a José Félix Romero por su amistad y ayuda en la parte eléctrica.

Especial agradecimiento merece Carlos González Morcillo, por brindarme la posibilidad de realizar este proyecto, por su inestimable ayuda en todo momento, dedicación y amistad.

Eduardo Aguilar Gallego

A mis padres, a mi hermano y a Alba,
por creer siempre en mí.

Capítulo 1

Introducción

Los ámbitos en los que se aplican las diversas ramas de la informática crecen cada día más, debido principalmente a la tendencia de la sociedad de automatizar e informatizar la vida cotidiana; o mejor dicho, de integrar sociedad y tecnología. Prueba de ello son el éxito de las redes sociales, los smartphones y tablets que van sustituyendo al computador personal, el uso de computadores y software en medicina, mediante los cuales se realizan completas operaciones quirúrgicas o diagnósticos, e incluso el cine, que se ha servido ampliamente de la informática en la creación de efectos especiales.

Esta expansión igualmente ha supuesto una integración de la computación con otras doctrinas que inicialmente estaban más separadas de la rama técnica, como la sociología, la filosofía o la anatomía, como medio para obtener software y hardware mejor adaptados al ser humano, más naturales.

1.1 Contexto

Siguiendo esta evolución, la tendencia actual en el desarrollo de interfaces es el diseño orientado al usuario final, con software más fácil de usar, que sea más «familiar», así como hardware más ergonómico.

Surge el *diseño centrado en el usuario*, una filosofía que tiene por objetivo la creación de productos que resuelvan las necesidades del usuario con la mayor satisfacción y mejor experiencia posible, requiriendo un esfuerzo mínimo por su parte. En ello intervienen multitud de áreas de conocimiento, no sólo de la informática, sino también la Psicología, la Sociología y la Ergonomía. El objetivo final es reconocer perfectamente las necesidades e intereses del usuario para conseguir un producto con la mayor «usabilidad». La usabilidad es un neologismo, proveniente del inglés *usability* (*facilidad de uso*); podríamos definirlo pues como la facilidad con que las personas pueden utilizar una herramienta particular fabricada por humanos con el fin de alcanzar un objetivo concreto. En *The Psychology Of Everyday Things* [Nor88], se listan cuatro sugerencias de lo que un diseño debe ser para alcanzar un sistema (o interfaz) usable:

1. Hacer fácil de determinar qué acciones son posibles en cualquier momento.
2. Hacer los elementos del sistema visibles, incluyendo el modelo conceptual del sistema,

las acciones alternativas y sus resultados.

3. Hacer fácil de evaluar el estado actual del sistema.
4. Seguir asignaciones naturales entre las interacciones y las acciones requeridas; entre las acciones y el efecto resultante y entre la información que es visible y la interpretación del estado del sistema.

Así, en el desarrollo de productos software es muy importante considerar la intervención del usuario. De todo lo descrito hasta ahora se encarga una rama de la informática denominada **Interacción Persona-Computador**, del término inglés *Human-Computer Interaction* (HCI).

Este acercamiento al usuario para desarrollar «software a medida», que sea más simple de utilizar cada vez y se adecúe mejor a sus necesidades, ha ido acuñando en el ámbito de la Interacción Persona-Computador un nuevo término, denominado en inglés Natural User Interface (NUI) o Interfaces de Usuario Naturales. Este término se refiere a interfaces de usuario que, además de ser adaptadas y usables por el usuario, son «naturales» a él; es decir, son interfaces de usuario prácticamente invisibles. Esta familia de técnicas emplean los mecanismos de interacción naturales de los humanos.

El campo de las interfaces naturales ha evolucionado exponencialmente en los últimos años, aunque sus bases se establecieron en los años 80 de la mano de Steve Mann [Man01]. Mann desarrolló una serie de estrategias de interfaces de usuario utilizando interacción natural con el mundo mundo real como alternativa a las existentes *interfaces por línea de comandos* (CLI) o *interfaces gráficas de usuario* (GUI). Mann se refiere a su trabajo como "Natural User Interfaces"(Interfaces de Usuario Naturales) o "Direct User Interfaces"(Interfaces de Usuario Directas). Una de estas interfaces que desarrolló Mann fue su *EyeTap*, un dispositivo montado en frente del ojo que actúa de cámara, grabando todo lo que el ojo ve y mostrándolo en un computador.

Figura 1.1: Evolución de las interfaces de usuario.

En las CLIs, el usuario debía aprender formas artificiales de entrada, como el teclado, y una serie de comandos codificados en una sintaxis estricta para llevar a cabo las acciones que deseaba. Con la llegada de las GUIs y el ratón, el usuario podía aprender de forma más simple movimientos y acciones, y explorar más la interfaz. Además, las interfaces gráficas se fundamentan en la *metáfora*, una representación gráfica, iconos o acciones y procedimientos que simulan en la interfaz una acción perteneciente a otro dominio. Un ejemplo simple es la metáfora de «escritorio» o «arrastrar». Sin embargo, no todos los sistemas implementan las mismas metáforas o difieren de unos sistemas a otros. En las NUIs, todos estos problemas desaparecen.

En 2006, Christian Moore estableció una comunidad abierta con el objetivo de expandir y desarrollar tecnologías NUI. Desde entonces, han aparecido nuevos tipos de interfaces naturales y especialiado y ampliado otros muchos.

El presente Proyecto Fin de Carrera trabaja con un tipo de interfaz NUI concreto: la Electroencefalografía (EEG). La EEG se basa en la captura y grabación de la actividad cerebral a lo largo del cuero cabelludo. Esta actividad cerebral es medida como fluctuaciones o diferencias de voltaje, resultado de los flujos de corriente iónicos de las neuronas del cerebro. En el contexto que nos interesa, hablamos de interfaces naturales cerebrales no invasivas.

Figura 1.2: Dispositivos de EEG para interfaces naturales.

En este tipo de interfaces, el usuario interacciona con el sistema mediante su actividad cerebral. Para leerla, debe colocarse un pequeño dispositivo, que puede tener forma de gorro, casco o auriculares (véase figura 1.2), que cuenta con una serie de sensores que miden las ondas que el cerebro produce, diferenciando varios tipos de onda (**Alpha**, **Beta**, **Gamma**, **Delta**, **Theta** y **Mu**). La información obtenida por el dispositivo se envía al computador y, una vez procesada, se puede entrenar el sistema y asociar patrones de pensamiento a acciones y procedimientos. De esta forma, nuestra forma de interactuar y controlar la aplicación se realiza simplemente con el pensamiento.

La aplicación de la EEG en el campo de las interfaces naturales es relativamente reciente, ya que hasta hace poco se restringía a usos clínicos y de investigación médica. Como

interfaces destacables, podemos señalar *The Audeo* [Cor08], producto de Ambient desarrollado en 2008 que permite al usuario utilizar las señales neurológicas del cerebro asociadas al control de las cuerdas vocales para poder comunicarse sin necesidad de hablar. En 2009 se comercializó el primer sistema de ortografía basado en EEG: intendiX [gmeG09]. Además de escribir texto mediante interacción cerebral, el usuario puede activar alarmas, hacer que el computador pronuncie el texto escrito, copiarlo en un e-mail o incluso mandar comandos a dispositivos externos. Como se comentará más adelante en la sección 2.1.3, desde 2006 han ido apareciendo también una serie de dispositivos comerciales al alcance de cualquier usuario que han contribuido en gran medida a la expansión de las interfaces naturales mediante EEG.

1.2 Motivación

A pesar del gran avance que ha tenido la EEG y los grandes hitos conseguidos, aún se encuentra en un estado «precario» en cuanto a difusión, soporte y, sobre todo, estandarización. Aún no ha pasado ni una década desde la aparición de los primeros dispositivos de Electroencefalografía comerciales, lo que deriva en que cada fabricante establezca sus propios estándares y normativas, y tengan un soporte limitado de los sistemas operativos.

Este es un gran problema, pues acotan y, sobre todo, condicionan el planteamiento y desarrollo de nuevos proyectos, ya sea por limitación del dispositivo, por el protocolo de comunicación, por sistemas operativos no soportados... Es especialmente relevante el deficiente soporte que que existe en sistemas operativos de código abierto, cuya API habitualmente no está desarrollada y mantenida por el fabricante.

Cuando un usuario compra un dispositivo EEG de cualquier fabricante disponible, está obligado a utilizar la API proporcionada, sin posibilidad de estudiarla o adaptarla a sus necesidades concretas. El usuario trabaja solo utilizando las llamadas de la interfaz que se le muestra; en ocasiones si se muestra cierto contenido de la API pero se limita la mejor funcionalidad a la solución privada. Esto además evita la compatibilidad entre dispositivos.

Otro problema es que no todas las interfaces EEG proporcionan medios para entrenar el dispositivo y adecuarlo a cada usuario. Se ofrece, en muchos casos, un dispositivo «genérico» que se supone adecuado para la mayor parte de usuarios. El problema en este caso es evidente: es imposible cubrir todo el espectro de usuarios, permitiendo que el dispositivo funcione de forma óptima para todos ellos, pues cada persona piensa de forma distinta.

En definitiva, no existe ninguna solución de código abierto multidispositivo que abstraiga de todos esos detalles concretos de implementación y proporcione una interfaz unificada para trabajar con cualquier dispositivo y su funcionalidad, personalizable para cada usuario del sistema, y que además dé soporte a sistemas de código abierto.

1.3 Propuesta

A raíz de estas motivaciones surge la presente propuesta de Proyecto Fin de Carrera. Se plantea diseñar e implementar una plataforma bajo sistemas GNU/Linux que permita la integración de diversos dispositivos de EEG y abstraiga al programador de todos los detalles concretos de implementación de los dispositivos.

Deberá contar con las siguientes características:

- Debe ser capaz de tratar con varios dispositivos de forma simultánea.
- Los dispositivos pueden ser de diferente modelo y/o fabricante.
- Debe encargarse de la comunicación con el dispositivo y tratamiento de las señales de onda recibidas.
- Independientemente del modo de comunicación computador-dispositivo, debe presentar una interfaz de comunicaciones común y genérica.
- Debe permitir personalizar el dispositivo, ya sea implementando diferentes módulos de entrenamiento o configuración de parámetros.
- Debe proporcionar retroalimentación al usuario y un modo de depuración para el desarrollador, donde muestre los valores de entrada del dispositivo y los valores transformados.
- Debe basarse en una arquitectura modular y extensible que permita añadir soporte a nuevos dispositivos.
- Debe poder ejecutarse en cualquier distribución GNU/Linux.
- Debe incluir un demostrador soportando al menos un tipo de dispositivo, con la ejecución de varias instancias simultáneas. En concreto se plantea la construcción de un juego demostrador con dos dispositivos EEG y un controlador Arduino.

Estas características y objetivos se detallan en el 'Capítulo 3. Objetivos'.

1.4 Estructura del documento

Este documento se ha estructurado según las indicaciones de la normativa de Proyecto Fin de Carrera de la Escuela Superior de Informática de Ciudad Real de la Universidad de Castilla-La Mancha, contando con los siguientes capítulos:

- **Capítulo 2. Antecedentes**
  En este capítulo se estudian los conocimientos que han sido necesarios adquirir para la realización de este Proyecto Fin de Carrera, así como sus áreas de aplicación.

- **Capítulo 3. Objetivos**
  En este capítulo se da primeramente una visión general del sistema, y posteriormente se especifican los subobjetivos con un mayor nivel de detalle, desglosándolos en requisitos funcionales propuestos para su consecución.
- **Capítulo 4. Método de trabajo**
  En este capítulo se analiza la metodología de desarrollo escogida para el desarrollo de la plataforma. También se listan y describen los recursos empleados, tanto hardware como software.
- **Capítulo 5. Arquitectura**
  En este capítulo se describe el diseño e implementación del sistema, detallando los problemas surgidos y las soluciones aportadas. Se estructura acorde a los módulos y submódulos que componen el sistema, dando primero una visión general y funcional del módulo, y pasando posteriormente a los detalles técnicos y de implementación.
- **Capítulo 6. Evolución y costes**
  En este capítulo se describe la evolución del sistema durante su periodo de desarrollo, detallando las etapas e iteraciones realizadas. Igualmente se aporta información relacionada con el coste económico y el análisis de rendimiento.
- **Capítulo 7. Conclusiones y propuestas**
  En este capítulo se hace un resumen a modo de conclusión final indicando las metas que se han alcanzado. Igualmente se realiza un análisis resumido sobre las posibles líneas de trabajo futuras, y una conclusión personal.

Capítulo 2

Antecedentes

La consecución de los objetivos propuestos en este Proyecto Fin de Carrera requieren un conocimiento previo, específico en diversas áreas.

En este capítulo se explicarán de forma detallada las áreas y principales conceptos estudiados a lo largo del Proyecto para obtener el conocimiento necesario para su realización. Se estructura el capítulo en tres secciones, correspondientes a las tres grandes áreas que abarca el proyecto: Electroencefalografía, Interfaces Naturales e Inteligencia Artificial.

2.1 Natural User Interface (NUI)

A menudo, el término *natural* es entendido como la imitación del *mundo real* [WW11]. Sin embargo, en el ámbito de las interfaces naturales es una filosofía de diseño que permite la creación de nuevas interfaces de usuario y paradigmas de interacción *invisibles* para el usuario. Así, podremos construir interfaces mediantes las cuales el usuario utilice el sistema por comandos de voz, gesticulación, dispositivos móviles, y otros medios que podamos imaginar.

Cada día la tecnología avanza más rápido, por lo que cada vez aparecen interfaces naturales más avanzadas, como la Electroencefalografía (EEG), que permite la interacción persona-computador a partir de la actividad cerebral. Consecuentemente, el potencial de estas nuevas tecnologías permite crear experiencias más cercanas a las experiencias humanas, optimizar la progresión de usuario novel a usuario experto y satisfacer las necesidades del usuario de forma más natural.

Por otra parte, el término natural también se refiere al comportamiento humano y su experiencia en el uso del sistema, no que la interfaz sea producto de procesos orgánicos. Esta conclusión es el resultado final de un diseño riguroso, aprovechando el potencial de tecnologías más modernas para calcar las capacidades humanas.

El objetivo de crear una interfaz natural es aprovechar los grados de libertad que proporcionan, diseñando un nuevo paradigma propio.

2.1.1 Definición

Las Interfaces de Usuario Naturales son sistemas de interacción persona-computador mediante el cual el usuario opera con el computador mediante acciones intuitivas y naturales, como si no hubiese ningún elemento ajeno a él; también se dice que son interfaces prácticamente invisibles o que se vuelven invisibles con cierto número de interacciones. Esto además implica que el usuario es capaz de utilizar la interfaz prácticamente sin necesidad de entrenamiento.

Las interfaces de usuario naturales son muy intuitivas, un hecho deseable en ellas. Generalmente también son flexibles, permitiendo al usuario personalizar la interfaz acorde a sus gustos y/o necesidades, haciéndola más eficiente. También las interfaces naturales son fluidas. Permiten al usuario usar la interfaz con la sensación de que no están utilizando ninguna en realidad. Con ello, el usuario utiliza el sistema de forma más eficiente y obtiene un mayor rendimiento de trabajo que en el caso que no utiliza interfaces naturales.

Para conseguir estos objetivos, podemos establecer una especie de *Guía de estilo* para obtener el resultado deseado. Los siguientes puntos no suponen una guía completa, pero sí suficiente para entender los principales objetivos de diseño a alcanzar [WW11].

Un sistema de Interacción Persona-Computador (IPC) natural debe:

- Crear una experiencia tal que el usuario experto sienta la interfaz como una extensión de su cuerpo.
- Crear una experiencia tal que el usuario inexperto o novel se sienta usuario experto.
- Crear una experiencia auténtica al medio, no copiar algo del mundo real.
- Construir la interfaz de usuario considerando el contexto, las indicaciones visuales, la retroalimentación al usuario de sus acciones, y los métodos de entrada salida.
- Evitar el error de copiar paradigmas existentes de interacción; cada interfaz natural de usuario debe tener su propio paradigma de interacción.

Un sistema de IPC natural debería:

- Hacer olvidar al usuario lo que significa natural, simplemente interactuar.

Un sistema de IPC natural podría:

- Aprovechar talentos y habilidades ya aprendidas por el usuario, como podría ser la integración de otras experiencias de interacción en las cuales el usuario ya es experto.

2.1.2 Ley de Fitts

La ley de Fitts es muy importante en el campo de la Interacción Persona-Computador (IPC). Es un modelo de movimiento que precie el tiempo necesario para recorrer una distancia, como una función de proporción entre la distancia al objetivo y su tamaño; es empleada como método de validación de la interacción.

La función de la Ley de Fitts es:

$$T = a + b * \log_2(1 + \frac{D}{W}) \tag{2.1}$$

donde T es el tiempo, a y b son constantes que representan el tiempo de inicio/parada del dispositivo en segundos y la velocidad inherentes del mismo respectivamente, D es la distancia desde el punto inicial al objetivo y W es el ancho del objetivo.

En el diseño de interfaces, el plazo de ejecución (T) es muy importante. Los usuarios son impacientes por naturaleza y no quieren interfaces lentas, navigacionalmente hablando. La Ley de Fitts proporciona algunas guías para hacer las interfaces más fáciles de usar, ya que como comentabamos anteriormente, la interfaz fácil de usar proporciona sensación de ser más natural.

2.1.3 Tipos de Interfaces Naturales

Pantalla táctil

Las interfaces naturales de pantalla táctil permiten al usuario interactuar con las aplicaciones de forma más intuitiva que las interfaces basadas en cursor: en lugar de mover el cursor por medio de un raton, ya sea para seleccionar un fichero, hacer click para abrirlo o arrastrarlo, el usuario toca la pantalla simplemente en el punto deseado, como si tocase el fichero. Mediante toque directo sobre su superficie, permite la entrada de datos y órdenes al dispositivo, y a su vez muestra los resultados introducidos previamente, actuando como periférico de entrada y salida de datos.

La interacción en una pantalla táctil puede ser por toque simple o por multi-toque (multi-touch). En el primer caso, las pantallas solo detectan el contacto en un punto a la vez, la del primer o último dedo que toque la pantalla. En el segundo caso, se detectan varias pulsaciones pertenecientes a varios dedos, de forma que pueden reconocerse diversos patrones (más o menos complejos) para implementar operaciones diferentes, como zoom sobre una imagen o rotación.

Las pantallas táctiles también puede estar construidas con diferentes tecnologías:

- Resistiva
- Capacitiva
- Rejilla infrarroja
- Proyección acrílica infrarroja
- Imágen óptica
- Reconocimiento de pulsación acústica

Figura 2.1: Selección de imágenes de forma táctil.

Multitáctil

En su inicio, las pantallas táctiles sólo permitían un solo punto de contacto en su superficie; denominadas de un toque o táctiles simplemente. Sin embargo, esta limitación tecnológica también limitaba la experiencia de interacción natural. La evolución natural era pues permitir varios puntos de contacto de modo que la mano completa pudiese interaccionar con la interfaz.

De esta forma surgen las pantallas multitáctiles. En ellas, se permite al usuario interaccionar con el dispositivo utilizando más de un dedo a la vez y con ambas dependiendo del comando a realizar. Estas pantallas suelen permitir también que interaccionen varios usuarios al mismo tiempo, algo especialmente útil en grandes escenarios como paredes interactivas.

La detección de varios puntos de contacto y su procesamiento se realiza mediante un sensor ASIC (*Application-Specific Integrated Circuit*, o Circuito Integrado Específico de la Apliación) integrado en la superficie táctil. Generalmente, el sensor ASIC es fabricado por una compañía diferente a la de la pantalla, y luego son combinados. En los últimos años se ha expandido mucho la industria «multitáctil», con sistemas diseñados desde el usuario doméstico a las grandes multinacionales.

Tipos de pantallas táctiles

De todos los tipos de pantallas comentados en el apartado anterior, las más importantes son las pantallas resistivas, capacitivas y de reconocimiento de pulsación acústica.

Las *pantallas táctiles resistivas* consisten en un panel de cristal recubierto de una capa metálica resistiva y otra conductiva. Estas dos capas están separada una de otra por un espa-

cio. Por encima de ambas, se encuentra una última capa resistente a arañazos que protege las anteriores. Entre las dos capas primeras (resistiva y conductiva) se hace circular una corriente eléctrica; cuando el usuario toca la pantalla, las dos capas hacen contacto en ese punto. En ese momento se produce un cambio en el campo eléctrico en ese punto y la unidad de proceso calcula las coordenadas en la pantalla. Una vez que las coordenadas son conocidas, el driver de pantalla traduce la posición para que pueda ser utilizada por el sistema operativo.

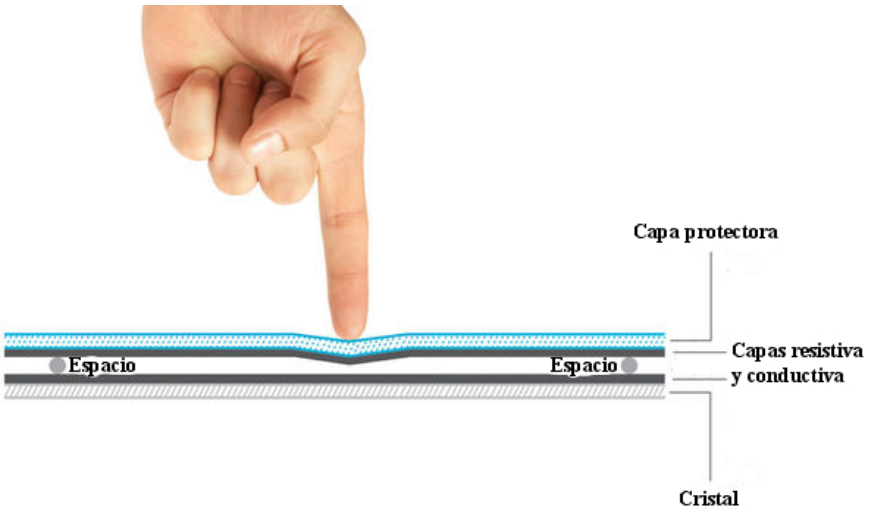


Figura 2.2: Pantalla táctil resistiva.

En las *pantallas táctiles capacitivas*, se sitúa una capa que almacena carga eléctrica en el cristal del monitor. Cuando el usuario toca la pantalla, parte de la carga se transfiere al usuario, por lo que la carga en la capa capacitiva de la pantalla disminuye. Esta disminución es medida mediante unos circuitos situados en los bordes y enviada a la unidad de proceso que, a partir de las diferencias de carga, calcula de forma precisa el lugar de la pulsación. De nuevo, las coordenadas son enviadas al driver de pantalla y éste traduce la posición para que pueda ser utilizada por el sistema operativo. La principal ventaja de este sistema es que proporciona una imagen más clara al ser capaz de transmitir el 90 % de la luz desde el monitor; el sistema resistivo transmite el 75 %.

En las *pantallas táctiles de reconocimiento de pulsación acústica*, se sitúan dos transductores (uno emisor y otro receptor) en los ejes X e Y del cristal del monitor. Se sitúan en él también reflectores encargados de reflejar una señal eléctrica enviada de un transductor a otro. Cuando el usuario toca la pantalla, altera la señal eléctrica y con ello la onda, y el transductor receptor lo detecta al instante, calculando la coordenada (x,y) de la pulsación. Después procede igual que los anteriores. Este tipo de pantallas táctiles tienen la mejor iluminación, el 100 %, al no haber ninguna capa metálica en la pantalla.

Figura 2.3: Pantalla táctil capacitiva.

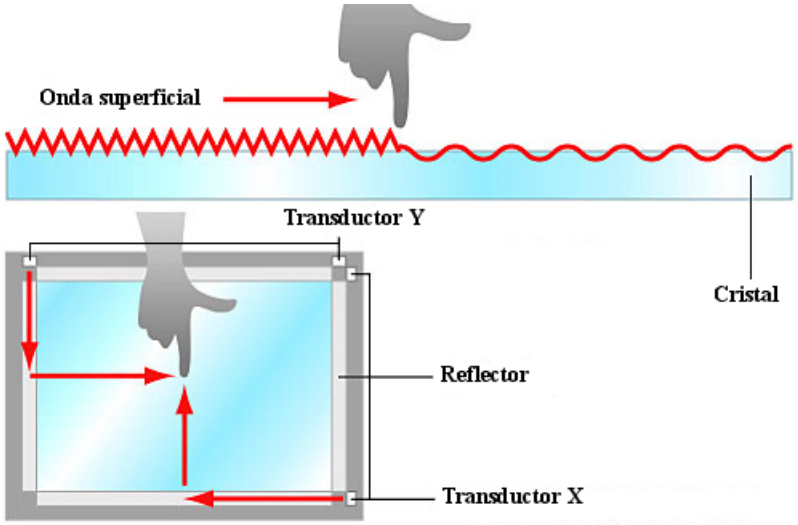


Figura 2.4: Pantalla táctil de reconocimiento de pulsación acústica.

Reconocimiento de signos

El reconocimiento de signos es un ámbito de la informática cuyo objetivo es interpretar los gestos humanos utilizando algoritmos matemáticos. Los gestos pueden ser ocasionados por cualquier movimiento o estado corporal, comúnmente la cara o las manos. La mayoría de aproximaciones utilizan cámaras y algoritmos de visión por computador para interpretar los signos. En estos sistemas, el usuario interacciona a través de gesticulación captada por una o varias cámaras. Estas cámaras captan el movimiento y lo convierten a representación 2D; si el movimiento realizado coincide con alguno de los patrones almacenados, ejecutará la orden asociada.

El reconocimiento de signos puede interpretarse como una forma de que los computadores «entiendan» el lenguaje corporal humano, mejorando así la comunicación entre computador y ser humano, que actualmente se encuentra limitada a las interfaces de texto o gráficas, que utilizan el teclado y el ratón.

El objetivo es permitir al usuario la comunicación con el sistema de una forma tan natural como el lenguaje corporal que emplea constantemente, sin ningún tipo de dispositivo mecánico. Este paradigma hace que los dispositivos de entrada convencionales e incluso otros más modernos como las pantallas táctiles, sean redundantes. Ejemplos representativos de este tipo de interfaces naturales son Kinect o PlayStation Move.

Tipos de gestos

Se distinguen dos tipos de gestos: gestos ‘offline’ y gestos ‘online’.

- Gestos ‘Offline’: Son aquellos gestos procesados después de la interacción del usuario con el objeto objetivo. Por ejemplo, la activación de un menú después de seleccionarlo.
- Gestos ‘Online’: Son aquellos gestos de manipulación directa del objeto. Suelen ser movimientos de escalado o rotación de un objeto tangible.

Dispositivos de entrada

Se necesitan diversas herramientas para capturar los movimientos del usuario y determinar cuáles están realizando. Aunque la mayoría de trabajos e investigaciones se enfocan en la tecnología de reconocimiento de gestos a través de vídeo e imagen, no son los únicos medios y herramientas existentes.

- Guantes inalámbricos: Proporcionan al sistema la información relativa a la posición y rotación de las manos. Utiliza para ello sistemas de inercia y sistemas magnéticos –acelerómetros-. Además, algunos guantes pueden detectar la flexión de los dedos con un alto grado de precisión (con un error de unos 5-10 grados), o incluso provocar estímulos hápticos que simulan el tacto.

Figura 2.5: Guantes inalámbricos para reconocimiento de signos.

- Cámaras con detección de profundidad: Estas cámaras especializadas pueden generar un mapa de profundidad de aquello que capturan a través de la lente, y utilizar esos datos para aproximar una representación en tres dimensiones de lo que están «viendo». Estos dispositivos son muy efectivos para detectar cualquier gesto o signo generado con las manos.
- Cámaras estéreo: Utiliza dos cámaras, las cuales conocen cada una su relación con la otra, para aproximar una representación 3D en la salida. La relación entre cámaras se puede obtener mediante emisores infrarrojos. Si ahora se combina con medidas de movimiento (visión 6D o Estereoscopía), los gestos pueden detectarse directamente con las cámaras.
- Gestos basados en controlador: Estos dispositivos actúan como una extensión del cuerpo humano; cada gesto realizado, o parte del movimiento realizado puede ser capturado por software. Para ello, capturan los cambios en la aceleración y orientación de los dispositivos a lo largo del tiempo. Y como en la mayoría de casos, en un nivel inferior trabajan con acelerómetros, giroscopios y brújulas u otros sensores que transformen movimiento en señales.

  Por otro lado, en este tipo de dispositivos es importante compensar el error producido por el temblor humano y el movimiento involuntario. Lo más común suele ser que implemente cierta comparación de la entrada con patrones de movimiento bien conocidos y entrenados, o establezca umbrales de movimiento.
- Cámara única: El dispositivo de entrada más simple, una cámara 2D estándar, que puede ser empleada para reconocer signos basándose en algoritmos de reconocimiento basados en imagen. El reconocimiento de signos por software utilizando una cámara 2D puede es muy extenso y bastante preciso, útil para entornos en los que no puede

Figura 2.6: Reconocimiento de gestos basados en controlador (Wii).

aplicarse otra técnica. Detectan con precisión gestos de las manos, signos formados con los dedos o incluso la posición de los mismos en el espacio.

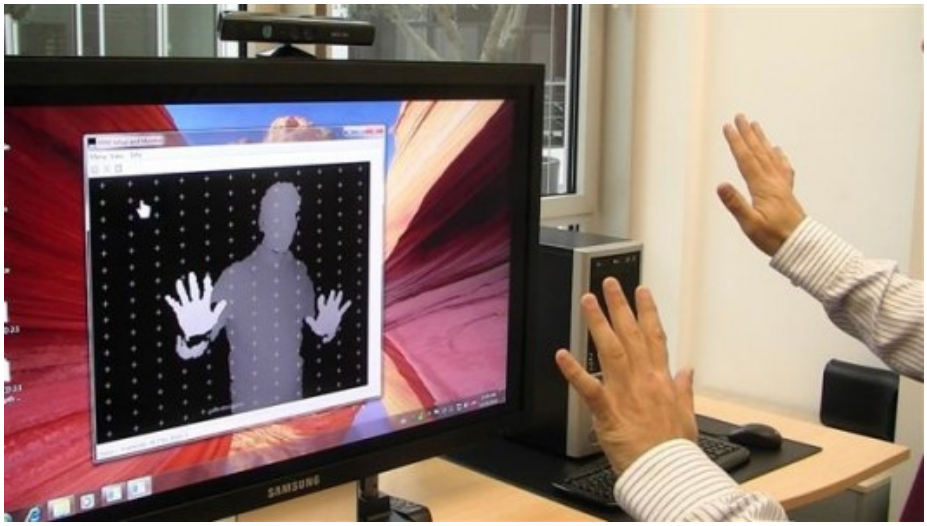


Figura 2.7: Reconocimiento de gestos por cámara (Kinect).

Reconocimiento de voz

Permite a los usuarios interactuar con el sistema a través de su voz. Estas interfaces captan la voz del usuario y la analizan y, si reconocen algún patrón de voz, ejecutan el comando asociado. El sistema debe identificar las palabras habladas, ya sean palabras individuales o frases. El problema que se plantea es el de hacer cooperar un conjunto de informaciones que provienen de diversas fuentes de conocimiento (acústica, fonética, fonológica, léxica, sintáctica, semántica y pragmática), en presencia de ambigüedades, incertidumbres y errores inevitables para llegar a obtener una interpretación aceptable del mensaje acústico.

Sin embargo, el reconocimiento no se realiza sólo analizando el espectro de voz y comparando; también pueden usar tecnologías diferentes como *speech-to-text*[1] que convierta las palabras pronunciadas en palabras escritas en el sistema y compare simplemente cadenas de texto.

Estos sistemas requieren además de entrenamiento previo, ya que cada usuario posee un patrón vocal distinto. Una vez entrenado con el tono y timbre de voz del usuario, deben grabarse y almacenarse los comandos. No obstante, la precisión alcanzada depende en gran medida de la longitud de la cadena a reconocer. Además, el término reconocimiento de voz puede referirse a identificar 'quién' habla más que a lo que dice. Reconocer al hablante puede simplificar la tarea de traducción o incrementar la seguridad de los sistemas.

Diseño del sistema

El diseño de los sistemas de reconocimiento de voz consta de tres fases o etapas:

- Aprendizaje: El primer paso es el más importante, el aprendizaje, ya que determinará el éxito del sistema. Un aspecto crucial en esta fase es la elección del tipo de aprendizaje a utilizar, que puede ser *Aprendizaje Inductivo* o *Aprendizaje Deductivo*.

  Las técnicas de Aprendizaje Inductivo se basan en la automatización del sistema a la hora de conseguir los conocimientos necesarios a partir de muestras reales. Los ejemplos o muestras constituyen aquellas partes de los sistemas basados en *modelos ocultos de Márkov* o *redes neuronales artificiales* que son configuradas automáticamente a partir de conjuntos de entrenamiento.

  Por otro lado, las técnicas de Aprendizaje Deductivo están basadas en la transferencia de conocimientos de un sistema experto humano al sistema informático. Un ejemplo de esta técnica o metodología son los Sistemas Expertos.

  Cabe destacar que en la práctica no se utiliza únicamente una técnica u otra, sino que se sigue un enfoque deductivo-inductivo en el cual los aspectos generales se obtienen por aprendizaje deductivo y la caracterización o aspectos concretos se obtienen a través del aprendizaje inductivo.

- Decodificación acústico-fonética: Las fuentes de información mencionadas en la sección 2.1.3, con los correspondientes procedimientos interpretativos, dan lugar al módulo decodificador acústico-fonético. También es llamado decodificador léxico.

  La entrada del decodificador es la señal vocal, sometida primeramente a preproceso. En este preproceso la señal se parametriza y se convierte a algún modelo de representación digital.

[1] Voz a texto.

- Modelado del lenguaje: Las fuentes de conocimiento sintáctico, semántico y pragmático conforman un modelo del lenguaje del sistema. En el caso de integración de la sintaxis y la semántica, se obtienen sistemas de reconocimiento de voz de gramática restringida, objeto de aplicaciones para tareas muy especificas. Los modelos de lenguaje más complejos necesitan, además, grandes «corpora de voz» y texto escrito para la fase de aprendizaje y su posterior evaluación. Gracias a estos sistemas tan complejos y completos, es posible abordar gramáticas muy complejas y procesar lenguajes naturales.

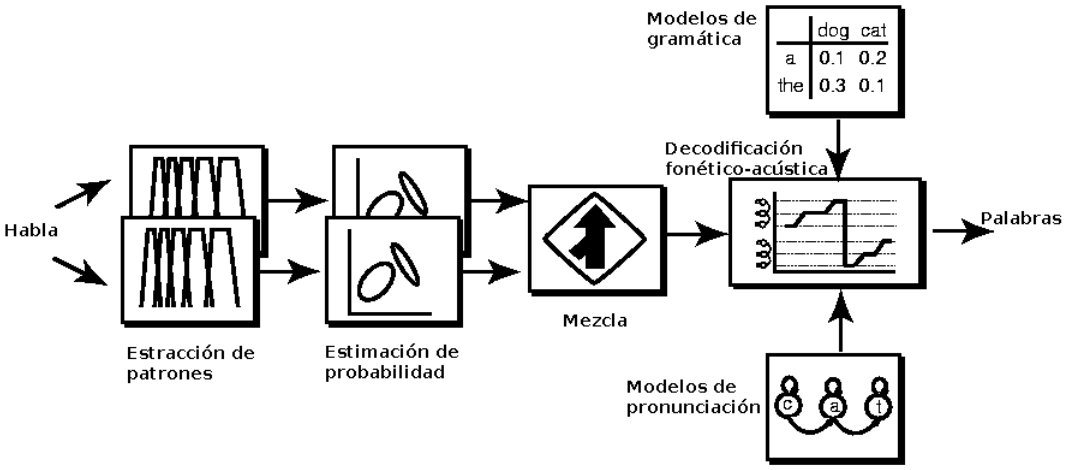


Figura 2.8: Proceso completo de reconocimiento de voz, con todas sus etapas.

Rendimiento y Precisión del reconocimiento de voz

El rendimiento de los sistemas de reconocimiento de voz se evalúa en términos de precisión y velocidad en la traducción o comparación. La precisión se mide generalmente con el *ratio* de error por palabra (*Word Error Rate* - WER), mientras que la velocidad se mide en tiempo real.

Sin embargo, la vocalización varía en términos de acento, pronunciación, tono, volúmen y velocidad, entre otros, al igual que puede ser distorsionada por ruido de fondo o eco, lo que conlleva a un aumento en el error. Generalmente, la precisión varía en función de:

- Tamaño del vocabulario
- Discurso aislado o continuo
- Reglas del lenguaje o restricciones de la tarea en cuestión
- Discurso leído o preparado frente al discurso espontáneo
- Condiciones adversa del entorno, como eco o ruido de fondo

Interfaces cerebrales

Este tipo de interfaces naturales, denominadas también Interfaces Cerebro-Computador por el término anglosajón Brain-Computer Interfaces (BCI), utilizan la actividad cerebral del usuario, captada a través de un dispositivo especial, para ser procesada e interpretada por el computador. Esta actividad se puede transformar en acciones reales en el entorno, estableciendo un camino para interaccionar con el exterior mediante nuestro pensamiento. Generalmente se encaminan a asistir, aumentar o reparar funciones cognitivas y/o motrices humanas.

Modelo funcional

En cualquier tipo de interfaz cerebral se sigue un proceso general como el mostrado en la figura 2.9, si bien algunas interfaces podrían no integrar algún componente o incluir algunos adicionales.

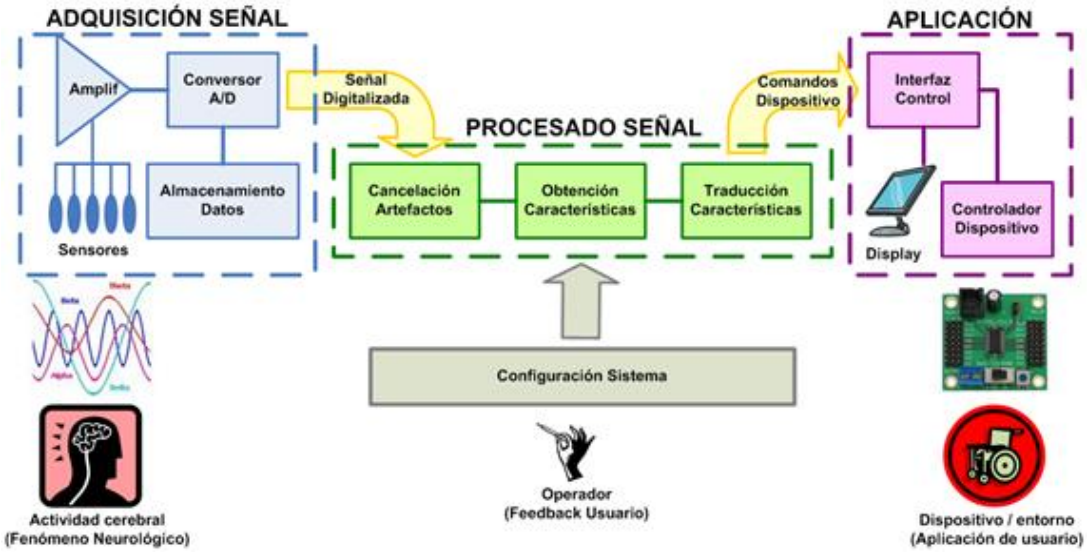


Figura 2.9: Modelo funcional de las Interfaces Cerebrales.

Se distinguen 4 bloques funcionales:

1. Adquisición de señal: Su objetivo es el registro de la actividad cerebral y su adecuación (digitalización) para la siguiente etapa de procesado de la señal. Se trata, pues, de la captura de las ondas cerebrales gracias a los sensores (en este caso, electrodos en el cuero cabelludo o microelectrodos implantados en la superficie del córtex). Aunque no es necesario que los sistemas BCI almacenen la señal registrada, casi todos implementan tal función con el objetivo de permitir posteriormente realizar diferentes análisis o procesados diferentes diferentes de señal (variando el algoritmo de procesado).
2. Procesado de señal: Este bloque funcional recibe la señal digitalizada y la transforma en comandos entendibles por el dispositivo del usuario. Este proceso lo lleva a cabo en

tres etapas secuenciales:

a) Cancelación de ruido: En esta fase se elimina el posible ruido recibido con la señal, fuente de otro tipo de actividad bioeléctrica como pueda ser el movimiento ocular. Esta fase no suelen implementarla muchos de los sistemas BCI; algunos la incluyen en la fase segunda.

b) Obtención de características: En esta fase se traduce la señal cerebral en un vector de características en correlación con el fenómeno neurológico asociado a la señal. Pueden ser vectores numéricos, donde cada valor se asocie a un nivel de onda.

c) Traducción de características: En esta fase el vector de características se transforma en una señal de control adecuada al dispositivo. Cuando la señal de control generada es un valor discreto, se habla de clasificación de la característica.

3. Aplicación: Este bloque funcional recibe los comandos de control y realiza las acciones correspondientes en el dispositivo a través del controlador. La señal procesada es expandida o transformada a través del interfaz de control y también puede incorporarse una pantalla que proporcione retroalimentación al usuario.

4. Configuración: Este último bloque funcional permite definir los parámetros del sistema. El operador que lo lleve a cabo no tiene por qué ser un operador técnico, puede ser el propio usuario del sistema; en el mejor de los casos esta labor la realizarán autómatas informáticos que ajusten el comportamiento del sistema en función de los resultados y la retroalimentación del usuario.

Tipos de interfaces BCI

Las interfaces cerebrales se clasifican en torno a tres grandes grupos, los cuales están compuestos de varios tipos de interfaces cada uno. Por un lado, tenemos las *Interfaces Cerebrales Invasivas*, en las cuales los electrodos han de colocarse directamente en el cerebro, en la materia gris, mediante neurocirugía. Estas interfaces ofrecen la mejor calidad de señal. Por otro lado, están las *Interfaces Parcialmente Invasivas*, cuyos sensores se implantan aún dentro de la cabeza, pero fuera del cerebro en lugar de la materia gris. Por último, tenemos las *Interfaces Cerebrales No Invasivas*, que ubican los sensores fuera de la cabeza, sobre la superficie del cuero cabelludo. Acorde a esta clasificación general, se analizan los diferentes tipos de interfaces dentro de cada una de ellas.

Interfaces Cerebrales Invasivas

Dentro de estas interfaces encontramos los siguientes tipos:

- Visión: Estas interfaces se desarrollaron con el objetivo de reparar la vista dañada y la ceguera, siempre y cuando no fuese congénita. El primer prototipo de esta BCI contiene 68 electrodos que se implantan en el córtex visual, estimulando el cerebro y

produciendo fosfenos, la sensación de ver luz (más concretamente, manchas luminosas). La interfaz incluye además cámaras colocadas en las gafas del usuario para enviar señales al implante. Con la ayuda de un computador, el implante permite ver sombras de gris en un campo de visión limitado.

La segunda generación cuenta con implantes más sofisticados que permiten una mejor asociación de los fosfenos en una visión coherente. Se consigue un campo de visión más próximo al natural.

- Movimiento: Se centran en neuroprótesis con el objetivo de devolver la movilidad a individuos con parálisis o para desarrollar dispositivos que les ayuden, tales como interfaces robóticas.

  En 2005 se construyó una mano artificial que fue capaz de utilizar Matt Nagle por medio de una interfaz cerebral. Para conseguirlo, implantaron 96 electrodos en la circunvolución precentral derecha (área del córtex motriz encargada del movimiento del brazo). Cuando el paciente pensaba en mover su brazo, se captaba la actividad cerebral y se interpretaba en comandos del brazo robótico.

  Recientemente han conseguido control directo sobre piernas robóticas con varios grados de libertad utilizando conexiones a las neuronas del córtex motriz en pacientes tetrapléjicos.

Interfaces Cerebrales Parcialmente Invasivas

Dentro de estas interfaces encontramos los siguientes tipos:

- Electrocorticografía: Mide la actividad eléctrica del cerebro de una forma similar a la Electroencefalografía, pero los electrodos se encuentran en un relleno protector situados sobre el córtex. Se utilizó en humanos por vez primera en 2004, obteniendo muy buenos resultados: el control es rápido, requiere muy poco entrenamiento y ofrece una buena relación entre calidad de la señal y nivel de invasión en el usuario. La señal puede ser subdural o epidural.

  La Electrocorticografía es prometedora debido a su gran resolución espacial, mejor relación señal-ruido que otras interfaces, amplio rango de frecuencia y la menor necesidad de entrenamiento por parte del usuario; al mismo tiempo, la dificultad técnica es menor, hay menos riesgo clínico que las totalmente invasivas y proporcionan una mayor estabilidad.

- Imagen reactiva a la luz: Estas interfaces aún son «teóricas», es decir, no se han desarrollado. Pretenden implantar un láser dentro del cráneo que pueda ser entrenado sobre una neurona y capture la reflexión del rayo con otro sensor. Cuando la neurona se active, el patrón de luz láser y longitudes de onda variarán, permitiendo a los investigadores supervisar neuronas individuales con menos contacto con el tejido y menor

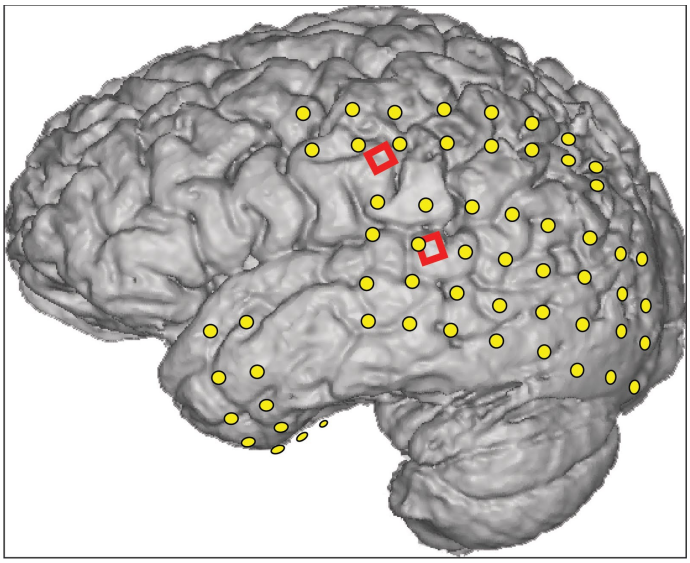


Figura 2.10: Posicionamiento de los electrodos sobre el córtex en la Electrocorticografía.

riesgo de producir cicatrices en el tejido.

Interfaces Cerebrales No Invasivas

Dentro de estas interfaces encontramos los siguientes tipos:

- Electroencefalografía: Es la interfaz no invasiva más estudiada y empleada, debido principalmente a su excelente resolución temporal, facilidad de uso, portabilidad y bajo coste. Sin embargo, no todo son ventajas; puede ser susceptible al ruido y requiere de amplio entrenamiento para obtener los mejores resultados.

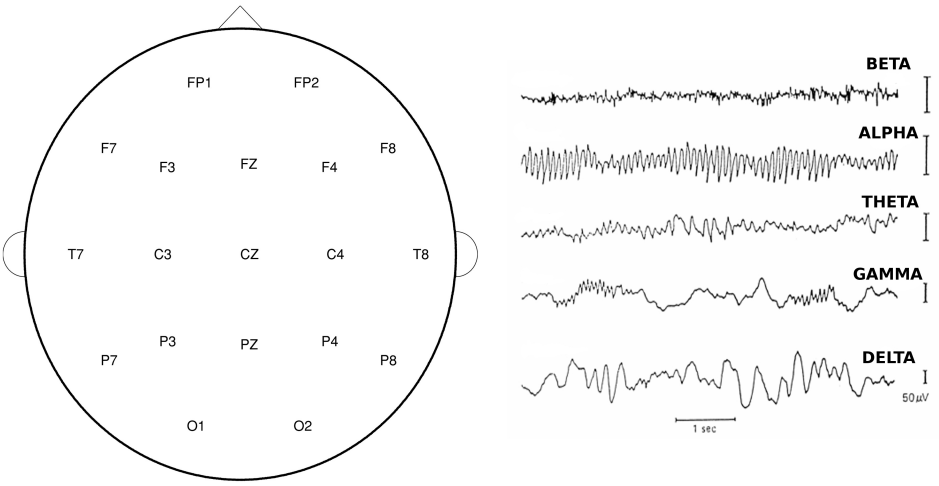


Figura 2.11: Electroencefalografía. Las letras representan la ubicación de los sensores; la parte derecha la salida de las ondas medidas.

Se basa en la captura y grabación de la actividad cerebral a lo largo del cuero cabelludo, medida como fluctuaciones o diferencias de voltaje, resultado de los flujos de corriente

iónicos de las neuronas del cerebro. La captura de la actividad cerebral se lleva a cabo mediante un pequeño dispositivo, colocado sobre el cuero cabelludo del usuario, y que cuenta con una serie de sensores que miden las ondas que el cerebro produce, diferenciando varios tipos de onda. La información obtenida por el dispositivo se envía al computador y se procesa.

- Magnetoencefalografía: Es una técnica de neuroimagen que analiza la actividad cerebral captando los campos magnéticos producidos por las corrientes eléctricas que tienen lugar de forma natural en el cerebro a causa de las neuronas. Los sensores utilizados en la captación son magnetómetros muy sensibles. El magnetómetro más común el SQUID (*Superconducting Quantum Interference Device*), y se estudia el uso de SERF (*Spin Exchange Relaxation-Free*).

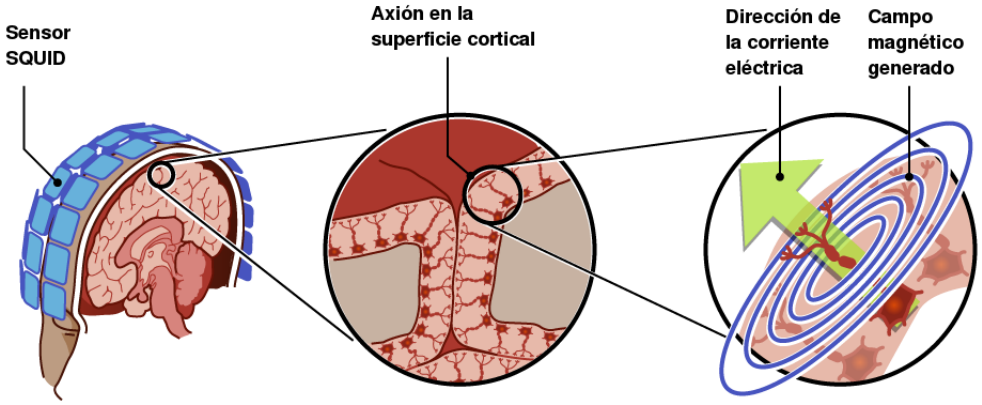


Figura 2.12: Magnetrografía utilizando el magnetómetro SQUID.

El principal problema reside en la debilidad de estos campos magnéticos. A 10 fT[2] de actividad cortical y a 10^3 fT del ritmo Alpha humano (onda Alpha), el campo magnético generado se encuentra por debajo del *ruido magnético ambiental*, en el orden de 10^8 fT. Para generar una señal detectable se necesitan, aproximadamente, 50.000 neuronas activas. La zona del cerebro capaz de generar campos detectables es la capa de celulas piramidales, situada perpendicularmente a la superficie cortical[3]. Los investigadores han experimentado con varios métodos de análisis de señal, pero aún no se ha encontrado un método clínico útil.

- Imagen por resonancia magnética: Es una técnica que mide la actividad cerebral mediante la detección de los cambios en la circulación de la sangre. Esta técnica se basa en el hecho de que el riego sanguíneo del cerebro y la activación neuronal están relacionadas, de modo que cuando un área del cerebro está siendo usada, el riego en esa sección se incrementa.

[2]Femtotesla
[3]Ley de Maxwell

En su inicio se utilizaba el contraste de nivel de oxígeno en sangre. Posteriormente se introdujeron otros métodos como el etiquetado de circulación arterial y la imagen por resonancia magnética de difusión.

La Imagen por resonancia magnética es utilizada tanto en investigación como en el mundo clínico. Puede ser combinada también con otras técnicas como la EEG. Incluso algunas compañías han desarrollado dispositivos comerciales.

Dispositivos BCI Comerciales

En los últimos años múltiples compañías han desarrollado dispositivos de de interfaz cerebral a bajo coste, generalmente interfaces de Electroencefalografía. Estos dispositivos se han desarrollado mayormente para el entretenimiento como medio de acercar esta tecnología a cualquier usuario; NeuroSky y Mattel han tenido especial éxito comercial.

Los dispositivos que han ido apareciendo en los últimos años son:

- En 2006, Sony patentó una interfaz neuronal que permitía producir señales en el córtex neuronal a partir de ondas de radio.
- En 2007, NeuroSky comercializó la primera interfaz EEG asequible por el consumidor medio junto a su juego NeuroBoy. Fue también el primero en utilizar la tecnología de *sensor seco*.
- En 2008, OCZ Technology desarrolló un dispositivo basado en Electromiografía para el control de videojuegos. Por otra parte, Square Enix trabajó junto a NeuroSky para crear *Judecca*, un juego controlado mentalmente (parcialmente, no control total) mediante EEG.
- En 2009, Uncle Milton Industries lanzó al mercado, junto a NeuroSky, *Star Wars Force Trainer*, un juego que creaba la ilusión de hacer levitar una pelota cuando el jugador se concentraba mentalmente. También Emotiv Systems desarrolló *EPOC*, un dispositivo de EEG de 14 canales que puede leer 4 estados mentales, 13 estados conscientes, expresiones faciales y movimientos de cabeza.
- En 2012, g.tec comercializó *intendiX-SPELLER*, el primer sistema BCI doméstico que permitía el control de aplicaciones y videojuegos. Puede detectar señales cerebrales con un 99 % de precisión.
- En 2013, Hasaca National University anunció el primer programa de interfaz cerebral en realidad virtual.

Realidad Aumentada

Las interfaces de realidad aumentada presentan una variación en el entorno, en el cual hacen percibir al usuario el mundo real con objetos virtuales integrados, dando la sensación de que ambos están compuestos y alineados. En otras palabras, componen en tiempo real vistas (ya sean directas o indirectas) del mundo físico, el cual ha sido «aumentado» añadiendo información virtual generada por computador.

Introducción

Los sistemas de realidad aumentada combinan el mundo real con el virtual de forma perfecta, mostrando una imagen sintetizada por computador. Una característica esencial es que son interactivos en tiempo real. Realizan el cálculo de la síntesis de imagen en un tiempo suficientemente pequeño como para evitar que se aprecie que no ocurren en el mismo instante de tiempo y parezca en tiempo real. Otra característica esencial es que la alineación de los objetos debe hacerse en el espacio 3D, no sobre planos bidimensionales.

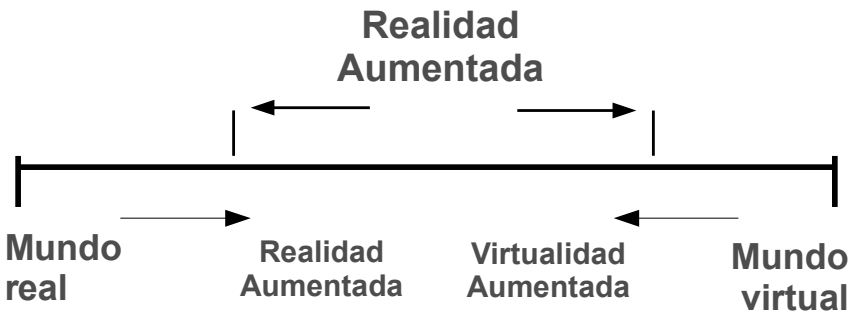


Figura 2.13: Descripción gráfica de la Realidad Aumentada.

La Realidad Aumentada (RA) simplifica la vida del usuario proporcionándole información adicional en sus alrededores y/o en cualquier vista indirecta de su entorno, como a través de una cámara. Aumenta también la percepción o la interacción con el entorno. Sin embargo, ha de diferenciarse bien de la Realidad Virtual, pues mientras en la Realidad Virtual (RV) la inmersión del usuario en el mundo virtual o sintético es total, en la RA es sólo parcial.

Aunque hasta ahora se ha hablado sólo del punto de vista gráfico del «aumento» de la realidad, del mundo real, la potencia de la Realidad Aumentada no se restringe a lo visual. Es posible aplicarla al resto de sentidos, como el olfato, el tacto o el oído. Un claro ejemplo son los cines que, en determinados momentos de la película, dispersan en el ambiente los olores que el usuario olería si estuviese en la piel del actor, en el lugar que él está.

Los elementos que componen la interfaz son:

- **Dispositivo de vídeo:** Capturan el mundo real. Pueden ser de diversos tipos, desde webcams personales hasta cámaras de alta resolución.

- **Unidad de proceso:** La unidad de proceso depende en gran medida de los requisitos de aplicación. Aplicaciones de bajos requisitos podrán utilizar portátiles o tablets como unidad de proceso. Cuando los requisitos de la aplicación son más exigentes, puede incluso dividirse la carga de trabajo entre la CPU del computador y la GPU de la tarjeta gráfica.
- **Dispositivo de visualización:** La imágen resultado de la composición de los «dos mundos» debe mostrarse por medio de algún dispositivo de visualización. Puede ser:
  - Tecnología óptica.
  - Tecnología de vídeo.

Visión por computador

La visión por computador se encarga de renderizar objetos 3D virtuales desde el mismo punto de vista desde el que se capta el mundo real a través de la/s cámara/s. El registro de la imágen se lleva a cabo de diferentes métodos, compuestos por dos fases: *tracking* y reconstrucción/reconocimiento. En la primera fase, la cámara detecta una serie de marcas o puntos de interés sobre los que situar los objetos virtuales. Esta detección puede basarse en detección de bordes o cualquier algoritmo de procesamiento de imágenes. En la fase segunda, se utilizan los datos obtenidos de la primera fase para reconstruir el mundo real coordinado con el virtual.

Asumiendo una cámara calibrada y un modelo de proyección, dado un punto p de coordenadas $(x, y, z)_T$ en la captura de la cámara, su proyección en el plano de imagen es $(x/z, y/z, 1)_T$.

En esta situación, nos encontramos con dos sistemas de coordenadas: el sistema de coordenadas del mundo W y el sistema de coordenadas de la imagen 2D. En la figura 2.14 en la página siguiente se da una visión conjunta de los sistemas de coordenadas.

Los métodos de tracking son muy dependientes del entorno, ya sea interior, exterior o un entorno mixto. Además puede ser fijo o móvil. Los diferentes entornos contribuyen a una representación más o menos precisa, haciendo menor o mayor el error de representación. También es más difícil el tracking en sistemas con movilidad.

Dispositivos de Realidad Aumentada

En la introducción se comentó brevemente los componentes de los sistemas AR. En este apartado los comentaremos más en detalle, siendo los principales los dispositivos de visualización, los dispositivos de entrada, los dispositivos de tracking y los computadores.

- **Dispositivos de visualización**: Nos permiten visualizar el contenido virtual sobre el mundo real. Hay tres tipos principales de dispositivos de visualización: HMD (*Head*

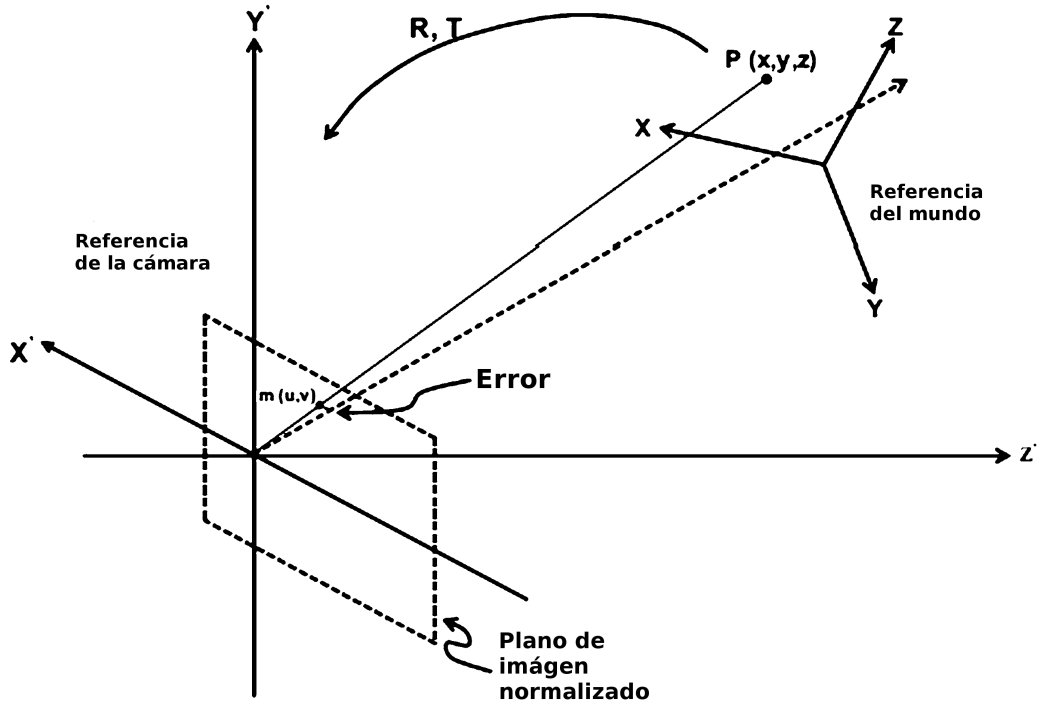


Figura 2.14: Visión por computador en Realidad Aumentada.

Mounted Displays o dispositivos «montados» en la cabeza), dispositivos de mano y dispositivos de visualización espacial.

Los *HMD* son dispositivos similares a cascos que colocan ambas imágenes del mundo real y el mundo virtual sobre el campo de visión del usuario. Pueden ser además monoculares o binoculares. En cuanto a la forma de visualizarlo se diferencian también dos tipos: visión de vídeo o visión óptica. En los primeros (más demandados) se requieren dos cámaras en el casco y se requiere que se procese la imagen de ambas cámaras para proporcionar la imagen de realidad aumentada; en los segundos, se utiliz un espejo que permite al usuario visualizar el mundo físico y a la vez visualizar la información superpuesta en el espejo, la cual refleja a los ojos del usuario. Mientras que el segundo enfoque consigue un mayor grado de realismo, más naturalidad, el primer enfoque proporciona más control del resultado obtenido.

Los *dispositivos de mano* emplean pequeños dispositivos de procesamiento, como tablets, que el usuario puede llevar en la mano para ir visualizando contenidos en Realidad Aumentada. En este enfoque, el usuario utiliza el dispositivo para orientar la cámara del mismo al lugar deseado y, sobre la pantalla, se realiza la superposición de imágenes. También se apoyan en otros sensores, no sólo la cámara. Utilizan por ejemplo sistemas de GPS y brújula para saber la posición y orientación y qué representar cuando no se da el caso que puedan identificar marcas.

Los dispositivos de mano que más prometen extenderse en la utilización para Realidad

Aumentada son los *smartphones*, ya que cada vez cuentan con CPUs más potentes, cámaras de mayor resolución, acelerómetros, sistema de GPS y brújula, haciendo de ellos una plataforma móvil muy completa, potente y manejable.

Los *dispositivos de visualización espacial*, por su parte, emplean videoproyectores, hologramas, etiquetas de radio frecuencia, y otros dispositivos que muestran la información digital directamente en el medio físico sin requerir ningún otro dispositivo por parte del usuario. Este enfoque busca separar los dispositivos de visualización del usuario e integrarla más en el mundo. La principal ventaja de ello es la escalabilidad del sistema frente a grupos de usuarios y la colaboración entre ellos.

- **Dispositivos de entrada**: Se utilizan varios dispositivos de entrada, muy diferentes unos de otros. Un tipo de dispositivo de entrada son los *guantes de realidad aumentada*. Aparentemente son guantes comunes, pero éstos cuentan con sensores que detectan el movimiento de los dedos, su posición y rotación, de forma que pueden traspasar esa información al mundo 3D y utilizarla para manipular los objetos como si pudiesen cogerse de verdad con la mano y moverlos. Ésto se lleva a cabo recreando la mano real en el espacio 3D y detectando las colisiones e interacciones con los demás objetos virtuales.

  Otros dispositivos de entrada son las *pulseras inalámbricas*. Estos dispositivos incluyen un lector RFID[4], acelerómetros en los 3 ejes (x-y-z) y capacidad de comunicación RF (Radio Frecuencia). La pulsera se utiliza como parte de una red de información en el cuerpo, compuesta además por un teléfono móvil y un auricular inalámbrico. La pulsera permite interacción directa con objetos que utilizan etiquetas de RFID. Una vez que el objeto es detectado, el usuario manipula la información mediante gestos de la muñeca o movimientos continuos de la mano.

  En los sistemas de realidad aumentada móviles, el móvil o tablet empleado es el propio dispositivo de entrada también. Sea cual sea el dispositivo de entrada empleado, es muy dependiente del tipo de aplicación a desarrollar.

- **Dispositivos de tracking**: Estos dispositivos consisten en una cámara digital y/o otros sensores ópticos, GPS, acelerómetros, etc. Cada tecnología tiene diferentes niveles de precisión y son muy dependientes del sistema final.

- **Computadores**: Los sistemas de Realidad Aumentada requieren gran capacidad de procesamiento por parte de la CPU y una cantidad de RAM considerable para sintetizar las imágenes de la cámara. Los sistemas móviles utilizan potentes computadores portátiles en su configuración, aunque con la aparición de *smartphones* y *tablets* cada vez más potentes, se espera que puedan sustituir al computador portatil. Por otro lado, los sistemas estacionarios siguen utilizando potentes estaciones de trabajo con gráficas

[4]Siglas de *Radio Frequency IDentification*, en español, Identificación por Radiofrecuencia

de alta gama.

Interfaces de Realidad Aumentada

Uno de los aspectos más importantes de la realidad aumentada es crear técnicas de interacción intuitivas y apropiadas a la situación y contexto en que hayan de aplicarse. Principalmente, hay cuatro formas de conseguirlo, ya sea mediante *interfaces tangibles*, *interfaces colaborativas*, *interfaces híbridas* o *interfaces multimodales*.

- **Interfaces de Realidad Aumentada Tangibles**: Estas interfaces proporcionan interacción directa con el mundo real, empleando objetos físicos reales y ciertas herramientas. A modo de ejemplo, se presenta la aplicación VOMAR, desarrollada por [K+00]. En ella, el usuario selecciona y recoloca el mobiliario de un salón en realidad aumentada mediante una paleta física.

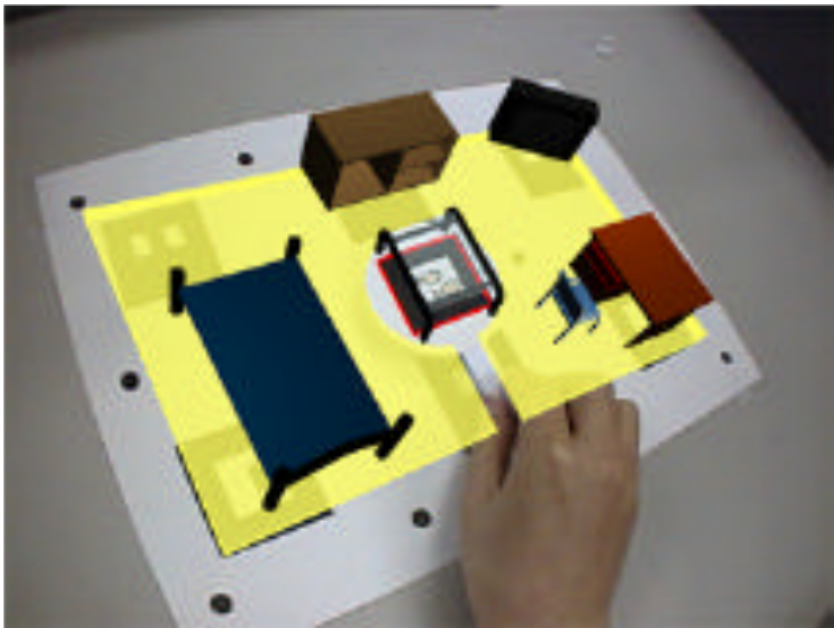


Figura 2.15: Interfaz de Realidad Aumentada Tangible. En concreto, VOMAR de [K+00].

  Los movimientos de la paleta se «mapean» o asocian a comandos basados en intuitivos movimientos, como golpear un objeto para hacerlo desaparecer, o recoger un objeto para posicionarlo en un lugar diferente.

- **Interfaces de Realidad Aumentada Colaborativas**: Las interfaces colaborativas incluyen múltiples pantallas para dar soporte a actividades remotas conjuntas. Utilizan interfaces 3D para mejorar significativamente la colaboración del grupo de trabajo. Es posible integrar todas las localizaciones en una representación 3D que simule que todos se encuentran el el mismo lugar, como ocurre en las teleconferencias, o que trabajan sobre los mismos objetos de trabajo en el mismo área.

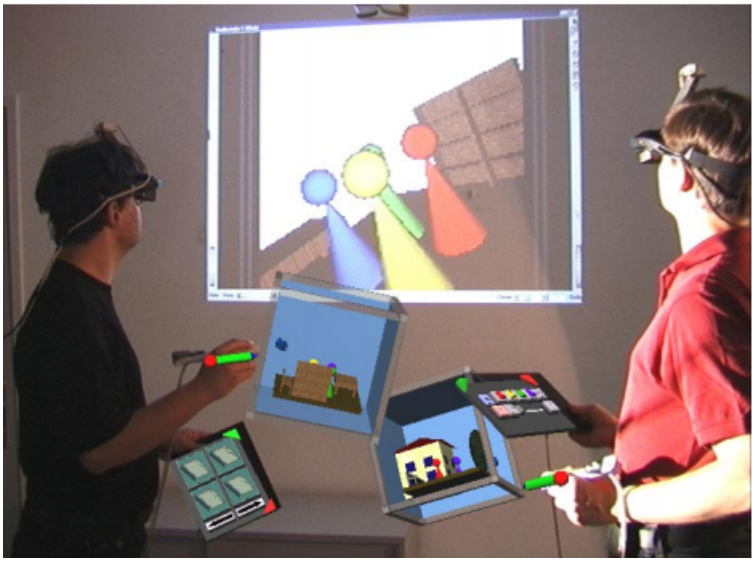


Figura 2.16: Interfaz de Realidad Aumentada Colaborativa. 'Storyboard' con dos usuarios y dos contextos. Imagen obtenida de [SFH00].

La figura 2.16 muestra un ejemplo del segundo tipo. La aplicación *Studierstube* ofrece colaboración en realidad aumentada con múltiples usuarios, múltiples contextos, diferentes formas de visualización e incluso diferentes sistemas operativos.

- **Interfaces de Realidad Aumentada Híbridas**: Las interfaces híbridas combinan, junto a las interfaces expuestas anteriormente, otras interfaces y dispositivos diferentes que proporcionan otras formas de interacción. Su principal ventaja es ofrecer una plataforma flexible, no planificada para un solo uso específico, sino para un amplio rango de aplicaciones que a priori no se conoce.

Figura 2.17: Interfaz de Realidad Aumentada Híbrida. Utiliza otra interfaz, en este caso un mando de videoconsola, y la combina con las interfaces AR. Imagen obtenida de [SOF05].

Estos sistemas RA se diseñan además para dar soporte al usuario final a asignar la interacción física con las operaciones sobre los objetos virtuales, así como la reconfiguración de dispositivos, objetos y operaciones con las que el usuario pueda interactuar con el sistema.

- **Interfaces de Realidad Aumentada Multimodales**: Las interfaces multimodales combinan objetos reales en la entrada con formas naturales de comunicación y comportamiento, como pueden ser el habla, el tacto, los gestos, etc.

  Estos tipos tipos de interfaces de Realidad Aumentada son aún recientes.

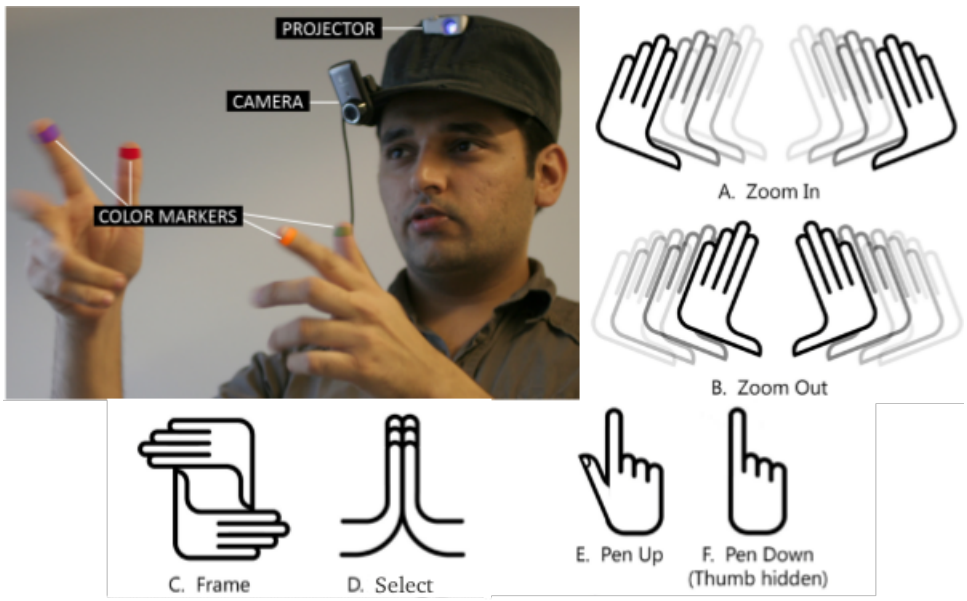


Figura 2.18: Interfaz de Realidad Aumentada Multimodal. Integra la interacción gestual; los gestos son: A - Acercar, B - Alejar, C - Enmarcar, D - Seleccionar, E - Dejar de escribir, F - Escribir

Por ejemplo, la interfaz multimodal 'WUW', proporciona al usuario información proyectada en superficies, paredes y objetos físicos, y la manipula a través de gestos naturales de la mano, movimiento de brazos y/o con el objeto mismo (si éste es real).

2.2 Electroencefalografía (EEG)

2.2.1 Definición

La Electroencefalografía es la captura y grabación de la actividad cerebral a lo largo del cuero cabelludo. Esta actividad cerebral es medida como fluctuaciones o diferencias de voltaje, resultado de los flujos de corriente iónicos de las neuronas del cerebro.

En un contexto clínico, la EEG se refiere a la grabación de la actividad eléctrica del cerebro en un corto período de tiempo –normalmente 20-40 minutos-. A pesar de usarse también, en este contexto, otras técnicas como los *Potenciales Relacionados con Eventos* y la *Magnetoencefalografía*, la Electroencefalografía es la única que se usa como estándar en pruebas y diagnósticos actualmente.

Generación de señal

El cerebro posee una carga eléctrica mantenida por miles de millones de neuronas. Las neuronas se encuentran cargadas eléctricamente, polarizadas, por una membrana que transporta proteínas. Entre las neuronas hay continuamente un intercambio de iones; iones de la misma carga se repelen y, cuando en ocasiones muchos iones son repelidos en el mismo instante de tiempo, provocan una repulsión en cadena que denominamos *onda*. El proceso en sí se llama *conducción de volumen*. Cuando las ondas alcanzan el cuero cabelludo, pueden ser capturadas como voltajes.

Si tuviésemos conectados un par de electrodos al cuero cabelludo, y a su vez conectados éstos a un amplificador, seríamos capaces de capturar esos voltajes y mostrarlos en el amplificador como una variación de voltaje a lo largo del tiempo. Este patrón de variación en el voltaje es conocido como Electroencefalograma (EEG).

La amplitud de un Electroencefalograma común suele situarse entre -100 y +100 microvoltios, con una frecuencia de 40 Hz o más.

Para cuantificar las medidas del contenido del electroencefalograma, se emplean medios magnéticos u ópticos en los que se graba la señal digitalizada. El electroencefalograma cuantitativo (qEEG) proporciona información que no puede ser extraída directamente (visualmente) del electroencefalograma.

Una vez que la señal digitalizada ha sido almacenada, puede ser transformada mediante el algoritmo de la Transformada de Fourier, pasando de un dominio «*amplitud-frecuencia*» a un dominio «*potencia-frecuencia*». Por ejemplo, la *potencia absoluta* es la medida de la intensidad de la potencia medida en microvoltios cuadrados y calculada en una serie de bandas de frecuencia, llamadas *espectro de potencia*.

Ventajas e inconvenientes de la Electroencefalografía

Existen otros métodos para el estudio del cerebro, tales como la *Magnetoencefalografía* o la *Espectroscopia Infrarroja*; y a pesar de que la Electroencefalografía (EEG) pueda parecer un método sensitivamente pobre, presenta múltiples ventajas que la anteponen a otros métodos como los expuestos anteriormente.

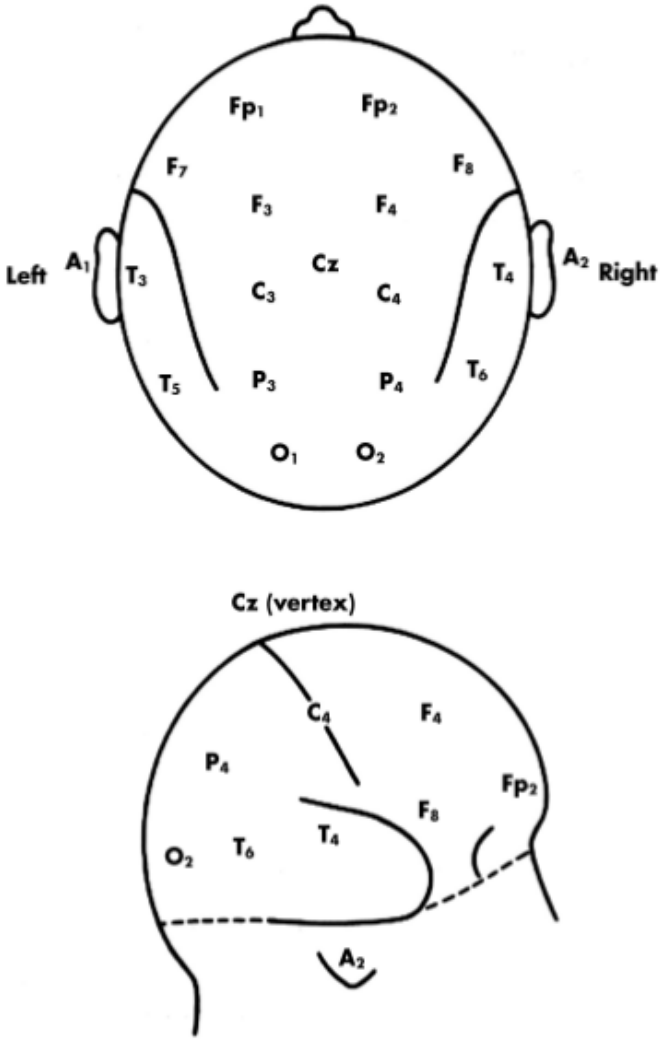


Figura 2.19: Posicionamiento de los electrodos. Abreviaciones: A = auricular; C = central; Cz = vertex; F = frontal; Fp = polo frontal; O = occipital; P = parietal; T = temporal. Imagen obtenida de [Kup04]

- Ventajas:
  - Los costes hardware son significativamente más baratos.
  - Los sensores EEG pueden ser colocados en mayor número de lugares que otras técnicas, además de permitir mobilidad en el sujeto o el equipo de pruebas[5].
  - La frecuencia de muestreo es muy superior a las demás, partiendo de 250 Hz y alcanzando los 2000 Hz. En equipos modernos mucho más sofisticados es posible superar los 20.000 Hz incluso.
  - La Electroencefalografía es silenciosa, lo que posibilita un mejor estudio cuando el sujeto debe experimentar estímulos sensoriales.

[5] Otras técnicas requieren que el sujeto/equipo permanezca inmóvil.

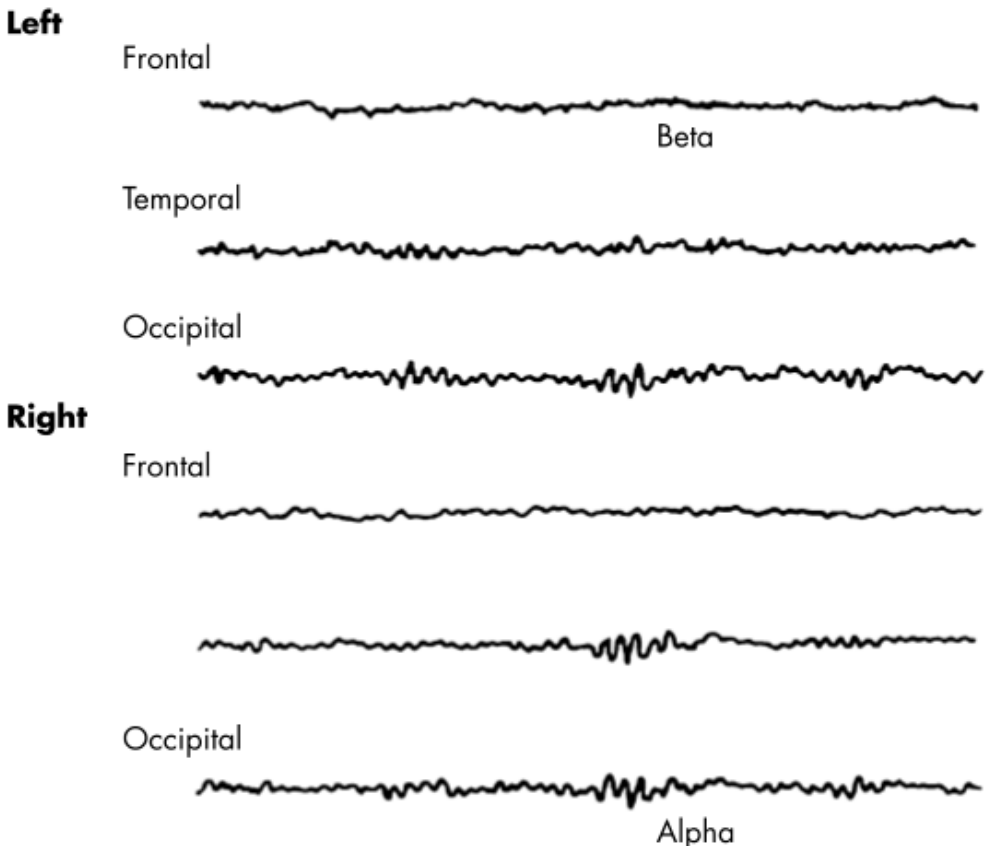


Figura 2.20: Ejemplos de ondas digitalizadas.

- La Electroencefalografía también puede detectar procesamiento de información encubierto (que no requiere una respuesta).
- Puede ser usada en sujetos incapaces de dar respuestas motrices.
- Es extremamente no invasiva, a diferencia de otras técnicas como la *Electrocorticografía*[6].

Sin embargo, también cuenta con algunos inconvenientes:

- Inconvenientes:
  - Baja resolución espacial del cuero cabelludo. Por ejemplo, fMRI[7] muestra directamente áreas activas del cerebro, mientras que EEG requiere de intensa interpretación.
  - La actividad neuronal que tiene lugar bajo el *cortex* se determina con poca precisión.
  - A veces es necesario mayor tiempo de preparación del sujeto, ya que requiere un posicionamiento de los electrodos muy preciso.

[6] La Electrocorticografía requiere la colocación de los electrodos en la misma superficie cerebral.
[7] *functional Magnetic Resonance Imaging*

- La relación señal-ruido es más bien pobre en algunos casos, por lo que los análisis requieren gran cantidad de muestras y un análisis muy sofisticado.

2.2.2 Tipos de onda y Patrones

Generalmente, se estudia la actividad cerebral y se registran los valores en las bandas pertenecientes a las ondas Delta, Theta, Alpha y Beta. Sin embargo, no son la totalidad de bandas en las que se divide el espectro recogido por los electrodos.

A continuación se listan y detallan los tipos de onda existentes, explicando en que frecuencias tienen lugar, que localización física craneal tienen asociada, y los valores normales y patológicos de las mismas. Se presentan también los patrones característicos de cada tipo de onda.

Delta

Se encuentran en una frecuencia inferior a los 4 Hz, y localizadas en la parte frontal en adultos y la parte posterior en niños. Normalmente se asocian con el sueño profundo y la inconsciencia. Patológicamente se asocian con lesiones subcorticales, profundas, y con la *hidrocefalia encefalopatía metabólica*.

Patrón Son el tipo de onda más lento, pero con la mayor amplitud de onda. Las ondas Delta aparecen en el estado 3 de sueño, dominando todo el espectro en el estado 4; el estado 3 de sueño se define por tener menos del 50 % de actividad en Delta, y el estado 4 por tener más del 50 %. En las mujeres se presenta una mayor actividad de delta que en los hombres, aunque esta diferencia no se hace evidente hasta la etapa adulta (entre los 30 y los 40 años de edad). La causa de esta discrepancia puede ser el mayor tamaño craneal de los varones[8].

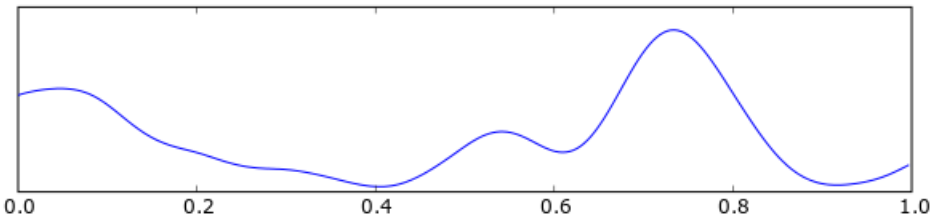


Figura 2.21: Muestra de onda Delta en un período de oscilación.

[8]No ha sido probado al 100 % aún.

Theta

Se encuentran entre los 4 y los 8 Hz, localizadas en posiciones no relacionadas con tareas específicas. Normalmente se asocian con la relajación profunda y la modorra. Patológicamente se asocian con lesiones subcorticales, encefalopatía metabólica.

Patrón Presenta un patrón oscilatorio de menor amplitud que las ondas Delta. A diferencia, las ondas Theta pueden ser de dos tipos:

- Theta Hipocampo, encontrada en mamíferos como roedores, perros o gatos en la la zona del hipocampo.
- Theta Cortical, encontrada en los humanos, de menor frecuencia que Theta Hipocampo.

Theta Cortical (Theta «humana») no presenta relación con la zona del hipocampo (a diferencia de los animales); se presenta en el rango de los 4-7 Hz independientemente del origen[9], manteniendo acotada la oscilación.

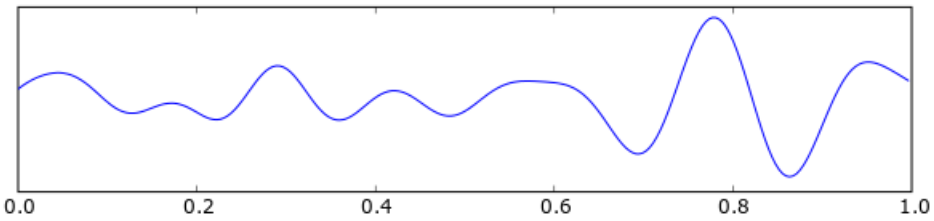


Figura 2.22: Muestra de onda Theta en un período de oscilación.

Alpha

Se encuentran entre los 8 y los 13 Hz, localizadas en las regiones posteriores de la cabeza, ambos lados y centrales c3-c4 (véase Figura 2.19). Normalmente se asocian con la relajación, la meditación, la inhibición del control (aparentemente inhibición temporal de la actividad cerebral) o el acto de cerrar los ojos. Patológicamente se asocian con el coma.

Patrón Corresponde al denominado *ritmo básico posterior*, observado en las regiones posteriores de la cabeza a ambos lados, siendo de mayor amplitud en el lado dominante del individuo (una persona diestra mostrará ondas Alpha de mayor amplitud en el lado derecho posterior). Aumentan su valor al cerrar los ojos y con la relajación. En personas jóvenes y niños, las ondas Alpha muestran una frecuencia menor de los 8 Hz.

[9]En 2.2.2 se comentó que las ondas Theta estaban localizadas en posiciones no relacionadas con tareas específicas.

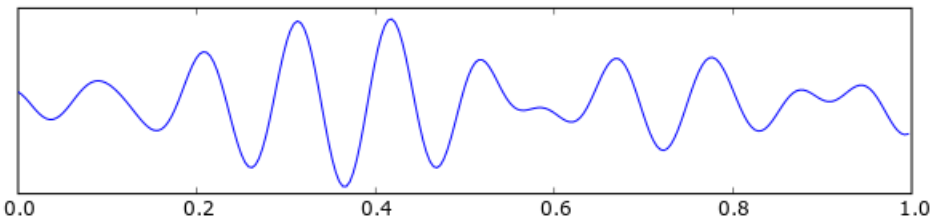


Figura 2.23: Muestra de onda Alpha en un período de oscilación.

Beta

Se encuentran entre los 13 y los 30 Hz, localizadas en ambos lados de forma simétrica, pero más predominantes hacia el frontal. Normalmente se asocian con el estado de alerta, actividad o alta concentración y pensamiento. Patológicamente se asocian con la benzodiazepina –medicamentos psicotrópicos-.

Patrón Presenta un patrón oscilatorio con baja amplitud y múltiples y variables frecuencias, mayores cuanto más activo es el pensamiento y mayor es la concentración. Sobre el cortex motriz se asocian con las contracciones musculares producto de los movimientos isotónicos. Se producen grandes incrementos de Beta en posición estática o supresión voluntaria del movimiento y se reduce en gran medida en los cambios de movimiento.

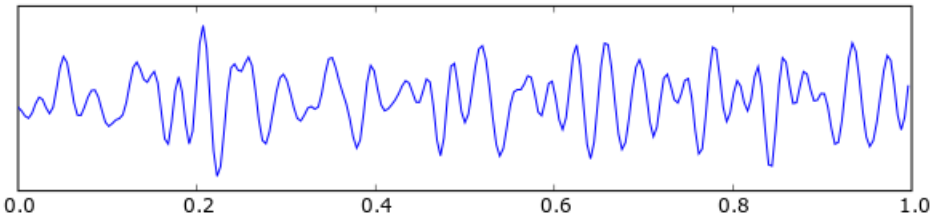


Figura 2.24: Muestra de onda Beta en un período de oscilación.

Gamma

Se encuentran entre 30 Hz y frecuencas iguales o superiores a 100 Hz, localizadas en el cortex somatosensorial. Normalmente se asocian con los sentidos y la memoria. Patológicamente se asocian con la pérdida cognitiva.

Patrón Las ondas Gamma son un patron de oscilación neuronal caracterizado por una frecuencia muy alta, con una amplitud muy pequeña. En torno a los 40+ Hz en humanos. No pueden ser medidas mediante Electroencefalografía analógica, sólo mediante Electroencefalografía digital[10]. Los neurocientíficos creen que las ondas gamma son capaces de conectar

[10]El límite de frecuencia de la Electroencefalografía es de 25 Hz.

información de todas las partes del cerebro, siendo capaces de «recorrerlo» desde el tálamo (parte posterior) hasta el frontal y volver 40 veces por segundo; según los neurocientíficos, tienen valores más altos en personas con mayor coeficiente intelectual y mejor memoria. También en personas con gran autocontrol y compasión.

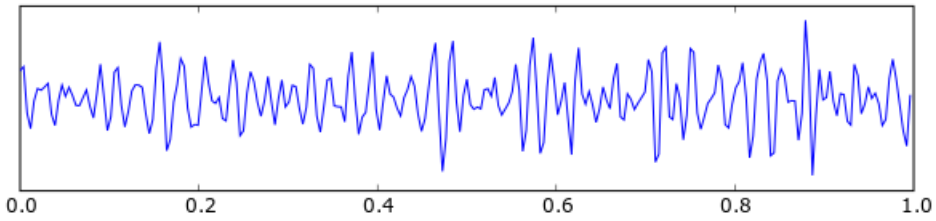


Figura 2.25: Muestra de onda Gamma en un período de oscilación.

Mu

Se encuentran entre entre los 8 y los 13 Hz, al igual que las ondas Alpha, pero en distinta zona; mientras que las ondas Alpha se encuentran en la parte visible del cortex en la parte trasea de la cabeza, las ondas Mu se encuentran en el cortex motriz, aproximadamente de oreja a oreja. Normalmente se asocian con el estado de relajación de las neuronas motrices (estado de relajación física). Patológicamente se asocian con el autismo.

Patrón Llamadas también *Ritmos Sensomotrices*, corresponden a patrones sincronizados de actividad cerebral producida por un gran número de neuronas en la parte del cerebro que controla el movimiento voluntario. Estos patrones tienen una repetición constante de 8-13 Hz, con mayor prominencia en estados de relajación física, no mental.

Las ondas Mu se suprimen en dos situaciones:

- Cuando se ejecuta una acción motriz.
- Cuando se observa una acción motriz ajena.

Esta supresión se denomina *Desincronización*.

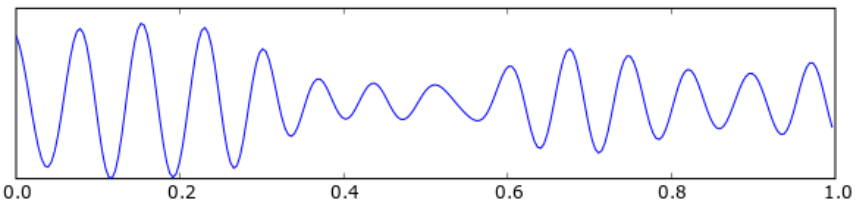


Figura 2.26: Muestra de onda Mu en un período de oscilación.

En la figura 2.27 se muestran todas las ondas juntas, a modo de resumen y a la vez de comparación entre ellas. Se pueden apreciar las diferencias de frecuencia y amplitud comentadas.

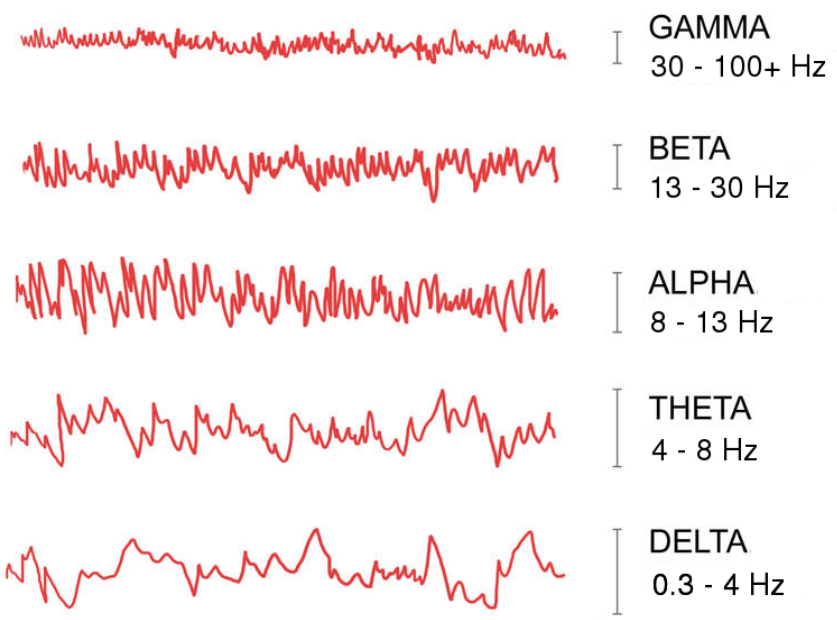


Figura 2.27: Espectro de ondas y su frecuencia.

2.2.3 Procesamiento de señal. Dispositivos de EEG.

En esta sección se discute cómo se procesa la señal EEG en bruto y cómo se extraen los patrones mencionados en la sección anterior. El procesamiento de señal es la labor más compleja en cualquier aplicación de EEG.

Como se comenta en la sección 2.2.1, las señales EEG en bruto son de una magnitud muy pequeña, por lo que puden ser contaminadas por ruido e interferencias con facilidad. Es por ello por lo que el primer paso en el procesamiento de señal es limpiar la señal de ruido e interferencias. Para ello, se aplica en primer lugar un filtro digital que elimina las bandas con frecuencias superiores a 50 Hz, que pueden producir interferencias con la red eléctrica. Eliminarlas no supone gran pérdida de información, ya que la frecuencia más alta en humanos suele situarse en torno a 40 Hz, en la onda Gamma. En un segundo paso, puede submuestrearse la señal de EEG desde su frecuencia de muestreo original de 512 muestras por segundo a otra menor más manejable, aunque este submuestreo no lo realizan todos los dispositivos.

A continuación, se aplica un algoritmo para eliminar de la señal el ruido producido por la

actividad muscular del usuario. Esta actividad muscular se refiere a movimiento de los ojos, parpadeo o movimiento de las extremidades, que alteran los componentes de la actividad cerebral. El método más sencillo para ello es situar sensores adicionales cerca de los ojos, o encima de los párpados, de forma que pueda recogerse información de los movimientos musculares y utilizarse para corregir la señal de entrada eliminando este tipo de ruido.

Antes de pasar a la fase de extracción de características y posteriormente poder clasificarlas en estados mentales del usuario, es necesario crear una representación significativa de la señal original, reduciendo el tamaño de los datos de entrada sin perder información.

Un método sencillo es el denominado método de "banda de potencia espectral" (*band spectral power*), que consiste en aplicar al espectro de onda un filtrado en distintas bandas, centradas en las frecuencias correspondientes a las ondas Delta, Theta, Alfa, Beta y Gamma, desde la frecuencia 0 a la 49 (a partir de 50 Hz se eliminan). Esto determina además el tamaño de las estructuras de datos empleadas para los valores del espectro de onda base y normalizado que se explicarán en la sección 5.4.2.

En este método es común utilizar las transformadas de Fourier, que representan una función periódica $x(t)$ (onda en bruto) como una suma infinita de términos discretos.

2.2.4 Aplicaciones

El uso básico de la Electroencefalografía es el análisis rutinario encefalográfico, consistente en la grabación de la actividad cerebral los últimos 20-30 minutos y su posterior análisis. Generalmente, con este análisis es posible:

- distinguir las crisis *epilépticas* de otros tipos de enfermedades, como *crisis psicogénicas* no epilépticas, *síncope* (desmayo), trastornos del *movimiento sub-cortical* y variantes de *migraña*.
- diferenciar la *encefalopatía «orgánica»* de síndromes psiquiátricos como la *catatonia*.
- servir como test de *muerte cerebral*.
- pronosticar en pacientes en coma.
- determinar cuando administrar medicamentos anti-epilépticos.

Sin embargo, a veces, no es suficiente este análisis rutinario y se precisan de otras medidas y análisis más completos.

Además, la EEG puede servir como medio para monitorizar:

- la profundidad de la anestesia.
- de forma indirecta, la perfusión cerebral en la *endarterectomía carotídea*.
- efecto amobarbital durante la *prueba de Wada*.

Otro uso, relativo a unidades de cuidado intensivo y supervisión cerebral:

- Supervisar los ataques no convulsivos epilépticos.
- Supervisar el efecto de sedantes en paciente en coma inducido.
- Supervisar daños cerebrales colaterales en condiciones de *hemorragia subaracnoidea* (actualmente en investigación).

En el marco de la investigación, es un medio excelente para proporcionar una «ventana» al cerebro y su funcionamiento. Permite la observación de su comportamiento y la actividad que muestra antes ciertos estímulos o situaciones, permitiendo buscar y establecer correlaciones.

Con respecto a la Interacción Persona-Computador, proporciona una oportunidad única de desarrollar interfaces de aplicaciones que utilicen la Electroencefalografía de una forma eficaz, no invasiva y de «bajo coste» comparado con otras técnicas. En este aspecto se ha desarrollado el término *Brain Computer Interface (BCI)* como nuevo paradigma de interacción, sustituyendo las interfaces tradicionales, como ratón y teclado, por interacción cerebral a partir de los valores obtenidos de la EEG. No obstante, los sistemas BCI se apoyan en otros paradigmas:

- Captación de movimiento e imagen
- Potenciales evocados de estado firme
- P300[11]

Proyectos recientes han integrado Electroencefalografía con deportes y juegos. En los deportes, el entrenamiento mental está considerado una parte del éxito físico del deportista; hay fuertes correlaciones entre los cambios de ondas cerebrales y el grado de atención y concentración del deportista en deporte de alto rendimiento. Gracias a la Electroencefalografía, estos cambios y actividad cerebral pueden ser medidos en tiempo real, en las situaciones reales que ocurren, con un mínimo de molestia a causa del artefacto. En un futuro puede que sea posible, gracias a estos análisis, estimular ciertas partes del cerebro para obtener un mayor rendimiento en el deporte.

En cuanto a los juegos, se ha demostrado que en los momentos en que los jugadores producen altos niveles de emisión de ondas Alpha, su concentración y receptividad para el aprendizaje aumenta considerablemente; por tanto, el objetivo es desarrollar juegos que se controlen directamente con la actividad cerebral, de forma que los usuarios puedan entrenarse a sí mismos reproduciendo estos estados mentales para aumentar el aprendizaje. Otro uso, aparte del uso trivial de un videojuego (entretenimiento y diversión), es el tratamiento

[11] La onda P300 es un potencial evocado que puede ser registrado mediante EEG como una deflexión positiva de voltaje con una latencia de unos 300ms.

del ADHD, o *Attention Deficit Hyperactivity Disorder* (Trastorno por déficit de atención con hiperactividad).

2.3 Inteligencia Artificial

Una parte importante de este Proyecto Fin de Carrera es la Inteligencia Artificial. A veces, aplicando la definición de Inteligencia Artificial, se piensa en máquinas inteligentes sin sentimientos, que «obstaculizan» la resolución o la búsqueda de la mejor solución a un problema dado. Muchos pensamos en dispositivos artificiales capaces de concluir miles de premisas a partir de otras premisas dadas, sin que ningún tipo de emoción tenga la opción de obstaculizar dicha labor. Pero, ¿qué es, en definitiva, la Inteligencia Artificial?

2.3.1 Definición

Como primera aproximación, podríamos definir la Inteligencia Artificial (IA) como la capacidad de razonar de un agente no vivo.

Sin embargo, Russell considera que no se puede definir la IA con una definición tan simple [RN04], que lo englobe todo, pues ha de atenderse a diferentes enfoques y procesos que componen la IA. Propone la definición de la Inteligencia Artificial organizada en cuatro categorías, como puede apreciarse en la figura 2.28.

Sistemas que piensan como humanos	**Sistemas que piensan racionalmente**
«El nuevo y excitante esfuerzo de hacer que los computadores piensen... máquinas con mentes, en el más amplio sentido literal» (Haugeland, 1985) «La automatización de actividades que vinculamos con procesos de pensamiento humano, actividades como la toma de decisiones, resolución de problemas, aprendizaje...» (Bellman, 1990)	«El estudio de las facultades mentales mediante el uso de modelos computacionales» (Charniak y McDermott, 1985) «El estudio de los cálculos que hacen posible percibir, razonar y actuar» (Winston, 1992)
Sistemas que actúan como humanos	**Sistemas que actúna racionalmente**
«El arte de desarrollar máquinas con capacidad para realizar funciones que cuando son realizadas por personas requieren de inteligencia» (Kurzweil, 1990) «El estudio de cómo lograr que los computadores realicen tareas que, por el momento, los humanos hacen mejor» (Rich y Knigth, 1991)	«La Inteligencia Computacional es el estudio del diseño de agentes inteligentes» (Poole *et al.* 1998) «IA... está relacionada con conductas inteligentes en artefactos» (Nilsson, 1998)

Figura 2.28: Definición de la Inteligencia Artificial por Russell.

Las definiciones de la parte superior se refieren a *procesos mentales* y *razonamiento*; las definiciones de la parte inferior se refieren a la *conducta*. Por otra parte, las definiciones a la izquierda definen el éxito a la hora de actuar fielmente a como lo harían los humanos, mientras que las definiciones de la derecha toman como concepto ideal la *racionalidad*, pudiendo hablar de sistema racional.

Sistema Racional: Es aquel sistema inteligente que hace lo «correcto» en función del conocimiento que posee. Se puede extraer de lo anterior que lo que *él* considere correcto no tiene por qué ser *lo* correcto, ya que el conocimiento poseído puede ser escaso/erróneo.

Existe un enfriamiento entre los enfoques centrados en el ser humano y los enfoques centrados en la racionalidad. Ésto se debe principalmente a que el enfoque racional implica una combinación de matemáticas e ingeniería, mientras que el enfoque centrado en el ser humano no debe ser una ciencia empírica basada en hipótesis y confirmación.

A continuación se detallan en mayor medida los enfoques explicados.

Comportamiento humano: Enfoque de Turing

La **Prueba de Turing** (Alan Turing, 1950) se diseñó para proporcionar una definición operacional y satisfactoria de inteligencia. Para Turing, un computador supera su prueba y se considera inteligente si un evaluador humano no es capaz de distinguir si las respuestas, a una serie de preguntas dadas, son de una persona o de una máquina. Programar un sistema así, capaz de superar esta prueba, supone un trabajo considerable, ya que el mismo sistema debería ser capaz de:

- **Procesar el lenguaje natural** para ser capaz de comunicarse de forma satisfactoria.
- **Representar el conocimiento** que posee o extrae, que «siente».
- **Razonar automáticamente** para utilizar la información almacenada para responder a las preguntas o ser capaz de obtener nuevo conocimiento a partir de las conclusiones.
- **Aprender de forma automática** para adaptarse a nuevas situaciones y detectar patrones.

La prueba de Turing evita, por otra parte, la interacción física entre evaluador y computador, bajo el supuesto de que la simulación física del individuo no aporta más a la muestra de inteligencia del computador. No obstante, Turing propuso su *Prueba Global de Turing* en la cual incluía una señal de vídeo que permitiese al evaluador evaluar la capacidad de percepción del evaluado; ésto se debe a que en la prueba original no se tenía en cuenta la interacción física entre evaluador y evaluado. Así, añadimos nuevas características al sistema:

- **Visión computacional** para percibir objetos.
- **Robótica** para manipular objetos.

La prueba de Turing aún está vigente tras más de 50 años. Sin embargo, los investigadores han dedicado poco esfuerzo a la evaluación de sistemas con la Prueba de Turing debido a que consideran que 'es más importante el estudio de los principios de la inteligencia que la duplicación de ejemplares'.

Versiones

Aunque Saúl Traiger sostiene que existen al menos tres versiones principales de la prueba de Turing, en este punto se describirán las dos más importantes: El Juego de Imitación y el Test Estándar.

En el Test Estándar, un sujeto C, que actúa como el interrogador, tiene la labor de determinar cuál de los dos jugadores A y B es un computador y cuál de ellos es un ser humano. Cada uno de los sujetos A, B y C se encuentra separado del resto. El medio para comunicarse es el lenguaje natural en forma escrita. De este modo, el interrogador deberá determinar quien es el computar y quien el ser humano a partir de las respuestas escritas proporcionadas a sus preguntas, transmitidas del mismo modo. Se concluye que el computador supera el Test Estándar de Turing si el juez, el interrogador C, no es capaz de diferenciar por medio de las respuestas quien es el computador.

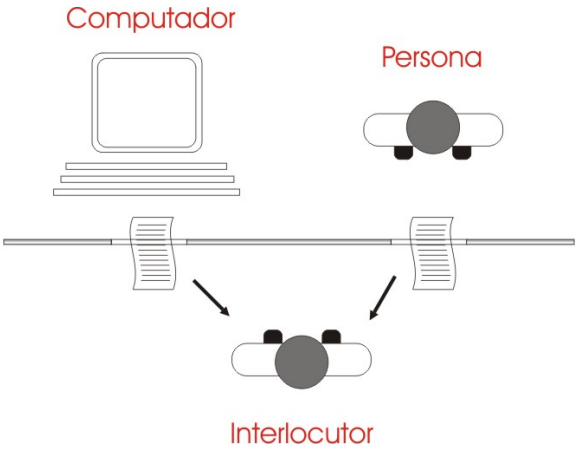


Figura 2.29: Representación gráfica del Test Estándar.

En el Test Estándar ha de hacerse un apunte: el test no comprueba la habilidad de dar la solución correcta a las preguntas, sino de dar la respuesta de forma fiel a como lo haría el ser humano (pues el ser humano puede que no conozca la respuesta de todas las preguntas que se le formulen).

Por otra parte, en el Juego de Imitación, el jugador A es un hombre, cuya función la desarrolla un computador, y el jugador B es una mujer, humana. Se comunican con el interrogador, sujeto C, con lenguaje natural en forma escrita. En esta versión, el interrogador, por medio de preguntas escritas a los jugadores A y B debe determinar cual de los dos es el

hombre y cual la mujer. Sin embargo, en el Juego de Imitación los jugadores A y B no se limitan a contestar a las preguntas, sino que el jugador A intentará engañar al juez para que elija mal, mientras que el jugador B lo ayudará. Si el juez no es capaz de señalar al hombre (interpretado por el computador), se dice que el computador supera el Juego de Imitación.

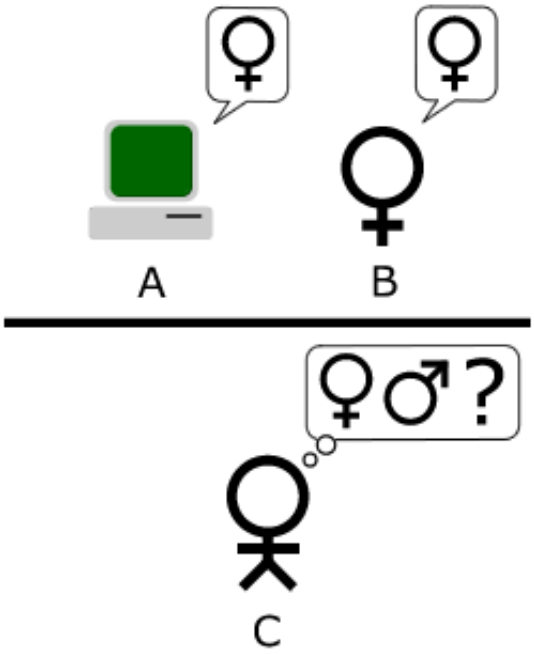


Figura 2.30: Representación gráfica del Juego de Imitación. A es el computador y juega al engaño, y B la mujer real.

También el éxito es medido por la proximidad de las respuestas del computador cuando el jugador A es un hombre humano.

Pensamiento humano: Enfoque cognitivo

Para afirmar que un sistema piensa como un ser humano, primero ha de determinarse cómo piensa un humano. Se lleva a cabo de dos formas: mediante introspección y mediante experimentos psicológicos. Una vez obtenida la teoría de funcionamiento de la mente, se aplica en la programación del computador.

Se afirmará que un sistema piensa como un humano si los tiempos de respuesta y la entrada/salida del programa coincide en gran medida con los de un ser humano.

Pensamiento racional: Enfoque de Leyes del Pensamiento

Este enfoque intenta seguir un proceso de razonamiento irrefutable, basado en el campo de la lógica. Del mismo modo que Aristóteles con sus silogismos, partiendo de premisas

correctas, ha de alcanzarse un conocimiento final –conclusiones- siempre correctas[12].

En el siglo XIX se desarrolló, gracias a los estudiosos en el campo de la lógica, una notación precisa para definir sentencias sobre todo tipo de elementos del mundo y especificar interrelaciones entre los mismos. En 1965 había ya programas que resolvían problemas en notación lógica[13]. Esta construcción de sistemas inteligentes a partir de estos programas se denomina *tradición* ***logística***.

Este enfoque presenta dos problemas:

1. No es fácil transformar conocimiento informal y expresarlo en términos formales lógicos.
2. Se presenta una gran diferencia en poder resolver un problema en la teoría y encontrar la solución en la práctica.

Actuar racionalmente: Enfoque del agente racional

Un **agente**[14] es un ente capaz de razonar (véase sección 2.3.1, **Sistema Racional**). Sin embargo, el agente debe diferenciarse del mero programa informático convencional: se espera que tenga otros atributos, tales como control autónomo, percepción de su entorno o que se adapten a los cambios, entre otros. El **agente racional** actúa con la intención de alcanzar el mejor resultado o, en su defecto, el mejor resultado esperado.

Según las leyes del pensamiento, todo el énfasis se centra en hacer inferencias correctas. La obtención de estas inferencias correctas se condira, por lo general, parte de lo que se considera un agente racional. Sin embargo, obtener inferencias correctas no depende siempre de la *racionalidad*, ya que existen situaciones para las que no hay nada correcto que hacer y en las que hay que tomar otra decisión diferente a la «correcta». También hay formas de actuar racionalmente que no implican hacer inferencias[15].

Estudiar la IA desde el enfoque del agente racional ofrece dos ventajas:

1. Es un enfoque más general que el proporcionado por las «leyes del pensamiento», ya que efectuar inferencias correctas es sólo uno de varios mecanismos que garantizan la racionalidad.
2. Es más afín a la forma en la cual se ha producido el avance científico en este ámbito, dado que la norma de la racionalidad está claramente definida y es de aplicación general.

[12] Por ejemplo: «Sócrates es un hombre; todos los hombres son mortales; por lo tanto, Sócrates es mortal»
[13] Si no se encuentra la solución, el programa nunca debe parar la búsqueda.
[14] *agente* del latín *agere*, hacer
[15] En esta afirmación se enmarcan, por ejemplo, las acciones que se hacen por acto reflejo

2.3.2 Aprendizaje automático

Hemos definido qué es la Inteligencia Artificial y qué se entiende por razonamiento. Se establecen las leyes del pensamiento y razonamientos que un sistema artificial debe mostrar para considerarse «inteligente», pero en ningún momento se cuestiona de dónde provienen esos conocimientos, que ha debido adquirir o **aprender** en algún momento. Por tanto, no podemos hablar de inteligencia sin aprendizaje.

Sin aprendizaje, todo tarea o razonamiento es nueva para el sistema aunque ya la haya llevado a cabo varias veces, por lo que reconstruye una y otra vez las mismas soluciones y repite los mismos errores. Un sistema así no es eficiente; el sistema debe ser capaz de aprender para ser eficiente.

Herb [Sim83] define el aprendizaje como los 'cambios en el sistema que son adaptativos, permitiendo llevar a cabo la misma tarea de un modo más eficiente y eficaz'. Estos cambios mejoran con la experiencia, debido al refinamiento de las habilidades y la adquisición de nuevos conocimientos. La idea de poder aprender de la experiencia en la resolución de problemas nos lleva a esperar mejores soluciones en un futuro.

Refinando aún más la definición de aprendizaje, éste es la captura y transformación de conocimiento en un formato utilizable por el sistema, para mejorar el rendimiento en la resolución de problemas. Problemas que podrían no ser resolubles o suficientemente precisos sin aprendizaje.

Taxonomía

El aprendizaje automático, o *Machine Learning*, está basado en el aprendizaje humano, y es la clave de la inteligencia artificial como medio de obtener sistemas inteligentes. En este tipo de aprendizaje el proceso de adquisición del conocimiento puede ser de diferentes tipos, según diferentes criterios (taxonomía).

Según el grado de realimentación, puede ser:

- **Aprendizaje Supervisado**: El proceso de aprendizaje es apoyado por otros (sistemas, muestras...), contando con los valores reales que debe producir en cada caso.
- **Aprendizaje No supervisado**: El proceso se adquiere sin ayuda de otros; descubre patrones en los datos y forma agrupaciones.
- **Aprendizaje por Refuerzo**: El sistema recibe refuerzos positivos o negativos cuando produce una respuesta, ajustando su comportamiento según el refuerzo.

Según el grado de paradigma, puede ser:

- **Aprendizaje Inductivo**: Se establece el razonamiento a partir de un conjunto de ejemplos para producir reglas generales o procedimientos. El sistema debe razonar sobre nuevas instancias aunque no habrá garantía de que será correcto.

- **Aprendizaje Analítico**: Extrae conclusiones de conjuntos pequeños de observaciones. Se emplean observaciones tanto positivas como negativas. Parte de muy pocos ejemplos junto con una teoría del dominio.
- **Algoritmos genéticos**: Inspirados en mecanismos biológicos como la selección natural, la recombinación o las mutaciones aleatorias. Comienzan con una población de individuos reducida y aplican operadores genéticos a ciertos individuos. En cada iteración se evalúan estos individuos y sólo los más convenientes sobreviven. El objetivo es encontrar ejemplos que maximicen las conveniencias.
- **Algoritmos conexionistas (redes neuronales)**: Inspirados en la interconectividad en el cerebro, establecen capas de nodos (simulan las neuronas) muy interconectados entre si, capaces de aprender complejas funciones. Muy útiles en reconocimiento de patrones.

Según el problema que resuelve, puede ser:

- **Aprendizaje de resolución de problemas**: Utiliza problemas y datos empíricos de los que extraer reglas y conocimientos para resolver problemas semejantes.
- **Aprendizaje de conceptos**: Los datos se clasifican de acuerdo a una o más categorías predefinidas (conceptos).

Modelo de Conocimiento

No obstante, ningún tipo de aprendizaje de los descritos en el apartado anterior puede llevarse a cabo sin una representación inteligible por el sistema artificial del conocimiento del ámbito en cuestión. Esta representación simbólica de lo que se sabe sobre el ámbito de un dominio determinado se denomina **Modelo de Conocimiento**.

La principal característica que todo modelo posee es la *computabilidad*; el modelo debe ser computable por el sistema. Sin embargo, no se puede conseguir un modelo computable que contenga todo el conocimiento del dominio. Generalmente, el modelo es creado por el Ingeniero del Conocimiento, que conoce métodos y herramientas para desarrollar los modelos. Además se relaciona con expertos, fuentes documentales, etc. para obtener mayor dominio del campo en cuestión. De acuerdo a ellos, y a sus habilidades, elabora los modelos de conocimiento.

Como se aprecia en la figura 2.31, el conocimiento realmente representado es mucho menor al conocimiento existente en el dominio, debido a que ni siquiera el experto a quien consulta tiene todo el conocimiento, el ingeniero no puede obtener la totalidad del conocimiento del experto, y por último puede que ni siquiera todo el conocimiento captado sea formalizable. Por lo que elaborar el modelo de conocimiento es una de las labores más complejas del aprendizaje automático.

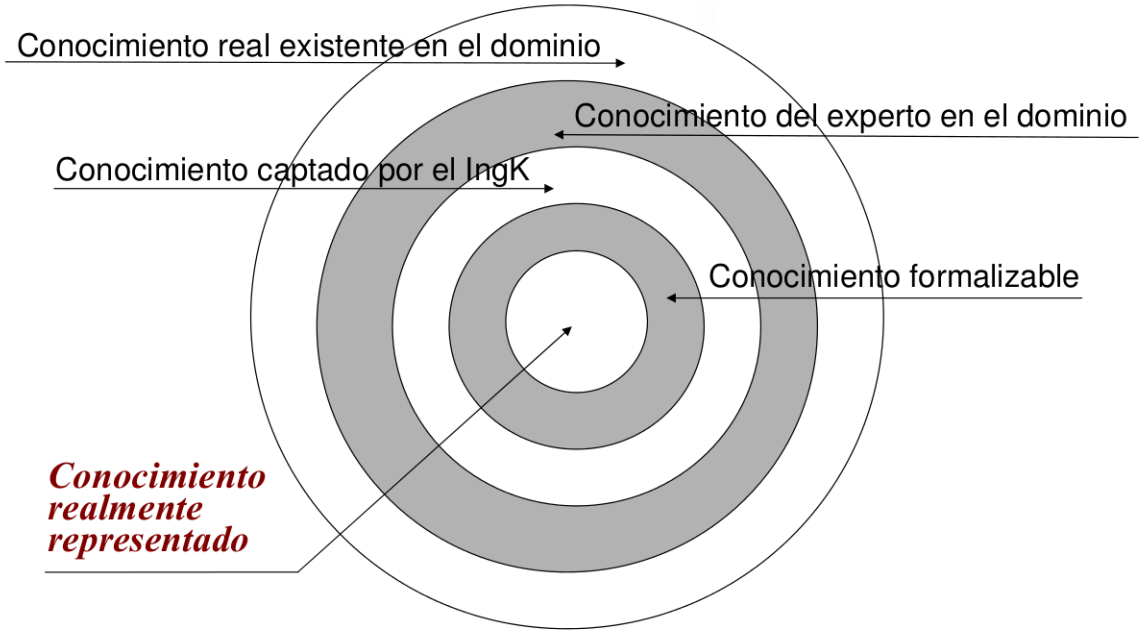


Figura 2.31: Conocimiento representado en el aprendizaje automático.

Además, este proceso es iterativo e incremental, por lo que siempre puede completarse añadiendo más conocimiento o mejorando el ya representado, aunque siga siendo de un modo subjetivo (el del ingeniero del conocimiento). El proceso está muy ligado además con la lingüística y la semántica.

2.3.3 Técnicas fundamentales de la Inteligencia Artificial

En las últimas décadas ha tenido lugar un cambio social profundo debido al avance de la tecnología. La presencia de nuevos paradigmas en el tratamiento de la información, denominado "Gestión del Conocimiento", al igual que el procesamiento masivo de esta información, han contribuido a dicho avance.

Con el propósito de abordar estos paradigmas se han ido desarrollando técnicas de la Inteligencia Artificial y Aprendizaje Automático que deben ser capaces de considerar la información cualitativa y, a partir de ella, diseñar e implementar modelos de conocimiento computacionales mediante los cuales resolver los diversos problemas que se presenten.

En las secciones siguientes se presentan una serie de técnicas de la Inteligencia Artificial relacionadas estrechamente con los objetivos de este Proyecto Fin de Carrera y, para finalizar, se estudia en más detalle la principal técnica de la IA utilizada: las redes neuronales.

Redes bayesianas

También denotadas *redes causales probabilísticas*, son herramientas estadísticas que representan un conjunto de incertidumbres asociadas sobre la base de relaciones de independencia condicional que se establecen entre ellas. Dicho de otra forma, la red bayesiana es un conjunto de variables, con una estructura gráfica que las conecta y un conjunto de distribuciones de probabilidad condicional sobre ellas.

En una red bayesiana, cada variable es un nodo, que a su vez representa una entidad del mundo real. En la red bayesiana más simple encontraremos dos nodos (o variables) y un arco desde la primera hasta la segunda, como puede apreciarse en la Figura 2.32.

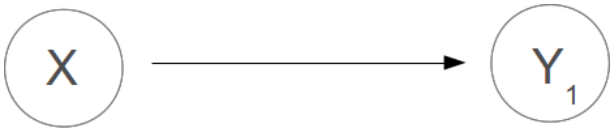


Figura 2.32: Red bayesiana simple. X puede ser la prueba que confirma dicha enfermedad Y, por ejemplo.

Definición intuitiva

Si X es una variable binaria, su presencia se denota con $+x$, y su ausencia por $\neg x$. La información cuantitativa de una red viene dada por:

- La probabilidad a priori de los nodos que no tienen padres.
- La probabilidad condicionada de los nodos con padres.

Ejemplo de red bayesiana:

- P(+x) = 0.003 indica la probabilidad a priori del suceso *X*, un 0.3 %.
- P(+y_1/+x) = 0.992 indica que el suceso *Y* condicionado a *X* tiene un 99.2 % de probabilidad de ocurrir.
- P(+y_1/$\neg$x) = 0.0006 indica que el suceso Y, cuando no está condicionado por la presencia de *X*, tiene un 0.06 % de probabilidad de ocurrencia.

A partir del teorema de Bayes podríamos, por ejemplo, calcular la prioridad a priori de Y_1,

$$P(+y_1) = P(+y_1/+x)P(+x) \; + \; P(+y_1/\neg x)P(\neg x) = 0{,}00357. \tag{2.2}$$

$$P(\neg y_1) = P(\neg y_1/+x)P(+x) \; + \; P(\neg y_1/\neg x)P(\neg x) = 0{,}99643. \tag{2.3}$$

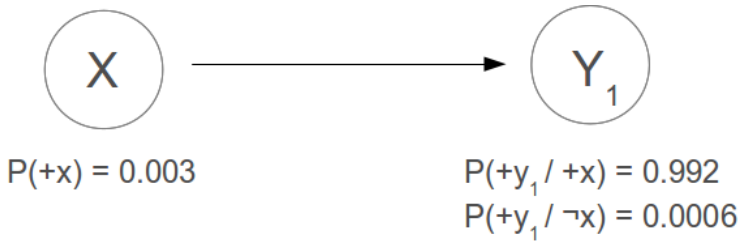


Figura 2.33: Red bayesiana simple completa.

Definición formal

Antes de dar una definición formal es necesario definir otros conceptos básicos:

Arco: Par ordenado (x, y). Se representa gráficamente con una flecha.

Grafo dirigido: Es un par G = (N, A) donde N es un conjunto de nodos y A es un conjunto de arcos definidos sobre los nodos.

Camino: Es una secuencia ordenada de nodos $(X_{i1}, \ldots, X_{ir}$ tal que $\forall j = 1, \ldots, r-1 \mid X_j \longrightarrow X_{j+1}$ ó $X_{j+1} \longrightarrow X_j \in A$.

Camino dirigido: Es una secuencia ordenada de nodos $(X_{i1}, \ldots, X_{ir}$ tal que $\forall j = 1, \ldots, r-1 \mid X_j \longrightarrow X_{j+1}$.

Ciclo: Es un camino no dirigido que empieza en el mismo nodo X que acaba.

Grafo acíclico: Grafo que no contiene ciclos.

Padre: X es un padre de Y si y sólo si existe un arco X $\rightarrow$ Y. Se dice también que Y es hijo de X.

Antepasado o ascendiente: X es un antepasado o ascendiente de Z si y sólo si existe un camino dirigido de X a Z.

Descendiente: Z es un descendiente de X si y sólo si X es un antepasado de Z.

Una red bayesana es un grafo, definido como un par G = (V, E), donde V es un conjunto finito de vértices, nodos o variables y E es un subconjunto del producto cartesiano V x V de pares ordenados de nodos llamados enlaces o aristas. Concretamente, es un grafo de tipo dirigido acíclico (GDA). Es dirigido porque los enlaces entre los vértices de la estructura están orientados; es acíclico porque no pueden existir ciclos o bucles en el grafo.

A $\rightarrow$ B indica relación directa entre las variables; se representa que B depende de A –o que A es la causa de B y B el efecto de A-. También se dice que A es padre de B y que B es el hijo de A. Aunque la presencia de arcos entre nodos codifica información esencial sobre el modelo representado en la red, la ausencia de arcos entre nodos aporta una valiosa información ya que el grafo codifica independencia condicional.

Principio de Independencia condicional

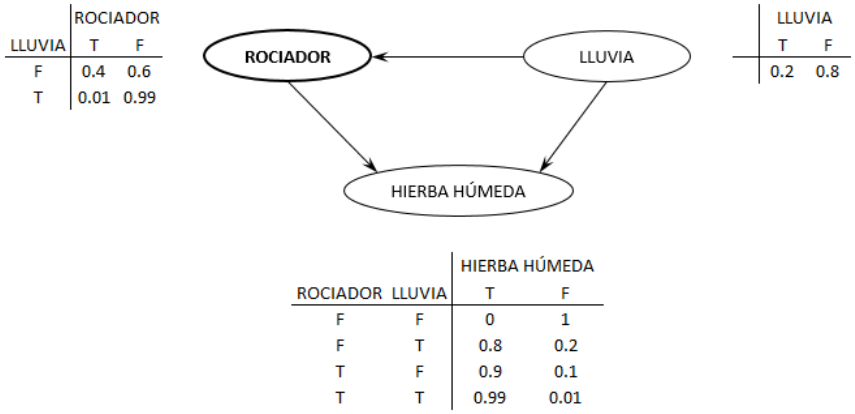
Def: Un grafo acíclico, conexo y dirigido G = (V, E), y una distribución de probabilidad conjunta P definida sobre las variables del grafo, se dice que cumplen las hipótesis de independencia condicional si para toda variable X de V el conjunto de los padres directos de X (denotado por pa(X)) separa condicionalmente a X de todo otro nodo Y de la red que no sea X, ni sus descendientes ni sus padres.

$$\forall X \in V \text{ y } \forall Y \subset V - \{X \cup de(X) \cup pa(X)\} \text{ se tiene que } P(X/pa(X), Y) = P(X/pa(X)). \tag{2.4}$$

donde de(X) denota al conjunto de descendientes de X.

Ejemplo

Supongamos dos causas principales de que cierta hierba esté humeda: el rociador (aspersor) está activo o está lloviendo. La lluvia además tiene efecto directo sobre el aspersor, ya que si llueve no se activa. La red queda descrita como la imagen 2.34.



LLUVIA	ROCIADOR T	ROCIADOR F
F	0.4	0.6
T	0.01	0.99

LLUVIA T	LLUVIA F
0.2	0.8

ROCIADOR	LLUVIA	HIERBA HÚMEDA T	HIERBA HÚMEDA F
F	F	0	1
F	T	0.8	0.2
T	F	0.9	0.1
T	T	0.99	0.01

Figura 2.34: Representación del problema de Red Bayesiana. T y F son los valores de Verdadero y Falso respectivamente (True y False).

La función de probabilidad conjunta es:

$$P(G, S, R) = P(G|S, R)P(S|R)P(R) \tag{2.5}$$

donde G = Hierba húmeda(Grass), S = Rociador activo y R = Lloviendo (Raining).

Ahora queremos saber, por ejemplo, *cuál es la probabilidad de que esté lloviendo dado que la hierba se encuentra húmeda.* Dada la fórmula de la probabilidad condicional:

$$P(B|A) = \frac{P(B \cap A)}{P(A)} \tag{2.6}$$

y operando las variables, tenemos:

$$P(R = T|G = T) = \frac{P(G = T, R = T)}{P(G = T)} = \frac{\sum_{S \in T,F} P(G = T, S, R = T)}{\sum_{S,R \in T,F} P(G = T, S, R)} \tag{2.7}$$

Sustituyendo los valores de la imagen 2.34 en la página anterior en la ecuación 2.7, obtenemos:

$$= \frac{(0{,}99 * 0{,}01 * 0{,}2 = 0{,}00198_{TTT}) + (0{,}8 * 0{,}99 * 0{,}2 = 0{,}1584_{TFT})}{0{,}00198_{TTT} + 0{,}288_{TTF} + 0{,}1584_{TFT} + 0_{TFF}} \simeq 35{,}77\,\%. \tag{2.8}$$

Concluimos que *la probabilidad de que esté lloviendo dado que la hierba se encuentra húmeda* es de **un 35 %**.

Aplicaciones

Las redes Bayesianas se utilizan para modelar conocimiento. Entre sus usos más frecuentes podemos citar:

- Biología computacional y bioinformática.
- Clasificación de documentos.
- Recuperación de información.
- Búsqueda semántica.
- Procesamiento de imágenes.
- Sistemas de soporte de decisiones.
- Juegos.

Inconvenientes

Las redes bayesianas presentan una serie de inconvenientes que aumentan la dificultad del diseño y su utilización en este Proyecto Fin de Carrera:

- El proceso de creación de la red lo realizan casi exclusivamente los expertos en el dominio modelado. Por tanto es necesario un experto en el dominio para crear la red.
- Aunque existen métodos que permiten adaptar la red e implementar mecanismos de aprendizaje, a priori resulta difícil adaptarla en un entorno dinámico.

Algoritmos genéticos

Los algoritmos genéticos son una variante de búsqueda de soluciones de haz estocástica[16], en la que los estados sucesores se generan por combinación de dos estados padre. Un algoritmo genético es una analogía de la selección natural, al igual que la búsqueda de haz estocástica, pero con reproducción sexual (dos estados padre) en lugar de reproducción asexual (un estado padre).

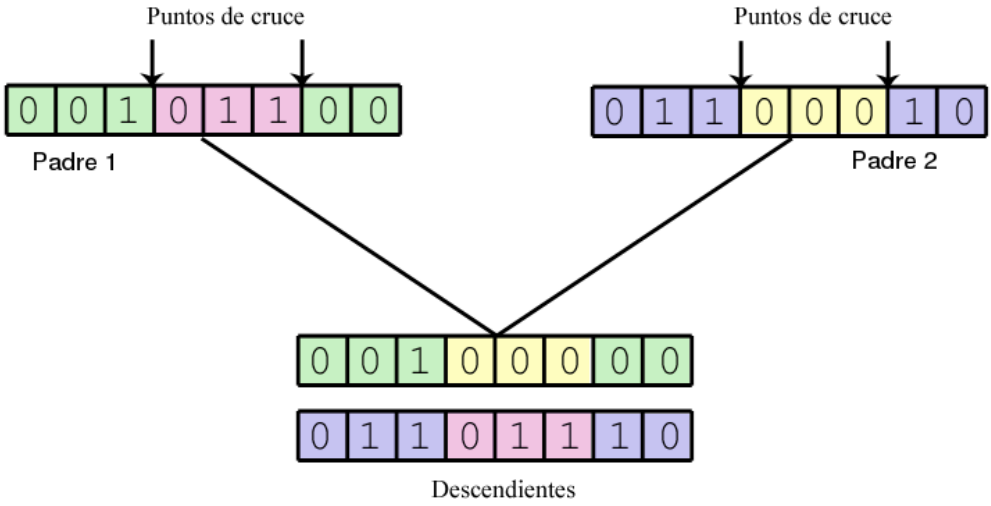


Figura 2.35: Proceso de cruce en los algoritmos genéticos.

Se comienza con un conjunto *k* de estados generados aleatoriamente, conformando la **población** inicial. Cada estado (**individuo**) está representado como una cadena finita sobre un alfabeto finito[17]. La producción de la siguiente **generación** de la población se lleva a cabo mediante la aplicación a cada estado de la **función idoneidad**; esta función devuelve valores más altos para los mejores especímenes (estados mejores). Con estos valores se seleccionan pares y, por cada par, se elige aleatoriamente un **punto de cruce** a partir del cual cruzar cadenas. Por último, cada posición puede sufrir **mutación** tras el cruce (probabilidad pequeña); la mutación transforma un dígito de la cadena en otro. El listado 2.1 en la página siguiente muestra todos estos pasos mediante pseudocódigo.

Como en la búsqueda por haz estocástica, los algoritmos genéticos combinan una tenencia ascendente con exploración aleatoria y cambian la información entre los hijos paralelos de búsqueda.

[16] Algoritmo de búsqueda en el cual los estados sucesores (descendientes) de un estado (organismo) pueblan la siguiente generación según su valor (idoneidad, salud, adaptabilidad).

[17] Lo más común es utilizar cadenas de 0s y 1s.

```
funcion ALGORITMO-GENETICO (poblacion, IDONEIDAD) devuelve un
    individuo
        entradas: poblacion, un conjunto de individuos
        IDONEIDAD, una funcion que mide la capacidad de un individuo

        repetir
                nueva_poblacion ⟵ conjunto vacio
                bucle para i desde 1 hasta TAMANO(poblacion) hacer
                        x ⟵ SELECCION-ALEATORIA (poblacion,
                            IDONEIDAD)
                        y ⟵ SELECCION-ALEATORIA (poblacion,
                            IDONEIDAD)
                        hijo ⟵ REPRODUCIR (x,y)

                        si (probabilidad aleatoria pequena) entonces
                            hijo ⟵ MUTAR (hijo) anadir hijo a
                            nueva_poblacion
                poblacion ⟵ nueva_poblacion

        hasta que algun individuo es bastante adecuado, o ha pasado
            bastante tiempo
        devolver el mejor individuo en la poblacion
```

```
funcion REPRODUCIR (x, y) devuelve un individuo
        entradas: x, y, ambos padres

        n ⟵ LONGITUD(x)
        c ⟵ numero aleatorio de 1 a n
        devolver ANADIR (SUBCADENA(x, 1, c), SUBCADENA(y, c + 1, n))
```

Listado 2.1: Algoritmo genético.

Ejemplo

Pongamos como ejemplo el problema de las 8 reinas en el tablero de ajedrez sin amenazarse. Un estado del problema en este caso debe especificar las posiciones de las ocho reinas, cada una en una columna. Dado que las filas y columnas del tablero de ajedrez son 8 en cada caso, se requerirán $8 * \log_2 8 = 24$ bits; otra forma de representarlo sería una secuencia de 8 dígitos, cada uno de los cuales pertenece al rango del 1 al 8.

Dada una población inicial, como la mostrada en la figura 2.36 en la página siguiente (a), se pretende resolver el problema. El primer paso es evaluar el estado mediante la función idoneidad: en este caso, la función idoneidad devolverá un número entero con el valor de pares de reinas no atacadas (figura 2.36 en la página siguiente (b)), siendo el valor de la solución 28.

En figura 2.36 en la página siguiente (c) se selecciona de forma aleatoria los individuos y posteriormente se cruzan en (d). En este caso podemos ver que se seleccionó un individuo 2 veces, y se produjeron cruces con los individuos 2-1 y 2-3. En el primer caso el punto de

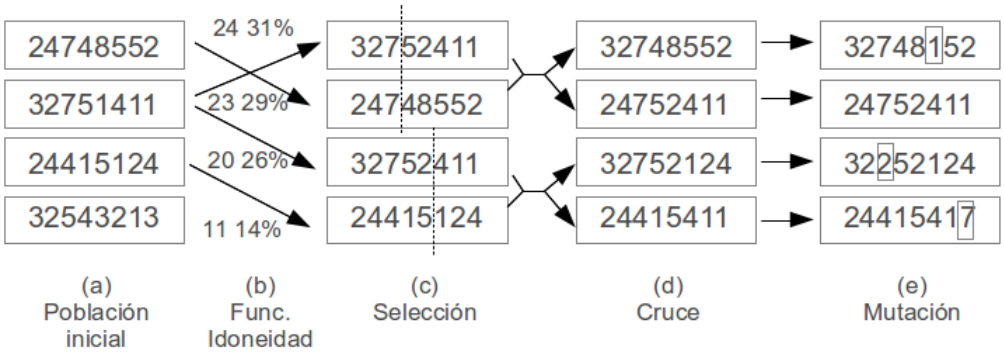


Figura 2.36: Ejemplo de algoritmo genético para el problema de las 8 reinas. Imagen obtenida y editada de [RN04].

cruce fue a partir del tercer dígito; en el segundo cruce, a partir del quinto dígito.

En última instancia, en algunas ocasiones ocurrieron mutaciones, cambiando el valor de un dígito aleatorio, como se aprecia en (e). Repitiendo este proceso de forma iterativa se alcanzará la/una solución del problema. En el caso del ajedrez, por ejemplo, la cadena **35281746**[18].

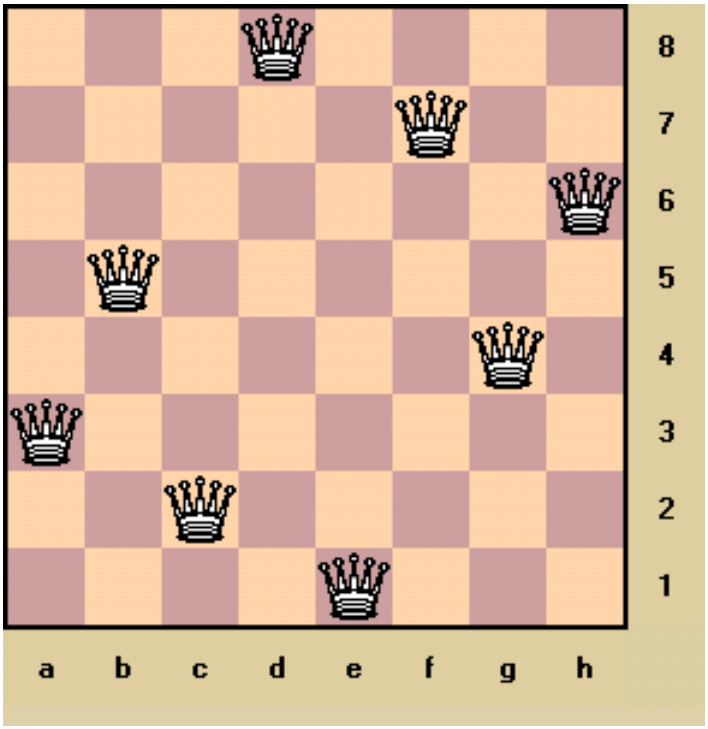


Figura 2.37: Solución al problema de las 8 reinas con la cadena 35281746.

Aplicaciones

La naturaleza de los algoritmos genéticos y potencia a la hora de revolver problemas ha

[18]El problema de las 8 reinas tiene varias soluciones.

servido para que se hayan aplicado a diversos problemas y modelos en ingeniaría, y en la ciencia en general, destacando:

- **Optimización**: Tareas de optimizacion como optimización numérica y combinatoria.
- **Programación automática**: Desarrollar programas para tareas específicas y estructuras computacionales como el autómata celular y las redes de clasificación.
- **Aprendizaje máquina**: Aplicaciones tales como el desarrollo de los pesos en una red neuronal, las reglas para sistemas de clasificación o sistemas de producción simbólica.
- **Economía**: Modelizar procesos de innovación y desarrollo de estrategias de puja.
- **Ecología**: Modelización de fenómenos ecológicos como armamento biológico, simbiosis y coevaluación de la relación parásito-huésped.
- **Evolución y aprendizaje**: Estudio de relaciones entre aprendizaje individual y evolución de la especie.
- **Sistemas sociales**: Estudio de los aspectos de los sistemas sociales, como la evolución del comportamiento social en grupo de individuos, la cooperación y la comunicación en sistemas multi-agente.

Inconvenientes

Los algoritmos genéticos presentan una serie de inconvenientes que aumentan la dificultad del diseño y su utilización en este Proyecto Fin de Carrera:

- Elegir una codificación numérica correcta para el dominio puede ser una labor demasiado complicada o difícilmente aplicable.
- Puede que tarden demasiado en converger o no llegar a converger nunca, de modo que no se alcanza solución. En el caso contrario, pueden converger prematuramente dando lugar a una solución poco óptima.

Lógica difusa

La lógica difusa es un tipo de lógica multivaluada, una extensión de la lógica convencional que expande el concepto de verdad al concepto de verdad parcial. En la lógica tradicional las variables toman dos únicos valores: verdadero o falso, X o no X; en la lógica difusa, el valor de las variables no es dicotómico, pertenece al rango entre 0 y 1. Esta asignación es más próxima al modo de razonamiento humano que la lógica booleana.

El concepto de multivalencia es el concepto fundamental utilizado por Lofti Zadeh en la década de los 60s para introducir la lógica difusa. Las características esenciales de la lógica difusa son las siguientes:

- El razonamiento exacto es un caso límite del razonamiento aproximado.
- Todo es un grado de verdad.
- Todo sistema lógico tradicional puede ser implementado con lógica difusa.
- El conocimiento se interpreta de acuerdo a un conjunto de restricciones difusas sobre las variables.
- La inferencia es el proceso de propagación de las restricciones.

La tercera característica define la lógica tradicional como un subconjunto de la lógica difusa. Esto nos lleva a la teoría y definición de los conjuntos difusos o conjuntos borrosos.

Conjuntos Difusos Los conjuntos difusos se componen de dos elementos: el conjunto difuso en sí mismo, que es un conjunto formado por etiquetas lingüísticas, y la etiqueta lingüística, que es una palabra o conjunto de palabras que nombran a los conjuntos. A diferencia de los conjuntos clásicos, donde se sabe si un elemento del universo del discurso pertenece a él o no, se acude a la lógica booleana: 0 no pertenece, 1 pertenece. En el caso del conjunto borroso, la pertenencia de un elemento del universo del discurso a él está definida en el intervalo [0, 1], y el valor asociado no será absoluto, sino gradual (todo es un grado, 3ª característica). Esto es conocido como el *grado de pertenencia* de un elemento.

Por otra parte, el *universo del discurso* de un conjunto difuso es el intervalo en el que se incluyen los posibles valores que pueden tomar los elementos del conjunto. Independientemente de cuales sean estos valores y su codificación, siempre estarán normalizados al intervalo [0,1].

La definición formal, derivada de lo expuesto anteriormente, y su principal principio, se establece como:

Conjunto difuso: Es un conjunto de pares, de forma que el primer elemento del par es un número real contenido en el intervalo [0,1] que indica el grado de pertenencia de un elemento y el segundo elemento es dicho elemento. El primer elemento puede ser también la etiqueta lingüística codificada.

Principio de extensión: Si existe una función que asocie elementos o puntos de un conjunto clásico X a elementos o puntos de otro conjunto clásico Y, esta puede generalizarse, de forma que puedan asociarse subconjuntos borrosos de X a subconjuntos borrosos de Y.

Operaciones básicas de los conjuntos difusos Con un conjunto difuso pueden realizarse tres operaciones básicas:

1. **Intersección:** El resultado de la intersección de dos conjuntos borrosos es un conjunto borroso en el cual se encuentran los elementos de ambos conjuntos. De forma gráfica,

si superponemos ambas gráficas de pertenencia, la intersección es la zona en la que coinciden ambas.

2. **Unión:** El resultado es un conjunto en el que se encuentren todos aquellos elementos de un conjunto que no existen en el otro. Gráficamente, serían aquellas zonas en las que la superposición de ambas funciones de pertenencia no tuviera puntos en común.
3. **Negación:** La negación de un conjunto son todos aquellos elementos que forman parte de su universo de discurso, pero no forman parte del conjunto.

Lofti Zadeh sugirió en su primer artículo que la intersección era el operador mínimo de los conjuntos difusos y la unión el operador máximo.

En las figuras 2.38 y 2.39 tenemos dos conjuntos difusos A y B respectivamente. En las figuras 2.40, 2.41 y 2.42 se muestran la intersección, la unión y la negación de ambos conjuntos.

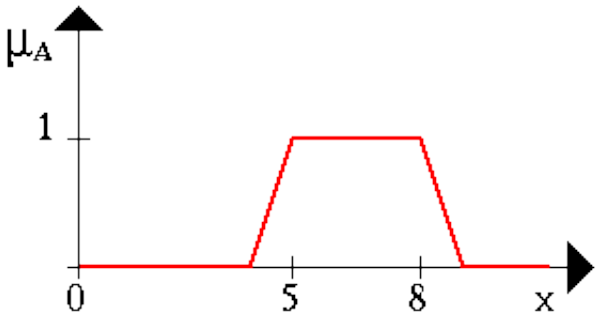


Figura 2.38: Conjunto difuso A.

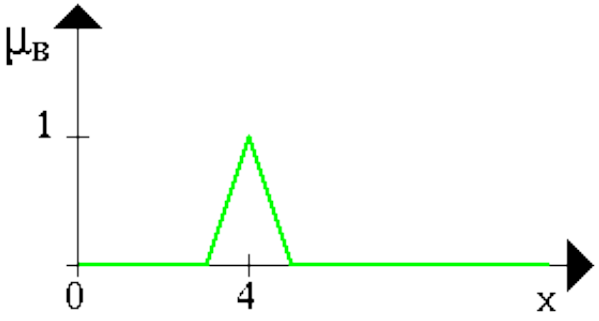


Figura 2.39: Conjunto difuso B.

Ejemplo

Pongamos el ejemplo de un sistema de conducción automática. Queremos que el coche acelere y frene de forma automática. En una primera aproximación, podríamos decir «si quiero

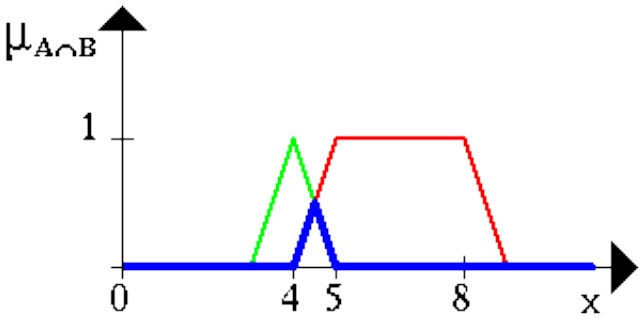


Figura 2.40: Intersección de los conjuntos A y B, en azul el resultado.

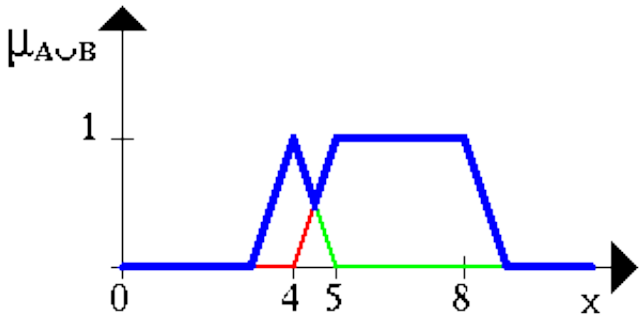


Figura 2.41: Unión de los conjuntos A y B, en azul el resultado.

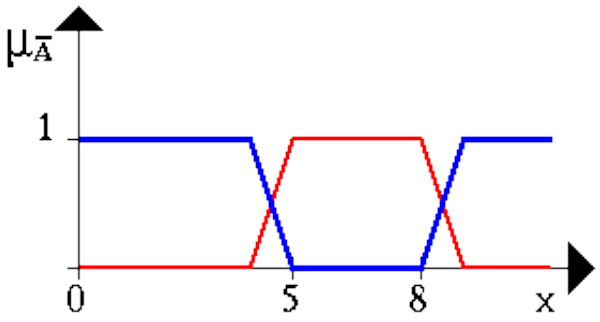


Figura 2.42: Negación del conjunto difuso A.

parar, entonces frenar», pero esto haría que el coche frenase brúscamente; en su lugar, deberíamos codificarlo «si quiero parar y la velocidad es alta, frenar ligeramente» y «si quiero parar y la velocidad es baja o muy baja, frenar al máximo». Para ello, debemos *«fuzzificar el dominio de la velocidad»*. Una posible codificación podría ser la mostrada en la figura 2.43.

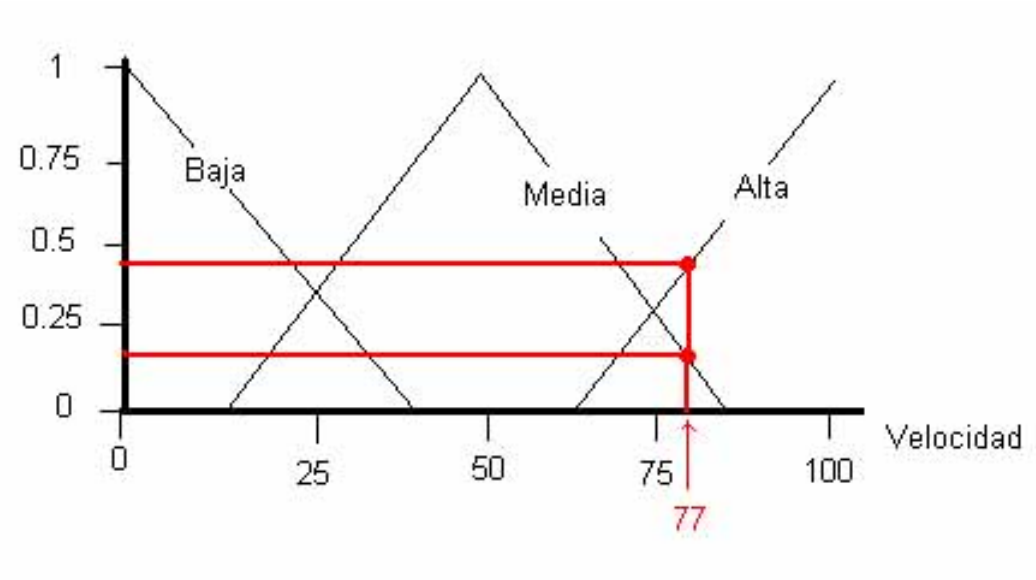


Figura 2.43: Sistema difuso ejemplo.

El eje vertical indica el valor de verdad de cada velocidad. El eje horizontal la velocidad.

Imaginemos que el coche viaja a 77km/h, operaríamos de la siguiente forma:

1. Buscamos en el eje horizontal la velocidad a la que vamos y trazamos una vertical marcando las intersecciones con los triángulos.

2. En este caso tenemos dos intersecciones, la primera intersecta con el triángulo de Velocidad Media a una altura de 0.20, la segunda intersección corta con el triángulo de Velocidad Alta a una altura de 0.45.

3. El sistema concluye que a 77km/h el coche va más bien a una velocidad alta más que a una una velocidad media; si quisiese parar en este momento, debería hacerlo de forma más bien ligera en lugar de frenar bruscamente.

Para cualquier valor en general, deberíamos aplicar la función de pertenencia al conjunto. En el caso de velocidad baja, la función de pertenencia es:

$$\Pi(u;a,b) = \begin{cases} 1 & \text{si } u = a \\ 1 - \frac{u}{b} & \text{si } a < u < b \\ 0 & \text{si } u \geq b \end{cases} \tag{2.9}$$

Para Velocidad Media:

$$\Pi(u;a,b,c) = \begin{cases} 1 & \text{si } u = b \\ \frac{u}{b} & \text{si } a < u < b \\ 1 - \frac{u}{c} & \text{si } b < u < c \\ 0 & \text{si } u \geq c, u \leq a \end{cases} \tag{2.10}$$

Para Velocidad Alta:

$$\Pi(u;a,b) = \begin{cases} 1 & \text{si } u \geq b \\ \frac{u}{b} & \text{si } a < u < b \\ 0 & \text{si } u \leq a \end{cases} \tag{2.11}$$

Aplicaciones

Se ha probado que los sistemas difusos son excelentes aproximadores universales, especialmente cuando la complejidad del proceso es muy alta y no existen modelos matemáticos precisos, para procesos altamente no lineales y cuando se envuelven definiciones y conocimiento impreciso o subjetivo.

Aunque los sistemas difusos son buenos aproximadores de funciones algebraicas, no es su principal atributo. Son muy buenos aproximadores de comportamiento en los cuales las funciones analíticas y las relaciones numéricas no existen; la lógica difusa o sistemas difusos poseen un alto potencial de entendimiento de sistemas complejos.

Hablando en un contexto general, podemos citar dos aplicaciones:

1. Resolución de situaciones que involucran sistemas complejos cuyo comportamiento no se entiende en su totalidad.
2. Resolución de situaciones que requieren de una solución rápida lo más aproximada posible.

Como aplicaciones específicas, cabe resaltar su uso en el campo de la industria en:

- Sistemas de control de acondicionadores de aire.
- Sistemas de foco automático en cámaras fotográficas.
- Electrodomésticos inteligentes.
- Optimización de sistemas de control.
- Sistemas de escritura.

- Eficiencia de uso de combustible en motores.
- Simuladores de comportamiento humano.
- Bases de datos difusas, que manipulan información imprecisa.
- Sistemas de control complejos cuyas acciones son diferentes a *Sí/No*.

Centrándonos en el ámbito de la Inteligencia Artificial, la principal aplicación de la lógica difusa es la expansión de la dicotomía de verdad (verdad o no verdad) para incluir un rango de valores intermedios, que permitan simular de una forma más precisa las reglas de decisión humanas, de forma que puedan construirse sistemas de IA más precisos e «inteligentes». Pongamos el ejemplo de un sistema de refrigeración de un motor: en lugar de actuar solo en base a si el motor está caliente o no lo está, la lógica difusa nos permite definir varios grados de temperatura en el motor, como *muy* caliente, *ligeramente* caliente, etc. y actuar de forma diferente dependiendo del estado. En resumen, establecen reglas en sistemas complejos que son «reglas humanas».

Inconvenientes

La lógica difusa presenta una serie de inconvenientes que aumentan la dificultad del diseño y su utilización en este Proyecto Fin de Carrera:

- Aunque es posible definir un sistema de reglas difuso que modele el comportamiento del sistema, es necesario un experto en el dominio para su construcción.
- Las técnicas de aprendizaje automático basadas en Lógica Difusa resultaban inicialmente más complejas que el uso de la técnica que estudiaremos a continuación: Redes Neuronales.

Redes neuronales

En esta sección se explicará en profundidad este campo de la Inteligencia Artificial. Se definirá qué se entiende por red neuronal; cuál es su origen, enfocado en el ámbito biológico del que parte; qué elementos componen la red, y bajo qué arquitecturas se presentan [LIM08]; y se resaltará el modelo elegido en este PFC.

Definición

Una red neuronal artificial es un modelo, inspirado en el sistema nervioso animal –particularmente el cerebro-, capaz obtener conocimiento y patrones cognitivos a partir de aprendizaje artificial; sistemas computacionales que aprenden a partir de datos. Se presentan como sistemas

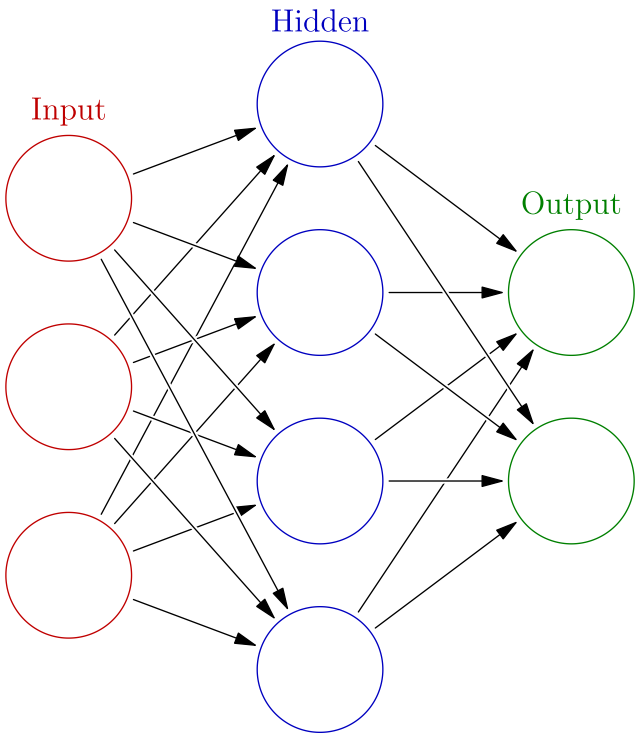


Figura 2.44: Representación gráfica de una red neuronal. Fuente: `http://en.wikipedia.org/wiki/Artificial_neural_network`

de «neuronas» interconectadas capaces de computar valores de entrada y propagar la información por la red.

Gráficamente, podemos representar la red neuronal como en la Figura 2.44; la red neuronal tiene grupos de nodos interconectados, llamados neuronas. Cada círculo representa una de estas neuronas y cada flecha representa una conexión desde la salida de una neurona a la entrada de otra. No es mandatorio que de cada neurona salgan tantas flechas hacia otras como neuronas haya en la siguiente capa, ni que entren a otra tantas como neuronas en la capa anterior.

Introducción biológica

En 1888, Ramón y Cajal demuestra que el sistema nervioso está compuesto por una red de células individuales ampliamente interconectadas entre sí: las neuronas.

La información fluye desde las dendritas hacia el axón, atravesando el soma, y siendo éstos conceptos:

Soma: Cuerpo celular de la neurona (10-80 micras de longitud).

Dendrita: Ramificaciones surgidas del soma; denominado árbol dendrítico también.

Axón: Fibra tubular que parte del soma y que se ramifica para conectar con otras neuronas (100 micras hasta 1 metro de longitud).

Las neuronas se entienden como procesadores sencillos de información. Analizando en forma de sistema con entrada/salida, tenemos que el soma es el órgano de cómputo de la neurona, las dendritas constituyen el canal de entrada de la información a la neurona y el axón constituye el canal de salida restante. Cada neurona recibe información de unas 10.000 neuronas y envía impulsos a algunas menos. Otras neuronas, las neuronas exteriores, reciben la información directamente del exterior y no de otras neuronas.

Extrapolando ésto con la red de la Figura 2.44 tenemos:

- Los circulitos representando la neurona representan la unidad de cómputo, el soma.
- Las flechas de entrada a una neurona representan las dendritas.
- Las flechas de salida a otra neurona representan el axón.

Del mismo modo que el cerebro se modula durante el desarrollo del ser vivo, la red neuronal se ajustará durante el proceso de entrenamiento de la red. En este proceso, el sistema neuronal puede modelarse:

- Estableciendo nuevas conexiones.
- Rompiendo conexiones existentes.
- Modificando las "pesos"de las uniones existentes.
- Mediante muerte y reproducción nueronal (en la red neuronal artificial, ésto refleja que una neurona o unidad de cómputo se active o desactive para el cómputo final).

De forma similar al sistema nervioso vivo, la red neuronal intentará emular tres conceptos. Por un lado, emulará el *procesamiento paralelo* con el que trabajan los miles de millones de neuronas involucradas en determinado proceso. Emulará la *distribución de memoria* que está presente en las redes neuronales biológicas, a diferencia de las posiciones bien definidas de memoria presentes en un computador. Por ultimo, emulará la *adaptabilidad al entorno*, por medio de la información de las sinapsis ("pesos"de las conexiones entre neuronas).

Elementos de una red neuronal

El elemento básico es la *neurona artificial*, organizada en *capas de neuronas*; varias capas compondrán la *red neuronal artificial*. La red neuronal junto a los interfaces de entrada y salida conformarán el *sistema global de proceso*.

Siguiendo el modelo estándar según los principios descritos en [RM86] y [MR86], la $i-$ésima neurona artificial estándar consiste en:

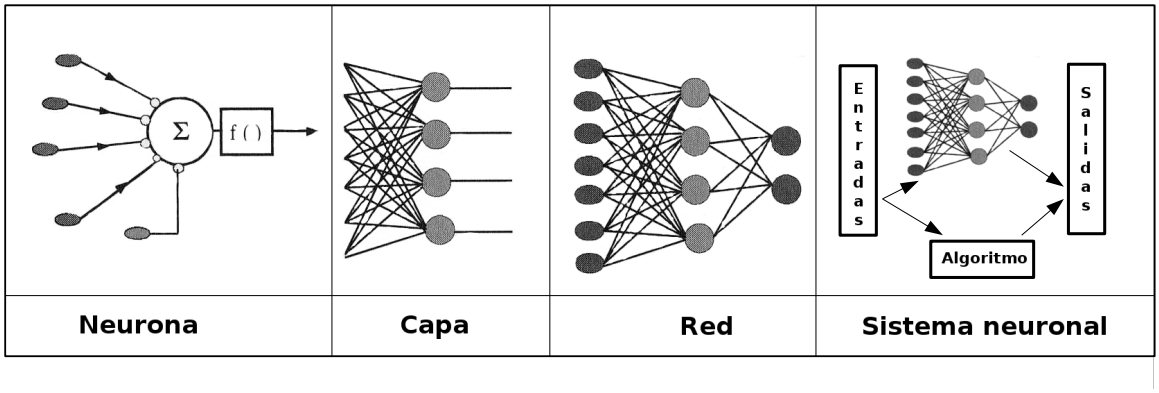


Figura 2.45: Sistema global de proceso. Imagen obtenida de [LIM08].

- **Conjunto de entrada X**: Conjunto de entradas x_j, con pesos sinápticos w_{ij} con j = 1, $\ldots, n$
- **Regla de propagación H**: Cada $i-$ésima neurona cuenta con una función h_i definida a partir de entradas y pesos sinápticos:

$$h_i(x_1, \ldots, x_n, w_{i1}, \ldots, w_{in}) \tag{2.12}$$

La regla de propagación más simple combina linealmente entradas y pesos, teniendo a partir de 2.12:

$$h_i(x_1, \ldots, x_n, w_{i1}, \ldots, w_{in}) = \sum w_{ij}x_j \tag{2.13}$$

Sin embargo, lo habital es añadir un parámetro adicional θ_i, denominado umbral, al conjunto de pesos de la neurona. Es decir, añadiendo a 2.13:

$$h_i(x_1, \ldots, x_n, w_{i1}, \ldots, w_{in}) = \sum w_{ij}x_j - \theta_i \tag{2.14}$$

- **Función de activación Y**: Representa la salida de la neurona y su estado de activación de forma simultánea. Cada $i-$ésima neurona cuenta con una función y_i definida como:

$$y_i = f_i(h_i) = f_i(\sum w_{ij}x_j - \theta_i) \tag{2.15}$$

En este apartado podemos resaltar algunas funciones comunes de activación:

- *Neuronas todo-nada*: En este tipo de nueuronas, la función de activación es una función escalonada, teniendo:

$$y_i = \begin{cases} 1 & \text{si } \sum w_{ij}x_j \geq \theta_i \\ 0 & \text{si } \sum w_{ij}x_j < \theta_i \end{cases} \tag{2.16}$$

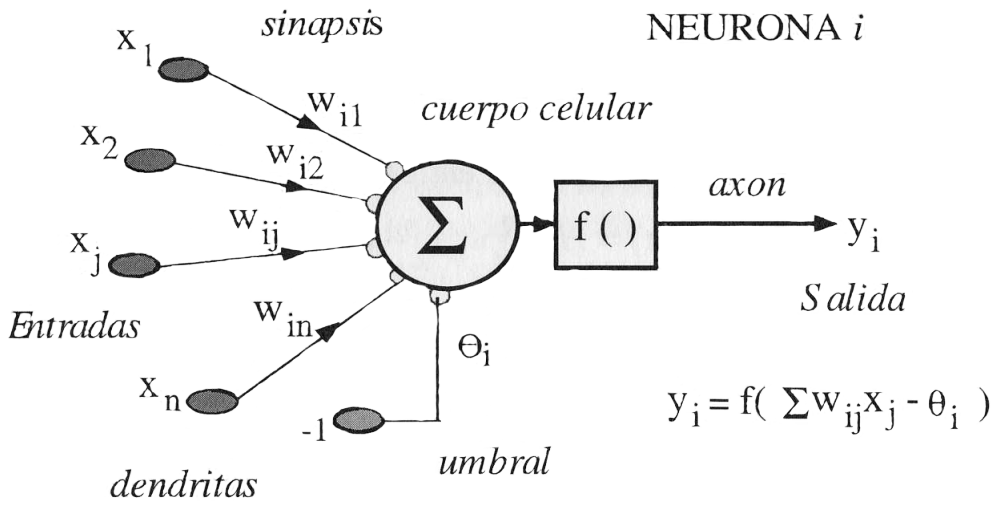


Figura 2.46: Estructura de la neurona artificial estándar. Imagen obtenida de [LIM08]

Es la función de activación usada en el modelo de neurona del perceptrón original.

- *Neurona continua sigmoidea*: Esta función de activación produce una salida continua. Puede establecer el intervalo de salida $y_i \in [0, 1]$ para la sigmoidea normal, e $y_i \in [-1, 1]$ para la sigmoidea simétrica.

$$y_i = \frac{1}{1 + e^{-(\sum w_{ij} x_j - \theta_i)}}, \text{ con } y_i \in [0, 1] \tag{2.17}$$

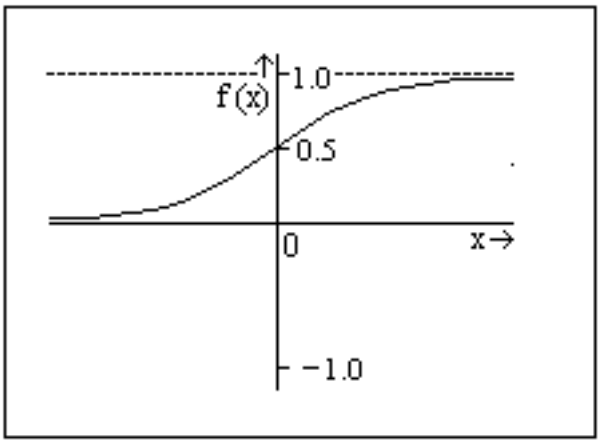


Figura 2.47: Gráfica de la función de activación sigmoidal. Fuente: `http://zerkpage.tripod.com/ann.htm`

- *Neurona de transferencia gaussiana*: Función de activación más adaptativa que las funciones sigmoidales al poder adaptar los centros y anchura de la función.

Mapeos que suelen requerir dos niveles ocultos en redes con fuciones sigmoidales pueden hacerse con un sólo nivel oculto con funciones gaussianas.

$$y_i = e^{-(\sum w_{ij} x_j - \theta_i)^2}, \text{ con } y_i \in [0, 1] \tag{2.18}$$

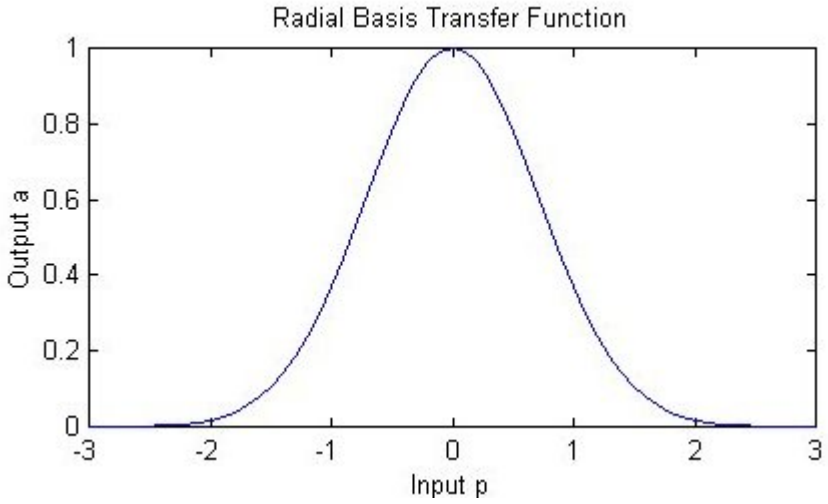


Figura 2.48: Gráfica de la función de activación gaussiana. Fuente: `http://www.dtreg.com/rbf.htm`

Arquitecturas y modelos de redes neuronales

Arquitectura de la red neuronal: Se denomina arquitectura de la red neuronal a la topología, estructura o patrón de conexionado de la misma.

Los nodos se conectan por medio de sinapsis, estando el comportamiento de la red determinado por la estructura de conexiones sinápticas. Las neuronas o nodos se suelen agrupar en *capas*. El conjunto de una o varias capas constituye la red neuronal.

Tres tipos de capas en la red neuronal:

- Capa de entrada.
- Capa de salida.
- Capa oculta.

La *capa de entrada* está compuesta por neuronas 'exteriores', que reciben los datos o señales del entorno. La *capa de salida* la componen las neuronas que proporcionan al medio la respuesta del sistema global neuronal. Se denomina *capa oculta* a aquella capa que no tiene conexión directa con el entorno, ya sea sensor o efector.

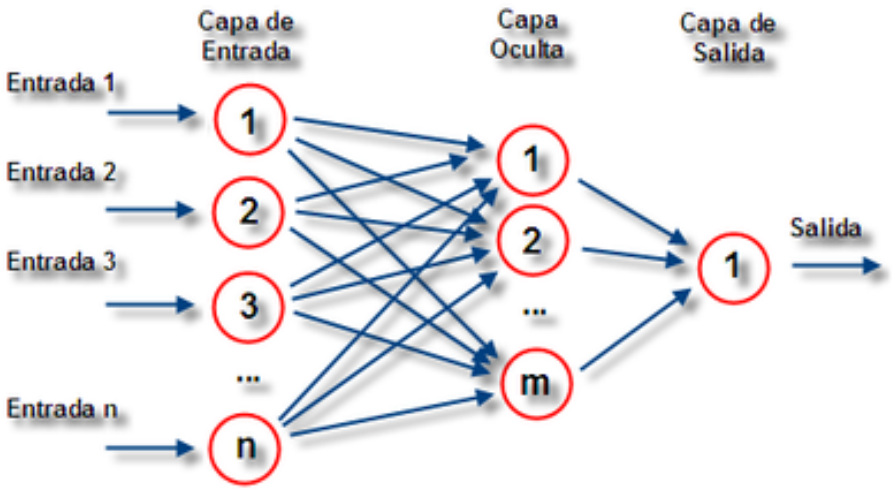


Figura 2.49: Tipos de capas en la red neuronal.

Atendiendo a esta estructura, hablaremos de *redes monocapa* o bien *redes multicapa*. En las redes monocapa sólo encontramos una capa de neuronas, mientas que las redes multicapa presentan varias.

Atendiendo al flujo de datos, caracterizamos las redes neuronales como *redes unidireccionales*, en las cuales la información circula en un único sentido, o como *redes realimentadas o recurrentes*, en las cuales la información puede circular en cualquier sentido, ya sea éste entrada-salida o salida-entrada.

Por último, en función de cómo la red actualiza sus estados, distinguimos red *dinámica síncrona* –todas las neuronas pertenecientes a una misma capa se actualizan en el mismo instante de tiempo- y red *dinámica asíncrona* –cada neurona actualiza su estado de forma independiente al resto-.

Perceptrón multicapa

El perceptrón multicapa es una Red Neuronal Artificial (RNA) formada por múltiples capas, con una o varias capas ocultas; ésto le permite resolver problemas que no son linealmente separables –ésta es la principal limitación del perceptrón simple, compuesto por una capa de entrada y una de salida, sin capas ocultas-. El perceptrón multicapa puede ser total o localmente conectado. En el primer caso, cada salida de una neurona de la capa "i" es entrada de todas las neuronas de la capa "i+1"; en el segundo, cada neurona de la capa "i" es entrada de una serie de neuronas (región) de la capa "i+1".

El perceptrón multicapa se prueba como un Aproximador Universal. Puede aproximar relaciones no lineales entre datos de entrada y salida; es una de las arquitecturas de RNA más empleadas en la resolución de problemas reales. Entre sus múltiples aplicaciones:

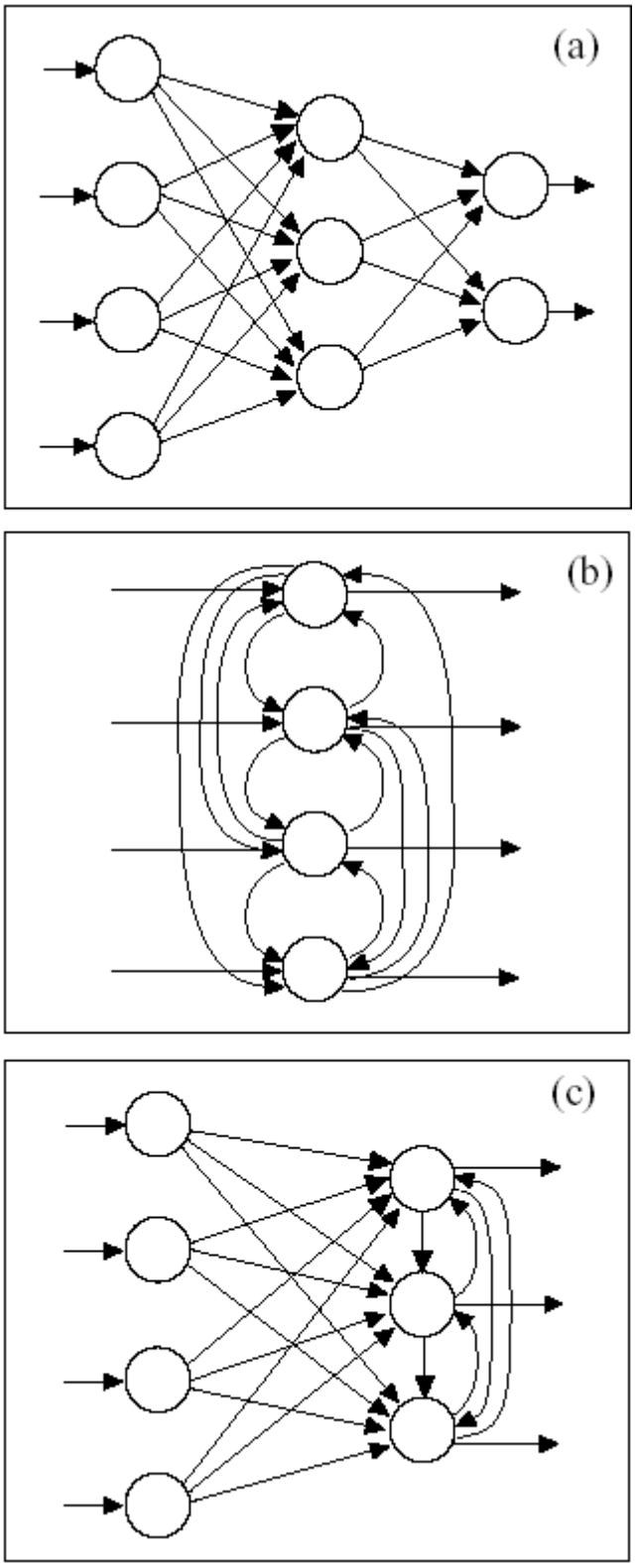


Figura 2.50: Diferentes arquitecturas de redes neuronales. *(a)* muestra una red multicapa unidireccional. *(b)* muestra una red monocapa recurrente. *(c)* muestra una red multicapa recurrente.

- Reconocimiento de voz.
- Reconocimiento de imágenes.
- Conducción de vehículos.
- Diagnósticos médicos.

La arquitectura del Perceptrón Multicapa se caracteriza por:

- *Capa de entrada* encargada exclusivamente de recibir señales de entrada y propagarlas a la capa siguiente.
- *Capa de salida* encargada de proporcionar la respuesta de la red ante un determinado patrón de entrada al exterior.
- *Capas ocultas* (una o varias) encargadas de realizar un procesamiento no lineal de los datos.
- Ser redes neuronales de *conectividad total*.

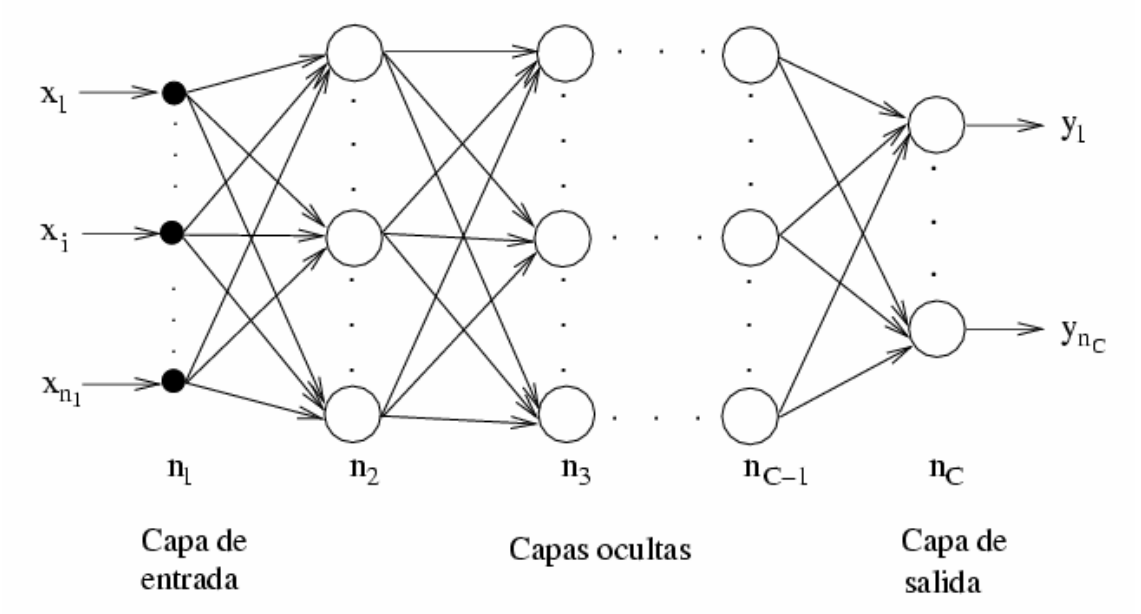


Figura 2.51: Arquitectura del perceptrón multicapa.

La propagación de los patrones de entrada se realiza hacia delante, desde la entrada hacia la salida. Cada neurona procesa la información recibida en sus entradas, produce una salida o activación, y la propaga hacia las neuronas de la siguiente capa.

Se define la *activación de las neuronas de entrada* como

$$a_i^1 = x_i \text{ para } i = 1, 2, \ldots, n_1 \tag{2.19}$$

donde $X = (x_1, x_2, \ldots, x_{n1})$ representa el vector de entrada.

La *activación de las neuronas de la capa oculta* se define

$$(a_i)^c = f(\sum {w_{ji}}^{c-1} {a_j}^{c-1} + {u_i}^c) \text{ para } i = 1, 2, \ldots, n_c \text{ y } c = 2, 3, \ldots, C-1 \qquad (2.20)$$

donde ${a_j}^{c-1}$ son las activaciones de la capa j-1.

Entre las funciones de activación más comunes en un Perceptrón Multicapa encontramos: la función Sigmoidal y la función Tangente Hiperbólica (Figura 2.52).

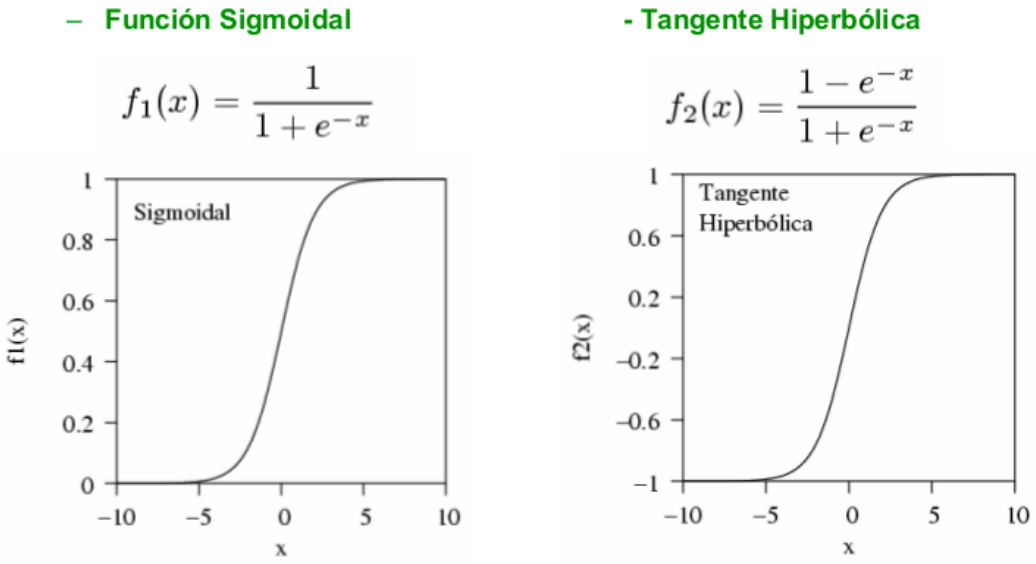


Figura 2.52: Funciones de activación comunes en un MLP.

Diseño de la arquitectura

En el momento de decidir emplear un el perceptrón multicapa para determinada labor del sistema, ha de diseñarse su arquitectura cuidadosamente, en concreto:

- **Función de activación**: Basarse en el recorrido deseado más que en la capacidad de la red.
- **Número de neuronas de entrada y de salida**: Definidas por las propias variables que definen el problema. Si no se conoce el número de variables relevantes del sistema es conveniente realizar previamente un análisis de las variables de entrada.
- **Número de capas y neuronas ocultas**: Elegido por el diseñador en base a su criterio, pues no hay un método óptimo para determinar el número de capas y neuronas ocultas. Generalmente es buena técnica elegir una o dos capas ocultas que contengan el doble de neuronas que el número de neuronas de entrada.

Entrenamiento: algoritmo *Backpropagation*

El algoritmo *Backpropagation* o de retropropagación es un algoritmo de aprendizaje supervisado en dos fases. En la primera fase, o fase de propagación hacia delante, se aplica un patrón de entrada a la red, se propaga desde la primera capa o capa de entrada hasta la capa de salida. Cada señal de salida (si hay varias) se compara con la salida deseada y se calcula el error. En la segunda fase, de propagación hacia detrás o retropropagación, el error calculado se propaga hacia detrás, hacia todas las neuronas que contribuyen a producir la salida. Reciben una fracción del error total (relativa a la contribución de la neurona o peso), y se ajustan a él para corregir la salida. Este proceso se repite continuamente hasta un número máximo de repeticiones o de haber conseguido un error inferior al deseado. Después del entrenamiento, cuando se presente un patrón arbitrario de entrada, aunque contenga ruido o esté incompleto, las neuronas de la capa oculta de la red responderán con una salida activa si la nueva entrada contiene un patrón que se asemeje a aquella característica que las neuronas individuales hayan aprendido a reconocer durante su entrenamiento.

El error cuadrático medio de las neuronas de la capa de salida se calcula como:

$$\varepsilon(n) = \frac{1}{2}\sum_{j\in C} e_j^2(n) \tag{2.21}$$

siendo C el conjunto de neuronas de la capa de salida y $e_j^2(n)$ el valor instantáneo del error cuadrático de la neurona de salida j, definido $e_j(n)$ como:

$$e_j(n) = d_j(n) - y_j(n) \tag{2.22}$$

siendo $d_j(n)$ el valor esperado e $y_j(n)$ la salida obtenida.

La corrección del peso $w_{ji}(n)$ de la neurona ji es proporcional al gradiente instantáneo

$$\frac{\delta\varepsilon(n)}{\delta w_{ji}(n)} \tag{2.23}$$

Por tanto el objetivo es buscar la corrección $\Delta w_{ji}(n)$ que se debe hacer. El listado 2.2 muestra el pseudocódigo de este proceso y el algoritmo en sí.

Aclarar en el listado anterior la variable $\delta v_j(n)$, que se define como:

$$v_j(n) = \sum_{i=0}^{p} w_{ji}(n) y_i(n) \tag{2.24}$$

donde p es el número de estímulos a la neurona j, w_{ji} el peso e y_i su salida.

funcion RETROPROGACION (*muestras de entrenamiento*, η, n_{in}, n_{out}, n_{hidden})

Cada muestra de entrenamiento es un par $(\vec{x},\vec{t})$*, donde* $\vec{x}$ *es el vector de entrada y* $\vec{t}$ *el vector de salida esperado.*

η *es el ratio de aprendizaje.* n_{in}*,* n_{out}*,* n_{hidden} *son el numero de neuronas de las capas de entrada, salida y ocultas respectivamente.*

La entrada de la neurona i a la neurona j se denota x_{ji}*, y el peso de la neurona i a la j* w_{ji}

- Crear red neuronal con n_{in} entradas, n_{out} salidas y n_{hidden} neuronas ocultas.
- Iniciar los pesos de la red de forma aleatoria.

Hasta obtener error deseado, **hacer:**

Por cada $(\vec{x},\vec{t})$ en *muestras de entrenamiento*, **hacer:**

Propagar la entrada a traves de la red

1. Introducir el vector de entrada $\vec{x}$ y calcular la salida $y_j(n)$

Propagar el error hacia detras en la red

2. Por cada neurona de salida calcular su error $\delta_j(n)$, siendo este $e_j(n)$

3. Por cada neurona oculta calcular su error proporcional $\delta_j(n)$, siendo:

$$\delta_j(n) = -\frac{\delta\varepsilon(n)}{\delta y_j(n)} * \frac{\delta y_j(n)}{\delta v_j(n)}$$

4. Actualizar cada peso w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

donde

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

Listado 2.2: Algoritmo de Retropropagación.

Capítulo 3

Objetivos

3.1 Visión general

La visión general de la propuesta del Proyecto Fin de Carrera es la creación una plataforma que permita abstraer al programador de los aspectos relativos a la gestión de dispositivos de Electroencefalografía, la comunicación con el computador y el tratamiento de los paquetes de datos, entre otros aspectos.

La plataforma deberá reconocer los dispositivos conectados y obtener los datos que envíen, independientemente del tipo de comunicación que utilicen. Deberá tratar cada dispositivo en un canal de comunicación diferente. Además, en función del dispositivo, podrá aplicar un tipo de tratamiento de onda u otro, pues cada dispositivo puede utilizar diferente formato de datos. Los datos recibidos se transformarán a una representación independiente del dispositivo, que utilizará la plataforma. Además, se entrenará el dispositivo con los datos obtenidos para adaptarse al usuario, ofreciendo mayor precisión en el control de aplicaciones.

La Figura 3.1 sintentiza, en un diagrama de bloques, este enfoque. Se describe a continuación:

La plataforma se encargará de la manipulación de los puertos a los que se conecten los receptores de señal de los dispositivos (USB, bluetooth...) y la comunicación que por ellos tenga lugar, a través de la Interfaz de Conexión del Dispositivo. Debido a que habrá varios dispositivos simultáneos, esta interfaz también se encargará de gestionar la comunicación de cada dispositivo, separando los paquetes recibidos en diferentes instancias de dispositivo. Después, estos paquetes pasarán a la capa de Decodificación, donde se analizarán los paquetes enviados por los puertos y, si éstos forman mensajes válidos, los convertirá a datos e información válidos para las capas superiores. Hasta este punto son capas dependientes del dispositivo.

Los datos anteriores pasan a las capas superiores, independientes del dispositivo en cuestión, donde son procesados antes de ser utilizados por otros componentes del dominio de la aplicación. Primero, la onda en bruto recibida se procesa en una capa de Tratamiento de Onda, donde se normaliza y separa en los diferentes tipos de onda que trate el dispositivo. Los datos de las ondas también serán utilizados por una capa de Entrenamiento y Patrones, utili-

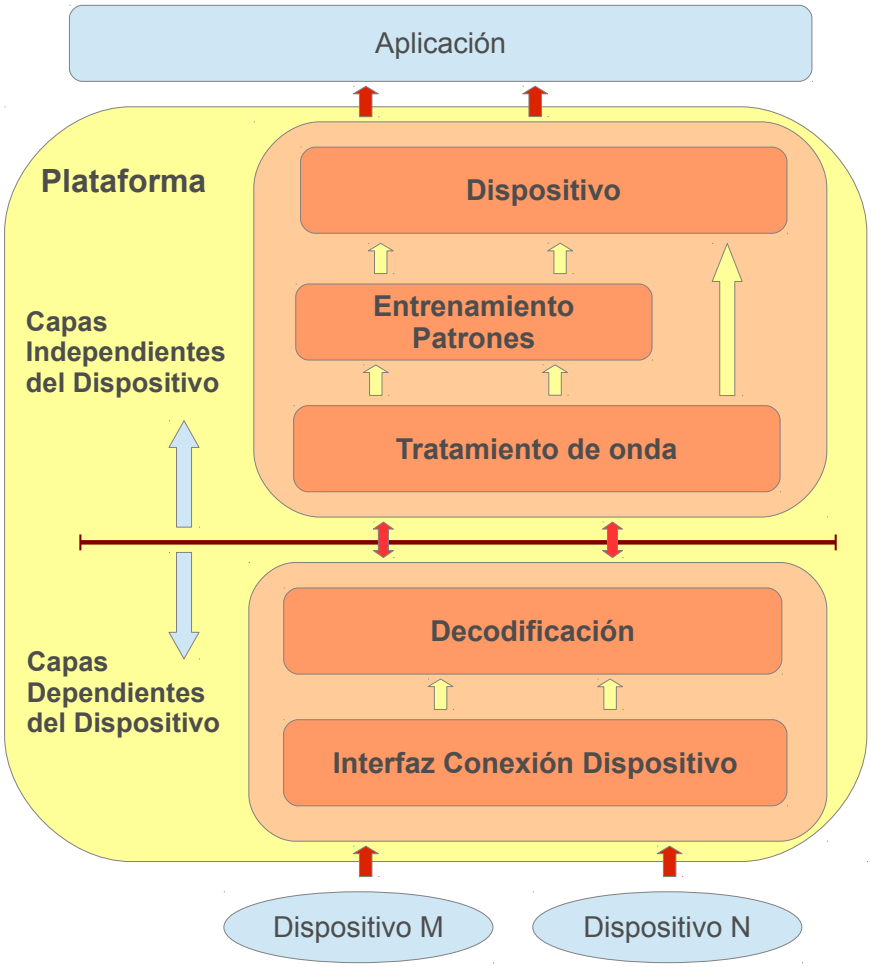


Figura 3.1: Visión general del sistema.

zada para "personalizar" el dispositivo. La capa de más alto nivel será la capa de Dispositivo, que será la utilizada para el control de la aplicación final.

3.2 Objetivos principales

El objetivo principal del presente Proyecto Fin de Carrera es el desarrollo de una plataforma que permita al programador desarrollar aplicaciones NUI con dispositivos de Electroencefalografía. Como aplicación concreta de la plataforma, se propone una aplicación similar a BrainBall[1], en la cuál dos usuarios compiten por llevar una pelotita al área contraria sólo mediante interacción mental.

Como se comentaba en la sección anterior, en última instancia se obtienen datos válidos para el dominio a partir de los mensajes enviados por los dispositivos.

[1] `http://www.youtube.com/watch?v=oBeGv_x4Tbs`

La información que cada instancia de dispositivo controlará será:

- Onda de entrada del dispositivo.
- Onda normalizada, separando los tipos de señal en función de la frecuencia.
- Valores de onda media y larga de cada tipo de onda tratada.

A partir del objetivo principal descrito anteriormente, se describen los siguientes subobjetivos funcionales que deben ser alcanzados en el Proyecto Fin de Carrera:

- El tratamiento a bajo nivel de la comunicación de los dispositivos con el computador. Será capaz de manipular los puertos y comunicarse con los dispositivos, separando además las señales de cada dispositivo.
- El tratamiento de la señal y normalización de la misma. La plataforma se encargará de limpiar y adaptar la señal que proporcionan los dispositivos EEG.
- El desarrollo de una interfaz para la obtención de los valores del dispositivo y ejecución de comandos. La plataforma abstraerá la etapa de obtención de valores de cada dispositivo conectado y permitirá ejecutar comandos para el cambio de estado interno.
- El desarrollo de una interfaz de comunicación con sistemas externos, como Arduino, y el control de los mismos.
- Ofrecer simultaneidad de dispositivos conectados. La plataforma será diseñada para permitir la conexión de varios dispositivos EEG, de diferente fabricante.
- Demostrar su uso real mediante una aplicación de prueba. Este demostrador tendrá, a su vez, los siguientes requisitos funcionales:
  - Deberá permitir el uso de 2 cascos EEG simultáneos.
  - Será una aplicación tipo BrainBall.
  - Mostrará el estado de cada dispositivo en una aplicación 3D.
  - Según el estado del juego, se desplazará físicamente una bola mediante un controlador Arduino y un circuito electrónico que se diseñará para ese fin.

Capítulo 4

Método de trabajo

En este capítulo se describe la metodología de desarrollo aplicada, sus ventajas y motivo de elección. También se listan y describen todas las herramientas utilizadas en el desarrollo, ya sean hardware o software.

4.1 Metodología de trabajo

Como metodología de desarrollo se ha optado por una aproximación a Scrum, siguiendo un modelo iterativo e incremental. Como aproximación de Scrum se han utilizado las reuniones cada una o dos semanas (serían los denominados *sprints*) con el director del proyecto. En estas reuniones se analizaban los hitos conseguidos y se proponían los siguientes en caso satisfactorio. También servían para controlar la planificación del proyecto y estudiar posibles módulos nuevos, que darían lugar a nuevos incrementos y mayor tiempo de desarrollo.

Figura 4.1: Proceso iterativo e incremental.

En cuanto a su naturaleza de modelo iterativo e incremental, esquematizado en la figura 4.1, se divide el desarrollo en ciclos o iteraciones en tiempos relativamente pequeños. En cada iteración, se completan todas las fases de desarrollo sobre cada módulo (requisitos,

análisis, diseño, implementación y pruebas) y se genera un prototipo o una versión completa del software mejorada respecto a la anterior, suponiendo incrementos en la funcionalidad del producto. El análisis de cada iteración se basa en la retroalimentación del usuario (en este caso el director) en la iteración anterior. De este modo, es posible definir nuevos requisitos con el director del proyecto, que serán desarrollados en una nueva iteración incremental en el sistema.

4.2 Herramientas de desarrollo

En esta sección se listan y detallan los recursos software y hardware empleados en la construcción de la plataforma. Además de una breve explicación del recurso, se enuncia la versión utilizada y sobre qué plataformas opera.

4.2.1 Lenguajes

Los lenguajes de programación y especificación utilizados son:

- **C++**: Es el lenguaje empleado para codificar el núcleo y la mayor parte del proyecto. También empleado en la parte de gráficos en OGRE3D. No se ha empleado el último standard C++11.
- **Arduino**[1]: Es el lenguaje de programación de los microcontroladores homónimos. Está basado en Wiring y permite la programación de rutinas para el control de sensores y actuadores conectados al micro.

4.2.2 Hardware

La mayor parte del desarrollo, casi la totalidad, se ha llevado a cabo en un computador de sobremesa con procesador Intel Core I7 930 con 8 núcleos a 2.80 GHz, 6GB de RAM y gráfica ATI Radeon HD 5850. En determinados puntos del desarrollo, se han implementado ciertas partes o realizado pruebas en un computador portatil Samsung NP-RC510 con procesador Intel Core I5 M480 con 4 núcleos a 2.67 Ghz, 4GB de RAM y gráfica NVIDIA 320M con Optimus.

Como dispositivos de Electroencefalografía se han utilizado dos dispositivos MindWave EEG propiedad de NeuroSky. Estos dispositivos son monocanal, trabajando a una frecuencia de 2.420 - 2.471 GHz RF, por medio de batería. Permiten la captación de onda cerebral en bruto, valores inmediatos de meditación y atención, y detección de guiños.

En el demostrador se han utilizado diversos recursos hardware. Por un lado, se ha utilizado un microcontrolador Arduino Nano v3, con chip ATMega 328P. Proporciona una serie de entradas y salidas analógicas y digitales, y voltaje de salida (5V) y tierra.

[1] El lenguaje de programación tiene el mismo nombre que el microcontrolador.

Por otro lado, se ha implementado un circuito integrado, conectado con el arduino, que se encarga de mover el demostrador BrainBall. Se ha empleado en el circuito:

- Batería de 12V
- Dos relés de 12V de dos canales
- Dos relés de 6V de un canal
- Dos transistores BC182
- Resistencias de varias medidas resistoras
- Un circuito integrado CNY70 de luz infrarroja
- Dos pulsadores en modo pull-up

4.2.3 Software

A continuación se listan detalladamente las bibliotecas externas y software de terceros empleados en en la realización de este Proyecto Fin de Carrera:

Sistema operativo

- **Ubuntu**: El sistema operativo principal de desarrollo es Ubuntu Linux, versión 12.04[2]
- **Fedora**: Se ha empleado también un segundo sistema operativo RHEL, a modo de prueba de portabilidad y rendimiento, en concreto Fedora 17[3].

Software de desarrollo

- **Emacs**: GNU Emacs es un editor de texto –y más- extensible y configurable, cuyo núcleo es un interprete de Emacs Lisp, un 'dialecto' del lenguaje de programación Lisp con extensiones para la edición de texto [Sta07]. GNU Emacs es el principal entorno de desarrollo y editor del proyecto; todo el código, a excepción del código Arduino, ha sido escrito con Emacs. Versión 22.3.
- **Arduino IDE**: Es un entorno de desarrollo propio de Arduino para desarrollar programas bajo su lenguaje homónimo, Arduino. Se ha empleado en la programación de las rutinas del microcontrolador. Versión 1.0.
- **GCC**: GCC[4] es un conjunto de compiladores creados por el proyecto GNU. Es software libre bajo licencia GPL. El compilador concreto utilizado es GCC-G++, compilador de la suite GCC para el lenguaje de programación C++. Versión 4.6.3.

[2]Precise Pangolin `http://releases.ubuntu.com/precise/`
[3]`http://fedoraproject.org/get-fedora`
[4]GNU C Compiler

- **GDB**: GDB[5] es el depurador estándar para el compilador GNU. Es un depurador portable y funciona para varios lenguajes de programación como C, C++ y Fortran. Se ha empleado para realizar toda la depuración de código en el PFC. Versión 7.4.
- **Make**: Es una herramienta de generación o automatización de código, muy usada en los sistemas operativos tipo Unix/Linux. Se ha empleado para la compilación de código y generación de ejecutable del proyecto de forma automática. Versión 3.81.
- **Perf**: Linux Perf es una herramienta de análisis de rendimiento en Linux, disponible a partir del kernel 2.6.31[6]. Se ha empleado para medir el rendimiento de los módulos del programa y detectar posibles cuellos de botella. Versión 3.2.0.

Documentación y gráficos

- **GIMP**: Herramienta de manipulación de gráficos, utilizada para la creación de overlays para OGRE3D y gráficas de módulos. Versión 2.6.12.
- **CEGUI Unified Editor**: Editor de *layouts*, *imagesets* y *schemes* para Crazy Eddie's GUI System (CEGUI). Incluye gestión de proyectos. Empleado para la creación y parametrización de Widgets gráficos. Versión 7.x.
- **Blender**: Blender es una suite 3D de modelado, animación, renderizado y post-produccion. Su interfaz está implementada completamente en OpenGl, contando con 'bindings' de Python para scripting. Se ha empleado en la creación de los modelos 3D y texturizado de los mismos, empleados en el demostrador. Versión 2.63.
- **LibreOffice Draw**: Potente herramienta de dibujado vectorial perteneciente a la suite ofimática LibreOffice. Utilizada para la generación de diagramas para la documentación. Versión 3.5.4.
- **LibreOffice Calc**: Potente hoja de cálculo perteneciente a la suite ofimática LibreOffice. Utilizada para el análisis de resultados de la red neuronal empleada, datos estadísticos, generación de gráficas, comparación de valores... Versión 3.5.4.
- **Latex**: Es un sistema de composición de textos, orientado especialmente a la creación de libros, documentos científicos y técnicos que contengan fórmulas matemáticas. Elegido para la generación de la documentación mediante la distribución Texlive. Versión 2009-15.
- **Texmaker**: Editor de LaTeXempleado para la generación de la documentación.

Bibliotecas

- **boost-system**: Biblioteca que proporciona, de forma simple y ligera, tratamiento y encapsulación de errores del sistema, y una forma abstracta y portable de encapsular

[5] GNU Debugger
[6] `http://kernelnewbies.org/Linux_2_6_31`

condiciones de error. Se ha empleado para el tratamiento de errores de la plataforma en puertos serie. Version 1.46.

- **boost-thread**: Biblioteca con herramientas de gestión de paralelización y concurrencia, escritas en C++, que permiten ejecutar múltiples hilos de ejecución asíncronos e independientes. Se ha empleado para la creación de hilos independientes encargados de leer de los distintos puertos serie (dos para MindWave y uno para Arduino). Version 1.46.
- **CEGUI**: Crazy Eddie's System es una biblioteca libre que proporciona renderizado de ventanas y widgets para APIs gráficas y motores de renderizado que no proporcionen esa funcionalidad (como ocurre con Ogre3D). Está escrita en C++, Orientación a Objetos (OO) y orientada a desarrolladores de juegos. Se ha empleado en la creación de Widgets de la plataforma que representan los valores y estados posibles de los dispositivos. Versión 7.x.
- **OGRE3D**: Motor de renderizado flexible y orientado a la escena, escrito en C++, y designado para ser fácil e intuitivo para programadores gráficos. Abstrae las capas inferiores de representación, ya utilice las bibliotecas de OpenGL o Direct3D por debajo. Se ha empleado Ogre3D para producir el juego-demostrador de la plataforma. Versión 1.7.4.
- **FANN**: Biblioteca de código abierto de redes neuronales. Implementa redes neuronales multicapa totalmente conectadas y parcialmente conectadas. Está implementada en lenguaje C, con un wrapper C++, y en punto fijo y punto flotante. Dispone de 'bindings' para más de 15 lenguajes y un framework para manejar los datos de entrenamiento. Versión 2.2.0.

Capítulo 5

Arquitectura

En este capítulo se analiza y detalla la arquitectura de la plataforma, estructurada modularmente. Se sigue un enfoque *Top-Down*, comenzando por una descripción funcional del módulo: qué función desempeña, cuáles son sus relaciones con otros módulos (si las tiene) y qué artefacto produce. Después, se irá concretando más la descripción y refinándola hasta llegar a los aspectos complejos y de implementación, qué decisiones de diseño de clases se tomaron y por qué, qué problemas surgieron.

La plataforma, descrita en la figura 5.1 en la página siguiente, se compone de los siguientes módulos:

- **Módulo de comunicaciones:** gestiona los puertos del sistema para la comunicación con los dispositivos EEG.

- **Módulo de tratamiento de ondas:** obtiene los datos en bruto provenientes del dispositivo y los trata en función de parámetros de onda para obtener datos de instancia de dispositivo.

- **Módulo de dispositivo:** gestiona toda la información y estado de los dispositivos EEG. Esta información puede ser utilizada por la aplicación.

- **Módulo de trazas:** proporciona soporte para depuración y generación de trazas para estudio de la onda o entrenamiento de algoritmos en modo "offline"; es decir, sin necesidad de conectar un dispositivo posteriormente.

- **Módulo de representación gráfica:** ofrece una representación 3D de todos los datos del dispositivo, a partir de widgets, un modo debug y representación 3D del sistema físico del demostrador.

- **Módulo de control del microcontrolador:** proporciona métodos para controlar el microcontrolador encargado de controlar la parte hardware.

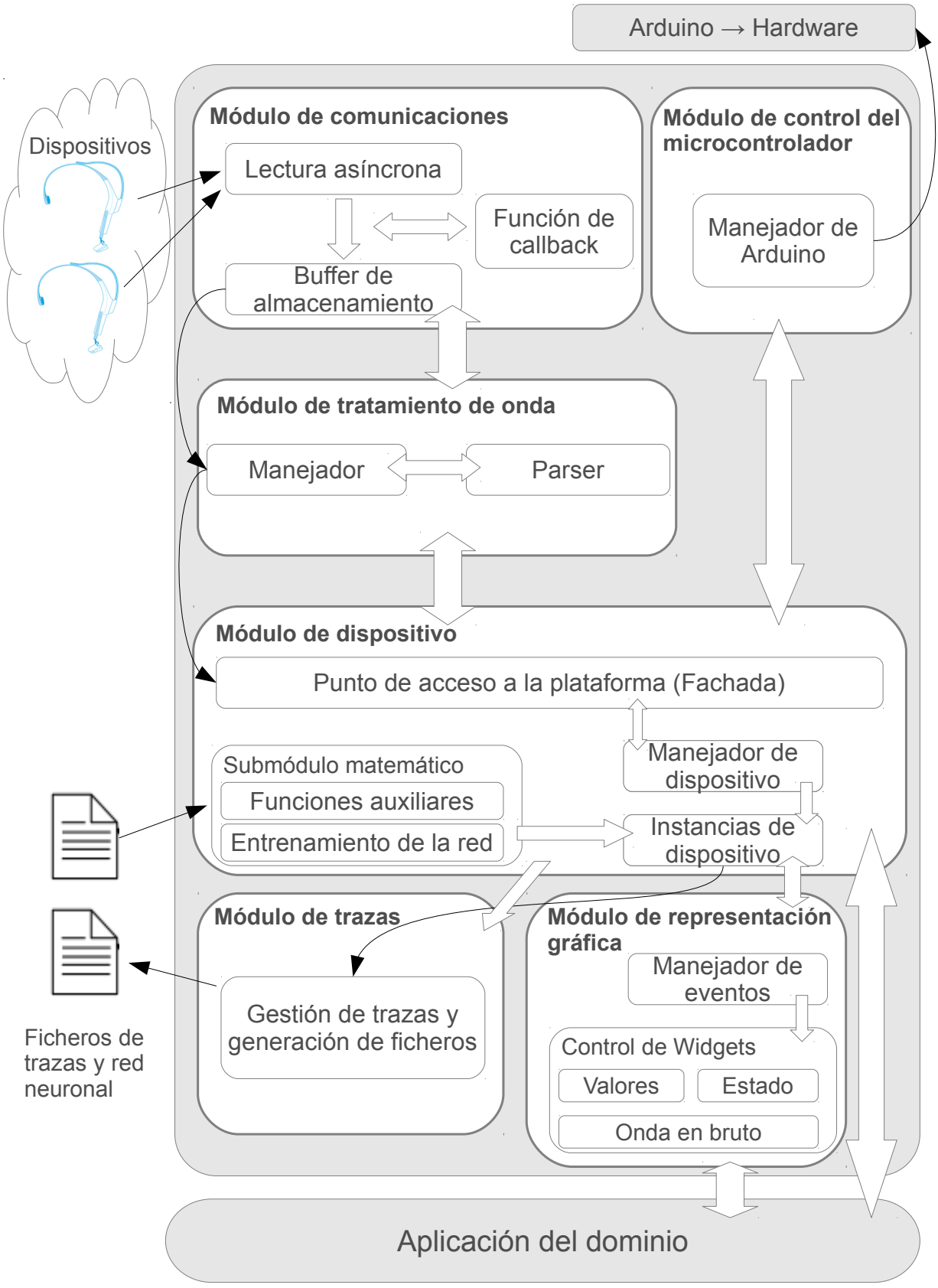


Figura 5.1: Arquitectura modular del sistema.

5.1 Descripción general

La plataforma de desarrollo objeto de este Proyecto Fin de Carrera se desarrolló siguiendo una arquitectura modular. Los módulos que la componen serán descritos detalladamente en las secciones siguientes: Módulo de comunicaciones, sección 5.2; módulo de tratamiento de ondas, sección 5.3, módulo de dispositivo, sección 5.4, módulo de trazas, sección 5.5, módulo de representación gráfica, sección 5.6, módulo de control del microcontrolador, sección 5.7. Para finalizar se incluye una sección adicional sobre la integración con OGRE 3D y la estructura de *estados de juego* empleada. En este punto se dará una descripción general del conjunto del sistema.

El sistema proporciona un «*framework*» que facilita el desarrollo de aplicaciones con interfaces naturales de Electroencefalografía, abstrayendo al programador de los detalles concretos y de bajo nivel de los dispositivos. Si lo prefiere, también puede extender la funcionalidad de los dispositivos o añadir nuevos, extendiendo la interfaz genérica de dispositivo. Esta interfaz proporciona el esqueleto de la instancia de dispositivo y la funcionalidad mínima que debe implementar para ser funcional. También es independiente del dispositivo concreto, pues las diferencias posibles ya habrán sido tratadas en las capas inferiores de comunicaciones y de tratamiento de onda. De esta forma, mientras se mantengan las interfaces y se respeten los formatos de datos entre capas, el sistema podrá trabajar con cualquier dispositivo EEG comercial.

El módulo de comunicaciones también implementa una interfaz genérica que puede extender el desarrollador en otras interfaces concretas si desea o necesita cambiar el sistema de comunicación de los dispositivos, por ejemplo, de USB a Bluetooth. Las comunicaciones de los puertos son tratadas en hilos de ejecución distintos entre sí y distintos al de la aplicación principal para evitar que sean cuello de botella o produzcan bloqueos, consiguiendo un buen rendimiento. El desarrollador podrá utilizar los hilos por defecto, implementados con Boost Thread, o implementar otros si lo desea. Aunque se da siempre opción a extender el sistema, la configuración por defecto contiene toda la funcionalidad; el desarrollador debería preocuparse solamente de programar la aplicación y utilizar los datos de dispositivo.

Proporciona también funcionalidad de entrenamiento del dispositivo. El entrenamiento requiere un elevado número de muestras y mucho tiempo para aumentar su precisión. Para reducir el esfuerzo de entrenamiento se proporciona funcionalidad para capturar y almacenar trazas en disco, siguiendo un formato específico, para que pueda ser tratada posteriormente sin necesidad de tener conectado el dispositivo, e incluso se puedan probar varios algoritmos de forma simultánea para una misma traza. La plataforma incluye además funcionalidad matemática para el entrenamiento y el tratamiento de onda.

Se cuenta además con funcionalidad de representación 3D de los datos de instancia del dispositivo por medio de Widgets. Estos Widgets se han implementado utilizando Crazy

Eddie's GUI System, por lo que son compatibles con los motores gráficos que éste soporte; a saber: Direct3D, OpenGL, OpenGL 3, OGRE 3D e Irrlicht. Por defecto está preparado para trabajar con OGRE 3D.

Dado su carácter de plataforma más que de framework (la plataforma incluye hardware), proporciona funcionalidad para el uso de microcontroladores Arduino. Estos microcontroladores pueden programarse para el control de cualquier circuito, manipulando la comunicación por USB. Basta implementar una serie de comandos y enviarlos desde la plataforma; también pueden recibirse comandos desde el microcontrolador.

A continuación se detallan los módulos mencionados en la figura 5.1.

5.2 Módulo de comunicaciones

El módulo de comunicaciones es el encargado de administrar los recursos hardware mediante los cuales el dispositivo de EEG se comunica con el sistema. Todo dispositivo de EEG no utiliza los mismos medios para enviar la actividad cerebral captada al computador. Unos pueden comunicarse por USB, otros lo harán por bluetooth...

En cualquier caso, es necesario controlar ese puerto de comunicaciones y administrar los datos que por él se reciban/envíen (el envío de datos suelen ser siempre comandos de arranque/parada del dispositivo, aunque hay otros comandos especiales que se discutirán más adelante). Ha de asegurarse la apertura del puerto, la recepción de los datos, el envío de comandos y, para tratar los casos de diferencia entre frecuencia de recepción de datos y frecuencia de uso de esos datos, algún sistema de almacenamiento intermedio. Todas estas funciones son las que realiza este módulo.

Pensando en estos aspectos, se han diseñado dos submódulos:

1. Un primer módulo encargado de leer de forma asíncrona en el puerto.
2. Un segundo módulo encargado de recoger la lectura asíncrona del primero y almacenarla en un buffer intermedio entre el módulo de comunicaciones y el módulo de tratamiento de ondas.

Así, teniendo del puerto una entrada de datos irregular tanto en tamaño como en tiempo, se puede dar una salida de datos más uniforme. Gráficamente, la descripción funcional del módulo es la mostrada en la figura 5.2.

Se tiene una entrada irregular de bytes, debido a que el dispositivo envía diferentes tipos de paquetes a diferentes frecuencias cada uno. Por mencionar un ejemplo aclarativo, el dispositivo MindWave envía paquetes de onda en bruto 512 veces por segundo y paquetes de intensidad de la señal 1 vez cada segundo, conteniendo los primeros 2 bytes de información y los segundos 1 byte. Esos bytes se leerán de forma asíncrona, sin bloquear el puerto hasta que haya un número determinado de bytes y después se pasarán al submódulo de almacena-

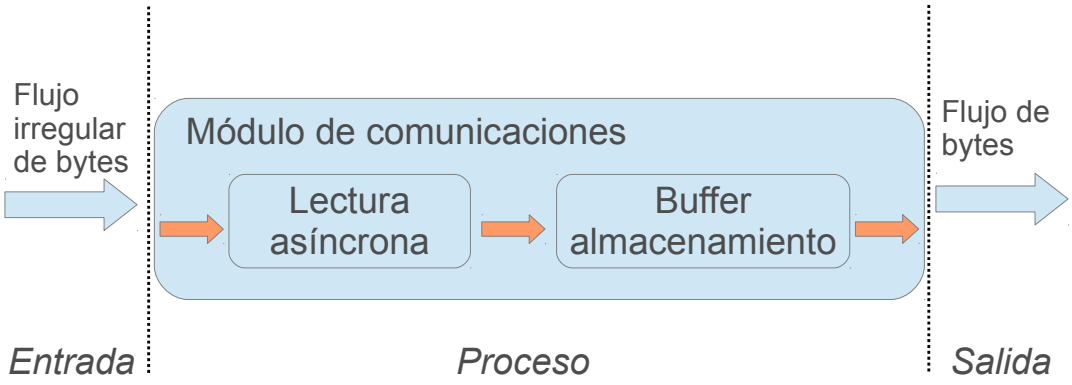


Figura 5.2: Descripción funcional del módulo de comunicaciones.

miento. Los datos del buffer son los que se proporcionan al siguiente módulo. Se menciona que la salida del módulo es regular debido a que el submódulo de almacenamiento informa al módulo de tratamiento cuando hay cierta cantidad de datos, de forma que pueda leer siempre esa misma cantidad; la frecuencia a la que lo hace es casi constante (puede haber pequeñas variaciones).

En cuanto a las dependencias con otros módulos, al ser el módulo de más bajo nivel, de la capa física, no tiene dependencias con otros módulos. En la entrada tampoco interactúa con otro módulo. Es a la salida donde presenta interacción con el módulo de tratamiento de ondas; se encarga de proporcionarle al segundo un flujo de bytes a tratar para convertir en información de la actividad cerebral captada por el dispositivo EEG, y que envía por el puerto codificada.

5.2.1 Problemática

La funcionalidad que debe tener el módulo puede parecer sencilla en cuanto a cometido, pero su diseño entraña mayores problemas. Ésto se debe a que debe atender a un gran número de características para que el diseño sea eficiente desde el principio, robusto y extensible. Por otro lado, debe proporcionar facilidades al programador para extender su funcionalidad e incorporar nuevos mecanismos de comunicación si lo desease.

Desde el principio se ha tenido en cuenta que ha de ser un módulo extensible, que el desarrollador puede necesitar un medio de comunicación distinto al soportado (a tiempo de escribir este documento, puerto USB) y ésto no debe suponer un enorme problema. Simplemente debe poder extender la interfaz para incorporar su solución.

Más complejo es platear un diseño robusto a la par que eficiente; un módulo robusto generalmente incorpora mecanismos de control de errores que aumentan la latencia del mismo, y este módulo debe tener una latencia mínima, pues debe proporcionar datos a una frecuencia elevada en tiempo real y de varios dispositivos a la vez. Este punto ha supuesto el mayor de los problemas, pues en su primera versión ralentizaba el sistema en un 300 % aproximada-

mente al conectar varios dispositivos, haciendo un mismo valor no se actualizase hasta pasados 3-4 segundos, lo cual es inadmisible; ésto se comentará detalladamente en el siguiente apartado. El enfoque asíncrono multi-hilo eliminó este problema.

En términos de eficiencia también supuso en problema el submódulo de almacenamiento, pues como se comentará en el siguiente apartado y en la sección 5.3, no puede asignarse un tamaño cualquiera. El tamaño del buffer debe atender a la frecuencia a la que se rellena y a la frecuencia a la que se desean extraer los datos. Como se aprecio en los primeros prototipos del módulo, un tamaño muy pequeño hacía que las esperas fuesen demasiado largas y un tamaño grande hacia que se almacenasen más paquetes de los que se podían tratar, por lo que al desconectar el dispositivo aún había conversiones y actualización del estado del dispositivo remanentes.

5.2.2 Solución

En este punto se discute la solución propuesta a los problemas anteriores y su implementación. De los problemas anteriores se establece una serie de características deseables y medidas para resolverlos y obtener el resultado deseado.

El objetivo buscado es que el módulo sea capaz de administrar el puerto de comunicaciones y la información que por él fluya de una forma eficiente y a la vez robusta, que no sea cuello de botella bajo ningún escenario multidispositivo. Además, debe establecer un almacenamiento adecuado y óptimo para proporcionar al módulo de tratamiento de ondas una salida consistente y constante. Por encima de todo ello, la funcionalidad debe ser extensible fácilmente y ser una solución estándar para múltiples sistemas. Para ello, se establecen una serie de características o puntos de desarrollo:

- Se establece una interfaz genérica que debe implementar cada tipo de comunicación, de formas que todas puedan tratarse igual aún teniendo diferente implementación.
- Cada interfaz de comunicación concreta leerá de forma asíncrona en el puerto, de forma que no lo bloquee hasta leer un número de bytes determinado.
- Además cada interfaz de comunicación concreta se ejecutará en un hilo de ejecución diferente, de forma que no sobrecargue el hilo principal de trabajo.
- Se implementará un submódulo de almacenamiento que regule la salida de forma uniforme, con un tamaño de buffer óptimo.
- Implementará mecanismos de seguridad y recuperación de errores.

Así pues, se ha implementado una interfaz abstracta llamada `CommInterface`. Esta interfaz define las operaciones básicas que cualquier comunicación debe implementar para ser válida y extensible a la plataforma. Implementando la interfaz y mediante polimorfismo, el sistema establecerá la comunicación y recibirá los datos sin necesidad de conocer que sistema de E/S está utilizando. Las operaciones definidas en la interfaz se muestran en el listado 5.1.

```
1   class CommInterface {
2     public:
3       virtual bool isOpen() const = 0;
4       virtual void close() = 0;

6       virtual void doRead() = 0;
7       virtual void readEnd(const boost::system::error_code& error,
            size_t bytes_transferred) = 0;
8       virtual void doWrite() = 0;
9       virtual void writeEnd(const boost::system::error_code& error) =
            0;
10      virtual void doClose() = 0;

12      virtual void setErrorStatus(bool e) = 0;
13  };
```

Listado 5.1: Interfaz genérica de comunicaciones.

La interfaz obliga a tener operaciones para saber si la conexión está abierta y para poder cerrarla. Ha de hacerse un inciso en el hecho de que la operación de apertura no aparezca en esta *interface*. Ésto es debido a que cada sistema E/S y las bibliotecas empleadas para tratarlo requiere una parametrización diferente para la apertura de la comunicación, por lo que ha de ser propia de cada implementación. Las siguientes operaciones que debe completar cada implementación son las operaciones de lectura y escritura del/en el puerto. Hay dos tipos de lectura/escritura:

1. **doRead/doWrite**: Realizan una lectura/escritura de los datos que haya disponibles.
2. **readEnd/writeEnd**: Añaden los datos leídos/escritos en las operaciones doRead/doWrite en el buffer a través de una función de retrollamada (clase `CallbackAsyncSerial`). Al finalizar invocan otra lectura/escritura asíncrona (doRead/doWrite).

La última operación de la interfaz sirve para establecer un estado de error en caso de producirse algún conflicto o error. Se establece el estado de error a `true/false` y el error concreto se devuelve en el método que puede producirlo. El error ha sido implementado haciendo uso de la biblioteca `Boost::System`. Esta biblioteca proporciona un *wrapper* portable para encapsular las condiciones de error provenientes del sistema operativo o cualquier aplicación de bajo nivel o cualquier API. De esta forma, se puede tratar el error de una forma más abstracta o de más alto nivel y, además, hacerla portable.

Así, la clase encargada de la lectura asíncrona en los puertos USB, `AsyncSerial`, hereda de esta interfaz e implementa todas las operaciones especificadas. Además, para ser funcional, ha de añadirse la operación de apertura del puerto (puede ser también el constructor de la clase) junto a las demás. Ha de hacerse un inciso que se explicará detalladamente más adelante: `AsyncSerial` no realiza realmente estas funciones, sino su implementación privada de la clase `AsyncSerialImpl`.

Para la gestión de los puertos USB, se ha optado por utilizar la biblioteca *Boost*; en concreto, se utiliza `Boost::Asio` para la lectura asíncrona, `Boost::Thread` para la creación y gestión de hilos, y `Boost::Bind` para la asociación de métodos o funciones para ciertas operaciones internas de *Boost*. Los parámetros que `Boost::Asio` precisa para abrir un puerto USB son los siguientes:

- Nombre del dispositivo
- Frecuencia del dispositivo
- Paridad del puerto
- Tamaño del carácter (char)
- Control de flujo
- Bits de parada

Exceptuando los dos primeros argumentos, el resto tienen valores por defecto y se pueden omitir. En cuanto a los dos primeros, son obligatorios, y deben ser introducidos por el programador. La frecuencia o *baud rate* viene determinada por el número de unidades de señal por segundo, y sólo se podrá asignar un valor a cada dispositivo. El nombre de dispositivo es el identificador que el sistema operativo da al dispositivo en cuestión. En sistemas GNU/Linux, los nombres utilizados comienzan por '/dev/ttyXXXX', correspodiendo XXXX a un código único para cada uno de ellos; al puerto USB se les asignan identificadores '/dev/ttyUSB0', '/dev/ttyUSB1'...

Siguiendo los puntos de desarrollo que establecimos al principio, necesitamos implementar un enfoque multi-hilo para escenarios multidispositivo. Para ello, *Boost* proporciona su biblioteca `Boost::Thread`, que se encarga de la creación y gestión de hilos de forma transparente al desarrollador. También proporciona los mecanismos de exclusión mutua necesarios. Para crear un hilo en boost precisamos de una instancia u objeto de la clase `boost::thread` y un método *run* de una clase que herede de `Thread`; dado que tratamos con servicios de *Asio*, la función «bindeada» (utilizando `Boost::Bind`) es `asio::io_service::run`. En este punto tenemos un submódulo que se ejecuta en un hilo de ejecución diferente y que lee de forma asíncrona en el puerto. El siguiente paso es establecer el submódulo de almacenamiento.

Para la realización de este submódulo se ha planteado la solución OO mostrada en la figura

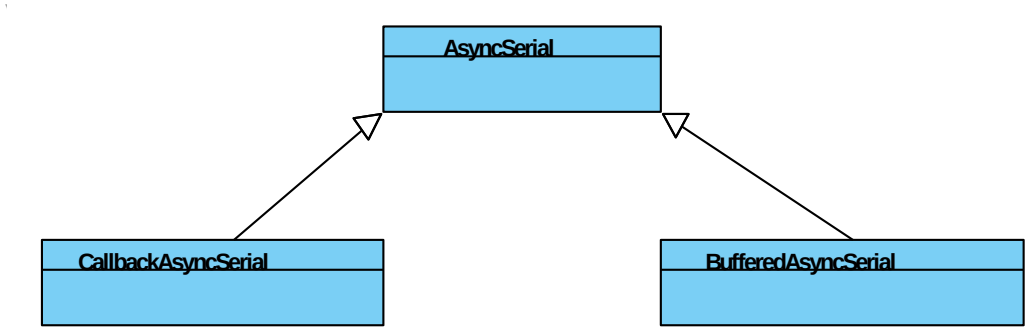


Figura 5.3: Diagrama de clases del submódulo de almacenamiento.

5.3. La clase `BufferedAsyncSerial` es la encargada de contener el almacén temporal de los bytes recibidos. El motivo de la herencia es que la clase `AsyncSerial` contiene una implementación privada, en la cual se hizo un inciso anteriormente (por lo cual realiza más bien la función de clase *abstracta*), basada en el patrón *Handle-Body*, que se discutirá en el punto 5.2.3 en la página 95. Mediante la herencia emplea la funcionalidad de AsyncSerial, y el acceso a la implementación privada con los métodos de la clase padre, y la complementa con el buffer de almacenamiento.

Sin embargo, en este punto las lecturas supondrían un cuello de botella, pues se estaría continuamente «preguntando» al puerto por nuevos datos o, en el argot informático, leyendo en el puerto mediante *polling*. Esta sobrecarga la elimina la clase `CallbackAsyncSerial`. De forma contraria a las demás, aquí se expone su implementación multi-hilo, lo que requerirá también de mecanismos de exclusión mutua. También hereda de `AsyncSerial` de la misma forma y los mismos motivos que `BufferedAsyncSerial`.

Su principal funcionalidad es la de establecer métodos de retrollamada que ejecuten ciertas acciones sólo cuando sea necesario. Así, continuamente se estará leyendo en el puerto en su hilo de ejecución independiente mientras se realizan otras operaciones y, sólo cuando haya datos disponibles y suficientes, se ejecutará la acción necesaria. La acción que queramos que se ejecute será una operación de inserción en el buffer. Los pasos necesarios a seguir son:

1. Implementar un método de inserción en el buffer que será llamado cuando haya datos suficientes.
2. Asociar ese método con el método de Callback en la apertura de comunicaciones.
3. Realizar lectura asíncrona.
4. Cuando se reciban datos, ejecutar la acción de retrollamada o callback.

Como última cuestión de la solución, todo este proceso ha de realizarse de forma segura, tanto siendo capaz de recuperarse de errores como asegurando la integridad de los datos asegurando la exclusión mutua de los objetos compartidos – el buffer de almacenamiento - en las operaciones de lectura/escritura. El caso más simple de corrupción de los datos sería

que mientras se estuviesen extrayendo datos desde el módulo de tratamiento de ondas se escribiesen nuevos datos, de forma que la secuencia de bytes quedase alterada con respecto al orden secuencial de llegada y, con ello, la alteración y corrupción de paquetes de onda.

Para solucionar este problema, recurrimos al mecanismo de «candados» que nos proporciona la biblioteca *Boost*: `boost::mutex`. En cada operación si situará un *mutex* al comienzo de forma que nos aseguraremos que cualquier otra operación no podrá modificar los objetos compartidos hasta que ésta finalice y se libere el candado. *Boost* proporciona además un mecanismo inteligente llamado *lock guard* que permite que la liberación del candado se haga de forma automática al finalizar el método o cuando deje de utilizarse el objeto; esto es especialmente util en escenarios donde el objeto compartido se emplea como retorno de una función.

El pseudocódigo de un *lock guard* es el siguiente:

```
1   funcion operacionMultiHilo (parametros...)
2       lock_guard<mutex> l(''nombre del mutex'')
3       ...
4       Operaciones sobre el objeto compartido
5       ...

7       fin de metodo o retorno del objeto
```

Listado 5.2: Pseudocódigo lock guard.

Para el control de excepciones se han empleado la biblioteca *Boost System* de la misma forma que se comentó más arriba en la definición de la interfaz genérica.

Como conclusión de la solución, se presentan un diagrama de secuencia y diagrama de clases finales del módulo, correspondientes a las figuras 5.4 y 5.5 respectivamente.

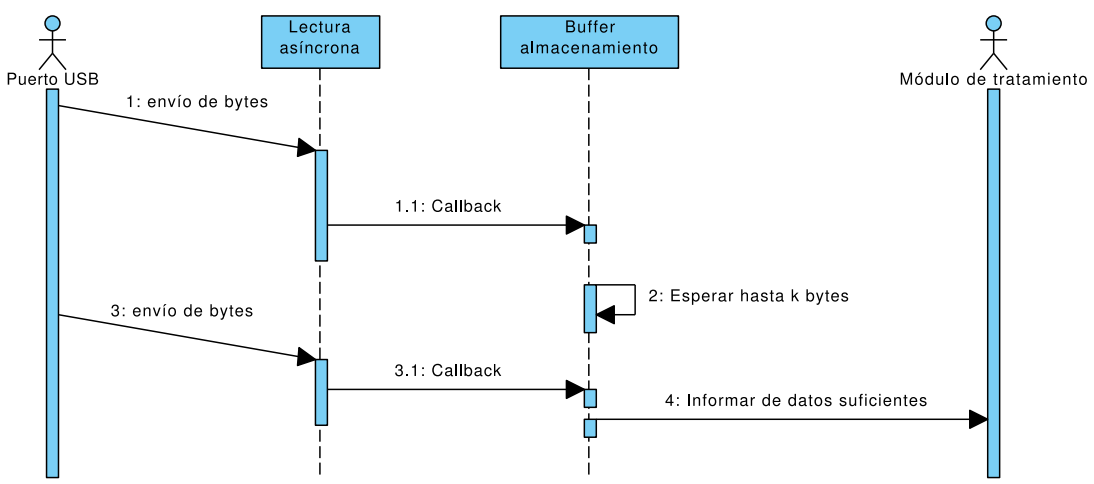


Figura 5.4: Diagrama de secuencia de comunicaciones.

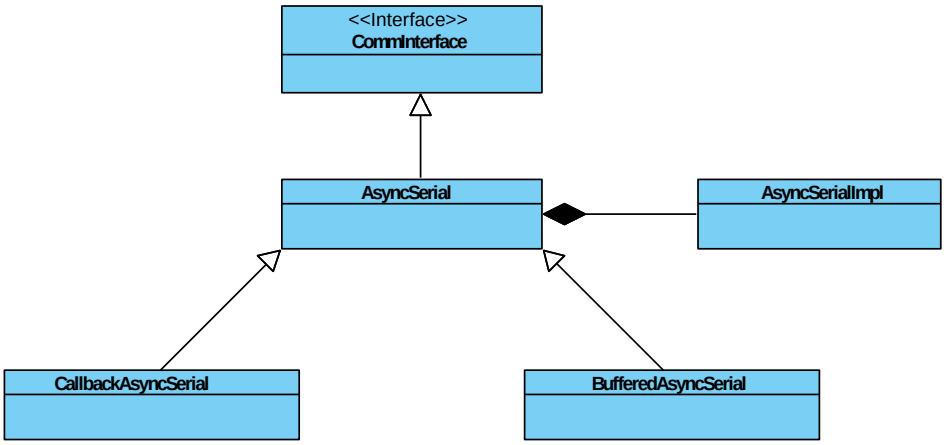


Figura 5.5: Diagrama de clases de comunicaciones.

5.2.3 Ventajas

La solución obtenida aporta grandes ventajas al sistema. Gracias a la utilización de una interfaz genérica que han de implementar todos los mecanismos de comunicación y al polimorfismo, podemos extender la funcionalidad del sistemas incorporando fácilmente nuevos mecanismos de comunicación, como recepción de los datos por medio de Bluetooth. Además, en este punto también es destacable la utilización del patrón Handle-Body como medio adicional de abstracción de la interfaz, de forma que la implementación concreta y los detalles de implementación quedan ocultos al resto del sistema.

El patrón Handle-Body es un patrón de diseño cuyo cometido es separar los detalles de implementación concretos de la interfaz, además de reducir el acoplamiento entre el uso de la clase y su implementación. La forma más sencilla de implementar el patrón consiste en separar en dos clases diferentes la interfaz pública de los detalles de implementación. El objeto público carece de cualquier detalle de implementación pero tiene un miembro privado que es un puntero al objeto de implementación.

Gracias al enfoque multi-hilo, también es posible atender peticiones de varios dispositivos de forma simultánea sin sufrir sobrecarga en el hilo principal y sin sufrir bloqueos. Con el añadido de las funciones de retrollamada además solo se ejecutan ciertas operaciones cuando es necesario y no de forma continua.

Por último, la implementación de un sistema de almacenamiento intermedio nos permite regular el flujo de datos irregular que se obtiene del dispositivo de EEG y mostrarlo de forma regular al módulo de tratamiento de ondas, estableciendo un tamaño de buffer óptimo que impida que se hagan esperas o que se almacene más de lo que se puede procesar.

5.3 Módulo de tratamiento de ondas

Este módulo tiene como función recibir el flujo de datos proveniente del módulo de comunicaciones, procesarlo y convertirlo en información válida para el dispositivo, debido a que el flujo de datos que envía el dispositivo no contiene información directamente utilizable. Cuando el sistema informe internamente de la llegada de suficientes datos, de lo que se encarga el módulo de comunicaciones, el *manejador* de dispositivo obtendrá un flujo de datos de la capa de comunicaciones, más concretamente, de un buffer de almacenamiento intermedio donde se va almacenando la entrada de datos asíncrona. Una vez obtenidos estos datos, habrá de convertirlos pasando byte a byte en el orden que llegaron. Sólamente una vez convertidos a paquetes válidos para el dispositivo se pasarán a la capa superior, al módulo de dispositivo. Además el mecanismo de conversión debe ser conocido únicamente por el módulo.

Para desempeñar esta labor, se han diseñado dos submódulos:

1. Un submódulo encargado de la gestión y control del flujo de datos y paquetes de dispositivo. Este submódulo es compartido con el módulo de dispositivo.
2. Un segundo submódulo encargado de recibir una serie de bytes o flujo de entrada y convertirlo en paquetes válidos de dispositivo si los hay en la serie recibida.

Con paquetes válidos de dispositivo nos referimos a una estructura de *código-valor/es* almacenadas en variables separadas de las que se pueden extraer directamente valores, pues el flujo de datos que se recibe del dispositivo son una serie de caracteres seguidos de los que nada puede extraerse directamente sin un proceso previo de conversión.

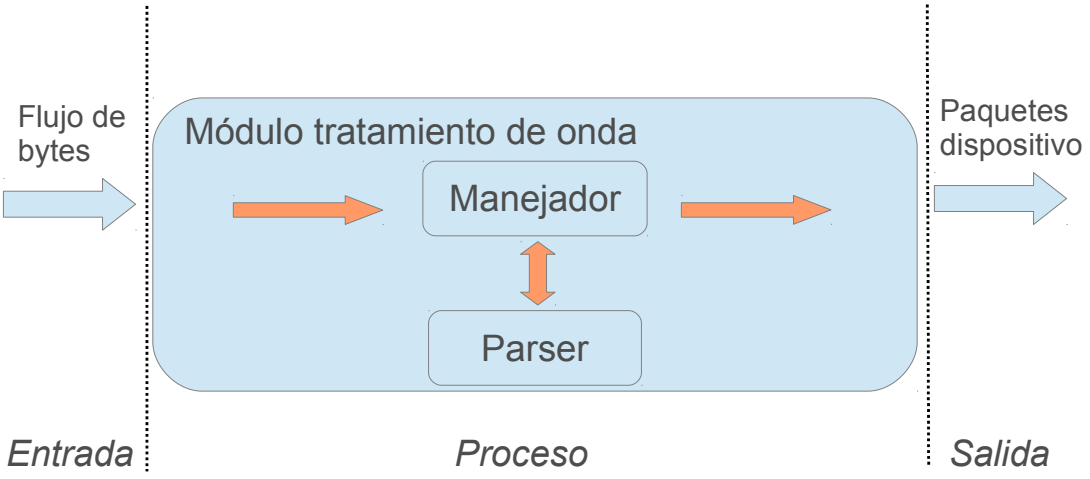


Figura 5.6: Descripción funcional del módulo de tratamiento de onda.

Por tanto, este módulo se relaciona por debajo con el módulo de comunicaciones, del que obtiene el flujo de bytes a tratar y, al producir la salida, con el módulo de dispositivo, al que

manda los paquetes de salida. Estos paquetes de salida, de forma más detalla que la expuesta brevemente más arriba, son unas estructuras caracterizadas por un código de tipo de dato, definido por un valor hexadecimal (se detalla más adelante), un tamaño de paquete, el valor o valores que se asocian a ese tipo, y un último valor perteneciente al ID del dispositivo destino.

5.3.1 Problemática

Este módulo planteó una de problemas que obligó a cambiar el diseño original, debido principalmente a convertirse en un cuello de botella en la primera versión, y a la mayor complejidad que tenía a la hora de manipular instancias de clase.

Principalmente se planteó en el diseño del módulo que las instancias estuviesen en un contenedor secuencial de tipo *vector*[1]. Con pocos dispositivos conectados no suponía un problema, pero cuando había varios de ellos, y se realizaba una operación de desconexión de alguno de los primeros, obligaba a reorganizar todo el vector y todos los identificadores, pues el ID ya no coincidia con la posición en el vector. Además esto suponía una sobrecarga adicional innecesaria. Para resolverlo, se sustituyó *vector* por *map*. El ID se asociaba a la instancia y no dependía de posición, inserción o borrado, a excepción, por supuesto de intentar guardar un ID existente. Además simplificaba la labor de programación.

Por otro lado, surgió un problema que no se había tenido en cuenta antes al no suponer problema con un solo dispositivo. Cuando había varios dispositivos conectados, a la hora de recuperar información de los *buffers* el sistema se ralentizaba (véase sección 5.2.1). Esto era debido a la diferencia de velocidad de llenado del buffer y la velocidad de adquisición de esos datos. Para ello, se estudió cual era la frecuencia de lectura del USB, la frecuencia a la que se obtenían y que cantidad de información se había leído en ese espacio de tiempo. Se estableció con todo ello un tamaño óptimo de buffer de 1024 bytes (con este buffer nos referimos al contenido en la clase `MindWaveHandler` que es el que completa con la información del buffer asíncrono del puerto). Además de seguir un enfoque basado en el patrón *Observer* para no acceder innecesariamente.

Otra decisión que se tomó en este módulo fue la de guardar la información referente a si un dispositivo enviaba datos o no en un *map* semejante al de las instancias de dispositivo, con el mismo ID de las mencionadas, en lugar de guardarlo en la propia instancia. El motivo por el que se hizo fue para evitar un apilamiento excesivo en la pila de llamadas del sistema, ya que cada vez que quiere obtenerse un dato de la instancia de dispositivo, ha de pasarse por el manejador primero, y luego al dispositivo; si esa información se encuentra antes, se realizan menos llamadas. Debido a que ese valor cambia relativamente poco, generalmente una vez al conectar el dispositivo y otra al apagar el sistema, no supone inconveniente añadir código

[1] *vector* es un template de la biblioteca estándar de C++ que gestiona de manera adecuada arrays de tamaño variable de un tipo de elementos.

adicional en el manejador para tratarlo. Esto supone una mejora de rendimiento directa en el sistema al evitar sobrecargar de más la pila de llamadas; también supone una mayor facilidad para el programador, ya que es más sencillo modificar el valor asociado a una clave en un *map* que el hecho de tener que buscar primero el dispositivo por su ID y luego invocar algún método.

5.3.2 Solución

El objetivo buscado es diseñar un módulo que realice la función que se necesita salvando los problemas que se han encontrado. A muy alto nivel, el módulo debe ser capaz de gestionar los datos de entrada, prepararlos y convertirlos en información útil, y posteriormente actualizar el estado de los dispositivos con ello. En términos de extensibilidad, la conversión de datos ha de realizarse en un submódulo a parte del de control para que sea posible modificar y cambiar el proceso de conversión sin necesidad de modificar el resto del sistema. Para ello, se establecen una serie de características o puntos de desarrollo:

- Se debe poder gestionar de forma sencilla y controlar los flujos de datos provenientes del módulo de comunicaciones.
- El proceso de conversión ha de realizarse en un submódulo diferente al de control y gestión para reducir el acoplamiento y mejorar la extensibilidad del módulo.
- Este módulo debe tener la responsabilidad de gestionar el proceso de conversión y de actualizar el estado de los dispositivos posteriormente.

Siguiendo el primer punto de desarrollo, se ha diseñado un submódulo de control, encargado de recibir y gestionar los flujos de datos de entrada. Un flujo de datos de entrada es una secuencia ordenada de bytes. Se codifica el byte mediante un *char* en C++; se utilizar char en lugar de unsigned char porque es necesario almacenar valores con signo. Siguiendo la especificación de datos de MindWave, los valores en bruto de onda recibidos son valores con signo de 16 bits, comprendidos en el rango de -32768 a 32767. Así, utilizaremos dos bytes en el envío y obtendremos el valor desplazando el primer byte y realizando una operación OR:

```
1  short raw = (Value[0]<<8) | Value[1];

3  raw = Value[0]*256 + Value[1];
4  if( raw >= 32768 ) raw = raw - 65536;
```

Listado 5.3: Codificación del flujo de datos.

Value[0] representa el primer byte o byte más significativo, mientras que Value[1] es el byte menos significativo. En la primera línea se produce el desplazamiento y la operacion

OR. El código relativo a la línea 3 (excluyente con la línea 1 aunque mostrado conjunto) se debe a que en ciertos sistemas no puede realizarse el desplazamiento de bits, por lo que se sustituye por una multiplición por 256 (8 bits, como el desplazamiento) y una suma. La línea 4 controla que en caso de obtener un valor superior al máximo de 32767, se ajuste al rango.

Sin embargo, este proceso no representa la totalidad de los casos. Primeramente, no todos los tipos de datos están formados por dos bytes; algunos de un solo byte y otros de más. Segundo, los bytes se reciben acorde a una estructura de paquete, con cabecera de control y cuerpo de datos donde van incrustados esos valores. La estructura de paquete, descrita en la figura 5.7, impone una cabecera de 3 bytes, un cuerpo del paquete de un máximo de 169 bytes, y una tercera parte de *checksum* o suma de comprobación de 4 bytes para saber si el paquete ha llegado correctamente o se ha corrompido.

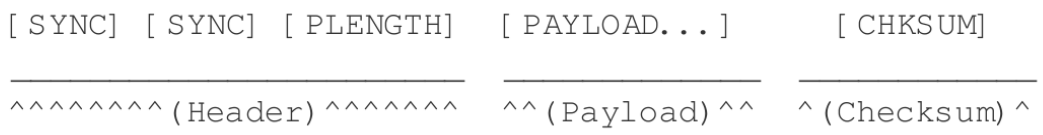


Figura 5.7: Estructura de un flujo de bytes de entrada.

Los dos primeros bytes de la cabecera corresponden a bytes de sincronización, que como veremos más adelante en la tabla de códigos (véase tabla 5.2) se codifican con el valor hexadecimal `0xAA`. Sólo si estos dos bytes primeros son correctos, se sigue procesando el paquete; si no lo son, entonces se desecharán bytes hasta que se lean válidos (dos `0xAA` consecutivos). El tercer byte indica la longitud del cuerpo de datos, tomando un valor de comprendido entre 0 y 169. Un valor superior codifica un error en el paquete.

El cuerpo de datos es una simple secuencia de bytes, referentes a datos exclusivamente que, en caso de ser válidos, pueden ser extraídos acorde a una estructura interna. La longitud de la serie de bytes está definida por el campo `[ PLENGTH ]` de la cabecera.

El campo de *Checksum* se utiliza para verificar el cuerpo de datos. Se calcula en un proceso en tres pasos:

1. Se suman todos los bytes del cuerpo.
2. Se quitan los 8 bits menos significativos de la suma.
3. Se realiza la inversa (complemento a uno) de esos 8 bits.

Si el valor contenido en `[ CHKSUM ]` no coincide con el calculado, se desecha el paquete. Si son válidos, entonces se puede «parsear» o convertir el cuerpo del paquete en valores de dispositivo. La estructura del cuerpo se define en la figura 5.8.

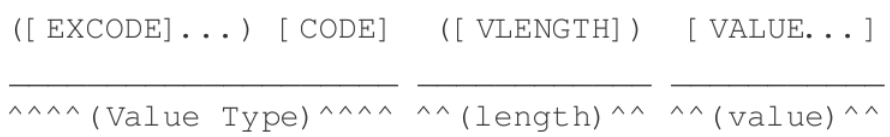


Figura 5.8: Estructura de un flujo de bytes de entrada.

El cuerpo puede comenzar con 0 o varios campos `[ EXCODE ]` (**Ex**tended **Code**), codificados en hexadecimal `0x55`. Estos bytes `[ EXCODE ]` se utilizan en conjunto al campo `[ CODE ]` para determinar el tipo de paquete. Lo más importante del campo `[ CODE ]` es el hecho de que determina si se tiene un valor (campo `[ EXCODE ]`) de múltiples bytes o de 1 solo byte.

Si `[ CODE ]` está entre `0x00` y `0x7F`, entonces `[ VALUE ]` es de un solo byte, por tanto, no hay campo `[ VLENGTH ]` (Value Lenght).

Si `[ CODE ]` es superior a `0x7F`, entonces `[ VLENGTH ]` aparece detrás de `[ CODE ]` y denota el número de bytes en `[ VALUE... ]` (múltiples valores).

La lista de códigos y su significado se detalla a continuación, separándolos en códigos de un solo byte y códigos de múltiples bytes:

Códigos de un solo byte:

Extended Code	CODE	LENGTH	Significado
0	`0x02`	-	Calidad de la señal (0-255)
0	`0x04`	-	Valor de atención (0-100)
0	`0x05`	-	Valor de meditación (0-100)
0	`0x16`	-	Fuerza del pestañeo (0-255) Solo si ocurre dicho evento

Cuadro 5.1: Códigos de operación de un byte.

Para realizar todo este proceso, y siguiendo el segundo punto de desarrollo, se ha diseñado un submódulo de conversión. El punto de entrada es la clase `StreamParser`. Esta clase es una clase C++ estática que encapsula una estructura de datos y métodos sobre ella. La estructura, llamada `ThinkGearStreamParser` y definida en el listado 5.4, implementa un autómata finito para el reconocimiento de paquetes válidos.

El carácter `type` se utiliza para denotar si el paquete es inválido o no. El carácter `state` se utiliza para indicar el estado del autómata o estado de la conversión del paquete. El resto

Códigos de varios bytes:

Extended Code	CODE	LENGTH	Significado
0	0x80	-	Valor de la onda en bruto: valor con signo de 16 bits en complemento a dos. Rango de -32768 a 32767
0	0x83	-	Potencia del electroencefalograma: ocho grupos de 3 bytes cada uno que representan los valores de las ondas *delta, theta, low-alpha, high-alpha, low-beta, high-beta, low-gamma* y *mid-gamma*
0	0x55	-	No se utiliza. Reservado para `[ EXCODE ]`
0	0xAA	-	No se utiliza. Reservado para `[ SYNC ]`

Cuadro 5.2: Códigos de operación de múltiples bytes.

```
1  typedef struct _ThinkGearStreamParser {

3      unsigned char   type;
4      unsigned char   state;

6      unsigned char   lastByte;

8      unsigned char   payloadLength;
9      unsigned char   payloadBytesReceived;
10     unsigned char   payload[256];
11     unsigned char   payloadSum;
12     unsigned char   chksum;

14     void (*handleDataValue)( unsigned char extendedCodeLevel,
15                              unsigned char code, unsigned char
                                    numBytes,
16                              const unsigned char *value, void *
                                    customData );
17     void  *customData;

19 } ThinkGearStreamParser;
```

Listado 5.4: Estructura de conversión de paquetes.

de caracteres guardan los campos relativos a la longitud del paquete, la carga, y la suma comprobación, tanto la recibida como la calculada. Contiene un puntero a función (primer `void`) que utiliza para devolver al manejador el paquete convertido. El último puntero a void, `void *customData`, puede almacenar cualquier tipo de variable; en este caso se utiliza como entero para guardar el ID de dispositivo.

El proceso seguido será esperar en la entrada el byte (carácter o char) de sincronización `0xAA` – la llegada de bytes al autómata se realiza mediante el médoto `THINKGEAR_parseByte`, que recibe como parámetro la propia estructura y el siguiente byte -. En ese estado, sólo podrá avanzar al siguiente recibiendo un segundo byte `SYNC`; con cualquier otro valor volverá al estado inicial. Tras el segundo byte `SYNC` se reconoce cabecera de paquete, y el siguiente

byte será el campo `LENGTH` de longitud. Según el valor de este campo, podrá pasar al estado de longitud correcta (si la longitud es menor de 170) o volverá al estado inicial (si la longitud es mayor o igual a 170). En el estado actual (4) calculará la suma de comprobación y pasará al estado de *checksum*; si es correcto, llegará al estado final del autómata y podrá procesar el cuerpo de datos del paquete, pero si es incorrecto volverá al estado inicial nuevamente.

Cuando llega al estado final, se invoca al método `parsePacketPayload`, encargado del tratamiento de los datos de onda. Este método leerá la secuencia de bytes extrayendo los campos de operación y, en función de cual sea el código, extrayendo un valor o varios. Cuando todo se ha realizado correcto, se llama al método de retrollamada, el contenido en la estructura como `void (*handleDataValue)`, se pasan los datos y se da por concluida la conversión de la trama.

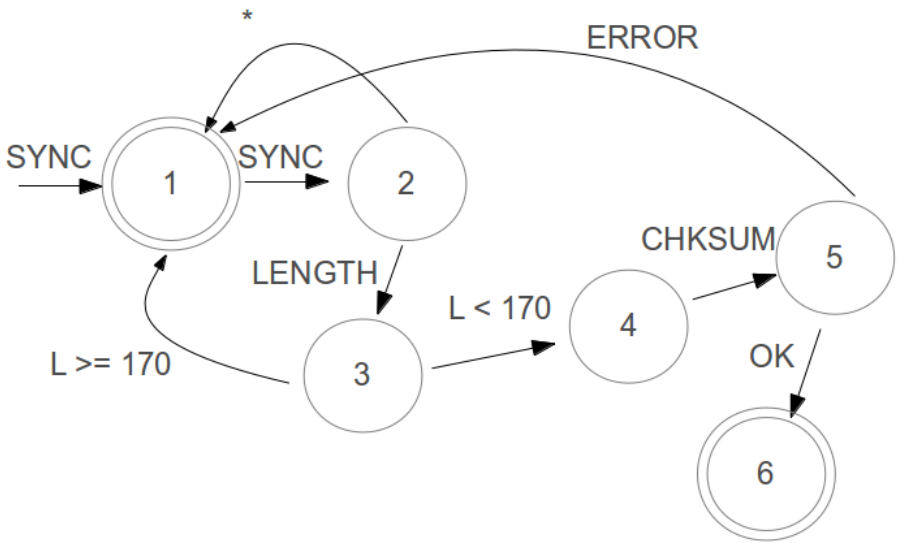


Figura 5.9: Autómata finito de conversión de paquetes. Los estados representan: 1 - Sync, 2 - Sync confirmado, 3 - Recibir longitud, 4 - Longitud correcta, 5 - Suma de comprobación, 6 - Suma correcta, proceso de la carga de datos.

Sin embargo, los valores mandados al manejador a través del método de retrollamada aún no son válidos para la aplicación. Primero, aún no se han asignado a un dispositivo concreto; segundo, sólo se ha enviado el código de operación y el/los byte/s que le corresponden. Por tanto, el manejador hará una última conversión y enviará los datos finales a las instancias de dispositivo, listos para uso de la aplicación o entrenamiento del dispositivo a partir de ellos.

El control de todo el proceso anteriormente descrito lo lleva a cabo el submódulo de control y gestión, tal y como se matizó en el punto tercero de desarrollo, empleando para ello un *handler* del dispositivo. En el momento de escribir este documento, se dispone sólo del

manejador de dispositivo MindWave; para ello, tenemos una clase `MindWaveHandler`.

Esta clase se encarga de realizar las operaciones posibles sobre las instancias de dispositivo. Sin embargo, en esta sección nos centraremos en aquellas relativas al proceso de tratamiento de ondas, conversión del flujo de datos, etc. La interfaz completa se discutirá en la siguiente sección 'Módulo de dispositivo'. Los aspectos que nos interesan por ahora se muestran en la siguiente interfaz (listado 5.5), exceptuando aquellos relativos a otras operaciones.

```
1   class MindWaveHandler {
2         public:
3           ...
4           void update(int id);
5           ...
6           void completeData(int idDisp, unsigned char code, unsigned
                char valueLength, const unsigned char *value);
7           ...
8         private:
9           StreamParser* _parser;
10          char _bytes[MAXBUFF];

12          map<int, BufferedAsyncSerial*> _serialPorts;
13          map<int, MindWaveDisp*> _disps;
14          ...
15  };
```

Listado 5.5: Interfaz del manejador de dispositivo en el tratamiento de ondas.

Esta clase guarda guarda instancias del estado de los dispositivos y de los bufferes de almacenamiento intermedio de la lectura asíncrona en los puertos en un *map*. *Map* es un *template* de la biblioteca estándar STL (*Standard Template Library*) de C++, encargada de almacenar elementos asociados a una clave identificadora.

Utilizará el elemento i del *map* `_serialPorts` cuando el buffer asociado (de la clase `BufferedAsyncSerial` vista en la sección 5.2 en la página 88) al elemento i se encuentre lleno. Se notificará a la clase `MindWaveHandler` utilizando el método `update`. Este método tiene como parámetro el identificador i a utilizar. Cuando la notificación llega, entonces se procede a extraer un *stream* de datos y guardarlo en el *array* `_bytes`. El *array* de bytes es enviado a la instancia de la clase `StreamParser`, cuya función es realizar el proceso descrito en el apartado anterior de reconocimiento y conversión de paquetes. Cuando el paquete reconocido es válido, se llama a la función de retrollamada almacenada que, a través de la fachada de la plataforma (se detalla en la sección 5.4) se envía al método `completeData`.

La función que se desempeña aquí es el tratamiento final de los datos antes de guardarlos en la instancia de dispositivo correspondiente. Recibe como datos de entrada el ID del dispositivo destino, el código de operación, la longitud de los valores, y los valores correspondientes. En función de la longitud sabrá si es un paquete *single-byte* o *multi-byte*. En

cualquiera de ellos, separará los datos de forma sencilla y completará la información de dispositivo empleando los métodos *setter* que posee.

En todo momento se tiene en cuenta cualquier excepción que pueda tener lugar. Para un correcto funcionamiento, preparado para cualquier excepción, han de tenerse en cuenta los siguientes escenarios posibles:

- Llega un aviso de *buffer* lleno de un dispositivo que se ha dado de baja en el sistema.
- Se produce cualquier excepción en el puerto mientras se están extrayendo los datos del buffer.
- No se reconoce ningún paquete válido.
- Una vez obtenido el paquete válido, el dispositivo ya no está conectado al sistema.

Observando los posibles escenarios, vemos que el error más probable es la desconexión del dispositivo o del *dongle*[2] del mismo. El primer y cuarto caso se controlan con una comprobación en el *map* sobre la existencia del dispositivo.

La comprobación se realiza con los métodos `find` y `end` de *map*: si el `id` requerido existe, `find` devuelve un *iterator*[3] apuntando al objeto; si no lo encuentra, devuelve la última posición del *map*, que es un puntero a `NULL` (el mismo que devuelve la función `end`), y en este caso no realiza las operaciones dentro del condicional *if*. De este modo, cualquier comportamiento erróneo posible producido por una desconexión, agotamiento de baterías, etc. no producirá efecto adverso en el sistema y continuará operando.

En el tercer caso no hay ninguna operación o comprobación necesaria que realizar, ya que si no se obtiene ningún paquete válido, simplemente no se actualizará la información de la instancia de dispositivo y seguirá mostrando el anterior estado.

En el segundo caso, y dado que la lectura se lleva a cabo empleando la libería Boost como se comentó en la sección 5.2, se emplea Boost_System para el control de errores y excepciones. De este modo, basta encerrar la llamada en un bloque y capturar una excepción propia de tipo `boost::system::error`.

A continuación se detallan los diagramas de clase y de secuencia finales del módulo, en las figuras 5.10 y 5.11 respectivamente.

Desde el puerto serie llega una notificación al manejador de que el buffer intermedio tiene datos suficientes para extraer. Entonces se procede a leer k bytes y se recuperan. La repetición de las llamadas `parseBytes` y `completeData` representa el hecho de que en el proceso de lectura secuencial se reconocen paquetes válidos antes de «parsear» toda la secuencia en

[2] *Dongle* es el nombre que recibe el receptor inalámbrico del dispositivo que va conectado al puerto USB.

[3] *Iterator* es un patrón de diseño que permite recorrer un contenedor de datos sin necesidad de conocer su estructura interna.

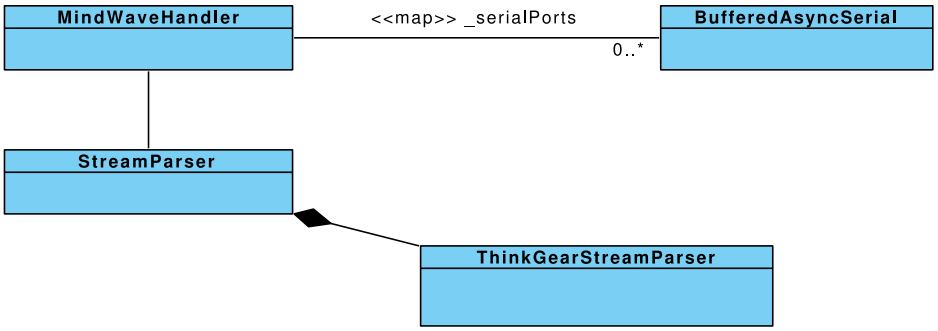


Figura 5.10: Diagrama de clases de tratamiento de ondas.

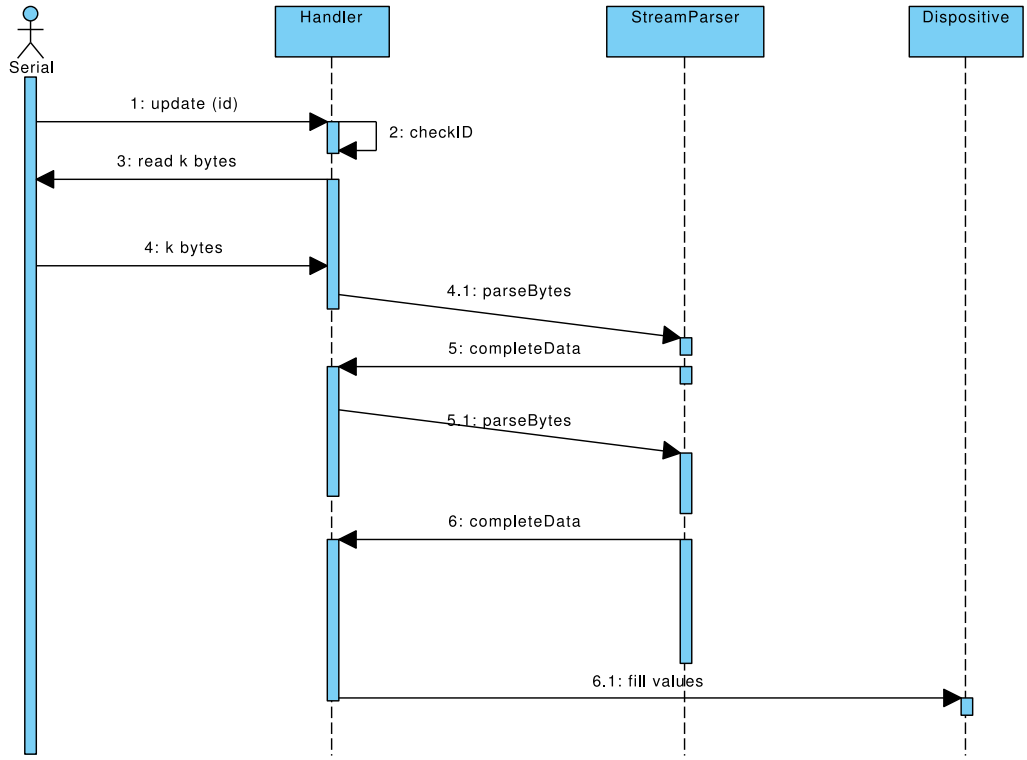


Figura 5.11: Diagrama de secuencia de tratamiento de ondas.

el autómata `StreamParser`. Una vez obtenido el paquete válido, se actualiza el estado del dispositivo (*fill values* no representa un método real, sino el hecho de esa actualización).

5.3.3 Ventajas

Gracias al módulo implementado se tiene una gestión eficiente de los datos y el proceso de conversión. El hecho de acceder a los datos de entrada sólo cuando están disponibles y son suficientes, evita que el sistema se ralentice al estar continuamente realizando operaciones innecesarias; además al estar diferenciadas por un identificador, las operaciones sobre ellas son directas gracias al empleo de estructuras *map*.

El tener un submódulo aparte para la conversión de datos evita que el submódulo de control tenga responsabilidades de más que no le corresponden, reduciendo el acoplamiento. Además, mejora la extensibilidad del sistema, ya que un cambio en el sistema de tratamiento de los datos no implica una modificación del sistema, sólo del submódulo. La conversión por medio de un autómata finito proporciona un método muy simple y muy eficiente de tratar los datos, mejorando el rendimiento del sistema.

Nuevamente el enfoque multi-hilo implementado entre las comunicaciones del módulo anterior y el control de éste permite aprovechar varios procesadores del sistema y ejecutar las operaciones rápidamente, evitando un retraso o *lag* excesivo en cualquiera de ellas.

5.4 Módulo de dispositivo

El módulo de dispositivo es uno de los módulos principales de la plataforma. Este módulo se encarga de funciones tales como representar el dispositivo de EEG conectado y almacenar su estado, la gestión de los mismos, su conexión/desconexión del sistema, el entrenamiento del dispositivo y la emisión de operaciones a otros módulos.

En este módulo se caracteriza cada dispositivo y se representan unas características propias de él que conforman su estado en cualquier instante T. Este estado puede ser, o no, la totalidad de todos los tipos de información que envía, sólo una parte, o incluso nuevos atributos que puedan ser calculados y/o derivados de los anteriores. En cualquier caso se representará el estado del dispositivo de forma mínima con la información del identificador del dispositivo, la onda en bruto que se reciba, y la señal de emisión. Además debe asegurar que cualquier dispositivo que se conecte a la plataforma cumpla esas características mínimas.

Además, el módulo puede ampliar el estado del dispositivo calculando nuevos valores específicos que derivan de la onda en bruto de entrada, que guarda el espectro completo de las ondas descritas en la sección 2.2.2. De la misma forma, este módulo proporciona funcionalidad de entrenamiento del dispositivo que permita adaptarlo al usuario, proporcionando valores de *Atención y Meditación* adaptados a él. El método de entrenamiento se discutirá más adelante, y está calculado a partir de la onda en bruto.

Tiene también la función de administrar y gestionar todas las instancias de dispositivo y las operaciones que sobre las mismas pueden realizarse, tales como desconectarse del sistema, conectar en el *puerto XXXX* el primer dispositivo disponible o conectar al *dispositivo concreto YYYY*. Cuando dé de baja un dispositivo, o cuando se conecte, deberá hacer lo mismo con la comunicación, instanciando puertos de comunicación. Ésto se llevará a cabo desde el manejador de dispositivo mencionado en 5.3.

Como última función, se encarga de ocultar toda esta funcionalidad del resto del sistema estableciendo un único punto de entrada que presenta las operaciones disponibles; se le denomina *Fachada* y se discutirá en 5.4.2.

Para proporcionar la funcionalidad descrita se han diseñado los siguientes submódulos:

- Un primer submódulo encargado de representar el estado de dispositivo y gestionar las operaciones sobre los mismos.
- Un segundo submódulo encargado de proporcionar operaciones matemáticas sobre el estado del dispositivo para obtener nuevos valores.
- Un tercer submódulo con la función de entrenar el dispositivo para adaptarlo al usuario.
- Un último submódulo encargado de proporcionar un único punto de acceso a la plataforma y encapsular toda la funcionalidad de los módulos inferiores.

Gráficamente, la descripción funcional del módulo es la mostrada en la figura 5.12.

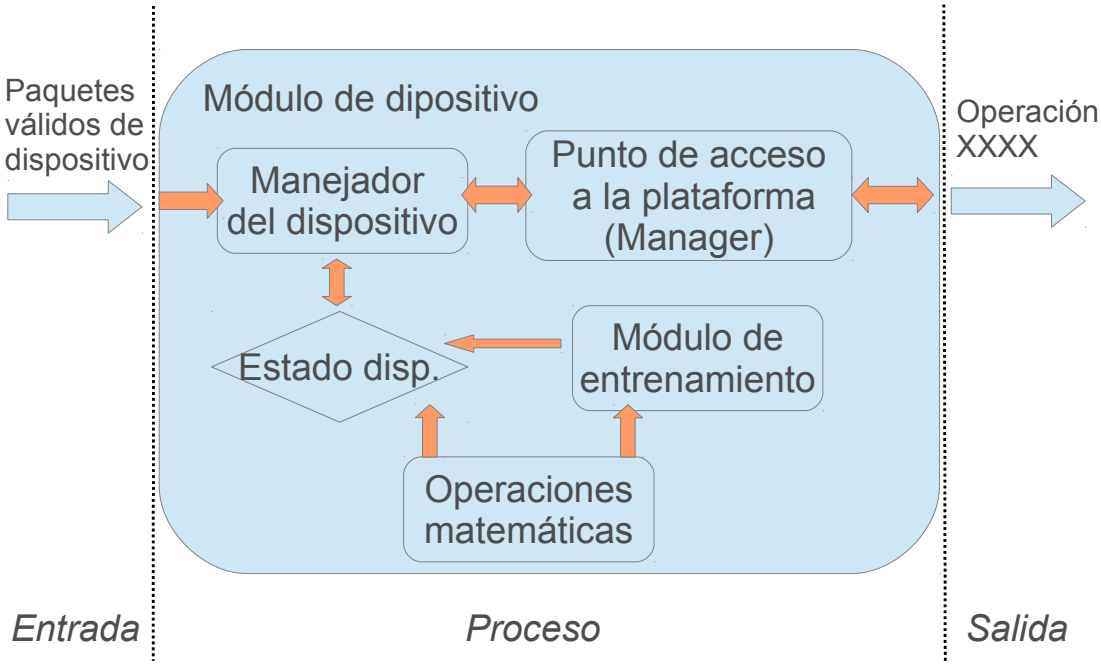


Figura 5.12: Descripción funcional del módulo de dispositivo.

Aclarar en la descripción gráfica que el módulo de entrenamiento interacciona con el estado del dispositivo en lugar del manejador porque no envía información al manejador con

los datos a actualizar, sino que directamente condiciona el cálculo de valores y los modifica en el mismo proceso, sin necesidad de llamadas a procedimientos adicionales.

En cuanto a la conexiones y dependencias del módulo, depende totalmente del módulo de tratamiento de ondas, que a su vez dependía del módulo de comunicaciones; sin dispositivos ni entrada de datos este módulo no tiene función alguna. En la salida no tiene dependencia con otros módulos, pero sí conexiones con el módulo de representación gráfica y el módulo de control del microcontrolador, así como cualquier aplicación que haga uso del estado de dispositivo.

5.4.1 Problemática

Uno de los principales problemas que se tratan de resolver en este módulo es que se pueda extender fácilmente la funcionalidad de gestión del estado del dispositivo y sea posible incorporar nuevos dispositivos de una forma eficaz y sencilla. De forma parecida a como se procedió en la sección 5.2.2 con una interfaz genérica para las comunicaciones, podríamos proceder para las instancias de dispositivo que almacenan su estado. Sin embargo, cada dispositivo puede llegar a ser un mundo, y tener atributos que otras no tengan o calculen, o por el contrario no tenerlos; puede ocurrir que todos los dispositivos no soporten el mismo número de tipos de onda (por ejemplo, el dispositivo MindWave de NeuroSky utilizado en este Proyecto Fin de Carrera no detecta la actividad del *ritmo Mu*). Es por ello por lo que supuso un problema establecer la interfaz genérica de dispositivo, con los atributos y operaciones mínimos que cualquier dispositivo debería soportar o poder calcular.

Otro problema de robustez que ya se presentó en la sección 5.3.1 se presentaba en este módulo: la estructura de gestión de las instancias de dispositivo inicialmente se diseñó utilizando un *vector*; al observarse el problema en el módulo de tratamiento de onda se modificó también el diseño de dispositivo que tratamos aquí. Al sustituirlo por una estructura indexada como *map*, se simplificaban las operaciones sobre ellos y permitía administrarlas con el mismo ID de la comunicación.

Desde el primer momento se pensó en encapsular toda la funcionalidad proporcionada y mostrarla bajo un punto de acceso controlado, tanto por seguridad como por facilidad para el programador de aplicaciones, que solo debe buscar la operación deseada en la *fachada* para poder utilizar la información en la capa de aplicación.

Otra cuestión que se planteó al principio, y que fue motivada por un problema surgido en las primeras pruebas con el dispositivo, fue que la plataforma fuese capaz de seguir mostrando información congruente cuando el dispositivo presentaba problemas de señal o no se adaptaba bien al usuario. Habían ocasiones en las cuales el dispositivo dejaba de emitir señales al 100 % de intensidad y dejaba de enviar paquetes de datos de los tipos ‘Atención’ y ‘Meditación’, pero sí enviaba los paquetes de datos de los tipos ‘Onda en bruto’ y ‘Valo-

res de onda *Low-High*', cuando los primeros son obtenidos a partir de los segundos. En otras ocasiones el dispositivo se hallaba bien colocado pero no respondía bien ante el usuario. Para resolver estos problemas se planteó incluir una solución que permitiese seguir calculando y mostrando valores congruentes y válidos aunque surgiese algún problema (siempre y cuando se recibiese al menos el espectro de onda u onda en bruto).

Sin embargo, cuando se implementó la solución de entrenamiento, se vio que requería demasiado tiempo no sólo en la captura de datos, sino mucho más en el entrenamiento. Por ello se planteó incorporar la solución de entrenamiento como aplicación externa y añadir un módulo adicional de trazas que permitiese generar archivos con los paquetes recibidos y utilizarlos posteriormente, en otro equipo diferente si se deseaba mientras se seguía utilizando la plataforma, para realizar el entrenamiento (entrenar el sistema para que tenga un grado de acierto bueno puede requerir unas 500 o 600 muestras – recibiendo cada muestra una vez por segundo - y unas 6 horas de entrenamiento).

En términos de eficiencia se planteaba el problema de que el cálculo o estimación de valores requiriese demasiado tiempo como para hacer perceptible el retraso en la actualización de los mismos e impedir mostrarlos a la frecuencia que se recibían. Para ello se llevó a cabo un estudio sobre el rendimiento de implementaciones de la *Transformada de Fourier*, necesaria para descomponer el espectro de onda, y la mejor configuración del entrenador en cuanto a precisión/rendimiento (con entrenador nos referimos al algoritmo de entrenamiento, para el cual se ha optado por una red neuronal, concretamente un perceptrón multicapa, descrito extensamente en la sección 2.3.3).

5.4.2 Solución

Este módulo presenta una gran complejidad y muchos y diversos problemas en su diseño, por lo que no es sencilla su implementación. Funcionalmente, el módulo se encargará de representar el estado de los dispositivos, estableciendo una interfaz genérica que permita conectar cualquier tipo de dispositivo. Como todos los dispositivos no tienen las mismas características ni soportan las mismas funcionalidades, habrá que extraer aquellas básicas y comunes a todos ellos, y posteriormente extender todo lo adicional que se precise. En esa representación guardará e irá actualizando todos los valores y medidas que extraiga de los paquetes válidos de dispositivo que lleguen del módulo de tratamiento de ondas. Sin embargo, debido a los problemas comentados en el punto anterior de Problemática, necesitará tratar primero los datos antes de almacenarlos todos, de modo que a partir de la entrada pueda obtener unos valores calculados/estimados para las situaciones de fallo comentadas. Además, tendrá la función de entrenar el sistema para adaptarlo al usuario, para lo cual utilizará los datos calculados en el proceso anterior. Por último, ofrecerá un punto de acceso único y que encapsule toda la funcionalidad del módulo, además de encargarse de administrar las instancias de dispositivo, crear nuevas o eliminarlas, envíar comandos al dispositivo, etc.

Para ello, se establecen una serie de características o puntos de desarrollo:

- Se debe crear una representación abstracta del estado de los dispositivos.
- Esta representación debe implementar un mecanismo que permita extender la cantidad de dispositivos que pueda manejar el sistema, implementando una interfaz genérica que todos implementen.
- Deberá gestionar las instancias de dispositivo y sus conexiones de forma eficiente y simple, libre de errores y para cualquier cantidad de ellas.
- Además podrá enviar comandos específicos al dispositivo, tales como conectar a un ID específico.
- Incluirá funciones matemáticas para ser capaz de estimar valores cuando se de una situación de pérdida parcial de señal y/o paquetes.
- Permitirá al usuario entrenar el sistema.
- Encapsulará toda la funcionalidad del módulo (Fachada).

El primer paso en la implementación de la solución es crear una representación del estado del dispositivo. El estado del dispositivo lo componen el identificador de dispositivo que lleva asociado de fábrica, todos los valores de ondas cerebrales que detecta, otros posibles valores asociados a las ondas que pueda enviar, como nivel de atención del usuario, y la información sobre la señal recibida y/o el estado de la conexión.

En el momento de redactar este documento sólo se dispone de soporte para dispositivos MindWave de NeuroSky, por lo que sus características no pueden ser tomadas como universales sin antes comparar con otros dispositivos. Para ello se ha realizado una comparación de los tres dispositivos comerciales analizados en el Anteproyecto sobre los que se eligió el dispositivo objeto de estudio en este Proyecto Fin de Carrera.

Dispositivo	**Frecuencia neurológica**	**Entrada de onda en bruto**	**Valores inmediatos derivados**
NeuroSky MindWave	0.5-50 Hz	Sí	Atención, Meditación y pestañeo (pestañeo sólo mediante API privativa)
NeuroSky MindSet	3-50 Hz	Sí	Atención, Meditación y pestañeo (pestañeo sólo mediante API privativa)
Emotiv EPOC	0.2-50 Hz	Sí	Expresiones faciales, estados emocionales, nivel de Atención y Concentración

Cuadro 5.3: Dispositivos de EEG y sus características.

Como se aprecia en la figura 5.3, todos los dispositivos trabajan en un rango de frecuencia semejante, soportan la entrada en bruto de ondas (*Raw EEG*), y algunos valores calculados a

partir de éstas. Los tres tiene en común el valor de Atención, teniendo cada uno otros propios. Aunque el valor de Meditación no se encuentre en los tres (lo proporcionan los dispositivos NeuroSky), puede ser calculado a partir de la señal «Raw». Además, y aunque no se incluya en la tabla por motivos de espacio, todo dispositivo tiene un ID único asociado. De este modo, el estado del dispositivo se establece con las siguientes características, mostradas en la interfaz del listado 5.6 y descritas a continuación:

```
1   class DispInterface {
2     protected:
3       int _id;
4       StatusID _status;

6       char _ms, _ls;

8       int _meditationLevel, _attentionLevel;
9       deque<int> _RawWaves;
10      int _signal;
11  };
```

Listado 5.6: Interfaz genérica de dispositivo.

- Un valor entero `ID` que representa el identificador del dispositivo dentro del sistema; el ID propio está representado por los *char* `_ms` y `_ls`. Se utilizan dos bytes por si el identificador es demasiado largo para almacenarlo en uno solo.
- Un estado de conexión del dispositivo (`StatusID`) que indica si el dispositivo se encuentra *desconectado*, *conectado*, o *conectado con baja intensidad* (que puede ocasionar pérdida de paquetes).
- Un valor entero que mida la intensidad de la señal.
- Dos valores enteros para almacenar los valores calculados por el dispositivo de la Atención y la Meditación del usuario.
- Una cola doblemente enlazada de enteros para almacenar los valores de la onda en bruto. La cola es doblemente enlazada porque tendrá intensa actividad en los extremos más que en el recorrido secuencial, debiendo insertar valores al final y, alcanzado un tamaño máximo, extraer del principio de la cola.

En la interfaz se incluyen además las operaciones básicas sobre los atributos; de esta forma, cualquier implementación que extienda de la interfaz será válida y utilizable en todo el sistema, ya que se trabaja sobre la interfaz. El desarrollador ahora debe programar la implementación concreta de la interfaz para determinado dispositivo y extendiendo si lo desea los atributos y operaciones disponibles. Por ejemplo, si es capaz de calcular un valor para el nivel de sueño (el cual se asocia con las ondas Delta) y desea que lo utilice el sistema, primero deberá crear una clase que herede de la interfaz `DispInterface`, implementar las

operaciones que impone la interfaz y, posteriormente, añadir las suyas, obteniendo una nueva clase como se muestra en la figura 5.13. De esta forma la clase `nuevoDispositivo` cuenta con la funcionalidad estándar más la funcionalidad para tratar niveles de sueño, codificados como otro valor entero.

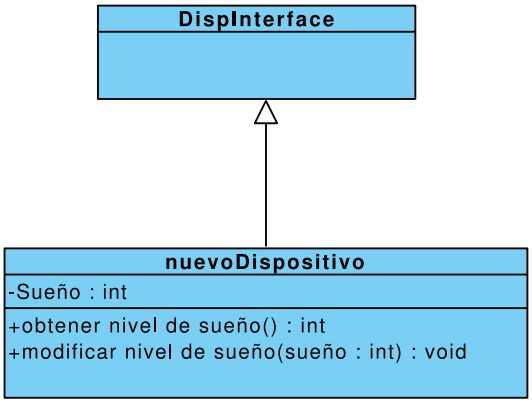


Figura 5.13: Incorporación de un nuevo dispositivo.

Por defecto, el sistema incluye la implementación concreta para los dispositivos MindWave, que incorporan una serie de campos y operaciones adicionales para almacenar valores de onda *Low-High* y valores estimados y entrenados de atención y meditación. Una vez que se tienen las clases de representación de estado del dispositivo, es necesario completar el submódulo con mecanismos que los gestionen.

De esta labor se encarga el manejador de dispositivo, que fue introducido en el módulo de tratamiento de ondas y que se comentó que compartía con el módulo actual. El resto de la interfaz que no se comentó en aquel momento incluye operaciones para enviar comandos al dispositivo, actualizar el estado de uno de ellos(identificado por el ID del sistema), operaciones para incorporar, eliminar y recuperar un dispositivo concreto, y obtener su estado de conexión y señal. Los atributos faltantes en el manejador son las estructuras que guardan las instancias de conexión, dispositivo y señal de envío de datos.

Para gestionar los dispositivos (el mismo proceso se aplica en las conexiones o señal de envío), el manejador trabaja sobre las estructuras *map* ya vistas en este capítulo empleando el ID del sistema, ya que es más simple trabajar con un valor entero que con una secuencia de bytes. Cuando se conecta un dispositivo D al sistema, se le asocian una conexión y una instancia de estado. Para ello, se asinga un identificador no utilizado y se crea una nueva entrada indexada en las 3 estructuras, una para cada objeto, con el ID como clave de indexación. Recuperar un determinado objeto implica ejecutar el método de búsqueda en *maps* `find()`; este método devuelve un *iterador* apuntando a la instancia asociada a la clave in-

```
1   class MindWaveHandler {
2     public:
3       void writeCommand(int id, const char& command);
4       void update(int id);

6       void addDisp(int id, MindWaveDisp* disp);
7       void removeDisp(int id);
8       MindWaveDisp* getDisp(int id);
9       int isSending (int id);
10      int getPoorSignal(int id);

12    private:
13      map<int, BufferedAsyncSerial*> _serialPorts;
14      map<int, MindWaveDisp*> _disps;
15      map<int, int> _sendingData;
16  };
```

Listado 5.7: Interfaz del manejador para dispositivo.

dexada si ésta se encuentra, o un *iterador* al final de la estructura (que es un `NULL`) si no se encuentra. Se procede del mismo modo en el borrado.

Este método proporcionado por *map* también nos sirve para implementar mecanismos de seguridad y recuperación de errores en el sistema al mismo tiempo que se implementa la funcionalidad, ya que el *iterador* nos proporciona lo necesario. El procedimiento implementado es el siguiente:

1. Declarar iterador sobre la estructura de datos.
2. Lanzar la búsqueda con el ID del sistema (`find(id)`).
3. Comparar iterador con el final de la estructura.
4. Si es válido, realizar la operación en cuestión; si no lo es, tratar el error y recuperar la ejecución del sistema mostrando el mensaje oportuno.

Este mismo procedimiento se ha implementado para las operaciones restantes. En la operación de envío de comandos (`writeCommand`) han de implementarse otras secuecias de control, pues en la sección 5.4 mencionamos brevemente que los comandos podrían ser de tres tipos: dirigidos a todos los dispositivos existentes, dirigido a uno específico, o dirigido al primero disponible en un puerto específico, puntualizando que esta diferencia se da en los comandos de conexión, pues para la desconexión solo hay uno, independientemente del número de dispositivos destino. Esto se ha implementado acorde al diagrama de flujo de la figura 5.14.

En caso de ser para todos los dispositivos, conectados, un comando de difusión como apagar todos, simplemente se itera en las conexiones y se envía el comando. Pero en el caso de ir en un puerto específico, se ha de diferenciar si es a un dispositivo específico o al primero disponible. Para el primer dispositivo disponible, se utiliza el código `0xC2`, y no es necesario

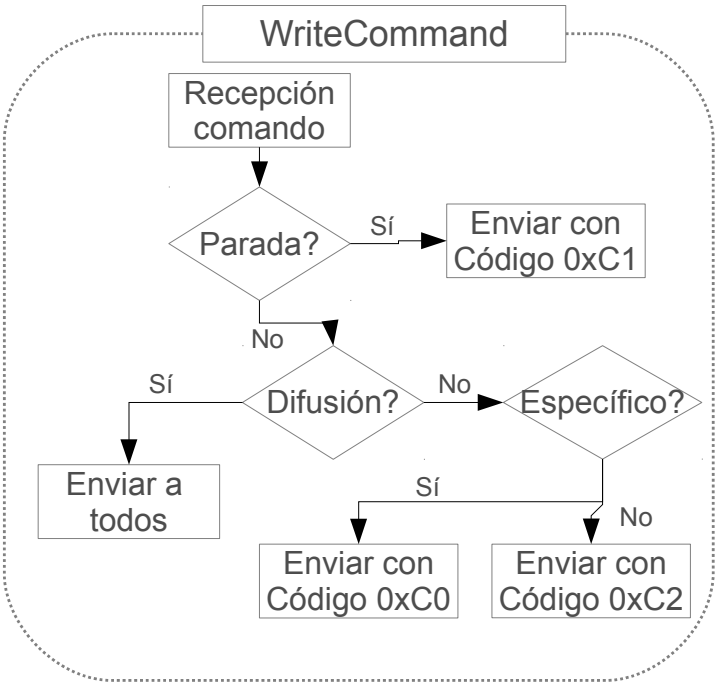


Figura 5.14: Diagrama de flujo para el envío de comandos.

mandar bytes adicionales; en el caso de dispositivo específico, se envía el comando `0xC0` y a continuación se envían los bytes correspondientes al ID de fábrica del dispositivo. El código de parada o desconexión para todos ellos es el mismo: `0xC1`.

En la sección 5.3.2 se comentó ya como se pasaba la información de los paquetes válidos al estado de dispositivo; llegados a este punto tenemos la funcionalidad descrita en los puntos de desarrollo 1, 2, 3 y 4. El siguiente paso es el submódulo matemático que ofrezca funcionalidad para calcura y/o estimar valores derivados de la actividad cerebral que recoge el dispositivo de EEG (*Raw EEG*), así como el entrenamiento.

Antes de poder describir el cálculo de valores, que va ligado al entrenamiento, pues se estima acorde al usuario, es necesario describir cómo se han de transformar los datos para prepararlos y cuál ha sido la implementación para dicho proceso.

Como ya se comentó en el módulo de tratamiento de ondas, y siguiendo la especificación de datos de MindWave, los valores en bruto de onda recibidos son valores con signo de 16 bits, comprendidos en el rango de -32768 a 32767. Estos valores contienen la información del espectro de onda captado, pero un valor en sí mismo no aporta nada, ha de analizarse junto a otros; ha de buscarse un algoritmo eficiente que tome un conjunto de muestras de entrada y descomponga la señal. Se ha utilizado un algoritmo que computa la energía en cada frecuencia especificada por la bandas de frecuencia de onda resultado de aplicar a las muestras de entrada la Fast Fourier Transform (FFT), o Transformada Rápida de Fourier.

FFT es un eficiente algoritmo que permite calcular la transformada de Fourier discreta

(DFT, Discrete Fourier Transform) y su inversa. Es un algoritmo de gran importancia en el tratamiento digital de señales y filtrado digital [Bri88]. El algoritmo establece algunas restricciones en la señal y en el espectro resultante (bandas), ya que, por ejemplo, la señal de la que se tomaron muestras y que se va a transformar debe consistir de un número de muestras igual a una potencia de dos. La mayoría de los analizadores utilizan las potencias de dos de 512, 1024, 2048 o 4096 muestras. El rango de frecuencias cubierto por el análisis depende de la cantidad de muestras recogidas y de la proporción de muestreo. Para nuestro módulo se ha establecido el tamaño de la entrada en 1024 muestras y el rango de frecuencias de salida en 50.

El proceso seguido se muestra en la figura 5.15.

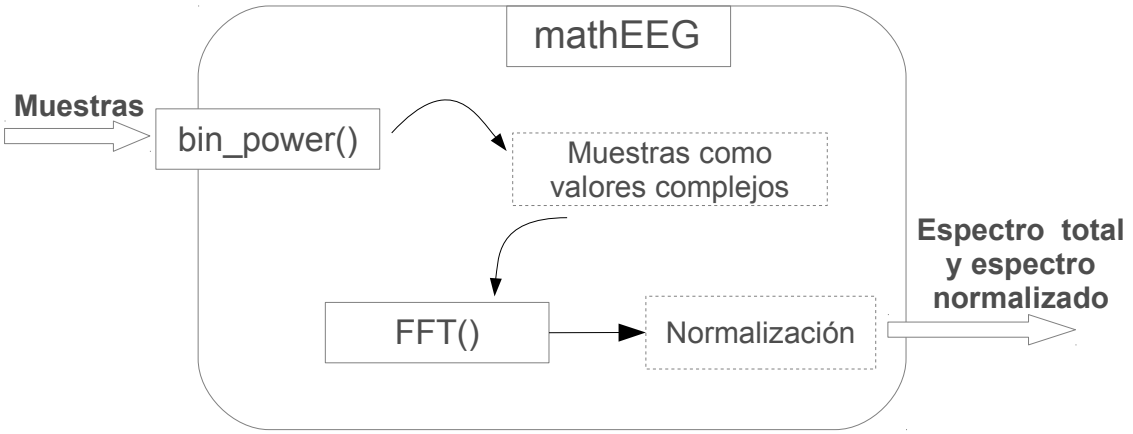


Figura 5.15: Proceso de descomposición de las muestras de EEG y obtención del espectro normalizado.

Primero se reciben las muestras, la frecuencia de las bandas (establecida de forma uniforme en un rango) y la frecuencia de muestreo. Dado que las muestras se reciben como valores enteros y el algoritmo de la Transformada de Fourier emplea valores complejos, se transforman estableciendo la parte *real* con el valor entero (convirtiendo a *double*) y la parte *imaginaria* se establece a $0{,}0$. Si en algún caso se recibiesen menos muestras de una potencia de 2, se añadirán números complejos (0 + 0j) hasta la potencia de dos más cercana. Con los datos ya preparados, se aplica la transformada de Fourier inversa.

Aplicada la transformada, se procede a calcular la salida de la función de potencia (bin_power), es decir:

- Un *array* de 50 valores correspondiente al espectro de onda en cada frecuencia de onda.
- Un *array* de 50 valores normalizados correspondiente al espectro de onda en cada frecuencia de onda normalizada al total de las frecuencias.

Básicamente el procedimiento llevado a cabo consiste en recorrer cada banda de frecuencia, y en cada una de ellas obtener las posiciones de inicio y de fin del *array* de la transfor-

mada en base a las frecuencias actual y siguiente (es función de la frecuencia de muestreo también), y sumar los valores comprendidos entre esas posiciones. Al mismo tiempo se van sumando los valores en un acumulador *sum* para emplearlo luego en la normalización sin necesidad de repetir el bucle. Una vez terminado este proceso, se completan los valores del *array* normalizado estableciendo en cada posición i el valor del espectro en i partido de la suma del espectro total.

Ambos arrays son devueltos como punteros en la misma llamada al método `bin_power()`. Con estos dos *arrays* ya podemos calcular valores derivados y entrenar el sistema. Antes de ello, es conveniente detallar qué bandas (y cuántas) corresponden a cada tipo de onda. Esta información se muestra en la tabla 5.4.

Número de bandas	Rango	Onda representada
3	1 - 3	Delta
5	4 - 8	Theta
6	9 - 14	Alpha
18	15 - 32	Beta
18	33 - 50	Gamma

Cuadro 5.4: Bandas del espectro de onda y su asociación con los tipos de onda.

Con esta información puede pasarse a implementar el algoritmo de entrenamiento del sistema. Para esta labor, y dadas las características proceso a simular, se ha obtado por la implementación de una red neuronal artificial. Primeramente, se estudió si era una solución válida, y si realmente se presentaba una relación clara entre el espectro de onda y los valores inmediatos de atención y meditación, la entrada presentaba variaciones continuas o bruscas.

Para la implementación de la red neuronal artificial se ha empleado la biblioteca FANN (Fast Artificial Neural Network Library). Esta biblioteca proporciona una gran parametrización de la red, permitiendo establecer muy diversas configuraciones de la red. También permite el guardado y recuperación de redes neuronales completas.

Se ha creado una red neuronal multicapa, más concretamente un perceptrón multicapa con 50 neuronas en la capa de entrada, dos capas ocultas del doble de neuronas cada una (100), y una capa de salida con 2 neuronas. Se ha creado totalmente conexa y empleando la función de activación Sigmoidal y el algoritmo de Retropropagación de error. El primer paso es declarar una instancia de `FANN::neural_net` que representa la red neuronal y llamar a su constructor con los parámetros de configuración básicos de la red:

- El ratio de aprendizaje, establecido en 0.9.
- El número de capas (incluyendo la capa de salida y entrada), establecido en 4.
- El número de neuronas en cada una de las capas, establecidos en 50, 100, 100 y 2 respectivamente.

Después se configura la parte de algoritmia. Se establece la función de activación en *Sigmoide* y el algoritmo de entrenamiento en Retropropagación. Una vez hecho esto, sólo queda comenzar el entrenamiento estableciendo un número máximo de iteraciones en las cuales la red debe entrenarse con el error mínimo establecido (en este caso, se ha establecido en 0.0001). Cuando el proceso acabe, la red está entrenada y puede almacenarse en un archivo.

Este archivo es el utilizado por el módulo de entrenamiento para crear la red en el inicio del sistema, de forma que cuando se pide calcular los valores de Atención y Meditación, no debe realizar todo el proceso anterior, que sería inviable; simplemente utiliza la red entrenada para calcular la salida. Esta operación la lleva el último método del submódulo matemático: `calculateValues`.

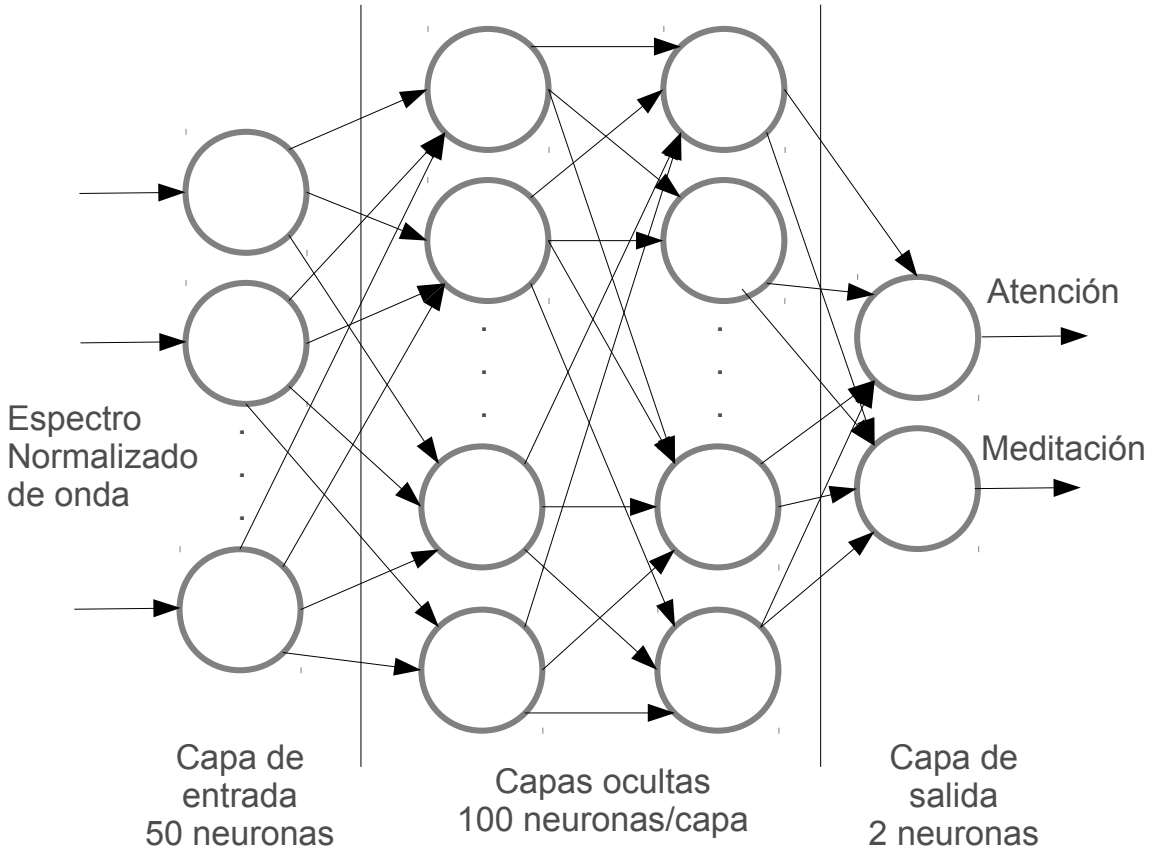


Figura 5.16: Proceso de cálculo de los valores aproximados de Atención y Meditación.

Cuando se llama a `calculateValues`, se pasa como parámetro el espectro de onda normalizado que se tiene en ese instante T y éste es pasado a la red neuronal, que se encuentra encapsulada en la clase `NueralNetwork`. Esta clase tiene dos funciones básicas: crear la red a partir de un archivo cuando se inicia el sistema, y encapsular la llamada *run* de la red neuronal de FANN. A partir de esos valores de entrada, la red neuronal calcula los valores de salida. Dado que tanto los valores de entrada como los de salida estaban normalizados (los valores de salida se normalizaron del rango 1-100 al rango 0-1) se deshace el proceso

multiplicando la salida por 100. Éstos valores ya son los finales y se devuelven al módulo de dispositivo, que los almacenará en el estado del dispositivo correspondiente.

Aunque el error alcanzado por la red para las muestras de entrada es muy bajo, en la práctica, cuando hay otras ondas cerebrales de entrada que presentan picos o fuertes variaciones, el error puede incrementarse bastante. En la figura 5.17 se presenta el error en un escenario real de la red empleada en la plataforma.

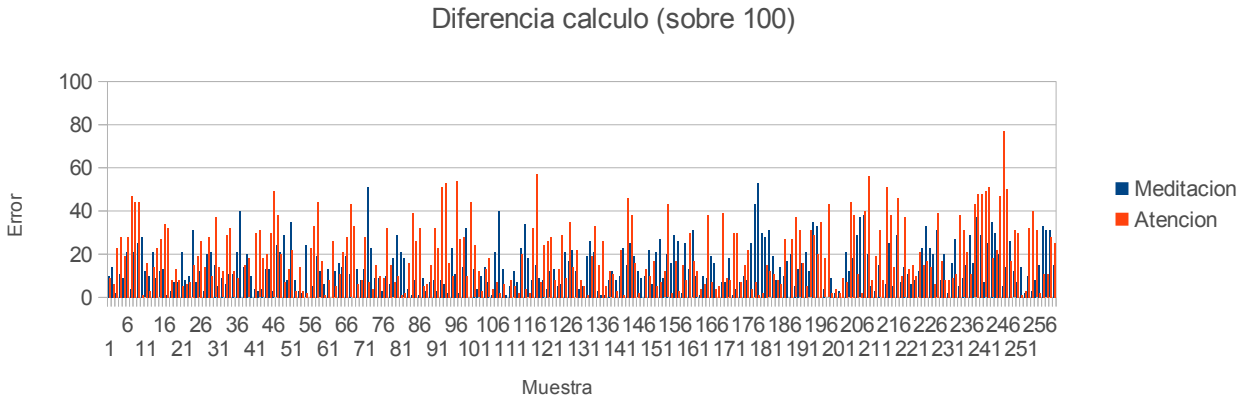


Figura 5.17: Error de la red neuronal en un escenario real.

Como puede apreciarse en la figura, el error se eleva en torno a una diferencia de 20. Concretamente, los datos estadísticos son los siguientes:

- Diferencia media: Para Atención, 19.16; para Meditación, 14.10.
- Coeficiente de variación: Para Atención, 0.76; para Meditación, 0.73.
- Percentiles para Meditación: P25 - 6.75; P50 - 12; P75 - 21.
- Percentiles para Atención: P25 - 7; P50 - 15; P75 - 30.

Dada la poca variación entre los valores deseados y los calculados, y la imposibilidad de aproximar con más precisión el algoritmo utilizado por NeuroSky para el cálculo de valores (es un algoritmo privativo, no se conoce su implementación real), la red neuronal se ha considerado una buena solución.

El último punto de desarrollo establecido es el de proporcionar un punto de acceso único que encapsule toda la funcionalidad del módulo. Para ello se ha empleado el patrón *Facade* o *Fachada*. El patrón Fachada eleva el nivel de abstracción de un determinado sistema para ocultar ciertos detalles de implementación y hacer más sencillo su uso. Se construye una clase entre el cliente y los subsistemas de menos nivel de abstracción, proporcionando una visión unificada del conjunto y, además, se controla el uso de cada componente.

El uso del patrón Fachada proporciona una estructura de diseño como la mostrada en la figura 5.18.

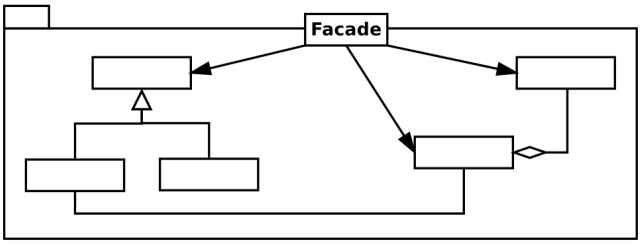


Figura 5.18: Estructura del patrón Fachada. Imagen obtenida de [FA12]

La clase *fachada* de la plataforma es la clase `PlatformMgr`. En su interfaz se exponen todas las operaciones necesarias para acceder a la funcionalidad necesaria de las capas inferiores, sin proporcionar acceso a otras funciones internas. De cara al desarrollador, también mucho más simple y cómodo buscar la funcionalidad deseada en una única clase en lugar de ir buscándola módulo a módulo debiendo conocer los detalles concretos de implementación; a través de la operación de la fachada todo queda oculto y abstraído. Su interfaz presenta las operaciones siguientes:

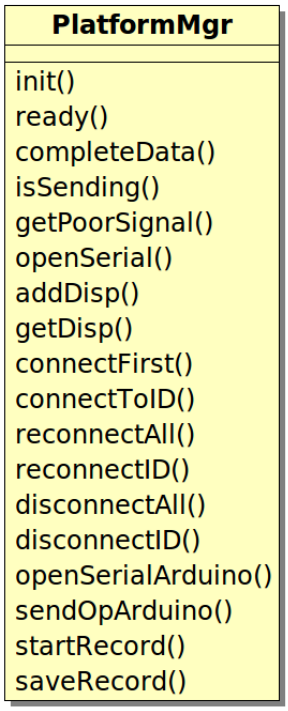


Figura 5.19: Interfaz fachada del sistema.

- Inicio de la plataforma.
- Apertura de comunicación para dispositivo y para microcontrolador.
- Conexión y desconexión de dispositivos, ya sea a dispositivo específico o no.
- Obtención de un dispositivo para acceder al estado del mismo.

- Saber si está enviando datos y con qué intesidad.
- Comandos para dispositivo y microcontrolador.

Como conclusión de la solución, se presentan un diagrama de secuencia y diagrama de clases finales del módulo, correspondientes a las figuras 5.20 y 5.21 respectivamente.

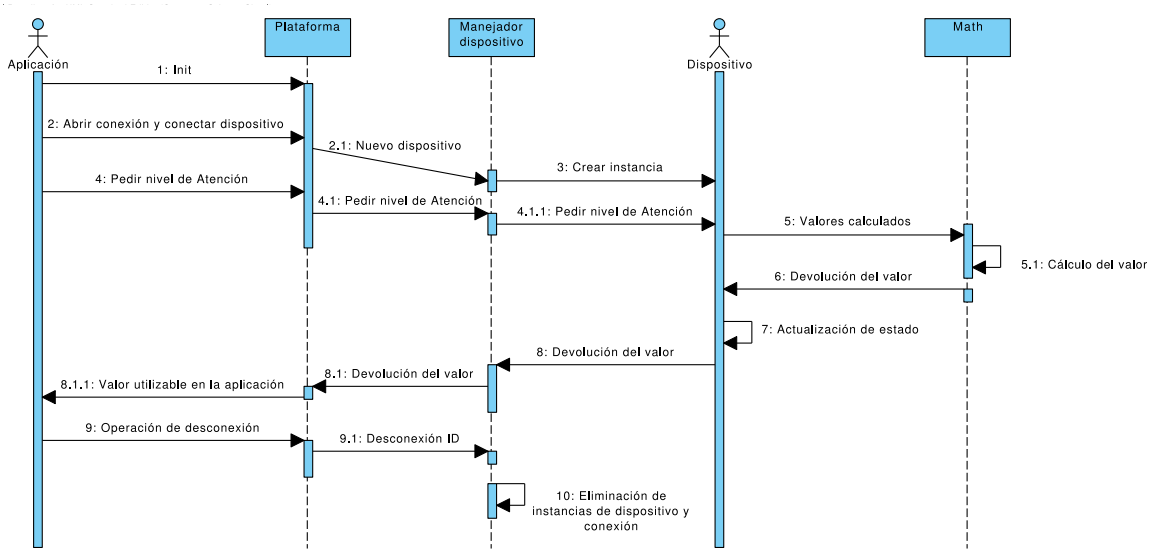


Figura 5.20: Diagrama de secuencia de Dispositivo.

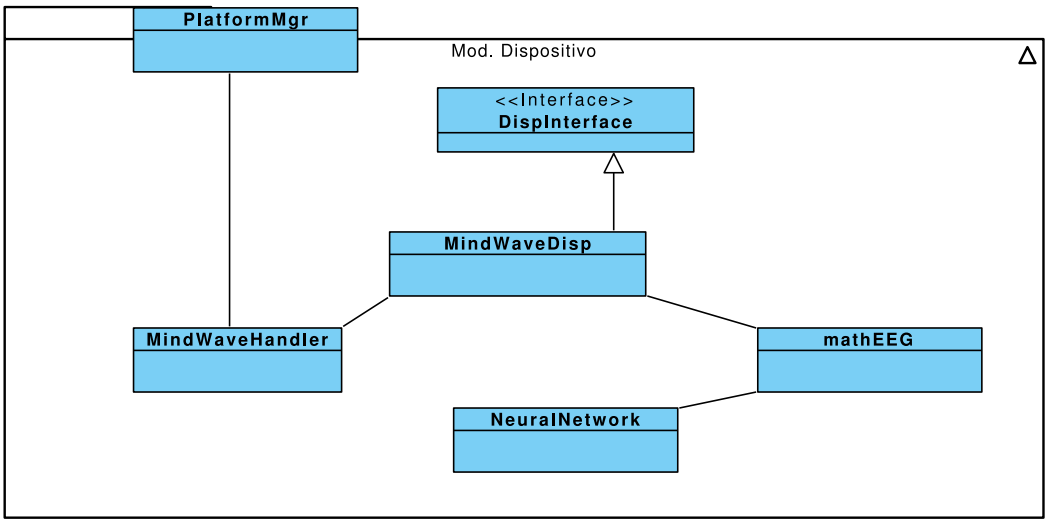


Figura 5.21: Diagrama de clases de Dispositivo.

5.4.3 Ventajas

Gracias al diseño planteado en este módulo, es posible tener un mecanismo de gestión de dispositivos extensible, robusto y eficiente. Es posible añadir nuevos tipos de dispositivos al

sistema, gestionar fácilmente los estados de los dispositivos conectados y tener separadas las operaciones del estado con la utilización de los *handlers* o manejadores.

El módulo matemático proporciona además operaciones muy útiles para el acomodamiento de la onda de entrada y su utilización para estimación de valores. Dado el caso que el dispositivo no envíe correctamente los paquetes de valores inmediatos, tenga problemas de señal o de ajuste al usuario, será capaz de seguir mostrándolos gracias al empleo de la red neuronal (entendiendo que al menos llegan paquetes de onda en bruto).

Por último, gracias al empleo del patrón Fachada se ha conseguido elevar el nivel de abstracción, de forma que oculta los detalles de implementación de las capas inferiores, y hace más sencillo el uso del sistema al eliminar la necesidad de que el usuario tenga que conocer las relaciones existentes entre clases. A la vez, sirve como medio para incrementar la seguridad, ya que se controla el acceso y la forma en que se utiliza el sistema.

5.5 Módulo de trazas

Este módulo se encarga de registrar la actividad cerebral enviada por el dispositivo de EEG y almacenarla en ficheros acorde a un formato determinado. El objetivo es permitir futuros análisis sobre los datos y poder someterlos a diferentes pruebas y tratamientos independientes sin necesidad de que el usuario deba estar presente durante todo el tiempo, que puede durar horas.

Así, cuando se necesita guardar trazas de las señales recibidas, se activa el estado de *debug* y el módulo comienza a almacenar en un fichero la información deseada. Esta información será el espectro de onda obtenido después de descomponer la onda en bruto de entrada con el mecanismo visto en el Módulo de dispositivo, y los valores de 'Atención' y 'Meditación'. Se encarga también de establecer el formato de los datos en el archivo para no tener que ser tratado posteriormente.

Este módulo tendrá además funcionalidad para tratar cualquier error surgido durante el modo *debug* y los posibles errores de corrupción de fichero que ocasione.

Para este módulo, dada su sencillez, no se han implementado submódulos; el módulo se puede considerar el único submódulo existente, el submódulo de grabación. Gráficamente, la descripción funcional del módulo es la mostrada en la figura 5.22.

En lo referente a dependencias y conexiones con otros módulos, éste módulo solo está conectado –y depende de él- con el módulo de dispositivo, del cual necesita los datos, y el cual activa el modo de *debug*.

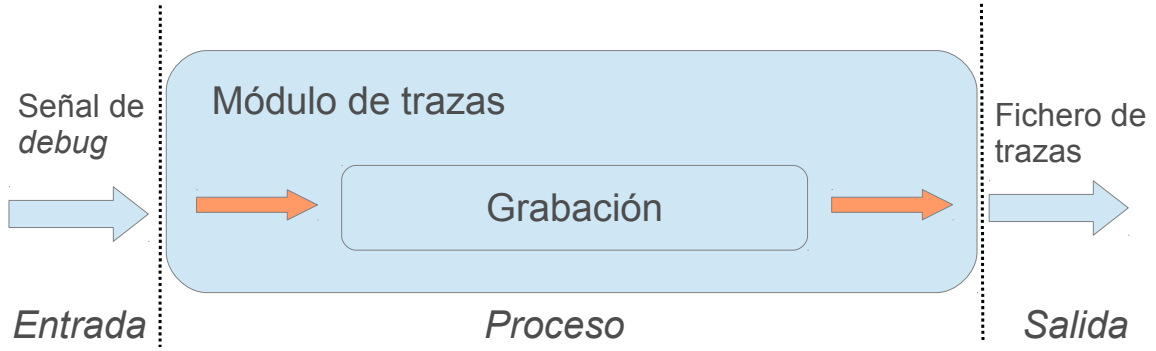


Figura 5.22: Descripción funcional del módulo de trazas.

5.5.1 Problemática

Este módulo supuso pocos problemas dada su sencillez. El principal problema, y prácticamente el único, fue establecer el formato de fichero y su estructura. La información había de guardarse de forma sencilla y eficiente, sin producir una bajada de rendimiento en modo debug.

Este problema se resolvió a la par que se diseñó el submódulo de entrenamiento, pues la biblioteca empleada en esa labor también imponía una estructura sobre las muestras de entrenamiento y las muestras de pruebas sobre ficheros de texto plano. Se decidió tomar esa estructura para las trazas, pues era simple, fácil de tratar por el desarrollador y eficiente.

Un segundo problema que surgió, en términos de eficiencia, fue decidir si los datos serían almacenados en memoria hasta que se cancelase el modo *debug* o se irían escribiendo en fichero según se recibían. Para muestras pequeñas, no supondría un problema de memoria mantenerlas en una estructura en memoria principal hasta el momento de volcado a disco; sin embargo, podría llegar a suponerlo en algún momento que se decidiese almacenar una cantidad muy grande de muestras. Dadas las características de los computadores de hoy día no debería suponer un problema, por lo que los datos se almacenan en una estructura de datos que sea capaz de recorrerse linealmente de forma eficiente al igual que haga eficientemente las inserciones al final de la estructura, y posteriormente se pasan a fichero.

5.5.2 Solución

Para realizar una implementación que salve los problemas expuestos en el apartado anterior, ha de desarrollarse un módulo capaz de obtener la información deseada del estado del dispositivo y almacenarla en un fichero de forma simple y eficiente en términos de cómputo y memoria.

Durante el uso de la aplicación, el usuario podrá guardar la actividad cerebral que está captando el dispositivo y utilizando la aplicación. Entonces activará la opción de *debug* mediante una opción en la interfaz gráfica, se enviará la señal al manejador de dispositivo y el

módulo se activará. Cada vez que llegue una actualización de datos, el módulo los recogerá, siendo éstos el espectro de onda normalizado y descompuesto en bandas de frecuencia, y los valores de 'Meditación' y 'Atención'. Junto a la onda de entrada se ha decidido incorporar también esos valores para que el mismo fichero pueda servir también para entrenamiento de la red.

Para ello, se establecen una serie de características o puntos de desarrollo:

- Se deben poder obtener los datos del estado del dispositivo, o ser enviados por el módulo de dispositivo.
- Se establece una estructura en el fichero para organizar los datos.
- Se ha de controlar el proceso de grabación y la cantidad de muestras salvadas.
- Ha de implementar mecanismos de seguridad para que en caso de error se cierre el fichero con los últimos datos válidos.

El módulo ha de tener acceso a los datos o recibirlos del módulo de dispositivo. Puesto que diferentes dispositivos pueden guardar los datos de forma diferente, se ha decidido que sea el módulo de dispositivo quien se los envíe a través de una llamada a método. Estando activo el modo *debug*, en cada actualización de datos el manejador de dispositivo llamará al módulo de trazas enviándole la información con el método `addRecord`.

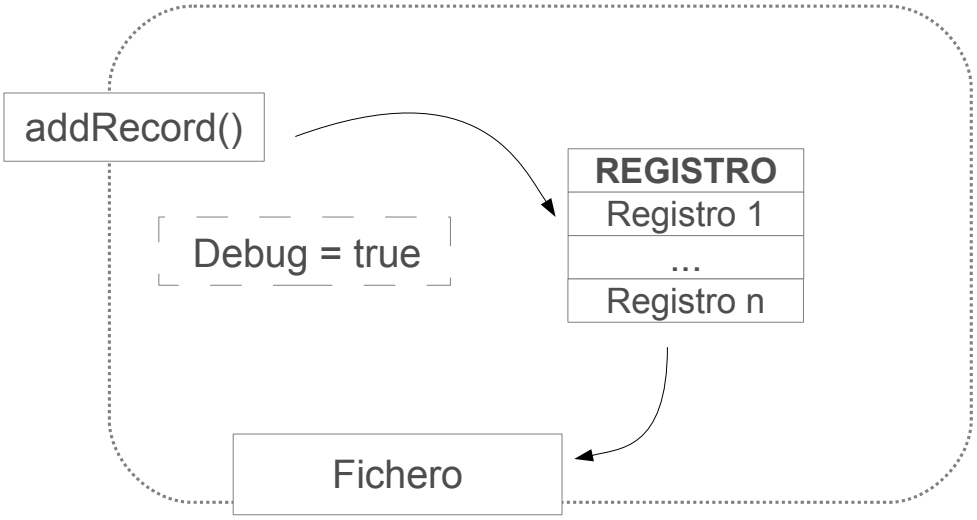


Figura 5.23: Envío de registros al módulo de trazas.

Del mismo modo que el módulo de dispositivo se encarga de enviar la información al módulo de trazas, prepara los datos si fuese necesario para que cumplan un determinado formato que simplifique la operación. El método `addRecord` impone una serie de parámetros que han de respetarse. Los valores de entrada que se establecen son:

```
1    void addRecord(double* powerIn, int inLength, int medOut, int attOut
        );
```

Listado 5.8: Método de adición de trazas al registro de ondas.

- El primer parámetro `powerIn`, con tipado *puntero a double*, es una *array* de valores en punto flotante en doble precisión que codifican el espectro de onda normalizado.
- El parámetro `inLength` de tipo entero indica el tamaño del array de valores en doble precisión.
- Los valores `medOut` y `attOut` a los valores correspondientes al espectro actual de Meditación y Atención, respectivamente.

Una vez recibidos los datos, se guardan en estructuras de datos y se incrementan los contadores apropiados. El primer parámetro, el espectro de onda se guardará en un *vector* que contiene punteros a *double*. Se elige *vector* por tener una complejidad lineal *n* en el recorrido secuencial y por tener complejidad *1* en la inserción al final de la estructura, ya que siempre guarda un puntero *apuntando* al final de la misma. Debido a que en la inserción en el vector sigue una política de copia, y almacenamos un puntero, que perderá la referencia al salir del método, se procede de otra forma:

1. Se inserta en *vector* un nuevo puntero a *double*, que genera una nueva dirección de memoria.
2. Se copia el contenido del puntero de entrada en el nuevo puntero insertado en la estructura, como se muestra en el listado 5.9.

```
1    _inRecords.push_back(new double[inLength]);
2    copy(powerIn, powerIn + inLength, _inRecords.back());
```

Listado 5.9: Copia de punteros en *vector*.

El caso de los valores de Meditación y Atención es mucho más simple. En otro *vector* son insertados ambos en un par de valores (correspondiente a la estructura *pair* de STL). En este caso la copia del par no supone problema, al no ser referencias, sino valores en double precisión. Éstos valores, que van en un rango de 1 a 100 se normalizan al rango [0, 1] del mismo modo que está normalizado el espectro.

Para el contador de muestras se ha empleado el tamaño del *vector*, cualquiera de los dos pertenecientes al módulo pues contienen el mismo número de registros. De esta forma se evita el tener otra variable más que ha de incrementarse en cada inserción y reiniciarse al finalizar la operación.

Para la función de volcado a fichero se ha seguido el mismo enfoque. El manejador establece el modo *debug* a `false`, y se envía una señal al módulo, que mediante la función `saveToFile` guardará todos los registros en el fichero, siguiendo la estructura mostrada en la tabla 5.5.

Valores	Significado
200 50 2	El primer valor indica el número de muestras que contiene el registro, el segundo valor indica el número de campos del espectro normalizado, y el tercer valor indica el número de campos para atención y meditación.
0.131515 0.0139569 0.0405525 0.0620602 0.0503825 0.0825269 0.0628824 0.0530446 0.0376424 0.0713687 0.0451423 ...	Contiene los 50 valores del espectro de onda normalizado completo.
0.34 0.56	Contiene los valores de meditación y atención respectivamente.

Cuadro 5.5: Estructura del fichero de trazas.

La primera fila solo aparece una vez al comienzo del fichero; las líneas 2 y 3 se repetirán en ese orden tantas veces como registros haya, indicado por el primer valor de la primera fila. Para ello, se obtiene el tamaño del registro y las longitudes de los mismos; después se procede a iterar sobre las estructuras y guardar primero el espectro de onda y, a continuación en la línea inferior, los valores de Atención y Meditación. Una vez escrito todo, se cierra el fichero y se vacían las estructuras. Dado que una de las estructuras guarda punteros, esta memoria no es liberada de forma automática al vaciar la estructura; por tanto:

1. Se itera sobre el *vector* liberando explícitamente la memoria mediante la operación `free()`.
2. Se vacía la estructura *vector* mediante la operación `clear()` que implementan.

Como conclusión de la solución, se presentan un diagrama de secuencia y diagrama de clases finales del módulo, correspondientes a las figuras 5.24 y 5.25 respectivamente.

5.5.3 Ventajas

Gracias a este módulo, es posible proporcionar funcionalidad al desarrollador para almacenar estados del dispositivo que más tarde pueda analizar, aplicar diferentes procesos o algoritmos de entrenamiento, de una forma sencilla y eficiente. Además, gracias a este enfoque, no es necesario que el sistema esté ejecutándose y el usuario deba tener el dispositivo conectado durante todo el proceso, ya que puede ser tedioso debido al tiempo que requiere.

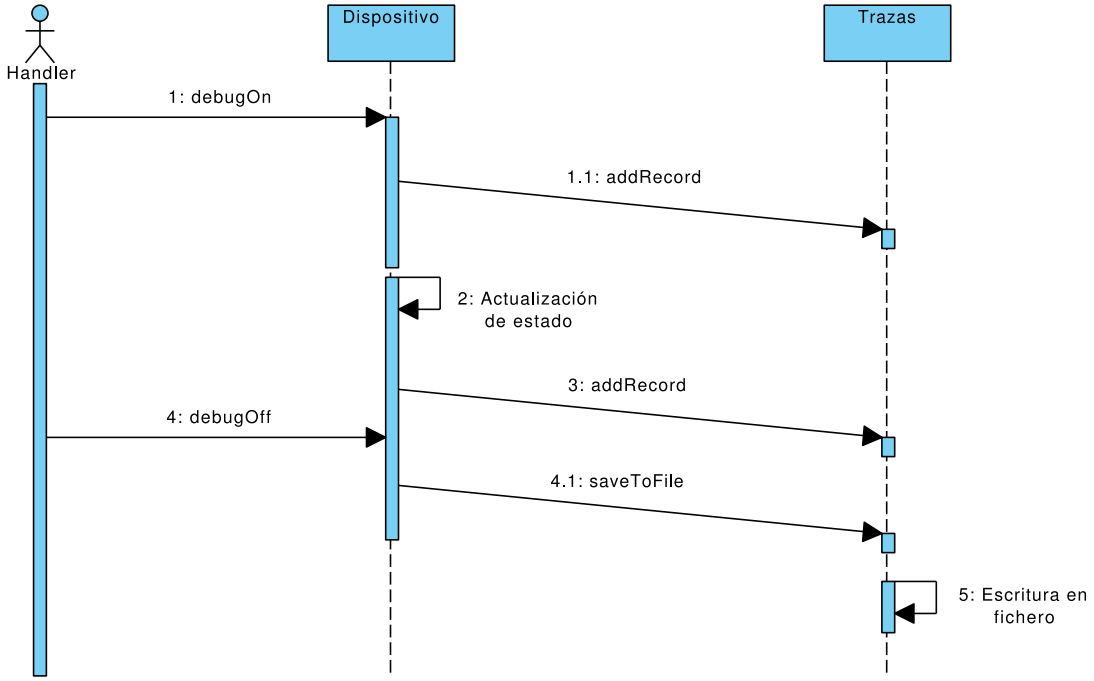


Figura 5.24: Diagrama de secuencia de trazas.

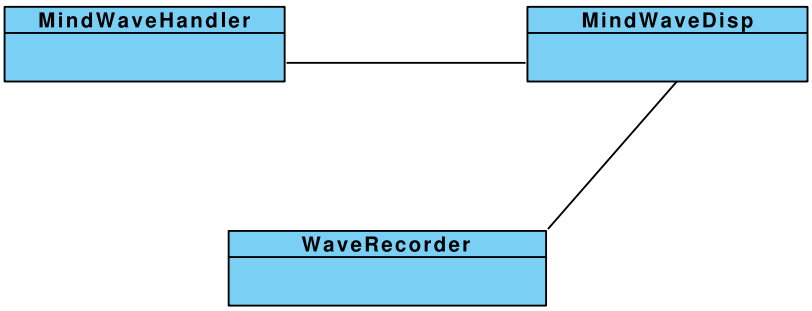


Figura 5.25: Diagrama de clases de trazas.

También se reduce el acoplamiento entre módulos al no tener el módulo de trazas que romper el encapsulamiento del estado del dispositivo en el módulo homónimo, ni tener que formatear valores si no cumplen su interfaz.

5.6 Módulo de representación gráfica

El modulo de representación gráfica es el encargado de mostrar visualmente la información de la plataforma y la retroalimentación del usuario. Cuando los dispositivos estén conectados mostrará del estado de los mismos la información de la conexión, a saber el *status* y la intesidad de la conexión; mostrará también los valores calculados de Atención y Meditación que son los que utilizará la aplicación.

Adicionalmente, si el usuario lo desea, podrá representar la onda en bruto de entrada tanto para observar en tiempo real la fluctuación como para labores de depuración (detectar picos, fluctuación errónea,...).

Tendrá también funcionalidad para ocultar y volver a mostrar estos valores si se desease, todo fácilmente a partir de un menú gráfico. Para que esto funcione, deberá ser capaz de detectar los eventos producidos en el menú por el ratón o por teclas.

Para desempeñar estas funciones se han diseñado los siguientes submódulos:

- Un submódulo encargado de representar gráficamente los valores deseados en *Widgets* personalizables que pueden ocultarse/mostrarse.
- Un segundo submódulo encargado de capturar los eventos del menú y realizar las operaciones correspondientes.

Gráficamente, la descripción funcional del módulo es la mostrada en la figura 5.26.

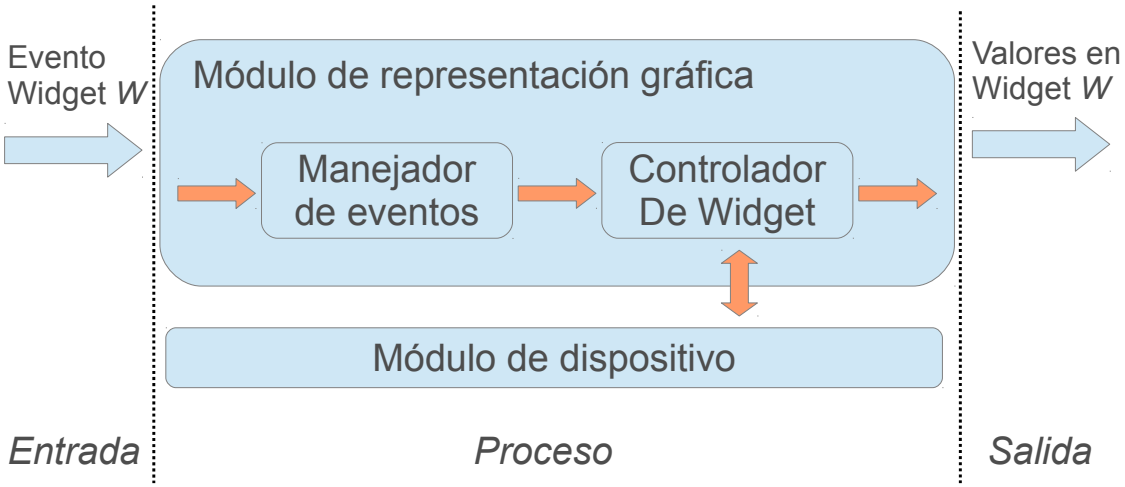


Figura 5.26: Descripción funcional del módulo de representación gráfica.

Se recibe un evento correspondiente a un determinado widget y se activa o desactiva su visualización según corresponda. Si es para visualización, el controlador obtiene los valores del estado del dispositivo y activa el widget con los valores en la interfaz gráfica.

En cuanto a la conexiones y dependencias del módulo, tiene únicamente dependencia de entrada con el módulo de dispositivo, pues necesita los valores del estado del dispositivo para representarlos.

5.6.1 Problemática

El diseño de este módulo ha conllevado ciertos problemas en términos de eficiencia. Cuantos más polígonos tenga la escena, más sofisticada sea la interfaz gráfica y los widgets, y más cálculos se realicen internamente, mayor será el tiempo que se tardará en procesarse cada frame, lo que puede conllevar que se produzca una pérdida de rendimiento en la visualización final de los datos a pesar de haber conseguido una solución eficiente en los niveles inferiores. Por esto la parte gráfica no debe arruinar la implementación del resto de módulos.

Unos de los problemas que se presentan es la elección de un buen subsistema de interfaces gráficas y accesorios sobre el que fundamentar el módulo de widgets, que sea eficiente y extensible, aportando facilidades de configuración y programación para el desarrollador. Además debe poder incorporarse fácilmente al motor de renderizado empleado. Además debe ser dinámico y poder capturar eventos, ser cambiado de posición, etc., a diferencia de lo que ocurre con los *overlays*.

El mayor problema lo ha supuesto la representación de la onda en bruto. Dado que el motor de renderizado que se ha empleado es OGRE 3D, y éste no tiene soporte para representación 2D al estilo de los *surfaces* de OpenGL, se ha estudiado una representación 3D de la onda, incrustándola en el widget mediante un render a textura. Esta representación podría suponer un problema al tener que mantener cientos de puntos, representados en un *Scene Node* o nodo de escena, y posteriormente generar una textura. Para evitar sobrecarga se ha optado por reducir el número de puntos que se representan en la onda. Aunque la solución supone una pequeña bajada de rendimiento, ha sido la mejor solución disponible.

Un último error que se presentó en este módulo fue la actualización de la textura en cada frame. Mediante una opción de «autorenderizado» de las texturas de OGRE 3D, la textura debería actualizar el contenido de forma automática. Sin embargo, ésto no ocurría, de modo que hubo que actualizar manualmente la textura en cada frame e incorporarla al widget.

5.6.2 Solución

En este punto se discute la solución propuesta a los problemas anteriores y su implementación. Se establece para ello una serie de características deseables y medidas para resolverlos y obtener la solución deseado.

El módulo recibirá un evento desde la interfaz gráfica, ocasionado por una acción llevada a cabo en un menú que muestra las opciones disponibles (hacer click sobre el botón correspondiente al widget deseado), lo capturará y decidirá que acción realizar. Si el widget deseado se está mostrando, entonces lo oculta. Si por el contrario está oculto, entonces crea el widget, establece los valores correspondientes y lo muestra. En el caso de la onda en bruto deberá generar además un *render a textura* e incrustarla en la ventana del widget.

Por otro lado, deberá generar primero esta onda en el espacio 3D, recuperando cierta cantidad de muestras del estado del dispositivo y generando con ellas la línea 3D correspondiente.

Para ello, se establecen una serie de características o puntos de desarrollo:

1. Deben poder crearse fácilmente widgets que muestren la información deseada del dispositivo.
2. Éstos pueden mostrarse u ocultarse a conveniencia desde un menú en la interfaz de usuario. Este menú debe poder capturar eventos de teclado y/o ratón para que el usuario interaccione con él.
3. Debe generarse una representación 3D de la onda de entrada en forma de línea.
4. La visualización de la línea debe estar oculta en el espacio 3D y verse en una textura en un widget propio.

Para llevar a cabo los puntos de desarrollo 1 y 2, se han diseñado una serie de clases (*wrappers*) que utilizan por debajo el sistema de interfaces gráficas CEGUI. Se utilizan dos clases: `Widget`, encargada de crear los widget necesarios, gestionarlos, asignarles su valor y asignar la textura; y `WidgetEventHandler`, encargada de capturar y tratar los eventos registrados en los widgets. A continuación se detalla la implementación explicando a la par el motor de CEGUI.

Para poder utilizar los elementos que CEGUI proporciona, primero debe inicializarse su motor de gestión de Widgets. Está orientado al uso de *scripts* y hace uso del patrón *Singleton* para implementar los diferentes subsistemas. Los más importantes son:

- **CEGUI::System**: Gestiona los parámetros y componentes de la biblioteca.
- **CEGUI::WindowManager**: Se encarga de la creación y gestión de los *windows* (ventanas) de CEGUI.
- **CEGUI::SchemeManager**: Gestiona los diferentes esquemas que utilizará la interfaz gráfica.
- **CEGUI::FontManager**: Gestiona los diferentes tipos de fuente que utilizará la interfaz gráfica.

Estos subsistemas deben ser inicializados previamente para poder utilizar cualquier funcionalidad, además de asociarse a un sistema de *render*. Esta acción se lleva a cabo en el constructor de la clase Widget; cuando se inicie, iniciará consigo el renderer de OGRE 3D para que los widgets sean visualizables y después iniciará los subsistemas necesarios.

```
1   Widget::Widget
2   ()
3   {
4     _renderer = &CEGUI::OgreRenderer::bootstrapSystem();

6     //Se crean los grupos por defecto para cegui
7     CEGUI::Scheme::setDefaultResourceGroup("Schemes");
8     CEGUI::Imageset::setDefaultResourceGroup("Imagesets");
9     CEGUI::Font::setDefaultResourceGroup("Fonts");
10    CEGUI::WindowManager::setDefaultResourceGroup("Layouts");
11    CEGUI::WidgetLookManager::setDefaultResourceGroup("LookNFeel");
12    ...
13  }
```

Listado 5.10: Inicialización de Widget.

A continuación se establecen los valores por defecto para el esquema, la fuente, el cursos y el *layout* por defecto. Los layouts imponen la estructura, elementos y configuración de los widgets o ventanas. Estos layouts son los que utilizaremos en los métodos de creación de los Widgets.

Así, para crear nuestro Widget de estado del dispositivo, invocamos el método `addStatusDisplay` de la clase `Widget` y procedemos cargando primero el layout correspondiente a través del `CEGUI::WindowManager`, añadiéndolo al *window* raíz para que pase a visualizarse, y estableciendo a `true` la variable de activación del widget.

```
1   CEGUI::Window* window = CEGUI::WindowManager::getSingleton().
        loadWindowLayout("status.layout");
2   _sheet->addChildWindow(window);

4   _activeStatus = true;
5   }
```

Listado 5.11: Carga de layout de estado y visualización.

Estos ficheros de *layout* son en realidad archivos XML con una estructura determinada donde se declaran los elementos de la ventana y sus propiedades. Un cuadro de texto se declara en el layout como sigue en la figura 5.12.

En él se ha declarado un *window* de tipo *StaticText*, con las propiedades que definen su área, el texto que muestra, el tamaño máximo y la alineación del texto. Todos estos valores

```
1   <Window Name="rootStatus/status/dispOneText" Type="TaharezLook/
       StaticText">
2          <Property Name="UnifiedAreaRect" Value="
              {{0,8},{0,11},{1,-139},{1,-184}}" />
3          <Property Name="Text" Value="Disp 1" />
4          <Property Name="UnifiedMaxSize" Value="{{1,0},{1,0}}" />
5          <Property Name="HorzFormatting" Value="CentreAligned" />
6   </Window>
```

Listado 5.12: Declaración de cuadro de texto en layout.

son modificables, haciendo al widget personalizable. Mencionar además que las medidas de los window se establecen en un sistema de dimensión unificado, con valores entre 0 y 1 que indican la proporción del alto y ancho de la ventana o *window* padre al que pertenecen. En una resolución de 640 puntos de ancho, 1 sería 640p y 0.5 sería 320p.

Para el caso del widget que muestra la onda ha de realizarse un proceso más complejo, que involucra la creación de texturas en OGRE 3D desde una cámara, la creación de un *ImageSet* de CEGUI para ella y su inclusión en algún *Window* de tipo *StaticImage* declarado en el layout del Widget.

Para crearemos primero una cámara de OGRE 3D que «mire» hacia la representación 3D de las ondas. A continuación, creamos manualmente una textura de OGRE con el tamaño requerido para el widget; para que pueda albergar el objetivo de una cámara la textura se declara del tipo `Ogre::TU_RENDERTARGET`.

El siguiente paso es crear el *viewport* para la cámara de la textura y establecer las opciones `setClearEveryFrame` y `setAutoUpdated` a verdadero. Una vez hecha la parte de OGRE 3D, se realizar la parte de CEGUI. Primero se crea la textura de CEGUI a partir de la de OGRE y luego se crea un *ImageSet* para ella, ya que CEGUI trata las imágenes como un *array* de imágenes.

El último paso es cargar el layout correspondiente, extraer el elemento *StaticImage* que albergará la textura, y asociarlos. Al añadir el layout al window raíz, se visualizará el widget con el render a textura. En el listado 5.13 se muestra el código principal del proceso descrito.

Estos widgets deben activarse desde un menú en la interfaz gráfica al hacer click sobre la opción deseada; es necesario capturar esos eventos de ratón sobre la ventana, tanto su posición como la acción del ratón. Dado que CEGUI no incluye gestión de entrada se deben *inyectar* desde OIS (biblioteca de OGRE para entrada).

Cualquier evento se controlará en la clase `WidgetEventHandler`. Esta clase contiene las operaciones necesarias para la inyección de eventos de OIS y el tratamiento de los eventos de CEGUI. Para el primer caso, incluye 3 operaciones, que han de ser llamadas desde los

```
1     Ogre::Camera* _camTex = Ogre::Root::getSingletonPtr()->
          getSceneManager("SceneManager")->createCamera("BackCamera");
2     ...
3     //Ogre::TexturePtr tex
4     tex = Ogre::Root::getSingletonPtr()->getTextureManager()->
          createManual(..., Ogre::TU_RENDERTARGET);
5     // Ogre::RenderTexture* rtex;
6     rtex = tex->getBuffer()->getRenderTarget();

8     Ogre::Viewport* v = rtex->addViewport(_camTex);
9     v->setClearEveryFrame(true);
10    rtex->setAutoUpdated(true);

12    CEGUI::Texture& guiTex = _renderer->createTexture(tex);
13    CEGUI::Imageset& imageSet = CEGUI::ImagesetManager::getSingleton().
          create("RTTImageset", guiTex);

15    CEGUI::Window* ex1 = CEGUI::WindowManager::getSingleton().
          loadWindowLayout("render.layout");
16    CEGUI::Window* RTTWindow = CEGUI::WindowManager::getSingleton().
          getWindow("rootRender/CamWin/RTTWindow");
17    RTTWindow->setProperty("Image", CEGUI::PropertyHelper::imageToString
          (&imageSet.getImage("RTTImage")));

19    _sheet->addChildWindow(ex1);
```

Listado 5.13: Widget con render a textura.

respectivos métodos de *callback* de OIS:

1. `injectMouseMove`: Inyecta la posición del cursor en pantalla.
2. `injectMouseButtonUp`: Inyecta la liberación de un botón en el ratón.
3. `injectMouseButtonDown`: Inyecta la pulsación de un botón en el ratón.

Estos métodos son invocados en los métodos de *callback* de OIS pasando los argumentos de OIS como argumentos de CEGUI. Las otras 3 operaciones pertenecientes a eventos en widgets de CEGUI son:

1. `valuesDisplayClicked`: Evento de pulsación para el widget de valores del dispositivo.
2. `statusDisplayClicked`: Evento de pulsación para el widget de estado del dispositivo.
3. `wavesDebugClicked`: Evento de pulsación para el widget de debug de la onda de entrada.

Estas tres últimas operaciones trabajan de forma diferente a las primeras y previamente han de ser asociadas a algún *window* de CEGUI mediante la operación `subscribeEvent`, que recibe como parámetros el tipo de evento a capturar y el método que ha de ejecutarse en caso de producirse.

Para el último elemento gráfico, la representación de la onda 3D, se ha empleado la solución *DynamicLineDrawing*[4] de OGRE 3D. Mediante esta clase es posible dibujar líneas 3D en OGRE a partir de los puntos dados. El procedimiento seguido es:

1. Se instancia un objeto de la clase `Dynamic Lines`.
2. Se recuperan X valores de la onda *Raw* de entrada.
3. Se añaden esos valores como puntos en el espacio 3D al objeto `Dynamic Lines`.
4. Se actualiza el renderizado.
5. Si los valores cambian en cada frame, se actualiza el valor de los puntos.

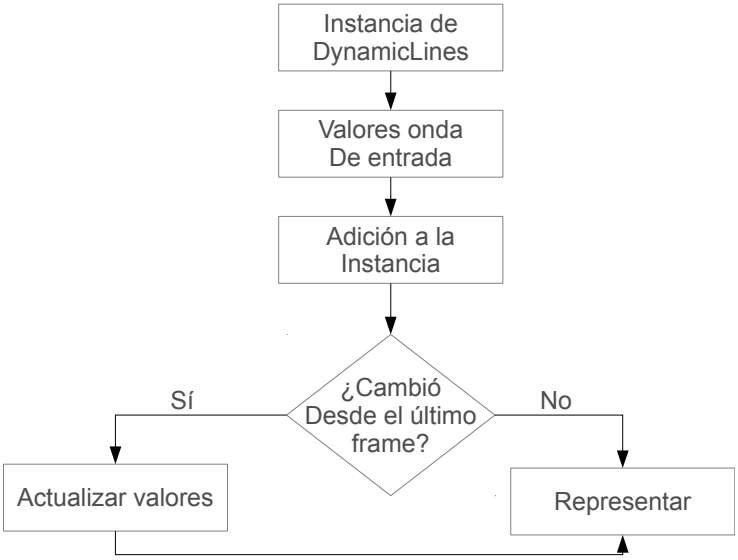


Figura 5.27: Diagrama de flujo de Dynamic Lines.

Como conclusión de la solución, se presentan un diagrama de clases final del módulo, correspondiente a la figura 5.28.

5.6.3 Ventajas

Mediante la solución obtenida, el sistema puede mostrar visualmente toda la información relativa a los dispositivos conectados y la actividad cerebral del usuario. Gracias al enfoque dado a esta representación en forma de Widgets, el usuario decide qué información mostrar en determinado momento y que información ocultar, además de poder posicionarla sobre la interfaz libremente.

Exceptuando la representación de la onda en bruto, que es dependiente de OGRE 3D, los widgets restantes pueden ser utilizables en otros motores de renderizado, como OpenGl,

[4]http://www.ogre3d.org/tikiwiki/tiki-index.php?page=DynamicLineDrawing

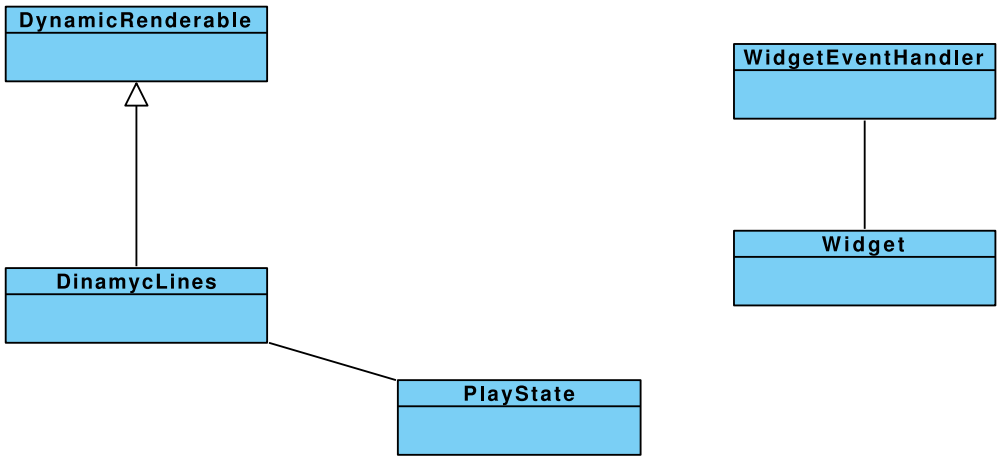


Figura 5.28: Diagrama de clases de representación gráfica.

gracias a trabajar por debajo con CEGUI (Crazy Eddie's GUI). Esto beneficia la portabilidad y reutilización del módulo.

Además, dado que CEGUI es un sistema de interfaces gráficas diseñada para videojuegos, se obtiene un rendimiento muy bueno y gran personalización.

5.7 Módulo de control del microcontrolador

Este módulo es el encargado de controlar el microcontrolador Arduino, el cuál se encarga de la programación del circuito electrónico construido para la parte hardware. Cuando en el demostrador se determine quien va ganando, la pelotita deberá moverse en el sentido del otro jugador; la lógica del juego enviará esa información a la plataforma y ese movimiento deberá reflejarse físicamente. Para ello, se envía la información al módulo del microcontrolador, que enviará la operación correspondiente al Arduino y éste, acorde al programa que tenga cargado, la transformará en señales eléctricas al motor y demás circuitería de control.

Además, también se encarga de interpretar las señales eléctricas que lleguen al Arduino, como puede ser la pulsación de un final de carrera. Cuando llegue un comando del Arduino, lo transformará en las órdenes correspondientes para la lógica de la aplicación.

Para desempeñar esta labor, no se han diseñado submódulos; se tiene el módulo de control únicamente debido a que realiza operaciones sencillas. Sin embargo, podría considerarse como submódulo el programa cargado en el Arduino, que convierte las señales eléctricas en operaciones y operaciones en señales eléctricas.

Gráficamente, la descripción funcional del módulo es la mostrada en la figura 5.29.

Dada una operación O (avanzar a la izquierda, parar, posicionar al centro,...), ésta se enviará al módulo de control, que la codificará y la enviará al Arduino a través del puerto

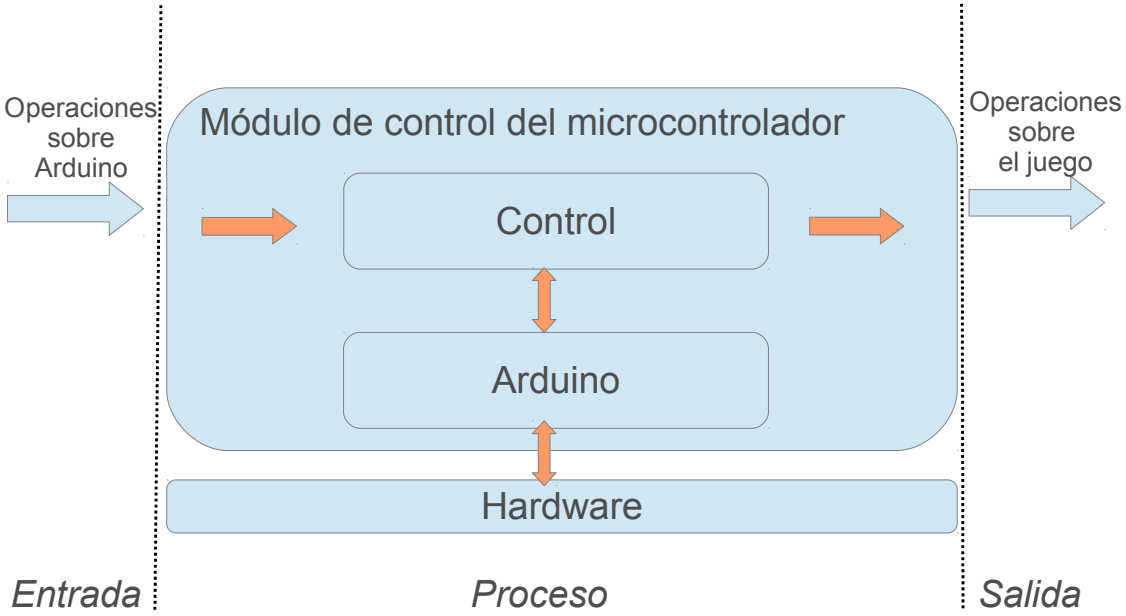


Figura 5.29: Descripción funcional del módulo de control del microcontrolador.

serie. El Arduino la leerá del puerto y ejecutará la acción pertinente y, en caso de producirse alguna señal de entrada, la enviará a través del puerto. Si el controlador recibe de vuelta una señal, la transmitirá a la plataforma.

En cuanto a las relaciones y dependencias con otros módulos, este módulo depende del módulo de dispositivo, del que recibe los comando a ejecutar (a través del *manager* de la plataforma). No se relaciona ni depende de más módulos.

5.7.1 Problemática

En cuanto a la problemática de este módulo, se encontraron mayores dificultados en la parte hardware que en la parte software. En la parte software, crear un módulo capaz de comunicar con el microcontrolador Arduino y enviar y recibir datos no supuso mayor complejidad que estudiar el manual y los ejemplos de Arduino. Sin embargo, implementar la parte hardware no fue trivial para nada.

Por una parte se necesitaba, ante todo, un motor y un mecanismo que, movido por el motor, pudiese desplazar horizontalmente alguna plataforma o base, sobre la que colocar un imán para mover la pelota metálica. Además, ese desplazamiento debería ser discretizado en posiciones mediante algún sistema de posicionamiento preciso. La base debería desplazarse en cualquiera de los dos sentidos, indicando la posición concreta y volver al centro cuando alcanzase uno de los dos extremos. Al llegar al extremo enviaría también una señal de control. Y todo ello controlado mediante el Arduino.

El mayor problema fue montar el sistema de posicionamiento preciso mediante un circuito integrado compuesto de un LED infrarrojo y un fotosensor, capaz de «contar» líneas blancas

sobre fondo negro. Se requería de un circuito independiente para su funcionamiento, alimentado a través del Arduino, mediante el cual no funcionaba. El circuito del sensor retasó mucho el desarrollo del proyecto y se precisó ayuda externa.

Por otro lado, el motor giraba más rápido de lo requerido y no era posible acoplar un sistema reductor mediante engranaje o polea, por lo que se decidió hacer la reducción en la tensión del motor.

Además, en todo momento la tensión del circuito y su montaje debía ser el correcto para evitar que algún componente se sobrecargase o se produjesen cortocircuitos que dañasen la implementación.

5.7.2 Solución

El objetivo es implementar una solución con la funcionalidad descrita, salvando los problemas encontrados y descritos en el punto anterior. Se enviará al módulo un código de operación que deberá codificar y enviar a través del puerto serie al que esté conectado el Arduino. Éste, a su vez, deberá interpretarlo y activar o desactivar las señales eléctricas necesarias en el circuito implementado. A su vez, si se produce en el Arduino alguna entrada del circuito, la codificará y enviará por el puerto serie de vuelta al módulo de control. Esta información se utilizará en la lógica de juego para sincronización de la parte física y la virtual.

Para ello, se establecen una serie de características o puntos de desarrollo:

- Debe abstraerse de la conexión al puerto y la forma de enviar y recibir información.
- Debe implementar códigos de operación para las diferentes acciones posibles.
- Ha de implementarse un circuito electrónico para el demostrador y ser controlado mediante Arduino.

Para llevar a cabo los primeros puntos de desarrollo se ha diseñado la clase `ArduinoMgr`. Esta clase se encarga, por un lado, de controlar el puerto serie al que va conectado el Arduino de una forma semejante a como vimos en el módulo de comunicaciones, pero en una versión mucho más reducida. En este caso, no es necesario mantener lectura asíncrona, callbacks ni buffers, pues el tráfico entre el Arduino y la plataforma será relativamente reducido (unas 2-3 veces por segundo).

Nuevamente se utiliza la biblioteca *Boost* para la entrada/salida en el puerto, por lo que esta clase tiene como atributos privados dos objetos de las clases `IO Service` y `Serial Port` respectivamente; se ha añadido también una variable para condición de error. Cuando se instancia el *manager*, se abre la comunicación en el puerto estableciendo el *nombre de dispositivo* y la *frecuencia*. Para el caso del Arduino la frecuencia es 9600 *bauds*. Establecida la comunicación, el siguiente paso es poder enviar comandos al Arduino.

Para el envío de comandos se utiliza la escritura en el puerto de caracteres con signo (`signed char` en C++). Ésto nos permite guardar información de hasta 1 byte, tanto con valor positivo como negativo. Se ha decidido codificar las operaciones como enteros con signo, por lo que se pueden codificar un total de 255 valores u operaciones en el rango de -128 a 127. Se lleva a cabo mediante la operación `writeCommand` de la clase `ArduinoMgr`. Sin embargo, como se comentó en el módulo de dispositivo (véase 5.4.2), esta funcionalidad no está disponible directamente al usuario, que debe realizar la acción por medio de la *fachada* de la plataforma, mediante la llamada a `sendOpArduino`. Por último, se escribe en el puerto mediante la función `asio::write` de *Boost*.

El código lo recibirá e interpretará el programa cargado en el Arduino. Pero antes de entrar a describirlo, es necesario describir primero el circuito montado, para poder entender todas las entradas y salidas conectadas en el microcontrolador. Se describe a continuación.

Para construir el demostrador al estilo *BrainBall*[5], se requiere un mecanismo capaz de moverse horizontalmente de forma precisa en ambas direcciones. Como parte motriz se ha empleado el carro de impresión de una fotocopiadora, que ya incluía el motor también. El carro se mueve a lo largo de una barra de hierro en ambas direcciones mediante una polea dentada, movida por el eje del motor. Este motor funciona con 9-12V, aunque a 12V la velocidad de giro es superior a la deseada.

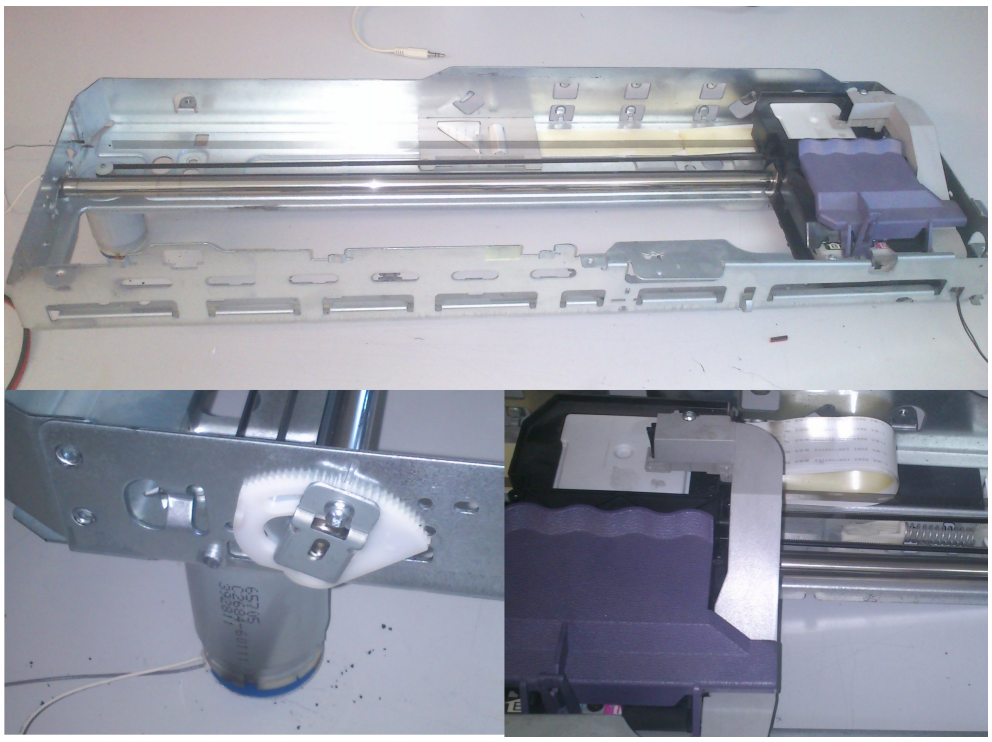


Figura 5.30: Carro de impresión con motor.

Una vez obtenido esto, se monta primeramente el circuito necesario para moverlo e inver-

[5] `http://www.youtube.com/watch?v=oBeGv_x4Tbs`

tir el sentido de la corriente. Para activar y desactivar el circuito, así como para invertir el sentido, se han empleado relés que permiten el paso de corriente por una patilla u otra de salida en función de si se activa o no la bobina. Lo ideal es alimentar las bobinas y el motor con el Arduino, pero dado que Arduino da tensión de 5V y se necesitan al menos 9V para el motor, se ha diseñado el circuito como sigue:

1. El motor se alimentará con una fuente de alimentación de 12V.
2. Con 12V se alimentarán los relés de 12V que dejan pasar la corriente al motor y alternan su sentido.
3. Para controlar cuando deben activarse los relés del motor, se utilizan otros dos relés adicionales y señales del Arduino. Los relés son de 6V para que puedan activarse con la señal de 5V del Arduino.
4. Por último, para controlar la activación mediantes señal del Arduino se utilizan dos transistores, de forma que cuando reciben señal en la base (patilla del Arduino) dejan pasar la corriente por el relé de 6V que activa o desactiva uno de 12V, con su correspodiente acción en el motor.

El circuito queda como la figura 5.31.

Dado que con 12V el motor gira demasiado rápido, y no es posible acoplar reductores físicos, se han empleado un potenciómetro para reducir el voltaje que llega al motor. El potenciómetro tiene tres patillas, como se muestra en la figura 5.32.

En la patilla de la derecha se conectan la entrada de 12V y una patilla del relé que activa/desactiva la corriente; en la patilla del medio se conecta la otra patilla de salida del relé, que irá al relé de cambio del sentido de la corriente; la patilla de la izquierda se conecta al negativo de la batería. De esta forma, sea cual sea el sentido de la corriente, la tensión llega reducida siempre.

En este punto hay que tener en cuenta que el carro llegue a un extremo y deba pararse el motor en lugar de seguir recibiendo corriente. Para ello, se han situado en los extremos del carro dos pulsadores con resistencia en *pull-down*, de modo que cuando se pulsa en la entrada hay 5V y cuando no se pulsa hay 0V. Se conectan las patillas del pulsador siguiendo el siguiente esquema de la figura 5.33.

La conexión de este circuito con el Arduino, para su posterior programación y asociación de los pines, es la siguiente:

1. Los pines de `+5V` y `Gnd` se conectan a los mismos en la placa de prototipado.
2. Los pines digitales 2 y 3 se conectan a la base de los transistores que activan los relés de 6V (uno a uno).
3. Los pines digitales 11 y 12 se conectan a la entrada de cada uno de los pulsadores.

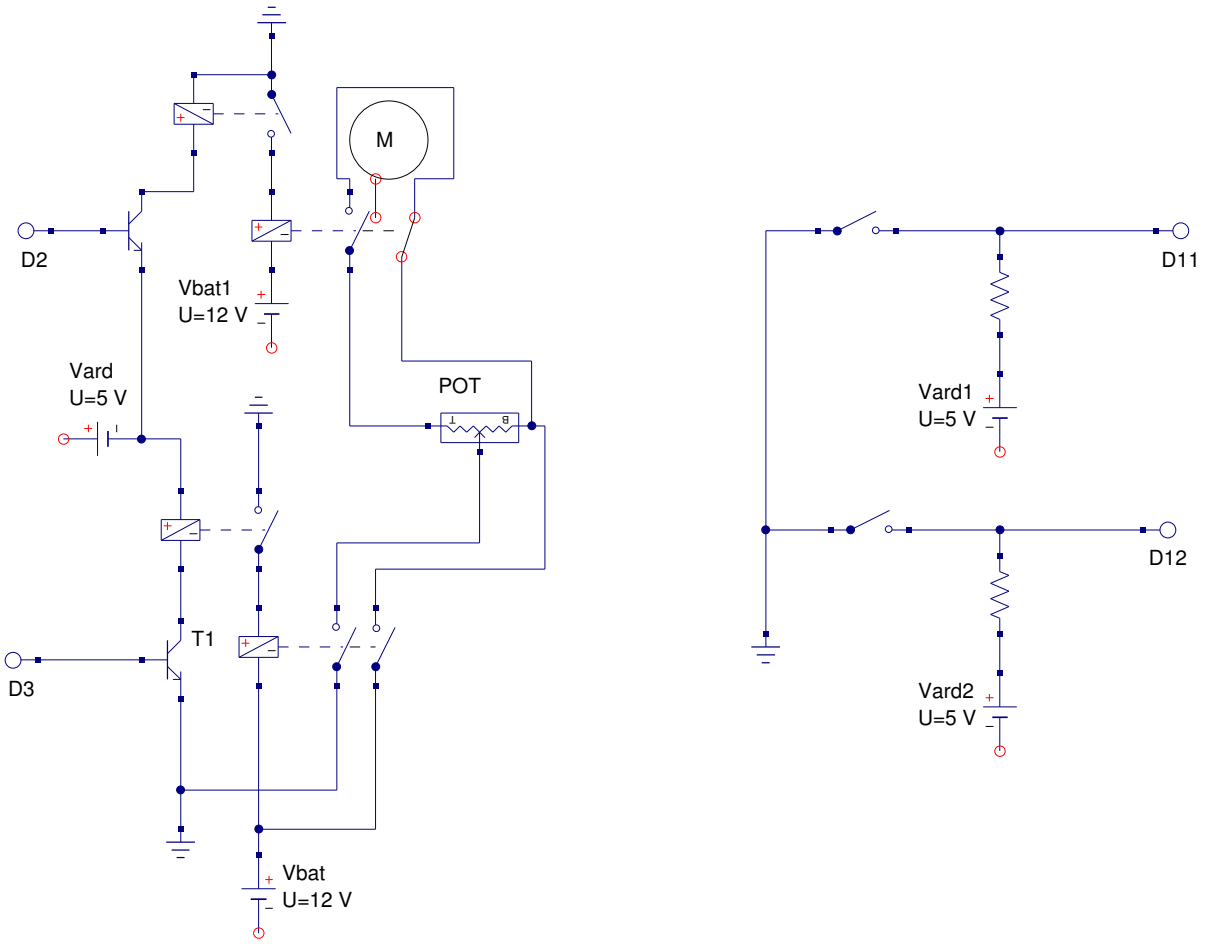


Figura 5.31: Circuito electrónico de controlador del demostrador.

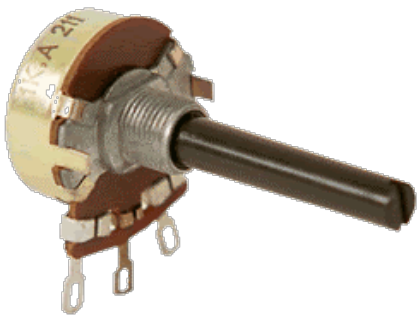


Figura 5.32: Potenciómetro.

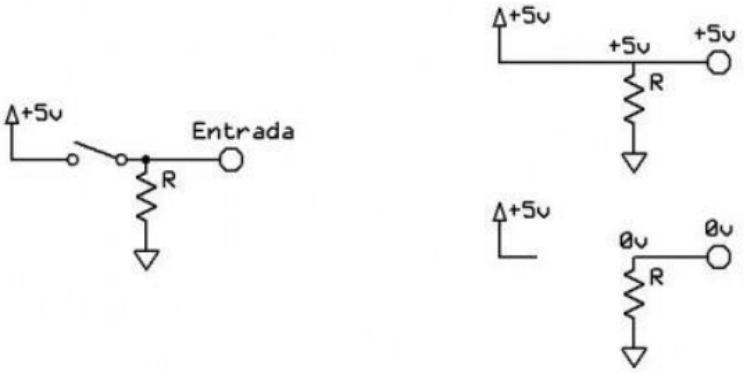


Figura 5.33: Pulsador con resistencia *pull-down*. El triángulo invertido representa la Tierra (Gnd).

Llegados a este punto, el carro puede arrancar y parar, puede moverse en un sentido y el contrario, y detecta cuando llega al final; aún falta el sistema para hacer el desplazamiento preciso. Para ello se ha empleado un circuito integrado CNY70, compuesto de un LED infrarrojo y un fotosensor. Cuando el fotosensor recibe la luz infrarroja deja pasar la corriente de su circuito; cuando no la recibe, la resistencia en tan alta que no deja circular corriente. Para recibir la luz infrarroja reflejada debe situarse encima del sensor una superficie de color blanco. Sobre otro color no reflejará y no dejará circular corriente. Así, se sitúa en un lateral una tira de franjas negras y blancas, cerca de las cuáles pasará el sensor al moverse el carro (suficientemente cerca para capturar el reflejo). Cada vez que pase sobre la franja blanca, se activará y el Arduino recibirá un 1 en la entrada; si las franjas están suficientemente cerca, se puede conseguir movimiento muy preciso. La forma de conectar el circuito integrado con el Arduino se muestra en la figura 5.34.

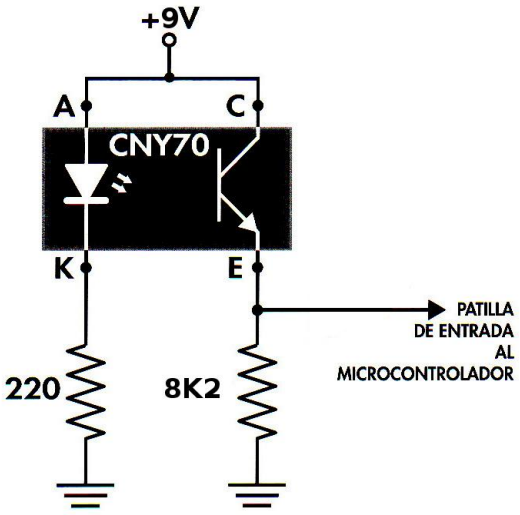


Figura 5.34: Circuito integrado CNY70 en Arduino.

El circuito integrado cuenta con 4 patillas, de las cuales 2 son para el LED infrarrojo y las otras 2 son para el fotosensor. Las patillas A y C se alimentan con una pila de 9V. La otra patilla del LED va a una resistencia de 220 y a Tierra (Gnd). La salida E del fotosensor es diferente. A la salida se coloca un divisor de tensión, de forma que al pasar la corriente, una parte circule por la resistencia de 8K2 (8200 Ω) y la otra vaya a la entrada del microcontrolador. Si la resistencia fuese mayor, mayor tensión iría al microcontrolador y menor a Tierra; si la resistencia fuese menor, mayor tensión iría a Tierra y menos al microcontrolador. De esta forma se consigue que con una entrada de 9V al Arduino lleguen solo 5-6V, lo necesario para detectar un 1 lógico en la entrada.

El pin digital empleado para el sensor es el pin 7.

En el programa de control del Arduino, escrito en el lenguaje homónimo, se diferencian cuatro bloques principales: bloque de definición de variables y constantes, bloque de inicio de las señales de entrada y salida del controlador, bucle principal y bloque de eventos en el puerto serie.

En el primer bloque se han definido las constantes pertenecientes a los pines de entrada/-salida descritos anteriormente, y una serie de variables necesarias, como un valor entero para la posición actual del carro. Las constantes y variables se definen en Arduino igual que en C, tal y como muestra el listado 5.14.

```
1  const int onOff = 3;
2  const int rigthLeft = 2;
3  const int stopRigth = 12;
4  const int stopLeft = 11;
5  const int counter = 7;
```

Listado 5.14: Constantes definidas en Arduino.

Estas variables son empleadas en el bloque de inicialización, compuesto por el método `setup()`, donde mediante la función `pinMode` se establece si el pin es de entrada (`INPUT`) o de salida (`OUTPUT`). También se inicializa el puerto serie mediante un objeto `Serial` e indicando la frecuencia del mismo; para el Arduino el valor es 9600 *bauds*.

El bloque de eventos en el puerto serie está definido por el método de *callback* `serialEvent()`. Este método es invocado por Arduino cuando detecta un evento en el puerto serie. En él se leen los bytes disponibles en el puerto (pueden ser varios), y con ellos se establece el código de operación a realizar en la siguiente iteración el bucle principal; el código de operación será la posición a la que debe dirigirse el carro, o si debe pararse.

En el bloque principal se lleva a cabo la lectura de las señales de entrada, correspondientes a los pines 7, 11 y 12, que son el sensor CNY70 y los fines de carrera respectivamente, y el movimiento del carro a la posición deseada. Primero, se leen los valores de estos sensores. Si algunos de los fines de carrera manda señal alta (1 lógico), se anula el valor de operación

actual y se establece en parada, si no, se procede con el código leído en el bloque de eventos. Después se lee la señal de entrada del sensor. Para saber cuando atraviesa una franja y permitir la cuenta, se utiliza el contraste entre blanco y negro (1 y 0 lógicos respectivamente) y una variable de estado del sensor:

- Si el estado es 0 y se lee 0, no ocurre nada.
- Si el estado es 0 y se lee 1, se establece el estado a 1.
- Si el estado es 1 y se lee 0, hemos atravesado una raya blanca. Se establece el estado a 0 y se incrementa o decrementa el valor de la posición del carro según el sentido en el cual se esté desplazando.
- Si el estado es 1 y se lee 1, no ocurre nada.

Tras este proceso, comprobamos la posición actual con la posición destino. Pudiendo ocurrir:

- La posición actual es igual a la posición destino, por lo que el código de operación se establece en parada.
- La posición actual no es igual a la posición destino, por lo que se ordenará mover el carro en un sentido u otro en función de si la posición actual es menor o es mayor que la posición destino.

El último paso es enviar retroalimentación a la plataforma a través del puerto serie. En Arduino, la escritura en el puerto se lleva a cabo mediante la llamada `println` del objeto `Serial`. Por ejemplo: `Serial.println(FIN_IZQUIERDA)`.

5.7.3 Ventajas

La solución obtenida aporta grandes ventajas al sistema, ya que permite crear respuestas físicas asociadas a la actividad cerebral del usuario. Esto crea efectos sorprendentes, que simulan que el usuario mueve objetos con el poder de su «mente».

Gracias a la implementación a través de Arduino, se obtiene una solución muy extensible dada la reprogramabilidad del microcontrolador, la posibilidad de conectarlo con casi cualquier componente y su potencia y eficiencia en la programación y control de los circuitos. Además, se puede montar el circuito de modo que el microcontrolador pueda desconectarse sin problemas y «pincharlo» en otro circuito (previa reprogramación), lo que facilita la reutilización.

Además, el módulo solución es independiente del dispositivo utilizado, por lo que no supone restricciones adicionales.

5.8 Integración con OGRE 3D. Estados de juego.

En esta sección se detalla cómo se ha integrado la plataforma para su ejecución en OGRE 3D, así como la estructura de clases empleada para basar la ejecución del programa en estados, basados en la idea de estados de juego de los videojuegos convencionales, que permite su división en etapas diferenciadas tanto por funcionamiento como por interacción con el usuario.

Desde un punto de vista general pueden diferenciarse los siguientes estados de juego:

- **Presentación**: Se muestra al usuario aspectos generales como la temática del juego o como utilizarlo.
- **Menú principal**: El usuario elige diferentes modos de juego y opciones.
- **Juego**: Se realiza la interacción con la aplicación.
- **Finalización**: Se muestra la información sobre la partida finalizada.

Esta clasificación es muy general y abstracta, por lo que los estados dependen mucho de la aplicación concreta. En nuestro caso concreto, se han implementado dos estados:

- **Inicio**: Se lleva a cabo el arranque de la plataforma y la conexión de los dispositivos y el Arduino en los puertos serie. Si todo es correcto, se pasa al estado siguiente.
- **Juego**: Se lleva a cabo toda la interacción con la plataforma, en forma de juego virtual BrainBall igual que el demostrador físico. Se muestra toda la información visual de la plataforma y se permite la interacción con los widgets descritos en la sección 5.6. Al finalizar el juego cierra todas las conexiones y apaga el sistema.

Para implementar esta solución en OGRE 3D se ha utilizado la planteada en [FA12]. Se define la clase abstracta `GameState`, que contiene una serie de funciones que deberán implementar los estados específicos, que heredarán de esta clase. Las funciones abstractas que define se clasifican en cuatro categorías:

1. **Gestión básica del estado**: Define las acciones a realizar al entrar, salir, pausar o reanudar el estado.
2. **Gestión básica de tratamiento de eventos**: Define qué hacer cuando se produce un evento de teclado o ratón.
3. **Gestión básica de eventos antes y después del renderizado**: Operaciones del renderizado de OGRE que se llevan a cabo en cada estado específico.
4. **Gestión básica de transiciones**: Operaciones para realizar las transiciones de estado.

No obstante, con esta clase únicamente no es posible implementar esta funcionalidad. La figura 5.35 muestra la relación con el resto de clases necesarias, con tres especializaciones ejemplo. `GameState` se relaciona con la clase `GameManager`, que es la clase encargada de

gestionar los diferentes estados, sus transiciones, y capturar los eventos de ratón, teclado y renderizado que propagará al estado de juego concreto que se esté ejecutando en ese momento.

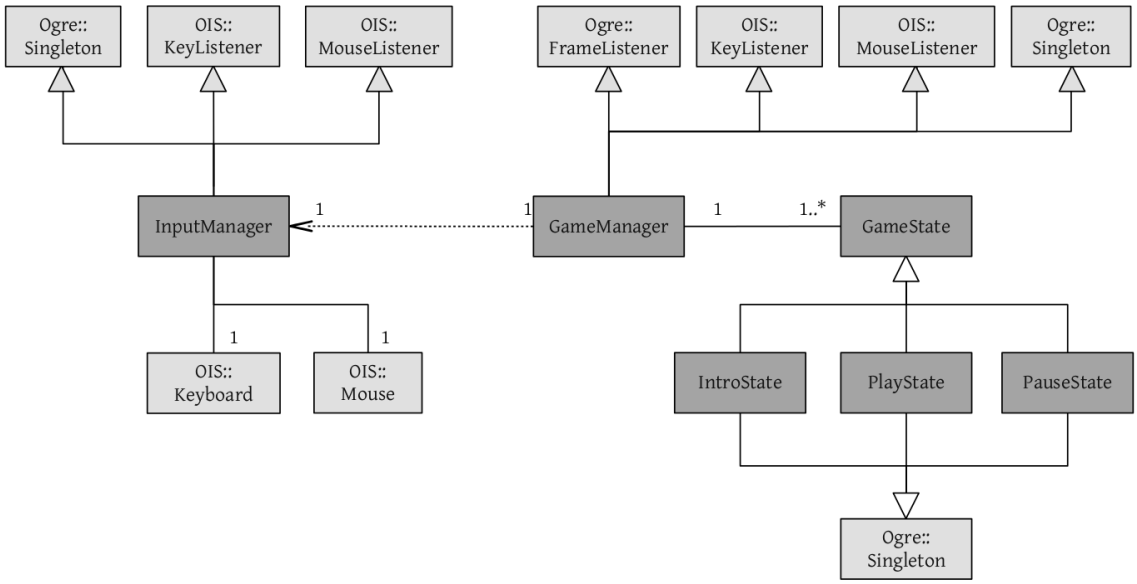


Figura 5.35: Diagrama de clases del esquema de gestión de estados de juego en OGRE. Imagen obtenida de [FA12].

La clase `InputManager` sirve como interfaz para aquellas entidades que quieran procesar eventos de entrada. Para ello mantiene operaciones para añadir y eliminar *listeners* de dos tipos: *OIS::KeyListener* y *OIS::MouseListener* (a través de herencia). El primero permanece a la escucha de los eventos de teclado, mientras que el segundo hace lo mismo con los eventos de ratón. La clase además implementa el patrón *Singleton* para que exista una única instancia.

Por su parte, la clase `GameManager` representa la entidad principal de gestión de estados. También implementa el patrón *Singleton* para asegurar una única instancia. Hereda también de *OIS::KeyListener* y *OIS::MouseListener*. El motivo es hacer posible el registro con `InputManager` como *listener* de teclado y ratón.

Por otra parte, mantiene una estructura de datos de tipo pila para gestionar las transiciones de estados. Si queremos realizar la transición de un estado a otro (sólo posible si el estado origen está en la cima de la pila) simplemente ejecutaremos el método de salida del estado origen, lo desapilaremos, apilaremos el estado destino y ejecutaremos su método de entrada.

Otro aspecto importante del diseño es que permite delegar el tratamiento de todos los tipos de evento (teclado, ratón y renderizado) al estado activo (cima de la pila), que contiene la implementación concreta de cada función.

Una vez definida la estructura de clases de estados de juego, se puede detallar la implementación concreta y la forma en que se ha integrado la plataforma.

Se ha implementado un estado de juego denominado `IntroState`, con el objetivo de iniciar la plataforma y conectar los receptores de dispositivo y el Arduino, así como registrar los códigos específicos de cada dispositivo. En este estado no hay renderizado, y la única entrada de teclado/ratón que se permite es la tecla ESC para cerrar el programa en caso de error. No obstante, se utiliza en bucle de renderizado para realizar la conexión de los dispositivos en el puerto: en cada iteración del bucle se intenta conectar a los puertos USB de los receptores inalámbricos y del Arduino hasta que ambos hayan devuelto estado válido de conexión. Para realizar esta interacción con la plataforma, la clase `IntroState` se comunica con `PlatformMgr`, la clase fachada. Una vez establecidas las comunicaciones, se pasa al siguiente estado.

La clase `PlayState` implementa la lógica de juego del demostrador y su representación gráfica, así como la representación de widgets con los valores de los dispositivos. También se asocia con la clase `BrainBallGame`, que controla la lógica del juego BrainBall. La comunicación con la plataforma se realiza siempre a través de la fachada `PlatformMgr`, excepto al módulo de Widgets, al que puede llamar directamente. En el método de entrada al estado crea la cámara y viewport necesarios para renderizar la escena, crea la escena 3D y posteriormente pide a la plataforma que realice la conexión a dispositivos mediante ID concreto (registrado en `IntroState`) y que active los Widgets de valores de dispositivo y estado de los mismos.

En cada actualización del bucle de renderizado, de los 60 *fps* establecidos, en `PlayState` se realizan las siguientes operaciones:

1. Se recuperan de la plataforma los valores de Atención y Meditación mediante los métodos `getMeditationLevel()` y `getAtentionLevel()`, que devuelve el valor eSense si se recibe del dispositivo o el valor estimado en caso de no recibirlo; en cualquier caso esta elección es transparente al usuario, de forma que siempre recibe los valores a no ser que se produzca una desconexión del dispositivo.
2. Se calcula, mediante una media de los dos valores, que jugador obtiene ventaja en el juego, y se actualiza la lógica de juego con esta información (clase `BrainBallGame`).
3. A continuación se pasan los valores de dispositivo y estado al gestor de Widgets, que los mostrará si los widgets correspondientes están activos.
4. Por último, se actualiza la representación 3D de la onda de entrada mediante la clase `DynamicLines` comentada en la sección 5.6.2. No es necesario informar de esta actualización al gestor de widgets, ya que el render a textura se actualiza automáticamente.

Cuando se desee terminar el juego, se pulsará la tecla ESC, dando lugar a la ejecución del método de salida del estado y posteriormente apagando el sistema.

Capítulo 6

Evolución y costes

En el presente capítulo se describe la evolución del proyecto en base a la metodología empleada (véase 4.1), los hitos conseguidos y su complejidad. Mientras que en el capítulo anterior se analizaba la solución final y sólo en algunos casos se mencionaba brevemente la existencia de varias versiones, en este capítulo se detallarán las iteraciones realizadas hasta conseguir la versión final del sistema.

6.1 Evolución del proyecto

En esta sección se mostrará la evolución del proyecto en base a la metodología de desarrollo. Antes de comenzar la etapa de desarrollo se llevó a cabo una etapa de estudio de estado del arte, tanto los proyectos existentes como los diferentes dispositivos hardware, para establecer cuál sería el empleado y qué posibilidades ofrecía. Después se realizó un análisis de requisitos inicial, estableciendo la funcionalidad que habría de tener la plataforma, su arquitectura general y las características del demostrador. A continuación se adquirieron los recursos hardware y software necesarios para comenzar el desarrollo, se realizó el diseño preliminar de los módulos y se comenzó su elaboración en iteraciones. Éstas se describen en los siguientes apartados.

6.1.1 Concepto del software

Cada vez más se oye hablar de nuevas tecnologías dispuestas a cambiar la forma en la que interactuamos con los computadores, o la forma en que estos «desaparecen» en nuestro ambiente. Así, podemos mencionar 'Google Glass' [Goo12], unas gafas con un microcomputador y una pantalla óptica incorporada en los cristales que permitirá «llevar puesto» el computador. A través de las gafas podrá mostrarse información de rutas, hacer fotografías, buscar información de ciudades,...

Del mismo modo comienzan a aparecer proyectos que utilizan una nueva tecnología, tan sorprendente o más como pueda ser la nueva interfaz de realidad aumentada de Google: las interfaces de interacción cerebral o de Electroencefalografía. Dado su potencial y el sorprendente efecto de controlar las aplicaciones mediante pensamiento, se decide orientar este Proyecto Fin de Carrera hacia este ámbito.

Tras comprobar las limitaciones que ofrecen todavía las soluciones disponibles (problemas comentados en 1.2 y las dificultades que presentan de cara al desarrollador de aplicaciones, se decide hacer una plataforma capaz de facilitar la integración de nuevos dispositivos, de añadir soporte en sistemas no privativos, además de abstraer al desarrollador de aplicaciones de multitud de detalles de implementación de los dispositivos.

Tras fijar las características del proyecto, se comenzó la etapa de análisis de requisitos.

6.1.2 Análisis preliminar de requisitos

En esta etapa del proyecto se realizó el estudio del estado del arte, gracias al cual se pudieron extraer los requisitos iniciales y, posteriormente, los objetivos concretos.

El primer paso fue analizar los diferentes dispositivos que había disponibles, sus características, funciones, limitaciones y coste. Además debían poder utilizarse bajo sistemas libres e incluir o poder hacerse bibliotecas para los mismos. Una vez escogido el dispositivo se compraron dos unidades para poder llevar a cabo el demostrador.

Con las instancias del dispositivo se realizaron una serie de pruebas para comprobar la eficiencia del mismo, la funcionalidad que proporcionaba y si había algún problema a la hora de utilizarlo. Se observó que presentaba problemas en la emisión de algunos paquetes cuando disminuía la intensidad de la conexión, por lo se planteó añadir un sistema de estimación de esos valores. Para ello se revisaron conceptos matemáticos necesarios para su consecución así como diferentes técnicas de Inteligencia Artificial.

Además se planteó incluir un demostrador físico, por lo que se estudiaron diferentes opciones de microcontroladores y componentes electrónicos para montarlo. Se decidió utilizar Arduino por la facilidad de desarrollo, la amplia comunidad de usuarios y por ser un proyecto de hardware libre.

También en esta etapa se determinó la utilización de una interfaz gráfica 3D y la creación de Widgets para ella en la cual mostrar la información necesaria. Desde el primer momento se eligió OGRE 3D como motor gráfico; los widgets se plantearon primero como *overlays* y más tarde se decidió utilizar un motor de widgets (CEGUI).

Teniendo determinado el concepto del software, y tras estas decisiones, se establecieron una serie de requisitos básicos que debían ser satisfechos:

1. Debe construirse mediante software libre, bibliotecas, frameworks, etc.
2. Debe ser capaz de soportar varios dispositivos y de forma simultánea.
3. Debe ser capaz de obtener en todo momento valores, recuperándose en el caso de pérdida de datos puntual.
4. Debe poder representarla gráficamente e incluir un conjunto mínimo de widgets para ello.

5. Debe incluir un demostrador físico, estilo BrainBall.

Después de los requisitos iniciales se pasó al diseño general del sistema.

6.1.3 Diseño general

Se planteó desde el primer momento un diseño modular y extensible. Para ello, se diseñaron una serie de módulos (Capítulo 5) de forma que cumpliesen funciones específicas y que fuesen lo más independientes. De esta forma, si fuese necesario cambiar la biblioteca sobre la que se apoya cierto módulo, no fuese necesario reescribir el resto de módulos. Además, pensando en la extensibilidad del sistema y la integración de diferentes dispositivos, se ha realizado una programación «contra interfaz», de forma que cualquier nuevo componente del sistema que respete la interfaz establecida pueda ser añadido y utilizado aunque tenga diferente implementación.

Se ha seguido un enfoque multi-hilo y la aplicación de diversos patrones de diseño (tales como *Facade*, *Singleton* o *Adapter (Wrapper)*), así como guías de estilo de código C++ para obtener una mejor solución y un código legible y bien estructurado.

6.1.4 Iteraciones

En este apartado se muestran las iteraciones realizadas en el desarrollo, describiendo las partes implementadas, rediseñadas y los hitos conseguidos en cada una de ellas. En el Capítulo 5 se detallan las clases y módulos mencionados a continuación.

Iteración 1

En esta primera iteración se comenzó creando la estructura básica de directorios y el archivo *Makefile* para la compilación automática de los ficheros y la generación del ejecutable. Dado que el proyecto contendría ficheros de muy diversa naturaleza, se orientó la estructura de ficheros en torno a ello (ficheros de cabecera, ficheros multimedia, ...). En esta primera versión del Makefile se añadieron las opciones del *linker* básicas para los componentes detallados a continuación.

Se creó un programa básico en OGRE 3D para capturar las pulsaciones de teclado y una interfaz mínima con *overlays* sobre la que mostrar resultados. Con ésto, se desarrolló e integró la primera versión del módulo de comunicaciones. En el primer momento, se disponía de una versión simple, compuesta por la clase `SimpleSerial` únicamente, cuya función era esperar en el puerto a leer k bytes y devolverlos a la clase pertinente y poder escribir bytes en el puerto. Esta clase no operaba en otro hilo externo y bloqueaba el puerto hasta que hubiese datos; si no había datos suficientes, no «rompía» el bloqueo.

Junto a esta clase se desarrolló el primer manejador, con funcionalidad para pedir k bytes a la clase anterior, almacenarlos en una estructura, e irlos enviando byte a byte al autómata

de conversión o parser. Cuando recibía paquetes válidos, a partir de una función de *callback*, los mostraba por consola, dado que en este primer momento aún no se había creado la clase de estado del dispositivo. Además, dado que aún se operaba con un solo dispositivo, la actualización la realizaba el manejador en cada iteración del bucle de renderizado, pero dado que no había ningún tipo de sobrecarga en el puerto y no se mostraban los *fps* al no haber ningún contenido virtual, no se apreciaba el problema que surgiría más tarde al reducir la frecuencia con la que se pedían actualizar los datos (unas 2-3 veces por segundo).

Como resultado de esta primera iteración era posible pedir conectar un dispositivo desde la interfaz, abriendo la comunicación en el puerto (primer dispositivo disponible aún), y empezar y recibir paquetes y convertirlos. Sin embargo, esa información sólo era mostrada para comprobar que el proceso podía llevarse a cabo. Por tanto, puede resumirse el objetivo general de esta iteración como la creación de la estructura básica necesaria para poder conectar un dispositivo desde la interfaz, leer el *stream* de bytes que envía, y convertirlo en paquetes válidos para comprobar cómo se realiza el proceso y con qué rendimiento (primeras bases de los módulos de comunicaciones, tratamiento de onda y dispositivo).

Esta iteración se lleva a cabo desde el 4 de Febrero del 2013 hasta el 14 de Marzo de 2013.

Iteración 2

Establecida la funcionalidad básica, se procedió a extenderla. En esta iteración se implementó la primera versión de la clase `MindWaveDisp`, encargada de almacenar el estado de dispositivo para MindWave. En esta segunda iteración ya era posible crear una instancia y almacenarla para poder manipularla. Esta versión almacenaba únicamente la información obtenible solamente a partir de los paquetes de dispositivo:

- Atención y Meditación
- Onda en bruto
- Valores High-Low de los 5 tipos de onda.
- Se añade además el ID en la plataforma.

Con esta versión se gestionan las intancias en el manejador en estructuras *vector*, lo que al final acabará siendo cambiado por los problemas descritos en la sección 5.3.1. No obstante, dado que solo se controla un dispositivo en esta etapa tan temprana de desarrollo no supone ningún problema emplear *vector*, y la futura refactorización es fácil llevarla a cabo.

En esta segunda iteración se detecta el problema de lectura en los puertos, al añadir más información visual y observar que la tasa de *frames* es de unos 4 *frames* por segundo, contando una tarjeta gráfica de gama alta. La lectura que emplea la clase `SimpleSerial` es síncrona y bloquea el programa hasta leer los bytes especificados. Al pedirlo en cada frame,

la actualización del frame se bloquea hasta que hay datos. Es por ello por lo que se cambia el diseño original y se sustituye el precario módulo de comunicaciones por el diseño «casi» final, a falta de establecer la interfaz genérica y operaciones estándar. Se cambia la clase `SimpleSerial` por la estructura de clases mostrada al final de la sección 5.2.2, que proporciona a la solución una lectura asíncrona, en un hilo independiente por cada puerto, y que además incorpora un buffer de almacenamiento intermedio. Ahora la lectura no bloquea el renderizado, de modo que se muestran en torno a 1000 *frames* por segundo.

Al acabar la iteración se captura un nuevo tipo de paquete del dispositivo y se incluye la intensidad de la conexión del dispositivo.

Como resultado de la iteración, ya es posible tener instancias del estado de los dispositivos y almacenar en ellas la información básica. Se incorpora la funcionalidad para gestionarlas, aunque todavía en una versión simplificada. También se desacopla la comunicación del resto de procesos, permitiendo que haya hilos diferentes para cada uno. Por tanto, puede resumirse el objetivo general de esta iteración como la ampliación y mejora de los objetivos de la iteración inicial.

Esta iteración se lleva a cabo desde el 15 de Marzo de 2013 hasta el 9 de Abril de 2013.

Iteración 3

Esta iteración viene marcada por el esfuerzo de crear la *fachada* del sistema y de posibilitar la operabilidad de varios dispositivos simultáneos, además de conseguir su conectividad por dispositivo específico y no primero disponible.

Se desarrolló la clase `PlatformMgr`, la clase fachada, cuya función es ocultar las clases y funcionalidad subyacente, de forma que no sea necesario conocer los detalles de implementación y manejar las clases directamente. Además proporciona un único punto de acceso y una visión unificada de la plataforma, y ofrece mayor facilidad al desarrollador.

En este momento, muestra las operaciones de añadir un dispositivo y recuperarlo, iniciar la comunicación, actualizar los dispositivos e información básica sobre su estado de funcionamiento. Junto al manejador de dispositivo que encapsulaba se añadió también el manejador del microcontrolador, con funcionalidad para abrir la comunicación en el puerto y enviar bytes. El resto se completa en la iteración 6.

Se siguió completando y mejorando el tratamiento de las conexiones y los dispositivos. Se preparó la estructura de datos relativa a las comunicaciones y las operaciones necesarias para tratar varias y se asociaron con los dispositivos. En el método de actualización de estado de los dispositivos pasó de actualizarse el primer dispositivo de forma explícita a recorrer las conexiones existentes y actualizarlas. De esta forma desde la interfaz se pedía conectar los dos dispositivos y pasaba a mostrarse los valores de los dos. Sin embargo, aún presentaba dos problemas: había muchos conflictos entre dispositivos a la hora de intentar conectar varios, y

la actualización de estado se hacía por *polling*. Además, todos los dispositivos se gestionaban al mismo tiempo, cuando podrían tener diferentes frecuencias. Por ello se implementaron nuevos mecanismos.

Finalmente, para dispositivos MindWave se gestionó la conexión mediante los identificadores de fábrica (inscritos en la tapa de la batería). Enviando un código de conexión diferente y posteriormente ese código de fábrica, permite buscar únicamente ese dispositivo. Se asignaron dos códigos únicos para cada dispositivo para evitar el conflicto mencionado anteriormente.

Se cambió el método de actualización de dispositivos, de modo que en lugar de pedir la actualización cada cierto tiempo y hacerla de todos los dispositivos, cada uno de ellos informase cuando tenía datos, de forma que no hubiese llamadas innecesarias. Se implementó en la clase `BufferedAsyncSerial` un nuevo atributo ID, de forma que al superar el buffer el tamaño deseado pudiese informar al manejador de qué dispositivo específico actualizar. También se modificaron las interfaces para cambiar los métodos necesarios para conseguir dicha funcionalidad.

Por último, se limitó la lógica de la aplicación en 60 fps para evitar un exceso de llamadas de obtención de valores del dispositivo, reduciendo considerablemente el retardo en el renderizado.

Esta iteración fue bastante compleja, y como resultado de la iteración, ya es posible utilizar la plataforma de forma segura bajo un único punto de acceso, que además facilita la labor de desarrollo y abstrae de los detalles de implementación y relación entre componentes. También es posible conectar dispositivos específicos en el puerto deseado y que varios de ellos operen de forma simultánea a gran velocidad de muestreo. Por tanto, puede resumirse el objetivo general de esta iteración como la compleción del módulo de comunicaciones y la parte de gestión y control del módulo de dispositivo.

Esta iteración se lleva a cabo desde el 10 de Abril de 2013 hasta el 31 de Mayo de 2013.

Iteración 4

La cuarta iteración del proyecto se dedicó al submódulo matemático y al entrenamiento por red neuronal. Para hacerlo posible se realizaron dos estudios sobre la red neuronal. También se integró el arranque automático del sistema al iniciar la aplicación.

Se añadió el submódulo matemático `mathEEG`, dotándole de parte de la funcionalidad completa. Se implementaron las funciones `bin_power` y la FFT (Transformada Rápida de Fourier) que era necesaria para la función de potencia. Ésto permite obtener el espectro de onda normalizado a partir de la onda en bruto. Como tal, el espectro por sí mismo, no podía emplearse en la estimación de valores, pero en esta primera versión se asoció la estimación con una banda de potencia para comprobar que los cálculos se realizaban correctamente.

Junto con este incremento en el sistema hubo que añadir nuevas estructuras de datos en el manejador de dispositivo para gestionar la nueva información. Se añadieron tres nuevas estructuras, todas del mismo tamaño:

- Una estructura que contenía la frecuencia en cada banda.
- Una estructura para el espectro de onda tras la función de potencia.
- Otra estructura para el espectro de onda normalizado.

Además de estas estructuras se añadieron nuevos atributos para los valores estimados y nuevas operaciones para poder obtener esos valores. Ahora el usuario/desarrollador puede escoger qué valores recuperar, si los del dispostivo o los estimados. Aprovechando el cambio de estructuras se reimplementaron también las estructuras de la clase `MindWaveHandler`, pasando de *vector* a *map*.

Se implementó la estructura base y la interfaz de la clase `NeuralNetwork` y se asociaron las llamadas para el cálculo de valores. Esta clase carga un fichero de la red entrenada al inicio y utiliza la red para calcular la salida en función de la entrada. Después de preparar la estructura se dedicó un esfuerzo adicional a implementar el módulo de trazas, necesario para poder comenzar el estudio de la red.

Se diseñó la clase `WaveRecorder` y el estado de debug de los dispositivos, permitiendo guardar en ficheros los espectros de onda en ese instante de tiempo junto a los valores de Meditación y Atención que les correspondían, de forma que pudiesen analizarse más tarde. Estos ficheros eran almacenados en el directorio *Data*.

Con estos ficheros comenzó el primer estudio, en el que se centraron los esfuerzos en encontrar las relaciones que presentaban los valores del espectro de onda, y qué bandas, con respecto a los valores de Atención y Meditación. También se estudió qué espectro aportaba mayor precisión, si el base o el normalizado. También se orientó a buscar el tamaño del conjunto de entrenamiento que producía el error menor. Tras observar que en la teoría la red era capaz de conseguir un error muy bajo y de estimar muy bien la salida, pero que al probarla en casos reales mostraba comportamientos extraños, como valores negativos, se decidió realizar un segundo estudio más completo, de forma que se corrigiesen esos fallos.

En este segundo estudio se centraron los esfuerzos en encontrar la mejor función de activación y algoritmo de entrenamiento, ya que los valores negativos se debían a que en el primer análisis la función de activación permitía un rango de salida de -1 a 1, necesitándose de 0 a 1. Se probaron diferentes funciones con diferentes tamaños de muestras.

Tras este segundo estudio se estableció la red neuronal final, eligiéndose como función de activación la función *Sigmoide* y el algoritmo de entrenamiento de *Backpropagation*, obteniendo un error real bajo, tal y como se mostró en la sección 5.4.2, figura 5.17. Ahora la clase `NeuralNetwork` ya podía cargar el fichero en el arranque y construir la red mediante

la cual calcular los valores de salida dado un espectro de entrada.

Como último esfuerzo en esta iteración, se implementó el arranque automático y conexión automática de los dispositivos en el arranque de la plataforma, de modo que el usuario final no tuviese que preocuparse de conectarlos manualmente ni ninguna otra acción, simplemente «enchufar y jugar».

Como resultado de la iteración, es posible iniciar la plataforma y los dispositivos de forma automática, además de integrar todo el submódulo matemático y de entrenamiento de onda. Por tanto, puede resumirse el objetivo general de esta iteración como la automatización de la plataforma y la compleción del módulo de dispositivo y de trazas, de modo que el usuario pueda iniciar la plataforma y utilizar los dispositivos con «un solo botón» y obtener siempre los mejores resultados del dispositivo adaptado a él.

Esta iteración se lleva a cabo desde el 3 de Junio de 2013 hasta el 26 de Junio de 2013.

Iteración 5

En este momento se comienza el desarrollo del módulo de representación gráfica final. Se desecha la interfaz basado en *overlays* que se había estado usando hasta el momento y se empieza la implementación de la interfaz basada en widgets.

Para ello se implementa la clase `Widget`. Esta clase es un `adaptador` para CEGUI, en el cual se basan los Widgets. Esta clase permite definir diferentes tipos de Widgets en la plataforma, tales como widget de estado o widget de onda, y mostrarlos en la interfaz gráfica, empleando el motor de renderizado de OGRE 3D. Incorpora también un menú mediante el cual pueden elegirse los widgets que se desean mostrar u ocultar. Dado que CEGUI no tiene sistema de entrada, ésta ha de capturarse mediante otro sistema de entrada (OIS en nuestro caso) e inyectarla a CEGUI. Para este tipo de eventos de inyección y los propios eventos que puedan producirse en los widgets se implementa la clase `WidgetEventHandler`. Esta clase permite capturar y tratar cualquier evento relacionado con los widgets o el motor de CEGUI y realizar las acciones convenientes.

La declaración de widgets se realiza en ficheros externos con extensión `.layout`, en lugar de emplear código CEGUI en la clase `Widget`. Con esto se consigue que la modificación de los widgets se más simple e intuitiva y no precise de recompilado para hacerse efectiva. Por tanto `Widget` cargará esos ficheros `.layout` cuando cree el widget particular en lugar de hacerlo manualmente por código.

Se implementan la lógica necesaria para utilizar las clases `DynamicLines` y `DynamicRenderable` para incluir la representación 3D de la onda de entrada. Precisa de un vector de entrada en forma de puntos en el espacio, por lo que se elige un plano en el espacio para representar la onda y se convierten los valores en coordenadas 2D (la tercera dimensión siempre tiene valor 0). Se pasa el vector de entrada a `DynamicLines` y éste lo representa fue-

ra de la visión de la cámara. Con una segunda cámara se captura la representación de la onda y se muestra en un render a textura en un widget, tal y como se comentó en la sección 5.6.2. Se establece la actualización de valores cada 0.3 segundos para no sobrecargar el sistema, ya que es un proceso «pesado».

Al final de la iteración se lleva a cabo la representación 3D del demostrador y la lógica necesaria para el juego. Se realiza un primer diseño en Blender del demostrador, recreando el juego MindBall. Para ello se implementan las clases `BrainBallGame` y `Ball`. La primera se encarga de toda la lógica del juego, tal como puntuación de los jugadores, si se está jugando o en tiempo de espera, etc. La segunda se encarga del encapsulamiento del *SceneNode* de Ogre y su desplazamiento y rotación, simulando el movimiento real de la pelota; recibe la dirección a la que debe ir y ésta se desplaza.

Como resultado de la iteración, se obtiene una solución con capacidad de representación 3D tanto del juego en cuestión como de la onda de entrada, y de representación 2D de valores de aplicación o dispositivos y texturas por medio de Widgets. Éstos son independientes unos de otros y admiten personalización y movimiento por la interfaz, así como su activación/desactivación. Por tanto, puede resumirse el objetivo general de esta iteración como la incorporación de retroalimentación gráfica al usuario/desarrollador de lo que está haciendo en la plataforma y los valores de salida que ésta está generando.

Esta iteración se lleva a cabo desde el 28 de Junio de 2013 hasta el 16 de Julio de 2013.

Iteración 6

La sexta iteración se centró en el montaje del circuito electrónico y la programación del microcontrolador para controlarlo. El circuito en sí mismo se componen de tres circuitos más pequeños; dos pueden apreciarse en la figura 5.31 en la página 139, que son los circuitos de movimiento del carro de impresión y de los pulsadores, y el tercero se muestra en la figura 5.34 en la página 140, correspondiente al circuito integrado CNY70 (LED infrarrojo + fotorresistencia).

Se implementó primero el circuito correspondiente al motor y se escribió el código Arduino necesario para controlar la activación del motor y el sentido de giro a partir de las operaciones correspondientes enviadas por el puerto. Desde el programa principal era posible mover el carro en un sentido u otro o pararlo, aunque aún no se detectaba el final.

A continuación se implementó el circuito de los pulsadores, que actuaban como fin de carrera. Se añadió al progama Arduino la lógica necesaria para capturar la pulsación y detener el motor; también para volver a dejarlo aproximadamente en la posición central.

Por último se implementó el circuito correspondiente al sensor infrarrojo, de modo que al colocar una franja de rayas negras y blancas en un lateral, y al estar el sensor ubicado en el carro cerca del lateral, se pueden contar franjas cuando el carro se mueve. Esto permite

implementar un sistema de posicionamiento preciso, indicando a qué franja se desea mover el carro. En este punto se redefinen los códigos de operación utilizados, de modo que numeros positivos indiquen la franja i de un jugador y números negativos la franja i del jugador contrario; el 0 es el centro; y el valor de parada es el 127 (el arranque del motor va inclusive en el movimiento hacia un lado u otro).

Por último se sincroniza la representación virtual con la física, de modo que ambas pelotitas se muevan al mismo tiempo.

Como resultado de la iteración, se obtiene una solución con capacidad de controlar componentes físicos mediante circuitos electrónicos y microcontrolador, consiguiendo utilizar la plataforma para interactuar con el entorno físico real mediante pensamiento. Por tanto, puede resumirse el objetivo general de esta iteración como la incorporación del demostrador de la plataforma, en forma de demostrador físico, y el módulo necesario para su control.

Esta iteración se lleva a cabo desde el 18 de Julio de 2013 hasta el 16 de Agosto de 2013.

Iteración 7

En esta última iteración, de corta duración, se llevaron a cabo mejoras puntuales en el código y se revisaron los contenidos multimedia. Se realizaron pequeñas refactorizaciones de código para mejorar la implementación y la estructura de clases y métodos. También se revisaron todos los contenidos multimedia y se cambiaron algunos, tal como las texturas de los objetos 3D y la colocación de las cámaras. Se reajustó también el render a textura acorde a la nueva distribución 3D.

También se revisó el demostrador físico, comprobando si el posicionamiento estaba bien implementado y ajustado y si la sincronización era buena.

Para finalizar, se comprobó que tanto software como hardware cumplían los objetivos propuestos y todo estaba correctamente reflejado en la documentación.

El objetivo general de esta iteración fue la revisión y corrección de los contenidos del proyecto, de forma que todo quedase lo mejor posible.

Esta iteración se lleva a cabo desde el 19 de Agosto de 2013 hasta el 29 de Agosto de 2013.

La tabla 6.1 recoge las fechas de todas las iteraciones.

Iteración	Fecha de inicio	Fecha de fin
1	4-02-2013	14-03-2013
2	15-03-2013	9-04-2013
3	10-04-2013	31-05-2013
4	3-06-2013	26-06-2013
5	28-06-2013	16-07-2013
6	18-07-2013	16-08-2013
7	19-08-2013	29-08-2013

Cuadro 6.1: Tabla de fechas de las iteraciones del proyecto.

6.2 Recursos y costes

En esta sección se describen los diferentes recursos utilizados, tanto en cuestión temporal como económica. Se detallan el tiempo de desarrollo y el coste estimado de recursos humanos, así como el coste de los recuros hardware empleados.

6.2.1 Coste económico

El período de desarrollo de este Proyecto Fin de Carrera ha comprendido 7 meses, concretamente desde el 4 de Febrero hasta el 29 de Agosto. En el se incluye el trabajo de implementación, la realización del estudio de las ondas cerebrales y redes neuronales y el montaje del demostrador físico y su circuito electrónico.

Recurso	Cantidad	Coste
Sueldo	1	15.166,66
Dispositivo MindWave	2	188
Arduino Nano v3	1	33
Componentes electrónicos	n/a	50
Computador	1	564
Total		**16.001,66**

Cuadro 6.2: Tabla de costes del proyecto.

El salario se ha estimado acorde a un salario anual de 26000 euros brutos.

6.2.2 Estadísticas del repositorio

Para llevar a cabo el control de versiones del proyecto se ha empleado un repositorio Mercurial ofrecido por GoogleCode. En las siguientes figuras se muestra la actividad del repositorio, el número de cambios subidos y cuantas líneas de código han sido modificadas a lo largo de los meses de desarrollo.

La figura 6.1 muestra la actividad en el branch por defecto del repositorio, en cual se ha llevado a cabo el desarrollo del proyecto. El eje X representa el tiempo de desarrollo. El eje

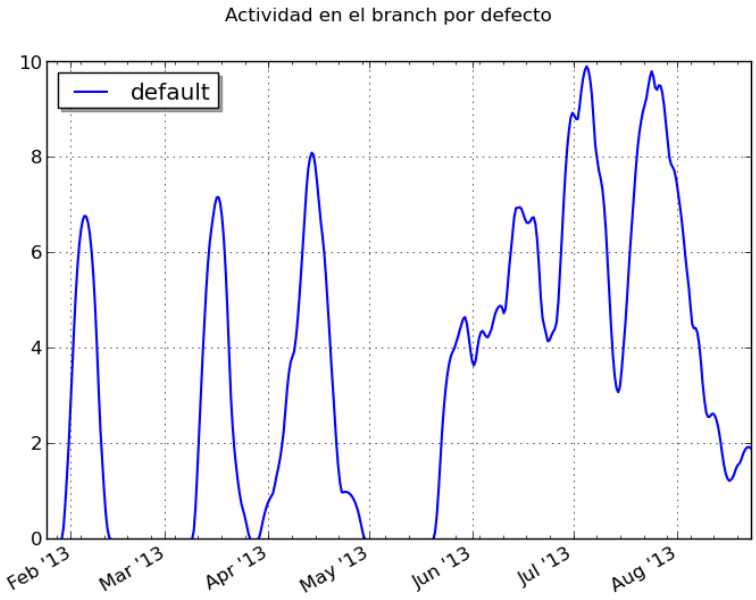


Figura 6.1: Actividad total en el repositorio.

Y muestra la actividad, medida en *commits* realizados. En la figura se aprecian además dos períodos de tiempo en los que no hubo actividad; estas paradas se debieron a períodos de exámenes y entregas de prácticas en los que fue imposible continuar el desarrollo a la par que realizando los exámenes. La actividad del mes de Agosto corresponde a la inclusión y revisiones de la documentación del PFC.

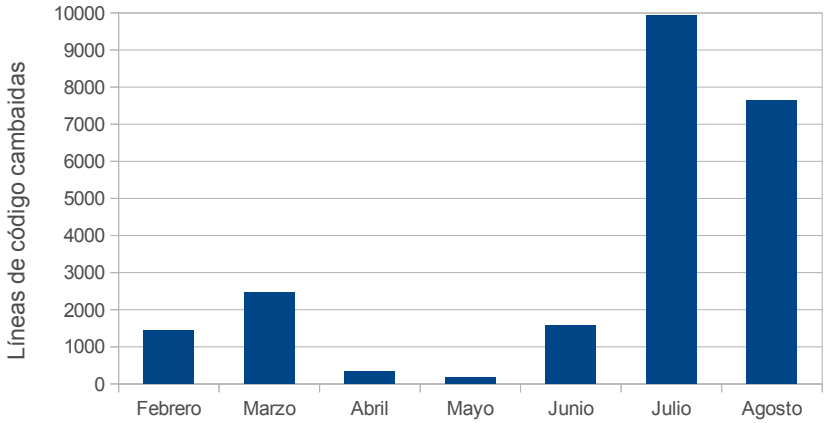


Figura 6.2: Cambios, en número de líneas de código, en el repositorio.

La diferencia tan notable de líneas de Julio y Agosto respecto a los demás es debida principalmente a los archivos XML, que se modificaron muchas veces hasta conseguir la estética de widgets deseada, y a la inclusión de la documentación, que al estar escrita en LaTeX, cuenta como código en los cambios. Debe aclararse también la aparente incongruencia con la

figura anterior en el mes de Abril, donde se muestra gran actividad y en cambio las líneas de código cambiadas son escasas; ésto es debido a los cambios y revisiones del Anteproyecto en código LaTeXincluidas en la actividad total pero no en el código del proyecto.

Se ha utilizado la herramienta *CLOC*[1] para contabilizar las líneas de código fuente que componen el proyecto. Se incluyen tanto los ficheros de cabecera en lenguaje C/C++ (aunque la totalidad del proyecto se ha realizado en el lenguaje C++, se dispone de una cabecera en código C sobre la que se apoya el *parser*), los ficheros de código fuente C++ y los ficheros en lenguaje XML empleados. La Tabla 6.3 muestra los resultados arrojados por CLOC.

Lenguaje	**Ficheros**	**Espacios**	**Comentarios**	**Código**
XML	3	0	0	5143
C/C++ header	26	321	439	859
C++	23	534	219	2641
make	1	15	7	42
Arduino	1	12	6	94
Total	**54**	**882**	**671**	**8779**

Cuadro 6.3: Resultado de la ejecución de CLOC.

[1] `http://cloc.sourceforge.net/`

Capítulo 7

Conclusiones y propuestas

En este capítulo se muestran los objetivos alcanzados durante el desarrollo de este Proyecto Fin de Carrera y se realizan una serie de propuestas futuras en relación con el mismo. También se da una conclusión final y personal de lo que ha supuesto llevarlo a cabo.

7.1 Objetivos alcanzados

Tal y como se ha presentado en el Capítulo 5, los objetivos principales que se expusieron en el Capítulo 3 han sido cumplidos. Se ha desarrollado una plataforma bajo sistemas GNU/Linux para desarrollar aplicaciones NUI que permite la integración de varios dispositivos de EEG y abstrae al programador de los detalles concretos de implementación de los dispositivos.

Se ha conseguido establecer una base de comunicaciones extensible y portable (ver Sección 5.2.2), de modo que pueden añadirse nuevas formas de comunicación para nuevos dispositivos. Además, estas nuevas comunicaciones pueden implementarse con la misma biblioteca, dada la extensa cantidad de protocolos que soporta *Boost* mediante *Asio*. Al ser *Boost* una biblioteca portable y muy extendida, el módulo de comunicación podría ser portado a otros sistemas. El enfoque multi-hilo que se ha seguido en el desarrollo del proyecto permite que la gestión de comunicaciones sea eficiente, y que varios dispositivos simultáneos no produzcan bloqueos entre varias peticiones.

Por otra parte se ha homogeneizado la interfaz de los dispositivos (ver Sección 5.4.2), de forma que éstos se presenten de la misma forma a la plataforma. Se ha separado por un lado el estado del dispositivo del control del mismo, de forma que el acoplamiento sea el mínimo posible. Esto posibilita que las particularidades del dispositivo queden en la *clase* del estado y no afecten al resto del sistema, y de forma que tratar varios no suponga tener en cuenta particularidades, ya que presentan la misma interfaz. La plataforma permitirá que varios dispositivos se conecten de forma simultánea y sean funcionales al tratar las comunicaciones de forma independiente.

El módulo de tratamiento de onda (ver Sección 5.3.2) permite que los datos que envíen los dispositivos descritos anteriormente se conviertan desde una representación de bajo nivel y propia del dispositivo a otra representación de datos común, que pueda ser utilizada por la plataforma. Dado que el proceso es distinto para cada dispositivo, cada uno de ellos debe implementar su *parser* o conversor particular, incluido en su manejador de dispositivo.

Una de las características más atractivas de la solución y que amplia los objetivos que se propusieron en la sección 3.2, es el ajuste de onda al usuario y estimación de valores. Este proceso requiere un tratamiento previo de la señal, dividiendo el espectro de onda en diferentes bandas o frecuencias, asociadas a cada tipo de onda, y posteriormente digitalizado y normalizado su valor en cada banda. Posteriormente se lleva a cabo un cálculo mediante redes neuronales y permite al usuario seguir utilizando el sistema, y que éste siga proporcionando valores inmediatos de Atención y Meditación, en los casos en los que el dispositivo deja de enviarlos. Esta característica soluciona el mayor problema que presentan los dispositivos MindWave en algunas situaciones.

Cumpliendo con el objetivo de la interfaz de comunicación con sistemas externos, se incluye el módulo de control, que permite la utilización del microcontrolador Arduino mediante los valores de dispositivo (ver Sección 5.7.2). Pueden enviarse códigos que utilizará Arduino en su programa interno para el control del circuito electrónico. Ésto ha servido a la vez para desarrollar el demostrador de la plataforma, que fue otro de los objetivos principales establecidos.

Se incluye además funcionalidad de representación gráfica, tanto creación y representación de Widgets como representación 3D (ver Sección 5.6.2). La plataforma permite representar entornos 3D que recreen el demostrador, junto con una interfaz de usuario basada en Widgets que muestren los valores de los dispositivos, su estado, conexión, lógica de juego, etc. Cumplimenta la representación física y proporciona retroalimentación al usuario, al igual que al desarrollador. Tanto el motor de gráficos OGRE 3D como el motor de widgets CEGUI son portables a otros sistemas.

El código se basa también en buenos principios de diseño, basados en el uso de patrones y guías de estilo de código para mejorar su estructura y legibilidad. Además, todas las bibliotecas y motores empleados son portables, por lo que la plataforma podría ejecutarse en otros sistemas operativos, aunque su objetivo principal sea funcionar en GNU/Linux.

Por último, se ha realizado un manual de usuario sencillo, explicando las características del sistema paso a paso, empezando desde las capas inferiores a las superiores.

7.2 Propuestas de trabajo futuro

Este proyecto pretende ser una plataforma de desarrollo extensible capaz de dar soporte a varios dispositivos, admitir nuevos tipos y poder realizar modificaciones. En el momento de

escritura de este documento sólo se dispone de un tipo de dispositivo EEG.

Como trabajo futuro, se realizan las siguientes propuestas de ampliación y mejora:

- **Incorporación de nuevos tipos de dispositivo**: Desde que se concibió este proyecto, se pensó en que la plataforma soportase múltiples dispositivos, fuesen iguales o de diferente fabricante o modelo, con el objetivo de ser una solución multidispositivo para los desarrolladores, que abstrajese los detalles de implementación y facilitase su uso. Dado que en el momento de realizar este Proyecto Fin de Carrera sólo se disponía de los dispositivos MindWave de NeuroSky, éstos son los únicos dispositivos para los que se incluye soporte.

  Sin embargo, existen diversos dispositivos comerciales de semejantes características y costes, utilizados igualmente por desarrolladores independientes. Estos dispositivos, como puedan ser EPOC de Emotiv o MindSet del mismo NeuroSky, deben poder estar soportados en la plataforma. Para ello deberían adquirirse los dispositivos, realizar el mismo estudio de comunicación y estructura de paquetes que utilizan como ya se hizo con MindWave, e incorporar el soporte. Habría que añadir las clases de dispositivo específicas por cada tipo, y que extiendan de la interfaz común.

  Concretamente, se necesitaría crear una nueva clase de instancia de dispositivo que herede de la clase `DispInterface` e implemente su funcionalidad. Además de esta nueva clase, habría que ampliar el manejador añadiendo una nueva clase de *parseo* o conversión de paquetes, ya que cada dispositivo tiene un formato propio. El proceso tiene poca complejidad, requiriendo más esfuerzo en el *parser* que en la instancia de dispositivo. Gracias al diseño de clases, incorporar un nuevo dispositivo es sencillo y facilita mucho la labor a otros desarrolladores.

- **Incorporación de nuevos tipos de comunicación**: Dado que la totalidad de dispositivos disponibles en el mercado no utiliza la conexión USB para conectar los dispositivos y enviar datos, la evolución natural de la plataforma es incluir más métodos de comunicación, como puede ser Bluetooth.

  Incluso podría darse el caso de que un mismo dispositivo soporte varias formas de comunicación, por lo que todas deberían estar soportadas. Ésto además aporta diversas ventajas en cuanto a robustez y eficiencia. Dado el caso de que falle un sistema, puede ser utilizado el otro cambiando simplemente el sistema de comunicación que quiere utilizarse en la plataforma, de forma que no impida utilizar el sistema. También en caso de que haya demasiada carga sobre un sistema de comunicación, USB por ejemplo, permita reducir la sobrecarga utilizando algunos dispositivos en otro medio de comunicación (si es posible).

  Añadir nuevos tipos de comunicación es una labor muy sencilla para el desarrollador. Debe crear una nueva clase que herede de la interfaz `CommInterface` e implemente las

funciones establecidas. Para implementar esta funcionalidad no es necesario integrar bibliotecas adicionales de entrada/salida, ya que en la plataforma se emplea *Boost Asio*, que incluye soporte para todos los tipos de comunicación. Esto permite implementar rápidamente las funciones necesarias variando sólo las partes de código particulares a cada tipo de comunicación y manteniendo el resto, de forma que no sea necesario dedicar mucho esfuerzo.

- **Cálculo de otros posibles estados mentales**: Para utilización y control en la lógica de aplicaciones se ha optado por incluir los estados mentales de `Atención` y `Meditación`, en un rango de 0 a 100. Sin embargo, mediante la onda de entrada de la actividad cerebral es posible detectar patrones de entrada y asociarlos con otros estados mentales.

  Tal y como se vio en el capítulo de Antecedentes, hay más estados mentales asociados a ciertos tipos de ondas. Llevando a cabo el estudio pertinente sobre cada estado y cada onda, y estableciendo los algoritmos necesarios y funciones para calcularlos, podríamos utilizar mayor cantidad de valores para controlar la aplicación, permitiendo opciones más complejas y precisas. No obstante, algunos de esos nuevos valores podrían ser muy complejos de calcular de forma precisa, o incluso imposible con determinados dispositivos, pero sería merecedor realizarlo.

  Valores que podrían añadirse nuevos: Sueño, Consciencia, Alerta, Memoria...

  Para realizar este incremento en la plataforma, el usuario debería estudiar y recoger muestras de los estados mentales que desea calcular. Con ellos, podría entrenar una nueva red neuronal para la estimación de valores. Por último, debería añadir las funciones necesarias en el submódulo `mathEEG` que devuelvan la estimación de esos valores empleando la nueva red neuronal, cargada desde un fichero diferente. Estas modificaciones requieren bastante tiempo debido al estudio previo a realizar y, sobre todo, al proceso de entrenamiento de la red.

- **Posibilidad de reajustar el entrenamiento en segundo plano**: Actualmente, y por las razones comentadas en la sección 5.4.1, el ajuste de entrenamiento se realiza como programa externo. Sin embargo, una vez integrada la red en el sistema, puede que siga sin ser óptima para determinados usuarios, acumulando mayor error.

  Sería conveniente que en tiempo real y en segundo plano pudiese ejecutarse un proceso que observase si se produce error en el valor entrenado con respecto al que produciría el dispositivo –comparando con el paquete si éste llega- y aplicase una rectificación para corregir, o en su defecto reducir, ese error cometido. De esta forma la precisión aumenta al tener por un lado la red neuronal para adaptarla al usuario y por otro un mecanismo de correción de posibles desviaciones cuando varía el usuario sin cambiar el entrenamiento.

Para llevar a cabo esta propuesta, sería necesario crear un nuevo submódulo matemático que se ejecutase en un hilo diferente en segundo plano y que fuese comparando los valores estimados con los valores recibidos del dispositivo. Mediante alguna función de suavizado y teniendo en cuenta el historial de errores, podría reajustar el valor calculado al valor correcto o más próximo a dicho valor.

- **Incorporación renderizado 2D real**: Con esta propuesta se pretender mejorar la representación de onda de entrada y añadir nuevas gráficas. Dado que OGRE 3D no soporta de forma nativa renderizado 2D es necesario es buscar soluciones externas. Sin embargo, las soluciones disponibles son muy restringidas o es necesario llevar a cabo operaciones demasiado complejas.

  La solución podría ser integrar alguna biblioteca de 2D vectorial, como puede ser Cairo, y hacer un wrapper que permita cargar en tiempo de ejecución el resultado como una textura en Ogre3D.

7.3 Conclusión personal

El desarrollo de este Proyecto Fin de Carrera ha supuesto la experiencia más cercana de toda la carrera a un proyecto real y de gran envergadura, tal y como los que se llevan a cabo en empresas. Es necesario saber capturar bien los requisitos de lo que se pretende hacer y llevar a cabo un diseño muy elaborado, pensando siempre en la robustez, la eficiencia y el usuario final, empleando para ello técnicas de Ingeniería del Software y Arquitectura de Computadores sin las cuáles no sería posible.

No obstante, este Proyecto Fin de Carrera me ha enseñado que el Ingeniero Informático no se restringe a estos campos únicamente, sino que debe dominar muchísimas más áreas de conocimiento para poder llevar a cabo su labor. Debe dominar desde Matemáticas, Álgebra, Algoritmia, Estadística, hasta otras como Sociología y Psicología.

Es por ello por lo que me siento orgulloso de haber escogido esta carrera al presentar mi plataforma y mostrar el potencial del Ingeniero Informático integrando tantas áreas de conocimiento en un producto, y alejando los estereotipos que se tienen y que he oído estos cinco años, tales como pasarse el día gestionando una base de datos o arreglando ordenadores, porque ante todo somos eso, **Ingenieros**, capaces de imaginar y desarrollar complejos sistemas que facilitan nuestro trabajo o mejoran nuestro entretenimiento.

También como experiencia particular me ha servido para afianzar conocimientos adquiridos a lo largo de la carrera, adquirir nuevos, y para mejorar como ingeniero y futuro trabajador, enfocando mi interés profesional en las tecnologías interactivas.

ANEXOS

Anexo A

Manual de usuario

En este anexo se describe la forma de utilizar la plataforma. Primero se describe cuáles son las dependencias de paquetes necesarios para su construcción y cómo se construye. Seguidamente se detallan dos guías de usuario, una guía para el usuario final que simplemente utiliza la aplicación y otra guía para el desarrollador de aplicaciones, que la utiliza su aplicación particular y/o realiza modificaciones en la plataforma.

A.1 Construcción

Para construir la plataforma se proporciona un fichero *Makefile*. Con este fichero puede construirse la plataforma desde el código fuente proporcionado, haciendo uso de un archivo `main.cpp`; puede ser el ya incluido, que inicia la aplicación del demostrador de la plataforma, o puede ser uno creado por el usuario con su aplicación particular.

No obstante, antes debe satisfacer una serie de dependencias. El sistema del usuario deberá contar con los siguientes paquetes y bibliotecas de terceros instaladas:

- Boost-System
- Boost-Thread
- FANN (*Fast Artificial Neural Network*)[1]
- OGRE 3D
- CEGUI

Satisfechas las dependencias, puede procederse a invocar el comando `make` de la siguiente forma:

```
make -f makefile
```

De esta forma la plataforma es compilada por defecto con opciones de depuración. Se incluyen modos de compilación adicionales, utilizando el argumento '`mode=`' seguido de uno de los siguientes valores:

[1] `http://leenissen.dk/fann`

- **release**: Se compila con optimizaciones y ninguna opción de depuración ni impresión de valores de *debug*.
- **debugex**: Se compila con opciones de depuración e impresión por consola de los valores de los paquetes recibidos directamente del dispositivo referentes a Atención, Meditación, onda en bruto y valores *Low-High*.
- **debugsig**: Se compila con opciones de depuración e impresión por consola de los valores de los paquetes recibidos directamente del dispositivo referentes a intensidad de la señal, conexión y desconexión.
- **debugha**: Se compila con opciones de depuración e impresión por consola de los valores de la representación interna de la información contenida en los paquetes de dispositivo, a excepción de la onda en bruto.
- **debugharaw**: Se compila opciones de depuración e impresión por consola de los valores de la representación interna de la onda en bruto.

A.2 Guía del usuario final

El usuario final únicamente interactuará con la plataforma mediante el demostrador o nuevas aplicaciones, pero en ningún caso realizará modificaciones ni "tocará" el código, de modo que sólo necesitará seguir una serie de indicaciones de empleo de los dispositivos. De este modo, se describe en esta sección cómo debe colocarse y utilizar los dispositivos EEG, y como iniciar e interactuar con la aplicación.

En la figura A.1 se muestra el dispositivo MindWave de forma detallada, nombrando cada elemento, de forma que puedan seguirse las indicaciones fácilmente.

Primero se enciende el dispositivo MindWave. Para ello se desliza el botón de encendido a la posición `On`. Una vez encendido, se encederá un LED situado encima del botón de encendido, pudiendo lucir con luz roja o azul, y de forma continua o parpadeando. El significado de estas situaciones se recoge en la tabla A.1.

Color	Parpadeo	Estado del dispositivo
Apagado		Dispositivo apagado
Rojo	No	Dispositivo encendido pero no conectado al receptor.
Azul	No	Dispositivo encendido y conectado al receptor.
Rojo o Azul	Sí	Batería baja.

Cuadro A.1: Significado del LED de estado MindWave.

A continuación, el usuario deberá colocarse correctamente el dispositivo en la cabeza, siguiendo para ello una serie ordenada de pasos. Una mala colocación del dispositivo dificultará la óptima utilización del mismo. Los pasos a seguir son:

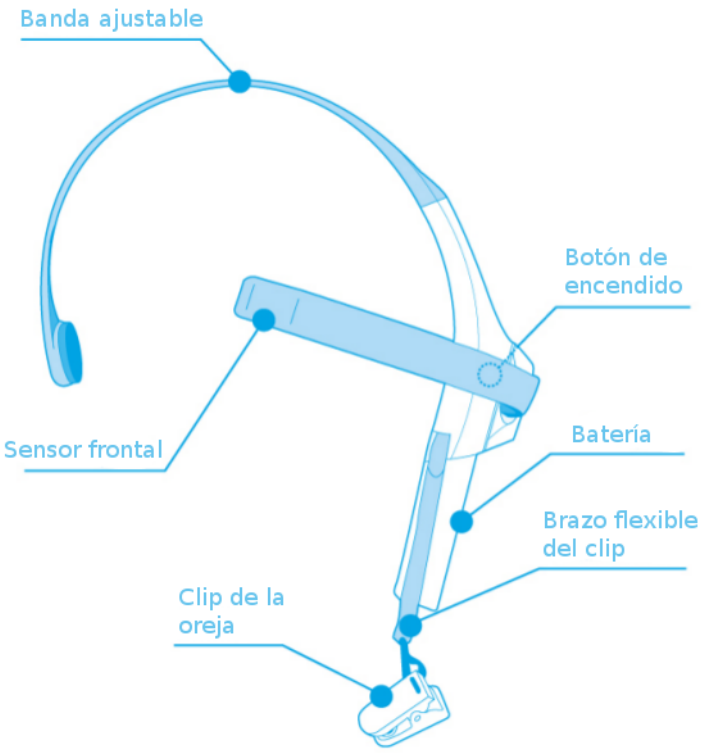


Figura A.1: Dispositivo de EEG MindWave.

1. Rotar el brazo del sensor frontal 90º hacia delante (lado opuesto del botón de encendido). Figura A.2.

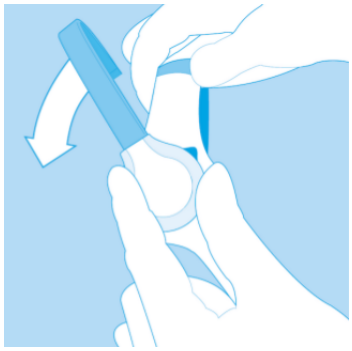


Figura A.2: Paso 1 de colocación del dispositivo Mindwave.

2. Extender la banda ajustable para facilitar la colocación y colocarse el dispositivo desde delante hacia detrás. El sensor frontal debe hacer contacto en la frente. Apretar la banda ajustable hasta que el dispositivo quede firmemente sujeto sin oprimir la cabeza. Figura A.3.
3. Pasar el brazo flexible de la oreja por detrás de ésta, y después enganchar el clip en el lóbulo de la oreja.

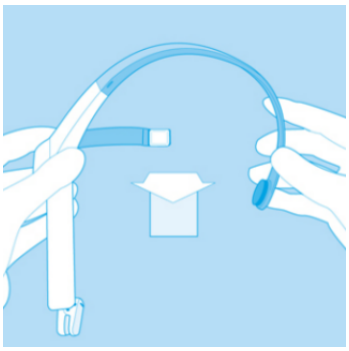


Figura A.3: Paso 2 de colocación del dispositivo Mindwave.

4. Asegurar que los dos metales de contacto del interior del clip se encuentran en contacto con la piel del lóbulo de la oreja. Tanto el cabello como cualquier pieza de joyería, como pendientes, han de ser quitados de la oreja. Figura A.4.

Figura A.4: Paso 4 de colocación del dispositivo Mindwave.

5. Ajustar el sensor frontal de forma que esté en contacto con la frente, justo por encima del párpado. Debe mantenerse aparado apartado el cabello y asegurarse que el sensor siempre mantiene contacto con la piel de la frente.
6. La figura A.5 muestra el resultado de la colocación del sensor. Si durante la utilización del dispositivo se experimenta pérdida de señal, debe repetirse el proceso hasta colocar correctamente el dispositivo.

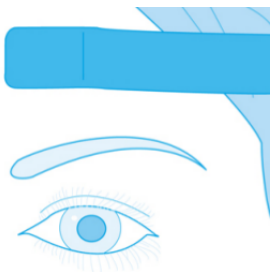


Figura A.5: Paso 6 de colocación del dispositivo Mindwave.

Colocado correctamente el dispositivo, sólo queda iniciar la aplicación y jugar. Dado que se ha implementado el arranque automático de la plataforma y la aplicación junto con el arranque del sistema, el usuario encenderá el computador y cuando haya arrancado, se iniciará de forma automática la aplicación y la configuración de la plataforma, conectando los dispositivos y comenzando el juego. Cuando el usuario desee terminar la sesión, presionará la tecla `Escape` y automáticamente comenzará el apagado de todo el sistema. Para finalizar, se quitará el dispositivo realizando los pasos de colocación de forma inversa y lo apagará mediante el botón de encendido.

A.3 Guía del desarrollador

La plataforma ha sido diseñada de forma que abstraiga al usuario de los detalles de implementación de los dispositivos y de la gestión de recursos, trabajando siempre por medio de la fachada de la plataforma que sirve como único punto de acceso. Si el usuario quiere desarrollar una aplicación que utilice la plataforma, sólo necesita trabajar con la fachada, sin tener que ocuparse de la gestión de hilos, creación de instancias ni gestión de memoria. La fachada implementa el patrón *Singleton*, de forma que sólo haya una instancia.

Para poder utilizar la plataforma y los dispositivos EEG en la aplicación en cuestión del desarrollador, éste debe seguir una serie de pasos:

1. Iniciar la plataforma al inicio de la aplicación mediante la función `init` .
2. Registrar los dispositivos que utilizará, declarando los IDs de plataforma y de fábrica, así como el tipo de dispositivo, mediante la función `addDisp`.
3. Iniciar el tipo de comunicación de los dispositivos, indicando el ID y el puerto mediante la función `openSerial`.
4. Enviar el comando de conexión por ID a los receptores mediante la función `connectToID`.
5. Recuperar el estado y valores de dispositivo y utilizarlos en la aplicación. Para ello primero deberá recuperar la instancia de dispositivo mediante la función `getDisp` y posteriormente recuperar el valor deseado con la función `getXXXX` correspondiente.
6. Si desarrolla la aplicación empleando un motor de renderizado compatible con CEGUI, puede utilizar también la clase `Widget` para representación de valores.

En el listado A.1 se muestra la estructura básica de una aplicación ejemplo que emplea la plataforma con un dispositivo de EEG MindWave.

El desarrollador puede optar también por incorporar nuevos tipos de dispositivo o comunicación. A continuación se detalla el proceso de inclusión de un nuevo tipo de dispositivo en la plataforma.

```
1   #include "PlatformMgr.h"

3   // Iniciar la plataforma
4   PlatformMgr::getPlatformMgr()->init();

6   ...
7   Inicio de la aplicacion
8   ...

10  // Registro del dispositivo de tipo MindWave y apertura del puerto
11  PlatformMgr::getPlatformMgr()->addDisp(1, PlatformMgr::MINDWAVE, 0
        x6C, 0x77);

13  PlatformMgr::getPlatformMgr()->openSerial(1, "/dev/ttyUSB0", 115200)
        ;

15  // Envio de comando de conexion a ID especifico
16  PlatformMgr::getPlatformMgr()->connectToID(1);

18  ...

20  // Bucle principal de la aplicacion
21  while ( !terminar ) {
22    sending_one = PlatformMgr::getPlatformMgr()->isSending(1);
23    if (sending_one) {
24      meditation_disp_one = PlatformMgr::getPlatformMgr()->getDisp(1)
            ->getMeditationLevel();
25      attention_disp_one = PlatformMgr::getPlatformMgr()->getDisp(1)->
            getAttentionLevel();
26    }

28    ...
29    Operaciones con los valores de dispositivo
30    ...
31  }
```

Listado A.1: Estructura básica de aplicación que utiliza la plataforma y un dispositivo EEG MindWave.

El primer paso es crear una nueva clase que herede de la interfaz `DispInterface`, incorporando sus atributos e implementando sus funciones. Para este ejemplo, se supone que el nuevo dispositivo tiene los mismos atributos que los dispositivos MindWave además de un atributo Sueño. La clase `EjemploDisp` queda como se muestra en el listado A.2.

Para que la plataforma tenga constancia del nuevo tipo de dispositivo y pueda ser registrado mediante `addDisp`, es necesario incluir un código único en `PlatformMgr` y ampliar el método en cuestión, como muestra el listado A.3.

Por último será necesario incluir un nuevo *parser* para el dispositivo. La implementación del parser es muy dependiente del dispositivo, por lo que sólo se describe a alto nivel la modificación necesaria. El desarrollador deberá crear una nueva clase encargada del proceso

```
#include "DispInterface.h"

class EjemploDisp : public DispInterface {
  public:
       EjemploDisp();
       EjemploDisp(int id, const char& ms, const char& ls);
       ~EjemploDisp();

       int getID();
    StatusID getStatus();

    int getAttentionLevel();
    int getMeditationLevel();
    int getSleepingLevel();
    int getPoorSignal();
    deque<int> getRawWaves() const;

    void setAttentionLevel(int att);
    void setMeditationLevel(int med);
    void setSleepingLevel();
    void setLowHighWaves(unsigned char lenght, const unsigned char *
        values);
    void setPoorSignal(int signal);

    void setRawWaves(const unsigned char *values);

 private:
    char _ms, _ls;

    int _meditationLevel, _attentionLevel, _sleepingLevel;

    vector<int> _LowHighWaves;
    deque<int> _RawWaves;
    int _signal;
};
```

Listado A.2: Cabecera del nuevo tipo de dispositivo *EjemploDisp*.

```
enum DispType {
  MINDWAVE = 1,
  EJEMPLODISP = 2,
  OTHER = 0};

  ...

void
PlatformMgr::addDisp
(int id, DispType type, const char& ms, const char& ls)
...
if (type == EJEMPLODISP)
  _MWMgr->addDisp(id, new EjemploDisp(id, ms, ls));
...
```

Listado A.3: Registro del nuevo tipo de dispositivo.

de conversión de paquetes, y la incluirá en el manejador de dispositivo. Establecerá una función de callback de modo que cuando convierta un paquete, lo envíe al manejador. El manejador completará la instancia de dispositivo *EjemploDisp* con los datos recibidos.

Para finalizar, el desarrollador podrá utilizar la plataforma con el nuevo tipo de dispositivo de la misma forma que se utilizaba el dispositivo MindWave en el listado A.1, cambiando `PlatformMgr::MINDWAVE` por `PlatformMgr::EJEMPLODISP`.

Anexo B

Estudio de la red neuronal

En este anexo se recoge el estudio realizado sobre la red neuronal con el objetivo de determinar las relaciones entre los patrones de entrada y las salidas, qué espectro (base o normalizado) de onda presenta mejores relaciones y qué funciones de activación y entrenamiento producen los mejores resultados.

En el primer análisis se estudian las relaciones entre patrones de entrada y salida, así como el espectro que se escogerá definitivamente. Para ello, se generaron dos ficheros de entrenamiento con la actividad cerebral capturada. En el primer fichero se tiene el espectro base; en el segundo el normalizado. Dado que aún no se había estudiado el tamaño de muestra que ajustaba mejor la red, se capturaron 50 muestras. Los ficheros generados acorde a la estructura descrita en la tabla 5.5 de la sección 5.5.2 presentaban el siguiente aspecto:

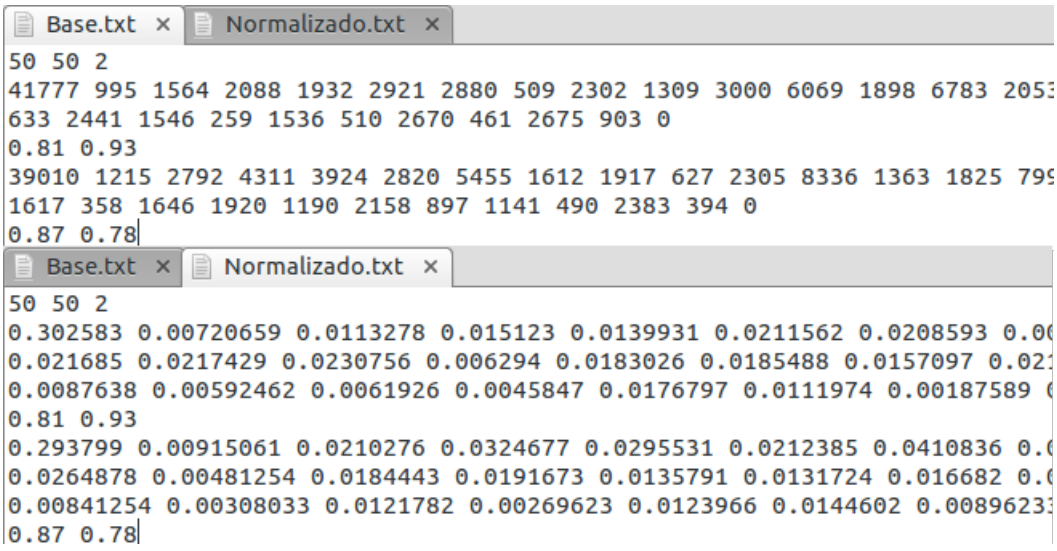


Figura B.1: Muestras base y normalizadas del primer análisis.

Así, al valor base 41777 le corresponde el valor normalizado 0,30, al valor 995 el valor normalizado 0,0072, etc. Ambos representan los valores de Meditación y Atención 81 y 93 respectivamente (codificado de 0 a 1).

Tras realizar esta prueba, se determinó que el espectro normalizado se comportaba mejor y producía menor error en la red, sobre todo por tratar mejor los valores pico que pueden presentarse en ocasiones en algunas de las bandas; a diferencia de los valores base, el valor no se dispara tanto al estar normalizado al total del espectro.

Tras este primer análisis, el siguiente tuvo como objetivo determinar la función de activación y algoritmo de entrenamiento. Para el caso del algoritmo de entrenamiento no fue necesario un análisis previo, sino que se escogió al comprobar que la biblioteca `FANN` sólo soportaba el algoritmo de *Backpropagation* en diferentes versiones. Las cuatro versiones soportadas son:

- **Incremental**: Algoritmo de retropropagación estándar donde los pesos son actualizados después de cada patrón de entrada.
- **Batch**: Algoritmo de retropropagación estándar donde los pesos son actualizados después de calcular el error cuadrático medio de todo el conjunto de entrenamiento. Más lento pero puede producir mejores resultados.
- **RPROP**: Una mejora de la versión Batch que no utiliza el ratio de aprendizaje, sino que es adaptativo.
- **QUICKPROP**: Una mejora de la versión Batch que utiliza el ratio de aprendizaje junto a otros parámetros avanzados.

De los cuatro se optó por la versión *RPROP*, ya que aún siendo más lenta, produce mejores resultados y, sobre todo, es adaptativa a las muestras, necesario dada la naturaleza del problema.

Escogido también el algoritmo de entrenamiento, es necesario determinar la función de activación de las neuronas. Para llevar a cabo esta elección se planteó un análisis extenso y completo, con diferentes tamaños del conjunto de entrenamiento. El proceso a realizar sería:

1. Tomar un número prefijado de muestras.
2. Analizar los espectros generando gráficas para observar las variaciones y continuidad de la onda.
3. Entrenar la red neuronal con las muestras anteriores.
4. Una vez entrenada, probarla con un conjunto de pruebas distinto al de entrenamiento y anotar los valores esperados y los obtenidos.
5. Generar gráficas y datos estadísticos del error obtenido y sobre los que decidir la mejor función.

Los tamaños de muestra escogidos fueron: 50, 100, 200, 300 y 500. El tamaño del set de entrenamiento fue de 260 muestras, igual para todas las funciones. De las funciones de

activación soportadas por la biblioteca `FANN` se probaron la función *Sigmoide* y la función *Elliot*, descartando las funciones simétricas por producir un rango de salida de -1 a 1.

Dada la extensión del estudio, se mostrarán sólo las gráficas y datos de la función de activación y el tamaño del conjunto de entrenamiento escogidos. Sí se incluirá una tabla donde se recojan los datos estadísticos de todas ellas, sobre los que se tomó la decisión. El estudio completo se adjunta en el CD del proyecto.

La figura B.2 muestra el patrón de entrada de los valores de Meditación y Atención provenientes del sensor eSense del dispositivo MindWave, patrón esperado obtener por la red neuronal. Se aprecia la variación fluida de valores, sin producirse saltos muy grandes de un valor al siguiente. Se muestra solo en un rango de 40 muestras, dado que el total son 260.

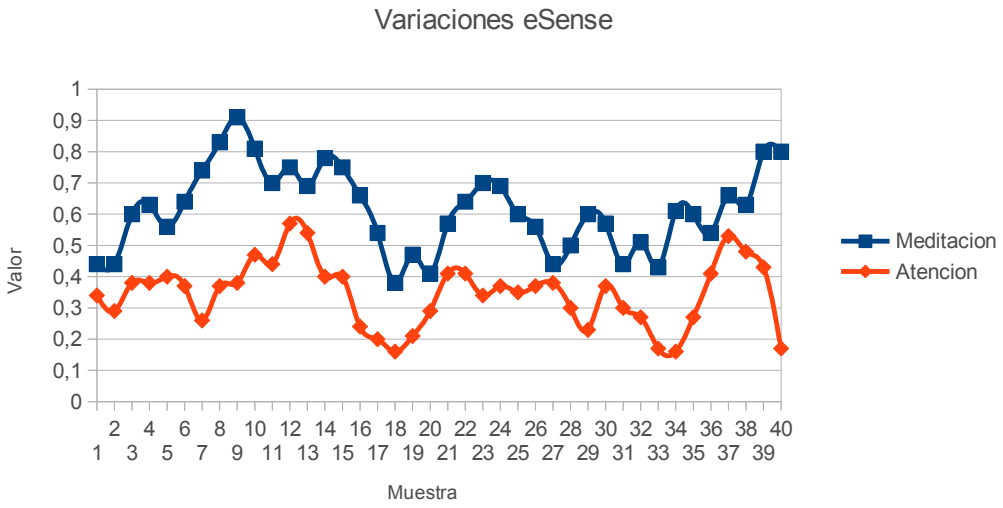


Figura B.2: Variación de la entrada del sensor eSense, correspondiente a los valores de Meditación y Atención.

Por cuestiones de espacio y claridad, se incluyen a continuación sólo los patrones de entrada de las ondas Delta y Theta en sus respectivas frecuencias en las figuras B.3 y B.4, ya que las ondas Alpha, Beta y Gamma tienen 6, 18 y 18 frecuencias respectivamente. Se muestran igualmente en un rango de 40 muestras.

La onda Delta presenta una variación muy pequeña y continua en sus bandas 2 y 3, en un rango de 0 a 0,05, destacando la primera banda tanto en tamaño de la variación y escala. Se mueve en un rango de 0,05 a 0,35 aproximadamente y sus variaciones son más bruscas.

En la onda Theta, sin embargo, todas sus frecuencias se mueven en un rango semejante, de 0 a 0,1 aproximadamente en el caso normal. En las muestras 18, 19 y 20 podemos observar un pico de entrada, más pronunciado en la banda 2.

Tras el entrenamiento de la red, se realiza el proceso de prueba. El conjunto de muestras de prueba se introduce en la red y se guarda la salida. El patrón de salida de los valores estimados de Meditación y Atención del eSense se muestran en la figura B.5.

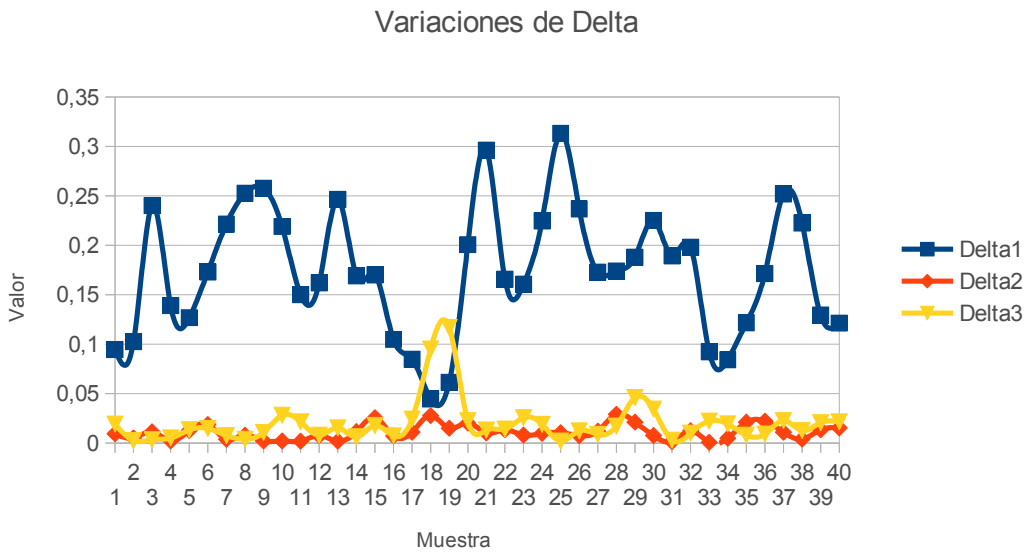


Figura B.3: Variación de la entrada de la onda Delta descompuesta en sus tres bandas de frecuencia.

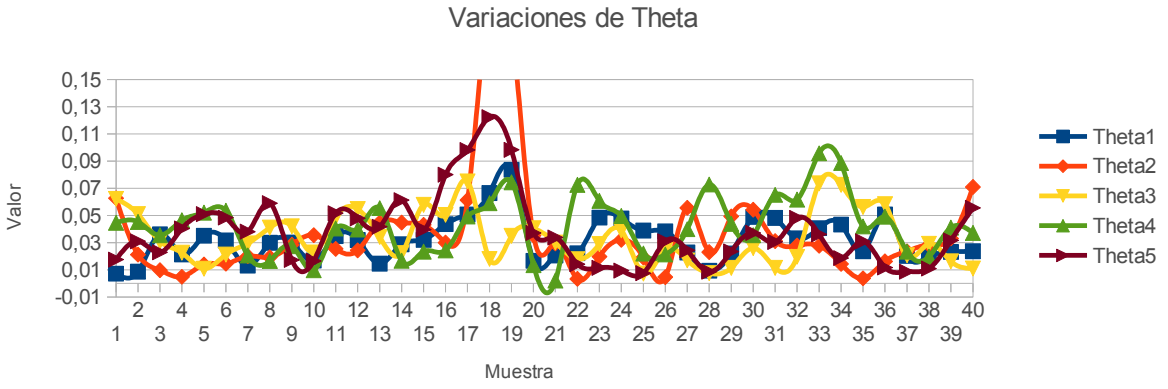


Figura B.4: Variación de la entrada de la onda Theta descompuesta en sus tres bandas de frecuencia.

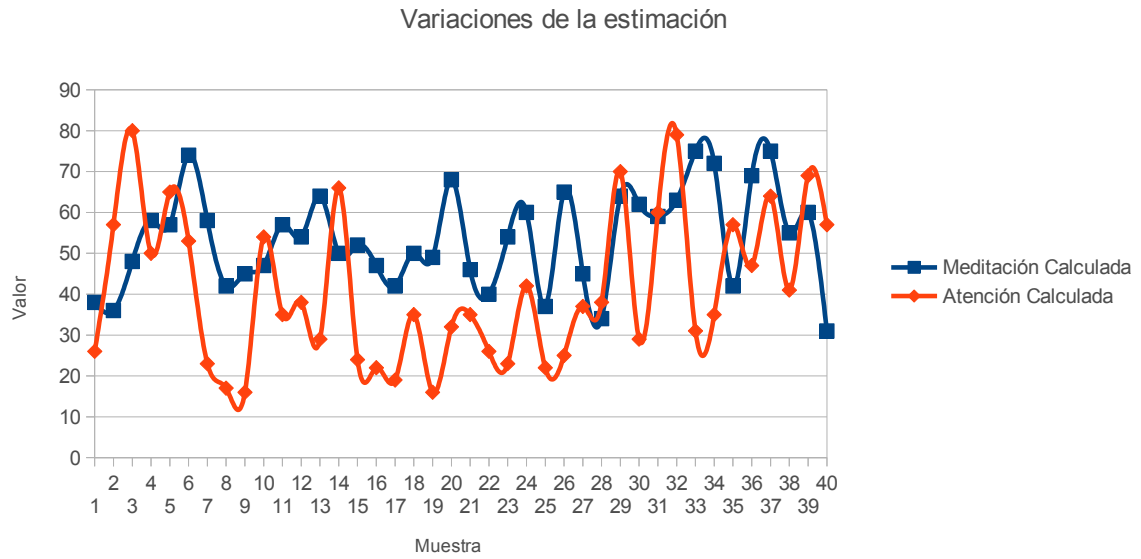


Figura B.5: Variación de la salida de la red neuronal, correspondiente a los valores del sensor eSense.

Puede observarse que el patrón de salida es muy semejante al mostrado en la figura B.2, aunque ha cometido errores. Aparentemente, el error es mayor para el valor Atención que para el valor Meditación, aunque tan importante es el error conjunto como el individual, ya que el sistema emplea los dos. En las figuras B.6 y B.7 se muestra el error individual y el error conjunto de la salida.

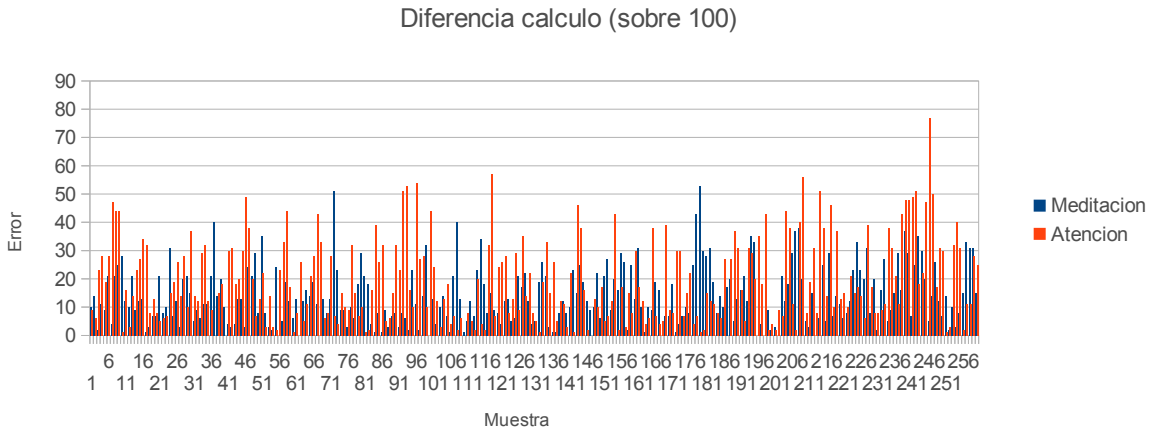


Figura B.6: Error individual cometido en la estimación de valores.

Para estudiar de forma más detallada el error, junto a los gráficos se ha generado una serie de datos estadísticos, recogidos en la tabla B.1.

En la tabla B.2 se recogen los datos estadísticos de todas las funciones/tamaños de muestra estudiados.

Como puede apreciarse en la tabla B.2, la función que menos error medio y acumulado

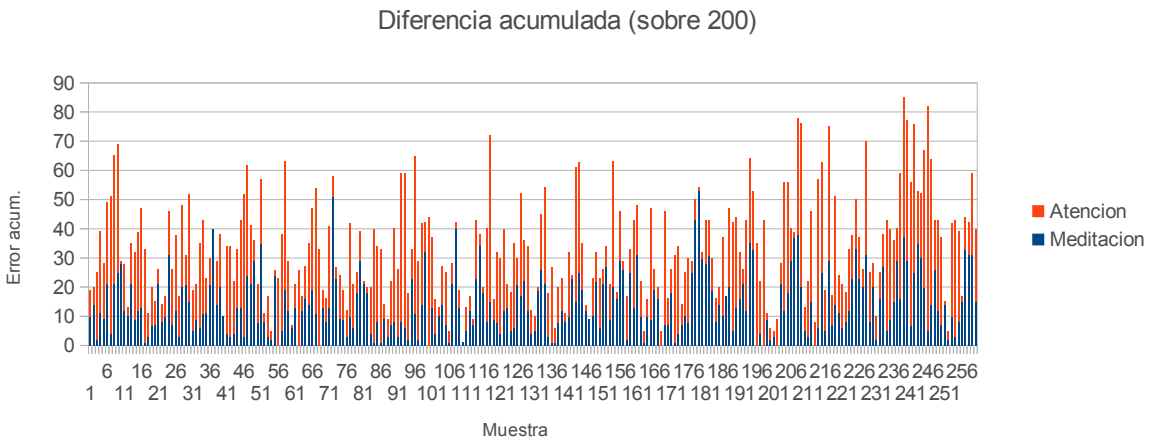


Figura B.7: Error conjunto cometido en la estimación de valores.

	Media	Desviación	C. Variación	Percentil 25	Percentil 50	Percentil 75
Meditación	14,103	10,308	0,730	6,75	12	21
Atención	19,161	14,677	0,765	7	15	30
Conjunto	33,265	17,117	0,514	20	32	43

Cuadro B.1: Datos estadísticos sobre el error generado en la estimación de valores con la función *sigmoide*.

muestra es la función *sigmoide* con un tamaño de muestras de entrenamiento de 300. Presenta además la menor desviación, aunque un mayor coeficiente de variación; la salida varía más pero el error se mantiene más cercano al error medio, que es el menor de todos. Un último dato interesante es el hecho de que el error vaya decrementándose conforme aumenta el número de muestras, pero a partir de 500 muestras el error comience a aumentar de nuevo.

	Media	Desviación	C. Variación	Percentil 25	Percentil 50	Percentil 75
Sigmoid/50						
Meditación	49,276	15,080	0,306	40	50	60
Atención	44,423	22,051	0,496	27	43	59,25
Conjunto	93,7	23,063	0,246	78,75	93	109
Sigmoid/100						
Meditación	48,242	15,055	0,312	38	47,5	59
Atención	33,25	22,789	0,685	14	30	49,75
Conjunto	81,492	27,588	0,338	63	82	100
Sigmoid/200						
Meditación	21,307	14,779	0,693	9	18	32
Atención	20,126	14,431	0,717	9	18	29
Conjunto	41,434	21,926	0,529	26	37	56
Sigmoid/300						
Meditación	14,103	10,308	0,730	6,75	12	21
Atención	19,161	14,677	0,765	7	15	30
Conjunto	33,265	17,117	0,514	20	32	43
Sigmoid/500						
Meditación	22,665	17,238	0,760	8	20	33
Atención	28,284	22,342	0,789	11,75	22	41
Conjunto	50,95	34,278	0,672	27	41	71
Elliot/50						
Meditación	22,653	14,627	0,645	9,75	22	33
Atención	21,715	15,500	0,713	10	18,5	31
Conjunto	44,369	23,346	0,526	27	41	58,25
Elliot/100						
Meditación	39,219	18,349	0,467	24	41	51
Atención	30,788	17,915	0,581	16	30	43
Conjunto	70,007	25,067	0,358	52,75	70	86
Elliot/200						
Meditación	27,130	15,320	0,564	16	26	37
Atención	18,380	13,658	0,743	7	16	26
Conjunto	45,511	21,618	0,475	30	43	58
Elliot/300						
Meditación	16,776	12,116	0,722	7	14	23
Atención	20,973	15,518	0,739	8	19	30,25
Conjunto	37,75	20,503	0,543	21,75	35,5	50
Elliot/500						
Meditación	23,284	16,308	0,695	10	21	35
Atención	23,411	17,998	0,769	9	20	36
Conjunto	46,696	24,210	0,518	30	43	63

Cuadro B.2: Datos estadísticos sobre el error generado en la estimación de valores de todas las funciones/tamaños de muestra.

Anexo C

Diagrama de clases

La figura C.1 muestra el diagrama de clases general de la Plataforma, recogiendo las más importantes y omitiendo algunas dependencias por simplicidad.

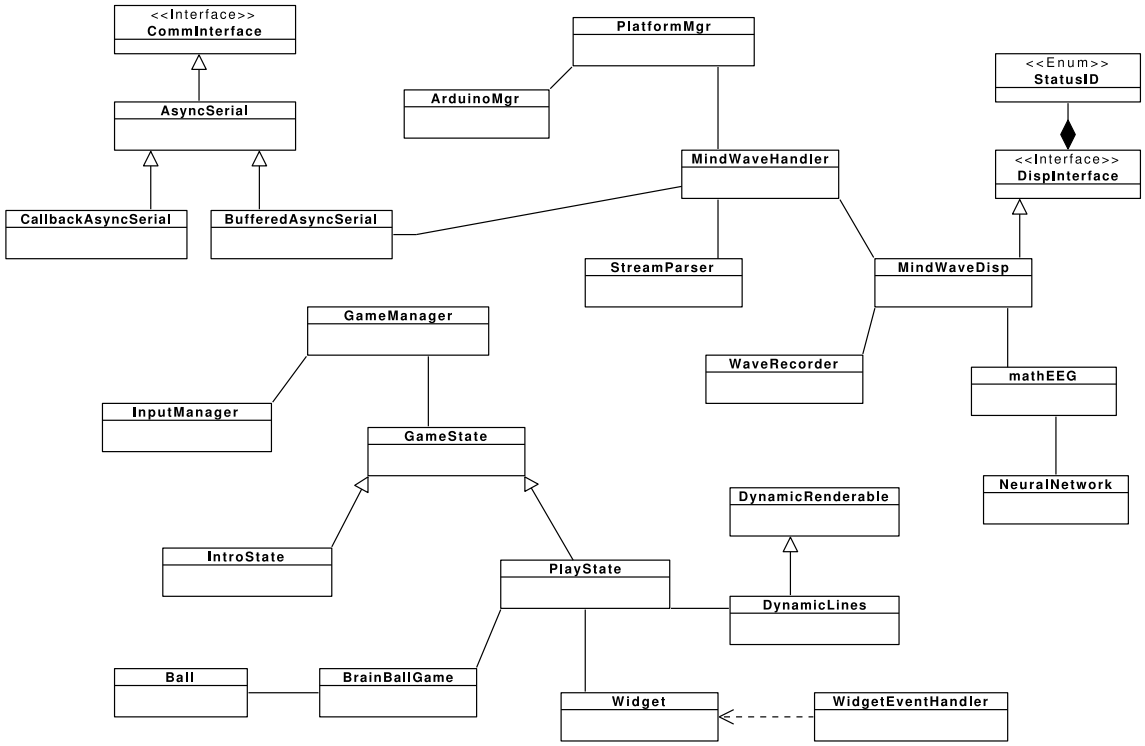


Figura C.1: Diagrama de clases de la Plataforma.

Anexo D

Código fuente

Dada la extensión del código fuente, éste no ha sido incluido en la documentación. Se incluye en versión electrónica en el CD adjunto a este documento.

Dicho código fuente se estructura en los siguientes directorios:

- **data**: Contiene el fichero de creación de la red neuronal y los ficheros de trazas que genere el sistema. Además contiene el directorio *Arduino* que contiene el código fuente del programa cargado en el microcontrolador.
- **include**: Contiene los archivos de cabecera C/C++ del sistema.
- **media**: Contiene los ficheros multimedia necesarios, a saber:
  - Archivos de mallas de la escena de OGRE 3D.
  - Archivos de texturas.
  - Subdirectorios *fonts*, *imagesets*, *layouts*, *looknfeel*, *schemes* y *xml_schemas* con los ficheros necesarios para el funcionamiento del módulo de Widgets (CEGUI).
- **src**: Contiene los archivos de código fuente C++ del sistema.
- El directorio raíz contiene los archivos de configuración de OGRE 3D y el makefile de generación.

Bibliografía

[ARP93] James Anderson, Edward Rosenfeld, y Andras Pellionisz. *Neurocomputing*. Mit Press, 1993.

[BR99] Wolfgang Banzhaf y Colin Reeves. *Foundations of Genetic Algorithms*. Number v. 5. Morgan Kaufmann Publishers, 1999.

[Bri88] Oran Brigham. *The Fast Fourier Transform and Its Applications*. Prentice-Hall Signal Processing Series. Prentice Hall, 1988.

[CGJ+11] Pedro Campos, Nicholas Graham, Joaquim Jorge, Nuno Nunes, Philippe Palanque, y Marco Winckler. *Human-Computer Interaction – INTERACT 2011*. Springer Berlin Heidelberg, 2011.

[Cor08] Ambient Corporation. The Audeo, 2008. url: `http://www.theaudeo.com/`.

[FA12] David Vallejo Fernández y Cleto Martín Angelina. *Desarrollo de Videojuegos: Arquitectura del Motor*. Bubok, 2012.

[FMA+12] Francisco Moya Fernández, Carlos González Morcillo, David Villa Alises, et al. *Desarrollo de Videojuegos: Técnicas Avanzadas*. Bubok, 2012.

[FT+05] Assaf Feldman, Emmanuel Munguia Tapia, et al. ReachMedia: On-the-move interaction with everyday objects. *Ambient Intelligence Group, MIT Media Laboratory*, 2005.

[Fur11] Borko Furht. *Handbook of Augmented Reality*. SpringerLink : Bücher. Springer New York, 2011.

[GHJV96] Erich Gamma, Richard Helm, Ralph Johnson, y John Vlissides. *Design Patterns*. Addison-Wesley, 1996.

[gmeG09] g.tec medical engineering GmbH. intendiX, 2009. url: `http://www.intendix.com/`.

[Goo12] Google. Google Glass, 2012. url: `http://www.google.com/glass/start/`.

[GP12] Ilya Grinblat y Alex Peterson. *OGRE 3D 1.7 Application Development Cookbook*. Packt Publishing, 2012.

[K+00] Hau Kato et al. Virtual Object Manipulation on a Table-Top AR Environment. 2000.

[Ker10] Felix Kerger. *OGRE 3D 1.7 Beginner's Guide*. Packt Publishing, 2010.

[Kot11] Stefan Kottwitz. *LaTeX Beginner's Guide*. Packt Publishing, 2011.

[Koz92] John Koza. *Genetic Programming: vol. 1 , On the programming of computers by means of natural selection*. MIT Press, 1992.

[Kup04] Gina Kuperberg. Electroencephalography, Event-Related Potentials, and Magnetoencephalography. *Essentials of neuroimaging for clinical practice*, 2004.

[LF08] Raquel López y José Miguel Fernández. *Las Redes Neuronales Artificiales*. Netbiblo S.L., 2008.

[Lib09] Mark Libenson. *Practical approach to electroencephalography*. Saunders W.B. Elsevier/Saunders, 2009.

[LIM08] Pedro Larrañaga, Iñaki Inza, y Abdelmalik Moujahid. Redes Neuronales. 2008.

[Man01] Steve Mann. *Intelligent Image Processing*. John Wiley and Sons, 2001.

[Mit97] Tom Mitchell. *Machine Learning*. McGraw-Hill Series in Computer Science. McGraw-Hill Science, 1997.

[MJC+12] Carlos González Morcillo, Javier Albusac Jiménez, Sergio Pérez Camacho, et al. *Desarrollo de Videojuegos: Programación Gráfica*. Bubok, 2012.

[MJF+12] Francisco Jurado Monroy, Javier Albusac Jiménez, David Vallejo Fernández, et al. *Desarrollo de Videojuegos: Desarrollo de Componentes*. Bubok, 2012.

[MMC09] Pranav Mistry, Pattie Maes, y Liyan Chang. WUW – Wear Ur World – A Wearable Gestural Interface. 2009.

[Moo03] James Moor. *The Turing Test: The Elusive Standard of Artificial Intelligence*. Springer, 2003.

[MR86] James MacClelland y David Rumelhart. *Parallel Distributed Processing. Vol. 2. Psychological an Biological Models*. MIT Press, 1986.

[Neg05] Michael Negnevitsky. *Artificial Intelligence: A Guide To Intelligent Systems*. Addison Wesley Publishing Company Incorporated, 2005.

[Nor88] Donald Norman. *The Psychology of Everyday Things*. Basic Books, 1988.

[OB09] Jonathan Oxer y Hugh Blemings. *Practical Arduino*. Apress, 2009.

[RM86] David Rumelhart y James MacClelland. *Parallel Distributed Processing. Vol. 1. Foundations*. MIT Press, 1986.

[RN04] Stuart Russell y Peter Norvig. *Inteligencia Artificial: Un enfoque moderno*. Pearson Educación S.A., 2004.

[Ros09] Timothy Ross. *Fuzzy Logic with Engineering Applications*. Wiley, 2009.

[RP03] Piedad Tolmos Rodríguez-Piñero. *Introducción a los algoritmos genéticos y sus aplicaciones*. Working papers. Universidad Rey Juan Carlos, Servicio de Publicaciones, 2003.

[SFH00] Dieter Schmalstieg, Anton Fuhrmann, y Gerd Hesina. Bridging Multiple User Interface Dimensions with Augmented Reality. 2000.

[Sim83] Herbert Simon. *Reason in Human Affairs*. Stanford University Press, 1983.

[Sim96] Herbert Simon. *The Sciences of the Artificial*. Mit Press, 1996.

[Sky10] Neuro Sky. *MindSet Communications Protocol*. Apress, 2010.

[SOF05] Christian Sandor, Alex Olwal, y Steven Feiner. Immersive mixed-reality configuration of hybrid user interfaces. 2005.

[Sta07] Richard Stallman. *GNU Emacs Manual*. Gnu press. Free Software Foundation, 2007.

[Ste12] Gideon Steinberg. Natural User Interfaces. *University of Auckland*, 2012.

[Str97] Bjarne Stroustrup. *The C++ Programming Language, 3ª Ed.* Addison-Wesley, 1997.

[THBK07] William Tatum, Aatif Husain, Selim Benbadis, y Peter Kaplan. *Handbook on EEG Interpretation*. Demos Medical Publishing, 2007.

[WW11] Daniel Wigdor y Dennis Wixon. *Brave NUI World: Designing Natural User Interfaces for Touch and Gesture*. Elsevier Science, 2011.

Este documento fue editado y tipografiado con LaTeX
empleando la clase **arco-pfc** que se puede encontrar en:
`https://bitbucket.org/arco_group/arco-pfc`