UNIVERSIDAD DE VALLADOLID

E.T.S.I. TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA EN TECNOLOGÍAS ESPECÍFICAS DE TELECOMUNICACIÓN

MENCIÓN EN TELEMÁTICA

Desarrollo de aplicaciones en LabVIEW y Android para sensores inalámbricos con giróscopo y comunicación con un simulador de conducción basado en Unity

Autor:
D. Manuel Álvarez Marbán

Tutor:
D. David González Ortega

Valladolid, 30 de mayo de 2017

TÍTULO:	**Desarrollo de aplicaciones en LabVIEW y Android para sensores inalámbricos con giróscopo y comunicación con un simulador de conducción basado en Unity**
AUTOR:	**D. Manuel Álvarez Marbán**
TUTOR:	**D. David González Ortega**
DEPARTAMENTO:	**Departamento de Teoría de la Señal y Comunicaciones e Ingeniería Telemática**

TRIBUNAL

PRESIDENTE:	**Dña. Miriam Antón Rodríguez**
SECRETARIO:	**D. David González Ortega**
VOCAL:	**D. Mario Martínez Zarzuela**
SUPLENTE:	**D. Francisco Javier Díaz Pernas**
SUPLENTE:	**D. José Fernando Díez Higuera**

FECHA:	**30 de mayo de 2017**
CALIFICACIÓN:	

Resumen de TFG

Este Trabajo de Fin de Grado se enmarca en la monitorización fisiológica mediante sensores del conductor, para la detección de posibles estados de fatiga o baja atención del mismo. Los sensores empleados durante el mismo pertenecen a la empresa Shimmer. Este trabajo se centra principalmente en el uso de uno de estos sensores (giróscopo) para medir la velocidad angular de giro del volante del coche.

Bajo este objetivo principal, se llevaron a cabo desarrollos de aplicaciones en diversas tecnologías: una aplicación Android y una aplicación LabVIEW. Con ellas, partiendo de funciones básicas de recepción de datos y sincronización con todos los sensores, implementamos una serie de procesados sobre la señal del giróscopo, tomando como referencia varios artículos científicos.

Por otra parte, se ha desarrollado la comunicación y sincronización de la aplicación Android con un simulador de conducción desarrollado en Unity para la recogida de datos del conductor.

Palabras clave

Sensores fisiológicos, giróscopo, Shimmer, LABVIEW, Android, Unity, simulador

Abstract

This end-of-degree Project focuses on the driver's physiological monitoring through sensors to detect possible states of fatigue or low attention. The used sensors belong to the Shimmer company. This work mainly focuses on the use on one of these sensors (gyroscope) to measure the angular speed of turn of the car's steering wheel.

Under this main aim, developments of applications of diverse technologies were carried out: an Android application and a LabVIEW application. With them, from basic functions of data reception and synchronization with all the sensors, a series of processing on the gyroscope signal was implemented similarly to several scientific papers.

Moreover, the communication and synchronization of the Android application with a driving simulator developed with Unity have been fulfilled to store driving performance data.

Keywords

Physiological sensors, gyroscope, Shimmer, LABVIEW, Android, Unity, simulator

Agradecimientos:

A mi familia, a mis padres, por su apoyo incesable.

A mi novia, por su apoyo y comprensión en época de exámenes.

A mi tutor David, por su ayuda y siempre cercana atención.

Y a todos los compañeros que me han ayudado durante estos años.

ÍNDICE

1. INTRODUCCIÓN 1
1.1 Motivación y objetivos 1
1.2 Fases del proyecto 3
1.3 Medios y elementos empleados 4
1.4 Estructura de la memoria 5
2. SEGURIDAD VIAL, SENSORES Y TECNOLOGÍA 7
2.1 Seguridad vial y accidentes 7
2.1.1 Fatiga y somnolencia al volante 11
2.2 Introducción a los sensores 16
2.3 Tecnología y sensores en vehículos actuales 17
2.3.1 Sensores en el volante 19
2.3.2 Visión artificial: cámaras de reconocimiento facial 20
2.3.3 Empresas y sistemas de monitorización actuales 22
3. SENSORES SHIMMER 26
3.1 Introducción a los sensores fisiológicos 26
3.2 Sensores Shimmer 27
3.2.1 Hardware Shimmer 27
3.2.2 Software Shimmer 35
4. TECNOLOGÍAS QUE EMPLEAREMOS 38
4.1 Introducción 38
4.2 LABVIEW y la programación gráfica 39
4.2.1 Introducción 39
4.2.2 Conceptos básicos de LABVIEW 40
4.2.3 Software LABVIEW necesario 43

4.3 Android y sistemas operativos móviles 46
4.3.1 Introducción y comparativa con otros sistemas móviles 46
4.3.2 Herramientas de desarrollo para Android 49
4.4 Unity y motores gráficos de videojuegos 51
4.4.1 Introducción y otros motores gráficos 51
5. LABVIEW 53
5.1 La aplicación base 53
5.2 Nuestra aplicación LABVIEW 57
5.2.1 Introducción 57
5.2.2 Leds de izquierda y derecha 59
5.2.3 Posición, número de vueltas, y reposo del volante 60
5.2.4 Media y desviación 64
5.2.5 Rangos de variación de los giros 68
5.2.6 Pasos por cero (Zero-Crossings) 70
5.2.7 Valores máximos de los giros 73
5.2.8 Media y Desviación típica sobre total de muestras 76
5.2.9 Resumen de datos extraídos con LABVIEW 77
6. ANDROID 79
6.1 La aplicación base 79
6.2 Nuestra aplicación Android 81
6.2.1 Introducción 81
6.2.2 Leds de izquierda y derecha y manejo de datos 85
6.2.3 Posición y vueltas completas del volante 87
6.2.4 Media y desviación 88
6.2.5 Rangos de variación de los giros 90
6.2.6 Valores máximos de los giros 92
6.2.7 Pasos por cero (Zero-crossings) 93

6.2.8 Resumen de datos extraídos con Android........................... 95
7. UNITY... 96
7.1 Introducción a nuestro desarrollo Unity..................................... 96
7.2 Comunicación Unity-Android: primer acercamiento.................. 96
7.2.1 Primer Paso: Comunicación Unity-Unity.............................. 98
7.2.2 Segundo Paso: Comunicación Unity-Android.................... 108
7.3 Acercamiento definitivo a la comunicación Unity- Android: Sockets... 116
7.3.1 Primer Paso: Comunicación Unity-Unity definitiva............ 118
7.3.2 Segundo Paso: Comunicación Unity-Android definitiva, con Socket... 122
7.4 Implementación de la conexión sobre simulador..................... 138
8. PRESUPUESTO DEL PROYECTO... 148
9. CONCLUSIONES Y LÍNEAS FUTURAS................................... 150
BIBLIOGRAFÍA... 156
ANEXO I... 159
Proceso de emparejamiento... 159
ANEXO II... 161
Importar proyecto en Eclipse... 161

ÍNDICE DE FIGURAS

Figura 1. Número de fallecidos por accidente de tráfico 7
Figura 2. Desglose accidentes 2015 10
Figura 3. Comparativa de tiempos de reacción con fatiga 13
Figura 4. Resultado de estudio de síntomas sufridos por conductores . 14
Figura 5. Sensores en un coche. 19
Figura 6. Representación matricial de una imagen a color 21
Figura 7. Sistema de reconocimiento facial de Continental. 22
Figura 8. Sistema de detección de fatiga de Bosch 23
Figura 9. Sistema Iveco de detección de patologías del sueño 25
Figura 10. Sensores Shimmer 27
Figura 11. Unidad básica de Shimmer 29
Figura 12. Shimmer Dock v.2 30
Figura 13. Módulo ECG y colocación de electrodos 31
Figura 14. Módulo GSR y colocación 32
Figura 15. Módulo EMG y su colocación. 33
Figura 16. Módulo 9DOF. 34
Figura 17. Giróscopo 34
Figura 18. Interfaz de Shimmer 9DOF Application 36
Figura 19. Panel frontal 40
Figura 20. Paleta de controles 41
Figura 21. Diagrama de bloques y panel frontal correspondiente 41
Figura 22. Paleta de funciones 42
Figura 23. Acceso a esquema. 44
Figura 24. Esquema librería Shimmer de LABVIEW 45
Figura 25. Uso de sistemas operativos móviles según países 46
Figura 26. Arquitectura de Android 49
Figura 27. Android Studio 50
Figura 28. IDE Eclipse Mars.2 50
Figura 29. IDE de Unity 52
Figura 30. Panel frontal app inicial 53
Figura 31. Diagrama estado sensores 54
Figura 32. Instrucciones de uso simultáneo 56

Figura 33. Panel frontal giróscopo inicial 56
Figura 34. Bloque Shimmer para LABVIEW 57
Figura 35. Data Cluster.. 58
Figura 36. Nuestro diagrama de bloques.. 58
Figura 37. Diagrama de bloques y panel frontal, datos y leds 59
Figura 38. Shift Registers LABVIEW... 60
Figura 39. Wait Until Next ms Multplie 61
Figura 40. Diagrama de bloques, posición del volante 62
Figura 41. Diagrama de bloques, vueltas completas 62
Figura 42. Media y Desviación en LABVIEW.. 66
Figura 43. Array de entrada a media y desviación................................ 66
Figura 44. Panel frontal de media y desviación sobre N muestras 67
Figura 45. Diagrama flujo de rangos.. 68
Figura 46. Diagrama de bloques de rangos.. 69
Figura 47. Panel frontal de rangos... 69
Figura 48. Paso por cero 70
Figura 49. Vi de pasos por cero... 70
Figura 50. Diagrama de bloques de pasos por cero 72
Figura 51. Panel frontal de pasos por cero 72
Figura 52. Vi de máximos absolutos.. 74
Figura 53. Diagrama de bloques de máximos 75
Figura 54. Panel frontal de máximos 75
Figura 55. Media y desviación sobre muestras totales 76
Figura 56. Diagrama de bloques de media y desviación total................ 77
Figura 57. Pantalla incial ShimmerApp base 79
Figura 58. Manejo de ficheros en app base.. 80
Figura 59. ManageDevices en la app base... 81
Figura 60. Botón de selección de actividad 82
Figura 61. Estructura de datos en librería de Shimmer.......................... 84
Figura 62. Leds en Android.. 86
Figura 63. Layout con posición y vueltas del volante............................ 88
Figura 64. Media y desviación en Android 90
Figura 65. Rangos de giro en Android 91
Figura 66. Valores máximos de giro en Android 93

Figura 67. Gráficas de máximos y pasos por cero en Android 94
Figura 68. Comunicación Unity-Android: primer acercamiento.............. 97
Figura 69. Comunicación con Master Server de Unity........................... 99
Figura 70. Botón app Unity de prueba 100
Figura 71. Botón en app Unity cliente.. 101
Figura 72. Server en app Unity PC 103
Figura 73. Conexión Unity establecida en app puente 104
Figura 74. Botón de RPC.. 104
Figura 75. Esquema Remote Procedure Call 105
Figura 76. Añadir “Component” a GameObject 106
Figura 77. Marcar como librería en Eclipse 109
Figura 78. Añadir librería en Eclipse.. 109
Figura 79. Añadir librería Classes.jar en Eclipse 110
Figura 80. Exportar JAR en Eclipse.. 111
Figura 81. Esquema de comunicación final Unity-Android................... 117
Figura 82. Aplicación puente definitiva 121
Figura 83. App Unity de prueba final 121
Figura 84. Esquema segundo paso comunicación Unity-Android definitiva.. 124
Figura 85. Interfaz actividad de comunicación en app Android............ 125
Figura 86. Resumen de comunicación en Android entre servicio y actividad principal 129
Figura 87. Interfaz Android, start.-all llamado 130
Figura 88. Interfaz actividad Android con envío de ficheros 135
Figura 89. Interfaz app de prueba en PC tras recibir ficheros.............. 136
Figura 90. Ejemplo de fichero de datos 137
Figura 91. Conexión a Master Server de Unity en simulador............... 139
Figura 92. IP registrada en simulador... 140
Figura 93. Conectar socket al comenzar simulador............................ 141
Figura 94. Pausa en simulador y en Android...................................... 142
Figura 95. Simulador recibiendo datos con barra de progreso 146

ÍNDICE DE TABLAS

Tabla 1. Tabla de estados de sensores 55
Tabla 2. Datos del artículo de Xuesong Wang, Chuan Xu (2015) 63
Tabla 3. Resumen datos extraídos y ficheros....................................... 78

1. INTRODUCCIÓN

Para introducir el presente proyecto, primero trataremos los objetivos y motivaciones para la realización del mismo. Describiremos también las fases y métodos que hemos seguido para su realización, así como los medios empleados.

Además, se comentará la estructura que seguirá el resto del documento, describiendo de forma resumida sobre qué tratará cada uno de los siguientes apartados.

1.1 Motivación y objetivos

Como veremos en el segundo apartado del proyecto, en la última década se ha mejorado en gran medida la seguridad y las cifras de siniestralidad en las carreteras. A pesar de ello, continuar mejorando y reduciendo las cifras de mortalidad y accidentes viales es un objetivo plenamente actual, al que se destinan bastantes recursos.

Veremos que, uno de los factores más determinantes en los accidentes de conducción es el factor humano, y dentro del mismo, el posible estado de somnolencia o baja alerta del conductor es una de las principales causas de errores humanos en la conducción.

De la mano del avance tecnológico y el desarrollo de sensores de pequeño tamaño, tema que introduciremos también en el segundo apartado del

proyecto, implementar sistemas de monitorización y detección del estado del conductor, que alerten y avisen al mismo de un posible estado de somnolencia con precisión y anticipación, es un objeto de investigación y empleo de recursos notorio en las empresas actuales relacionadas con la automoción. Posteriormente, en el punto dos del proyecto se mostrarán ejemplos de algunos de estos desarrollos de empresas actuales.

En resumen, la realización del presente proyecto se enmarca en este ámbito de investigación de plena actualidad en el sector de la automoción.

Por tanto, los principales objetivos para la monitorización del estado del conductor del proyecto se centrarán en el desarrollo de aplicaciones para tecnologías diversas, que introduciremos más tarde, que extraigan y procesen datos de los sensores fisiológicos de Shimmer, más concretamente:

- Partiendo de la aplicación base desarrollada por Khadmaoui, Amine (2012) [2] para extraer datos mediante LABVIEW de los sensores, profundizar en el procesado de datos del giróscopo, añadiendo, de acuerdo a diversos artículos que veremos, medidas, procesados y elementos a la interfaz de usuario para los datos del giróscopo en particular.
- Partiendo también de una aplicación base desarrollada por Khadmaoui, Amine (2014) [3] para extraer los datos de los sensores mediante una aplicación para dispositivos Android, mejorar de manera general el procesado y visualización de datos de los sensores al usuario, e implementar, más concretamente, los mismos procesados y medidas de datos del giróscopo que hicimos para LABVIEW, además del almacenamiento de los mismos.
- Trabajando con Unity, desarrollar código y *scripts* extrapolables a cualquier simulador de conducción creado con este motor gráfico, que implementen la comunicación con la aplicación Android, para sincronizar el estado de transmisión de la misma con los sensores desde el simulador de conducción, además de transferir los datos extraídos por la aplicación Android al terminal o PC en el que

estemos ejecutando el simulador. Para esta comunicación veremos que se empleará una aplicación puente que desarrollaremos con Unity para dispositivos Android.

En resumen, para tratar de desarrollar métodos efectivos para monitorizar el estado de alerta del conductor, los objetivos principales del presente proyecto son trabajar con los sensores Shimmer que detallaremos en el punto tres del proyecto, específicamente con el giróscopo, sobre los proyectos base de Amine Khadmaoui [2] y [3], para ampliar la información obtenida de éste y el procesado de cálculos estadísticos sobre dicha información. Además de desarrollar un sistema de comunicación Unity-Android, para comunicar la aplicación Android que extrae los datos con cualquiera de los simuladores de conducción en Unity.

1.2 Fases del proyecto

Las fases que vamos a seguir para realizar el proyecto presente son:

- ✓ Estudio breve sobre seguridad vial, mostrando datos recientes y la tendencia de los mismos en años recientes, respecto a la mortalidad y siniestralidad en vías urbanas e interurbanas.
- ✓ Referenciar algunos sistemas actualmente desarrollados por empresas del sector de la automoción para monitorizar el estado del conductor.
- ✓ Análisis e introducción en el manejo de los sensores fisiológicos Shimmer en su versión 2.
- ✓ Análisis y estudio de las tecnologías de programación gráfica como LABVIEW.
- ✓ Análisis y estudio de sistemas operativos móviles actuales, Android.
- ✓ Análisis y estudio de motores gráficos de videojuegos, Unity.

- ✓ Análisis del proyecto base para LABVIEW, desarrollo de las nuevas implementaciones y procesados centrados en el giróscopo.
- ✓ Análisis del proyecto base para Android, desarrollo de las nuevas implementaciones y procesados centrados en el giróscopo.
- ✓ Desarrollo del método de comunicación de la aplicación Android con simuladores de conducción Unity. Desarrollo de aplicación puente con Unity para dispositivos Android basándose en *sockets* tras intentar otros métodos fallidos.
- ✓ Análisis de datos obtenidos, conclusiones y líneas futuras.
- ✓ Elaboración de un presupuesto económico básico del coste real del proyecto presente.

1.3 Medios y elementos empleados

Para la realización del proyecto, los medios principales empleados fueron:

- Sensores *Shimmer* V2. (Ver siguiente capítulo).
- Ordenador portátil *HP Elitebook 8460p*; procesador *Intel Core i5-2540M* a 2.60 GHz y 6 GB de RAM.
- Móvil Android *Huawei P8*; procesador 8-core 64bit *HiSilicon Kirin 935* a 2.0GHz y 3 GB de RAM.
- Entornos de desarrollo software necesarios para LABVIEW, Android y Unity, además de las librerías Shimmer y otras necesarias para LABVIEW de pago.
- Material de apoyo detallado en la bibliografía.
- Recursos y medios facilitados para las pruebas por el laboratorio del Grupo de Telemática e Imagen que pertenece al Departamento de

Teoría de la Señal y Comunicaciones e Ingeniería Telemática de la ETSI de Telecomunicación de la Universidad de Valladolid. Un ejemplo de estos recursos es el volante con pedales *Logitech G27*.

1.4 Estructura de la memoria

Esta memoria se va a componer de varios apartados que tratarán de exponer, tanto información general para poner al lector en situación dentro del tema de la monitorización al conductor, como de explicarle qué soluciones e implementaciones se han realizado y cuáles han sido las razones para hacerlo de la forma en que se ha hecho.

En este primer capítulo hemos visto una introducción sobre los objetivos y motivaciones del proyecto, así como de las fases y medios empleados.

En el segundo capítulo trataremos el tema de la seguridad vial, expondremos datos recientes e introduciremos el problema de la somnolencia al volante y la tecnología de sensores y su desarrollo actual y ejemplificar con algunos sistemas actuales de monitorización del conductor.

En el tercer capítulo introduciremos el empleo de los sensores fisiológicos de Shimmer.

En el cuarto capítulo haremos un análisis e introducción de las distintas tecnologías empleadas para tratar datos de los sensores y monitorizar fisiológicamente al conductor; sistemas de programación gráfica como LABVIEW, plataformas móviles como Android y motores gráficos de videojuegos como Unity.

En el quinto capítulo detallaremos el desarrollo de las implementaciones hechas para la aplicación LABVIEW.

En el sexto capítulo detallaremos el desarrollo de las implementaciones hechas para la aplicación Android.

En el séptimo capítulo trabajaremos con Unity y simuladores de conducción y explicaremos la comunicación Android-Unity desarrollada.

El octavo capítulo consiste en un desglose del coste económico del proyecto, teniendo en cuenta tiempos de amortización, de materiales y horas de trabajo.

El capítulo final consta de las conclusiones últimas y las líneas futuras del proyecto.

2. SEGURIDAD VIAL, SENSORES Y TECNOLOGÍA

2.1 Seguridad vial y accidentes

Según estudios de DGT y Fremap [4], el factor humano es la causa principal de accidentes de tráfico en España, con un porcentaje estimado entre el 70% y el 90%.

La seguridad vial y siniestralidad en las carreteras siempre ha sido motivo de preocupación pública debido al uso cuasi diario que la mayoría de la sociedad hace del coche como medio de transporte principal. En los últimos años, los esfuerzos de concienciación y campañas han mejorado la seguridad vial y reducido el número de accidentes en España, como se aprecia en la

Figura 1. Número de fallecidos por accidente de tráfico, años 2011-2016

figura.

Podemos ver una clara línea descendente, como ya hemos dicho, en el número de fallecidos por accidente de tráfico en las carreteras en los últimos siete años, gracias presumiblemente a todas las medidas y campañas de concienciación vial implantadas. A pesar de ello, observamos que en el último año, 2016, se registraron 1160 fallecidos en accidentes de tráfico, cifra más alta que en el año 2015 anterior. Este aumento en fallecidos no ocurría desde hace trece años. Y es que en total en 2016 se han registrado 1.038 accidentes mortales en vías interurbanas que se han cobrado la vida de 1.160 personas, y esto corresponde con 15 accidentes y 29 muertos más respecto a 2015, invirtiéndose así la tendencia positiva en disminución que se había conseguido. Las razones de este aumento de fallecidos en el último año, como ha apuntado la DGT, son entre otras:

- Un aumento del 5% en el número de viajes de larga distancia, lo que supone 18,6 millones de viajes más respecto a 2015, hasta un total de 392 millones en 2016.

- En 2016, y por efectos de la crisis económica, también se ha producido un aumento de la antigüedad media del parque de vehículos implicados en accidentes de tráfico: 13,6 años de media tenían los turismos en los que viajaban los fallecidos, 11,1 en el caso de las furgonetas y 9,5 años las motos. Se observa que, a pesar de las ayudas del Plan PIVE, no se ha conseguido rejuvenecer el parque de vehículos lo suficiente, y este año pasado se ha cerrado con una antigüedad media alarmante que roza los 12 años (según datos de ANFAC, Asociación Nacional de Fabricantes de Automóviles).

- También se han cometido más infracciones relacionadas con el consumo de drogas ilegales. En 2016 la DGT ha realizado 60.942 pruebas, de las cuales 23.822 fueron positivas (el 39%). Llama la atención que en los controles preventivos el 38% dieron positivo, una cifra bastante alta.

- Otro factor clave en el aumento de la siniestralidad tiene su origen también en la crisis económica, y lo encontramos en el estado de las carreteras, cuya inversión en mantenimiento por parte del Gobierno y Comunidades Autónomas se ha reducido al mínimo en los últimos años, tanto que la Asociación Española de la Carretera cifra el déficit de conservación en 6.617 millones de euros, un 7% más que en 2013.

A pesar de los datos de 2016, España se encuentra entre los países de la Unión Europea con menor índice de siniestralidad: 36 muertos por millón de habitantes, muy por debajo de la tasa media en Europa, que está en 52. Y en líneas generales la mejora en las cifras conseguida en las últimas décadas es considerable.

En cualquier caso, el número de fallecidos y accidentes en las carreteras sigue siendo bastante significativo, como podemos ver en este extracto del anuario estadístico de accidentes de 2015, el último elaborado por la DGT en el momento de realización del proyecto [6].

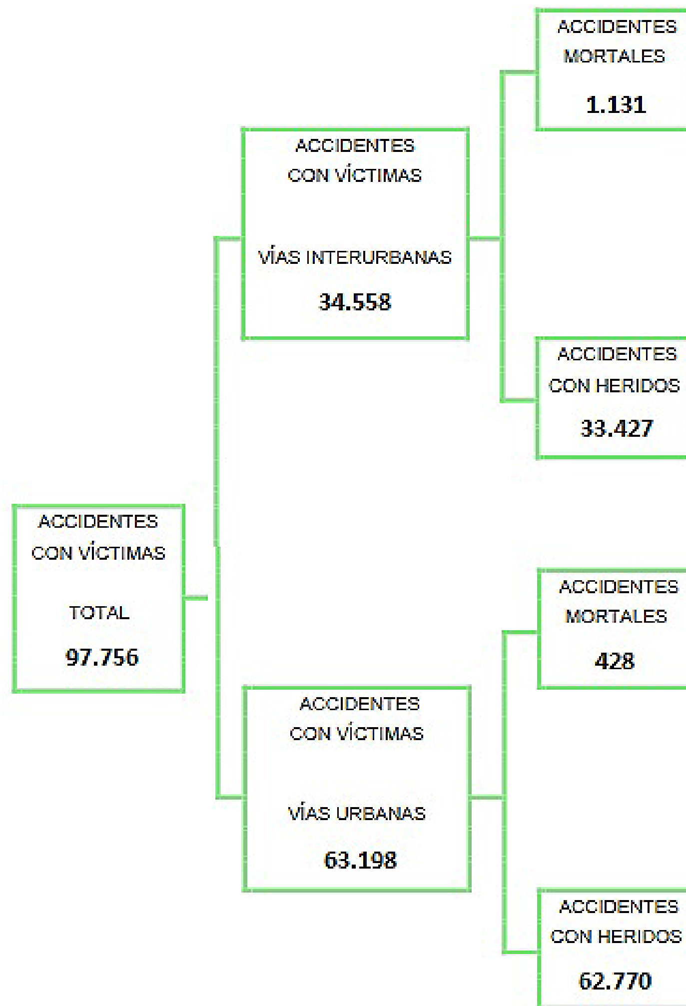


Figura 2. Desglose accidentes 2015

Por lo tanto, podemos afirmar que la siniestralidad en las carreteras es un tema de preocupación e investigación absolutamente actual.

Nosotros nos centraremos en el presente proyecto en una de las principales causas atribuidas a estos accidentes por factor humano que mencionamos anteriormente; la fatiga, cansancio y distracción del conductor. En resumen, analizaremos el bajo nivel de vigilancia del conductor [24], por ser una variable que intentaremos medir y controlar mediante la acción combinada y datos recogidos de los distintos sensores, aunque nosotros nos enfocaremos

en el giróscopo, como veremos y explicaremos en capítulos siguientes.

2.1.1 Fatiga y somnolencia al volante

La fatiga y la somnolencia, o hipersomnia idiopática [14], son palabras que se usan para describir dos condiciones similares; en ambos casos el conductor se ve afectado física y psíquicamente: disminuye la capacidad de atención, favorece las equivocaciones al ejecutar las maniobras y obliga a asumir más riesgos. Todo esto son factores de riesgo en la conducción que se materializan en accidentes.

La principal causa de la fatiga es no descansar o no hacerlo de una forma adecuada. Sin embargo los factores que influyen, condicionan la aparición de esta circunstancia y potencian su efecto, haciendo que la fatiga aparezca antes o sea más intensa, son muy variados. Algunos de estos factores que, como veremos, intentaremos reproducir en el simulador desarrollado en *Unity*, pertenecen al vehículo, otros a la vía y el entorno y otros al propio conductor. Pasamos a detallarlos:

Factores externos: La vía y el entorno

- Circular por una vía con una elevada densidad de tráfico, en la que te ves sometido a frecuentes retenciones y paradas, requiere aumentar la atención necesaria, lo que puede potenciar la fatiga.
- Conducir por una vía poco conocida también produce un efecto similar.
- Algunos tipos de firme hacen vibrar el vehículo, por lo que la conducción será más incómoda, difícil y cansada.
- Las condiciones climatológicas adversas también requieren aumentar la atención y, por tanto, potencian la fatiga. En el simulador de *Unity* tendremos un escenario con un anochecer para reproducir este efecto.

Factores del vehículo

- Una mala ventilación o una temperatura elevada pueden hacer más incómoda la conducción y alterar el estado del conductor.
- Una iluminación deficiente, si circulas por la noche, puede hacer más

difícil la conducción y requerir de ti un mayor nivel de atención sobre la conducción y aumentar la fatiga.

- El mal estado del vehículo como, por ejemplo, un ruido excesivo del motor o las vibraciones por defectos en la dirección o en la suspensión, pueden hacer que la conducción resulte más incómoda y potenciar la fatiga.

  Aunque estos factores sean más complicados de reproducir en la simulación, en cuanto a la vibración de la dirección, el volante de *Logitech* disponible para las pruebas incluye una emulación configurable de este efecto [10].
- Un diseño poco ergonómico del asiento o de otros elementos del vehículo también puede hacer más fatigosa la conducción.

Factores del conductor

- Conducir durante largos periodos, no parar a descansar, o hacerlo de una forma insuficiente o inadecuada, es una de las principales causas de la fatiga al volante.
- La prisa por llegar o mantener una velocidad excesiva durante mucho tiempo exige una mayor concentración que puede alterar el estado psicofísico del conductor.
- Conducir estando ya fatigado por las actividades realizadas antes de coger el vehículo.
- Conducir con hambre también requiere un mayor esfuerzo y cansancio en la conducción.
- El alcohol, las comidas copiosas, enfermedades, o el estrés, pueden alterar el estado del conductor y hacer más incómoda la conducción. Estos factores pueden ser más mensurables mediante el uso de sensores, como el electrocardiograma (ECG), los cuales explicaremos más detalladamente en el capítulo tres.
- Cambios en los hábitos normales de conducción, por ejemplo, conducir de noche cuando no se está habituado a hacerlo, exigen también un mayor esfuerzo en la conducción.

- Una postura incómoda del conductor también hace más fatigosa la conducción.
- Los conductores noveles, al no haber automatizado todavía muchos de los procesos implicados en la conducción, se fatigan con mayor facilidad.

Por tanto, como ya hemos dicho, la fatiga es uno de los factores de riesgo más importantes y peligrosos; altera la visión del conductor, su audición, movimientos, comportamiento, y toma de decisiones. En consecuencia, reconocer sus síntomas en cuanto empiezan a aparecer, como pretendemos hacer con los sensores fisiológicos, puede ser una acción que reduzca en gran medida las cifras de accidentes en las carreteras.

Una manera muy gráfica de analizar las consecuencias de la fatiga es comparar los tiempos de reacción necesarios, y ver cómo aumentan en presencia de esta, como podemos ver en la figura 3.

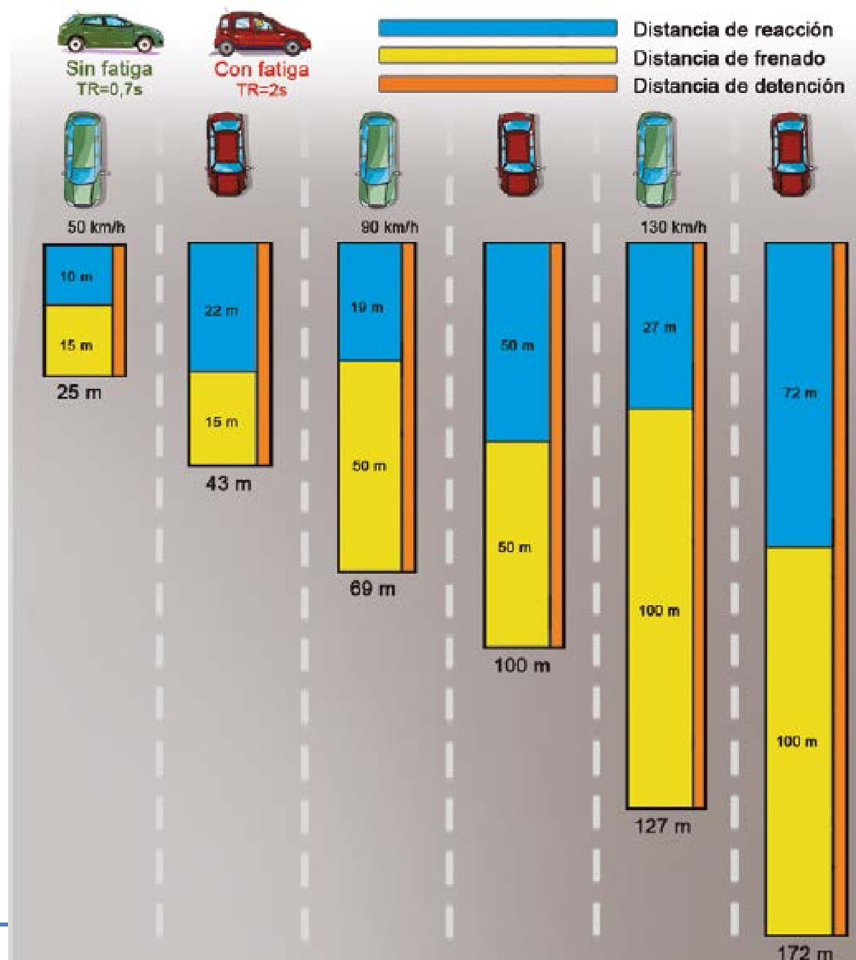


Figura 3. Comparativa de tiempos de reacción con fatiga

En resumen, tras lo expuesto podemos afirmar que la fatiga es uno de los mayores factores de riesgo al volante, debido a los múltiples puntos que hemos visto, que influyen en su aparición, y debido también a la gravedad de sus consecuencias.Para situar esta gravedad en datos, de acuerdo con un estudio elaborado por RACE [15]: “El 30% de los conductores españoles ha sufrido al volante una situación de riesgo por fatiga o sueño”. Lo cual supone una cifra más que significativa, más aun teniendo en cuenta la cantidad de desplazamientos que se realizan, por ejemplo, en los meses de verano, con millones de desplazamientos de corto y largo recorrido.

Según ese mismo estudio, la fatiga y somnolencia al volante suponen la cuarta causa de mortalidad en las carreteras españolas, por encima de otras como el alcohol, las drogas, o incluso los factores climatológicos.

Concretamente, y de acuerdo a este mismo estudio, desde el 2007 hasta el 2011, en España se registraron 683 accidentes mortales por fatiga y/o somnolencia, con un resultado de 784 víctimas mortales.

Además, en un estudio reciente (2015) [5], elaborado por CEA (Comisariado Europeo del Automóvil), se afirma que: “Un 71,65% de los conductores ha sentido somnolencia al volante”, y “un 59,22% reconoce haber sufrido micro-sueños”. Así que, como se muestra en la figura 4 extraída del estudio mencionado, la somnolencia es uno de los síntomas que los conductores afirman haber sufrido más recurrentemente al volante.

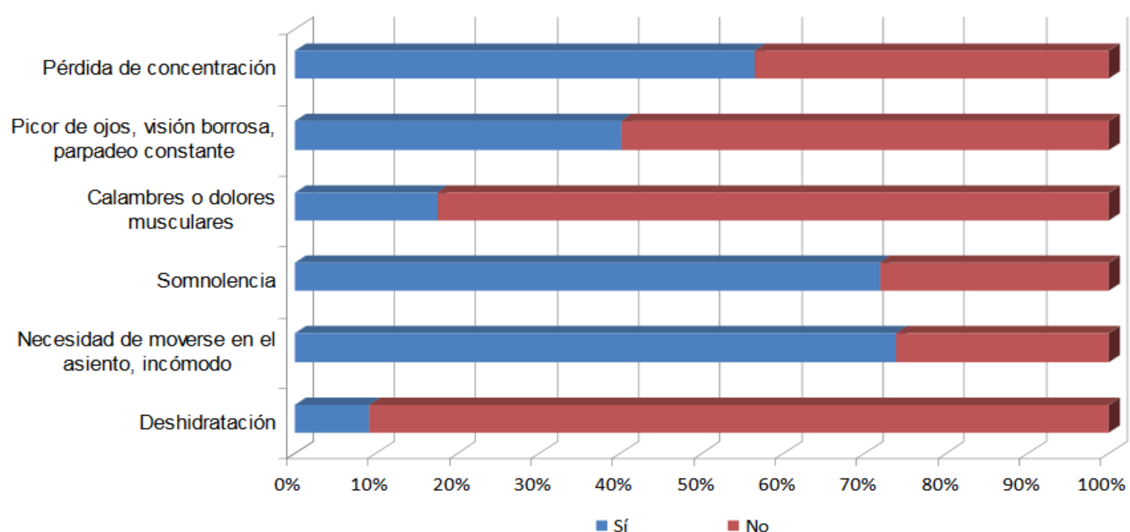


Figura 4. Resultado de estudio de síntomas sufridos por conductores

En otros países del mundo la fatiga y la somnolencia al volante es, lógicamente, una cuestión igual de preocupante.

Así, en Estados Unidos, varios estudios realizados por NHTSA (National Highway Traffic Safety Administration) estiman que se producen al año unos 56.000 accidentes provocados por la fatiga y/o somnolencia. Una investigación americana sobre la conducción profesional [13], señalaba que entre el 30% y el 40% de los accidentes con camiones pesados fueron causados por fatiga y/o somnolencia.

En Australia, el porcentaje es similar, ya que VicRoads (la Organización de Seguridad Vial Australiana), afirma que entre el 25 y el 35 por ciento de los accidentes son causados por la fatiga al volante.

En Noruega se realizó una encuesta a 9.200 conductores implicados en accidentes de tráfico, y se estimó que el 3,9% de los accidentes se había producido por somnolencia, y que en el 20% de los casos de accidentalidad nocturna estaba involucrada la fatiga y/o somnolencia.

En el Reino Unido, los porcentajes son de igual manera similares, ya que en estudios realizados se afirma que un 20% de los accidentes en carreteras son causados por la fatiga, lo que significa que miles de accidentes que se producen cada año en este país tienen su origen en esta causa.

Vemos, por tanto, que el patrón que siguen los datos es muy parecido para la mayoría de países que podamos considerar. Por ello, para resumir la importancia del objetivo del presente proyecto, utilizar los avances tecnológicos para detectar y prevenir la aparición de fatiga al volante, y no extendernos con más datos, podemos mencionar estudios realizados por ETSC (European Transport Safety Council), los cuales afirman que la fatiga al volante causa casi el 20% de los accidentes de vehículos comerciales en la Unión Europea, por lo que si se consiguiera reducir y paliar este factor, podríamos alcanzar el ambicioso objetivo planteado por la Unión Europea en 2015; reducir las muertes por accidente en nada menos que un 50%.

A tenor de los datos mostrados, podemos intuir la importancia y esfuerzos actuales destinados a reducir y detectar la aparición de fatiga y somnolencia al volante. Esto, unido al desarrollo y miniaturización de los dispositivos tecnológicos (como expusimos en la introducción), ha resultado en sistemas de detección de fatiga que ya son incorporados de forma embebida en muchos modelos de coches de fabricación reciente, además de inversiones de empresas de automoción para desarrollar sistemas de estas características.

2.2 Introducción a los sensores

En el ámbito tecnológico, la evolución y desarrollo de la electrónica desde sus comienzos, ha conllevado una disminución progresiva del tamaño de los chips y elementos hardware producidos.

Actualmente, esta evolución en la miniaturización de los componentes ha posibilitado la fabricación de elementos con capacidad de procesamiento y conectividad, sistemas embebidos miniaturizados y con estas capacidades, conocidos popularmente como sensores.

Los sensores han abierto toda una vía de aplicaciones tanto militares como civiles, que actualmente está explorándose de múltiples formas [9]: desde las redes inalámbricas de sensores inteligentes *(WSN: Wireless Sensor Network)*, que permiten formar redes *ad hoc* sin infraestructura física preestablecida ni administración central, con aplicaciones en uso ya en distintos campos tales como entornos industriales, domótica, entornos militares o detección ambiental hasta la monitorización fisiológica de seres humanos, en la que nos centraremos en el presente proyecto y detallaremos más adelante.

La monitorización de las constantes fisiológicas ha sido siempre un factor clave en medicina, para el tratamiento y seguimiento del estado clínico

del paciente. En este sentido, el desarrollo de la tecnología y la miniaturización de los sensores mencionados; con su reducido tamaño y su capacidad para obtener y almacenar información que nosotros no percibimos como las ondas electromagnéticas o los ultrasonidos con gran precisión, ha abierto muchas posibilidades para parametrizar de manera no invasiva constantes vitales principales como la frecuencia cardíaca, la frecuencia respiratoria, la presión arterial, la saturación de oxígeno o la temperatura corporal periférica, por ejemplo.

De hecho, el seguimiento y monitorización remota de pacientes clínicos y ancianos mediante dispositivos tecnológicos e Internet, se está postulando como un nuevo paradigma de asistencia médica, que previsiblemente, como afirma un estudio de IESE (Instituto de Estudios Superiores de la Empresa) (2013) [8], mejoraría la calidad de la atención, animaría a los pacientes a cuidar más su salud y a detectar precozmente los cambios de sus afecciones crónicas, y además ayudaría a reducir la aglomeración de pacientes y las visitas rutinarias al hospital.

En esta última línea de investigación, nos fijaremos en la aplicación de esta monitorización fisiológica a una cuestión siempre de actualidad: la siniestralidad en las carreteras por accidentes de tráfico, estableciendo así mediante estos sensores, una posible monitorización del estado del conductor, que conduzca a una mejora en la seguridad vial para todos.

2.3 Tecnología y sensores en vehículos actuales

Con el paso del tiempo y los avances tecnológicos, los coches han estado fabricándose con un número cada vez más creciente de dispositivos para velar por la seguridad: Airbag, ABS, ESP, etc.

Sin embargo, el desarrollo de tecnologías que protejan al conductor del sueño

o la fatiga, que tratamos en el presente proyecto, son un nicho tecnológico muy reciente, que se encuentra en sus inicios arrancando actualmente [5].

Así, se pretende, y se dispone ya, de sistemas que en caso de despiste o distracción del conductor puedan entrar en acción de forma automática para evitar un accidente o reducir sus consecuencias. Podemos encontrar aquí por ejemplo, desarrollados ya, sistemas de frenado automático que pueden frenar el coche por si mismos de acuerdo a la información recibida desde cámaras y sensores, para evitar un accidente en caso de detectarse que el conductor no está frenando a tiempo. También encontramos, en este grupo, asistentes de mantenimiento en carril, que son capaces de hacer girar la dirección del coche y evitar en cierta medida que este se desvíe de su carril de circulación.

En la figura 5 podemos ver algunas de estas funciones de los sensores en un coche, como el *Emergency Braking* (frenado de emergencia), del que hemos hablado.

Pero nosotros nos centraremos en dispositivos para proteger nuestra seguridad de una manera distinta, sistemas de naturaleza más indirecta y preventiva, que sirven para monitorizar al conductor y controlar en qué estado está conduciendo. Así con estos sistemas pretendemos ser capaces de detectar si el conductor se encuentra somnoliento o fatigado o se está durmiendo al volante, o incluso reconocer si el conductor está distraído o mirando a otra parte. Para ello tenemos que hacer uso de los sensores que hemos introducido al principio. Estos podrán ir colocados en distintas partes del coche; en el presente proyecto nos centraremos en el uso del giróscopo que, como a continuación explicaremos, se sitúa en el volante del coche.

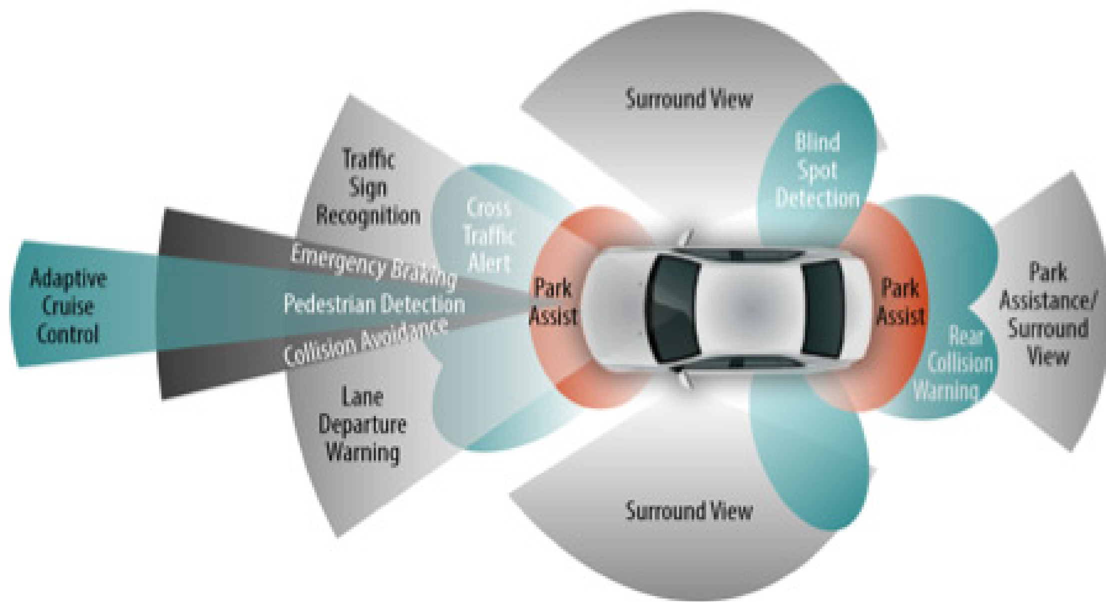


Figura 5. Sensores en un coche.

2.3.1 Sensores en el volante

En nuestro caso, el giróscopo se colocará en el volante, para detectar los giros de este y las variaciones en posición o velocidad de los mismos. Esto lo veremos más adelante.

Hablando de los sistemas de detección de fatiga y sueño comercializados que ya están disponibles, estos actualmente ya se incluyen de serie en algunos vehículos, no necesariamente de lujo.

El funcionamiento de este tipo de sistemas se basa en sensores instalados en el coche, que podrían ser giróscopos como el que nosotros usaremos, para aprender las pautas conductuales del manejo del volante que hace el conductor en condiciones normales, y así controlar cuándo nos desviamos de estas pautas para detectar un posible estado inusual del conductor. Así, cuando el coche detecte esto, podría, por ejemplo, mostrarnos un mensaje en el ordenador de a bordo indicándonos que paremos a descansar.

Estos sistemas serán el objeto de estudio de este proyecto utilizando el

giróscopo de Shimmer, así que más adelante entraremos más en detalle.

Aun así, cabe mencionar también otros sistemas que ya se usan en coches para estos fines, como los sistemas de visión artificial.

2.3.2 Visión artificial: cámaras de reconocimiento facial

La visión artificial es un subcampo de la inteligencia artificial que pretende emular la capacidad natural de los seres vivos para ver una escena, entenderla y actuar en consecuencia.

Esto involucra también al procesamiento digital de imágenes (PDI), con distintas fases como: adquisición de la imagen, procesamiento, e interpretación.

Aunque no es el principal cometido de este proyecto, podemos dar una idea sobre el funcionamiento de estos sistemas; una imagen en escala de grises es interpretada como una matriz bidimensional de nxm elementos, donde cada elemento tiene un valor de 0 (negro) a 255 (blanco). Mientras que para una imagen en color (RGB), tenemos una matriz tridimensional o array de matrices nxmxp donde 'p' representa la matriz o plano correspondiente para cada uno de los tres colores (Red Green Blue) [1], como se ilustra en la figura 6.

Esto se correspondería con una visión artificial bidimensional, para inferir las propiedades tridimensionales de esas imágenes, es decir, dotarlas de una visión en tres dimensiones con profundidad, como hace de manera natural el ojo humano, tenemos que situar en el espacio cada punto de los anteriores. Una forma de hacer esto es, por ejemplo, proyectar sobre la imagen microondas o ultrasonidos y medir el tiempo de reflexión, para situar cada punto de la imagen en el espacio.

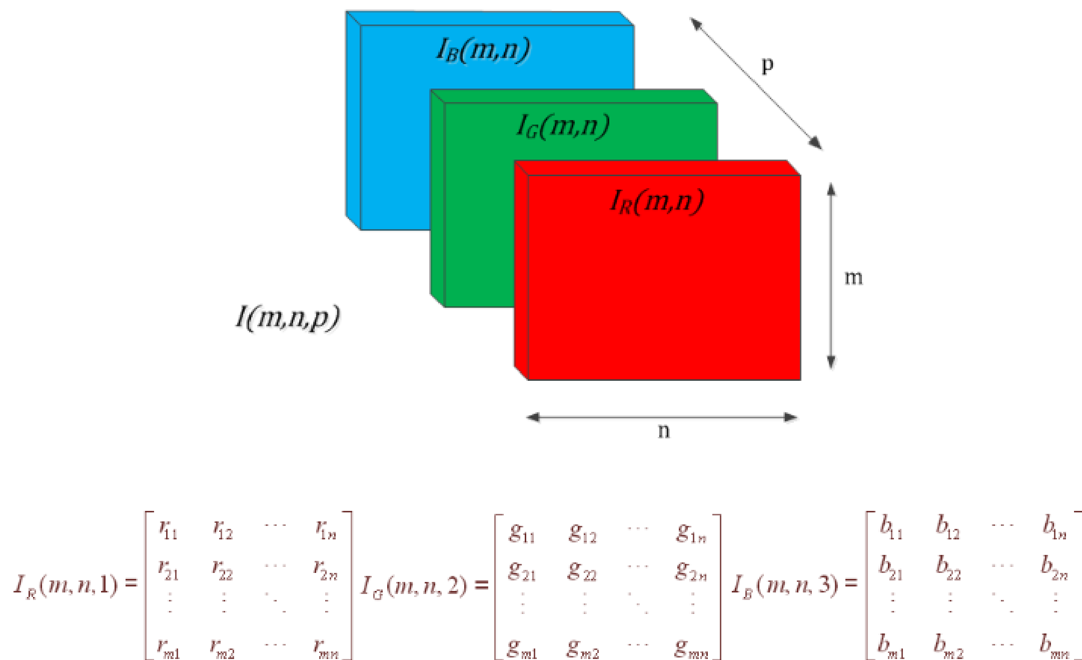


Figura 6. Representación matricial de una imagen a color

Así, el funcionamiento de este tipo de dispositivos para el coche se fundamenta en el uso de una cámara, que típicamente se coloca sobre el volante y un sistema de reconocimiento facial mediante esta visión artificial.

De esta manera, gracias a las arrugas y expresión de nuestra cara, se pretende que el coche sea capaz de reconocer nuestros signos de cansancio o sueño y tomar medidas al respecto. También, puede supervisar los ojos del conductor, para comprobar si el parpadeo es normal o indica sueño, además de ver si el conductor mira al frente o desvía la mirada fuera de la carretera durante un período muy largo de tiempo.

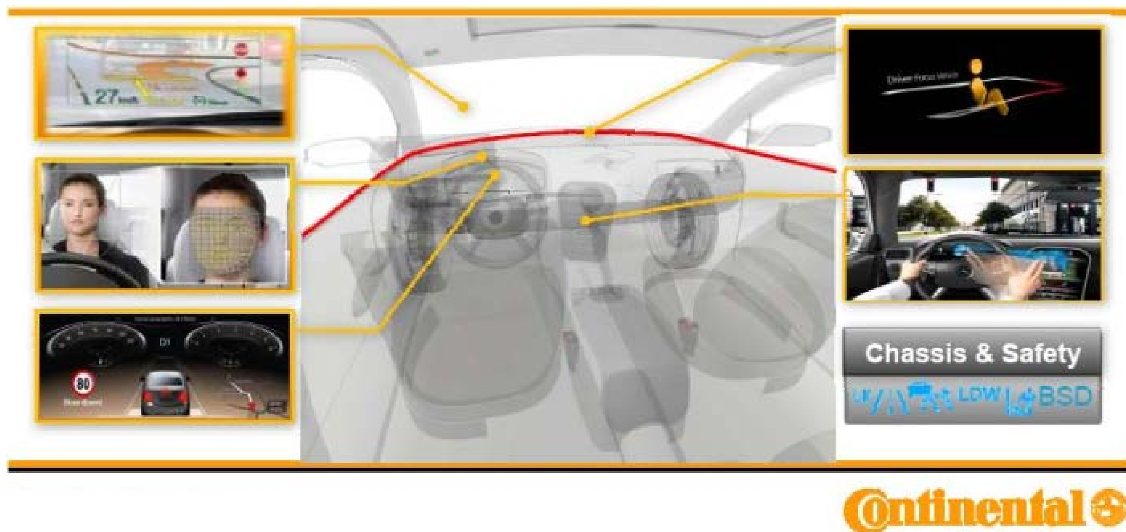


Figura 7. Sistema de reconocimiento facial de Continental.

Como hemos dicho, los fabricantes empiezan a concienciarse con el problema de la somnolencia, y algunas empresas como Continental, Bosch, PSA Peugeot Citroën, Ford, Toyota, o BMW ya desarrollan sistemas para controlar el nivel de vigilancia del conductor.

2.3.3 Empresas y sistemas de monitorización actuales

[5]

- **Continental Driver Focus**

Es un sistema desarrollado por CONTINENTAL para reducir los accidentes de tráfico por distracciones y somnolencia al volante.

El sistema se basa en una cámara de infrarrojos que realiza un seguimiento del movimiento de los ojos del conductor del coche, para detectar así, durante la conducción, distracciones o signos de fatiga. Si se mira hacia otro lugar que no sea la carretera en un tiempo que excede un determinado umbral, se cierran los párpados durante más tiempo del normal para un pestañeo, o se suceden

movimientos bruscos de cabeza, se acciona un mecanismo que activa una luz sutil que lleva a los ojos del conductor de vuelta al campo de visión correcto.

- **Sistema de Bosch**

Bosch también ha desarrollado un sistema de detección de fatiga para detectar la somnolencia del conductor analizando el comportamiento de este al volante y avisándole en caso de que exista un alto riesgo de que se duerma.

Este sistema combina datos recibidos desde el volante con información adicional para determinar la forma en la que se está conduciendo el vehículo. Esto, parecido a lo que nosotros haremos con el giróscopo de Shimmer, tiene una ventaja de costes sobre sistemas basados en video como el anterior, ya que el sensor de ángulo de giro del volante es una parte integrante del sistema ESP®, o la dirección asistida eléctrica con un sensor necesario, ya están instalados como estándar en muchos vehículos.

En este sistema se utilizan estos datos del comportamiento del conductor al volante para identificar lo que se denomina “punto muerto”. Estos puntos muertos son fases en las que el conductor no conduce (no hay movimiento del volante) durante un breve período de tiempo, y que finalizan con una corrección brusca del volante, siendo esto señal inequívoca de que la concentración del conductor en la carretera está disminuyendo.

Figura 8. Sistema de detección de fatiga de Bosch

Este sistema combina la fuerza y frecuencia de estas reacciones con otros datos como la velocidad del vehículo o la hora del día para calcular un índice de cansancio de acuerdo a un determinado algoritmo. Si ese índice supera un umbral establecido, se muestra una señal acústica de advertencia o un testigo luminoso, como se puede ver en la figura 8.

- **Sistema de PSA Peugeot Citroën**

Este grupo automovilístico en colaboración con el Centro de Transporte y Procesamiento de Señales de la Escuela Politécnica Federal de Lausanne (Suiza), trabaja en el desarrollo de una tecnología que emplea una cámara de video y un software de reconocimiento facial, parpadeo, y movimientos musculares para detectar si el conductor está distraído o está sufriendo somnolencia.

- **Otros sistemas**

Toyota también empezó a investigar hace años el reconocimiento facial del conductor, para detectar distracciones y fatiga, así como la agresividad o ansiedad del conductor, y evitar riesgos.

Ford desarrolla por su parte un sistema que difiere algo más de los anteriores. Este es un sistema de control de estrés al volante, que no se centra tanto en las distracciones como en el nivel de ansiedad y estrés del conductor. Para ello se sirve de varios sensores de temperatura, ritmo cardíaco, y ritmo respiratorio del conductor, además de las condiciones del tráfico, de la carretera, y del propio vehículo.

Iveco, junto con la Universidad de Génova, ha puesto en marcha una iniciativa de seguridad vial que busca concienciar sobre la importancia de un adecuado descanso de los conductores. Esta campaña se llama Iveco Check Stop, y está destinada a transportistas y profesionales de la conducción. Para ello cuenta con un vehículo desarrollado y preparado con la tecnología necesaria para exclusivamente llevar a cabo la revisión de todos los signos que ayudan a identificar un trastorno del sueño y detectar esta posible patología en

conductores profesionales. Los conductores pasan una prueba con el fin de detectar problemas como la excesiva somnolencia diurna, el insomnio, o la apnea del sueño.

Figura 9. Sistema Iveco de detección de patologías del sueño

3. SENSORES SHIMMER

3.1 Introducción a los sensores fisiológicos

En la introducción ya hablamos del desarrollo tecnológico que ha supuesto la miniaturización de sensores electrónicos y de las múltiples aplicaciones que estos tienen. Ahora nos centraremos en los sensores que usaremos a lo largo del proyecto; sensores fisiológicos [2].

Como hemos visto, un sensor es un dispositivo capaz de detectar magnitudes físicas o químicas, llamadas variables de instrumentación, y mediante un transductor transformarlas en variables eléctricas, como una corriente o una impedancia. En el caso de los sensores fisiológicos, estas variables de instrumentación están relacionadas con constantes y parámetros del cuerpo humano como, por ejemplo, la señal electrocardiográfica, temperatura, frecuencia cardiaca, etc.

Con el empleo de esta tecnología como apoyo a aplicaciones médicas y cuidado de la salud, con énfasis en lo relacionado con la captura y envío de datos en tiempo real, se ha llegado a la denominación de “telemedicina” para esta disciplina en auge.

Aunque, como ya hemos dicho, estos sensores tienen aplicaciones en diversos ámbitos: militares, donde se están desarrollando proyectos para monitorizar a los soldados, o civiles, con la monitorización de pacientes comentada, o, el caso que nos ocupa, la monitorización del conductor.

Por lo tanto, este es el tipo de sensores que usaremos para el control del nivel de vigilancia del conductor en el presente proyecto, ya que posibilitan el

seguimiento de las constantes en un conductor, para poder analizar los cambios surgidos en estas a la hora de producirse la fatiga o sueño y, por lo tanto, reconocer estos indicios y advertirle al conductor en caso de fatiga o una concentración insuficiente. En concreto, emplearemos los sensores desarrollados por Shimmer®, especialmente el giróscopo.

3.2 Sensores Shimmer

Figura 10. Sensores Shimmer

Shimmer® es una plataforma de sensores inalámbricos desarrollados por la empresa del mismo nombre, que se utilizan en aplicaciones dedicadas a la investigación biomédica.

Estos sensores son dispositivos pequeños, compactos y bastante ligeros (~ 20 g). Estas características hacen que estos sensores sean muy adecuados en aplicaciones de detección cinemática y fisiológica, ya que se pueden colocar fácilmente en el cuerpo humano de manera no muy invasiva y, además, porque son inalámbricos, y se comunican a través de Bluetooth.

3.2.1 Hardware Shimmer

Podemos primero dar un vistazo general de todos los sensores Shimmer que tenemos disponibles, los denominamos módulos de expansión, de ellos podemos obtener y procesar datos:

- Magnetómetro, permite medir un campo magnético, nos permite procesar la emisión electromagnética humana.
- GSR (Galvanic Skin Response, Respuesta Galvánica de la Piel), permite detectar cualquier cambio en la resistencia galvánica de la piel. El miedo o el estrés influyen en ciertas glándulas sudoríparas y estas inducen cambios en la resistencia eléctrica de la piel.
- ECG (Electrocardiography, Electrocardiografía), nos permite procesar la actividad del corazón de manera gráfica, evaluando el ritmo y la función cardiaca a través de la actividad eléctrica del corazón.
- EMG (Electromyography, Electromiografía), permite seguir la actividad eléctrica producida por los músculos, más concretamente por los nervios que controlan los músculos, mediante electrodos.
- Giróscopo, es el que emplearemos principalmente en nuestro proyecto, permite detectar y medir velocidades angulares en los tres ejes o dimensiones espaciales.

Estos sensores constan de una placa base o unidad básica:

Unidad básica de Shimmer

Es común a todos los sensores anteriores. Consta de un sensor de aceleración o acelerómetro (mide la aceleración de un movimiento) en los tres ejes espaciales, un led que nos indica el estado en el que se encuentra el sensor: si está en color verde indica que el dispositivo está conectado, si está en rojo, indica que la batería es débil y necesita recargar, y cuando parpadea en naranja, nos indica que el dispositivo está transmitiendo datos. Consta también de un botón para reiniciar/encender el sensor con una pulsación o apagarlo manteniendo pulsado 10 segundos. Incluyen también una ranura para memorias Micro SD para poder guardar los

datos además de transmitirlos, y una batería de litio de 450mAh, cuya duración es larga, ya que estos sensores, como todos en general, consumen poca energía como característica principal.

En la placa base se puede encontrar un conector interno que da la posibilidad de añadir diferentes módulos y así obtener más señales que puedan ofrecer información adicional. Estos módulos posibles son todos los que hemos comentado antes: el giróscopo, magnetómetro, ECG (electrocardiograma), EMG (electromiografía), GSR (Respuesta galvánica de la piel), etc.

En la figura 11 mostramos esta unidad básica que contiene un acelerómetro. El conector X que se muestra en la figura, sirve para cargar la batería, actualizar el firmware del sensor, o conectar el sensor al ordenador para leer los datos de la memoria Micro SD.

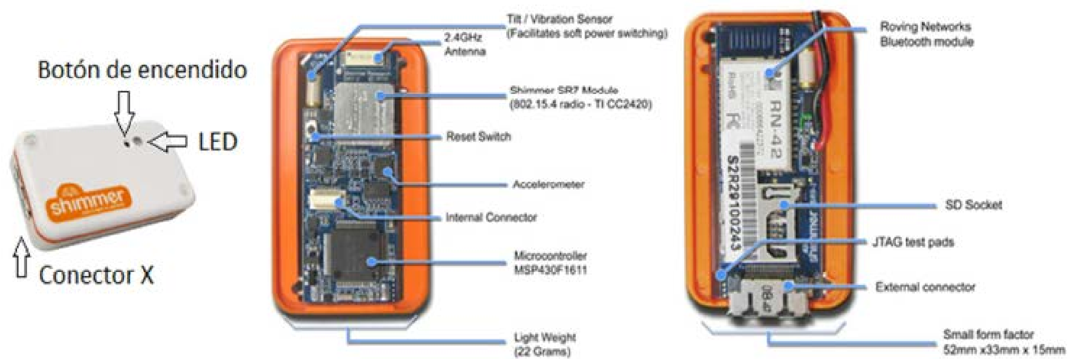


Figura 11. Unidad básica de Shimmer

Shimmer Dock

En nuestro caso tenemos disponible la versión 2 para la realización del proyecto. En el momento de realización del presente proyecto la última versión lanzada al mercado es la 3. –Aunque la guía de usuario que empleamos es para la versión 3 [17], se pueden obviar los cambios puesto que los puntos básicos no cambian–.

Shimmer Dock es un dispositivo al que podemos conectar nuestros módulos Shimmer para cargar la batería o conectarlos al ordenador vía USB

para una actualización de firmware o una posible lectura de los datos guardados en la memoria del sensor. Consta de tres botones, y un led. Según su color nos indica si el sensor está encendido (color verde), si el sensor se está cargando (parpadea con color naranja), y si estamos accediendo a la memoria del sensor (parpadea con color azul), o si se encuentra alguna anomalía (color rojo). Con los botones podemos apagar o encender el sensor.

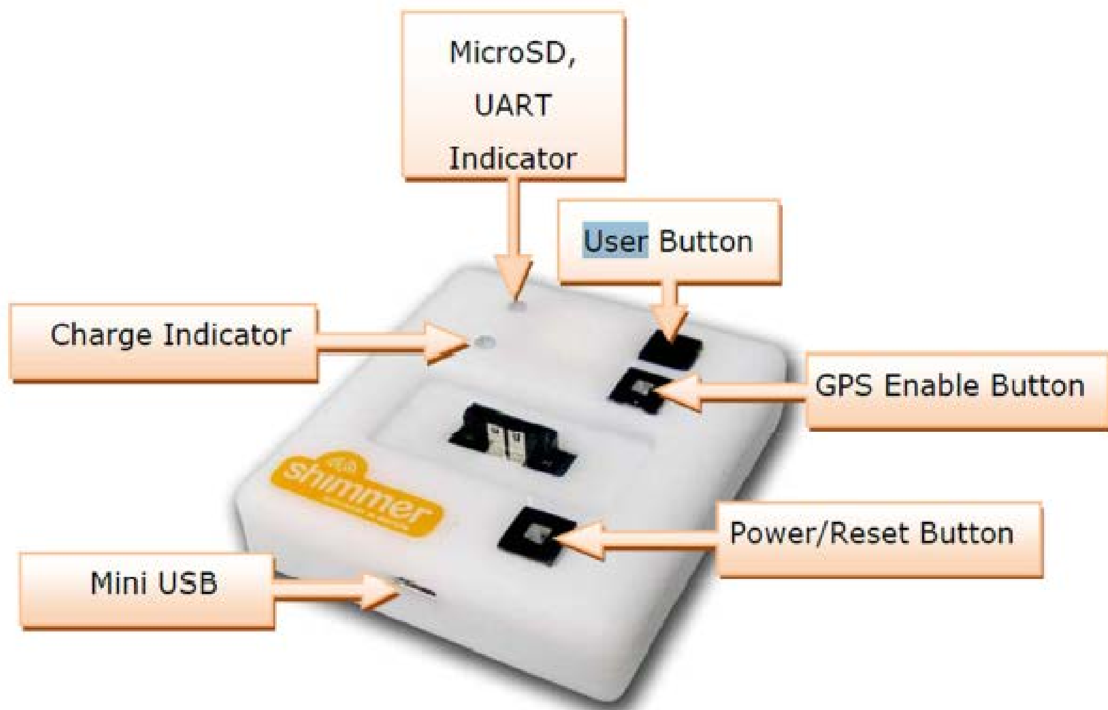


Figura 12. Shimmer Dock v.2

Módulos de expansión

Ya citamos anteriormente los módulos que tenemos disponibles. Estos se pueden añadir a la unidad básica para obtener datos empleando LABVIEW o la aplicación Android (capítulo cinco). Para detallarlos un poco más:

ECG (electrocardiograma)

Es el dispositivo que se emplea para medir y mostrar de forma gráfica la actividad eléctrica del corazón, puesto que para que la contracción cíclica del corazón se realice en forma sincrónica y ordenada, existe un sistema de estimulación y conducción eléctrica compuesto por fibras de músculo cardíaco

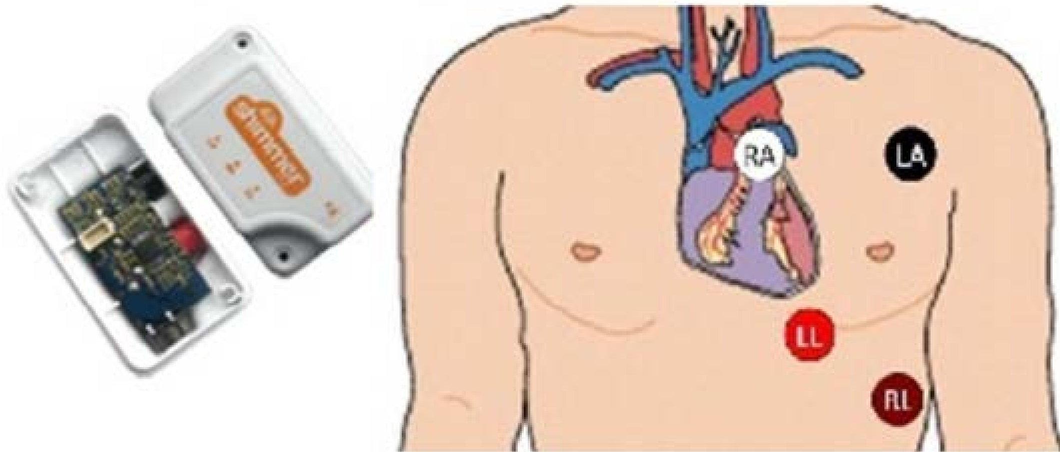


Figura 13. Módulo ECG y colocación de electrodos

especializadas en la generación y transmisión de impulsos eléctricos. Podemos medir y tomar datos por tanto de estos pulsos. Como inconveniente tenemos que es un método ligeramente invasivo, puesto que tenemos que colocar electrodos en determinados lugares del tronco. Estos electrodos se conectarían a nuestro módulo ECG, el cual nos permite obtener y procesar estos datos.

GSR (respuesta galvánica de la piel)

Los cambios en la resistencia galvánica o eléctrica de la piel dependen de ciertos tipos de glándulas sudoríparas que son abundantes en las manos y los dedos. Este fenómeno se conoce como respuesta galvánica (GSR) o conductancia de la piel (SRC), que consiste en una forma de medir las respuestas del sistema nervioso autónomo, específicamente del sistema nervioso simpático. Así, esta resistencia eléctrica de la piel responde en gran medida a pensamientos y a determinantes emocionales y motivacionales. Como reacción a, por ejemplo, estados de estrés, se producen por tanto

cambios en la conductividad eléctrica de la piel.

Figura 14. Módulo GSR y colocación.

EMG (electromiografía)

Este módulo se emplea para registrar y medir la actividad eléctrica producida por los movimientos y contracciones musculares. Puede evaluar la conducción nerviosa, la respuesta del músculo en un tejido lesionado, el nivel de activación, o se puede utilizar para analizar y medir la biomecánica del movimiento. En nuestro caso, medir el nivel de vigilancia del conductor, movimientos musculares bruscos tras un estado de reposo, podrían ser un signo de fatiga.

El sensor EMG es de superficie y, por lo tanto, nos da una representación de la actividad de todo el músculo. Para lograr un correcto funcionamiento, hay que seguir una serie de normas a la hora de colocar los electrodos encima del músculo que nos interesa, que son las siguientes:

- ✓ Los electrodos positivos y negativos deben colocarse en paralelo con las fibras musculares del músculo que se está midiendo.
- ✓ El electrodo de referencia debe ser colocado en un punto eléctricamente neutral del cuerpo, tan lejos como sea razonablemente posible del músculo que se está midiendo. El tobillo o la muñeca suelen ser

recomendables.

- ✓ El electrodo no se debe colocar ni cerca ni sobre el tendón del músculo. Tampoco se debe colocar sobre el punto motor, ni próximo al "borde" del músculo.
- ✓ La distancia máxima entre los electrodos positivo y negativo no debe superar 10cm. Se recomienda la distancia más corta posible.

Figura 15. Módulo EMG y su colocación.

9DOF

El módulo 9DOF de Shimmer combina la función de un giróscopo con la de un magnetómetro para ofrecer una potente solución de detección cinemática. Este módulo combinado con el acelerómetro incorporado en la unidad básica de Shimmer puede proporcionar una orientación estática y dinámica, de medición inercial y convertirse en un sensor complejo de movimientos en 3D.

Este módulo es altamente configurable, y obviamente ofrece la posibilidad de realizar una calibración, cuyo procedimiento veremos en el punto 3.2.2. El sensor magnético reduce el error de deriva en los cálculos cinemáticos permitiendo así capturar excelentes datos para una amplia gama de aplicaciones, incluyendo el seguimiento de objetos, la rehabilitación y el

entrenamiento deportivo.

Figura 16. Módulo 9DOF.

Giróscopo

Es el elemento que utilizaremos principalmente en el proyecto. Como ya hemos dicho, mide la variación de la velocidad angular (grados/sg o radianes) en cualquiera de los tres ejes espaciales.

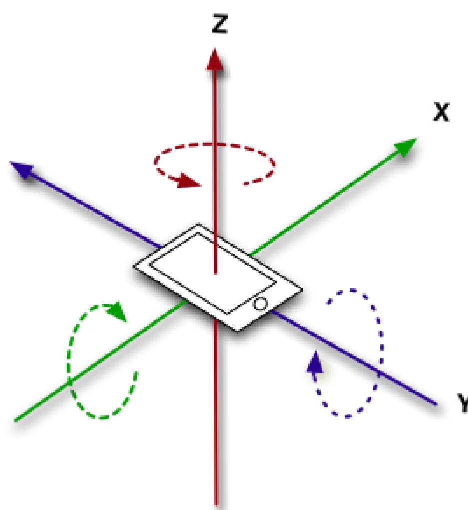


Figura 17. Giróscopo

Para nuestro objetivo, el giróscopo se coloca sobre el volante para medir las

variaciones en el giro de acuerdo a la conducción y extraer unas determinadas estadísticas que veremos en los capítulos próximos, para tratar de determinar el nivel de vigilancia del conductor, por lo que solo nos interesará medir la velocidad angular en uno de los tres ejes.

3.2.2 Software Shimmer

Shimmer 9DOF Calibration

Tanto para el acelerómetro de los sensores Shimmer como para nuestro giróscopo, realizar un proceso de calibración puede mejorar en gran medida la precisión de los datos extraídos del sensor. Para ello, Shimmer nos ofrece esta aplicación de manera gratuita. The Shimmer 9DOF Calibration es una aplicación desarrollada sobre LABVIEW que nos permite calcular los parámetros de compensación para el acelerómetro, las sensibilidades y alineaciones de los ejes, y parámetros de calibración para el giróscopo y magnetómetro. [18].

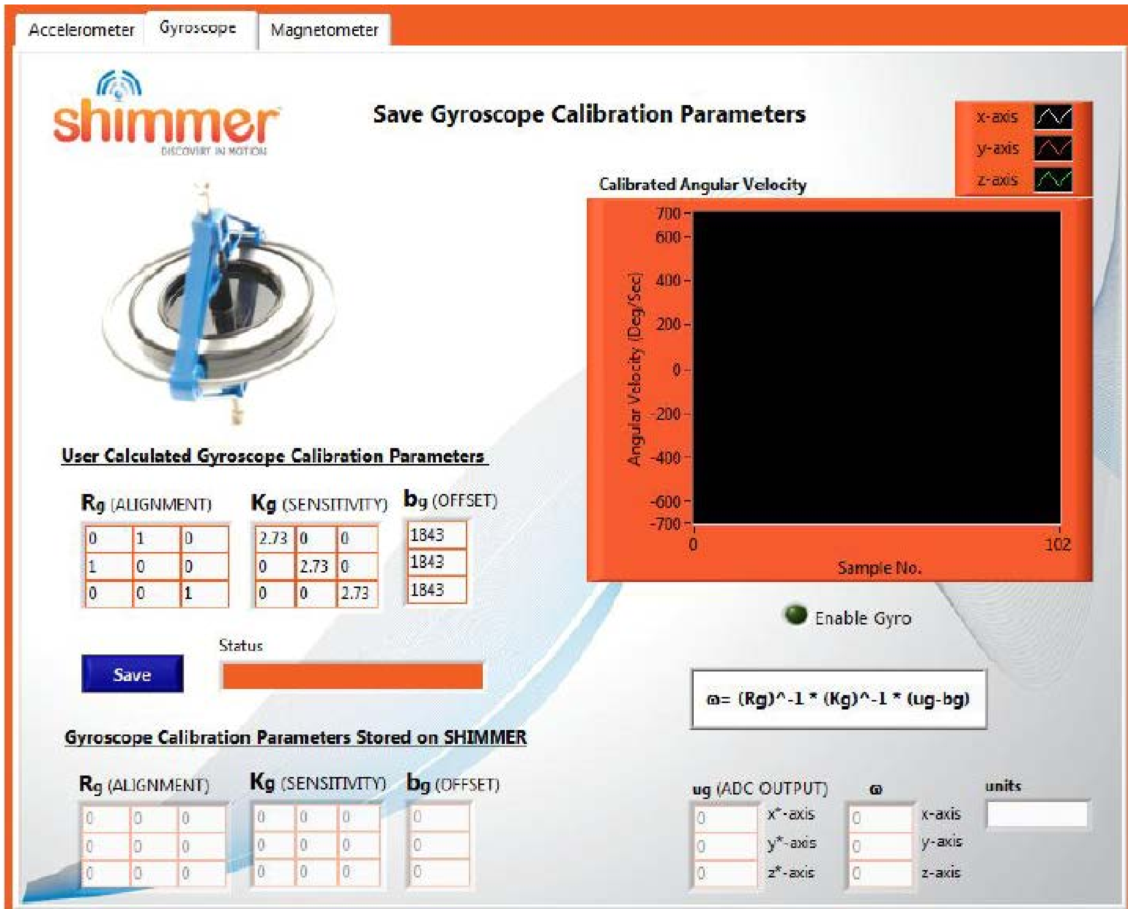


Figura 18. Interfaz de Shimmer 9DOF Application

En la figura 18 mostramos la interfaz de usuario de la aplicación para la calibración de nuestro giróscopo. De acuerdo al manual de usuario de la aplicación, podemos definir la velocidad angular (ω) determinada por el giróscopo según la siguiente fórmula:

$$y_g = k_g \omega + b_g \qquad \text{[ec.1]}$$

Dónde: ω es la velocidad angular obtenida, k_g el factor de escalado, b_g es el factor que modela el efecto de la velocidad lineal en la tasa de giro y es despreciable puesto que los giróscopos modernos se diseñan para minimizar este efecto.

Por tanto, esta aplicación nos proporciona este factor k_g que se aplica a nuestros datos. Como vemos en la figura 18, k_g es una matriz tridimensional, para aplicarse a los tres ejes espaciales en los que podemos determinar la velocidad angular:

$$K_g = \begin{bmatrix} k_{g,x} & 0 & 0 \\ 0 & k_{g,y} & 0 \\ 0 & 0 & k_{g,z} \end{bmatrix}$$ [ec.2]

Proceso de emparejamiento

Como parte del software de los sensores, podemos explicar el proceso de emparejamiento necesario para conectar vía bluetooth nuestro sensor Shimmer con nuestro PC. El proceso se detalla en el Anexo I.

4. TECNOLOGÍAS QUE EMPLEAREMOS

4.1 Introducción

Como ya hemos dicho, para comunicarnos con los sensores, y extraer y procesar datos de estos y del giróscopo en particular, vamos a hacer uso de aplicaciones desarrolladas para diferentes tecnologías, que veremos en este apartado. Hay variedad en cuanto a tecnologías disponibles para las implementaciones que vamos a realizar, pero su elección en nuestro proyecto va a estar determinada por las aplicaciones base de las que partimos.

Para sincronizar con los sensores, extraer datos, almacenarlos y presentarlos mediante interfaz gráfica al usuario partimos de las aplicaciones desarrolladas por Khadmaoui, Amine [2][3]. Vamos por tanto a hacer uso de un entorno de programación gráfica, como es LABVIEW y de una aplicación desarrollada para la plataforma Android.

Posteriormente, tenemos que establecer la comunicación de esta aplicación Android con el simulador de conducción, que está desarrollado con el motor gráfico Unity. Vamos a detallar cada una de estas tecnologías.

4.2 LABVIEW y la programación gráfica

4.2.1 Introducción

LABVIEW es un programa creado por *National Instruments* (1976), es un entorno de desarrollo diseñado específicamente para ingeniería y ciencia, con una sintaxis de programación gráfica.

Es un software altamente relacionado con lo que denominamos instrumentación virtual basada en el concepto de utilizar medios electrónicos como instrumentos de medición de señales, tales como temperatura, presión, caudal, etc.

Como ya hemos dicho, LABVIEW utiliza lenguaje de programación gráfico o G-code. Este es algo distinto de los que solemos usar habitualmente basados en líneas de código. Estos últimos, como C++ o Java, siguen un modelo de flujo de control en el que el orden secuencial de los elementos del programa determina el orden de ejecución final de la aplicación. Mientras, la programación gráfica consiste en bloques que reciben entradas y producen unas salidas, y se pueden conectar entre ellos; un modelo, por tanto, de flujo de datos en el que un nodo de nuestro diagrama de bloques se ejecuta cuando recibe todas las entradas requeridas. Cuando el nodo se ejecuta, produce datos de salida y pasa los datos al siguiente nodo en la trayectoria del flujo de datos. El movimiento de datos a través de los nodos determina el orden de ejecución de la aplicación final y las funciones generales en el diagrama de bloques. En nuestro proyecto veremos más adelante en los capítulos correspondientes estas diferencias entre estilos de programación, al extraer los mismos datos del giróscopo tanto en LABVIEW (programación gráfica) como en Android (programación secuencial basada en código).

LABVIEW, por tanto, presenta ciertas ventajas, puesto que permite una fácil integración con tarjetas de medición, adquisición y procesamiento de datos (incluyendo adquisición de imágenes) tanto del propio fabricante como de otros

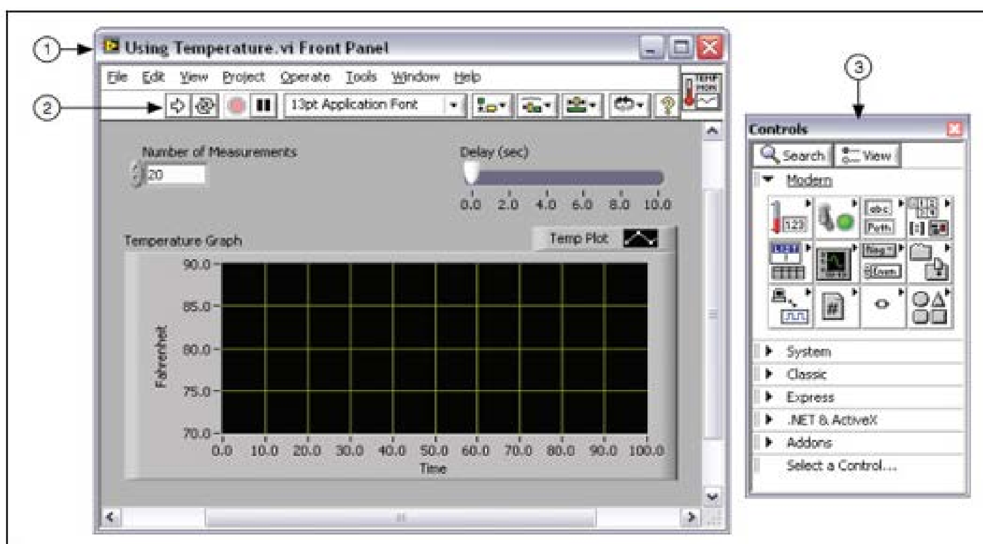
fabricantes. Posee además numerosos bloques con subrutinas muy variadas y facilidades para la representación gráfica de datos, que en otros lenguajes “normales” de programación son más laboriosos de programar.

4.2.2 Conceptos básicos de LABVIEW

Antes de detallar la parte de nuestro proyecto desarrollada con LABVIEW, vamos a introducir los conceptos básicos del programa y su estilo de programación gráfica. Los programas de LABVIEW son llamados instrumentos virtuales o VIs ya que en su apariencia generalmente imitan a los instrumentos físicos, como osciloscopios o multímetros (referencia a la instrumentación virtual de la que hemos hablado). Cuando se crea un nuevo VI, disponemos de dos ventanas: la ventana del panel frontal y el diagrama de bloques. [11].

4.2.2.1 <u>Panel frontal</u>

Es la interfaz visual para el usuario del VI correspondiente, representa de manera visual las salidas y resultados del programa mediante valores, gráficas, indicadores, etc. con interfaces similares a las que proporcionaría un instrumento electrónico real.



(1) Ventana de Panel Frontal | (2) Barra de Herramientas | (3) Paleta de Controles

Figura 19. Panel frontal

4.2.2.2 Paleta de controles

La paleta de Controles contiene los controles e indicadores que podemos utilizar para crear la interfaz de usuario del panel frontal. Está dividida en categorías y contiene una gran variedad de elementos que podemos usar: controles e indicadores numéricos, gráficos, osciloscopios, *arrays*, etc.

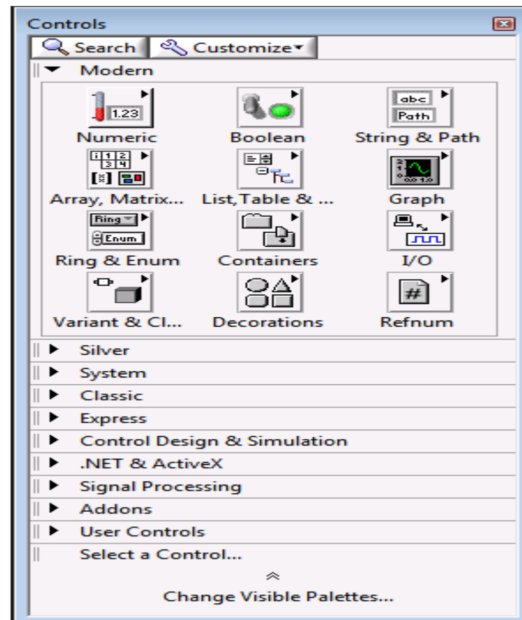
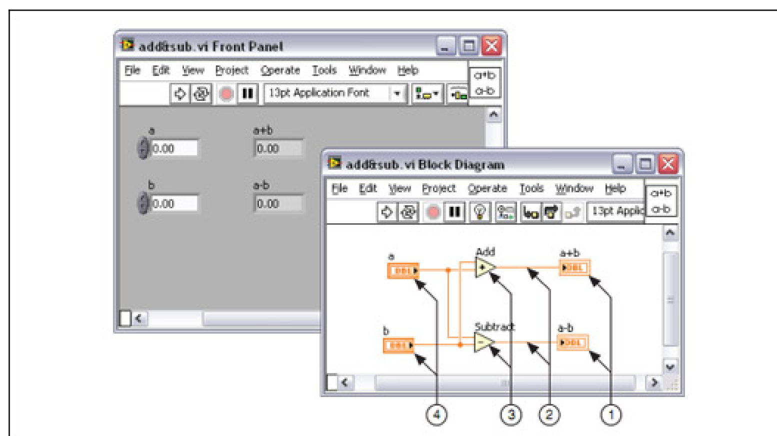


Figura 20. Paleta de controles

4.2.2.3 Diagrama de bloques

Es la otra ventana que conforma nuestro VI, en esta ventana es donde desarrollamos e implementamos nuestro programa mediante los bloques necesarios, uniendo estos para implementar el procesamiento de las entradas que queramos.



(1) Terminales de Indicador | (2) Cables | (3) Nodos | (4) Terminales de Control

Figura 21. Diagrama de bloques y panel frontal correspondiente.

Como vemos en la figura 21, en el diagrama de bloques podemos tener distintos elementos:

- Terminales, que representan los elementos del panel frontal.
- Cables, para unir los terminales y controlar el flujo de datos.
- Funciones y SubVIs, para realizar tareas específicas.
- Estructuras, son los bucles de la programación convencional.

4.2.2.4 Paleta de funciones

Es la paleta asociada al diagrama de bloques, contiene los VIs, funciones y constantes que podemos arrastrar como bloques al diagrama de bloques para crear nuestro programa.

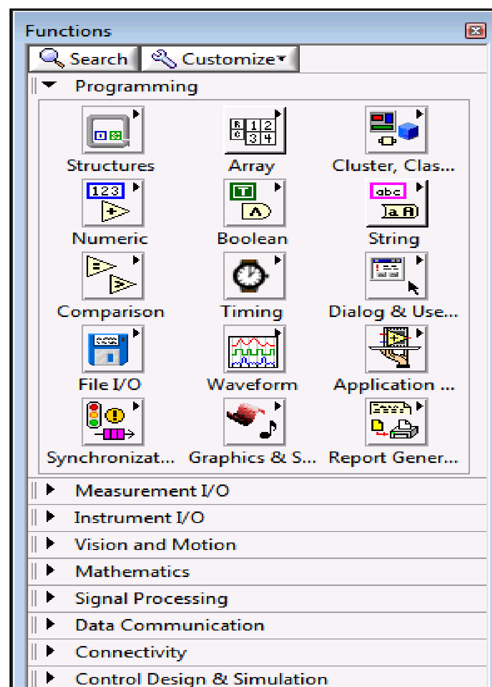


Figura 22. Paleta de funciones

4.2.3 Software LABVIEW necesario

Para desarrollar nuestro proyecto, partimos del programa base creado por Khadmaoui, Amine (2012) [2], por tanto, necesitamos instalar como añadido a LABVIEW, la misma familia de productos que en el proyecto citado:

4.2.3.1 LABVIEW Advanced Signal Processing Toolkit

Es un paquete de herramientas de software con utilidades para el análisis de señales en los dominios de tiempo y frecuencia. (Nosotros hemos usado, tanto para LABVIEW como para estas extensiones, la versión de 2010).

4.2.3.2 LABVIEW Digital Filter Design Toolkit

LabVIEW Digital Filter Design Toolkit extiende LABVIEW con funciones y herramientas interactivas para diseño, análisis e implementación de filtros digitales. Implementa herramientas de diseño interactivas e integradas, así como una gran variedad de algoritmos, topologías de filtros y herramientas de análisis para filtros digitales de punto fijo y flotante. Este módulo y el anterior se emplean para, por ejemplo, procesar la señal del electrocardiograma de Shimmer.

4.2.3.3 NI VISA

NI VISA (Virtual Instrument Software Architecture) es un estándar para configurar, programar y solucionar problemas de instrumentación que comprenden las interfaces GPIB, VXI, PXI, Serial, Ethernet y USB. VISA provee el interfaz de programación entre el hardware y ambientes de desarrollo como LABVIEW. Las interfaces estándar en la industria aseguran que el sistema de desarrollo permite seguir funcionando a lo largo del tiempo. Con estas tecnologías, a parte del bus de comunicación, se ofrece la posibilidad de programar una API estándar independientemente del bus de comunicación usado, da igual que sea GPIB, serie, Ethernet o USB. Mientras no se cambie la interfaz de programación del instrumento, el cambio entre un bus y otro no

requiere modificar el código en gran medida, lo que ayuda a garantizar la larga duración del sistema. Esta extensión nos será útil para comunicarnos con nuestros sensores, con sus puertos COM y extraer los datos necesarios.

4.2.3.4 Librería de Shimmer para LABVIEW

"The Shimmer LABVIEW Instrument Driver Library" es una librería de LABVIEW ofrecida de manera gratuita por Shimmer, para proporcionar a los usuarios una API de bloques ya programados para LABVIEW, que nos darán soporte para nuestro programa con sensores Shimmer 2 y 2R. [19]. (Como ya hemos dicho, usamos los sensores v2 para el desarrollo de nuestro proyecto).

Esta librería incluye un conjunto de VIs completos, para realizar operaciones de bajo nivel de comunicación con los sensores, siendo algo similar a los *drivers* o controladores que usa un PC para comunicarse con sus componentes. Además, incluye una serie de ejemplos de operacionescompletas de trabajo sobre nuestros sensores Shimmer.

La propia librería también incluye un esquema con todos los VIs (VI Tree) que tenemos disponibles para utilizar., A éste se accede haciendo clicc dentro de nuestra paleta de funciones (4.2.4), en Instrument I/O->Shimmer Drives, como podemos ver en las figuras 23 y 24:

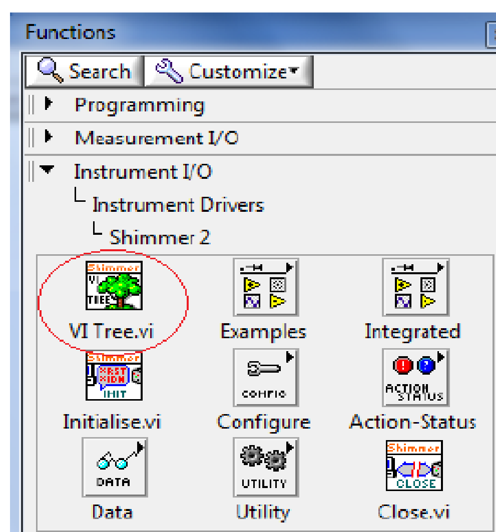


Figura 23. Acceso a esquema.

Initialise Configuration Action/Status Data Utility Close

Integrated VIs

Example VIs

Example Sub VIs

Figura 24. Esquema librería Shimmer de LABVIEW

Mencionar también, que para poder instalar esta librería Shimmer, necesitamos cumplir una serie de requisitos:

- ✓ Tener instalada la versión 9.0 de LABVIEW o posterior (En nuestro caso usamos la 10).
- ✓ La versión de la librería debe coincidir con la versión de LABVIEW. Actualmente Shimmer solo da soporte a las versiones 9.0 y 10.0. –Por ello, empleamos la versión 10 de LABVIEW para nuestro proyecto–.
- ✓ Tener instalado previamente el estándar NI VISA (4.3.3) para poder establecer una comunicación correcta con los puertos COM.

- ✓ Los sensores deben estar programados con la aplicación BoilerPlate v0.1 firmware. Los archivos están disponibles en la página oficial del fabricante.
- ✓ Realizar un previo emparejamiento de los sensores con el ordenador, y anotar los puertos COM correctos. (Ver en Anexo I).

Con todo esto instalado, ya podríamos empezar a desarrollar nuestra aplicación LABVIEW para obtener y mostrar datos del giróscopo Shimmer.

4.3 Android y sistemas operativos móviles

4.3.1 Introducción y comparativa con otros sistemas móviles

Android es en el momento actual el sistema operativo móvil más usado en *smartphones* de todo el mundo, venciendo a su competencia más directa: iOS, el sistema operativo de Apple, y Windows Phone, la apuesta de Microsoft. Podemos ver una ilustración de estos datos en la figura 25 [12].

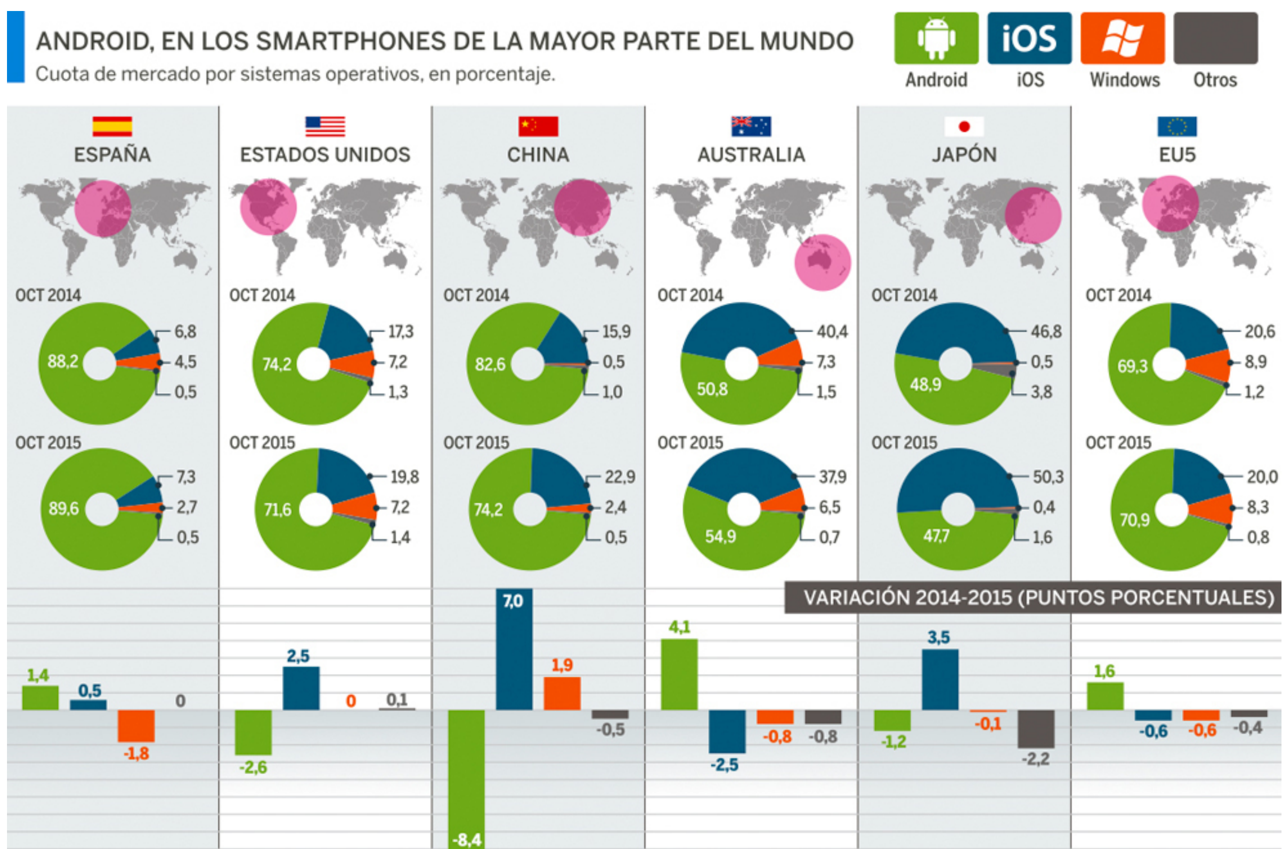


Figura 25. Uso de sistemas operativos móviles según países

Según este artículo [12], el sistema operativo de Google aumentó su cuota de mercado sobre nuevos *smartphones* de un 69,3% a un 70,9% en el conjunto de los 5 principales mercados europeos (Alemania, Francia, Reino Unido, Italia y España), según datos de ventas de teléfonos inteligentes de Kantar Worldpanel de finales de 2015

Y en cuanto a España, Android alcanzó el 89,6% de las ventas en este último tercio de 2015, lo que supuso un crecimiento de 1,6 puntos respecto al mismo período del año anterior. Por otra parte, en cuanto a Apple, iOS ha crecido desde el 6,8% del año anterior (2014) hasta el 7,3%, mientras Windows continuó con una tendencia a la baja perdiendo 1,8 puntos y hasta un 2,7% de cuota de mercado en ese período.

Android es, por tanto, el sistema operativo presente en la mayoría de los teléfonos. Este sistema, basado en el kernel de Linux y diseñado, fundamentalmente, para dispositivos móviles con pantallas táctiles como pueden ser *smartphones*, *tablets* o *notebooks*, fue desarrollado por la compañía Android Inc., que posteriormente fue comprada por Google en 2005. Android fue presentado en noviembre de 2007 junto a la fundación del Open Hanset Alliance (un conjunto de compañías de hardware, software y telecomunicaciones dedicadas al desarrollo de estándares abiertos para dispositivos móviles). El primer dispositivo móvil con Android (HTC Dream) se vendió en octubre de 2008.

La arquitectura de Android se divide en varias partes o componentes principales mediante los que desarrollar todas las funciones de sistema operativo:

- Núcleo Linux: Android utiliza el núcleo de Linux 2.6 como una capa de abstracción para el hardware disponible en los dispositivos móviles. Esta capa contiene los drivers necesarios para que cualquier componente hardware pueda ser utilizado mediante las llamadas correspondientes. Android se apoya en Linux para la mayoría de servicios de bajo nivel, como la seguridad, gestión de memoria,

gestión de procesos y el modelo de controladores utilizados.

- Tiempo de ejecución - Runtime de Android: Este sistema proporciona la mayor parte de las funciones disponibles en Java. Lo constituyen las *Core Libraries*, que son librerías con multitud de clases Java y la máquina virtual Dalvik, en la que cada aplicación ejecuta su propio proceso.
- Librerías: Estas han sido escritas utilizando C/C++ y proporcionan a Android la mayor parte de sus capacidades más características. Junto al núcleo basado en Linux, estas librerías constituyen el corazón de Android. Algunas de ellas son System C library, bibliotecas de gráficos, 3D, SQLite, entre otras.
- Marco de trabajo – Framework de aplicaciones: Representa fundamentalmente el conjunto de herramientas de desarrollo de cualquier aplicación. Toda aplicación que se desarrolle para Android, ya sean las propias del dispositivo, las desarrolladas por Google o terceras compañías, o incluso las que el propio usuario cree, utilizan el mismo conjunto de API y el mismo "framework", representado por este nivel. Entre las API más importantes ubicadas aquí, se pueden encontrar: *Activity Manager*, conjunto de API que gestiona el ciclo de vida de las aplicaciones en Android, entre otras.
- Aplicaciones: Este nivel contiene, tanto las incluidas por defecto de Android como aquellas que el usuario vaya añadiendo posteriormente, ya sean de terceras empresas o de su propio desarrollo. Todas estas aplicaciones utilizan los servicios, las API y librerías de los niveles anteriores. Las aplicaciones en Android, como lo que vamos a desarrollar en el presente proyecto, se programan empleando Java.

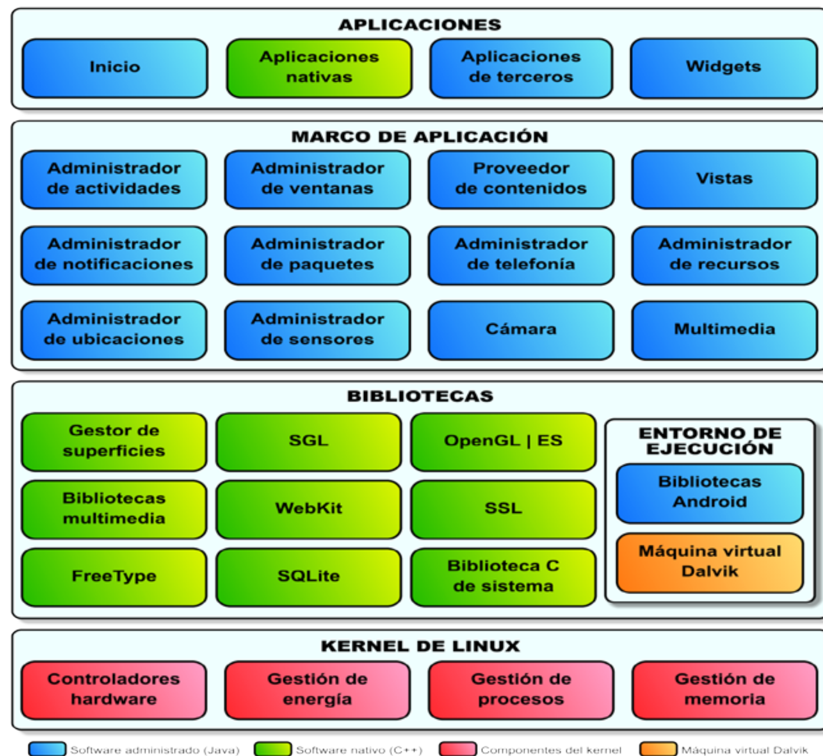


Figura 26. Arquitectura de Android

4.3.2 Herramientas de desarrollo para Android

Para desarrollar aplicaciones para Android, primeramente, además de tener instalado un kit de desarrollo *JAVA SE* (ya hemos dicho que JAVA es el lenguaje de programación de aplicaciones para Android), es actualmente recomendable instalar como entorno de desarrollo *Android Studio*. Es un entorno de desarrollo ofrecido por Google de manera gratuita, que proporciona todas las herramientas necesarias para la creación de aplicaciones en todos los tipos de dispositivos Android: con editor de código de primer nivel, depurador, herramientas de rendimiento, un sistema de compilación flexible y un sistema instantáneo de compilación e implementación.

Figura 27. Android Studio

Pero nosotros, a partir de un proyecto base de Amine Khadmaoui (2014) [3], emplearemos el *IDE (Integrated Development Environment, Entorno Integral de Desarrollo) Eclipse* por simplicidad a la hora de importar todos los archivos del proyecto.

Para ello, primero tenemos que instalar el conjunto de herramientas de desarrollo (*SDK*) que nos proporciona Google de manera gratuita. Este SDK de Android incluye un conjunto de herramientas, la API, para todas las versiones de Android disponibles en el mercado. En el momento de desarrollo del presente proyecto, instalamos y hacemos compatible la aplicación hasta la versión 6.0 (Marshmallow), la última versión disponible de Android.

Por último, para poder empezar a desarrollar aplicaciones Android con Eclipse, tenemos que instalarle el complemento ADT (Android Development Tools Plugin), aunque tenemos disponibles para descarga gratuita versiones de Eclipse que ya incluyen este complemento instalado. Nosotros para la realización del proyecto hemos usado la versión 4.5.2 (Mars.2).

Figura 28. IDE Eclipse Mars.2

Con las herramientas de desarrollo ya instaladas, simplemente resta importar en Eclipse el proyecto Android para poder trabajar con la aplicación.

–Una guía de los pasos a seguir para la importación se adjunta en el Anexo II–

4.4 Unity y motores gráficos de videojuegos

4.4.1 Introducción y otros motores gráficos

Unity es un motor de videojuego multiplataforma creado por Unity Technologies. Un motor de videojuego (*Game Engine*) es un término que hace referencia a una serie de rutinas de programación que permiten el diseño, la creación y la representación de un videojuego. Es decir, un motor proporciona una serie de funcionalidades básicas que permiten al programador trabajar sin preocuparse de la implementación de ciertas cuestiones comunes al mundo de los videojuegos, proporcionando así por ejemplo un motor de *renderizado* para los gráficos 2D y 3D, motor físico o detector de colisiones, sonidos, scripting, animación, inteligencia artificial, redes, *streaming*, administración de memoria o un escenario gráfico.

Hay una gran variedad de motores disponibles, como por ejemplo, *OGRE 3D* que es un motor gráfico gratuito "*open-source*". Además, compañías del sector del videojuego desarrollan sus propios motores en muchas ocasiones; Epic, Valve o Crytek han lanzado al público sus motores o SDK para que los usuarios interesados en el desarrollo de videojuegos puedan descubrir cómo se elaboran.

Nosotros en nuestro proyecto emplearemos Unity, puesto que es un motor gráfico gratuito, flexible y con una gran gama de recursos disponibles. Además de haber sido ya elegido para todos los escenarios de conducción desarrollados, Unity permite hacer buenos desarrollos con poco o ningún coste

y, además, al ser multiplataforma, cualquier proyecto se puede exportar a PC o sistemas operativos móviles que hemos visto, como Android, iOS o Windows Phone sin problemas.

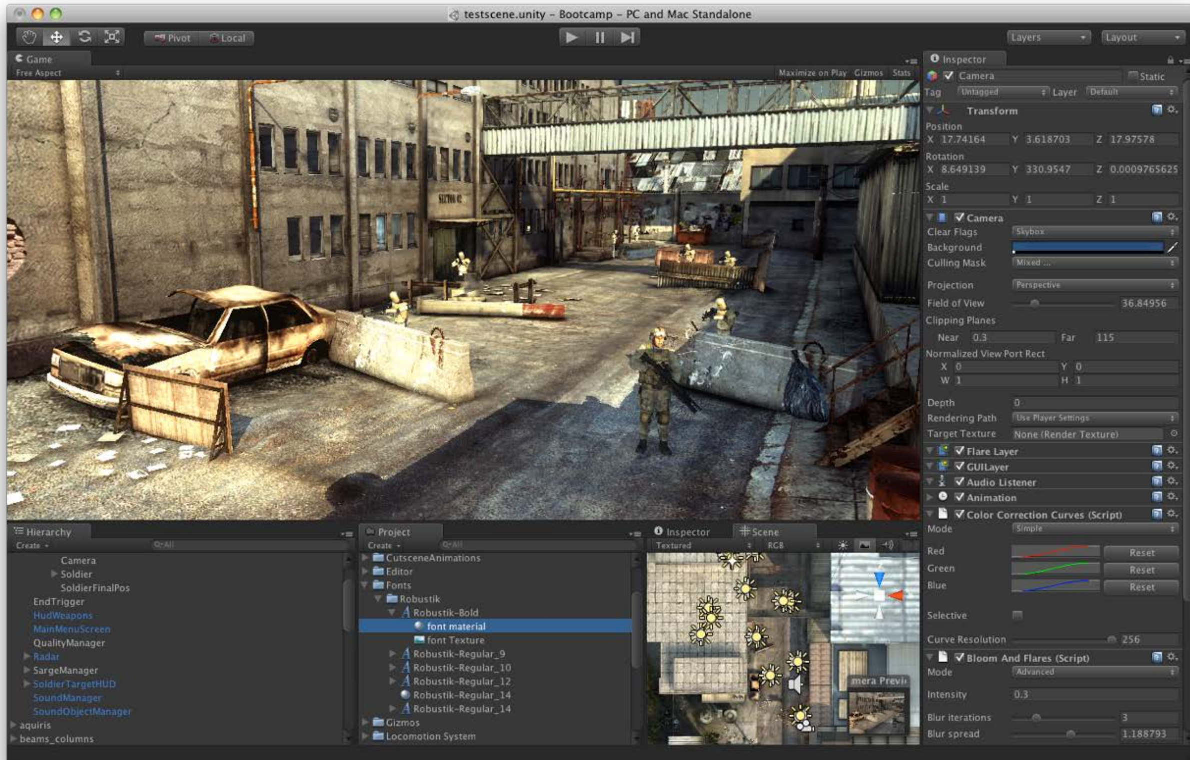


Figura 29. IDE de Unity

Para nuestro proyecto empleamos la versión 5.4.2f2 de Unity. A diferencia de Android, donde empleamos JAVA, el lenguaje de programación que necesitamos emplear para el desarrollo de un proyecto en Unity será C#.

Vistas ya todas las principales tecnologías que vamos a emplear para nuestro proyecto, empezamos a continuación a detallar todo lo que se ha desarrollado.

5. LABVIEW

5.1 La aplicación base

Como ya hemos dicho, partimos de la aplicación desarrollada por Khadmaoui, Amine (2012) [2], y extenderemos sus funciones para los datos extraídos del giróscopo.

En esta aplicación base, y con el uso de la librería Shimmer explicada anteriormente, teníamos implementados los mecanismos básicos para conectar con los sensores Shimmer, incluido el giróscopo. En la figura 30 mostramos el panel frontal de esta aplicación de la que partimos.

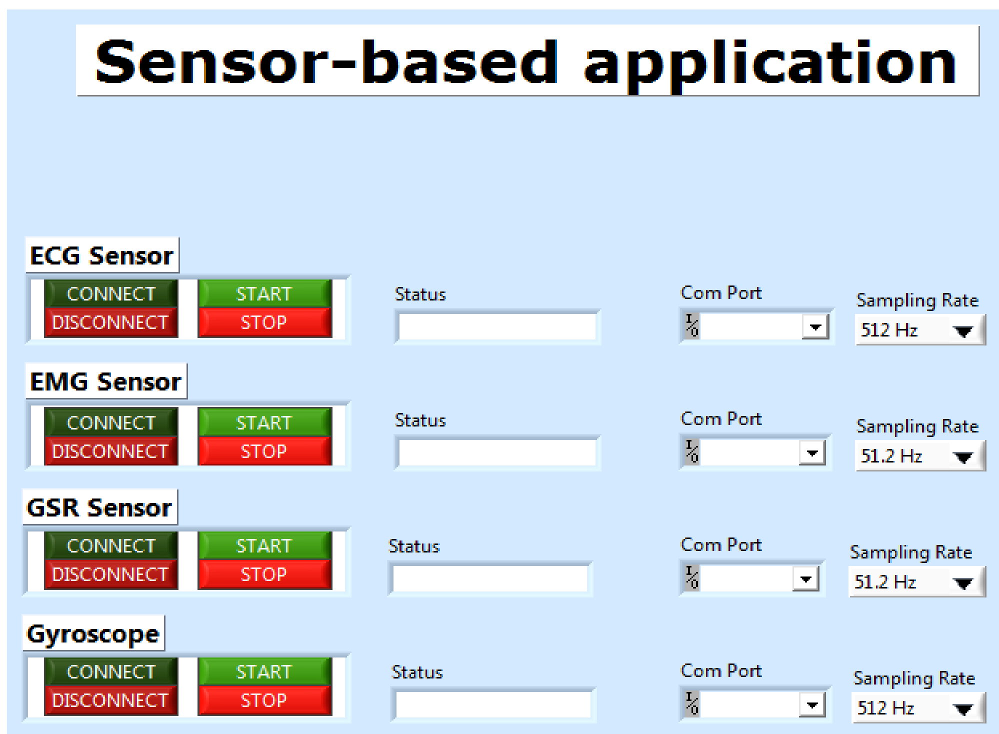


Figura 30. Panel frontal app inicial

Así que para conectar nuestro giróscopo solo tenemos que seleccionar la tasa de muestreo (Fs) [Hz] con la que extraeremos los datos (*Sampling Rate*):

$$Fs=1/T \qquad [ec.3]$$

Dónde: T es el intervalo temporal que transcurre entre cada muestra tomada.

-Cuanto mayor sea la frecuencia de muestreo que usemos, tendremos un archivo de muestras mayor, más significativo, pero las gráficas serán más complicadas de visualizar, debido a que se dibujarán más rápidamente-.

Luego, en el giróscopo, seleccionamos el puerto COM con el que tenemos emparejado nuestro sensor por Bluetooth (Ver anexo I). Y finalmente, pulsamos los botones “CONNECT” y “START” para empezar a adquirir los datos del giróscopo. Este proceso de conexión se corresponde con el diagrama de estados que tienen los sensores:

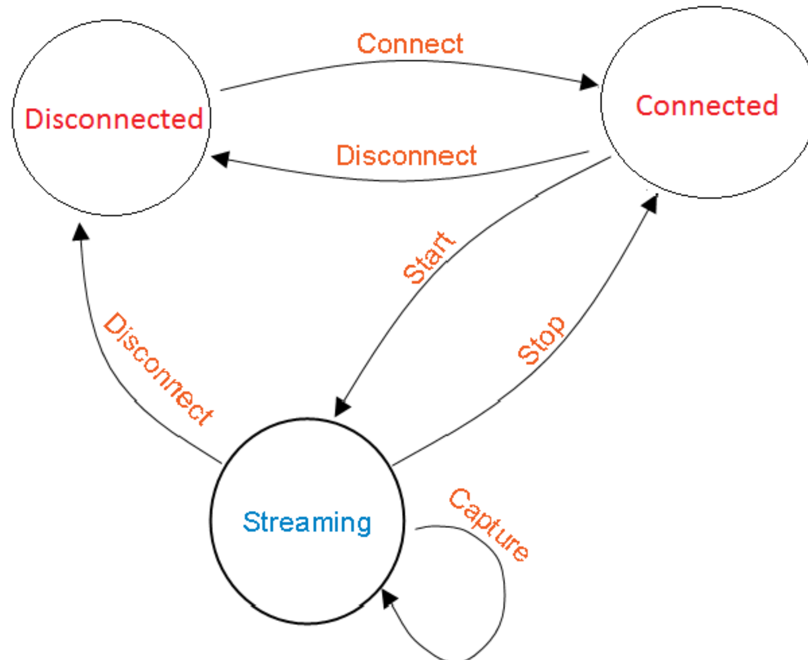


Figura 31. Diagrama estado sensores

Así, nuestro diagrama de estados y procedimiento descrito anteriormente para establecer la conexión se puede resumir de acuerdo a la siguiente tabla:

Estado Actual	Acción	Estado siguiente
Disconnected	Connect	Connected
Connected	Disconnect	Disconnected
Connected	Start	Streaming
Streaming	Stop	Connected
Streaming	Disconnect	Disconnected

Tabla 1. Tabla de estados de sensores

Cabe destacar que, en caso de querer utilizar los cuatro sensores simultáneamente, debemos seguir una serie de pasos en un orden concreto. Los pasos se describen a continuación:

- Primero, debemos conectar los cuatro dispositivos antes de empezar a transmitir.
- Una vez estén todos conectados, podemos proceder a la transmisión del Giróscopo, ya que es el que más tarda en empezar.
- Por último podemos iniciar la transmisión de los siguientes.

Esto es debido a que si iniciamos el proceso de conexión de algún sensor, mientras otro está transmitiendo, se produce un error en los datos transmitidos por este último.

Estas instrucciones se muestran además dentro de la aplicación, en la pestaña instrucciones:

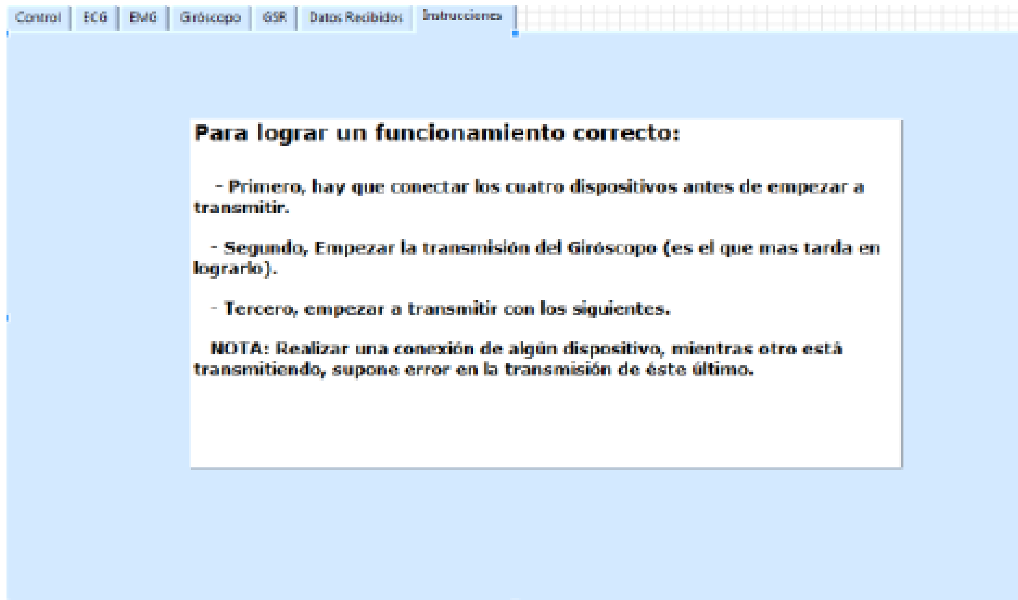


Figura 32. Instrucciones de uso simultáneo

Una vez establecida la conexión, nuestra aplicación base nos mostraba unos datos básicos para el giróscopo; básicamente una gráfica con la señal que íbamos obteniendo:

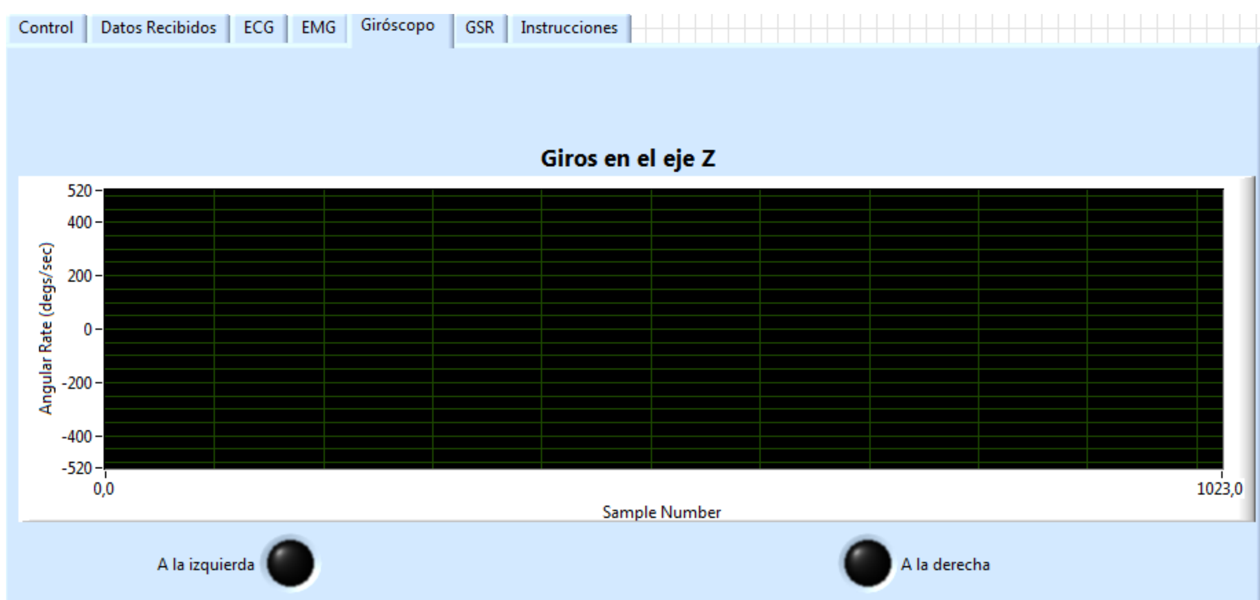


Figura 33. Panel frontal giróscopo inicial

Con esta base, es sobre la que hemos construido nuestra aplicación en el presente proyecto, para obtener unos datos más extendidos del giróscopo.

5.2 Nuestra aplicación LABVIEW

5.2.1 Introducción

Para introducir el desarrollo de nuestra aplicación, primero vamos a ver cómo se extraen los datos del sensor con el apoyo de la librería que proporciona Shimmer.

Para comunicarnos con los sensores, el giróscopo en nuestro caso, usamos el bloque Shimmer 9DOF que nos proporciona Shimmer para LABVIEW: (Con 'crtl+H' podemos ver la ayuda para los bloques)

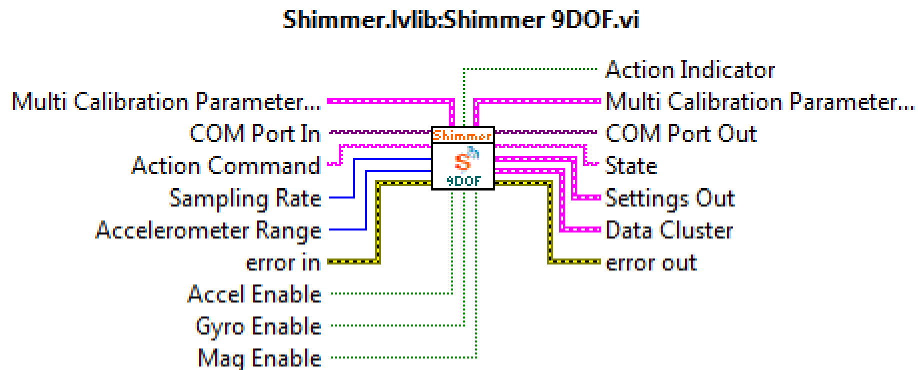


Figura 34. Bloque Shimmer para LABVIEW

De las salidas del bloque, nos interesa especialmente "*Data Cluster*", que es de donde extraeremos los valores de giro del giróscopo. Este *Cluster* es el bloque en el que se encuentran los datos del giróscopo en nuestro caso, que tiene la estructura mostrada en la figura 35.

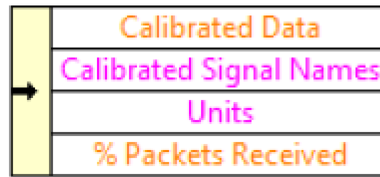


Figura 35. Data Cluster

Este *Cluster* tiene distintos vectores, el campo/vector "Calibrated Data" es en el que tenemos los datos de giro en sí, (el giróscopo Shimmer v2 proporciona una precisión de 520º/sg, lo que es casi una vuelta y media del volante por segundo). En el caso del giróscopo este campo es una matriz `nxm`, donde `n` es el número de muestras que tomemos de datos del giróscopo y `m` es 3, una columna por cada eje espacial del que podemos obtener datos de giro.

Pero, a nosotros solo nos interesa la velocidad angular en el eje espacial que se corresponda con el giro del volante, que es para nuestro sensor el eje Z. Para extraer los datos de este eje usamos el bloque de LABVIEW "Index Array", con el que extraemos la columna tres de la matriz, y pasamos de tener datos en paralelo a tener los datos en serie del eje Z. Mostramos esta parte del esquema de nuestro diagrama de bloques en la figura 36.

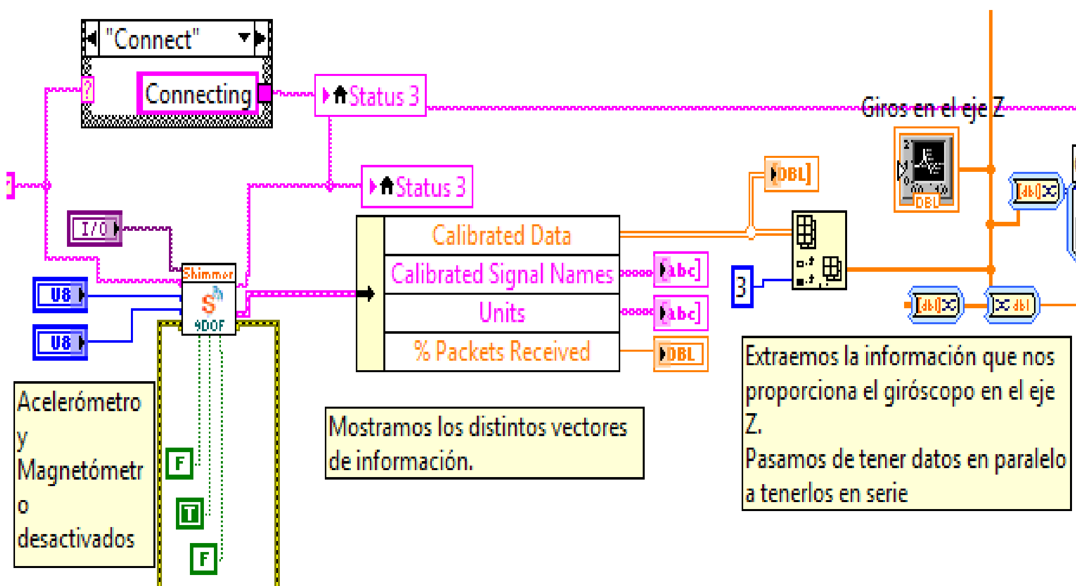


Figura 36. Nuestro diagrama de bloques

Almacenamos además un fichero con los datos extraídos en la ruta

`C\temp` (En esta ruta almacenaremos todos los distintos datos que vamos a procesar).

Por lo tanto, el primer dato que extraemos del giróscopo es, lógicamente, la velocidad angular de giro en el eje. En el fichero "*Gyro.lvm*", guardamos además las velocidades de giro en los tres ejes, junto con los datos del acelerómetro del giróscopo y las correspondientes marcas temporales.

Teniendo esto, podemos proceder a obtener y analizar los demás datos de interés que detallamos a continuación.

5.2.2 Leds de izquierda y derecha

Lo siguiente que mostramos en nuestro panel frontal son leds para indicar si el volante se gira a la izquierda o a la derecha. Para ello usamos valores umbrales (de 20º/sg), para que no se enciendan todo el rato los indicadores por mínimas variaciones y parezca algo aleatorio.

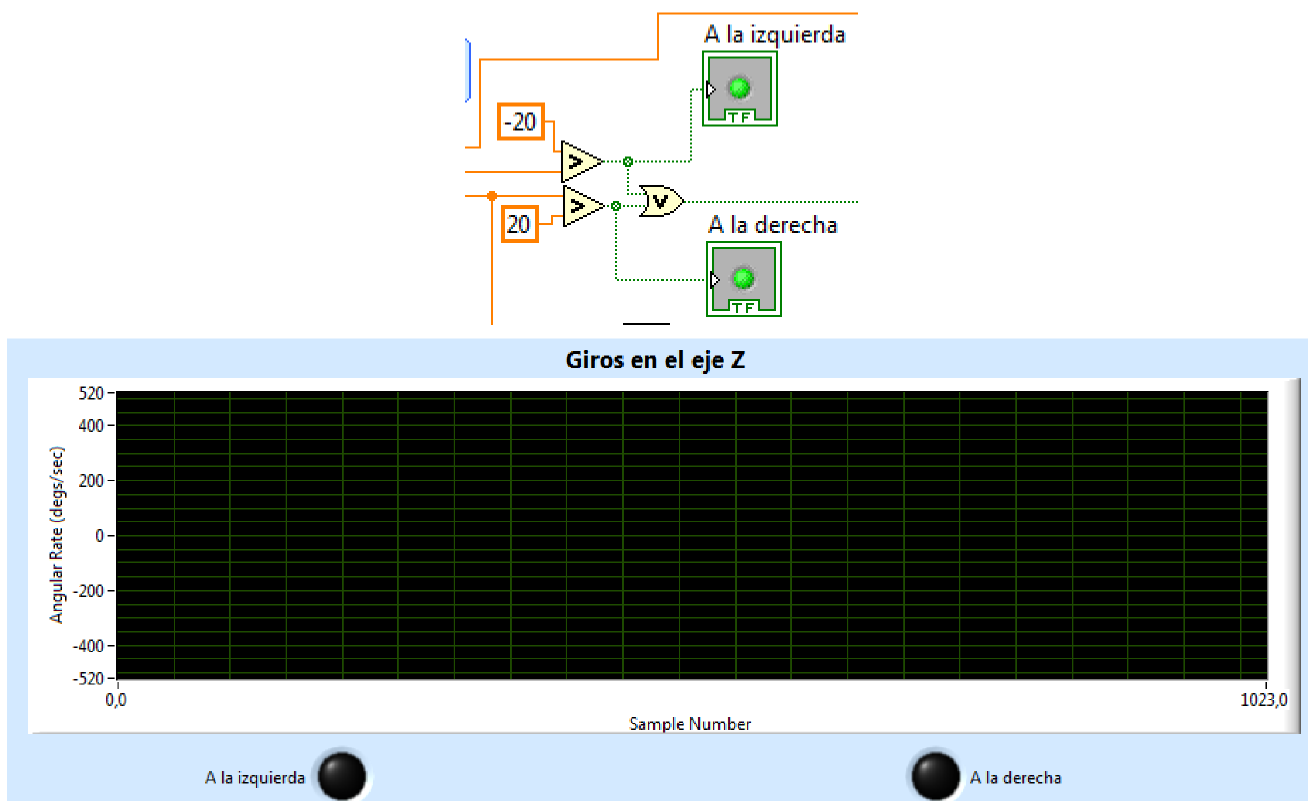


Figura 37. Diagrama de bloques y panel frontal, datos y leds

5.2.3 Posición, número de vueltas, y reposo del volante

El tiempo de reposo del volante es un valor que ya estaba correctamente implementado en la aplicación base, simplemente aumenta de manera progresiva si el valor de giro extraído no supera el umbral de 70º/sg, y se reinicia a '0' en caso contrario.

La posición del volante (en grados) también se mostraba inicialmente en la aplicación base, pero de manera incorrecta. Ahora tenemos una variable que almacena la posición del volante entre cada iteración del bucle general (bucle *while* que coge una muestra en cada iteración).

Estas variables de la programación habitual, como puede ser un "int", en LABVIEW las podemos implementar como un "*Shift Register*", un valor que se guardará entre cada iteración de este bucle. Para ello hacemos clic en el borde de la estructura *while* general mencionada anteriormente con el botón derecho, y seleccionamos "*Add Shift Register*". (Las iniciamos a '0' como vemos en la figura 38).

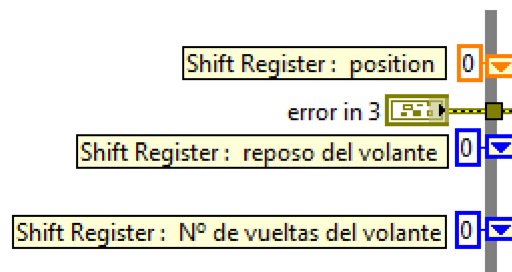


Figura 38. Shift Registers LABVIEW

Por tanto, para calcular la posición del volante, es decir, su variación en grados respecto a la posición inicial al arrancar nuestro programa, lo que hacemos es coger los datos de giro que vamos extrayendo en cada iteración del bucle general, estos datos miden variación angular en [grados/segundo]. Aclarar aquí que:

- La duración temporal de la ejecución de la iteración del bucle general "while" que engloba nuestra aplicación de LABVIEW se fija mediante el bloque "*Wait Until Next ms Multiple*", al que ponemos como entrada 100 milisegundos, la que será la duración de cada iteración.

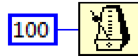


Figura 39. Wait Until Next ms Multplie

- Durante cada una de estas iteraciones, el número de muestras que extraigamos del giróscopo depende de la frecuencia de muestreo fijada (ver ecuación [3]). Lógicamente, a mayor velocidad de muestreo, tendremos más muestras en cada iteración, pero será más difícil seguir visualmente la evolución de las gráficas, como ya dijimos.

Sabiendo esto, lo que hacemos es sumar todas las velocidades (positivas o negativas según la dirección del giro) tomadas en cada iteración, así nos quedamos con el dato en [grados/segundo], luego sabiendo que la duración temporal de la iteración son 100 milisegundos (0.1 sg), multiplicamos por este valor para quedarnos con el dato de desplazamiento en grados (exactamente multiplicamos por 0.117 en vez de por 0.1 segundos por haberse comprobado de manera experimental que el cálculo de la posición es más exacto así).

A este valor que tenemos ya en grados, lo sumamos con el valor de posición que tuviéramos almacenado de la iteración anterior en la variable "Shift Register" correspondiente (ver figura 33). Con esto, ya podemos mostrar la posición en la que se encuentra el volante. Vemos esta parte de nuestro diagrama de bloques en la figura 40.

–Para que el cálculo de la posición sea más preciso, y no sensible a variaciones mínimas que nuestro Shimmer v2 es capaz de detectar, fijamos umbrales de 5 y -5 [º/sg]. Por debajo de estos valores, no se tienen en cuenta

los datos extraídos en la suma para el cálculo de la posición del volante–.

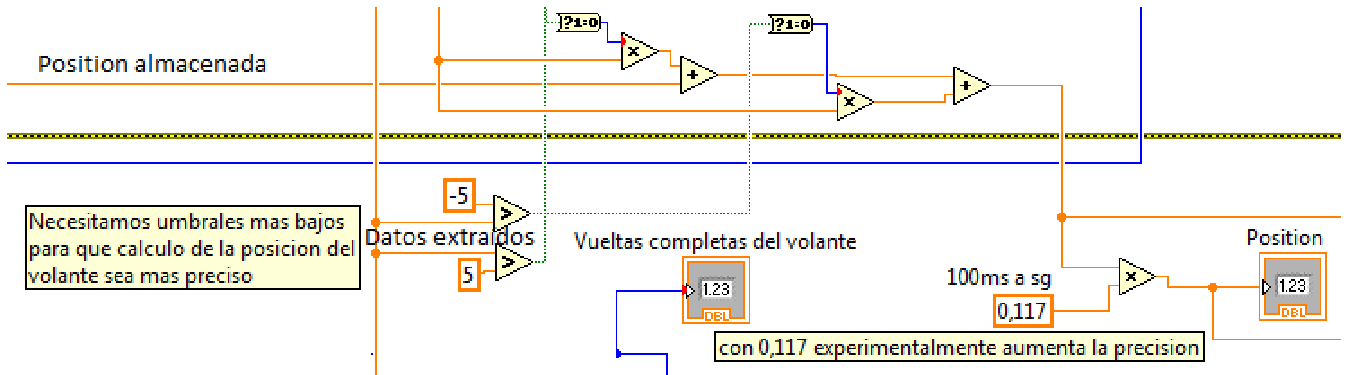


Figura 40. Diagrama de bloques, posición del volante

Teniendo calculada la posición, es fácil obtener el siguiente dato que nos interesa; el número de vueltas completas del volante “*steering wheel reversals*”.

Cuando el cálculo de la posición alcanza valores de 360º o -360º (valor absoluto mayor que 360 en nuestra implementación) aumentamos la variable que almacena el número de vueltas completas. Vemos la parte del esquema del diagrama de bloques correspondiente con esto en la figura 41.

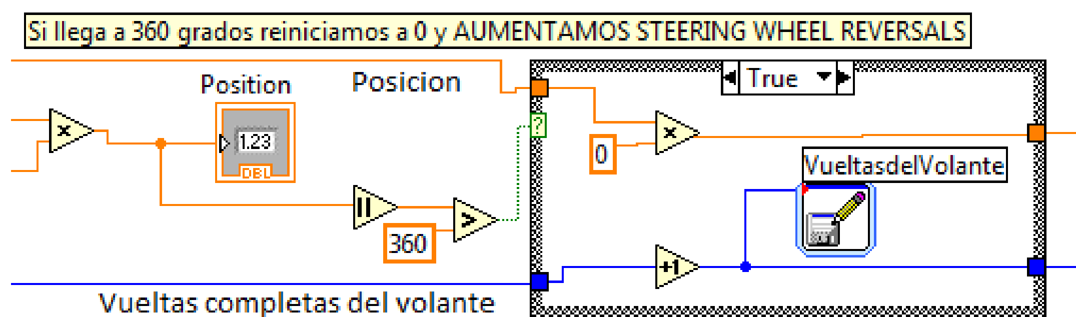


Figura 41. Diagrama de bloques, vueltas completas

Aquí, al igual que para el reposo del volante, hacemos uso de una estructura "Case", que se asemeja al uso de "if" en la programación lineal habitual. Conectamos a la entrada con la interrogación la condición que queremos evaluar (si el valor absoluto de la posición es mayor que 360, en nuestro caso) Por otra parte, tenemos las variables de posición y vueltas completas del volante como entrada, en función de que la condición sea verdadera o falsa. Estas serán modificadas (reiniciada a '0' y aumentada en '1' respectivamente), o saldrán sin ser modificadas. En la figura 41 vemos solo el caso de que la condición sea verdadera y se modifiquen las variables.

Para el valor de las vueltas completas del volante, de acuerdo con el artículo de Xuesong Wang, Chuan Xu (2015) [24], calculamos cada N muestras, la media y desviación típica de los datos obtenidos (explicaremos el procedimiento para calcular estos valores estadísticos en el siguiente punto).

Este artículo nos muestra una tabla, que podemos ver a continuación, con una relación de datos, como las vueltas completas del volante (*Steering Wheel reversals*), y los valores de media y desviación típica correspondientes a estos.

Driving behavior and eye feature metrics.

Metrics	Description of the variables	Mean	S.D.
Driving behavior			
LP_stdev	Standard deviation of lateral position (m)	0.306	0.134
LP_avg	Average of lateral position (m)	0.214	0.269
LD_Area[a]	Sum of lane departure time-space area (m s)	1.627	4.207
LD_TArea[b]	Sum of lane departure time-space area weighted by lane crossing time (m s)	6.129	35.383
LD_Frequency	Lane departure frequency	0.660	1.118
LD_Speed	Lane departure lateral speed (m/s)	0.046	0.099
LD_Tc	Time percentage of lane crossing of the vehicle center	0.002	0.017
LD_Te	Time percentage of lane crossing of the vehicle edge	0.021	0.045
SW_Speed_stdev	Standard deviation of steering angular speed (degree/s)	0.012	0.008
SW_Area_MA[c]	Area surrounded by steering angle and its moving average	0.440	0.257
SWM_Re	Steering wheel reversals	190.485	29.522
SW_Range_1	Percentage of steering speed in 0–2.5 degree/s	0.876	0.085
SW_Range_2	Percentage of steering speed in 2.5–5 degree/s	0.077	0.037
SW_Range_3	Percentage of steering speed in 5–7.5 degree/s	0.024	0.020
SW_Range_4	Percentage of steering speed in 7.5–10 degree/s	0.010	0.012
SW_Range_5	Percentage of steering speed exceeding 10 degree/s	0.013	0.024
Speed	Average speed (km/h)	117.424	6.650
Speed_stdev	Standard deviation of speed (km/h)	2.866	2.860
Speeding_T	Time percentage of speed exceeding the limit speed 120 km/h	0.311	0.372
Eye features			
Blink_Frequency	Average blink frequency per second	0.504	0.318
Blink_duration	Average blink duration (s)	0.402	0.054
PERCLOS	Percentage of eyelid closure	0.132	0.099
Pupil	Average pupil diameter (mm)	3.807	0.894

Tabla 2. Datos del artículo de Xuesong Wang, Chuan Xu (2015)

Aunque este artículo nos ofrece los resultados obtenidos, no detalla el procedimiento ni implementación empleada para obtenerlos.

Nosotros, por tanto, establecemos este valor “N”, número de muestras sobre las que calcularemos todas las medias y desviaciones típicas de nuestro proyecto, como un compromiso en el que:

- Cuanto mayor sea N, tendremos un número de muestras que será más significativo para el cálculo de la media y desviación típica.
- Aunque cuanto mayor sea, más tiempo de ejecución necesitaremos también para que los cálculos se realicen, y además, menos valores de media y desviación podremos calcular en un mismo tiempo de ejecución.

Nosotros originalmente fijamos “N” a ‘20’, un valor que experimentalmente nos ofrece un numero de datos bastante alto en poco tiempo de ejecución, y unos valores similares a los ofrecidos por el artículo (tabla 2).

--Estos datos vistos anteriormente se almacenan en sus ficheros correspondientes; en `C:\temp`, en los ficheros *ReposoVolante, VueltasVolante, MediaVolante y SDVolante*--.

5.2.4 Media y desviación

Por lo tanto, cuando el contador circular de muestras tomadas llega a nuestro valor fijado (‘20’), calcularemos la media y desviación típica de esas veinte muestras, y haremos esto para todos los datos de la aplicación de los que queremos calcular la media (*mean*) y desviación típica o SD (*Standard Deviation*) (ver tabla 2). Es decir, para los datos de giro generales (4.4.2.1), para el número completo de vueltas del volante (4.4.2.3), y para los porcentajes de variación según rangos de velocidad (esto lo veremos a continuación).

Para todos estos tipos de información, usaremos el mismo método para

calcular su media y desviación típica:

La media aritmética es el promedio del conjunto de nuestros 'N=20' valores:

$$(\mu_x) = \frac{1}{N}\sum_{n=1}^{N} x_n \qquad \text{[ec.4]}$$

Donde: N es el número de muestras que tenemos, 20 fijadas en nuestro caso.

X_n son las muestras correspondientes.

Por otra parte, la desviación típica o standard deviation (SD), es una medida del grado de dispersión de los datos con respecto al valor promedio o media calculada:

$$(\sigma_x) = \frac{1}{N}\sum_{n=1}^{N} (x_n - \mu_x)^2 \qquad \text{[ec.5]}$$

Donde: N es el número de muestras que tenemos, 20 fijadas en nuestro caso.

X_n son las muestras correspondientes.

μ_x es la media calculada. (Ver ecuación [4])

Introducidos ambos conceptos, vamos a proceder a explicar su implementación en LABVIEW.

Para calcular ambos valores usamos un VI que nos ofrece la librería

/Mathematics/Probability and Statistics/ de LABVIEW :

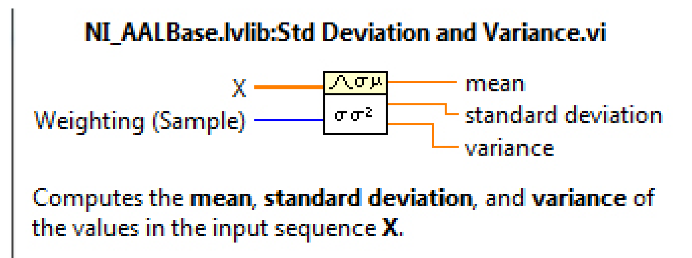


Figura 42. Media y Desviación en LABVIEW

Luego, simplemente tenemos que introducir un *array* de N muestras al bloque, y podemos tener a la salida la media, desviación típica y varianza (esta última no la empleamos). Implementamos el array de entrada de esta manera:

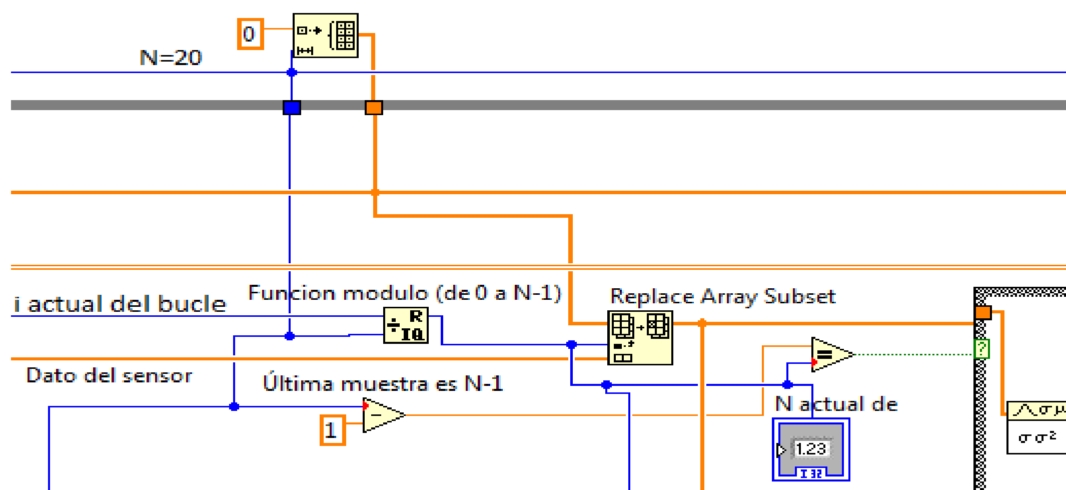


Figura 43. Array de entrada a media y desviación

Vemos en la parte de arriba de la figura que iniciamos un *array* del número de muestras (fuera del bucle while general) requerido, 20 en nuestro caso, este *array* lo llevaremos a todas las partes de nuestro diagrama de bloques en las que implementamos un cálculo de media y desviación. En él, mediante el bloque "*Replace Array Subset*", escribiremos el dato extraído del sensor en la posición correspondiente según la iteración i del bucle general en la que nos

encontremos (de manera circular gracias a la función modulo implementada). Mientras tanto, cuando la posición del *array* inicial en la que escribimos coincida con la N fijada (20), mediante el uso de una estructura *case* que aparece cortada en la figura, introducimos el array con las 20 muestras al bloque, que calculará la media y desviación típica.

Dibujamos también gráficas mediante simples "*waveform graph*", con la evolución de las medias y desviaciones obtenidas:

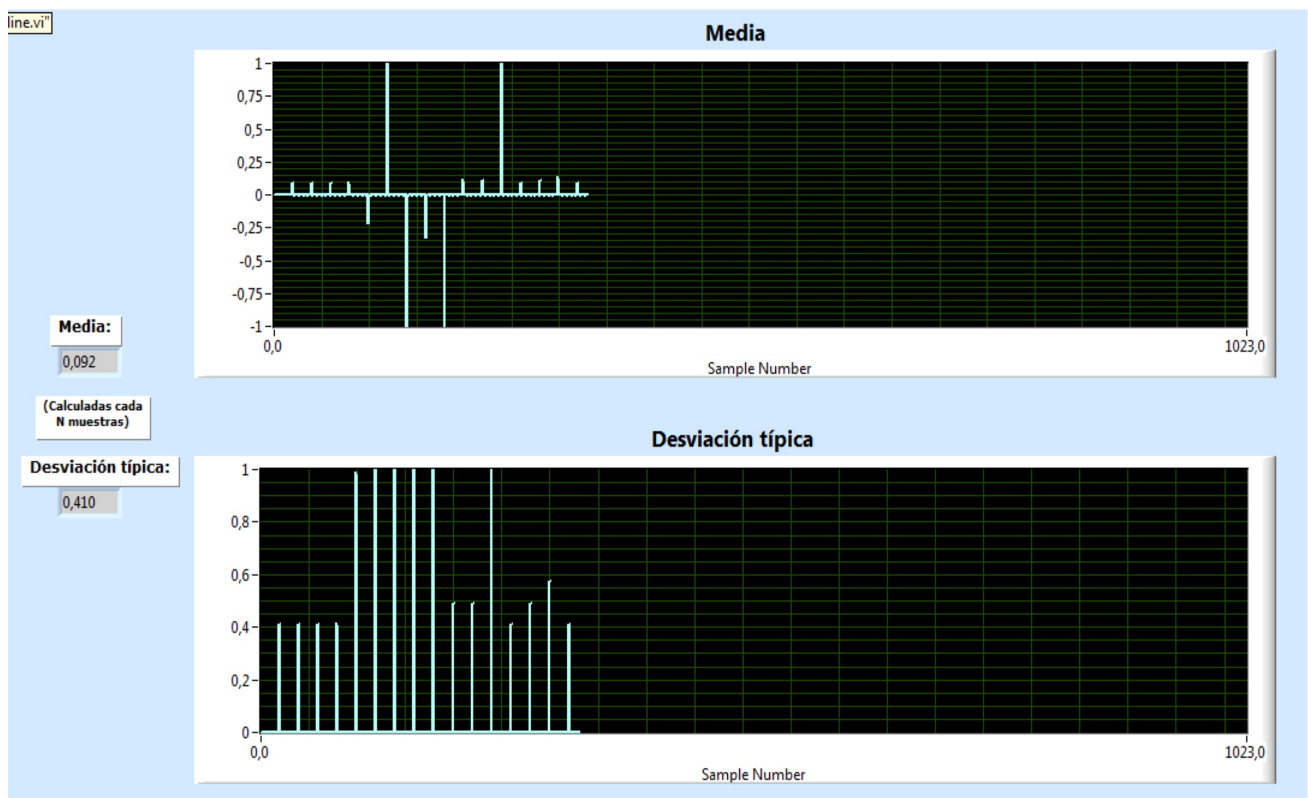


Figura 44. Panel frontal de media y desviación sobre N muestras

Vamos a detallar ahora la implementación de los siguientes datos a extraer, sobre los que también calculamos además la media y desviación con el procedimiento explicado.

5.2.5 Rangos de variación de los giros

Estos valores también los tenemos extraídos en el artículo de Xuesong Wang, Chuan Xu (2015) [24]. Podemos ver los resultados ofrecidos en la tabla 2, se corresponden con el campo SW_Range_1, …, SW_Range_5.

Lo que hacemos es clasificar las velocidades de giro extraídas en distintos rangos, obteniendo el número de muestras que cae en cada rango de velocidad de giro, calculando su porcentaje respecto al total de muestras, y la media y desviación típica para cada rango.

Para clasificar cada muestra en el rango de velocidad correspondiente, la evaluamos empleando el siguiente diagrama de flujo:

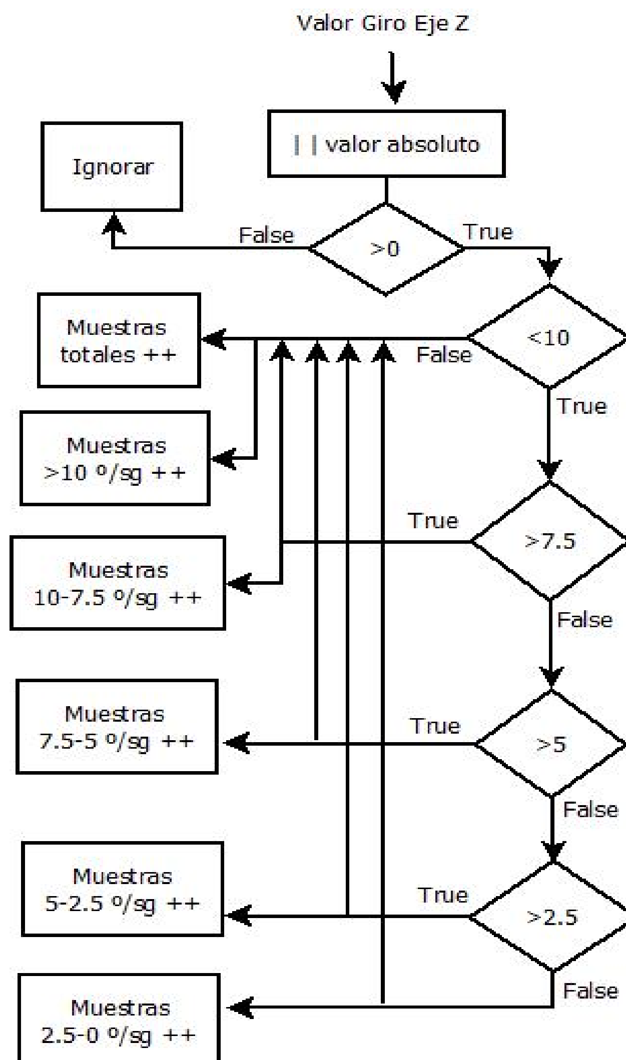


Figura 45. Diagrama flujo de rangos

Para implementar este diagrama de flujo en nuestro diagrama de bloques de LABVIEW necesitamos una estructura de *case* anidados:

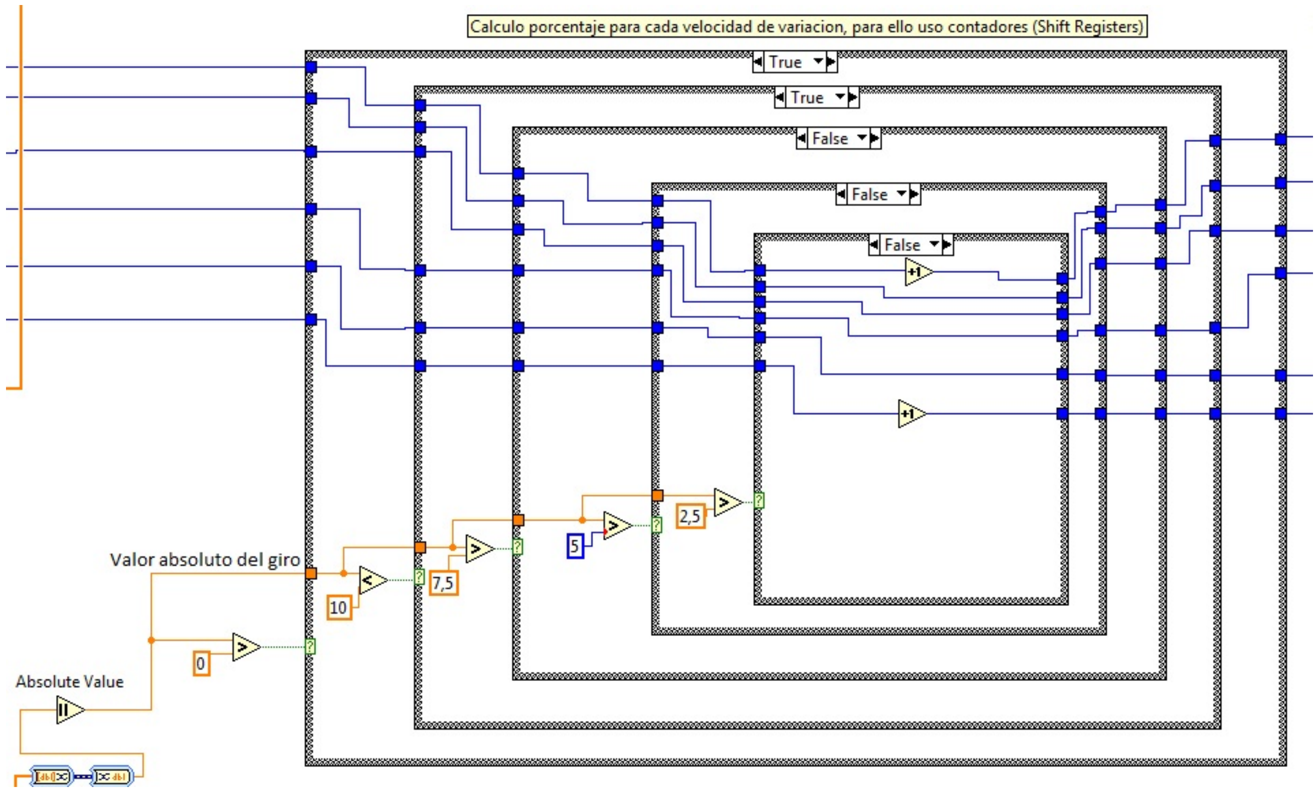


Figura 46. Diagrama de bloques de rangos

El resultado de esta toma de muestras, lo podemos ver en el panel frontal:

MEDIA Y DESVIACION TIPICA SOBRE DISTINTOS VALORES DE LA VELOCIDAD ANGULAR:

Muestras totales	Muestras >10	Muestras 10-7.5	Muestras 7.5-5	Muestras 5-2.5	Muestras 2,5-0	[deg/sg]
360	201	3	1	15	140	
Muestras en porcentaje:	55,833333 %	0,833333 %	0,277778 %	4,166667 %	38,888889 %	
Media:	0,029	0,000	0,000	0,002	0,018	
Desviación típica:	0,130	0,002	0,001	0,010	0,082	

Figura 47. Panel frontal de rangos

La siguiente medida que tenemos que tomar y vamos a implementar es el número de pasos por cero.

5.2.6 Pasos por cero (Zero-Crossings)

El cruce por cero de una señal o gráfica es un punto en el que cambia el signo de la misma:

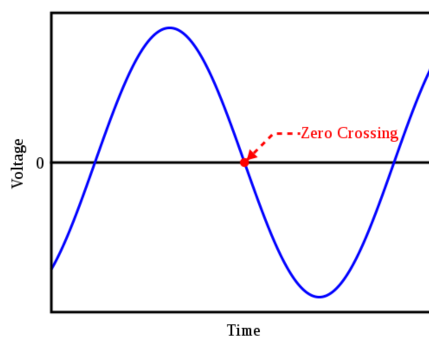


Figura 48. Paso por cero

Tenemos referencias del uso de esta medida del volante para intentar monitorizar el posible estado de somnolencia del conductor en el artículo de Sajjad Samiee, Shahram Azadi, Reza Kazemi, Ali Nahvi y Arno Eichberger. (2014). [16]. Según este artículo, un conductor que está alerta corrige continuamente o con mayor asiduidad la posición del volante para ajustarse al trazado, y por tanto, genera un número mayor de pasos por cero que un conductor somnoliento.

Para implementarlo en LABVIEW, usamos un bloque vi predefinido en sus librerías, *ZeroCrossingPtByPt.vi*:

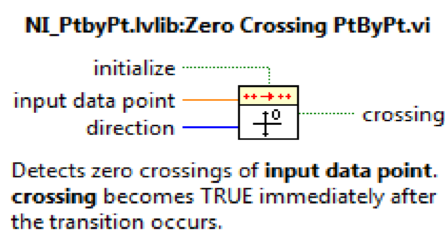


Figura 49. Vi de pasos por cero

Uno de los problemas a corregir es que variaciones muy mínimas, incluso fruto de la propia calibración del giróscopo, podían producir pasos por cero. Para ello establecimos un umbral experimental de 15 [grados/sg] para hacer la medida más precisa.

Hecho esto, la opción más lógica es fijar el valor a 0 si no supera el umbral, pero esto conllevaba que ese valor fijado a 0 fuera cuantificado como un paso por cero en el bloque de LABVIEW, luego si no se supera el umbral fijamos el valor de entrada al bloque, por ejemplo, a 0.5.

Esto genera un nuevo problema a resolver; la aparición de dos posibles casos de falso cruce por cero:

- Si, partiendo de esta posición, giramos el volante/giróscopo un valor mayor de -0.5, estaremos haciendo un falso cruce por cero.
- Si partiendo de un valor negativo, no superamos el umbral, el siguiente valor introducido al bloque será 0.5, lo que supondrá un cambio de signo que será contabilizado como un falso paso por cero.

Para solucionar estas cuestiones, usamos dos variables (o *Shift Registers* como vimos para LABVIEW). Ambas almacenan el valor de la iteración inmediatamente anterior del bucle, para detectar si estamos en algunos de los dos casos anteriores, y no contabilizar ese cruce por cero. Por ejemplo, en el segundo caso, usamos este registro con una puerta AND, que verifica si se cumple que el dato registrado anterior era negativo (menor que el umbral) y el valor de la iteración actual es 0, no contamos el paso por cero, es decir, no incrementamos la variable o registro que almacena los pasos por cero, ni lo introducimos en la gráfica. Para el primer caso el procedimiento es similar.

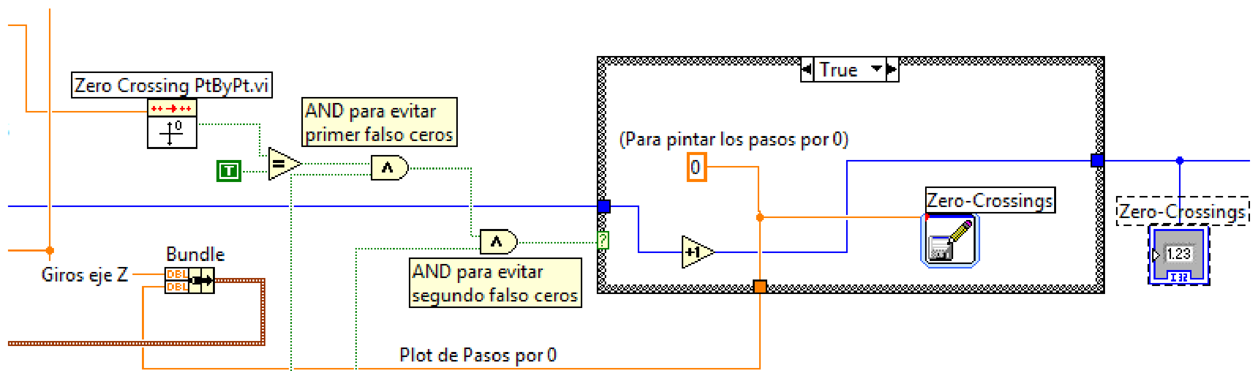


Figura 50. Diagrama de bloques de pasos por cero

Para dibujar la gráfica de pasos por cero, como se puede ver en la figura anterior, tenemos que unir dos arrays mediante un *bundle*, que es una estructura de datos de LABVIEW similar a un *cluster* o estructura de los lenguajes de programación habitual, uno de los arrays simplemente son los valores extraídos sin más para dibujar la gráfica, y otro es el que generamos en función de que haya un paso por cero o no.

Para mostrar este *bundle* al usuario necesitamos hacer uso de un objeto de tipo *chart*. La principal diferencia de este con el *Waveform Graph* habitual de LABVIEW, es que permite representar múltiples puntos de datos en cada intervalo temporal, en vez de solo un array 2-D de valor y tiempo.

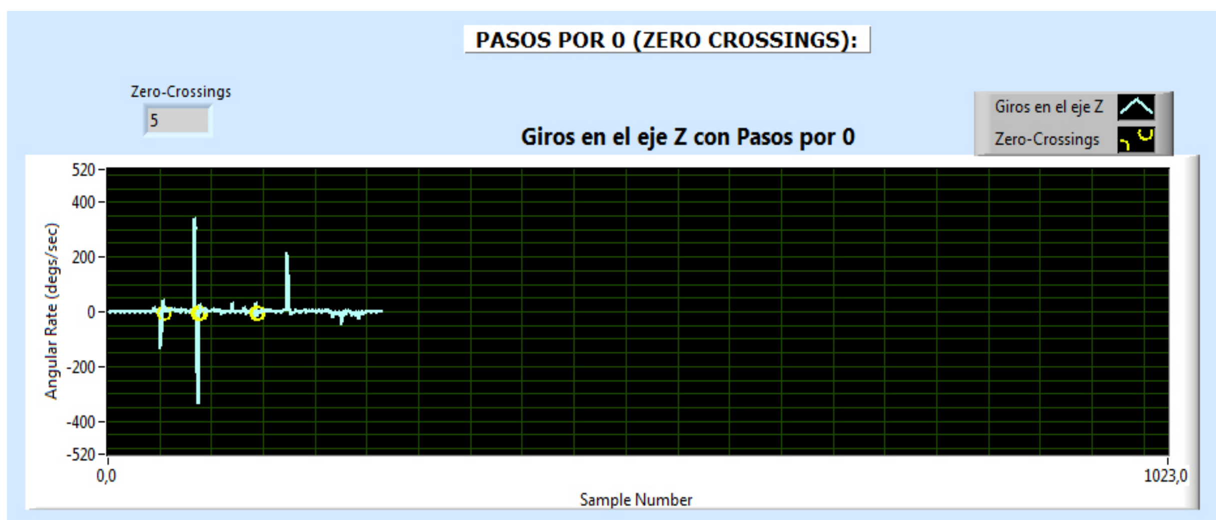


Figura 51. Panel frontal de pasos por cero

5.2.7 Valores máximos de los giros

Consideramos aquí el valor absoluto (obviando el signo que indica si el giro es a la derecha o a la izquierda) de los valores de velocidad de giro.

Este dato también se toma como referencia en el artículo de Sajjad Samiee, Shahram Azadi, Reza Kazemi, Ali Nahvi y Arno Eichberger. (2014). [16]. Según este, a diferencia de lo que pasaba con los cruces por cero, de media, un valor más alto del valor absoluto de la velocidad de giro se asocia con un conductor en un estado más bajo de alerta o somnoliento. Como el propio artículo nos indica, esto se debe a que un conductor somnoliento percibe las desviaciones del carril o camino correspondiente con mayor retardo, y por tanto, realiza giros más bruscos, de mayor velocidad. Es interesante independizar este análisis de los giros producidos por las propias curvas del trazado.

Para implementar este dato estadístico en LABVIEW, no nos interesa almacenar un conjunto de muestras en un array y obtener a posteriori el valor máximo de ellas, puesto que como en los pasos por cero, necesitamos pintar dinámicamente en la gráfica los máximos una vez sean alcanzados. Luego, necesitamos una variable o *shift register* que almacene el valor máximo alcanzado y lo actualice dinámicamente en función de los giros extraídos.

Fijamos de manera experimental esta variable con un umbral inicial de 15 grados/segundo, para que, al iniciar, cualquier mínimo giro no sea interpretado ya como un máximo.

De nuevo vamos a emplear un bloque VI que tenemos predefinido en LABVIEW, *peak detector*:

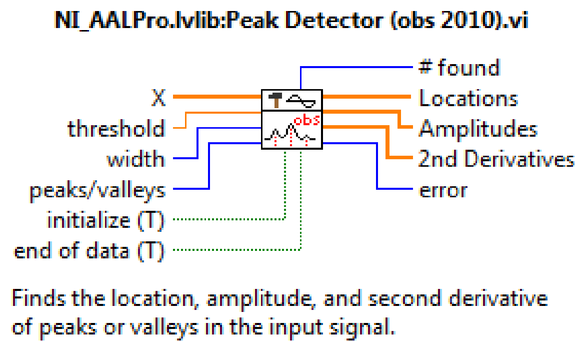


Figura 52. Vi de máximos absolutos

Aunque hemos dicho que no nos interesaba almacenar las muestras en un *array*, la entrada al bloque sí lo requiere, así que lo que haremos es iniciar a '0' un *array* de 1023 muestras (el número de muestras que se representa en cada instante temporal en el panel frontal) que se va actualizando de forma circular (con una función módulo como la que empleábamos en el cálculo de medias y desviaciones). Lo que hacemos es introducir esto al bloque, y en la entrada umbral ("*threshold*") que vemos en la figura, introducimos en cada momento el valor máximo almacenado en el register. Así, de manera dinámica, obtendremos un máximo cuando se supere el valor de máximo alcanzado ya en un momento determinado.

Este bloque nos detecta los picos (*peaks*) y los valles (*valleys*) de la gráfica o función, es decir, los máximos y mínimos respectivamente. A nosotros nos interesa quedarnos con el valor mayor de ambos, puesto que el máximo, en valor absoluto, podrá ser alcanzado con un giro negativo o positivo (volante a izquierda o derecha). Así que necesitamos implementar dos copias de este bloque, una para obtener los picos y otra para obtener los valles.

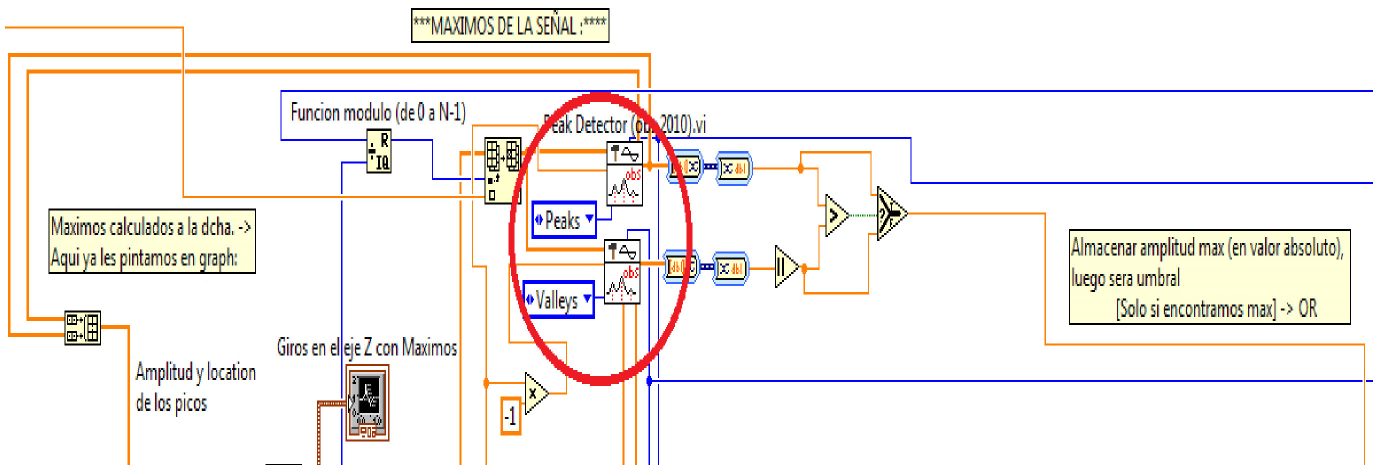


Figura 53. Diagrama de bloques de máximos

Así, en resumen, cuando un valor de giro supera el umbral introducido a los bloques (el valor máximo alcanzado actual), el bloque detectará un pico o valle, y nos quedamos con el valor absoluto mayor de los dos, que será nuestro nuevo máximo. Así, al final siempre tendremos el valor máximo alcanzado.

Para representar los máximos en una gráfica empleamos un procedimiento muy similar al usado en los pasos por cero; un *cluster* con un *array* 2-D para los valores de giro normales, y otro *array* para dibujar los máximos sobre esta gráfica, representado todo ello con un "*waveform chart"* de LABVIEW.

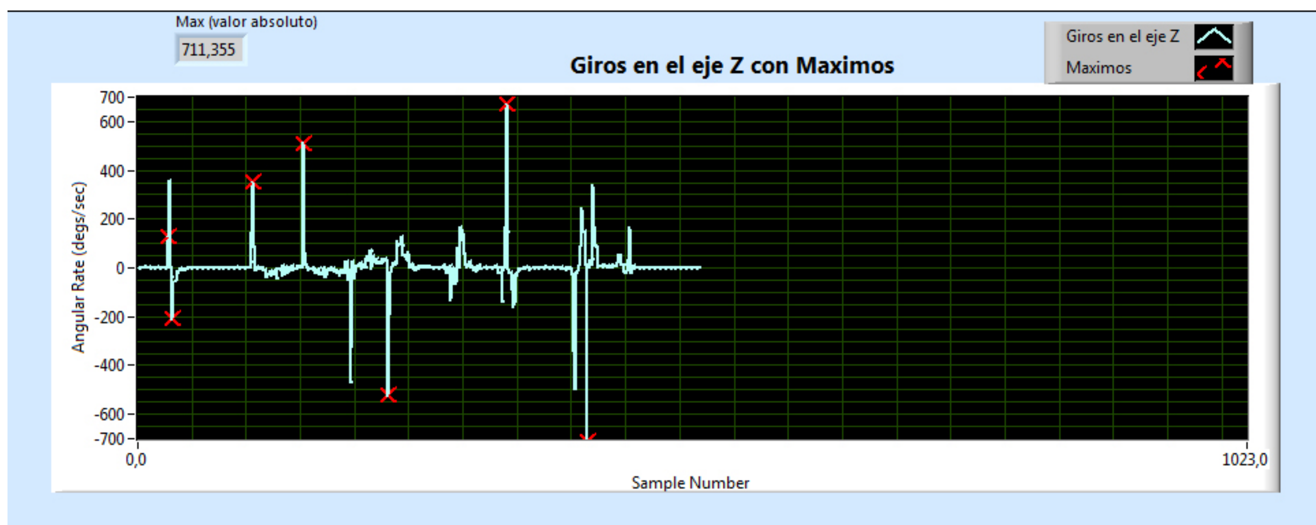


Figura 54. Panel frontal de máximos

5.2.8 Media y Desviación típica sobre total de muestras

Finalmente, como último dato estadístico que extraeremos de LABVIEW, calculamos la media y desviación típica sobre el total de medias y desviaciones calculadas sucesivamente con 'N' (=20) muestras, como ya explicamos.

El hecho de estar lógicamente obligados a fijar un número de muestras sobre las que hacer los cálculos de media y desviación típica nos ponía en un compromiso en la elección del tamaño de estas 'N' muestras, como expusimos en *4.4.2.3*. Así, ahora al detener la aplicación lo que hacemos es calcular la media y desviación de todas las calculadas, paliando así los efectos de la elección de un tamaño de 'N' u otro.

Esto lo implementamos al pulsar el usuario el botón "stop" de la pestaña "/Control/" para detener la recepción de datos del giróscopo:

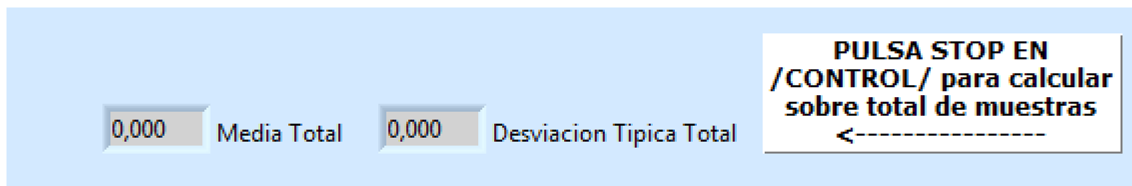


Figura 55. Media y desviación sobre muestras totales

Para ello lo que hacemos es crear dos variables o *shift registers* de LABVIEW, pero que esta vez serán *arrays* de tamaño no preestablecido, donde almacenaremos de manera dinámica las medias y desviaciones respectivamente calculadas, para luego calcular la media y desviación sobre todos los valores.

Esto implica un problema a resolver y es que el usuario pulse el botón "Stop" en un momento en el que hay muestras tomadas sobre las que aún no se han aplicado los cálculos de media y desviación y estas pueden representar un espectro muy significativo si el tamaño fijado para 'N' es muy grande. Para solucionar esta posible (más probable cuanto mayor sea N) incidencia, nos quedamos con una parte del array en el que íbamos almacenando las N

muestras con el bloque de LABVIEW "*Array Subset*", concretamente con la parte que tenga muestras escritas (recordar que *array* se llenaba sucesivamente de manera circular gracias a una función módulo) empleando el índice de la función módulo para saber esto. Posteriormente, calculamos la media y desviación típica de esas muestras, y el resultado lo concatenamos mediante un bloque "*Build Array*" con los *arrays* totales de media y desviación que describíamos al principio como *shift registers*.

Vemos la implementación práctica de este algoritmo en el siguiente fragmento del diagrama de bloques:

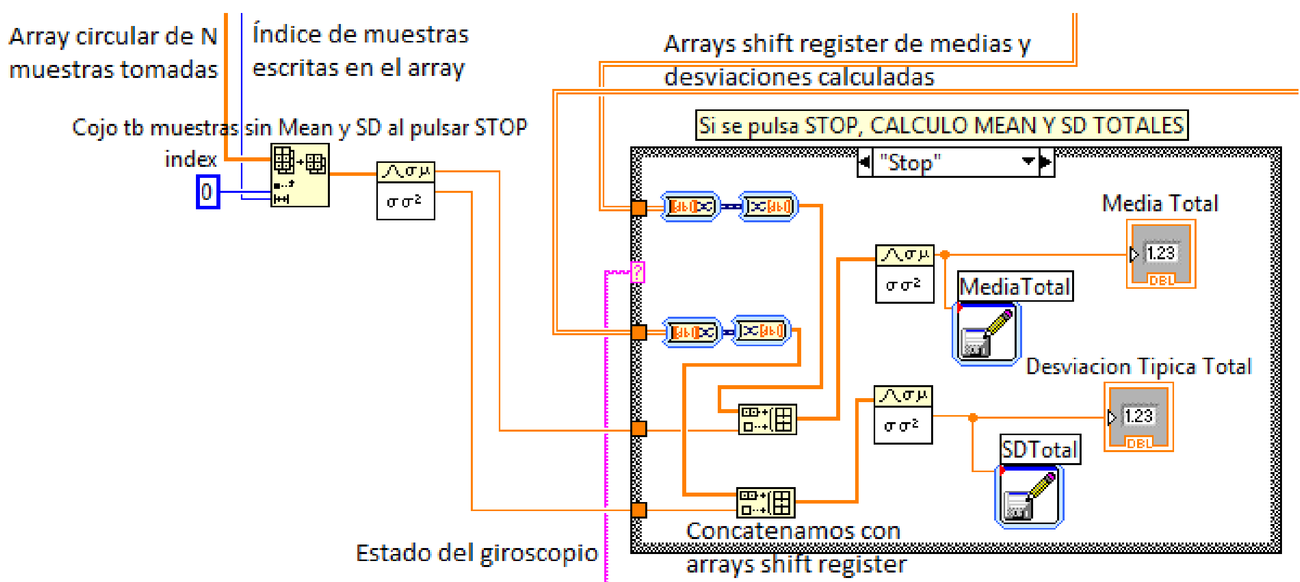


Figura 56. Diagrama de bloques de media y desviación total

5.2.9 Resumen de datos extraídos con LABVIEW

Mostramos a continuación una tabla, con todos los datos extraídos de nuestro giróscopo de Shimmer, cuyas implementaciones hemos detallado hasta aquí. Y el fichero correspondiente en el que almacenamos los resultados. *– Hay que recordar que los ficheros se almacenan en C:/Temp/ en formato .lvm-.*

Datos extraídos del giróscopo	Fichero en el que se almacenan
Velocidades simples de giro en los 3 ejes	Gyro.lvm
Número de vueltas completas del volante (Steering Wheel Reversals)	VueltasVolante.lvm
Media de vueltas completas	MediaVolante.lvm
Desviación típica de vueltas completas	SDVolante.lvm
Tiempo en reposo del volante	ReposoVolante.lvm
Media sobre N muestras	Media.lvm
Desviación típica sobre N muestras	DesviacionTipica.lvm
Media sobre el total de muestras	MediaTotal.lvm
Desviación sobre el total de muestras	SDTotal.lvm
Media y desviación de muestras >10 [Deg/s]	Media10.lvm y SD210.lvm
Media y desviación de muestras 10-7.5 [Deg/s]	Media10_75.lvm y SD10_75.lvm
Media y desviación de muestras 7.5-5 [Deg/s]	Media75_5.lvm y SD75_5.lvm
Media y desviación de muestras 5-2.5 [Deg/s]	Media5_25.lvm y SD5_25.lvm
Media y desviación de muestras 2.5-0 [Deg/s]	Media25_0.lvm y SD25_0.lvm
Pasos por cero (Zero-crossings)	ZeroCrossings.lvm
Máximos absolutos de giro	Maximos.lvm

Tabla 3. Resumen datos extraídos y ficheros

6. ANDROID

6.1 La aplicación base

Como ya hemos dicho, partimos de la aplicación para sensores Shimmer creada por Amine Khadmaoui (2014) [3].

Funcionalidades que nos ofrecía la aplicación base:

Vamos a hacer primeramente un repaso de las funciones de la aplicación base de la que partimos para desarrollar nuestro proyecto:

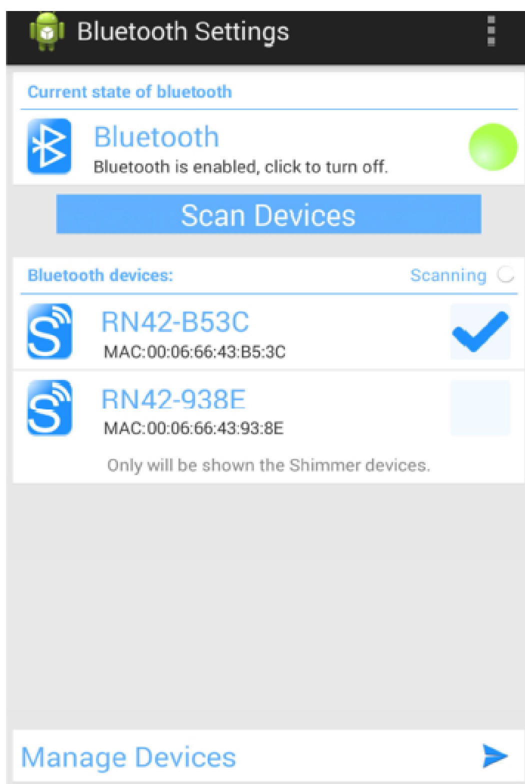


Figura 57. Pantalla incial ShimmerApp base

Teníamos una primera actividad (en Android a cada pantalla que muestre la aplicación la denominamos "actividad") con un botón para activar el Bluetooth del móvil.

Una vez activado el Bluetooth, podemos escanear en busca de dispositivos Shimmer (solo mostrará estos), y seleccionar los que queramos emplear. Desde aquí, podemos ir a la actividad de gestión de ficheros de los sensores (desde la barra de opciones de arriba) o ir a la actividad de recogida de datos de sensores (ManageDevices).

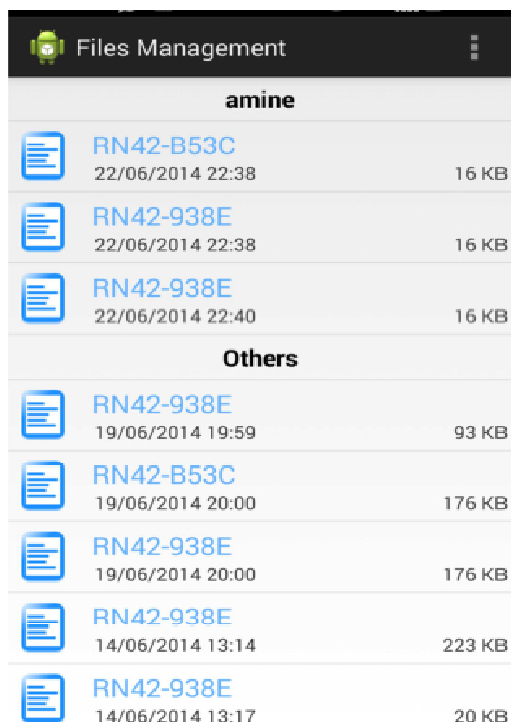


Figura 58. Manejo de ficheros en app base

Esta es la pantalla de la actividad de manejo de ficheros (*ManageFiles.java*).

Aquí podemos ver todos los ficheros guardados de datos de los sensores que han sido recogidos, clasificados con fecha y nombre de usuario.

Además, desde la barra de opciones de arriba podíamos eliminar todos o compartirlos con otra aplicación (mail, Dropbox,…).

Por último, teníamos la actividad de recogida de datos de los sensores (*ManageDevices.java*), que a su vez constaba de dos pestañas (*Fragments* en Android):

Una para mostrar los sensores conectados y el estado de sincronización con ellos (sensor transmitiendo, detenido, etc.), y otra para dibujar las gráficas con los datos básicos recogidos de estos, además de almacenarlos en los ficheros; acelerómetro y módulo Shimmer.

Vemos una captura de la interfaz descrita que teníamos para esta actividad en la siguiente figura:

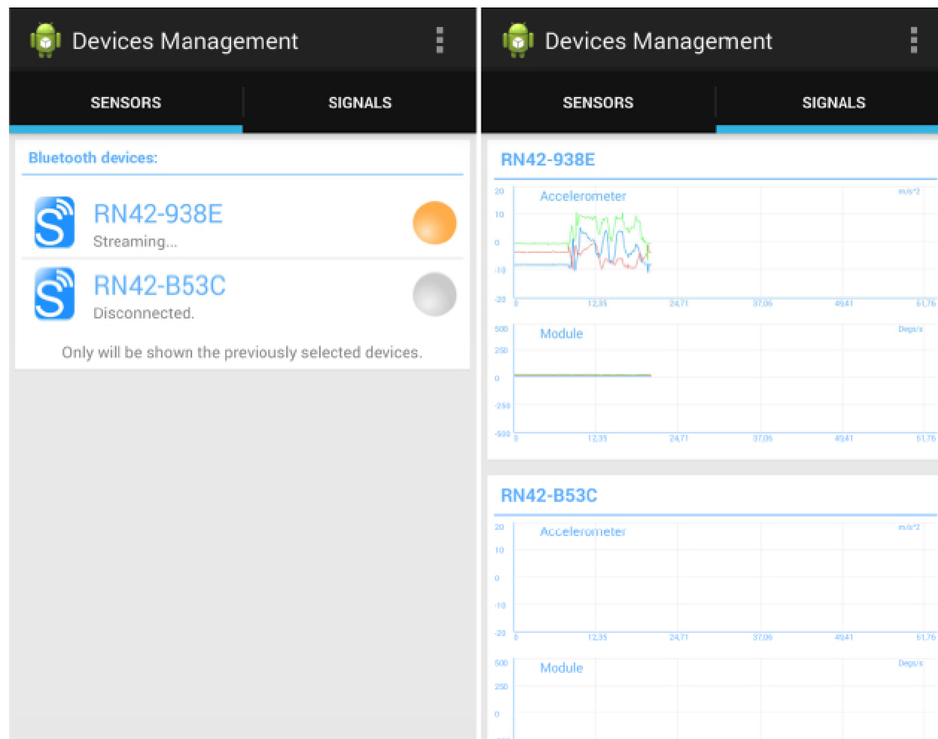


Figura 59. ManageDevices en la app base

6.2 Nuestra aplicación Android

6.2.1 Introducción

Nosotros lo que haremos en el presente proyecto es añadir dos funcionalidades principales a la aplicación anteriormente expuesta:

- Una recogida y exposición de datos más extensa, en caso de que uno de los sensores conectados sea nuestro giróscopo, recopilando y mostrando la misma información que desarrollamos en la aplicación para LABVIEW (ver 4.4.2). Este punto es el que vamos a desarrollar a continuación.
- Una comunicación vía *online* con cualquiera de los escenarios del simulador de conducción desarrollados en Unity. Este punto lo veremos

en el capítulo siete.

Dentro de la aplicación, iremos a una funcionalidad u otra activando o desmarcando un botón (*Toggle Button* en Android) que implementamos en la actividad principal de la aplicación. En caso de estar marcado, nos dirigiremos a la actividad de comunicación con Unity que veremos en el capítulo 7 del presente proyecto. Si lo dejamos desmarcado, veremos la interfaz que vamos a explicar a continuación, construida a partir de la aplicación base que hemos comentado.

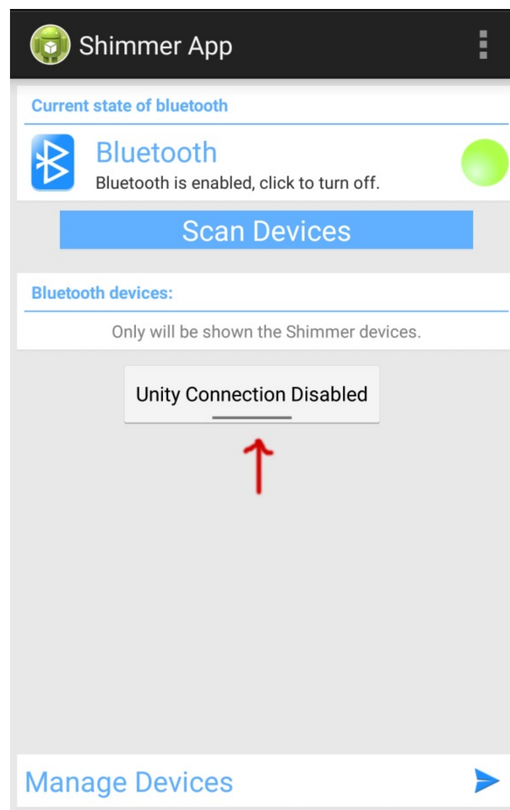


Figura 60. Botón de selección de actividad

Antes de empezar a desarrollar los puntos que habíamos implementado en LABVIEW y que vamos a implementar aquí en Android, comentamos algunos cambios hechos en primer lugar:

- Lo primero que hicimos es corregir la gráfica del giróscopo, ya que los datos no estaban bien calibrados (ver *Shimmer 9DOF Calibration* en 3.2.2).

Para ello, en la clase *GraphView.java*, que se encarga de la representación de las gráficas de la aplicación, en el método *setAdjustement()* que es llamado desde *ManageDevices.java* (la actividad principal como ya hemos visto) para representar las gráficas, creamos un set de unidades nuevas (*u12a*), que solo emplearemos para nuestro giróscopo Shimmer:

```
else if (dataType.equals("u12a")){setMaxValue(4095);offset=200;}
     //u12a, particular, hecho para el giróscopo
```

- Otra cuestión a corregir era mostrar solo los datos del eje Z, que es el que se corresponderá con los movimientos a izquierda o derecha del volante, puesto que inicialmente se mostraban en la gráfica del giróscopo los tres ejes espaciales. Para ello primero tenemos que entender un poco la estructura de datos de los sensores que nos ofrece la API de la librería de Shimmer [19]:

Tenemos una clase *ObjectCluster,* que contiene un *Multimap* de JAVA (es como un diccionario de Python, donde tenemos una lista de elementos en pares clave:valor) llamado *PropertyCluster* . Aquí cada clave (key) representa una propiedad, y cada valor el *FormatCluster* de esa propiedad. Este *FormatCluster* contiene tanto el formato, como las unidades, como los datos de dicha propiedad.

La razón de utilizar este formato, según el fabricante, es dotar a los controladores de cierta escalabilidad, es decir, permitir añadir nuevas propiedades tales como la aceleración o la orientación lineal.

Podemos resumir la estructura explicada en la siguiente figura:

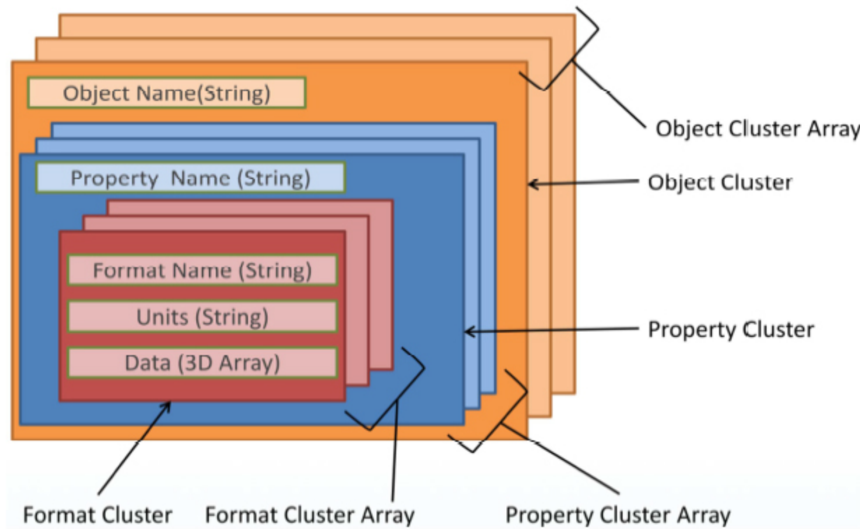


Figura 61. Estructura de datos en librería de Shimmer

Vemos el extracto del código donde nos quedamos con los datos del eje z (*DataProcessing()->ManageDevices.java*):

```
//---------------Para el GIRÓSCOPO solo queremos giros eje z---
if(Devices.get(Dev).getName().equals("RN42-B426")){
//SI TENEMOS GIRÓSCOPO-> Procesamos los datos. Para el Giróscopo solo queremos sacar eje Z
	for(j=0; j<ModuleNames[Dev].length;j++){
		Collection<FormatCluster> gyroZFormats=objectCluster.mPropertyCluster.get("Gyroscope Z"); //Datos para grafica

		formatCluster=((FormatCluster)ObjectCluster.returnFormatCluster(gyroZFormats, "RAW"));

		if(formatCluster!=null)
			ModuleDataArray[j] = (double)formatCluster.mData;

		formats = objectCluster.mPropertyCluster.get(ModuleNames[Dev][j]);
		formatCluster = ((FormatCluster)ObjectCluster.returnFormatCluster(formats,"CAL")); // Datos para guardar en fichero, unidades conocidas.
		if(formatCluster!=null)
			TotalDataArray[i+j] = formatCluster.mData;
	}
```

Ahora ya podemos empezar a hacer las mismas implementaciones de análisis de datos para el giróscopo que hicimos en LABVIEW, para ello definimos un *Layout* (estructura de una interfaz de usuario en Android) particular para el giróscopo, que solo se infle en el caso de que este esté conectado para mostrar todos los datos que vamos a implementar: *plot_fragment_graph_gyroscope.xml*.

6.2.2 Leds de izquierda y derecha y manejo de datos

Para representar los leds que nos indican si el giro es a izquierda o derecha, como los que teníamos en LABVIEW, usamos la clase *Button* de Android. Los botones tienen que tener dos estilos, uno para encendido y otra para apagado, podemos guardar y referenciar ambos estilos en */res/drawable-hdpi/buttonshapeon y buttonshapeoff* (*hay que recordar que los nombres que usamos para los recursos [/res/] de nuestro proyecto Android deben ir en minúscula*).

Para encender un led u otro, y para todos los demás datos que vamos a procesar, creamos una nueva clase: *GyroscopeDataprocessing.java*.

En esta clase fijamos los mismos umbrales y valor de N=20 que empleábamos en LABVIEW:

```
//Umbrales y valores prefijados: (LOS MISMOS QUE EN LABVIEW)
     static final int N = 20;
     //Numero de muestras sobre las que calculamos medias y
     desviaciones
     static final int LEDS = 20;
     static final int POSITION = 5;
     static final int MAXIMOS = 15;
     static final int ZEROS = 15;
```

Definimos en esta clase los métodos que emplearemos para procesar los valores de giro extraídos. Tenemos uno principal con el que tratar cada valor de giro que obtenemos: *setValorgiro()*. Además tenemos su equivalente para

obtener dicho valor de giro almacenado en cada instante de muestreo: *getValorgiro()*.

Para encender o no los leds tenemos, en esta clase, el método *onLeds()*, que devuelve un entero en función de encender" un led u otro:

```
public int onLeds(){
//Devuelve: 0-ningun led enciende, 1-led izq, 2-led der
      int valordevuelto=0;
      if(valorgiro<-LEDS){ //Led IZQ
            valordevuelto=1;
      }
      if(valorgiro>LEDS){ //Led DER
            valordevuelto=2;
      }
      return valordevuelto;
}
```

Por tanto, el *Layout* que tenemos hasta ahora se muestra en la figura 62.

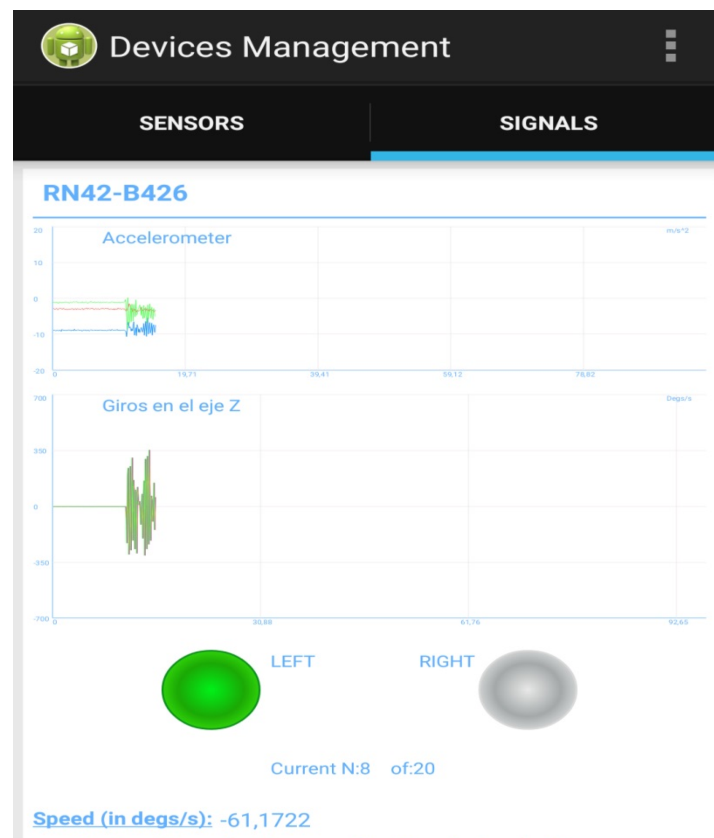


Figura 62. Leds en Android

6.2.3 Posición y vueltas completas del volante

Seguimos haciendo estas implementaciones en nuestra nueva clase: *GyroscopeDataprocessing.java.*

El algoritmo para calcular la posición del volante es similar a lo que hacíamos en LABVIEW, aunque su desarrollo, ahora en código JAVA, será lógicamente distinto. Responde a la siguiente ecuación:

Del número completo de posiciones del volante, de acuerdo al artículo ya citado de Xuesong Wang, Chuan Xu (2015) [24], y como hacíamos en LABVIEW, cada 'N' muestras calcularemos la media y desviación.

También añadimos al *Layout* el tiempo de ejecución del programa desde que se inicia la sincronización con los sensores, haciendo uso de las marcas de tiempos (*timestamp)* que obtenemos de los datos de los sensores.

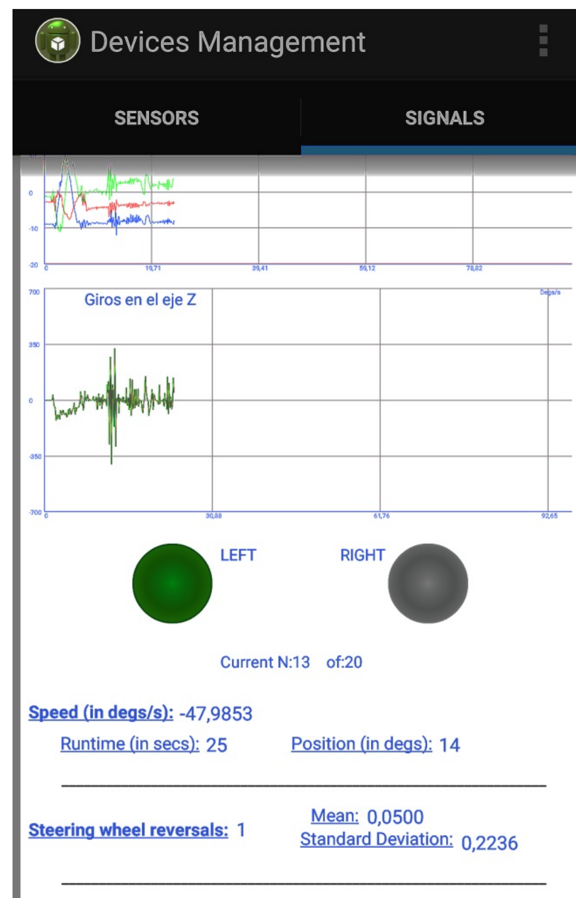


Figura 63. Layout con posición y vueltas del volante

6.2.4 Media y desviación

De nuevo, como hacíamos en LABVIEW, calcularemos la media y desviación típica (SD) para los datos de giro generales, para el número completo de vueltas del volante, como acabamos de comentar, y para los porcentajes de variación según rangos de velocidad.

Para ello creamos una nueva clase que se va a encargar de estos cálculos: *MeanAndSd.java* (ver ecuaciones 4 y 5).

El código necesario, por ejemplo, para calcular la desviación estándar de un conjunto de datos o *array* de enteros, de acuerdo a la ecuación 5, es el siguiente:

```
public double standardDeviationInt(int[] array){
//Metodo para calcular la desviacion tipica de un array de enteros
	List<Double> listofdifferences=new ArrayList<Double>();

		for (int i=0; i < array.length; i++){
			double difference=array[i]-this.mean;
			listofdifferences.add(difference);
		}
		List<Double> squares=new ArrayList<Double>();
		for(double difference:listofdifferences){
			double square=difference*difference;
			squares.add(square);
		}
		double sum=0.0;
		for(double number:squares){
			sum=sum+number;
		}
		double result=sum/(array.length-1);
		this.sd=Math.sqrt(result);
		return this.sd;
}
```

Vemos en la siguiente captura cómo mostramos la media y desviación típica calculada para ‘N’ muestras.

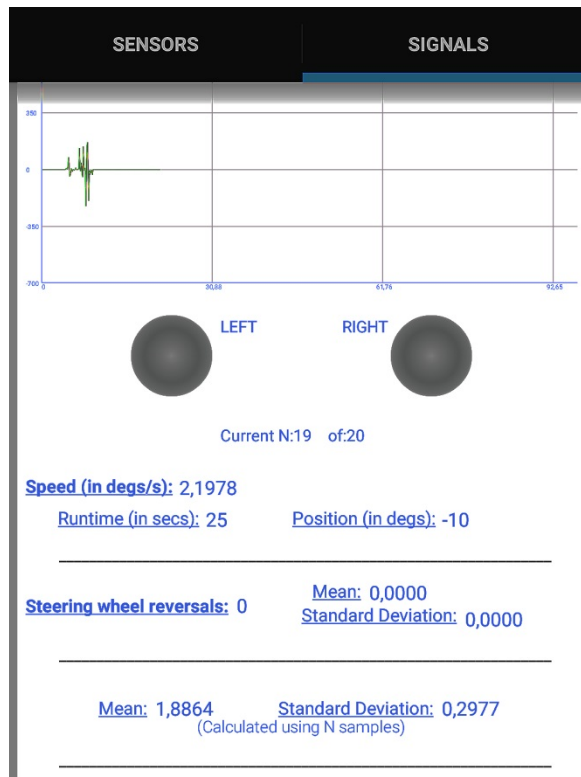


Figura 64. Media y desviación en Android

6.2.5 Rangos de variación de los giros

Estos valores también los tenemos extraídos en el artículo de Xuesong Wang, Chuan Xu (2015) [24]. Podemos ver los resultados ofrecidos en la tabla 2, se corresponden con el campo SW_Range_1, …, SW_Range_5.

El algoritmo que vamos a emplear es el mismo que hemos usado para LABVIEW (ver figura 40), así que podemos ver cómo difiere la implementación en lenguaje gráfico a la programación JAVA que empleamos ahora, en la que simplemente necesitamos estructuras “if” anidadas (implementamos esto en la función *setValorgiro()* para realizar el análisis sobre cada muestra que tomamos del giróscopo):

```
//****Analizamos rangos de veleocidad de la muestra****:
//Primero ponemos el giro en valor absoluto para quitaar el
signo:
double giroabs=Math.abs(giro);
//Ahora hacemos las comparaciones necesarias:
      if(giroabs<10){
            if(giroabs>7.5){
                  this.samples10_75=this.samples10_75+1;
            }
            else{
                  if(giroabs>5){
                        this.samples75_5=this.samples75_5+1;
                  }
                  else{
                        if(giroabs>2.5){
                              this.samples5_25=this.samples5_25+1;
                        }
                        else{
                              this.samples25_0=this.samples25_0+1;
                        }
                  }
            }
      }
      else{
            this.samplesmayor10=this.samplesmayor10+1;
      }
```

De nuevo, hacemos las modificaciones necesarias en nuestro *layout* (en la actividad principal: *ManageDevices.java*) para mostrar los resultados de estos cálculos, además de su media y desviación:

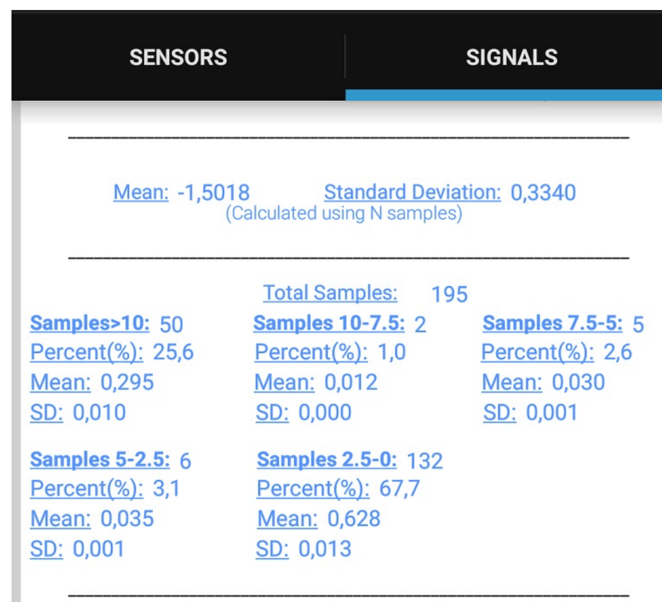


Figura 65. Rangos de giro en Android

6.2.6 Valores máximos de los giros

De nuevo pasamos a tomar la referencia del artículo de Sajjad Samiee, Shahram Azadi, Reza Kazemi, Ali Nahvi y Arno Eichberger (2014) [16].

Ahora, lógicamente, no vamos a usar ningún bloque VI de LABVIEW para la implementación, y es más fácil almacenar variables ahora que en LABVIEW. El proceso a seguir en cualquier caso será similar; un máximo almacenado que actúa como umbral a superar para establecer un máximo nuevo. Haremos el análisis de máximos para cada muestra tomada, de nuevo en la función *setValorgiro*():

```
//Maximos
    this.maximoalcanzado=false;
        if(Math.abs(giro)>MAXIMOS &&
        this.primermaximo==false){
            this.maximo=giro;
            //para el primer caso en el que solo superamos
            el umbral
            this.primermaximo=true;
            this.maximoalcanzado=true;
        }
        if(this.primermaximo==true &&
        Math.abs(giro)>Math.abs(this.maximo)){
            this.maximo=giro;
            this.maximoalcanzado=true;
        }
```

También tenemos que dibujar la gráfica donde pintar los máximos. Para ello en nuestra actividad principal tenemos que manejar un nuevo objeto de la clase que teníamos en la aplicación base para pintar las gráficas: *GraphView.java*. Dentro de esta clase crearemos un nuevo método para dibujar nuestros máximos, que llamamos: *setDataMax()*. En este método definimos un punto de un tamaño y color distinto para marcar los máximos en la gráfica:

```
addDataPoint(values2+offset, Scale, mColor[i%3], LastValue[i],
i);
    //Si tenemos que dibujar max:
    if(maxornot==true){
        addDataPointX(values2+offset, Scale, mColor[2],
        LastValue[i], i);//Pasamos color rojo para max
```

Tenemos que añadir la gráfica también en la clase *PlotFragment.java*, que se encarga de gestionar la pestaña que muestra los datos en la actividad principal, y en el recurso *plot_fragment_graph_gyroscope.xml*.

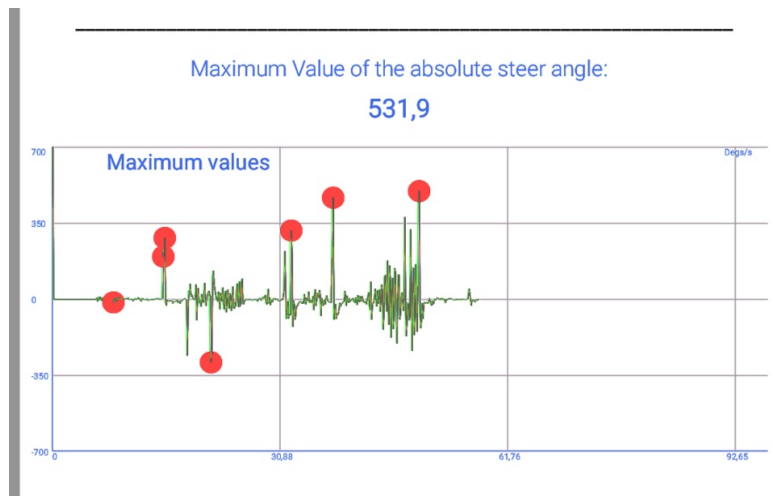


Figura 66. Valores máximos de giro en Android

6.2.7 Pasos por cero (Zero-crossings)

La implementación de un contador de pasos por cero, se ve ahora más simplificada sin usar bloques de LABVIEW. Tenemos que declarar una variable que almacene en cada instante de muestreo el valor de giro tomado para, en la siguiente iteración, comparar ese valor, que será ya anterior, con el dato actual tomado. Y ver así, si hay cambio de signo entre ambos valores; positivo a negativo o viceversa:

```
//Pasos por 0
     this.zeroalcanzado=false;
     if(Math.abs(giro)<ZEROS){
     //Si no superamos umbral fijamos la variacion a 0
          giro=0.0;
     }
     if(this.giroanterior<-ZEROS && giro>ZEROS){
     //Cruce desde un valor negativo a uno positivo
          this.zerocrossings=this.zerocrossings+1;
          this.zeroalcanzado=true;
```

```
}
if(this.giroanterior>ZEROS && giro<-ZEROS){
//Cruce desde un valor positivo a uno negativo
      this.zerocrossings=this.zerocrossings+1;
      this.zeroalcanzado=true;
}
```

-Hay que recordar que usamos los mismos umbrales que fijábamos en LABVIEW-

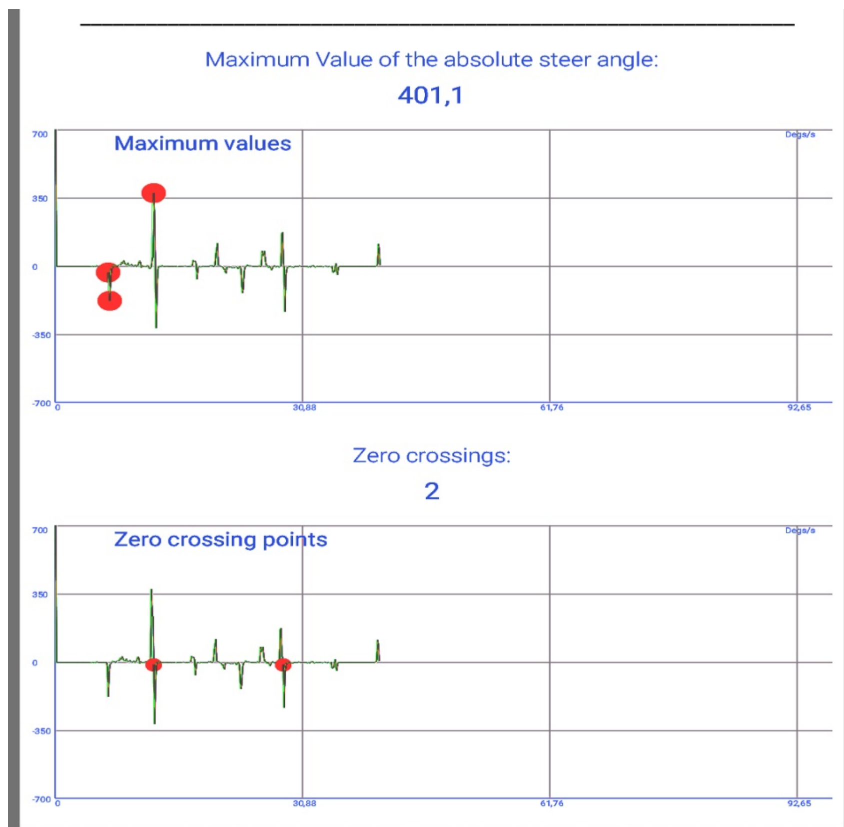


Figura 67. Gráficas de máximos y pasos por cero en Android

Para dibujar la gráfica de pasos por cero seguimos los mismos pasos que los detallados para la gráfica de máximos. Vemos el resultado final en la figura 67, con el contador además del número de pasos por cero hechos, y el valor absoluto (grados) máximo alcanzado.

6.2.8 Resumen de datos extraídos con Android

Los datos extraídos son, en resumen, los mismos que obtuvimos para LABVIEW (ver tabla 3). Aunque aquí, hemos compactado más los resultados incluyendo todas las medias y desviaciones típicas en el mismo fichero. Con uno para las calculadas de los datos generales, otro para las medias y desviaciones de las vueltas completas del volante, y otro para las medias y desviaciones del número de muestras clasificadas en cada rango de velocidad.

7. UNITY

7.1 Introducción a nuestro desarrollo Unity

Nuestro objetivo va a ser desarrollar la funcionalidad de comunicación entre la aplicación de Shimmer para Android que tenemos y cualquier proyecto Unity. Para que pueda empezar la sincronización con los sensores en la aplicación de manera automática una vez iniciado el escenario del simulador de conducción correspondiente, además de transferir los datos (ficheros) tomados de los sensores a nuestro PC en el que ha ejecutado el escenario de simulación y una vez haya finalizado el mismo.

Tendremos que generar un código estructurado, para así exportar la funcionalidad a cualquiera de los escenarios de conducción para Unity, que podremos emplear para realizar las pruebas y test sobre somnolencia y atención, tomando de manera conjunta datos de los sensores, con cualquiera de las dos aplicaciones (LABVIEW y Android) que hemos desarrollado.

7.2 Comunicación Unity-Android: primer acercamiento

Para realizar el proceso de comunicación de la aplicación Android para dispositivos móviles con la aplicación Unity para PC que vamos a crear para nuestras pruebas antes de trasladar el código a un simulador, necesitamos otra

aplicación desarrollada en Unity que instalaremos en el dispositivo Android, y que hará de "puente" entre ambas.

El proceso de comunicación, por tanto, puede resumirse de acuerdo al siguiente esquema:

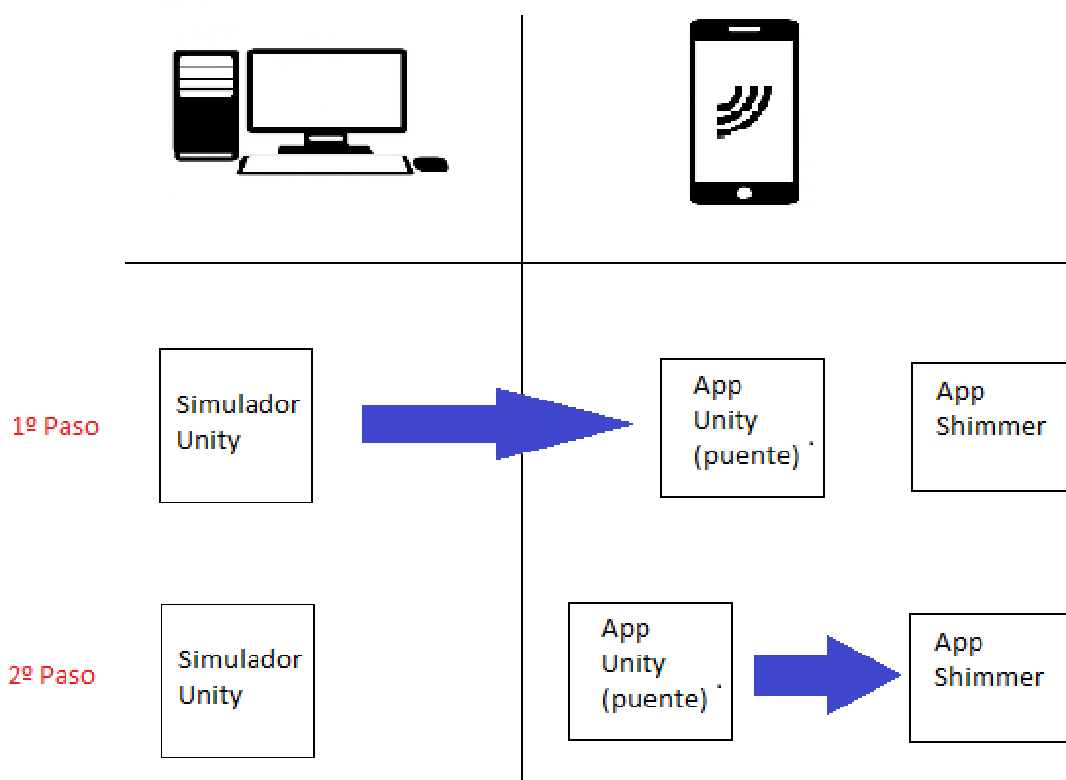


Figura 68. Comunicación Unity-Android: primer acercamiento

Por tanto, podemos dividir el proceso de comunicación en dos pasos diferenciados que detallaremos a continuación:

7.2.1 Primer Paso: Comunicación Unity-Unity

Para realizar la comunicación entre las dos aplicaciones Unity que se encuentran instaladas en dispositivos distintos usaremos una API estándar que originalmente ofrece Unity con funciones para realizar juegos multijugador, conocida como *Standard Unity Networking* [21].

En este proceso de la comunicación hacemos uso de un servidor Unity, que denominamos *Master Server*, y que Unity ofrece gratuitamente para las funcionalidades en red de los juegos de los desarrolladores. En este servidor registraremos nuestro servidor propio, que será la aplicación Unity para Android (el "puente"), instalada en el dispositivo móvil junto a la aplicación Shimmer para los sensores. Así, para registrar nuestro servidor propio (App Unity móvil) en el *Master Server* de Unity, indicamos el nombre de nuestro juego (aplicación), que actuará como identificador único (en nuestro caso "*com.ak.shimmer*"). Con esta referencia, el *Master Server* almacena también, cuando nos registremos, la correspondencia con un nombre que indiquemos para la "sala de juego" y la IP y puerto para el servidor propio, que como ya hemos dicho será nuestro móvil, con la aplicación Unity "puente" instalada.

Posteriormente, con esa misma referencia, el cliente mandará una solicitud al *Master Server* público de Unity, el cual le devolverá la IP y puerto del servidor propio (el dispositivo móvil en nuestro caso). Este conjunto de datos se denomina de manera estándar *HostData*.

A partir de este punto, obtenida la IP y puerto correspondiente, el cliente (en nuestro caso el PC con el simulador Unity) ya establecerá la comunicación directamente con el servidor propio (nuestro dispositivo móvil), cambiando el paradigma cliente-servidor por una conexión peer-to-peer.

-¡Por ello es necesario que ambos dispositivos se encuentren conectados a una misma red LAN con salida hacia internet!-

Podemos resumir este proceso de comunicación Unity-Unity esquemáticamente en la figura 69:

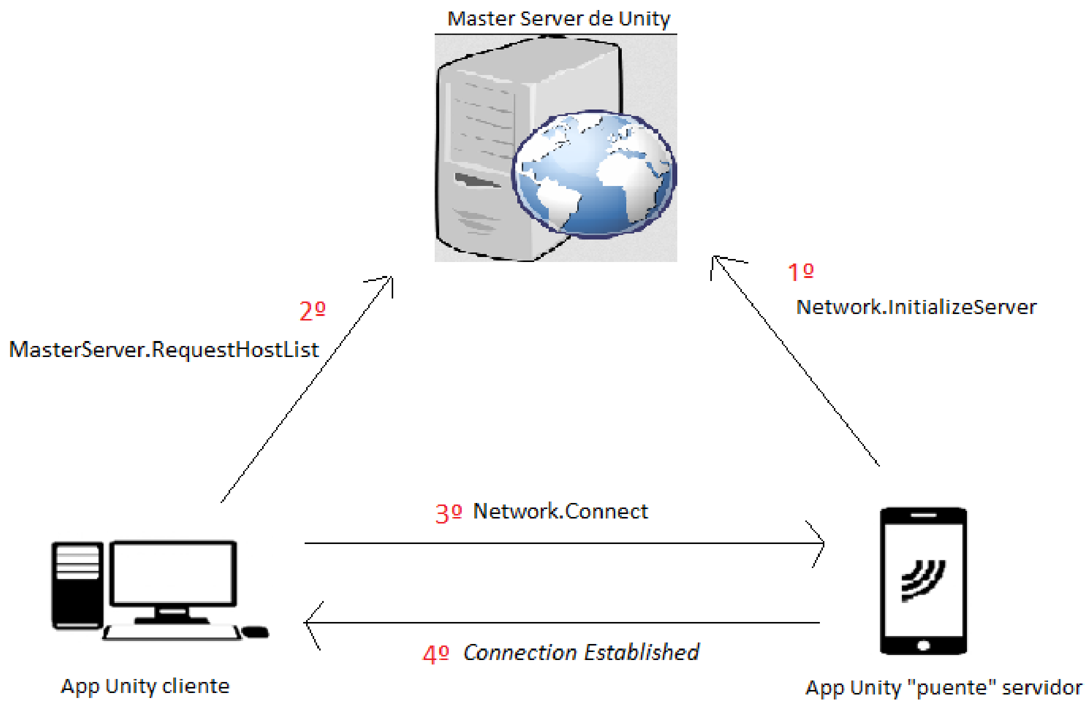


Figura 69. Comunicación con Master Server de Unity

A continuación, detallamos el código *C++* necesario en ambas apps (cliente y servidor propio), para establecer la comunicación relativa a estos pasos explicados:

Como hemos visto, lo primero es registrar el servidor propio (móvil Android) en el *Master Server* de Unity:

```
private const string typeName = "com.ak.shimmer"; //IdGameName
private const string gameName = "ShimmerSensoring"; //RoomName

    private void StartServer()
        {
            Network.InitializeServer(5,                    25000,
        !Network.HavePublicAddress());
```

```
        MasterServer.RegisterHost(typeName, gameName);
}
```

A este método *StartServer()* se le llama cuando pulsamos el botón de la interfaz de la aplicación Unity para Android:

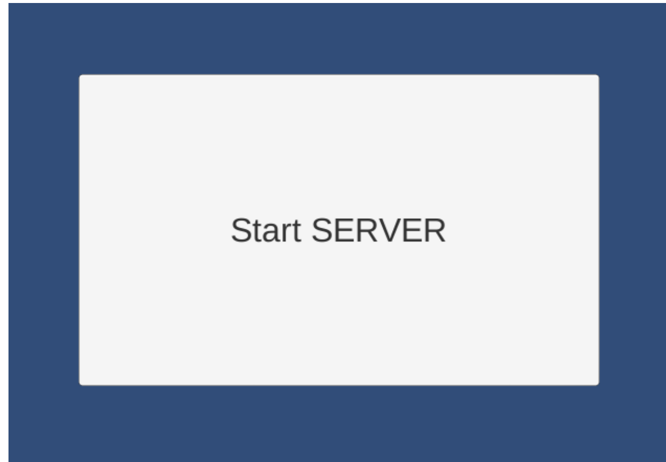


Figura 70. Botón app Unity de prueba

La función para iniciar el servidor es:

```
public static NetworkConnectionError InitializeServer(int connections, int listenPort, bool useNat);
```

Por lo que el parámetro '*5'* que pasábamos en el código, es el número de jugadores (conexiones del cliente en nuestro caso) máximas permitidas, que en nuestro proyecto no es muy importante, porque típicamente solo conectaremos con una aplicación Shimmer. El parámetro '25000' se corresponde con el puerto por defecto para el servidor de Unity.

Una vez inicializado el servidor, lo registramos en el *Master Server,* como ya hemos explicado, haciendo uso de la función *RegisterHost()*, que toma como parámetro el identificador establecido para el juego.

Una vez que el servidor ha sido inicializado correctamente, usamos el método *OnServerInitialized()* que detecta por defecto este evento para mostrar un mensaje en la consola de Unity:

```
void OnServerInitialized() //If  the  server  is  successfully
initialized, OnServerInitialized() will be called
           {
                Debug.Log("Server Initializied");
           }
```

Y una vez que nuestro servidor propio se ha registrado correctamente en el servidor Unity mediante la llamada a *MasterServer.RegisterHost(),* llamamos a este otro método detector de eventos para mostrar un cuadro con un mensaje informativo en el dispositivo móvil Android:

```
void OnMasterServerEvent(MasterServerEvent msEvent){
     if (msEvent == MasterServerEvent.RegistrationSucceeded){
                    Debug.Log("Registration Sucessfull");
                    registrocorrecto = true;
     }
}
```

Ahora que ya tenemos el servidor propio inicializado y registrado, lo siguiente será buscarlo en el *Master Server* de Unity y conectarnos a él con nuestro cliente (nuestra aplicación Unity de prueba, en este caso, el simulador en un futuro).

Para ello tenemos un botón en la interfaz Unity de la aplicación cliente:

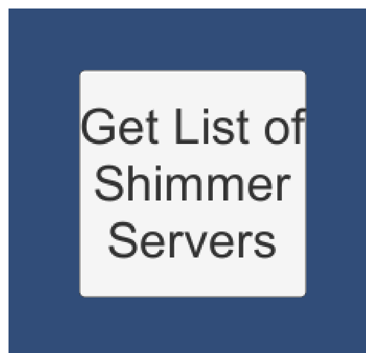


Figura 71. Botón en app Unity cliente

Al presionarlo, este llama al primer método que ejecutará el cliente: *RefreshHostList():*

```
private void RefreshHostList(){
        if (!isRefreshingHostList)
        {
            isRefreshingHostList = true;
            MasterServer.RequestHostList(typeName);
        }
}
```

Este método usa *RequestHostList* para solicitar al servidor de Unity la IP y puerto (*HostData*) de nuestro servidor propio (el dispositivo móvil).

Una vez que el *Master Server* de Unity nos responda, tenemos un método detector de eventos prefijado por Unity para detectarlo:

```
void OnMasterServerEvent(MasterServerEvent msEvent){
     if (msEvent == MasterServerEvent.HostListReceived)
                hostList = MasterServer.PollHostList();
}
```

Si el evento detectado coincide con un evento de recepción de HostData, los datos para conectarnos a nuestro servidor propio los almacenaremos en una variable que hemos declarado como:

```
private HostData[] hostList;
```

que, como vemos, es un array, por si interviene más de un servidor en el proceso. En nuestro caso, solamente usaremos el primer elemento del array, que contiene los datos de nuestro servidor propio, para establecer finalmente conexión con él.

Mostramos las "salas de juego" devueltas por el *Master Server* de Unity que se hayan correspondido con el nombre indicado en la petición:

```
void OnGUI(){
      if (!Network.isClient && !Network.isServer){
            if (hostList != null){
                  for (int i = 0; i < hostList.Length; i++){
                        if  (GUI.Button(new  Rect(400,  100  +
                        (110     *      i),     300,     100),
                        hostList[i].gameName))
                        JoinServer(hostList[i]);
                        //Al presionar el boton con el nombre
                        del server, nos unimos
```

Y mostramos, como vemos en el código, los servidores disponibles en la aplicación cliente Unity de nuestro PC:

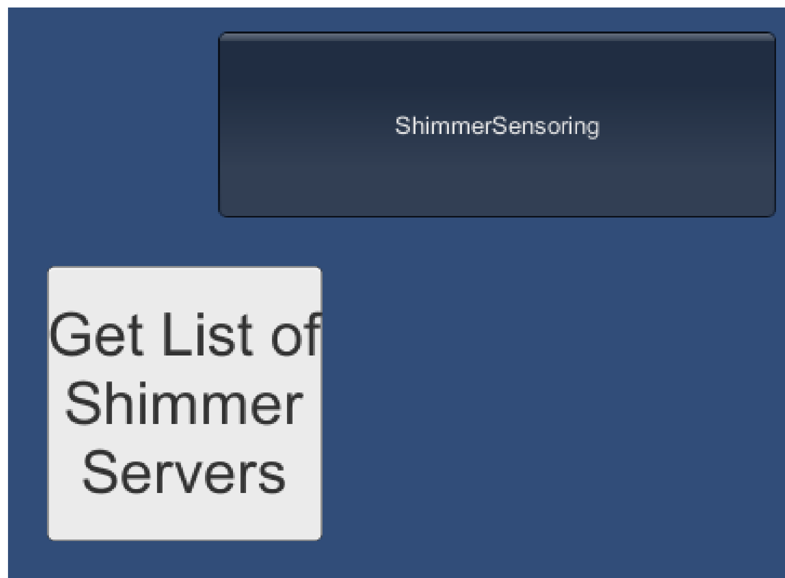


Figura 72. Server en app Unity PC

Y como podemos ver también en el código expuesto anteriormente, al presionar ese botón que muestra el nombre dado al servidor propio en el *Master Server* de Unity, llamamos al método *JoinServer()* que, pasándole como parámetro los datos (*HostData*) de nuestro servidor propio, establece la conexión peer-to-peer final con este.

```
private void JoinServer(HostData hostData)
    {
        Network.Connect(hostData);
    }
```

Una vez que la conexión se ha efectuado, usamos un método detector de eventos fijado por Unity en nuestro servidor (el dispositivo móvil), para mostrar un mensaje en pantalla y pasar a establecer la segunda parte del proceso; la comunicación mediante *RPC (Remote Procedure Call)* para poder enviar comandos u órdenes de ejecutar métodos al servidor (la aplicación puente que se comunica con Android).

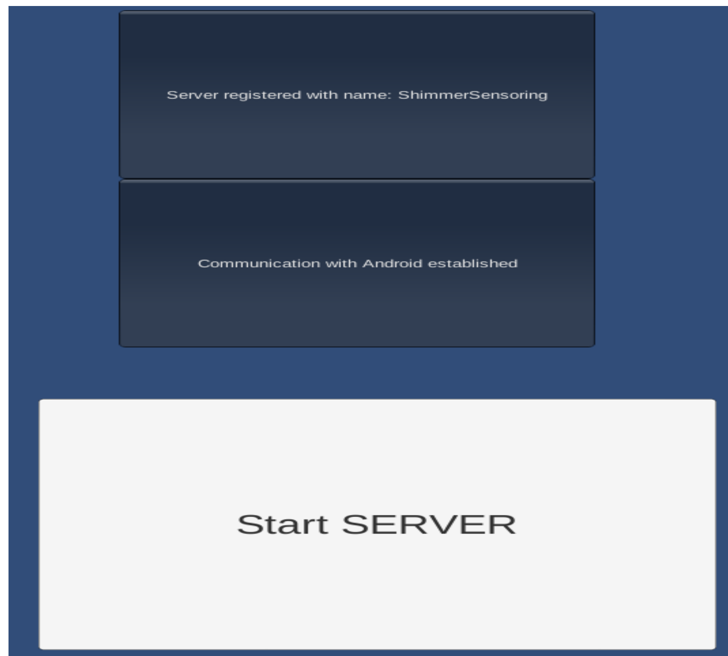


Figura 73. Conexión Unity establecida en app puente

Ahora que hemos establecido la conexión, mediante el método *OnConnectedToServer(),* mostramos en el cliente (la aplicación de PC), un botón para enviar ordenes al servidor mediante la comunicación RPC.

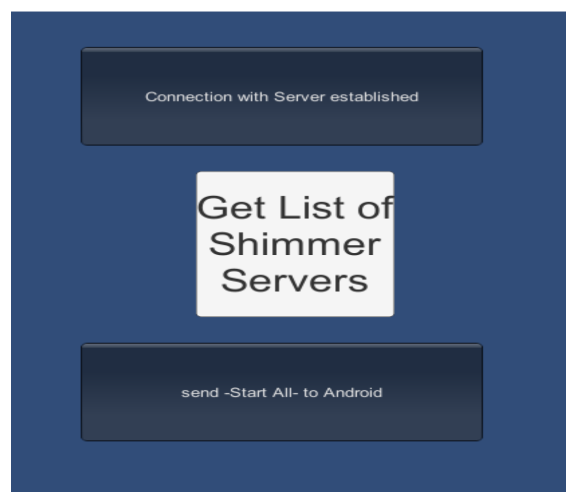


Figura 74. Botón de RPC

Al presionar el botón "*send –Start All- to Android*" se haría uso de otra implementación intermedia que ya hemos comentado, para realizar la comunicación final Unity-Unity; la llamada remota o RPC.

7.2.1.1 RPC (Remote Procedure call) [23]

Es una API que nos ofrece Unity para llamar a métodos remotos localizados en otra máquina a través de la red. Concretamente es una llamada que se ejecuta desde un objeto Unity (*GameObject*) en el cliente, a otro objeto en el servidor.

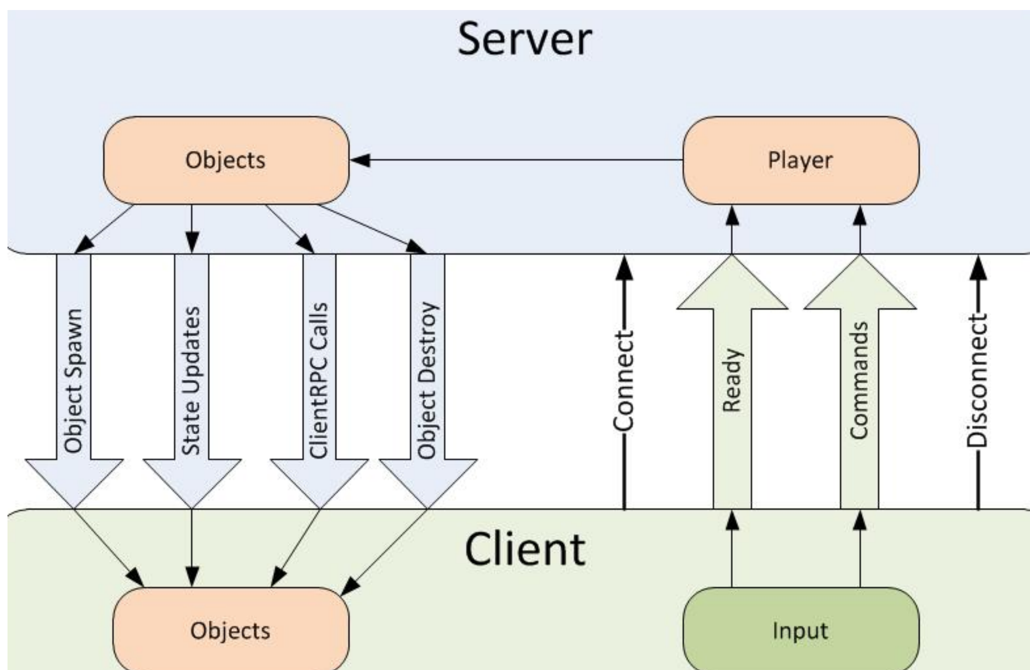


Figura 75. Esquema Remote Procedure Call

Por ello, un primer paso para poder realizar la comunicación es, en el elemento *Main Camera* de nuestras dos aplicaciones, donde teníamos anexados también los scripts con el código C++ que se ha detallado, adjuntar un componente Network View, como se muestra en la figura 76:

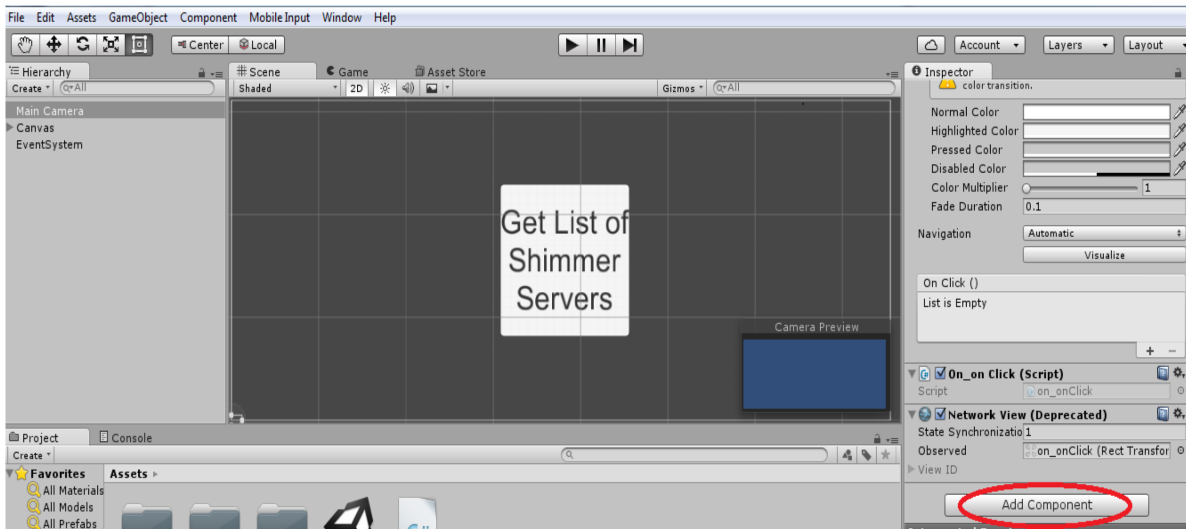


Figura 76. Añadir "Component" a GameObject

Además, tenemos que hacer que ese objeto (Network View) apunte a nuestro script (*On_onClick()*) donde tenemos el código, con el campo "Observed", como se muestra en la figura anterior.

En cuanto al código desarrollado, tenemos en el servidor (la aplicación Unity para Android instalada en el dispositivo móvil):

```
[RPC]
public void StartAllToAndroid(){
      conexstartallandroid = true;
            //CONEXION CON ANDROID--------------------------
      ----------
            //execute the below lines if being run on a
      Android device
      #if UNITY_ANDROID
      var androidJC = new
      AndroidJavaClass("com.unity3d.player.UnityPlayer");
      var jo =
      androidJC.GetStatic<AndroidJavaObject>("currentActivi
      ty");
      // Calling a Call method to which the current
      activity is passed -> para llamar a metodo statico no
      hace falta obtener el contexto -> probar llamando a
      metodo desde androidJC directamente
      var main = new
      AndroidJavaClass("com.ak.shimmer.MainActivity");
      main.CallStatic("calledFromUnity");
      #endif
}
```

Vemos que en los métodos que queremos que sean accesibles a través de la red, tenemos que añadir antes de su declaración: [RPC].

Por otra parte, el código desarrollado en el cliente (aplicación Unity para PC) para hacer la llamada remota RPC a este método del servidor es el siguiente:

Lo primero es declarar un objeto de tipo NetworkView:

```
NetworkView networkView;
networkView=GetComponent<NetworkView>();
```

Ahora, al presionar el botón, utilizamos ese Network View para realizar la llamada RPC al método remoto:

```
if (conexservercorrecta) { //Mostramos en pantalla que el cliente se ha conectado correctamente, y botones para enviar comandos al server
    GUI.Button (new Rect (240, 100, 300, 100), "Connection with Server established");
    if (GUI.Button (new Rect (240, 400, 300, 100), "send -Start All- to Android")) {
        networkView.RPC ("StartAllToAndroid",RPCMode.All);//Invocamos metodo remotodo en server con un Remote Procedure Call (RPC)
    }
}
```

Con esto realizaríamos las llamadas al método de la aplicación Android tantas veces como queramos pulsando el botón, una vez que la comunicación Unity cliente-servidor ha sido establecida. Mostramos además una ventana de información en la aplicación puente de Android, de que el método ha sido llamado desde el cliente.

```
//CONEXION CON ANDROID-------------------------------------
//execute the below lines if being run on a Android device
    #if UNITY_ANDROID
    var androidJC = new AndroidJavaClass("com.unity3d.player.UnityPlayer");
    var jo =
```

```
androidJC.GetStatic<AndroidJavaObject>("currentActivity");
          // Calling a Call method to which the current
     activity is passed -> para llamar a metodo statico no
     hace falta obtener el contexto -> probar llamando a
     metodo desde androidJC directamente

var                main                =               new
AndroidJavaClass("com.ak.shimmer.MainActivity");
main.CallStatic("calledFromUnity");
#endif
```

Este es el código, como ya hemos visto, que se corresponde con la comunicación con Android, que pasamos a detallar a continuación [22].

7.2.2 Segundo Paso: Comunicación Unity-Android

Lo que queremos ahora es ejecutar un método estático (*static*) de la actividad de Android, que llamamos *calledFromUnity()* desde el código Unity mostrado anteriormente, para que el proceso de comunicación se complete.

Para que esto sea posible, primero hay que integrar la aplicación Unity con nuestra aplicación Android desarrollada en el *IDE Eclipse.* Los pasos a seguir son los siguientes [22]:

1) En Unity, presionar *Build* para compilar normalmente nuestro proyecto (El nombre que demos a nuestro ejecutable es indiferente, pero el *TargetSDK* debe ser el mismo que para el proyecto Android en Java)

2) Una vez compilado, entrar en la carpeta de nuestro proyecto e ir a *Temp/StagingArea*. Copiar todos los ficheros de esta carpeta a una nueva carpeta (cualquiera).

3) Importar esta carpeta como un proyecto Android en Eclipse.

4) Una vez importado, indicar a Eclipse que este proyecto es una librería: botón derecho sobre el proyecto, pinchar en *Properties->Android* y marcar *is Library.*

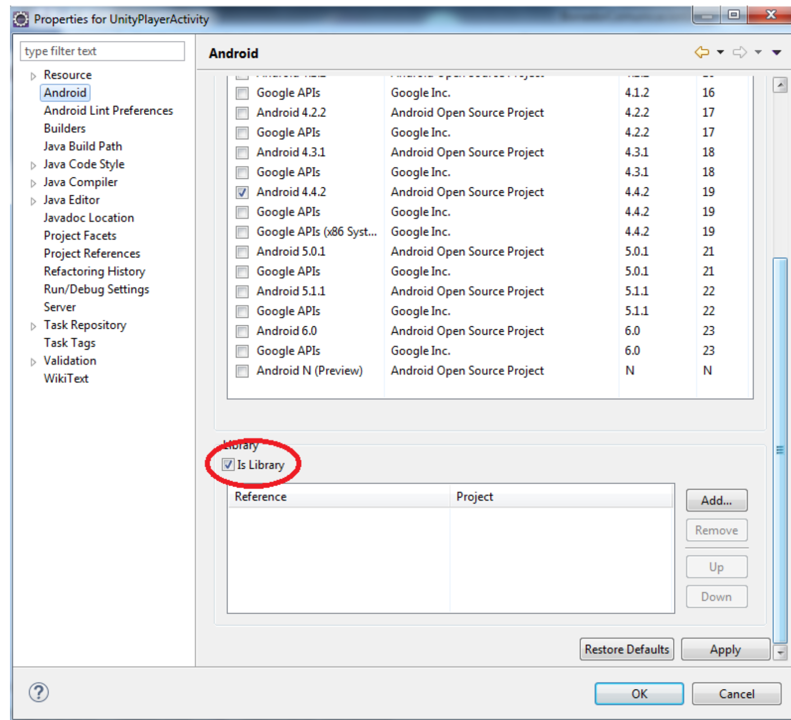


Figura 77. Marcar como librería en Eclipse

5) En el fichero androidmanifest.xml de nuestro proyecto Unity importado en Eclipse, borrar todas aquellas líneas que causen error.

6) Situarse ahora en Eclipse sobre el proyecto Android y hacer clic con el botón derecho, de nuevo *Properties->Android* y esta vez presionar add para añadir nuestro proyecto Unity como librería:

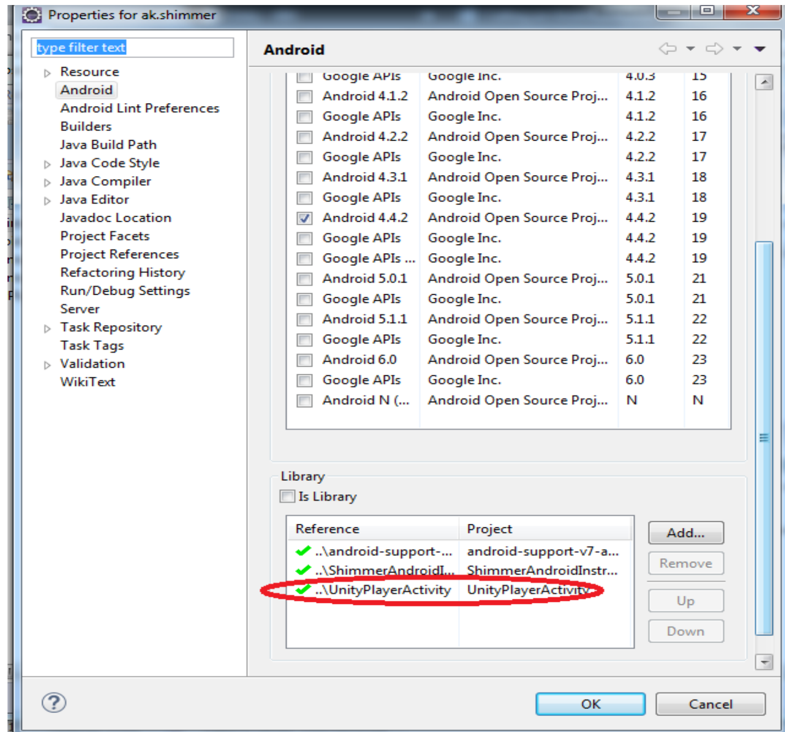


Figura 78. Añadir librería en Eclipse

7) Ahora tenemos que añadir otra librería predefinida de Unity a nuestro proyecto Eclipse, *classes.jar,* para ello:
Botón derecho sobre nuestro proyecto de Shimmer de nuevo, pinchar en *Properties->Java Build Path->Libraries* y hacer clic en *Add External Jar*, dirigirnos en la nueva ventana a nuestro directorio de instalación de Unity, a la ruta

\Editor\Data\PlaybackEngines\androidplayer\Variations\mono\Release\Classes

y seleccionar el fichero *clases.jar* que aparece.

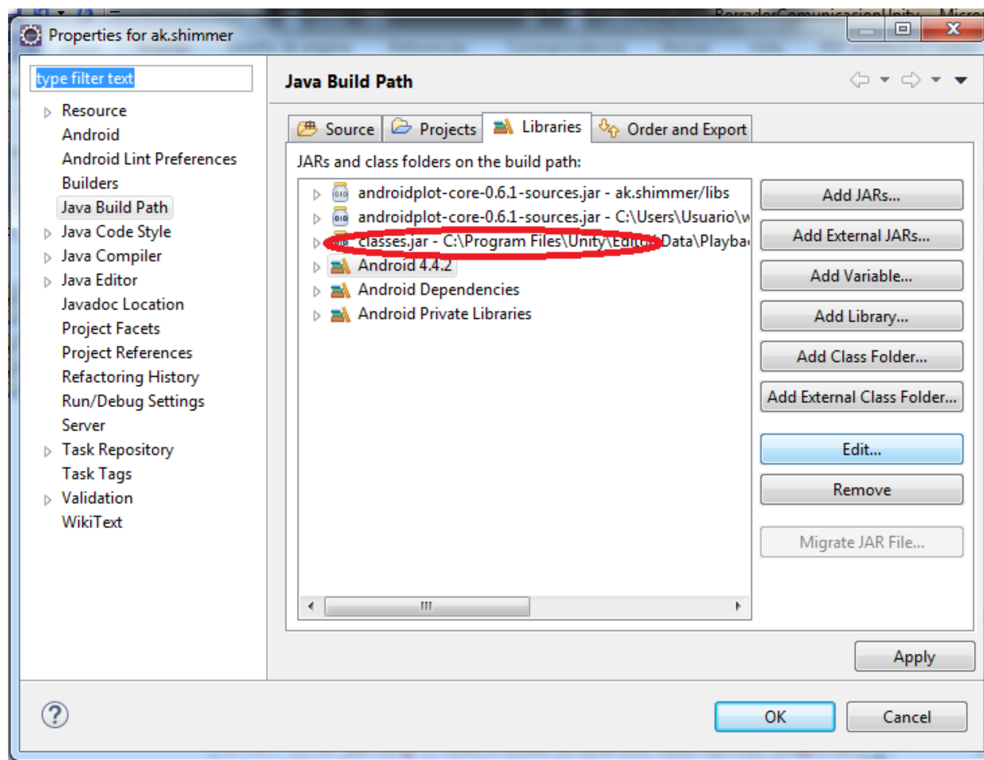


Figura 79. Añadir librería Classes.jar en Eclipse

A continuación, en esta ventana hacer clic sobre la pestaña *Order and Export* y marcar la librería *clases.jar*.

8) A continuación, dentro de la librería (proyecto) de Unity importada en Eclipse, mover todos los ficheros de las carpetas *assets* y *libs* a las mismas carpetas de nuestro proyecto Android de Shimmer. –Android no permite utilizar *assets* en proyectos marcados como librerías–.

9) Finalmente exportar nuestro proyecto resultante de eclipse, como un fichero Jar de Java, haciendo *File->Export->Java->Jar file*.

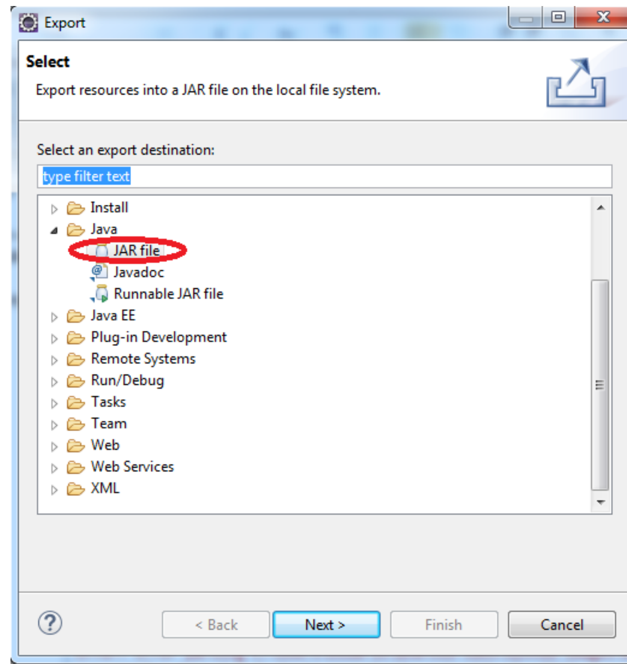


Figura 80. Exportar JAR en Eclipse

10) Guardar el fichero en la carpeta que queramos. Ahora lo tenemos que importar como un Plugin en Unity. Para ello, dentro de nuestro proyecto Unity, movemos este fichero a la ruta: *nuestro_proyecto\Assets\Plugins\Android* (Si no existen las carpetas las creamos, respetando las mayúsculas y minúsculas).

Con esto ya tendríamos la integración realizada. En cuanto a código, ya hemos mostrado el código C++ de Unity necesario para llamar al método.

En el fichero Java de nuestro proyecto Android, la clase en la que declaremos esa función debe heredar de *UnityPlayerActivity*, por ejemplo:

```
public class MainActivity extends UnityPlayerActivity{
```

Además de importar la librería correspondiente:

```
import com.unity3d.player.UnityPlayerActivity;
```

Por otra parte, en el fichero *androidmanifest.xml* de nuestro proyecto Android, debemos añadir lo siguiente a la declaración de la actividad de la clase correspondiente en la que añadamos nuestro método a ser llamado desde Unity:

```
<meta-data
    android:name="unityplayer.ForwardNativeEventsToDalvik"
    android:value="true" />
```

Con todo esto, terminaríamos con el proceso de comunicación Unity-Android, y podríamos, en teoría, llamar al método de nuestra aplicación Android mencionado anteriormente *calledFromUnity()*, desde Unity.

-Hay que remarcar que, ante un cambio en nuestro proyecto de Unity o Android, tendríamos que volver a realizar todos los pasos para la integración.-

7.2.2.1 Problema de la herencia múltiple

Cabe mencionar un problema existente en la integración de Unity y la aplicación Android.

Como hemos dicho antes, en nuestro proyecto Android, la clase que implementa el método que es llamado desde Unity debe heredar de *UnityPlayerActivity*, pero esto supone un problema en nuestra clase *ManageDevices*, actividad de Android en la que implementamos el manejo y presentación de los datos extraídos de los sensores, que ya debe heredar de *ActionBarActivity* para dar soporte a los métodos de manejo de la barra de acción y los fragmentos de dicha actividad.
Y, como sabemos, Java, como la mayoría de lenguajes de programación, no

soporta herencia multiple.

La programación orientada a objetos usa el concepto de herencia para extender la funcionalidad de los objetos. Cuando programamos un objeto, más tarde podemos necesitar crear un objeto muy parecido con solo ligeras diferencias (tal vez extender la funcionalidad de un objeto anterior en un nuevo contexto). Aquí es donde aparece la herencia. Un objeto que un programador "deriva" de otro objeto "base" hereda los métodos y variables de la clase y después puede añadirle más funcionalidad.
Esto puede dar lugar a intentar derivar una clase única de varias clases. Esto se conoce como "herencia múltiple" y puede provocar problemas, como el denominado "problema del diamante". Este problema ocurre cuando dos clases heredan de la misma clase (como la clase B y C derivan de la clase A), mientras que otra clase (D) deriva de B y C. Cuando se crea un objeto D, el sistema lo trata como un tipo de clase base. En el problema del diamante, el sistema no puede determinar de forma decisiva qué clase D (¿es tipo A-B-D o tipo A-C-D?) es la que causa problemas. Debido a estos problemas con la herencia múltiple, Java no la permite

Una solución que intentamos para esto, de manera infructuosa, fue modificar la clase *UnityPlayerActivity* nativa de Unity, para que heredara de *ActionBarActivity*, y así, al heredar de *UnityPlayerActivity*, ya estuviéramos heredando a su vez los métodos necesarios para el manejo de fragmentos y la barra de acción como hemos comentado antes.

Intentamos esto, aunque sin resultados. En cualquier caso, los pasos fueron los siguientes:

Primero, localizando el fichero fuente de Unity, UnityPlayerActivity.java, que se encuentra en:

```
<rutainstalacióndeUnity>/PlaybackEngines/AndroidPlayer/src/
```

Renombrando ese fichero y modificándolo para que herede de *ActionBarActivity*:

```
//Añadimos los import necesarios:
import android.support.v4.app.FragmentManager;
import android.support.v4.app.FragmentTransaction;
import android.support.v4.view.ViewPager;
import android.support.v7.app.ActionBar;
import android.support.v7.app.ActionBar.Tab;
import android.support.v7.app.ActionBarActivity;

public class MyUnityPlayerActivity extends
ActionBarActivity //Hacemos que herede de actionBar en
vez de Activity para solucionar problemad e herencia
multiple
```

A continuación, compilándolo para generar el fichero de bytecode de Java (.class), junto con la librería del SDK (Software Development Kit) de Android necesaria y el fichero classes.jar de Unity que contiene las clases Android de Unity, mediante el comando:

```
Javac -classpath "ruta_necesaria\android-support-
v4.jar;ruta_necesaria\android-support-v7-
appcompat.jar;ruta_instalacion\android-sdks\platforms\android-
X\android.jar;ruta_instalacion\Unity\Editor\Data\PlaybackEngines
\AndroidPlayer\Variations\mono\Release\Classes\classes.jar"
Nombre_puesto_para_nuevo_fichero_unity.java
```

-X es la versión de android objetivo de nuestra aplicación (Objetive Target).-

Y después, generando un fichero JAR con esta clase, con el comando:

```
Jar cvf Nombre_deseado.jar ruta_del_class_anterior\
Nombre_puesto_para_nuevo_fichero_unity.class
```

(En nuestro caso, el nombre puesto al nuevo fichero fuente fue MyUnityPlayerActivity.java).

Tras esto, en teoría solo restaba fijar estos ficheros como fuente para Unity. Copiando para ello el .jar generado a la ruta:

```
Mi_proyecto/Assets/Plugins/Android/
```

Y copiando a esa misma ruta, el fichero *AndroidManifest.xml* original de Unity desde su ruta:

```
Ruta_instalacionUnity\Unity\Editor\Data\PlaybackEngines\AndroidP
layer\Apk
```

Y modificando en este fichero, la fuente nativa de Android para la clase por el nombre puesto por nosotros al nuevo fichero:

```
<activity android:name="com.unity3d.player.UnityPlayerActivity"
```

Pero nada de esto dio finalmente resultado, así que, empleando parte de lo ya hecho y desarrollado hasta este punto, usamos un acercamiento distinto, que ha sido el puesto correctamente en funcionamiento; el empleo de sockets para establecer la comunicación.

Solucionando además el problema de la herencia múltiple con una nueva pantalla o actividad para la aplicación Android, con una interfaz muy simplificada y reducida, que no necesita heredar nada más que de la clase de Unity: *UnityPlayerActivity.*

7.3 Acercamiento definitivo a la comunicación Unity-Android: Sockets

Los *sockets* son una forma de comunicación entre procesos que se encuentran en diferentes máquinas de una red, los sockets proporcionan un punto de comunicación, empleando una arquitectura cliente-servidor, a través del protocolo de la capa de transporte TCP/UDP de Internet, por el cual se puede enviar o recibir información entre procesos. En nuestro caso, haremos uso de *sockets TCP*, por lo que estos quedan inconfundiblemente identificados en cada máquina por los siguientes elementos:

- Un par de direcciones IP, que identifican la computadora de origen y la remota.
- Un par de números de puerto, que identifican a un programa o proceso dentro de cada computadora.

Para establecer esta comunicación definitiva, seguimos los mismos dos pasos principales que ya establecimos anteriormente, puesto que mantenemos la aplicación Unity de puente en el móvil. Gracias a ella y a la API para el multiplayer networking de Unity ya comentado antes, podremos saber en el ordenador la dirección IP del móvil y viceversa, para así establecer el socket. (Teniendo en cuenta que ambas máquinas tienen que estar conectadas a la misma WiFi para pertenecer a la misma LAN).

Resumimos la comunicación implementada definitivamente en la siguiente figura 81.

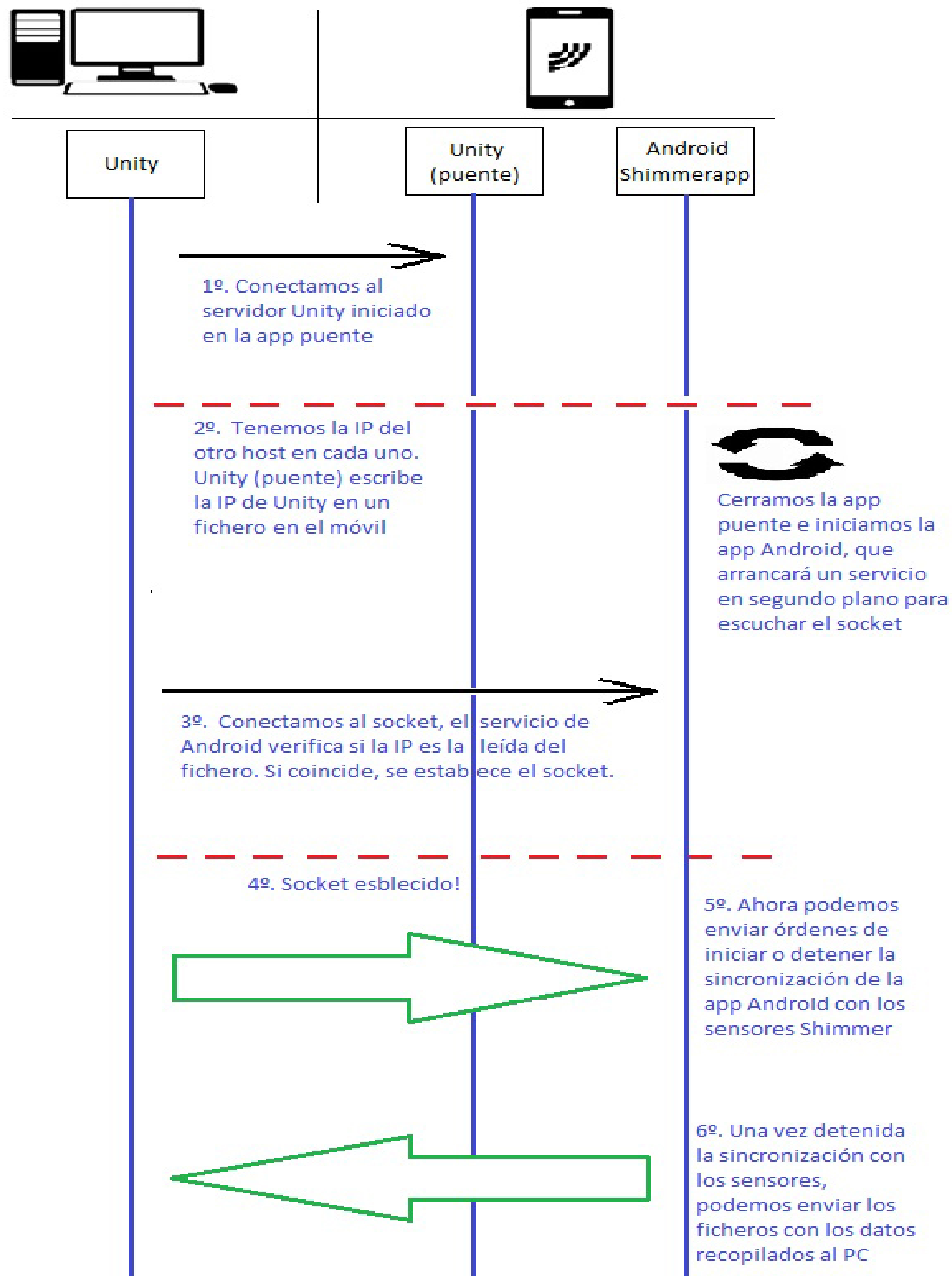


Figura 81. Esquema de comunicación final Unity-Android

7.3.1 Primer Paso: Comunicación Unity-Unity definitiva

El proceso de conexión con la aplicación puente de Unity en el móvil es el mismo que ya definimos e implementamos anteriormente, pero ahora, no vamos a intentar llamar a ningún método remoto (RPC), sino que vamos simplemente a almacenar la IP del cliente (ordenador, con su aplicación Unity) en un fichero en el móvil, que luego leeremos desde la aplicación Shimmer para Android para conectar el socket. El directorio donde lo almacenaremos es el mismo que usará la aplicación para almacenar los ficheros de datos extraídos. Obtendremos este directorio empleando la integración (pasos explicados anteriormente), para ejecutar el método de la aplicación Android en esta aplicación puente de Unity, y devolver una cadena de caracteres o *String* con la ruta del directorio, donde guardar un fichero de texto con la IP del ordenador conectado a nuestra aplicación puente.

-Hay que observar que la integración finalmente sí que funciona en este otro sentido, ejecutar parte de código Android y devolver un resultado a Unity. Sin pretender, como intentábamos, llamar a un método que se ejecute en la aplicación Android y haga cambios en esta. Puesto que esta parte de código Android realmente se ejecuta en la aplicación Unity, y como en un principio el método Android que pretendíamos ejecutar simplemente imprimía un mensaje que observábamos en nuestro IDE Eclipse, era en realidad la aplicación Unity, integrada ya con Eclipse también, la que estaba mostrando el mensaje en el registro de Eclipse, dándonos así falsas esperanzas de poder hacer cosas más complejas en este método llamado desde la aplicación puente Unity, como iniciar los sensores.

Esta es, por tanto, la utilidad y enfoque correcto de toda la integración de Unity con Android explicada anteriormente en los pasos detallados: ejecutar parte de un código de aplicación Android en nuestra aplicación Unity, devolviéndonos los resultados necesarios. De esta forma, la aplicación Android puede actuar como un 'plugin' de Unity-

El código correspondiente a la conexión Unity mediante la API *Standard Unity Networking* para establecer la conexión entre la app del móvil y el ordenador es el que ya hemos explicado anteriormente, la diferencia es que ahora, lo que hacemos es almacenar las IP en una variable:

```
void OnPlayerConnected(NetworkPlayer player){     //Metodo que se
 llama cuando se ha conectado un cliente (player) al server
        ipdelclient = player.ipAddress;
        portdelclient = player.port;
}
```

Y en la parte de la aplicación puente de Unity en el móvil, como ya hemos explicado, guardamos esta IP en un fichero, que después leeremos desde la aplicación Android. Para obtener el directorio en el que crear este fichero, usamos una llamada a un método de Android, mediante la integración implementada. Este método estático se sitúa en nuestro proyecto Android en el fichero correspondiente a la nueva actividad que hereda únicamente de *UnityPlayerActivity*, y que explicaremos más adelante. El método que llamamos desde Unity es el siguiente:

```
public static String filePathReturn(){     //Metodo          para
retornar la ruta en la que escribiremos desde Unity la IP del
cliente para luego leerla dsede aquí:
     String                        path                        =
     Environment.getExternalStorageDirectory().getAbsolutePath()
     + "/AK.Shimmer";
          return path;
}
```

La utilidad de este método proporcionado por la API de Android es devolver una ruta en la SD, si hay alguna instalada en el móvil, o en caso contrario devolver una ruta en la memoria interna del móvil.

Por tanto, el fichero con la IP que leemos desde la aplicación Android (*/clientipsaving/ipsaved.txt*) se almacenará en la misma ruta que el resto de ficheros de datos que recogemos, lo cual debe tenerse en cuenta en la actividad que gestiona la pantalla de ficheros (*ManageFiles*) en la aplicación:

```
     if(files_n1[i].isDirectory()){
```

```
path = files_n1[i].getAbsolutePath();

if(path.equals(Environment.getExternalStorageDirector
y().getAbsolutePath() +
"/AK.Shimmer"+"/clientipsaving")==false){
//Evitamos que coja directorio con fichero con IP que
escribimos desde Unity para establecer luego aqui el
socket

      files_n2 = new File(path).listFiles();

      if(files_n2.length>0){
      Files.add(new Files("", files_n1[i].getName(),
      "", ""));
            addFilesToArray(files_n2); // Se añaden los

      ficheros de la carpeta al array
```

Además de usar este método en la aplicación puente de Unity para que el directorio sea adecuado en función de la existencia de tarjeta SD o no, utilizando la integración para esto, nos aseguramos de que luego no haya problemas con los permisos de lectura del fichero desde la aplicación Android, puesto que se encuentran así en el mismo directorio.

Este método de Android, lo llamamos desde Unity, de manera similar a como hacíamos para llamar al método de prueba cuando desarrollábamos anteriormente la integración Unity-Android, mediante el siguiente código en *C#*:

```
//Probamos a guardar fichero llamando al plugin de android para
evitar problemas con permisos:
#if UNITY_ANDROID
var androidJC = new AndroidJavaClass("com.unity3d.player
.UnityPlayer");
var jo = androidJC.GetStatic<AndroidJavaObject>("currentActivity
");
var main = new AndroidJavaClass("com.ak.shimmer.ManageDevicesUni
ty");
myFinallypath = main.CallStatic<string>("filePathReturn");
#endif

//Vamos a crear fichero para guardar la IP del cliente y poder l
eerlo luego desde Android para establecer el socket
myFinallypath=myFinallypath+"/clientipsaving/";

if (!Directory.Exists (myFinallypath)) {
       Directory.CreateDirectory (myFinallypath);
```

```
}

//Guardamos ficheros con la IP del cliente
myFilepath = myFinallypath + "ipsaved.txt";
System.IO.File.WriteAllText(myFilepath, ipdelclient);
```

Mostramos también una captura de la aplicación puente definitiva tras establecer la conexión, que muestra ahora en la interfaz la IP y puerto del cliente Unity (PC) conectado en la figura 82.

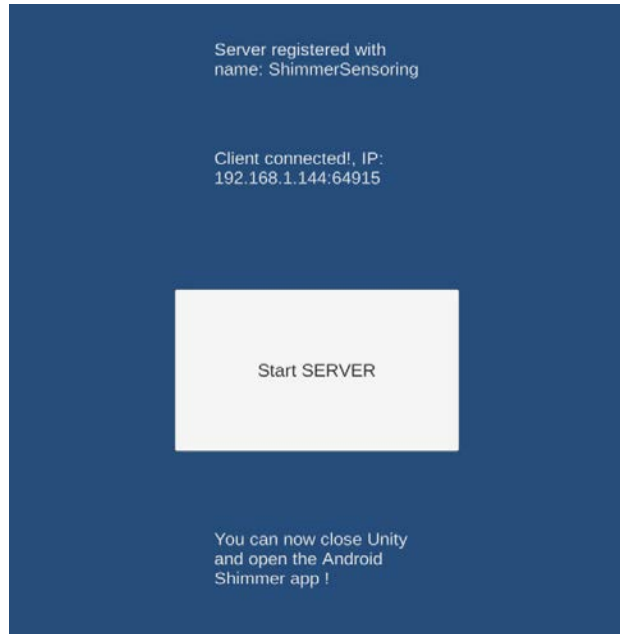


Figura 82. Aplicación puente definitiva

Y la aplicación Unity cliente que empleamos en nuestro ordenador:

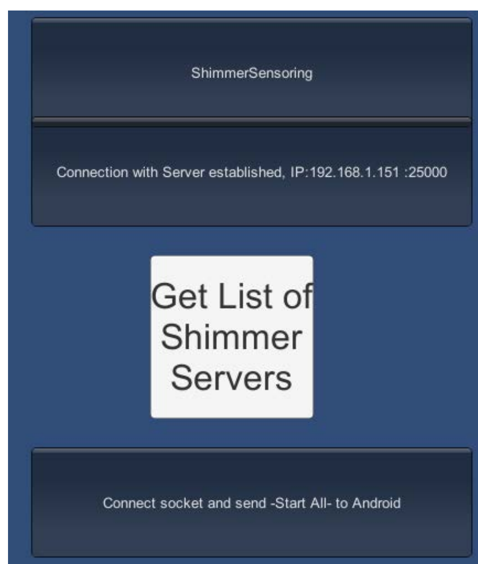


Figura 83. App Unity de prueba final

Vemos que seguimos registrando el servidor en Unity con el mismo nombre: "*ShimmerSensoring*" ya que, como ya hemos dicho, el proceso de conexión es el mismo que el explicado anteriormente.

7.3.2 Segundo Paso: Comunicación Unity-Android definitiva, con Socket

Una vez que hemos llevado a cabo esta conexión Unity entre nuestro móvil (servidor) y nuestro ordenador (cliente), podemos ver que ya disponemos en cada máquina de la IP del otro host respectivamente, por lo que ya podemos establecer nuestro socket para comunicar nuestro aplicación Unity del ordenador con la aplicación Android de los sensores.

También conocemos el puerto, pero no nos interesa demasiado, puesto que estableceremos el socket siempre en el puerto 8080. Este resulta adecuado, porque la API de Android para sockets, por cuestiones de seguridad, no permite establecer sockets en puertos menores al 1024 (los que pertenecen a protocolos más comunes), y el puerto 8080 es un puerto alternativo al 80 de *HTTP*, que típicamente no reportará ningún problema en cuanto a poder conectarnos sin abrir puertos o ser bloqueados por firewall.

Para implementar nuestro socket, en la parte Unity, con *C#*, usamos la API `System.Net.Sockets`,. En la parte Android, con *Java*, empleamos la API `java.net.Socket`.

Por tanto, una vez realizada la conexión, podemos cerrar ya o minimizar la aplicación Unity puente del móvil, y abrimos la aplicación Android para Shimmer. Esta aplicación será la parte servidor del socket, iniciará un servicio que escuchará en el puerto 8080 indicado anteriormente, esperando a que iniciemos la conexión desde nuestro ordenador.

Un servicio, en Android, es un componente de una aplicación el cual se

ejecuta en segundo plano (*background*), y puede realizar las mismas acciones que un *Activity* o actividad, con la única diferencia de que este no proporciona ningún tipo de interfaz de usuario.

Los servicios son útiles para acciones que se requieran realizar durante un tiempo en *background*, sin tener en cuenta lo que está en pantalla, tales como: ejecutar transacciones de red, reproducir música, ejecutar archivos, ejecutar procesos, o en nuestro caso, escuchar para establecer una conexión con un socket.

El resumen, por tanto, el proceso explicado hasta ahora, y lo que vamos a implementar, para esta parte final de la comunicación, se puede ver de manera esquemática en la figura 84.

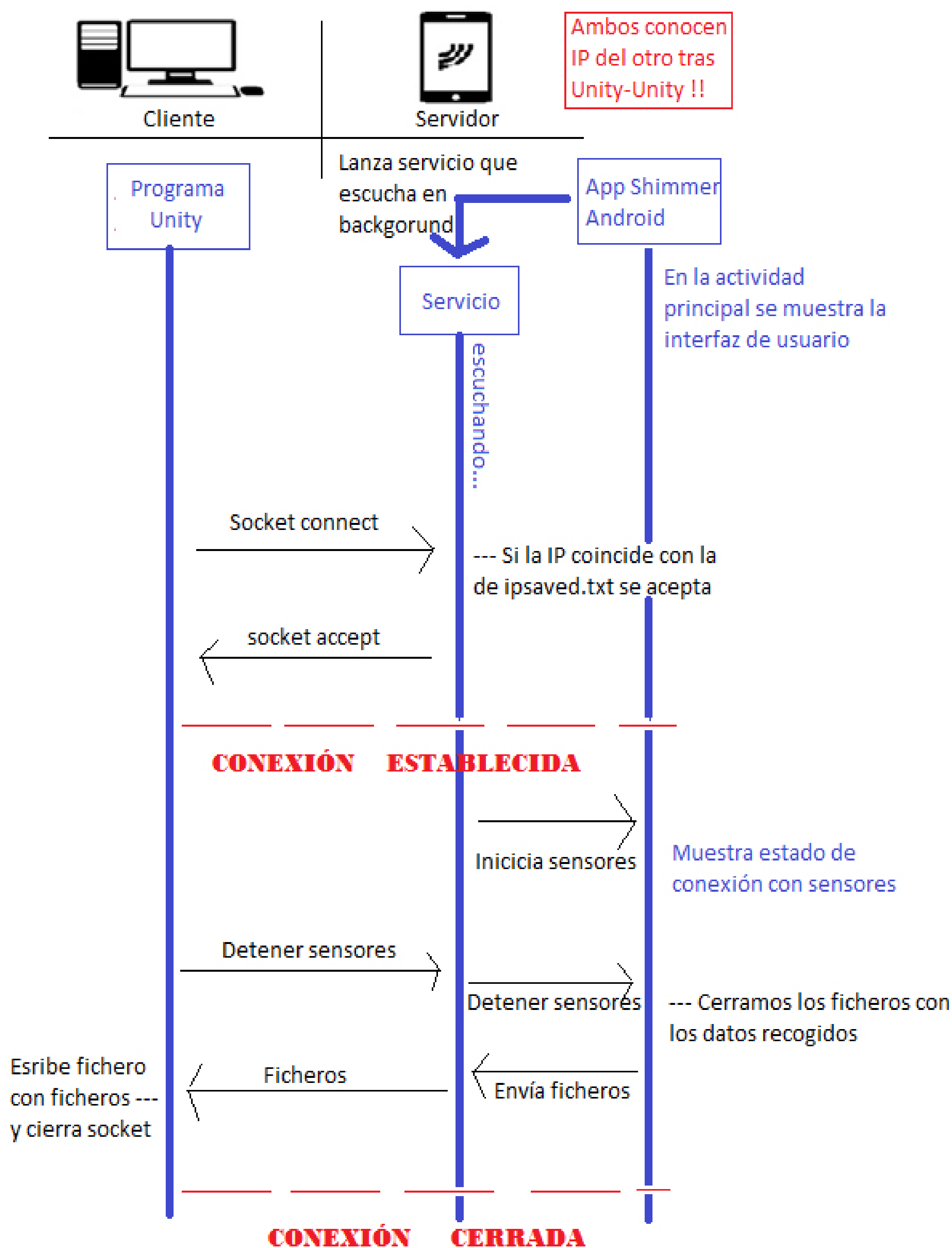


Figura 84. Esquema segundo paso comunicación Unity-Android definitiva

Como vemos en la figura 84, tenemos que gestionar además la comunicación entre el servicio y la actividad principal. Empezamos detallando desde el principio lo añadido a la aplicación Android para dotarla de la comunicación con sockets:

Lo primero que hicimos, como ya hemos dicho, es añadir un "*Toggle Button*" (ver figura 59) para que usuario seleccione si va a usar la comunicación con Unity o no. En caso de que este esté en "*Enabled*", al presionar *Manage Devices* la aplicación abrirá una nueva actividad (*ManageDevicesUnity.java*), distinta a la que explicábamos en el punto de Android. Con una interfaz muy simplificada para, como dijimos anteriormente, resolver el problema de la herencia múltiple no teniendo que heredar más que de *UnityPlayerActivity.*

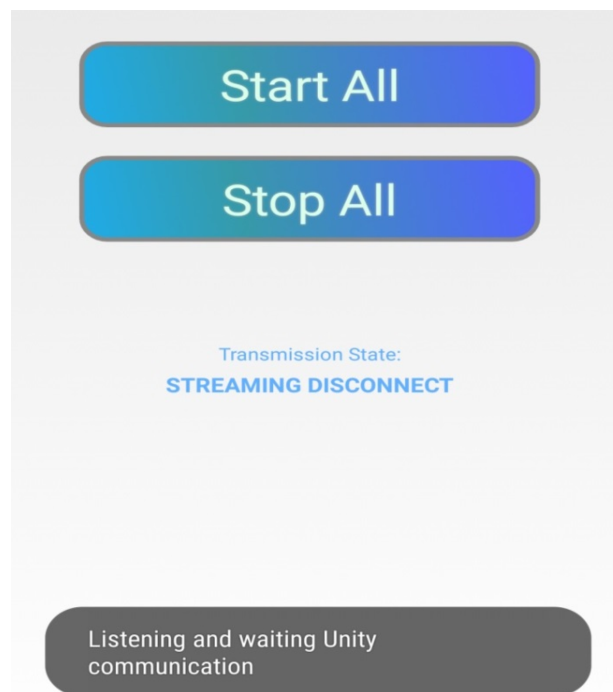


Figura 85. Interfaz actividad de comunicación en app Android

Esta nueva actividad, lanza, nada más iniciarse (en su método *OnCreate()*), el servicio que escuchará en segundo plano para establecer el socket y nos informa de ello con un breve mensaje “*Toast*”:

```
//Lanzamos el servicio que escuchara el socket con Unity:

Intent   intent   =   new   Intent(ManageDevicesUnity.this,
UnityService.class);
startService(intent);
```

Este servicio lo implementamos en nuestro proyecto de Android en *UnityService.java*. En el momento que se lanza el servicio, crearemos un hilo o *thread* nuevo, que será el que en realidad escuchará en el puerto determinado esperando que se solicite establecer el socket. Hacemos esto así para evitar bloqueos críticos en el hilo principal del servicio y la aplicación:

```
@Override
public void onCreate() {
      this.thread = new Thread(new CommsThread());
      //CommsThread es una clase definida aqui a
      continuacion, escuchamos socket en un thread distinto
      al que usa la app para no causar bloqueo con
      accept()
      thread.start();
}
```

En este hilo (*CommsThread*), que se ejecutará paralelo al principal del servicio o *main thread,* es en el que ponemos el socket a escuchar:

```
class CommsThread implements Runnable{

  public void run(){

      //Inicializamos el socket (Android sera la parte
      servidor y unity la parte cliente en cuanto a
      sockets)
      try {
      unityip=InetAddress.getByName(ipfromfile);
      } catch (UnknownHostException e) {
      e.printStackTrace();
      }

try {
           serverSocket=new ServerSocket(localport);
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }

    while(!Thread.currentThread().isInterrupted()){

            try {
                if (socket==null){
                socket=serverSocket.accept();    //accept
                bloquea hasta que conecta socket
                    //Inicializamos flujo para leer desde
                socket:
                in          =new        BufferedReader(new
                InputStreamReader(socket.getInputStream())
                );
                    conectado=true;
                }

            } catch (IOException e) {
                    e.printStackTrace();
            }
```

Observamos que empleamos el método *accept()* para escuchar, que se quedará bloqueado en este nuevo hilo, hasta que se envíe la petición de conexión desde Unity en nuestro ordenador.

Una vez que tenemos al servicio escuchando, podemos presionar en el programa Unity de nuestro ordenador (cliente) el botón de conectar, que ejecutará el siguiente código para conectar con nuestra aplicación Android:

```
//Metodo para establecer la conexion socket cliente con el socke
t de Android
public void setupSocket(){
  try {
     portdelserver=8080;    //Este es el puerto en el que
hemos puesto a escuchar el socket en la app android
     mySocket = new TcpClient(ipdelserver, portdelserver);
     theStream = mySocket.GetStream();
     theWriter = new StreamWriter(theStream);
     theReader = new StreamReader(theStream);
     socketReady = true;
  }
```

```
  catch (Exception e) {
        Debug.Log("Socket error: " + e);
  }
}
```

Vemos que aparte de conectar el socket, tenemos que abrir dos flujos distintos (*StreamWriter* y *StreamReader*), uno para escribir en el socket y otro para leer desde el socket.

Una vez que la petición de conexión se envía, en el servicio en Android (*UnityService*) verificamos si la IP desde que se realiza la petición es la misma que la leída en el fichero, y si es así se acepta la conexión y se envía a la actividad principal (la que muestra la interfaz de usuario) la orden de conectar con los sensores (*startAll()*):

```
while(!Thread.currentThread().isInterrupted()){
//Comprobamos si nos hemos conectado al socket desde Unity, y en
caso afirmativo, iniciamos el metodo startall en la actividad

      if(socket.getInetAddress().equals(unityip)
      && socket.isConnected()==true && mensajemostrado==false){
      //Comprobamos que la IP conectada es la que tenemos
almacenada de Unity
      System.out.println("Cliente Unity Conectado");
      sendMessagetoactivity(100);     //100=startall(ver       en
      ManageDevicesUnity.java
           mensajemostrado=true; //Sino con el bucle while lo
      imprime constantemente mientars estamos conectados
      }

      //Metodo para enviar mensaje a la actividad y recibirlos
con el broadcastreceiver
      private void sendMessagetoactivity(int myInteger) {
      Intent intent = new Intent("my-integer");
      // add data
           intent.putExtra("message",
           String.valueOf(myInteger));

      LocalBroadcastManager.getInstance(this).sendBroadcast(inten
      t);
      }
```

Podemos ver en este fragmento de código Android, en el que enviamos un mensaje a un "*BroadcastReceiver*" que se situará en la actividad principal, parte de cómo hemos implementado la comunicación entre el servicio y la actividad principal. Resumimos esta en la figura 86.

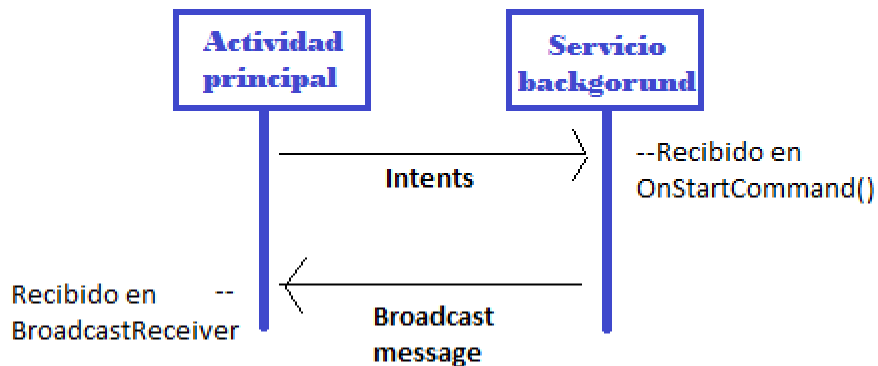


Figura 86. Resumen de comunicación en Android entre servicio y actividad principal

Luego, en este caso, enviamos el mensaje al *Receiver* de la actividad principal, con un entero que se corresponderá con una acción a realizar (en este caso conectar e iniciar todos los sensores que hayamos seleccionado en la pantalla previa de la aplicación).

Los *BroadcastReceivers* son componentes que permiten registrar eventos que se produzcan en el SO Android como, por ejemplo, cuando se recibe una llamada, se está agotando la batería, se enciende el *smartphone* o, en nuestro caso, se envíe un mensaje de difusión (*broadcast*) desde el servicio.

Nosotros lo implementamos así en la actividad principal:

```
//Receiver para escuchar mensajes enviados por el servicio que
se comunica con Unity mediante socket
private BroadcastReceiver mMessageReceiver = new
BroadcastReceiver() {

    @Override
        public void onReceive(Context context, Intent
        intent){
          // Extract data included in the Intent
          String message = intent.getStringExtra("message");
```

```
            int yourInteger = Integer.parseInt(message);

            //Gestionamos mensaje recibido desde el servicio
            if(yourInteger==100){     //100=startall
                startall=(Button)
                findViewById(R.id.startall_btn);
                 print("Start All called from Unity!");
                 startAll(startall);
            }
            if(yourInteger==101){     //101=stoptall
                stopall=(Button) findViewById(R.id.stopall_btn);
                print("Stop All called from Unity!");
                stopAllunity(stopall);
             }
};
```

Por tanto, mediante el código expuesto, hemos visto que se llamará a la función de iniciar con todos los sensores (*startAll*) nada más que conectemos el socket en el servicio. Mostraremos brevemente un mensaje para indicarlo y empezamos a conectar con los sensores:

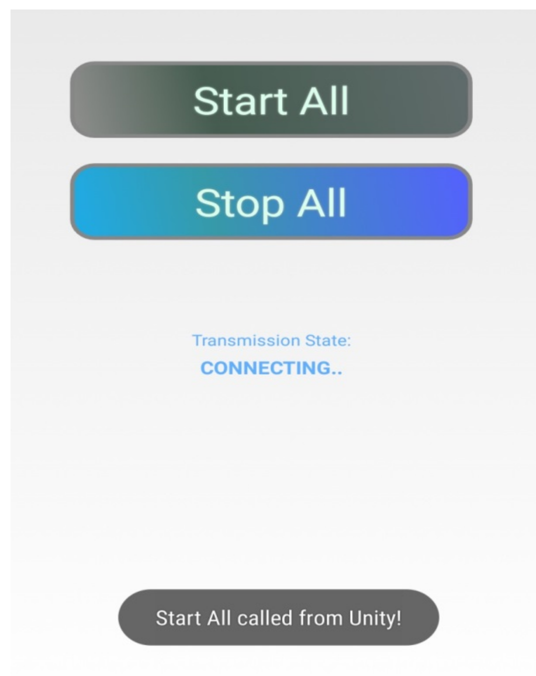


Figura 87. Interfaz Android, start.-all llamado

Así, la actividad principal permanecerá recogiendo datos de los sensores y almacenándolos en los ficheros correspondientes, aunque sin mostrar las gráficas de los datos ni nada parecido debido al problema de herencia múltiple

ampliamente detallado con anterioridad.

A su vez, mientras la actividad principal realiza la tarea con los sensores, el servicio Android en segundo plano seguirá escuchando en el móvil lo que pueda llegar a través de la conexión con el *socket*, y la aplicación Unity para PC también escucha permanentemente lo que pueda llegar por la conexión del socket. Para implementar esta labor de escucha permanente en Unity, como vemos en el siguiente breve fragmento de código, empleamos la función predeterminada por la API de Unity *Update(),* una función que es llamada en cada *frame* siempre que el programa esté en ejecución:

```
 void Update(){
//Lectura de socket y escritura en fichero:
string line = this.readSocket ();

}
```

Ahora, en el momento que se quiera, podemos desde nuestro programa de Unity en el ordenador enviar la orden a la aplicación de Android de detener la recopilación de datos de los sensores, presionando el botón que ejecutará este código:

```
if (GUI.Button (new Rect (200, 500, 400, 100), "send -Stop All-
 to Android")) {
   writeSocket ("stopall");    //Mandamos orden de detener
   los sensores
   }
```

Empleamos un flujo de escritura para escribir la cadena “stopall” en el socket:

```
      public void writeSocket(string theLine) {
        if (!socketReady)
            return;
        String foo = theLine + "\r\n";
        theWriter.Write(foo);
        theWriter.Flush();
 }
```

Y en el otro lado, en el servicio de la aplicación Android, estamos escuchando de la siguiente manera:

```
	//Bucle while anidado dentro del bucle general, para escuchar lo enviado desde Unity
	try {
		while ((unityInput = in.readLine()) != null) {

			if(unityInput.equals("stopall")){	//Si enviamos stopall desde Unity enviamos mensaje a la actividad principal para que llame al metodo

				sendMessagetoactivity(101);
				//101=stopall(ver en ManageDevicesUnity.java)
			}
		}
	} catch (IOException e) {
	e.printStackTrace();}
```

Por lo que, al recibir esta cadena, se enviará un mensaje de *broadcast*, mediante la función *sendMessagetoactivity()* cuyo código ya hemos expuesto, que será recogido por la actividad principal Android, gracias al *BroadcastReceiver* implementado y mostrado también.

Así, en la actividad principal se ejecutará la función que detiene la sincronización con los sensores, y cerrará los ficheros con los datos que se han recopilado de estos.

Una vez cerrados los ficheros, es cuando podemos enviar la orden al servicio para que los envíe a través del socket. Recordando que para comunicarnos con el servicio usábamos *Intents*:

```
//Nos comunicamos con el servicio si el método de detener ha sido llamado desde Unity - Tratamos de enviar los ficheros al PC desde el socket->If startService(intent) is called while the service is running, its onStartCommand() is also called.
	Intent intent = new Intent(getApplicationContext(), UnityService.class);
	intent.putExtra("message","sendFiles");
	startService(intent);
```

Esa última línea de código no iniciará un nuevo servicio, puesto que como este ya está corriendo se recogerá en el método prefijado por la API Android *onStartCommand()*.

Así, ya hemos recogido todos los ficheros que tenemos en el directorio de la aplicación (exceptuando el que contiene la IP que escribimos desde la app puente Unity) en un *ArrayList* que llamamos *"Files"*, ahora procedemos a enviar los elementos de este a través de la conexión establecida con el socket, e informar al usuario con mensajes en pantalla:

```
	//Ahora tenemos todos los ficheros en el arraylist Files, VAMOS A ENVIARLOS POR EL SOCKET:

	Toast.makeText(this, "Sending Files to Unity Client...", Toast.LENGTH_SHORT).show();

	for(int i=0;i<Files.size();i++){
		File myfile=Files.get(i);
		byte[] buffer = new byte[(int) myfile.length()];

	if(socket.isConnected()==true){
			try {
				fis = new FileInputStream(myfile);
				infile = new BufferedInputStream(fis);
			} catch (FileNotFoundException e) {
				e.printStackTrace();}
```

(Hemos abierto un buffer de lectura para leer los ficheros (*infile*), necesitamos un buffer o flujo también de escritura en el socket (*out*) para ir escribiendo todas las líneas que vamos leyendo de los ficheros)

```
		try {
			int count;
			out = socket.getOutputStream();
			System.out.println("Sending files");


			//Enviamos el nombre del fichero que toque:
				byte[] namefile={};
				String endline="\n";
				String tempfile=myfile.getName();
```

```
                        tempfile=tempfile+endline;
                        namefile=tempfile.getBytes("UTF-8");
                        out.write(namefile);
                        out.flush();

                        //Por ultimo enviamos el contenido del
fichero:
                        while ((count = infile.read(buffer)) >= 0)
                        {
                              out.write(buffer, 0, count);
                              out.flush();
                        }


                  }
                  else {Toast.makeText(this, "Error sending Files
            to Unity, socket isnt connected!",
            Toast.LENGTH_SHORT).show();}
      }//Fin del for
```

Observamos en el código presentado que primero enviamos el nombre del fichero y luego su contenido. Es importante también limpiar el flujo de escritura o *buffer* del *socket* tras escribir con la función *flush()*, para evitar problemas en el envío de los datos.

Tras esto, ya solo nos queda cerrar todos los flujos y la conexión con el *socket* e informar al usuario de que se ha terminado el envío:

```
      try {
      //Cerramos flujos: (el socket lo cerramos en onDestroy() de
la actividad)
                  out.close();
                  infile.close();
                  Toast.makeText(this, "Files sent to Unity!",
                  Toast.LENGTH_SHORT).show();
      }
```

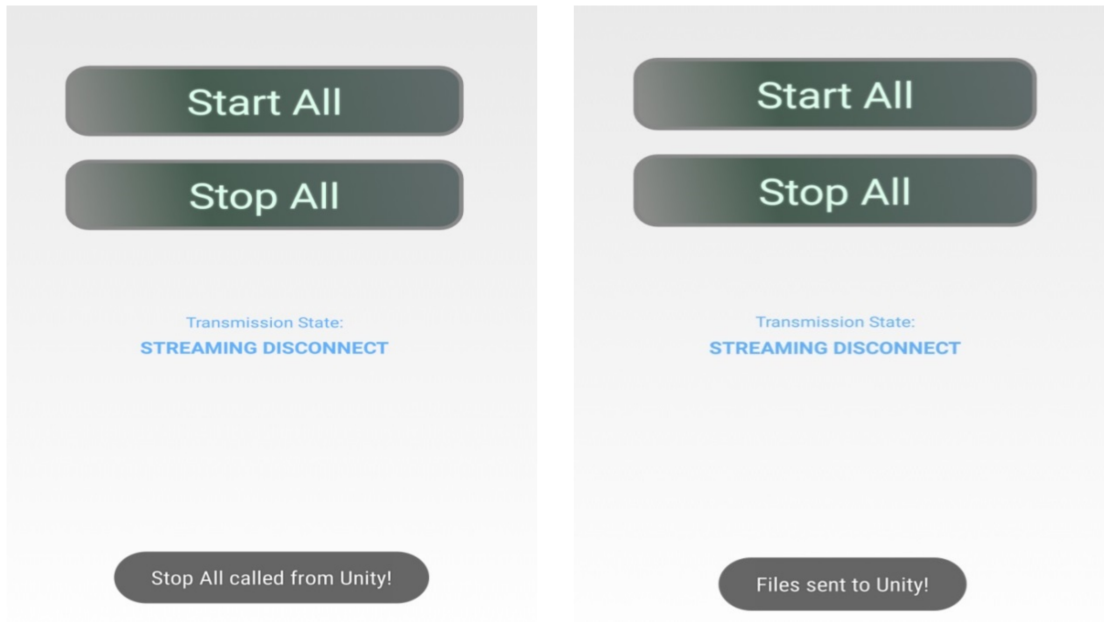


Figura 88. Interfaz actividad Android con envío de ficheros

Del otro lado, en el programa Unity en el ordenador, recogemos, como ya habíamos dicho, todo lo que nos llega por el *socket* en la función *Update*, de manera que si la conexión con el *socket* está establecida, en cualquier momento que nos empiecen a llegar datos, estos se leerán y escribirán en un fichero:

```
void Update(){
//Lectura de socket y escritura en fichero:
string line = this.readSocket ();

//Escribimos estos datos recibidos en un fichero:
if ((line != "") && (socketReady)) {
   receiveddatafromsocket = true;
   Debug.Log ("message:" + line);
   //Write a line of received text in the file:
   sw.WriteLine (line);
   sw.Flush ();
   fcreate.Flush ();
   theStream.Flush ();
}
```

Usamos una variable de control (*receiveddatafromsocket*) para que, una vez que hayamos escrito ya datos en el fichero, y nos dejen de llegar datos, podamos asumir que el envío ha terminado y cerrar los flujos y fichero correspondiente:

```
        if (line=="" && receiveddatafromsocket == true) {
        //Hemos transferido datos,cerrando el socket -
                        sw.Close ();
                        Debug.Log ("Stream closed");
                        closeSocket ();
                        Debug.Log ("Socket closed");
                        receiveddatafromsocket = false;
                        filewritten = true;
        }
    }
```

-Este método se perfeccionará más tarde, en la implementación de todos estos procesos de conexión en el simulador de conducción, con un contador de tiempo, para asegurar que no nos dejen de llegar datos por un corte puntual de conexión, y cerrar los ficheros antes de tiempo, como veremos.-

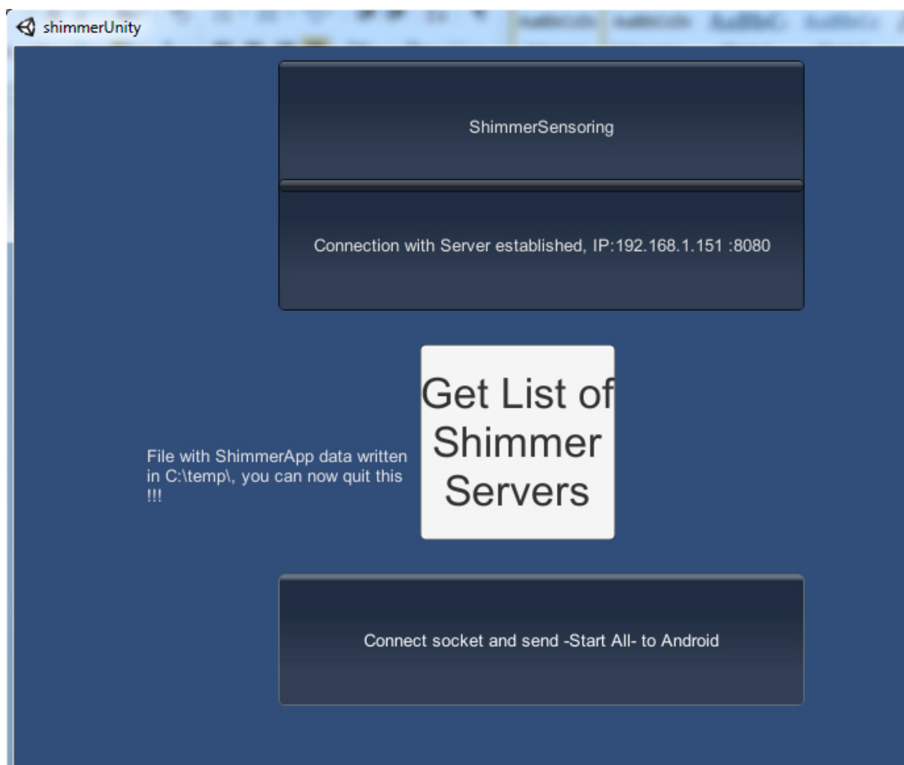


Figura 89. Interfaz app de prueba en PC tras recibir ficheros

Vemos que ya hemos escrito el fichero en el directorio indicado, que recopila todos los ficheros confeccionados por nuestra aplicación Android. Tendrá este aspecto (considerando que hemos conectado y recogido datos del giróscopo):

```
ShimmerAppData.dat
1  RN42-B426_17_11_2016.14-50.dat
2  Accelerometer X Accelerometer Y Accelerometer Z Gyroscope X Gyroscope Y Gyroscope Z Timestamp
3  m/(sec^2)*  m/(sec^2)*  m/(sec^2)*  deg/sec*  deg/sec*  deg/sec*  mSecs
4  1.2673267326732673  9.118811881188119  -2.98019801980198  42.12454212454213  2.197802197802198  2.197802197802198  624.053955078125
5  1.0693069306930694  9.08910891089109  -3.0396039603960396 42.12454212454213  2.197802197802198  1.8315018315018317  721.710205078125
6  1.00990099009901  9.158415841584159  -2.98019801980198  42.12454212454213  1.8315018315018317  2.197802197802198  819.366455078125
7  1.0792079207920793  9.01980198019802  -3.00990099009901  42.85714285714286  2.197802197802198  2.197802197802198  917.022705078125
8  1.108910891089109  9.07920792079208  -3.1683168316831685 42.490842490842496  2.5641025641025643  1.4652014652014653  1014.678955078125
9  1.0495049504950495  9.08910891089109  -3.118811881188119  41.75824175824176  1.4652014652014653  1.4652014652014653  1112.335205078125
10 1.1683168316831682  9.118811881188119  -3.198019801980198  42.12454212454213  2.197802197802198  1.8315018315018317  1209.991455078125
11 1.198019801980198  9.346534653465346  -2.9504950495049505 42.490842490842496  2.197802197802198  1.4652014652014653  1307.647705078125
12 1.0693069306930694  9.00990099009901  -2.98019801980198  42.12454212454213  2.197802197802198  2.197802197802198  1405.303955078125
13 1.0594059405940595  8.910891089108912  -2.8415841584158414 42.490842490842496  2.197802197802198  1.8315018315018317  1502.960205078125
14 0.9504950495049505  9.07920792079208  -3.4158415841584158 42.490842490842496  2.197802197802198  1.8315018315018317  1600.616455078125
15 1.1782178217821782  9.178217821782178  -3.1485148514851486 42.85714285714286  2.5641025641025643  2.197802197802198  1698.272705078125
16 0.9801980198019802  9.06930693069307  -3.0  42.12454212454213  1.8315018315018317  1.8315018315018317  1795.928955078125
17 1.0396039603960396  8.970297029702971  -2.8316831683168315 41.75824175824176  1.4652014652014653  1.4652014652014653  1893.585205078125
18 1.0693069306930694  9.03960396039604  -2.98019801980198  41.75824175824176  1.8315018315018317  1.4652014652014653  1991.241455078125
19 1.2475247524752475  9.07920792079208  -2.9504950495049505 42.85714285714286  2.197802197802198  2.197802197802198  2088.897705078125
20 VueltasVolante_17_11_2016.14-50.dat
21 Steering Wheel Reversals  Mean  SD  Timestamp
22 Mean&SD_17_11_2016.14-50.dat
23 Mean  SD  Timestamp
24 10Mean&SD_17_11_2016.14-50.dat
25 Number of samples (>10 degs/s)  Percent Mean  SD  Timestamp
26 10to7.5Mean&SD_17_11_2016.14-50.dat
27 Number of samples (10-7.5 degs/s)  Percent Mean  SD  Timestamp
28 7.5to5Mean&SD_17_11_2016.14-50.dat
29 Number of samples (7.5-5 degs/s)  Percent Mean  SD  Timestamp
30 5to2.5Mean&SD_17_11_2016.14-50.dat
31 Number of samples (5-2.5 degs/s)  Percent Mean  SD  Timestamp
32 2.5to0Mean&SD_17_11_2016.14-50.dat
33 Number of samples (2.5-0 degs/s)  Percent Mean  SD  Timestamp
34 MaximumValues_17_11_2016.14-50.dat
35 Maximum Values (degs/s) Timestamp
36 0.0 624.053955078125
37 0.0 721.710205078125
38 0.0 819.366455078125
```

Figura 90. Ejemplo de fichero de datos

En la figura 90 podemos ver la ristra de datos de los ficheros, con su encabezado correspondiente. (Recordar que como N – número de muestras que tomamos para calcular medias y desviaciones típicas, está fijado a 20, y en esta captura solo hemos dado tiempo a tomar 19 muestras, los ficheros con los datos de medias y desviaciones están vacíos y solo muestran sus encabezados).

-Es recomendable abrir estos ficheros de datos con un editor de texto más

avanzado que el bloc de notas por defecto de Windows, que a veces los presenta descentrados, como el usado, y gratuito, en este caso: Notepad++.-

Ahora que el proceso de comunicación funciona correctamente en la app Unity de prueba que hemos hecho para nuestro PC, podemos transportar el código e implementarlo sobre cualquier versión del simulador de conducción Unity del que disponemos con la necesidad de añadir además una funcionalidad, el envío de la orden de pausa y reanudación de recogida de datos de los sensores cuando el usuario pulsa el botón de pausa en el simulador.

7.4 Implementación de la conexión sobre simulador

Lo primero que tenemos que hacer es transportar la parte de la conexión de Unity que nos permite conocer en cada máquina o host la IP de la otra.

Esto lo hacemos en un nuevo script de *C#*, "*unityAndroidcommunication*", que añadimos a la versión del proyecto correspondiente del simulador, y anexamos a la scene *MainMenu* del simulador, el GameObject "*Camera*" concretamente.

En la función predefenida de Unity, *Start()*, que se ejecuta nada más iniciar el script, y por tanto, el simulador, incluimos el código para que solicite los datos del server Unity nada más iniciar el simulador, y muestre un botón para conectarse a los servidores encontrados (Por ello debemos haber iniciado la app puente de Unity en el móvil y pulsado para registrar nuestro servidor Shimmer, antes de arrancar el simulador).

```
void Start () {
        RefreshHostList();     //Solicitamos datos del server
        Update();
        networkView=GetComponent<NetworkView>();
    }
```

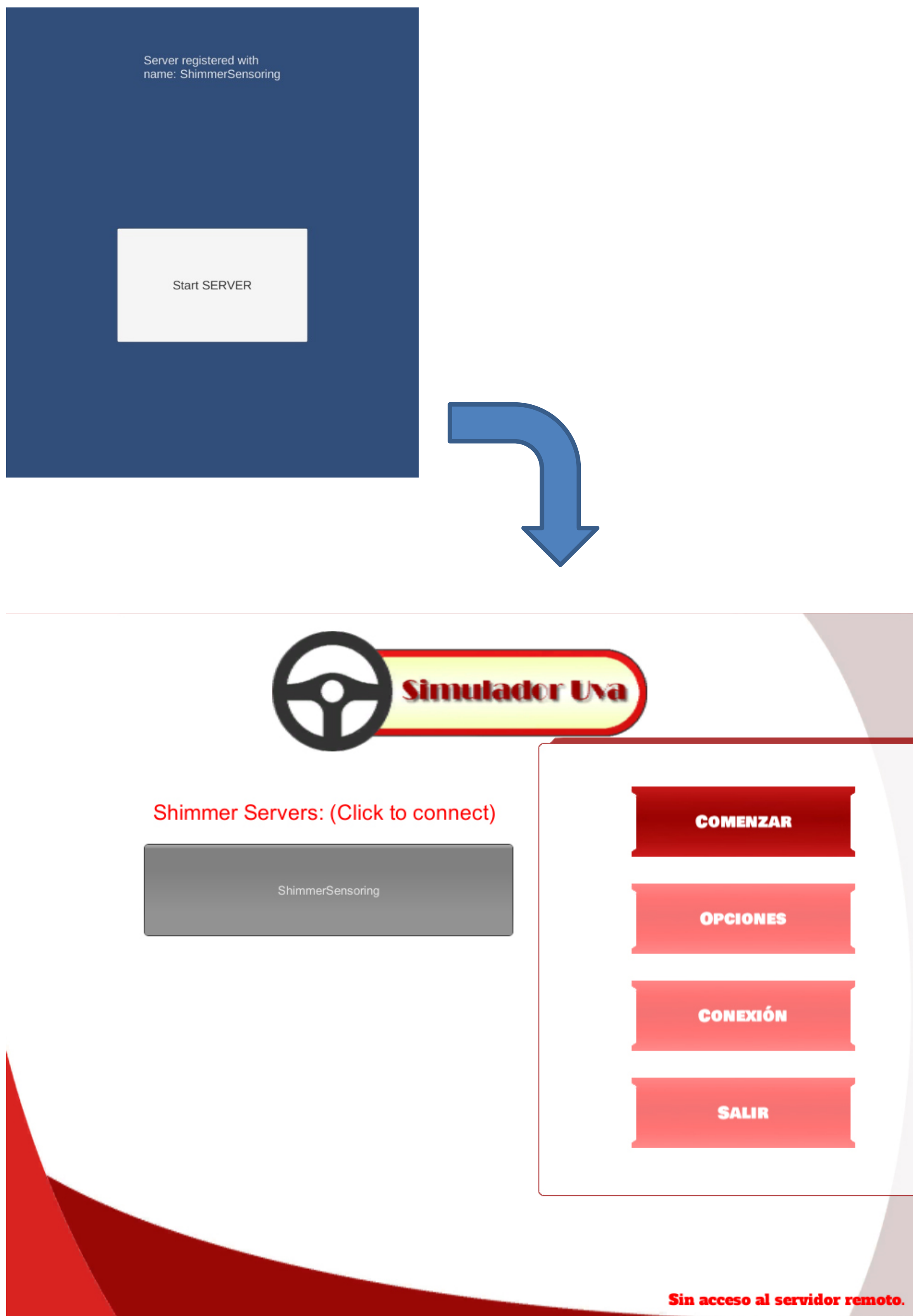


Figura 91. Conexión a Master Server de Unity en simulador

Al presionar el botón para conectar al servidor, y obtener la IP del móvil que ejecutará la aplicación Shimmer, lo mostramos en pantalla:

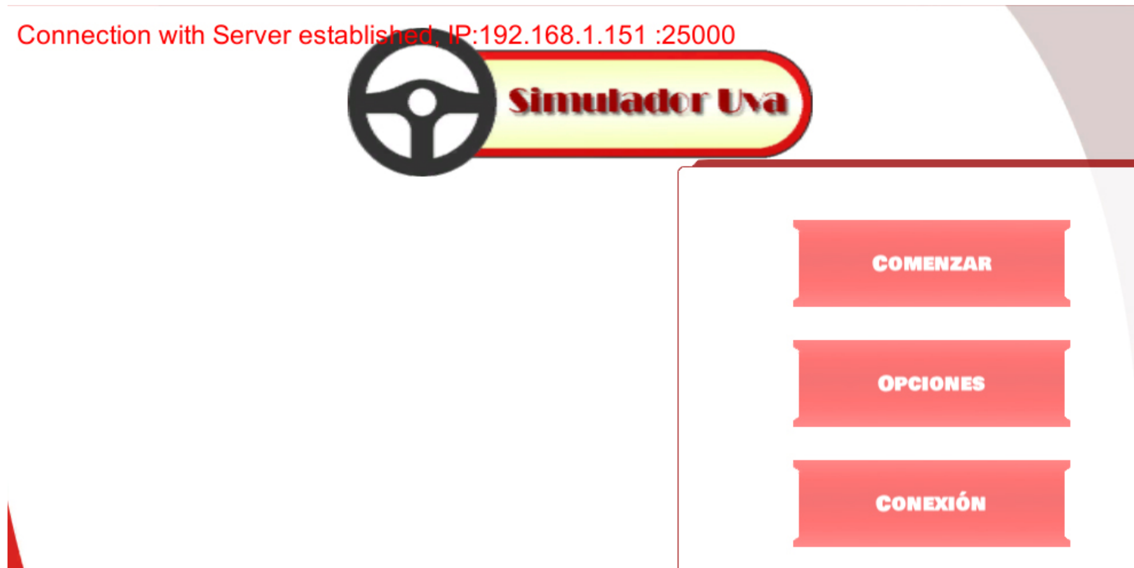


Figura 92. IP registrada en simulador

Y, a diferencia de lo que hacíamos en el programa Unity de prueba, que guardábamos la IP en una variable, ahora la almacenamos en las preferencias del juego, usando la clase de la API de Unity *PlayerPrefs*.

Esta clase permite guardar información recuperable aun cuando cerremos el juego, por lo que, mientras que no cambiemos las máquinas de LAN, no necesitamos volver a hacer este proceso de conexión con la app puente Unity una vez hecho ya, puesto que en el PC (cliente) tenemos guardada la IP del móvil en la clase *PlayerPrefs*, y en el móvil (servidor) tenemos guardado el fichero en el directorio de la aplicación con la IP del ordenador. Con esto tenemos implementado el primer paso de la conexión en el simulador, lo siguiente es conectar el socket.

Conectamos el socket tras haber elegido el escenario para la simulación, al pulsar el botón “comenzar” en la pantalla de opciones previa al inicio de esta, añadiendo el código que ya teníamos desarrollado y explicado para ello al script “*OptionsController.cs*”.

Figura 93. Conectar socket al comenzar simulador

Una vez presionado el botón "Comenzar", y cuando la conexión con el socket haya sido establecida, la simulación no empezará hasta que los sensores estén sincronizados con la aplicación Android en el móvil y esta pueda empezar a tomar los datos de la sesión.

Iniciada ya la simulación, existirá ahora la opción de que el usuario ponga en pausa el juego, presionando "P". Para esta situación, tenemos que implementar una nueva funcionalidad, que detenga los sensores para que no tomen datos mientras el juego se encuentra en pausa, y los vuelva a conectar una vez se quite la pausa (y de nuevo no arranque la simulación y quite la pausa hasta que los sensores hayan iniciado correctamente la sincronización).

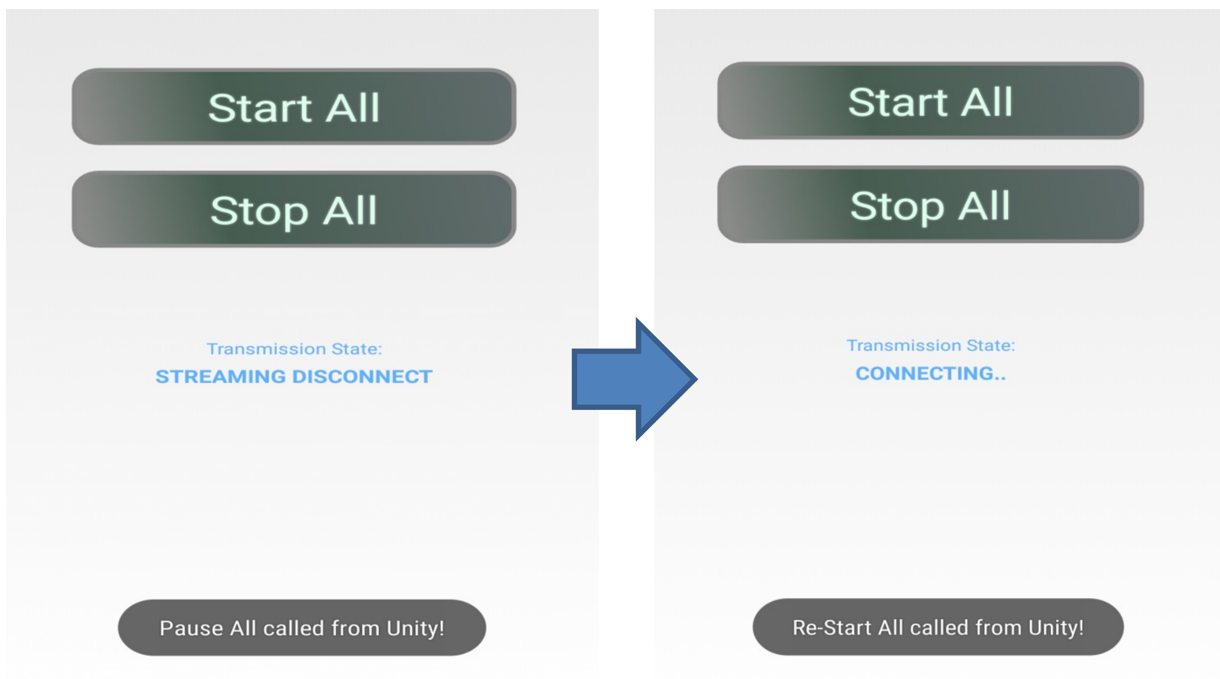


Figura 94. Pausa en simulador y en Android

Para insertar esta función de pausa, añadimos el código al script que controla durante el juego el menú de pausa: "*PauseController.cs*", en la carpeta del proyecto: /scripts/controllers/.

Simplemente lo que hacemos es añadir, a lo que ya teníamos, nuevas cadenas u órdenes a escribir en el socket, que luego deberán ser correctamente recogidas por el servicio de la aplicación Android.

Al presionar el botón de pausa enviamos la cadena "pauseall":

```
public void OpenMenuPause(){

Time.timeScale = 0f;
//<unityAndroidcommunication>

if (firstpause) {
//Recuperamos el socket para poder escribir y leer en el:
	myOptionsController = GameObject.Find ("RCCCanvas").
	GetComponent<OptionsController> ();
	mySocket = myOptionsController.getSocket ();
	theStream = mySocket.GetStream();
	theWriter = new StreamWriter(theStream);
	theReader = new StreamReader(theStream);
	firstpause = false;
}
```

Nos aseguramos también de que no se haya perdido la conexión con el socket:

```
if(!mySocket.Connected){
	ipdelserver=PlayerPrefs.GetString("ipdelserver");
	setupSocket ();
	if (mySocket.Connected) {
		writeSocket ("pauseall");	//Enviamos a
		 Android la orden de detener la transmision
	     de datos con los sensores
		pauseMenu.SetActive (true);
		this.IsGameInPause = true;
	}
}
if (mySocket.Connected) {
	writeSocket ("pauseall");	//Enviamos a Android la
		orden de detener la transmision de datos con los
	sensores
	pauseMenu.SetActive(true);
	this.IsGameInPause = true;
```

Al cerrar la pausa el proceso es el mismo, pero enviando la cadena "restartall".

Luego, estas cadenas deben ser recogidas en el servicio, y en la actividad principal de Android, añadiendo nuevos mecanismos al *BroadcastReceiver* que teníamos implementado:

```
//Receiver para escuchar mensajes enviados por el servicio que se comunica con Unity mediante socket
private BroadcastReceiver mMessageReceiver = new BroadcastReceiver() {
	@Override
	public void onReceive(Context context, Intent intent) {
			// Extract data included in the Intent
			String message = intent.getStringExtra("message");
			int yourInteger = Integer.parseInt(message);

			//Gestionamos mensaje recibido desde el servicio
			if(yourInteger==100){ //100=startall
				createnewfiles=true;
				startall=(Button)
			findViewById(R.id.startall_btn);
				print("Start All called from Unity!");
				startAll(startall);
			}
			if(yourInteger==101){ //101=stoptall
				stopall=(Button)
				findViewById(R.id.stopall_btn);
				sendfilesornot=true;
				print("Stop All called from Unity!");
				stopAllunity(stopall);
				//Metodo distinto cuando llamamos a stop desde Unity, porque tenemos que enviar ficheros al PC por el socket
			}
			if(yourInteger==102){ //102=pauseall
				stopall=(Button)
				findViewById(R.id.stopall_btn);
				sendfilesornot=false; //Queremos detener la transmision pero NO enviar los ficheros
			aun (menu pausa de Unity)
				print("Pause All called from Unity!");
				stopAllunity(stopall);
			}
			if(yourInteger==103){ //103=restartall
				createnewfiles=false;
				print("Re-Start All called from Unity!");
				startAll(startall);
			}
```

```
        }
};
```

Esta nueva función implica establecer dos nuevas variables *booleanas* de control en la aplicación, que podemos ver en el código anterior: "*createnewfiles*" y "*sendfilesornot*".

La primera es para, si hemos dado la orden de detener los ficheros desde la pausa, que luego en la función que volverá a conectar todos una vez haya quitado el usuario la pausa (*SyncDevices()* de nuestra actividad *ManageDevicesUnity.java*), no se creen nuevos objetos de la clase fichero correspondiente, y por tanto no se creen nuevos ficheros, y los datos se sigan así escribiendo en los mismos archivos que se estuviesen usando antes de que el usuario pusiera la pausa. Esto implica también tener cuidado con no sobrescribir los ficheros, fijando el segundo argumento a "*true*" al crear el flujo de escritura en el fichero:

```
writer = new BufferedWriter(new FileWriter(outputFile,true));
```

La segunda variable de control es para que no se envíen los ficheros por el *socket* en caso de que la orden de detener los sensores haya sido dada por el menú de pausa.

Por último, en cualquier momento de la simulación podemos detenerla, de tres maneras distintas:

- Pulsando pausa y luego: "Volver al menú principal"
- Pulsando pausa y luego: "Salir"
- Presionando la tecla "F" para detener la simulación

Para recibir los ficheros y escribir los datos en uno en nuestro ordenador, transportamos el código que ya teníamos desarrollado al script de *C#* "*PauseController.cs*" que, como hemos dicho, existía ya en el proyecto para

implementar las funciones del menú.

Así, antes de salir o volver al menú principal, cuando el usuario presiona la opción, recibimos e escribimos los datos de la aplicación. Como el proceso se puede prolongar en el tiempo si la cantidad de datos es amplia, hemos añadido una barra de progreso, para no dar al usuario la sensación de bloqueo durante el tiempo que tardamos en escribir el fichero de datos. Podemos observarlo en la captura siguiente.

Figura 95. Simulador recibiendo datos con barra de progreso

En la otra opción para salir del juego de la que dispone el usuario, el final de la simulación presionando la tecla “F”, tenemos que implementar el código dentro del script que ya existía en el proyecto de nuestro simulador: “*FinishSimulationController.cs*”.

Con el añadido de que aquí, tenemos que deshabilitar el script de pausa (*PauseController.cs*) al leer los datos del socket, puesto que si el usuario ha usado la pausa y luego reanudado el juego, el script de pausa está activado, y la escritura de datos en el fichero es incorrecta:

```
public PauseController mypausecontroller;
mypausecontroller= GetComponent<PauseController>();
mypausecontroller.enabled = false;
```

Lo que deshabilitamos así en realidad son las llamadas a la función *Update()* en cada *frame* por parte de este *script*. Justo lo que queremos. Volvemos a mostrar una barra de progreso en la pantalla de cierre de la simulación y escribimos el fichero con los datos, finalizando el proceso de comunicación, y obteniendo así en un fichero todos los datos recogidos durante la simulación para los análisis pertinentes.

8. PRESUPUESTO DEL PROYECTO

En este apartado vamos a hacer una aproximación económica al coste supuesto para este proyecto. Habrá que tener en cuenta no solo la inversión que hubiera sido necesaria en los medios hardware y software empleados, sino el elevado número de horas invertidas en la realización de un proyecto extenso como este.

En cuanto a los costes hardware, los sensores fisiológicos *Shimmer v2* empleados tienen un coste de 3.000 €, a lo que podemos sumar 500 € aproximados del resto de hardware empleado: coste de PCs, portátil, dispositivos Android, proyector para pruebas, etc.

No tenemos coste de software, luego tenemos un coste total en medios de 3.500 €.

Para la realización del proyecto estimamos las horas invertidas en 480. Consideramos un período aproximado de 12 meses, trabajando 4 horas diarias, 5 días a la semana. Consideramos un salario de ingeniero junior que podríamos estimar en 8 € la hora con los salarios actuales, un gasto total de 3.840 €.

Si consideramos al trabajador como autónomo, teniendo en cuenta que la cuota de la seguridad social en régimen de autónomos es de 264,44 € al mes, a lo largo de esos 12 meses ascendería el total a 3.173,28.

Por tanto, tendríamos un coste total de mano de obra de 3.840+3.173,28=7.013,28 €.

En resumen, el presupuesto final estimado para el proyecto sería el siguiente:

Coste total en medios Hardware y Software = 3.500 €

Coste total en mano de obra = 7.013,28 €

Coste total estimado para el proyecto = 10.513, 28 €

9. CONCLUSIONES Y LÍNEAS FUTURAS

El avance de la tecnología en materia de miniaturización de la electrónica ha potenciado el uso de los componentes conocidos como sensores. Tanto es así, que numerosas previsiones apuntan a estos componentes como parte fundamental en los futuros paradigmas que se vislumbran en las tecnologías de la información, por ejemplo, en el Internet de las cosas (*Internet of things*, abreviado *IoT*), un concepto que se refiere a la interconexión digital de objetos, a través de Internet, para crear entornos inteligentes en todos los ámbitos para el ser humano.

Otra de las aplicaciones remarcables actualmente en cuanto al uso de sensores es, como hemos visto a lo largo del presente trabajo, la monitorización fisiológica de las personas. En este proyecto, hemos profundizado, en concreto, en la monitorización física del conductor, con los sensores fisiológicos Shimmer, y de su estado de baja o alta alerta, lo cual es un objeto de investigación muy presente actualmente en la industria automovilística, en materia de prevención de accidentes.

Como enfoque y resumen general, en este proyecto teníamos como objetivo principal, de entre todos los sensores Shimmer disponibles ya detallados, la extracción y procesado de datos del giróscopo. Para ello nos hemos apoyado en diversas tecnologías: Partiendo de una aplicación base, hemos hecho implementaciones en *LABVIEW* y en Android, para dispositivos móviles, sobre los datos del giróscopo. Hemos hecho desarrollos en Unity, un motor gráfico para videojuegos, para las simulaciones experimentales.

Finalmente, también hemos empleado MatLab y Excel como software para el tratamiento y análisis de los datos extraídos de las pruebas con el simulador.

En primer lugar, fue necesario un acercamiento al funcionamiento de los dispositivos fisiológicos Shimmer, empleando su documentación para la sincronización mediante Bluetooth entre nuestro PC y los sensores y para la calibración de los mismos. Tras un acercamiento general, nos centramos, como ya hemos dicho, en el giróscopo.

Tomando como referencia varios artículos de investigación de carácter técnico-científico, que profundizaban en la monitorización del estado del conductor a partir de los giros y velocidad angular del volante, establecimos un conjunto de procesados a realizar con estos datos del giróscopo: medias, pasos por cero, etc. Teniendo, por tanto, que realizar estas implementaciones para las dos plataformas (LABVIEW y Android), lo cual implica tener en consideración las diferencias de programación e interfaz de los datos, de cara al usuario, entre ambas tecnologías.

Para el desarrollo de estas aplicaciones Android y LABVIEW, partimos de aplicaciones base implementadas en otros proyectos, las cuales realizaban la función de la comunicación y recogida de datos básicos de los sensores Shimmer. Por tanto, primeramente tuvimos que estudiar, aparte de conceptos básicos y más avanzados de LABVIEW y Android, la estructura y código de las aplicaciones base, además de lidiar con problemas de importación, especialmente en la aplicación base para Android. Fue también necesario, mediante páginas y manuales oficiales, profundizar en las librerías LABVIEW y Android correspondientes del fabricante Shimmer, empleadas ya en estas aplicaciones base.

Respecto a los artículos de referencia que hemos mencionado, otra cuestión para el desarrollo de estas aplicaciones fue la implementación en sí de algunos de los análisis sobre los giros que proponían, debido a que estos artículos se centraban en mostrar los resultados, pero no el proceso y algoritmo concreto llevado a cabo. Así, tuvimos que ajustar, por ejemplo, el número de

muestras de giro a considerar para hacer cálculos como la media o desviación típica de manera dinámica durante la ejecución de las aplicaciones, y poder mostrar, en tiempo real, los datos y gráficas correspondientes al usuario.

Por lo demás, el trabajo con el giróscopo fue relativamente sencillo, debido a su buen funcionamiento en cuanto a precisión, sincronización y duración de la batería.

Desarrolladas las aplicaciones, la siguiente tecnología a emplear fue el motor gráfico de videojuegos Unity. En este motor tenemos desarrollado el simulador que empleamos después para las pruebas con personas, teníamos varios escenarios urbanos e interurbanos creados en otros proyectos. Nuestro cometido era establecer un medio de comunicación entre la aplicación Android que recoge datos de los sensores Shimmer y nuestro simulador, para así sincronizar la recepción de datos con eventos en el simulador: inicio, pausa, etc. y enviar los datos recogidos desde la aplicación al PC en el que se ejecuta el simulador al finalizar una simulación. Aunque, como hemos visto, conseguimos finalmente implementar la comunicación mediante varios pasos, que implican a la API para el *multiplayer* local de Unity y una comunicación directa mediante sockets entre la aplicación y el simulador, lo cierto es que tuvimos que superar ciertos problemas. Con los primeros enfoques, usando clases predefinidas Unity para la integración mediante herencia en Android (*UnityPlayerActivity*) y una API llamada *RPCs (Remote Procedure Calls)*, no conseguimos establecer la comunicación, a pesar de intentarlo incluso modificando dicha clase predefinida de Unity, aunque luego aprovechamos este enfoque para obtener en la aplicación puente Unity el directorio en el que almacenar la IP del PC del simulador, ejecutando un método Android, para así leer después el mismo fichero con la IP desde la aplicación.

Para el enfoque que finalmente dio resultados, empleando sockets a través de una red local, tuvimos que pedir al administrador de la red de la UVA, modificaciones en el Firewall de salida a Internet de la misma (recordar que almacenamos primeramente la IP del dispositivo Android en un servidor global de Unity), puesto que la red de la UVA bloqueaba por defecto la conexión UDP

con el servidor de Unity.

Establecida la comunicación y realizadas pruebas de simulación con distintas personas, mediante los sensores Shimmer, nuestro simulador Unity y el volante *Logitech*, tuvimos que desarrollar scripts de MatLab y emplear Microsoft Excel para analizar la gran cantidad de datos obtenidos.

Aunque muchas de las correlaciones buscadas en los datos no han pasado nuestros tests estadísticos de verificación, hemos podido establecer ciertas conclusiones:

- ✓ Hemos visto gráfica y visualmente como, y de acuerdo con estudios aportados por la DGT de los que partimos, una velocidad media mayor ha implicado en nuestro simulador una media mayor de infracciones para los usuarios, considerando vías urbanas e interurbanas y ambos tipos de cambio (manual y automático) conjuntamente. Además hemos verificado la relación, especialmente en vías urbanas, entre el tipo de cambio y la media de velocidad, teniendo una media más alta para un tipo de cambio automático. Lo cual pueda, quizá, asociarse con el factor de manejar el simulador y cambiar de marcha en el mismo.

- ✓ Correlacionando con nuestros datos del giróscopo, empezando por la media de giros (asociada a una conducción más o menos brusca) y la desviación típica media (asociada con mayores volantazos puntuales), vimos que primeramente el tipo de vía influye en estos valores, puesto que tenemos medias y desviaciones más altas en vías urbanas que en interurbanas. Posteriormente, empleando nuestros métodos estadístico-matemáticos, hemos encontrado ciertas correlaciones entre tipos concretos de infracciones y nuestros datos del giróscopo, siendo la desviación típica media la variable más a tener en cuenta en vías urbanas (por tener correlación con un número mayor de distintos tipos de infracciones), y siendo la media de giros más significativa en vías interurbanas.

- ✓ También encontramos correlaciones que pasaron nuestros tests, para ambas vías (urbanas e interurbanas), entre los tipos de infracciones anteriores y los datos recogidos por nuestro simulador, como consumo medio de gasolina o revoluciones por minuto medias del motor.

- ✓ En cuanto a la posible somnolencia o baja atención del conductor, y los tests de somnolencia rellenados por todas las personas que participaron en las pruebas, desarrollamos un script de MatLab para intentar captar posibles estados de somnolencia, partiendo, de acuerdo con el artículo de Sajjad Samiee, Shahram Azadi, Reza Kazemi, Ali Nahvi y Arno Eichberger. (2014) [16], de los máximos absolutos en los giros y los pasos por cero. Según este artículo, un conductor alerta corrige continuamente o con mayor asiduidad la posición del volante y genera un número mayor de pasos por cero que un conductor somnoliento. Mientras que es un conductor somnoliento el que genera medias absolutas de giro mayores, puesto que percibe las desviaciones con mayor retardo, lo que le obliga a realizar giros más bruscos.

- ✓ Aunque no encontramos correlaciones que pasaran nuestros tests, entre los datos ofrecidas por el scripts anterior y los datos de somnolencia de los test, sí que encontramos una correlación que pasó nuestras pruebas estadísticas de validez, entre los datos rellenados para uno de los test (SSS) y la media de infracciones de los usuarios.

Durante el desarrollo de esta memoria, en resumen, se han establecido los objetivos principales para este trabajo fin de grado y las fases y métodos seguidos durante su desarrollo. A continuación se ha hablado de los factores

que afectan a la seguridad en la conducción y al estado físico del conductor. Seguidamente se ha abordado la plataforma hardware de los sensores Shimmer y las plataformas software empleadas. Seguidamente, se han tratado las implementaciones realizadas para las tecnologías correspondientes. Finalmente, se han explicado con detalle las pruebas realizadas con el simulador y los análisis hechos sobre los datos: scripts, etc.

En lo personal, ha sido una grata experiencia realizar un proyecto tan diverso como este, tanto en la parte hardware como software, en el que se han usado diversos dispositivos físicos y unas cuantas tecnologías software, plataformas y lenguajes de programación distintos. Además, he podido aprender sobre la distinta implementación de un mismo requisito, en función de la plataforma y lenguaje empleado, e investigar sobre un tema tan actual e interesante como es la monitorización mediante sensores en la conducción.

Las líneas futuras del proyecto son casi inagotables, debido al carácter del tema de investigación y a lo multidisciplinar del mismo. Podrían implementarse más procesados sobre cualquiera de los sensores, tanto en la aplicación LABVIEW como en la aplicación Android. Se podría también tratar de optimizar de alguna manera el cierre de comunicación entre la aplicación y el simulador, para agilizar el envío del fichero de datos de los sensores desde la aplicación al PC. Y por supuesto, cabría la posibilidad de hacer cualquier otro análisis sobre datos recogidos de pruebas con personas reales, además de tratar de hacer pruebas de conducción real con los sensores.

Espero que este trabajo sirva también, para concienciar de la importancia de este campo de la seguridad en la conducción, que afecta a toda la sociedad, siendo una causa importante de mortalidad actual.

Espero que sirva también para conocer mejor todas las plataformas vistas, Android, la más conocida, pero también *LABVIEW* y *MatLab*.

BIBLIOGRAFÍA

[1] Alberto López Germán. Universitat de Vic. (2014). *Manipulador robótico con visión artificial.*

[2] Amine Khadmaoui. Escuela Técnica Superior de Ingenieros de Telecomunicación, Universidad de Valladolid. (2012). *Control inteligente del nivel de vigilancia de un conductor mediante sensores fisiológicos y visión artificial 3D.*

[3] Amine Khadmaoui. Escuela Técnica Superior de Ingenieros de Telecomunicación, Universidad de Valladolid. (2014). *Desarrollo de un aplicación para el control de sensores fisiológicos mediante un dispositivo Android.*

[4] Ana Lago Moreda. Fremap. (2012). *Gestión práctica de riesgos.*

[5] CEA (Comisariado Europeo del Automóvil). (2015). *Sueño y fatiga, ¿cuáles son los hábitos de los conductores españoles?*. Recuperado en septiembre de 2016, de: http://www.fundacioncea.es/np/pdf/estudio-somnolencia-al-volante.pdf

[6] DGT (Dirección General de Tráfico). Servicio de estadística. (2015). *Anuario estadístico de accidentes 2015*. Recuperado en septiembre de 2016, de: http://www.dgt.es/Galerias/seguridad-vial/estadisticas-e indicadores/publicaciones/anuario-estadistico-de-accidentes/anuario-accidentes-2015.pdf

[7] DGT (Dirección General de Tráfico). (2016). *La contribución de la velocidad a la prevención de accidentes en España.* Recuperado en marzo de 2017, de: http://revista.dgt.es/images/La-contribucion-de-la-velocidad-a-la-prevencion-de-accidentes-en-Espana-V3.pdf

[8] Elena Reutskaja, Jaume Ribera. IESE Business School. (2013). *Gestión remota de pacientes. Un estudio sobre las percepciones de pacientes y profesionales en España*. Recuperado en septiembre de 2016, de: http://www.iese.edu/research/pdfs/ESTUDIO-305.pdf

[9] Héctor Ramos Morillo, Francisco Maciá Pérez, Diego Marcos Jorquera. Departamento de Tecnología Informática y Computación, Universidad de Alicante. (2013). *Redes Inalámbricas de Sensores Inteligentes y Aplicación a la Monitorización de Variables Fisiológicas*. Recuperado en septiembre de 2016, de: https://www.dtic.ua.es/grupoM/recursos/articulos/JDARE-06-H.pdf.

[10] Logitech. (2015). *Logitech Gaming Steering Wheel SDK.*

[11] Labview. *Web del fabricante con manual introductorio*. https://www.ni.com/getting-started/labview-basics/esa/environment. Última visita: Septiembre de 2016.

[12] Marta Juste. Expansión-diario económico. (2015). *¿Android o iOS? Los sistemas operativos más usados según el continente.* Recuperado en Noviembre de 2016, de: *http://www.expansion.com/economia-digital/companias/2015/12/09/56684be1ca474151018b4590.html*

[13] NCSDR/NHTSA (National Highway Traffic Safety Administration). (2006). *Expert Panel on Driver Fatigue & Sleepiness; Drowsy Driving and Automobile Crashes.*

[14] Perfecto Sánchez Pérez. DGT (Dirección General de Tráfico). (2014). *Otros factores de riesgo: La fatiga.* Recuperado en septiembre de 2016, de: http://www.dgt.es/PEVI/documentos/catalogo_recursos/didacticos/did_adultas/fatiga.pdf

[15] RACE (Real Automóvil Club de España). (2012). *Campaña sobre fatiga y somnolencia al volante.* Recuperado en septiembre de 2016, de: http://www.race.es/documents/10279/13355/INFORME+RACE+ANFABRA+FATIGA+2012/f5fc1066-5f96-4b87-9bc3-2dbab84e7d6b

[16] Sajjad Samiee, Shahram Azadi, Reza Kazemi, Ali Nahvi y Arno Eichberger. (2014). *Data Fusion to Develop a Driver Drowsiness Detection System with Robustness to Signal Loss*. Recuperado en noviembre de 2016, de: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4208253/

[17] Shimmer. (2016). *Shimmer Dock user guide*. Recuperado en septiembre de 2016, de: http://www.shimmersensing.com/images/uploads/docs/Shimmer_Dock_User_Guide_rev1.8.pdf

[18] Shimmer. (2016). *9DoF Calibration Application user manual*. Recuperado en septiembre de 2016, de: http://www.shimmersensing.com/shop/shimmer-9dof-calibration#download-tab

[19] Shimmer. (2011). *Shimmer LabVIEW Instrument Driver Library User Manual*

[20] Unidad de Epidemiología Clínica y Bioestadística. A Coruña. (2001). *Relación entre variables cuantitativas.* Recuperado en mayo de 2017, de: http://www.fisterra.com/mbe/investiga/var_cuantitativas/var_cuantitativas2.pdf

[21] Unity. *Multiplayer Networking*. https://unity3d.com/es/learn/tutorials/topics/multiplayer-networking. Última visita: Diciembre de 2016.

[22] Unity, foros. *Integrating Unity and Eclipse.* http://forum.unity3d.com/threads/integrating-unity-and-eclipse.71607/ Última visita: Diciembre de 2016.

[23] Unity, manuales. *Remote Procedure Call*. https://docs.unity3d.com/Manual/net-RPCDetails.html Última visita: Diciembre de 2016.

[24] Xuesong Wang, Chuan Xu. (2015). *Driver drowsiness detection based on non-intrusive metrics, considering individual specifics.* Recuperado en septiembre de 2016, de: http://docs.trb.org/prp/15-2196.pdf

ANEXO I

Proceso de emparejamiento

Para poder conectar los sensores con nuestro PC, y obtener los datos correspondientes en nuestra aplicación de LABVIEW, tenemos que llevar a cabo una serie de pasos para establecer la conexión Bluetooth. Para ello como requisito solo necesitamos, lógicamente, que nuestro PC disponga de adaptador Bluetooth. -Los pasos que se detallan a continuación son válidos para Windows.-

Primero tenemos que ir a “Panel de Control->Hardware y Sonido” y hacer clic en “Agregar un dispositivo Bluetooth”. Se nos abrirá una ventana que nos mostrará los dispositivos Bluetooth cercanos que el PC sea capaz de detectar (tenemos que tener el sensor encendido).

Una vez detectado, lo seleccionamos, hacemos *clic* en siguiente, y la clave que nos pide, para los sensores Shimmer es “1234”. Una vez introducida ya tenemos nuestro sensor emparejado con el PC.

Ahora que el sensor está emparejado con nuestro PC, tenemos que saber qué puerto COM está empleando. Para ello nos dirigimos de nuevo a “Panel de Control->Hardware y Sonido” y pinchamos en “Dispositivos e impresoras”. Podemos reconocer nuestro sensor mediante el nombre que coincide con el código “BT Radio ID” imprimido en el dispositivo, lo seleccionamos y entramos en las propiedades de este. Finalmente, en la ventana servicios podemos comprobar el puerto COM asociado para cualquier comunicación del sensor con nuestro PC. Esto se ilustra en la figura I.

Figura I. Emparejamiento

ANEXO II

Importar proyecto en Eclipse

La importación del proyecto base puede suponer, y de hecho nos supuso, ciertos problemas si no se siguen los pasos correctos:

1. Debemos importar los dos proyectos JAVA, ambos son los drivers proporcionados por Shimmer: (ShimmerDriver y ShimmerAndroidInstrumentDriverr), para poder comunicarnos con nuestros sensores Shimmer V2. (File->Import->General->Existing Projects into Workspace)

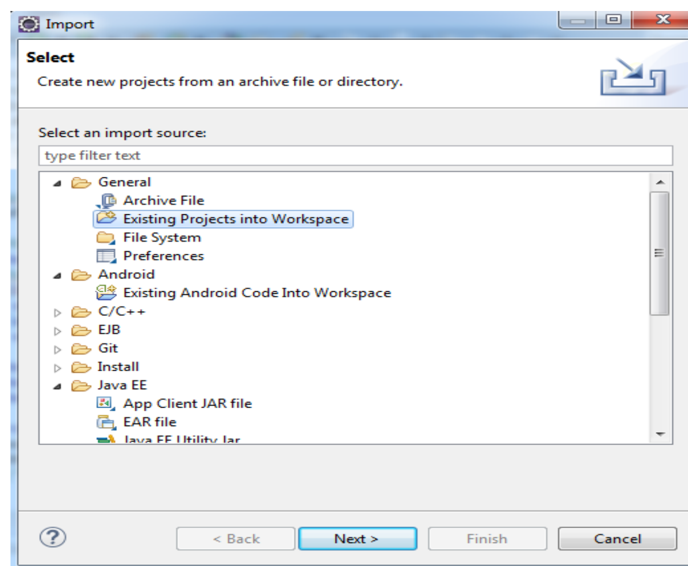


Figura II. Importación en Eclipse

2. Importamos la librería *AppCompat*. (File->Import->Android->Existing Android Code Into Workspace)

3. Importamos finalmente el proyecto de la aplicación (ak.shimmer). (File->Import->Android->Existing Android Code Into Workspace)

Tras la importación, es habitual la aparición de ciertos errores propios de Eclipse, que podemos solucionar continuando con los siguientes pasos:

- Ejecutar ‘project->clean and build’ después de cada paso –

4. Cambiar haciendo clic con botón derecho en el proyecto de AppCompat el *build target* a la API23 (o la que sea la última versión e Android disponible) (botón derecho->properties->Android)

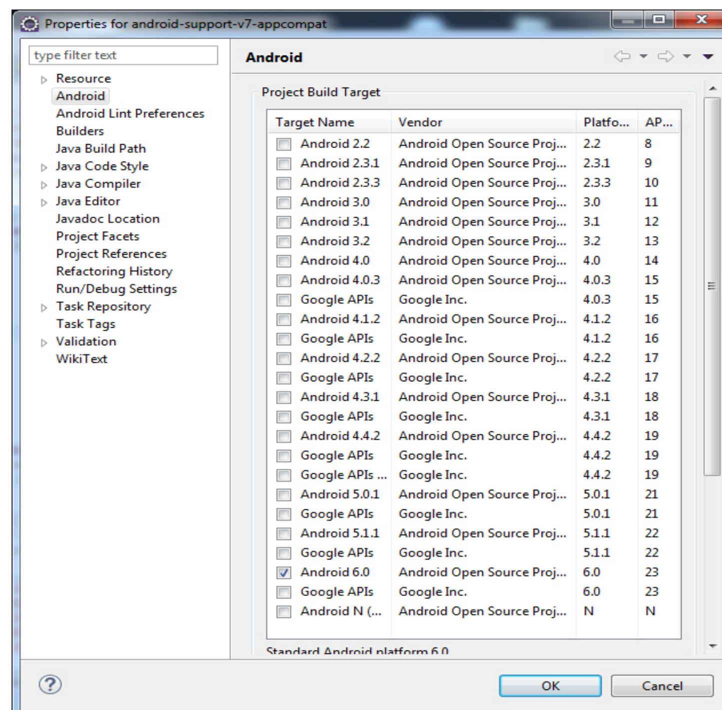


Figura III. Build Target

5. Solucionar posible mensaje de error en *my_custom_theme.xml* del proyecto *ak.shimmer*:

   Ir a carpeta “libs” del proyecto “appcompat” -> botón derecho en los dos archivos “JAR”-> build path-> add to build path. Luego botón derecho en ak.shimmer, properties -> android, bajar hasta abajo,

hacer *remove* de las librerias (tienen una X roja), y luego presionar add-> y añadir las nuevas librerías (*tick* verde) y aceptar.

6. Actualizar el *tarjetSDK* del *androidmanifest.xml* (de todos los proyectos) a la API 23 (o la que sea la última), hacer lo mismo pero con el *project build properties* (botón derecho->properties->android) sobre el proyecto ShimmerAndroidInstrumentDriver (marcando la API 23 o la misma que hayamos escogido antes). Y editar el fichero *project.properties* de todos los proyectos para que el campo *target* apunte a la API23 también (o la que hayamos usado).

   Tras esto, ya deberíamos haber eliminado todos los mensajes de error mostrados por Eclipse y estar en disposición de compilar nuestro proyecto. Podríamos ahora encontrarnos con un error en tiempo de ejecución tras correr la aplicación (*Fatal Main Exception*). Para solucionarlo, solo nos queda un último paso:

7. En todos los proyectos: botón derecho -> Build path -> Configure build path -> order and export -> marcar con *tick* todo lo que sean carpetas de otros proyectos y ficheros .JAR.

Ahora ya, deberíamos poder trabajar cómodamente con nuestra aplicación en Eclipse.