TRABAJO FIN DE GRADO
GRADO EN INGENIERÍA INFORMÁTICA

Inmersión e interacción sobre escenas 3D mediante dispositivos móviles

Autor

Adrián de la Torre Rodríguez

Directores

Alejandro José León Salas

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, 16 junio de 2018

Inmersión e interacción sobre escenas 3D mediante dispositivos móviles

Adrián de la Torre Rodríguez

Palabras clave: Interacción 3D, Manipulación 3D, Dispositivo móvil, Interpretación de información, Sensores de dispositivos móviles.

Resumen

El trabajo realizado en este proyecto consiste en desarrollar un método de interacción 3D que permite manipular un modelo 3D en un mundo virtual con uno y dos dispositivos móviles. El objetivo es la exploración espacial del modelo 3D de una manera sencilla e intuitiva.

La manera de interactuar con el modelo se basa en que al realizar movimientos físicos en los dispositivos móviles, estos serán capturados por los sensores de los dispositivos y traducidos en transformaciones del modelo. Al desarrollo de este sistema se le suma un estudio de los distintos métodos de interacción existentes y una comparación entre los distintos métodos de interacción desarrollados para uno y dos dispositivos móviles.

Immersion and interaction in 3D scenes with mobile devices

Adrián de la Torre Rodríguez

Keywords: 3D interaction, 3D manipulation, Mobile device, Interpretation of information, Mobile device's sensors.

Abstract

This project consists in a 3D interaction method which allows to manipulate a 3D model in a 3D virtual space with one or two mobile devices. It has been developed so it can be extended to the use of a virtual reality headset if needed. The goal of the project is to achieve a simple and intuitive 3D exploration of the model. The way of interacting with the model is by physically moving the devices and by pressing buttons in the interface of the device application.

Different ways of interaction has been developed with one and two devices. For example, with one device you can freely rotate the model by pressing the screen of the phone and moving the hand. Another way of interaction is by choosing the rotation axis and moving the hand to rotate the model in that axis.

Other manipulations has been included to the project such as translation and scaling. Regarding the scaling of the model, with one device you can zoom in and out on the model by sliding the finger on the screen. The model can be moved by rotating the device, if it's rotated upwards, the model will go up, if it's rotated to the left the model will move to the left and so on. The three different manipulations can't be done with a single interface at the same time without complicating the interface itself. The use of a virtual reality headset limits the amount of features we can add to the application interface of the device because we are not able to see the screen of it.

The final and most complete interaction method includes two mobile devices and allows to perform the three canonical manipulations to the model. Each phone is holded in one hand and they are in charge of making different manipulations. Thus, to rotate the model, the screen of the device of the right has to be pressed and the device has to be rotated meanwhile; to move the model, the screen of the device of the left has to be pressed and the device has to be rotated meanwhile; to scale the model the screens of both devices have to be pressed and the devices have to be rotated in opposite directions.

For this project, a communication system has been developed to send and receive data between two devices. The communication in the interaction with one device consists in sending the information of the sensors to the computer whenever needed and transforming this information into a manipulation to the model. On the other hand, with two devices, the information of both devices is mixed in one of them and sended to the computer where, again, it will be translated into actions in the model.

It has been studied which sensors where the most suitable for this task, how to make the interaction smooth and ergonomic and how to synchronize the information obtained by the sensors. It has also been tested the different ways of interaction and made a comparison between them.

Yo, **Adrián de la Torre Rodríguez**, alumno de la titulación Grado en Ingeniería informática de la **Escuela T´ecnica Superior de IIngenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 75579747E, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Adrián de la Torre Rodríguez

Granada a 16 de Junio de 2018.

D. **Alejandro José León Salas**, Profesor del Área de Lenguajes y Sistemas Informáticos del Departamento LSI de la Universidad de Granada.

Informa:

Que el presente trabajo, titulado ***Inmersión e interacción sobre escenas 3D mediante dispositivos móviles***, ha sido realizado bajo su supervisión por **Adrián de la Torre Rodríguez**, y autorizo la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expide y firma el presente informe en Granada a 16 de Junio de 2018.

El director:

Alejandro José León Salas

ÍNDICE GENERAL

3. Introducción

3.1 Motivación

La producción y proliferación de nuevos dispositivos para sistemas de realidad virtual/realidad aumentada (Virtual Reality/Augmented Reality, VR/AR) ha crecido en importancia en los últimos años. Esto se debe al abaratamiento de los dispositivos, las nuevas capacidades presentes en estos, tanto a nivel de realismo en la inmersión como a nivel de precisión en los sensores. Junto a esto, han proliferado y continúan haciéndolo multitud de aplicaciones distintas de las tradicionales asociadas a centros de investigación y grandes empresas multinacionales. Algunos ejemplos son las exploraciones de escenas digitales (virtuales) en varios ámbitos y los videojuegos. Por ejemplo Audi [10] lanzó una aplicacion en realidad virtual para que los usuarios pudieran ver su coche personalizado (Figura 3.1). Un ejemplo de videojuego es Adrift [9] donde se explora una nave en el espacio exterior (Figura 3.2).

Figura *3.1: Audi VR Experience*

Figura *3.2: Adrift*

A raíz de esto han surgido infinidad de estudios y proyectos en los que se pretenden innovar en interacción virtual ya sea con dispositivos hardware o aplicaciones software. Grandes y medianas empresas han invertido en desarrollar kits de interacción 3D bastante inmersivos pero de un coste elevado. Hay un hueco en el mercado en interacción con coste económico nulo para el usuario. Este hueco se pretende analizar y plantear una propuesta para suplirlo.

3.2 Objetivos

El objetivo del proyecto es conseguir interactuar con un espacio virtual 3D o con los objetos que están en él de manera intuitiva, fácil, inmersiva y con dispositivos relativamente económicos para que esté al alcance de la mayoría de personas. Además, el modelo de interacción desarrollado deberá ser extensible al uso de gafas de realidad virtual y afrontar los inconvenientes que surgen en el uso de estas.

Con la interacción se pretende hacer una exploración 3D de un modelo virtual que se desea examinar. La exploración se hará aplicando al modelo una serie de transformaciones que permitirán visualizar con mayor detalle la parte de él que se desee.

El uso que se le pretende dar al método de interacción es que pueda ser usado en la docencia y/o el estudio en distintas ramas del conocimiento. Así un estudiante podría observar la reproducción 3D de cualquier cosa y explorarla de manera más ergonómica a la que predomina actualmente que es el uso de teclado y ratón. Por ejemplo, se podría usar para la exploración de un modelo de corazón humano para una clase de anatomía o una reproducción de un monumento para una clase de arquitectura. En la figura 3.2 se muestra un ejemplo de las transformaciones necesarias para mostrar con detalle la zona señalada del modelo de una casa.

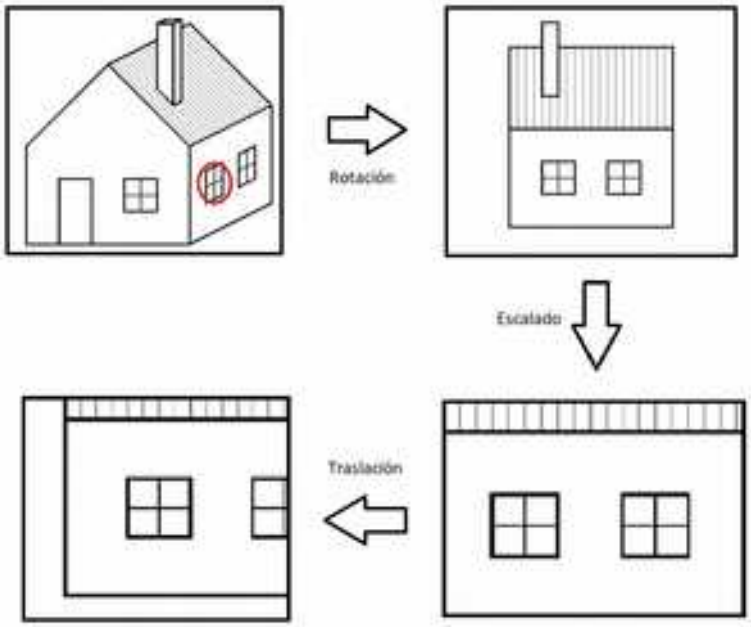

Figura 3.2: Exploración de un modelo de una casa

En la docencia se le pretende dar un uso parecido, por ejemplo mientras se muestra un modelo 3D en la pantalla de un proyector, el profesor podría explicar las distintas partes de él aplicándole las transformaciones comentadas

Este objetivo nos plantea unas preguntas iniciales que se deben resolver para empezar a abordar una solución partiendo de una base sólida y con un camino claro y marcado. Las preguntas son las siguientes:

- ¿De qué manera puedo interactuar con un espacio virtual 3D o con un objeto situado en él?
- ¿Qué tipo de acciones podría llevar a cabo el usuario en el mundo virtual mediante el método de interacción?
- ¿Qué métodos de interacción se han desarrollado anteriormente?
- ¿Qué dispositivos físicos hay disponibles para esta tarea y cuáles son los más apropiados partiendo de las anteriores premisas?
- ¿Qué puedo hacer yo para mejorar los anteriores métodos?
- ¿Cómo puedo crear un método de interacción compatible con gafas de realidad virtual?
- ¿Cómo puedo crear este sistema de interacción con los conocimientos adquiridos en la carrera?
- ¿Cómo puedo hacer para que sea intuitivo, ergonómico y con una respuesta correcta?

3.3 Alcance de la memoria

Esta memoria pretende dar una visión detallada del trabajo desarrollado, los problemas encontrados y cómo se han resuelto. También se describe el proceso de desarrollo, desde el análisis del problema y planificación hasta la implementación y pruebas. Por otro lado se incluyen manuales en los que se explican los pasos a llevar a cabo para instalar el proyecto y cómo usarlo.

Toda la información necesitada para la implementación del proyecto ha sido obtenida de diversos sitios de la red y de apuntes y documentos adquiridos durante el transcurso de la carrera. En cuanto al contenido teórico, se han obtenido de las páginas webs de los fabricantes de los productos y herramientas que se exponen y sobre todo del libro *3D User Interfaces: Theory and Practice* [1].

4. Estado del arte en interacción 3D sobre escenas virtuales

El mercado de la interacción 3D está experimentando un gran desarrollo desde hace algunos años debido a que la producción en masa de dispositivos de interacción e inmersión virtual han hecho que abaraten sus costes y puedan llegar a mayor parte de la población.

Con fines tanto profesionales como para el ocio, medianas y grandes empresas han desarrollado dispositivos que abarcan distintos métodos de interacción en espacios virtuales. A continuación se muestran algunos ejemplos:

- **Captura de movimiento:** mediante cámaras, dispositivos mecánicos u otros dispositivos se captura la posición del cuerpo humano y se transforma en acciones o movimientos en el espacio virtual. Microsoft introdujo al mercado hace unos años Kinect [3]; una cámara capaz de reconocer el cuerpo del usuario y

transformar sus movimientos en acciones en los videojuegos (Figura 4.1). Un ejemplo de captura de movimiento con dispositivos mecánicos es el *Gypsy 7 Motion Capture System* [4]. Se trata de un exoesqueleto que transmite la posición relativa de las partes que lo componen. Este último dispositivo es muy caros y prácticamente no se usa ya (Figura 4.2).

Figura *4.1: Kinect*

Figura *4.2: Gypsy 7*

- **Dispositivos hápticos**: estos dispositivos responden físicamente a las acciones realizadas en el espacio virtual, ya sea ofreciendo resistencia al movimiento o vibrando para crear una sensación de tacto. El principal uso que se les da es en simulaciones o entrenamientos para determinadas profesiones, por ejemplo una simulación de una operación. En la figura 4.3 se muestra SimOrtho [5], un simulador háptico de operaciones. Este simulador cuenta con una representación de una herramienta quirúrgica que va conectada a unos brazos mecánicos. El movimiento realizado a esta herramienta se verá representado en la simulación y, a su vez, los brazos mecánicos ofrecen resistencia al encontrarse en la simulación por ejemplo con huesos u otros tejidos.

Figura *4.3: Sim-Ortho*

- **Mandos clásicos**: al pulsar los botones del mando se realizan distintas acciones en el espacio 3D. La única respuesta sobre la interacción que tienen estos mandos puede ser vibración o alguna indicación luminosa. Las figuras 4.4 y 4.5 muestran los mandos más vendidos.

Figura *4.4: Mando Xbox One*

Figura *4.5: Mando Play Station 4*

- **Mandos con sensores**: estos dispositivos cuentan con sensores que permiten captar su posición relativa en el espacio y las aceleraciones que se le aplican. El ejemplo más extendido de estos mandos son los controladores de la Nintendo Wii [2]. Estos mandos cuentan con unos acelerómetros que permiten medir las fuerzas o aceleraciones aplicados a estos para calcular su orientación o movimiento. También tienen una cámara que capta las luces infrarrojas emitidas por una barra colocada en la televisión para calcular su orientación con respecto a la televisión (Figuras 4.6 y 4.7).

Figura *4.6: Mando de Nintendo Wii*

Figura *4.7: Barra de Nintendo Wii*

- **Kits de realidad virtual:** Estos kits permiten al usuario obtener una inmersión muy realista en el espacio virtual. Las últimas versiones de estos kits que han salido al mercado cuentan con un casco estereoscópico que permite obtener una imagen tridimensional del mundo virtual, además permite observalo en 360º con solo mover la cabeza. También cuentan con unos mandos con numerosos sensores y respuestas hápticas y unas cámaras destinadas a ser colocadas en el espacio físico donde se usará el kit. Estas cámaras sirven para capturar la posición del usuario en el espacio y detectar sus traslaciones. Un ejemplo de estos kits son las HTC Vive [6] que además de las características ya explicadas tienen unas cámaras en el casco estereoscópico a las que se puede dar uso con realidad aumentada(Figura 4.8).

Figura *4.8: HTC Vive*

Todos los ejemplos comentados anteriormente junto a todos los dispositivos existentes se caracterizan por su medio o alto coste económico y la necesidad de adquirir dispositivos externos.

Otra rama de la interacción es el uso de dispositivos móviles que, si bien tienen un coste económico muy variable, la mayor parte de la población ya tiene uno y no sería necesario ningún gasto extra. Los métodos de interacción más usados con los dispositivos móviles son el uso de la pantalla táctil y el uso de los sensores para interactuar con aplicaciones dentro del móvil.

Recientemente se está empezando a dar uso a los dispositivos móviles como medio de interacción con espacios virtuales, aplicaciones o juegos externos al propio dispositivo. Algunos ejemplos son:

- **Usar el móvil como controlador de aplicaciones:** Desde el propio móvil se pueden mandar acciones a una aplicación que se está ejecutando en otro dispositivo. Por ejemplo, en un ordenador está ejecutándose la aplicación de reproducción de música Spotify y puede ser controlada por la aplicación correspondiente para movil. También distintos dispositivos pueden ser controlados con el móvil como televisores y demás electrodomésticos.

- **Usar el móvil como mando para videojuegos:** A través de la pantalla táctil del móvil se puede usar como un mando tradicional para controlar videojuegos. Un ejemplo de esto es el sistema PlayLink [7] desarrollado por PlayStation (Figura 4.9) o la página web AirConsole[8] (Figura 4.10).

Figura *4.9: Videojuego Has Sido Tú con sistema PlayLink*

Figura *4.10: Página web AirConsole*

5. Desarrollo de la solución

5.1 Solución propuesta

Una vez comentados algunos de los principales métodos de interacción que hay en el mercado y teniendo en cuenta los objetivos previamente planteados, se debe proponer una solución que cumpla esos objetivos y además sea innovadora.

Como el usuario objetivo de mi proyecto se caracteriza porque usará el sistema ocasionalmente y no tiene gran capacidad adquisitiva(en el caso del estudiante), se propone que el sistema sea lo más económico posible. En el mercado se ofrecen numerosos kits de realidad virtual que ofrecen una interacción muy real con mundos virtuales, pero son de un precio muy elevado, requieren de una instalación muy compleja y de un espacio físico amplio para poder usarlos.

Mi propuesta para resolver eso es el uso de dispositivos móviles. El principal motivo es que la gran mayoría de personas y especialmente los estudiantes tienen ya uno, por lo tanto el coste económico para usar el sistema será mínimo o nulo. Además, el uso de estos dispositivos está ampliamente extendido y profundamente implantado en la sociedad por lo que no se requiere de un aprendizaje para su uso. Por otro lado, la mayoría de móviles cuentan con numerosos sensores que se pueden usar para capturar el movimiento del dispositivo de manera muy precisa.

Por lo tanto la idea general sería el uso de dispositivos móviles como medio de interacción en espacios 3D que, a diferencia de los ejemplos comentados en donde sólo se le daba uso a la pantalla táctil, se dará uso a los sensores para capturar movimientos y ser traducidos en acciones para explorar ese espacio 3D.

5.2 Planificación

Para la planificación de este proyecto se ha tenido en cuenta los recursos software y hardware disponibles, la cantidad de personas implicadas y el tiempo disponible. Una vez estimado todo se ha procedido a elegir una metodología de trabajo ágil que se adecúe a los recursos disponibles.

En cuanto a los recursos hardware y software, se deberán instalar los programas

que se estimen necesarios para desarrollar el proyecto. Al no contar con presupuesto ecónómico se dispondran de las herramientas con licencia libre en el mercado. Además será necesario más tiempo en solucionar problemas que surjan y en hacer tareas que, a priori no se habían predecido.

En conclusión, la instalación de herramientas conlleva un tiempo que dependerá de la conexión de internet y la potencia del ordenador. Sin embargo esta tarea es tan trivial que se puede paralelizar con otras, por ejemplo, mientras se instala un programa se puede empezar a investigar otra tarea a la vez. Por lo tanto, aunque el tiempo para esta fase no sea estimable con precisión, sí se puede deducir que será corto.

En cuanto a las personas trabajando en el proyecto y el tiempo que tienen disponible: en el proyecto han participado dos personas, yo (alumno de ingeniería informática) y el tutor del proyecto (profesor en la ETSIIT). Yo seré el encargado, en gran parte, de llevar a cabo el proyecto. La tarea del tutor deberá guiarme en el proceso y ayudarme a resolver problemas que surjan.

El tiempo que tengo disponible dependerá de la carga de trabajo que tenga de otras asignaturas. Así, como mucho podré estar trabajando en el proyecto no más del 50% del tiempo ya que tendré que compaginarlo con otros proyectos. El tiempo disponible del tutor dependerá también de su carga de trabajo en otras actividades ajenas a este proyecto. Por otro lado, el tutor tiene disponibles varias horas a la semana para tutorías que pueden ser usadas para planificación y resolución de dudas.

Así pues, como el tiempo disponible para cada persona es reducido, los tramos de más concentración de trabajo en este proyecto se verán reducidos a épocas en las que haya menos carga de otras asignaturas.

En cuanto al tiempo total del proyecto, teóricamente se contaba con todo el año lectivo del curso para realizarlo. A ese tiempo hay que restarle el periodo de preasignación de TFG’s en el que estuve buscando proyecto y tutor. Junto a lo anterior, el primer semestre tenía una carga de trabajo bastante grande por lo que era muy difícil y carecía de sentido comenzar un proyecto al que no podía dedicar tiempo.

Teniendo en cuenta todo esto, finalmente el tiempo disponible total para realizar el proyecto ha sido desde el comienzo del segundo semestre hasta la entrega de este, es decir, cuatro meses aproximadamente.

5.3 Metodología de desarrollo empleada

Debido a la naturaleza del proyecto se usará una metodología ágil de las estudiadas en la carrera. Estas metodologías cuentan con una mayor flexibilidad y se adaptan mejor a los cambios que las tradicionales.

Se ha optado por seguir una metodología de prototipado con algunas características de scrum [16]. Debido a que no es un gran proyecto que se pueda dividir en tareas sino que es una tarea de una complejidad elevada es más aceptable ir creando prototipos funcionales que vayan cumpliendo objetivos o requisitos del proyecto. Por ejemplo, un primer prototipo sería una aplicación capaz de transmitir datos al ordenador. Posteriormente a ese prototipo se le añadirán más funcionalidades incrementalmente.

La metodología de trabajo ha sido la siguiente: a partir del segundo cuatrimestre, se realizaban reuniones quincenales con el tutor en las que se mostraba el prototipo obtenido para esa fecha y se evaluaba. También se establecían el siguiente prototipo y las tareas a realizar para la siguiente entrega. Entre reuniones yo me encargaba de desarrollar el prototipo siguiente, siempre tomando nota en un diario de todo lo que hacía, problemas que encontraba, cómo los solucionaba, etc. Todo esto para poder realizar una memoria lo más detallada posible y que no se quedaran puntos importantes en el olvido.

5.4 Análisis del problema

Como ya se ha comentado, se propone interactuar con un modelo 3D en un espacio 3D para su exploración usando como medio uno o dos dispositivos móviles haciendo uso de los sensores que tienen.

Esto plantea otras preguntas que se deben sumar a las preguntas comentadas en el apartado de objetivos:

- ¿Cómo transmito la información de los sensores del móvil?
- ¿Qué sensores son los que aportan la información necesaria?
- ¿Cómo interpreto esta información?

Si además se quiere extender este método de interacción al uso de gafas de realidad virtual es necesario tener en cuenta el principal problema del uso de controladores físicos mientras se usan estas gafas. Las actuales gafas de realidad virtual no permiten ver el mundo real mientras se llevan puestas, por lo tanto, un dispositivo complejo será difícil de manejar. En nuestro caso, la interfaz del móvil no debe ser muy compleja. Para resolver esto existen dos soluciones:

- Representar virtualmente el dispositivo físico para poder ver en todo momento cómo se está usando.
- Crear una interfaz muy simple que no dé lugar a errores

Las acciones que se van a realizar son tan simples que no es necesario incluir mucha funcionalidad a la interfaz y se puede compensar con el uso de los sensores que obtienen datos del movimiento del dispositivo. Por lo tanto se propone crear una aplicación móvil con una interfaz extremadamente sencilla y que envíe los datos de los sensores y botones de la interfaz para poder ser interpretados en el modelo 3D.

En los siguientes apartados se enunciarán los requisitos que el proyecto deberá cumplir. Es necesario indicar que este proyecto no está destinado a ser un producto final entregable a un usuario. Su objetivo, más bien, es crear un método de interacción que pueda ser integrado en un proyecto más cuidado de interacción con objetos 3D.

5.4.1 Requisitos funcionales

- **RF1** - El sistema permitirá transformar los movimientos físicos de uno o dos móviles en transformaciones de un modelo 3D.

    - **RF1.1** - El sistema podrá enviar y recibir datos entre un dispositivo móvil y el ordenador.
    - **RF1.2** - El sistema podrá enviar y recibir datos entre dos dispositivos móviles y el ordenador.
    - **RF1.3** - El sistema interpretará los datos recibidos de los móviles en transformaciones de un modelo 3D.

- **RF2** - El sistema contará con varios métodos de interacción sobre el sistema 3D.
    - **RF2.1** - El sistema contará con varios métodos de interacción con un solo dispositivo móvil
    - **RF2.2** - El sistema contará con varios métodos de interacción con dos dispositivos móviles

- **RF3** - La app de los móviles tendrá una interfaz en la que se podrá elegir el tipo de interacción desarrollada.

- **RF4** - La app permitirá leer los datos de los sensores del propio dispositivo.

- **RF5** - Habrá un espacio virtual 3D donde probar los métodos de interacción.
    - **RF5.1** - En el espacio virtual habrá un ejemplo de modelo 3D para poder interactuar con él.

5.4.2 Requisitos no funcionales

- **RNF1** - La interfaz de un método de interacción será lo más simple posible.
- **RNF2** - El espacio 3D desarrollado será lo más simple posible para demostrar el funcionamiento del sistema.
- **RNF3** - La respuesta entre el envío de un mensaje y su interpretación debe ser lo más pequeña posible.
- **RNF4** - Las transformaciones en el modelo 3D serán suaves y correspondientes a los movimientos del móvil.
- **RNF5** - Los movimientos necesarios a realizar en el móvil deben ser lo más ergonómicos posible.
- **RNF6** - El sistema podrá ser importado en otros proyectos.
- **RNF7** - El sistema solo contará como máximo con dos dispositivos móviles, un ordenador y una red entre ellos. No habrá más dispositivos externos.

5.4.3 Requisitos de información

- **RI1** - La información que se envía y recibe debe ir empaquetada en un formato conocido y exportable.

5.5 Diseño de la solución

El proyecto cuenta con dos partes diferenciadas que estarán conectadas continuamente: la parte de los dispositivos móviles (Android) y la parte del ordenador (Unreal Engine [11]). A partir de aquí se comentará el diseño y desarrollo de las dos partes de manera separada ya que el medio por el que se comunican siempre será el mismo (usando TCP Sockets vía red Wifi). Lo único relevante en común a las dos partes es qué información se envía y se recibe y en qué formato se envía. Teniendo en cuenta eso, los demás aspectos del desarrollo como la implementación de ambos entornos, cómo obtenemos información de los sensores o cómo es interpretada podrá ser desarrollada individual e independientemente.

5.5.1 Diagramas

En este apartado se muestran todos los diagramas desarrollados en los que se muestran organización y composición de las clases, organización de comunicación y estructura del sistema. En cada diagrama se explica a qué parte del sistema pertenece y se incluye una descripción más detallada para permitir una comprensión del funcionamiento de esa parte en sí y el papel que toma en el sistema completo.

Antes de exponer el funcionamiento y composición de cada parte es conveniente tener una visión general del sistema. A continuación se muestra los esquemas de los componentes del sistema y cómo están relacionados para uno y dos dispositivos móviles.

En la figura 5.1 muestra un esquema de la organización del sistema para un dispositivo móvil. La aplicación del móvil ha sido desarrollada con la herramienta Android Studio [12] en el lenguaje de programación Java. Esta aplicación es la encargada de recolectar los datos de los sensores, empaquetarlos y enviarlos a través de un socket TCP al ordenador vía red Wifi. En la interfaz de esta aplicación se puede elegir el método de interacción deseado y por consecuencia, los datos que se envían serán distintos. El entorno del ordenador tendrá que estar configurado y preparado para ese envío de datos.

En cuanto a la parte del sistema correspondiente al ordenador, se ha desarrollado

un entorno 3D en el motor gráfico Unreal Engine 4 [11]. Este motor permite desarrollar clases c++ con el entorno de programación Visual Studio [14]. En él se ha implementado las clases encargadas de recibir los datos, desempaquetarlos y realizar las operaciones oportunas con ellos para poder ser interpretados por el motor. Por otro lado el motor Unreal Engine proporciona un método de programación por tablas de flujo en el que se han implementado los distintos métodos de interacción. Los aspectos más técnicos serán explicados en el apartado de implementación.

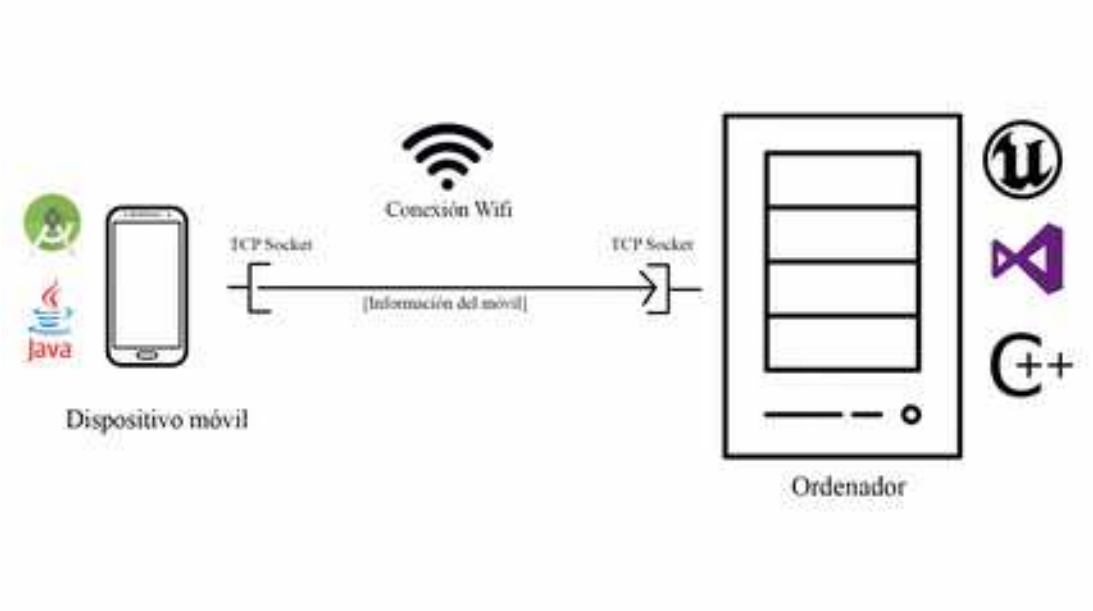

Figura 5.1: Esquema del sistema para un dispositivo móvil

La figura 5.2 muestra la organización a alto nivel del sistema para el uso de dos dispositivos móviles en la interacción. En este caso se han obviado los entornos de desarrollo ya que son los mismos y se ha hecho énfasis en cómo se comunican los distintos dispositivos.

Hay dos dispositivos móviles: izquierda y derecha según en qué mano serán sujetados. El dispositivo izquierda enviará sus datos al dispositivo de la derecha, este actuará como intermediario y juntará los datos que ha recibido con los suyos y los enviará al ordenador. Al igual que el caso anterior, los datos se empaquetan para enviarlos y se desempaquetan al recibirlos. Ahora el ordenador tendrá que trabajar con los datos de ambos móviles e interpretarlos. La interpretación será explicada con detalle en el apartado siguiente.

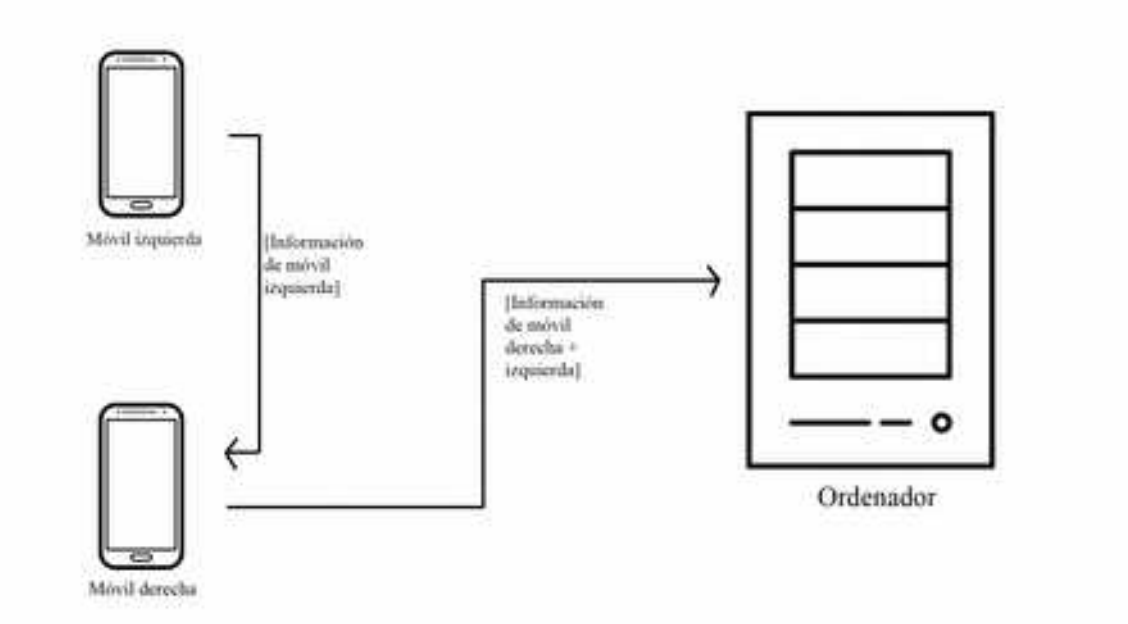

Figura 5.2: Esquema del sistema para dos dispositivos móviles

La figura 5.3 muestra el diagrama de clases UML que implementa la solución para la aplicación Android. Consta de varias clases java que en este entorno de programación se llaman actividades. Todas las actividades heredan de una clase madre llamada AppCompatActivity. Las actividades van acompañadas de un archivo xml en el que se muestra su apariencia por pantalla. Su uso se asemeja al modelo de vista controlador. Además se han implementado otras clases auxiliares, las cuales no son actividades sino que son clases estándar. A continuación se ofrece una explicación detallada de cada clase:

- **MainActivity:** Primera clase que aparece al ejecutar la aplicación. En ella aparecen una serie de botones para lanzar otras actividades encargadas de hacer la interacción.
- **UserActivityFree:** Esta actividad consta de un botón y un slider. Se enviarán mensajes a la IP indicada mientras se te tenga pulsado ese botón.
- **UserActivityAxis:** Esta actividad consta de tres botones, uno para cada eje de rotación (x,y,z). Mientras se tenga pulsado uno de esos botones se enviarán mensajes al receptor.

- **JSONActivity:** La funcionalidad de esta actividad se reduce al simple testeo del empaquetamiento de mensajes en formato JSON y su envío al ordenador.
- **ChooseLRActivity:** Esta actividad está pensada para el uso en conjunto de dos dispositivos móviles. En ella se elige qué tipo de dispositivo es (izquierda o derecha). Esta actividad lanza la siguiente actividad UserActivityLR.
- **UserActivityLR:** La interfaz de esta actividad es básicamente un botón que mientras sea pulsado enviará los mensajes al dispositivo correcto. Internamente se comportará de una manera distinta dependiendo del dispositivo elegido en la actividad anterior.
- **JSONUtilities:** Clase auxiliar que contiene los métodos para empaquetar los mensajes en formato JSON.

- **MessageSender:** Todas las actividades que envían mensajes tienen un objeto de esta clase ya que es la encargada de crear la conexión con el receptor y enviar los mensajes que se indiquen.
- **MessageReceiver:** Esta clase sólo la contiene la actividad UserActivityLR ya que, al haber un dispositivo intermediario, debe tener una clase encargada de recibir mensajes.

Tanto MessageSender como MessageReceiver heredan de la clase AsyncTask. Esta clase del paquete android.os permite lanzar una hebra que es usada para ejecutar el bucle de envío o recepción de mensajes. Además, todas las clases que necesitan acceso a los sensores del dispositivo implementan la interfaz SensorEventListener encargada de esta tarea.

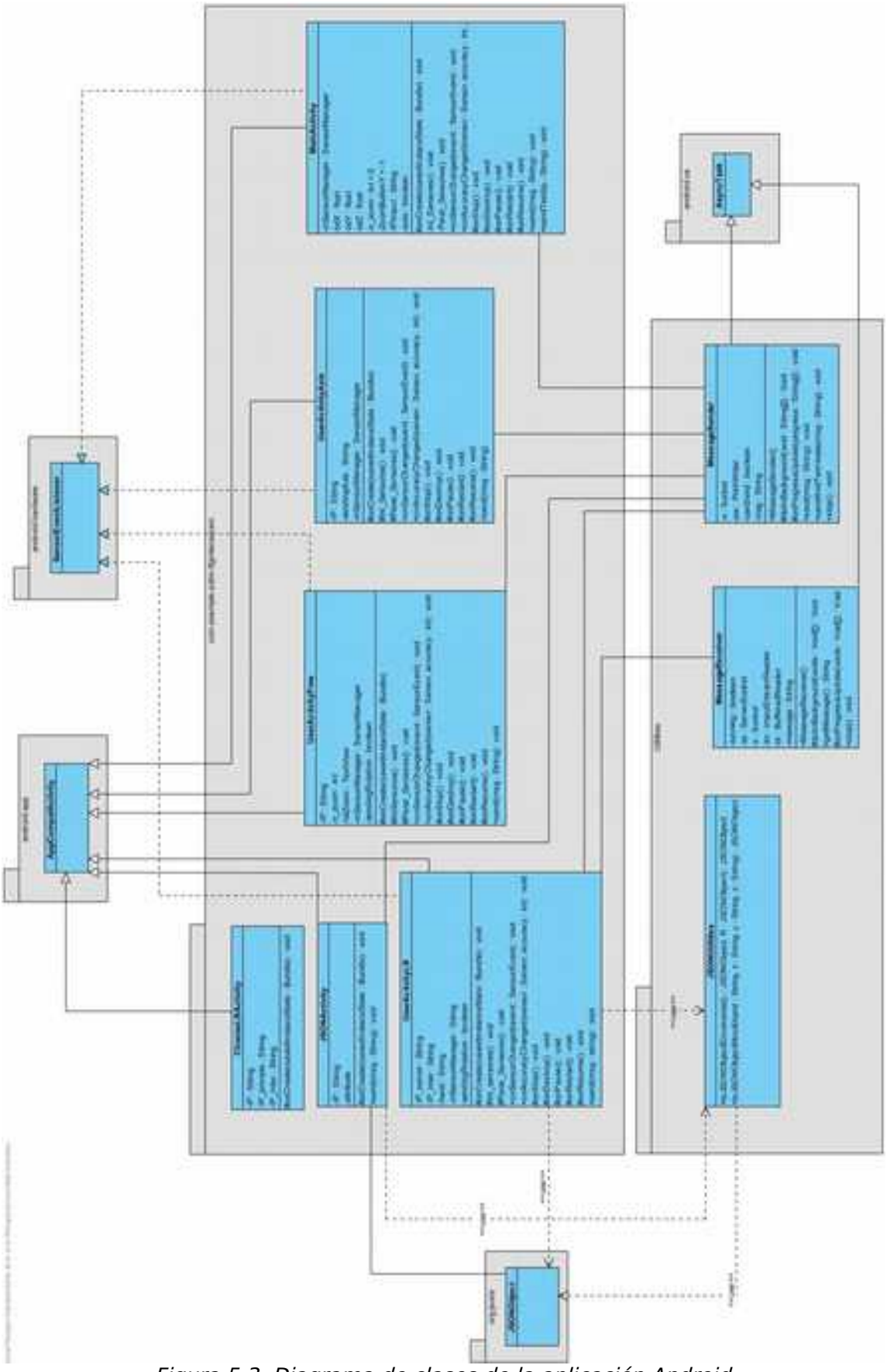

Figura 5.3: Diagrama de clases de la aplicación Android

Para lograr alcanzar una mayor comprensión de cómo se comunican los dispositivos involucrados se han realizado dos diagramas de secuencia, los cuuales son mostrados en las figuras 5.4 y 5.5. En ellos se explica el proceso de comunicación e interpretación en la interacción con un lenguaje de más alto nivel.

En la figura 5.4, el diagrama de secuencia muestra como la guarda “sending” se activará de diferente manera dependiendo del método de interacción elegido. Generalmente será true cuando se pulse un botón. Cuando se desee interactuar con el modelo, se pulsará el botón y la aplicación se encarga de recopilar datos de los sensores, empaquetarlo y enviarlos al ordenador. Una vez el ordenador los recibe, los desempaqueta y los interpreta.

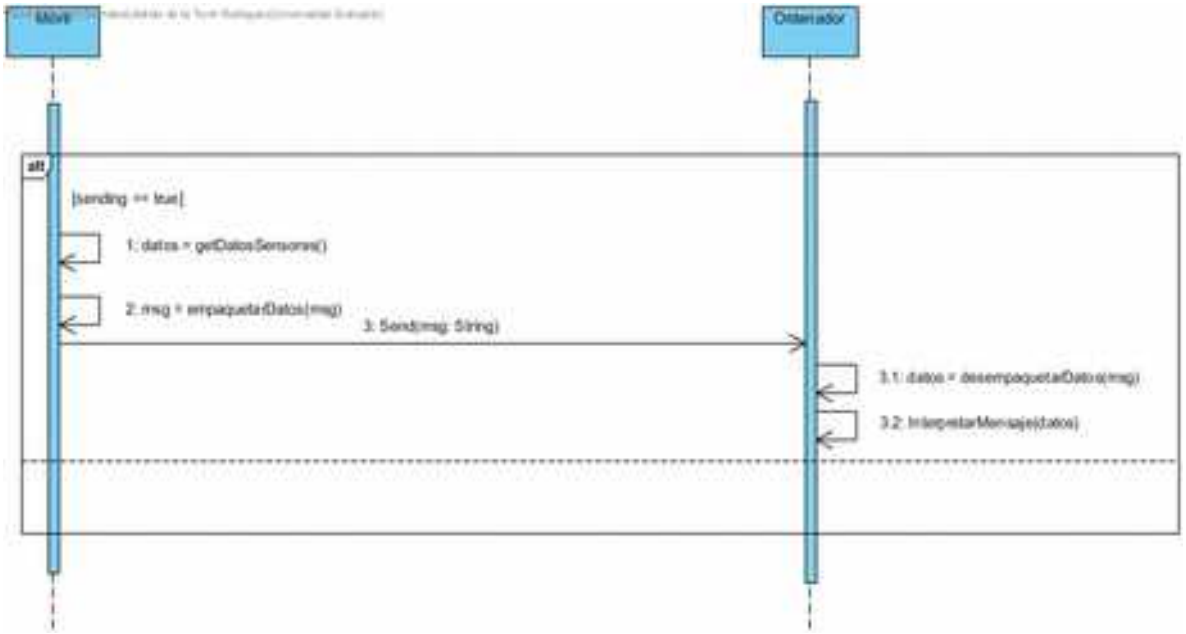

Figura 5.4: Comunicación usando sólo un dispositivo móvil

En la figura 5.5 se muestra un diagrama de secuencia que explica el siguiente comportamiento (notaremos por comodidad al móvil izquierda como L y al de la derecha como R):

- L siempre envía mensajes a R cuando se requiere.
- R siempre envía mensajes al ordenador cuando se requiere
- R reenvía el mensaje de L al ordenador si no se requiere que se envíen datos de R también.
- R envía una combinación de datos de L y R al ordenador cuando se requiere que se envíen datos de ambos dispositivos a la vez.

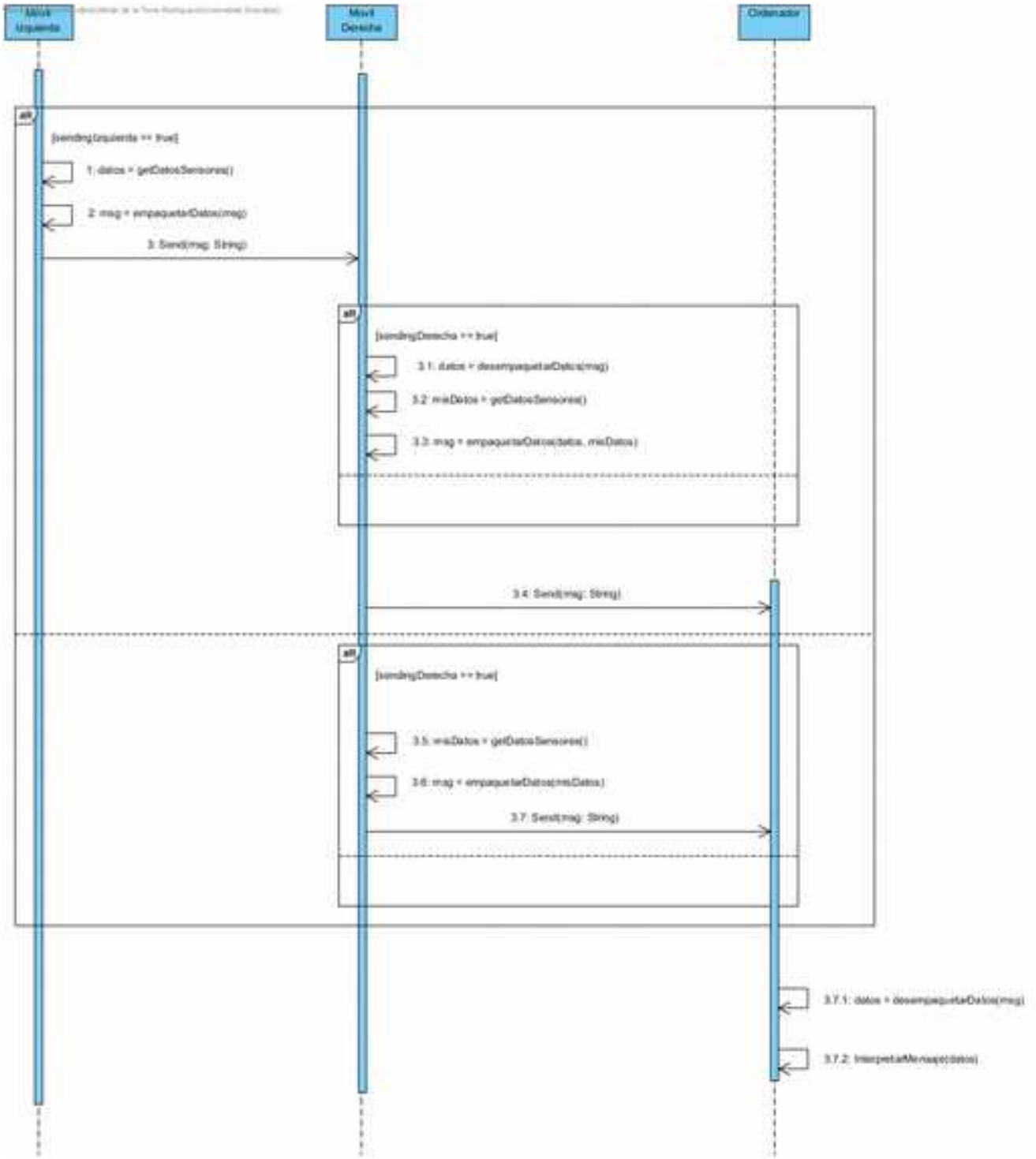

Figura 5.5: Comunicación usando dos dispositivos móviles.

Para la parte del ordenador se han creado clases dentro del proyecto de Unreal

Engine. Solo se indicarán las clases creadas por mí para lograr la interacción. La figura 5.6 muestra el diagrama de clases UML para la implementación en la parte de Unreal Engine 4. Las clases creadas son las siguientes:

- **AServerThread:** Esta clase implementa la interfaz FRunnable la cual aporta la funcionalidad de lanzar una hebra en paralelo a la hebra principal. Esta clase se encarga de la gestión de esta hebra y además cuenta con una clase anidada llamada ServerTCP que se explica a continuación.
- **ServerTCP:** Clase que se encuentra anidada en AServerThread. Es la que se lanza en la hebra y se encarga de ejercer la funcionalidad de recibir mensajes. Para ello crea todos los sockets y conexiones necesarias.
- **AControladorServer:** Esta es la clase más importante pues es la encargada de inicializar el servidor y además aporta todos los métodos necesarios para cada tipo de interacción. Hereda de la clase de Unreal AActor por lo que puede ser instanciada en el espacio 3D como un elemento de él. Por otro lado los métodos de esta clase cuentan con unos macros que sirven para que el motor reconozca las funciones y puedan ser usadas en el blue print.

Las funciones que serán usadas para la interpretación de datos están en AControladorServer pero se usarán en el BluePrint de la clase que, como he comentado anteriormente, se programa usando tablas de flujo con nodos. Las funciones creadas serían los nodos pero eso se comentará en el apartado de Implementación.

Es necesario indicar que los diagramas que se han mostrado en este apartado son los obtenidos finalmente. Como se ha comentado, el método de trabajo ha sido el prototipado, por lo tanto se han ido añadiendo clases a la aplicación Android y métodos a las clases de Unreal Engine conforme se iban desarrollando nuevos prototipos.

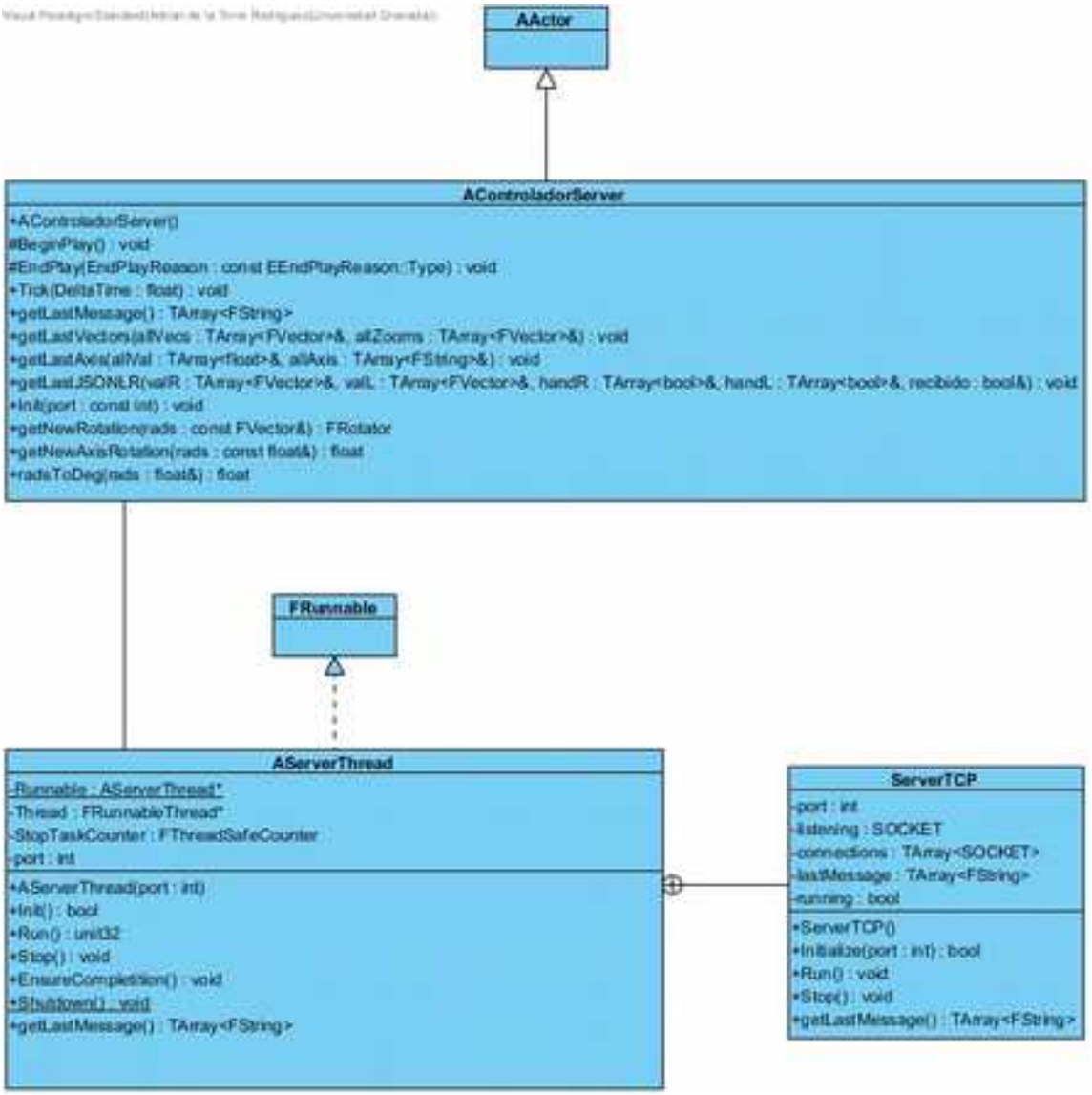

Figura 5.6 Diagrama de las clases creadas para el proyecto de Unreal Engine

5.5.2 Interpretación de datos y métodos de interacción

Una de las partes más importantes del proyecto es la interpretación de datos, es decir, cómo se transforman los datos recibidos de los móviles en acciones sobre el modelo. Antes de eso se explicará el sensor del móvil elegido y por qué.

Los dispositivos móviles actuales cuentan con muchos sensores tales como GPS, acelerómetro, magnetómetro, giroscopio, de gravedad y un largo etcétera. Inicialmente se intentó escoger como sensor el acelerómetro; este sensor es capaz de medir las aceleraciones o lo que es lo mismo, las fuerzas aplicadas al dispositivo (incluida la de la gravedad). Por lo tanto era posible detectar movimientos aplicados al móvil e incluso calcular la inclinación del plano SU XY con respecto al centro de la tierra como se ilustra en la figura 5.7:

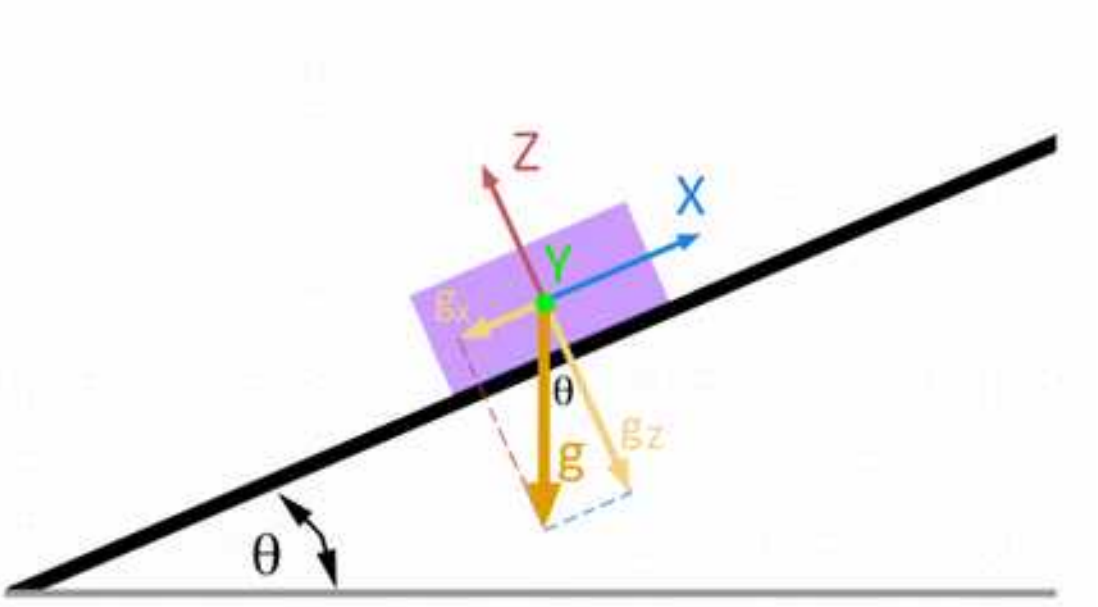

Figura 5.7: Esquema de fuerzas aplicadas al móvil

El ángulo Θ se puede calcular fácilmente con el *arccos(gz/g)*. Esto era muy útil para trasladar la orientación absoluta del dispositivo móvil al modelo, es decir que hubiera una correspondencia de orientación entre el móvil y el modelo, pero se encontró un gran inconveniente: no se detectaban las rotaciones en el eje Z. Como se muestra en la figura 5.8 ambos dispositivos leerían los mismo valores de fuerzas.

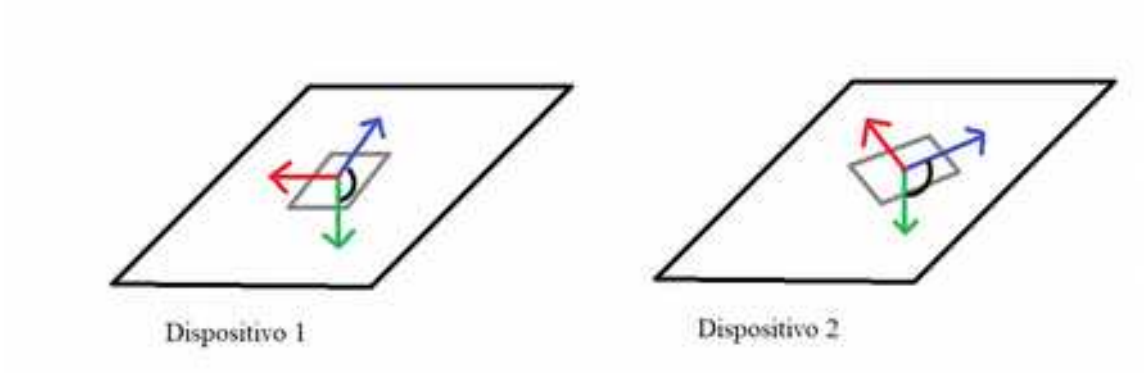

Figura 5.8: Orientacion de dispositivos

En esta figura el plano mostrado(negro) es perpendicular al eje z (verde) y como se puede observar, a pesar de que el móvil tenga orientaciones distintas en ambos casos, el ángulo del eje z con respecto al plano XY es siempre el mismo (90º). Por lo tanto las rotaciones en el eje z no se podrían captar. Por consecuencia se descartó el uso del acelerómetro y de interpretaciones de orientaciones absolutas para la interacción.

Tras este problema se optó por capturar las rotaciones relativas del móvil, es decir, las variaciones en la orientación del móvil sin tener en cuenta su orientación original en el mundo físico. Esto es una opción aceptable ya que con el anterior método si se quería ver la parte trasera del modelo habría que girar el móvil 180º lo que era muy poco ergonómico e ineficaz. Con el nuevo método basta con girar el móvil en la dirección deseada, se captura la diferencia de rotación y se aplica al modelo.

El nuevo sensor escogido es el giroscopio y es el que se ha usado finalmente en todo el proyecto. Este sensor calcula las velocidades de rotación en cada eje en radianes por segundo(rads/s). El uso que se le da a ese sensor es enviar los datos de rotación al ordenador cuando se desee. Llevado a la práctica consiste en pulsar un botón, mientras se tenga pulsado se enviarán los datos del giroscopio y estos serán interpretados en el ordenador. El hecho de pulsar el botón para enviar datos evita que se envíen movimientos involuntarios de la mano y se hagan acciones en el modelo no deseadas.

Los datos que se envían, el formato en que se envían y cómo se interpretan varían dependiendo del método de interacción. A continuación se exponen los distintos métodos de interacción creados y cómo se interpretan los datos en cada uno de ellos.

A) Rotación libre y zoom

Este método de interacción permite rotar el objeto en los tres ejes con todo grado de libertad y también se le puede aplicar un escalado. Los datos necesarios son los rad/s que captura el móvil en cada eje y el valor de zoom que se aplica que está comprendido entre 1 y 100.

Para este método se han estudiado cuáles son los valores más comunes de radianes por segundo en el movimiento de la mano. Para ello se ha usado una aplicación que muestra los datos del giroscopio y se ha observado que los valores más frecuentes van de 1 a 4 rads/s.

Con estos datos se ha hecho una correspondencia entre radianes por segundo y grados de rotación a aplicar al modelo. A los valores de radianes por segundo más comunes se les ha adjudicado un rango más amplio de grados. La correspondencia se muestra en la tabla 5.1:

Rads/s	deg
1	10
2	45
3	80
4	115
5	130
6	140
7	150
8	160

Tabla 5.1: Correspondencia entre rad/s y grados

Por otro lado se ha aplicado un filtro de paso alto que sólo interpreta valores de rads/s mayores a 0.5 para evitar movimiento involuntarios. En conclusión para los valores más comunes se ha usado la función de interpolación siguiente:

```
f(rads) = 1.5 * rads - 0.5
```

Para los valores mayores a 4 se ha usado la siguiente función:

```
f(rads) = rads + 4
```

Estas funciones han sido aplicadas sobre los radianes recibidos y se han obtenido grados que han sido usados para añadir las rotaciones al modelo en los ejes adecuados. Por otro lado, para el zoom se ha hecho una interpolación lineal que transforma valores de 0-100 a una escala 1-2 de la siguiente manera:

```
f(zoom) = zoom/100 + 1
```

El efecto que tienen estas funciones de interpolación de rotación es que si movemos la mano más rápido, se captará un valor de radianes por segundo más alto y esto será traducida en una rotación mayor. Esto consigue una interacción bastante cómoda ya que por ejemplo un giro de 360 grados en el modelo se puede conseguir con una rotación rápida de mano sin llegar a tener que girar el móvil los 360 grados.

B) Rotación sobre ejes

Este método difiere del anterior en que sólo se puede rotar un eje a la vez. Para este método solo se requiere los radianes por segundo captados en el eje Y del giroscopio del móvil y un eje que se desea hacer la rotación. Esta vez no se puede aplicar zoom porque complicaba demasiado la interfaz.

La interpolación usada es la misma que el método anterior pero solo se aplica la función sobre el eje indicado del modelo. Los datos enviados son un valor de radianes por segundo y un eje donde aplicar la rotación.

Una vez se reciben los datos, dependiendo del eje de rotación recibido se muestra un anillo en el modelo para asegurar el eje seleccionado y se rota según los radianes recibidos usando las funciones del método anterior. En la figura 5.9 aparece un ejemplo de modelo 3D que muestra los tres anillos simultáneamente:

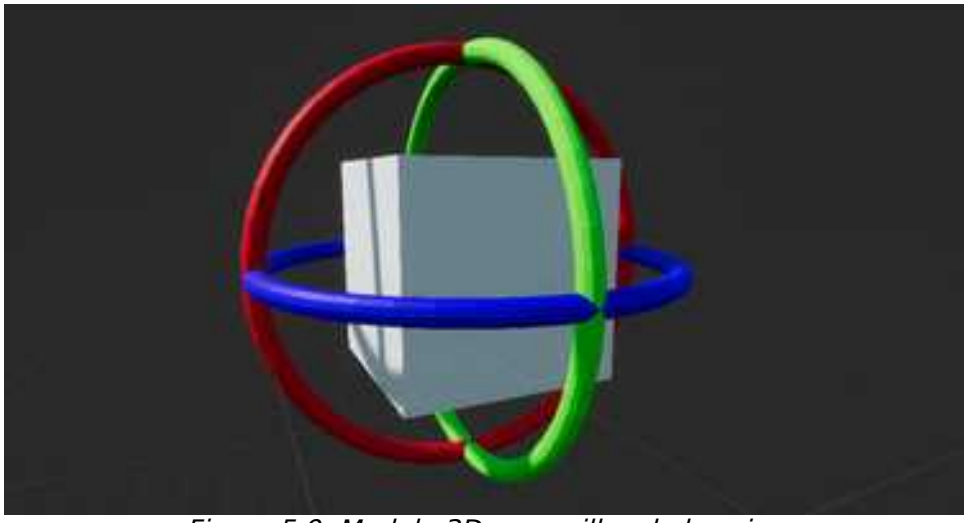

Figura 5.9: Modelo 3D con anillos de los ejes

C) Interacción con dos móviles

En esta versión tenemos dos dispositivos móviles que por separado realizan una acción en el modelo y en conjunto realizan una tercera. El método de transmisión usado para este método ya ha sido explicado anteriormente por lo que se explicará desde el punto de vista del ordenador.

El ordenador puede recibir o bien datos del móvil izquierda, datos del móvil derecha o de ambos. Si recibe sólo datos del móvil derecha se trata de datos de rotación del modelo por lo que se usarán los métodos explicados en los apartados anteriores. En el caso de que reciba datos del móvil izquierda se traducirán en traslaciones del modelo de la siguiente manera:

- Solo se harán traslaciones en los ejes X y Z(arriba, abajo, izquierda y derecha) ya que no interesan traslaciones en el eje Y que es el de profundidad.
- Los radianes por segundo necesarios son los del eje Y y X del móvil que controlarán la traslación en los ejes X y Z del modelo respectivamente.
- La función que se usa devuelve una traslación a sumar a la posición del modelo y es la siguiente:

```
f(rads) = rads*2
```

Cuando el ordenador recibe datos de los móviles izquierda y derecha simultánea-

mente se trata de una intención de hacer zoom. Para calcular el nuevo zoom se cogen los datos del eje Y del móvil derecha y se restan los datos del eje Y del móvil de la izquierda. La figura 5.10 muestra el zoom que se hace dependiendo de la rotación de los móviles. Si la resta sale positiva se aumenta el zoom y viceversa.

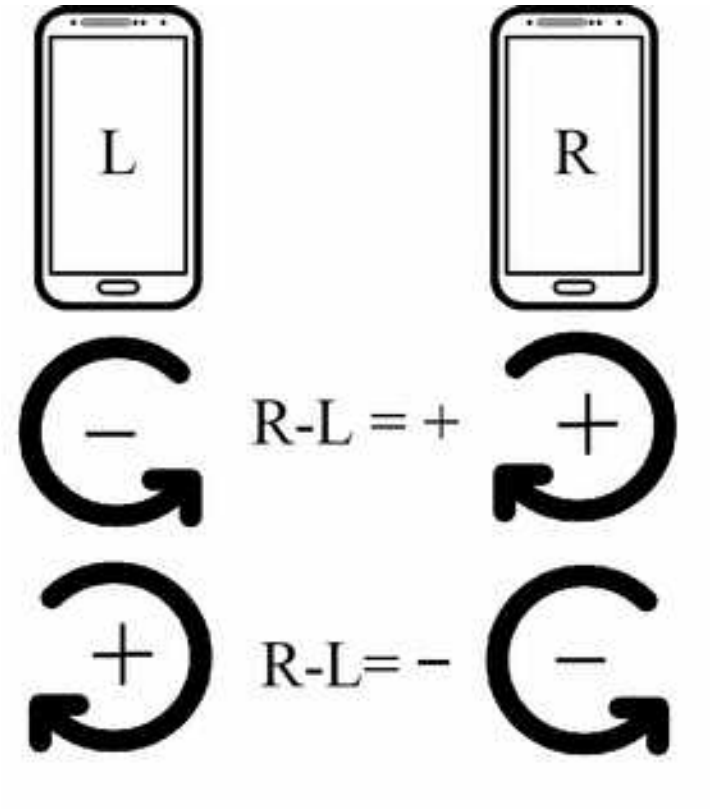

Figura 5.10: Cálculo de zoom

Con el resultado de la resta se aplica la siguiente función que devuelve el zoom a sumar o restar al modelo 3D.Esta función básicamente reduce mucho el resultado de la resta porque aplicar un escalado es una operación muy sensible y se deben usar valores pequeños.

```
f(resta) = resta * 0.01
```

En conclusión con este método de interacción podemos realizar las tres transformaciones canónicas propuestas con las interfaces de los móviles simplificadas al máximo(solo tienen un botón) y de manera simple y ergonómica.

5.6 Implementación

En este apartado se hablará de los aspectos más técnicos en el desarrollo de este proyecto tanto como los entornos de programación escogidos, el código creado, problemas que han surgido, cómo se han logrado los objetivos propuestos y las pruebas realizadas. También se dará una visión del desarrollo incremental que se ha seguido y los prototipos que se han hecho.

5.6.1 Elecciones tecnológicas

Este proyecto ha sido desarrollado gracias a una serie de entornos de programación y SDK's con licencia gratuita. Para la parte de la creación de la aplicación del móvil se ha usado Android Studio, concretamente la versión 3.0.1. Se ha elegido esta herramienta porque su uso es el más extendido, la interfaz es muy sencilla y tiene una basta compatibilidad con todas las versiones de Android. Además cuento con conocimiento previo en el desarrollo de aplicaciones en ella que adquirí en la asignatura del grado *Dirección y gestión de proyectos(DGP)*, así que no tuve que dedicar tiempo a aprender a usar otra herramienta. La aplicación se ha codificado y compilado con la API 15: Android 4.0.3(IceCreamSandwich) para que pueda ser ejecutada por la gran mayoría de dispositivos.

Para la parte de ordenador se ha usado el motor de desarrollo de videojuegos Unreal Engine en la versión 4.18.2. Aunque hay más herramientas en las que se hubiera podido realizar este proyecto como por ejemplo Unity3D o GameMaker:Studio, se ha elegido Unreal porque cuenta con un sistema de programación basado en tablas de flujo y nodos que es bastante visual e intuitivo de usar. El lenguaje que se usa es c++ y el entorno de programación es Visual Studio, concretamente he usado la versión 15.5.7. Además, también he trabajado con él en la asignatura *Programación Gráfica de Videojuegos(PGV)* y contaba con un conocimiento básico.

Por otro lado, para la realización de pruebas de comunicación, en el ordenador se ha usado NetBeans para montar un pequeño servidor que recibe mensajes en java y también se ha usado Visual Studio para crear ese mismo servidor en C++. En una ocasión se usó Blender[13] para diseñar un nuevo tipo de modelo 3D.

En cuanto al hardware usado, se ha usado un PC de sobremesa con sistema operativo Windows 10 el cual ejecuta Unreal Engine, Visual Studio, Android Studio y NetBeans. Además de ser capaz de ejecutar esos programas el ordenador debe estar

conectado a la misma red que los dispositivos móviles.

Los dispositivos móviles que se han usado han sido un Huawei Honor 8 con Android 7.0, conexión wifi y un sensor de giroscopio entre otros. Para la interacción con dos móviles también se ha usado un Xiaomi Redmi 3 con Android 5.1.1, conexión wifi y giroscopio. Tanto el modelo de los móviles como el de ordenador no importa siempre y cuando tenga sistema operativo Android y Windows 10 respectivamente y conexión a red local.

5.6.2 Arquitectura general de la solución

En este subapartado se explicará de manera ordenada desde el principio del proyecto hasta el final las tareas realizadas, objetivos cumplidos, prototipos desarrollados y problemas solucionados. Se incluirá, si es necesario, el código desarrollado para cada tarea.

A) Envío de información desde un móvil al ordenador

Para esta tarea se creó una aplicación móvil en Android Studio que creaba un socket y podía enviar un String a un dispositivo dadas una IP y un puerto. Para ello se creó una clase java encargada de esa tarea. La clase se llama MessageSender y este es su código:

MessageSender.java

```
/**
* Class in charge of creating a socket to connect to the server and send messages to it
* whenever requested.
* A thread will be created where the sender will be running
*/

public class MessageSender extends AsyncTask<String, String, Void> {

  ////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////
  ////////////////// Activity variables /////////////////////////
  ////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////

  //Socket that create a connection to the requested IP and port
```

```
private Socket s;
//Object in charge to write data on the output stream
private PrintWriter pw;
//State variables
private boolean canSend ;
private boolean running;
//Message to send
String msg;



/**
 * Constructor of MessageSender
 * Initially it will set to true the running state but it won't be able to send mes-
sages
 */
public MessageSender(){

    canSend = false;
    running = true;

    //Initially the message is set to "fail" just to test if the object has received
properly the message to send
    msg = "fail";
}


/**
 * Process to do on the thread. It will be started when the start method of the object
is called
 * @param voids Array of string that contains the information needed when the thread is
started.
 *           voids[0] = IP
 *           voids[1] = port
 */


@Override
protected Void doInBackground(String... voids){

    //Get the IP and port from the parameter
    String ip = voids[0];
```

```
    int port = Integer.parseInt(voids[1]);

    try{
        Log.i("I","Creando socket");
        s = new Socket(ip, port);
        Log.i("I","Socket creado, conectado a " + ip + " " + port);
        pw = new PrintWriter(s.getOutputStream());

        //Loop that will run while running == true
        while( running ){

            //It will only send a single message when the send method is called
            if(canSend){

                pw.write(msg);
                pw.flush();

                canSend = false;
            }
        }

        //When it's told to stop it will close all object created
        pw.write("Disconnected");
        pw.flush();
        pw.close();
        s.close();

    }
    catch (IOException e){

        e.printStackTrace();
    }

    return null;
}

/**
```

```
 * Similar method to send, we will use the send method since is more intuitive
 * @param progress
 */
@Override
protected void onProgressUpdate(String... progress){

    this.msg = progress[0];
    canSend = true;

}

/**
 * Send method
 * When this method is called it will set the msg variable with new content and
 * set the canSend state to true
 * @param msg message to send to the server
 */
public void send( String msg ){

    this.msg = msg;
    canSend = true;

}

/**
 * This method will send a message to the server and then stop the object
 * It is only used in debug mode.
 * @param msg message to send to the server
 */
public void sendAndTerminate( String msg ){

    this.msg = msg;
    canSend = true;
    try {
        sleep(100);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
```

```
        this.stop();

    }



    /**
     * Method to stop the sender
     */
    public void stop(){
        running = false;
    }
}
```

Esta clase lanza una hebra donde se ejecuta un bucle que manda mensajes siempre que se le indique con el método send. Al usar este método se actualiza una variable “msg” que tiene alcance en toda la clase y se pone a true la variable “canSend”, esto hace que en el bucle se pueda enviar el mensaje y se actualice de nuevo “canSend” a false.

Esta clase es usada por la actividad principal de la aplicación llamada MainActivity, la cual crea una instancia de ella y envía un mensaje cuando se pulsa un botón. El código de esta actividad, al incluir toda la funcionalidad del proyecto completo se incluirá en el siguiente apartado.

Para comprobar el funcionamiento se hizo un simple servidor en java sobre NetBeans el cual recibía mensajes y los imprimía por pantalla. Su código es este:

PruebaTCPSocket.java

```
/**
* This class is just to test the communication between a phone and the computer
* in the same net
* @author Adrián
*/
public class PruebaTCPSocket {


    /**
     * @param args the command line arguments
     */
    public static void main(String args[]) {
```

```
        ServerSocket ss;
        Socket s;
        InputStreamReader isr;
        BufferedReader br;

        try{
           //Create a new server socket
           ss = new ServerSocket(7800);
           //Waits until new connection
           s = ss.accept();
           //Number of messages is set to 0
           int n = 0;
           //Creation of the inputStreamReader and BufferedReader
           isr = new InputStreamReader(s.getInputStream());
           br = new BufferedReader(isr);
           System.out.println("Nueva conexion...");

           //Infinite loop
           while(true){
              //Waits until new message on the buffer
              String message = br.readLine();
              message = "Mensaje recibido: "+ (n++) +" "+ message;
              System.out.println(message);
              message = br.readLine();
           }

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

     }
}
```

Con esto ya se resolvió la tarea de envío y recepción de mensajes entre dos dispositivos sin embargo hubo que integrar ese servidor en Unreal Engine lo cual dio lugar a otra tarea que se explicará más adelante.

B) Obtención de la información de los sensores

Antes de integrar el servidor en el motor Unreal Engine se actualizó la aplicación móvil para que pudiese acceder a los sensores del teléfono. Para ello se buscó información por internet y se actualizó la MainActivity para que accediese a esos sensores, imprimiese por pantalla los datos del sensor deseado y además los enviase al servidor. En la interfaz se incluyeron dos botones “AUTO” y “TEST. El primer botón envia los datos de los sensores siempre que estuviese activo y otro botón envia un String de prueba cada vez que se pulsa. El código más importante para la obtención de los datos de los sensores de la MainActivity se muestra a continuación. También se muestra el código para lanzar nuevas actividades desarrolladas para prototipos siguientes. Para este apartado hay que centrarse en la funcionalidad de los botones “AUTO” y “TEST”:

MainActivity.java

```
/**
* Main activity of the application, here it can be found the buttons to launch the dif-
ferent interaction
* activities with the 3D model.
*
* This activity has a test feature to send messages to the server with the IP that is in
the text box. (This feature is deprecated)
*/
public class MainActivity extends AppCompatActivity implements SensorEventListener {


  ////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////
  ////////////////// Activity variables /////////////////////////
  ////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////


  //Sensor manager of the device
  private SensorManager mSensorManager;
  //Text areas where the information is shown
  TextView txtX, txtY, txtZ, txtZoom;
  //Current zoom of the model
  int n_zoom = 0;
  //"y" coordinate that is obtained by pressing the zoom button
```

```
int ZoomButtonY = -1;
//Object that sends messages to the server
private MessageSender sender;
//String where is stored the Ip inserted by the user
private String IPimput;

//State variables for the buttons
boolean auto;

////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
////////////////////// on Create ///////////////////////////////
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////

/**
 * Actions to do when the activity starts
 * @param savedInstanceState
 */

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    // Access to the sensor service
    mSensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);

    //textView initialization of the activity
    txtX = (TextView)findViewById(R.id.xVar);
    txtY = (TextView)findViewById(R.id.yVar);
    txtZ = (TextView)findViewById(R.id.zVar);
    txtZoom = (TextView)findViewById(R.id.ZoomVar);
    txtZoom.setText(String.valueOf(n_zoom+"%"));

    //Initially auto is set on false
    auto = false;
```

```
    //messageSender initialization
    IPimput = "192.168.1.39";
    sender = new MessageSender();
    String args[] = {IPimput,"7800"};
    //sender.execute(args); //Uncomment if you want to use the main activity as a test to send messages


    /**
     * AUTO button of the activity
     *
     * When pressed, if sender is connected to the server and is active, this button
     * will automatically send messages to the server with the gyroscope's information to the server.
     *
     * When pressed again, the sender will stop sending messages.
     */
    final Button AutoButton = findViewById(R.id.Auto_button);
    AutoButton.setBackgroundColor(Color.GRAY);
    AutoButton.setOnClickListener(new View.OnClickListener() {
       public void onClick(View v) {

           //Get information of the indicated IP
           IPimput = ((TextView) findViewById(R.id.IPimput)).getText().toString();

           //If auto mode is enabled it will set auto mode to false
           if(auto) {
              auto = false;
              AutoButton.setBackgroundColor(Color.GRAY);
           }
           /*
              If auto mode is false it will set it to true, stop the previous sender and start a new one
              that will send continuously messages to the server
           */
           else {
              auto = true;
              AutoButton.setBackgroundColor(Color.BLUE);
              sender.stop();
              sender = new MessageSender();
```

```
            String args[] = {IPimput,"7800"};
            sender.execute(args);
        }

    }
});

/**
 * TEST button of the activity
 *
 * When pressed, if sender is connected to the server and is active, this button
 * will stop any sender if exists and create a new one just send a test message
 */
final Button SendButton = findViewById(R.id.Send_button);
SendButton.setBackgroundColor(Color.GRAY);
SendButton.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {

        if(auto) {
            auto = false;
            AutoButton.setBackgroundColor(Color.GRAY);

        }
        sender.stop();
        IPimput = ((TextView)findViewById(R.id.IPimput)).getText().toString();
        //Send a test message to the server
        MainActivity.sendTest(IPimput);

    }
});
/**
 * USER FREE Button
 *
 * This button launches an activity to freely rotate the model by pressing one but-
ton
 */
final Button UserFreeButton = findViewById(R.id.User_free_button);
UserFreeButton.setOnClickListener(new View.OnClickListener() {
```

```
    public void onClick(View v){

        Intent intent = new Intent(MainActivity.this, UserActivityFree.class);
        IPimput = ((TextView)findViewById(R.id.IPimput)).getText().toString();
        intent.putExtra("IP", IPimput);
        sender.stop();
        startActivity(intent);

    }

});

/**
 * USER AXIS Button
 *
 * This button launches an activity to rotate the model in one of its axis
 */
final Button UserAxisButton = findViewById(R.id.User_axis_button);
UserAxisButton.setOnClickListener(new View.OnClickListener() {

    public void onClick(View v){

        Intent intent = new Intent(MainActivity.this, UserActivityAxis.class);
        IPimput = ((TextView)findViewById(R.id.IPimput)).getText().toString();
        intent.putExtra("IP", IPimput);
        sender.stop();
        startActivity(intent);

    }

});

/**
 * JSON button
 *
 * This button launches an activity just to try the JSON encoding feature of the app
```

```
     */
    final Button JSONButton = findViewById(R.id.JSON_button);
    JSONButton.setOnClickListener(new View.OnClickListener() {

        public void onClick(View v){

            Intent intent = new Intent(MainActivity.this, JSONActivity.class);
            IPimput = ((TextView)findViewById(R.id.IPimput)).getText().toString();
            intent.putExtra("IP", IPimput);
            sender.stop();
            startActivity(intent);

        }

    });

    /**
     * LR Button
     *
     * This button launches an activity that lets you choose the device you are using to
control the
     * model (Left or Right)
     */
    final Button LRButton = findViewById(R.id.LR_button);
    LRButton.setOnClickListener(new View.OnClickListener() {

        public void onClick(View v){

            Intent intent = new Intent(MainActivity.this, ChooseLRActivity.class);
            IPimput = ((TextView)findViewById(R.id.IPimput)).getText().toString();
            intent.putExtra("IP", IPimput);
            sender.stop();
            startActivity(intent);

        }

    });
```

```
}

////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
////////////////////// Sensor methods //////////////////////////
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////

/**
 * Method to start and register the sensors on the manager
 */
private void Ini_Sensores() {

    //We only need to register the gyroscope
    mSensorManager.registerListener(this,
            mSensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE),
            SensorManager.SENSOR_DELAY_FASTEST);

}

/**
 * Method to stop listening to the sensors
 */
private void Parar_Sensores() {

    mSensorManager.unregisterListener(this,
            mSensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE));

}

/**
 * Method that is triggered when the sensor's data is changed
 * @param event
 */
@Override
public void onSensorChanged(SensorEvent event) {
```

```
    synchronized (this) {
        Log.d("sensor", event.sensor.getName());

        switch (event.sensor.getType()) {

            //If the sensor that triggers this method is the gyroscope
            case Sensor.TYPE_GYROSCOPE:
                //Only if auto mode is enabled it will send the new data to the server
                if(auto)
                    //Format: gyroscope.x gyroscope.y gyroscope.z zoom
                    this.send(event.values[0] + " " + event.values[1] + " " + event.val-
ues[2]+ " " + n_zoom + "\n");

                //Update the information showed on screen
                txtX.setText(String.valueOf(event.values[0]));
                txtY.setText(String.valueOf(event.values[1]));
                txtZ.setText(String.valueOf(event.values[2]));

                break;

        }
    }
}

/**
 * Method that is triggered when the accuracy of the sensor is changed
 * @param sensor
 * @param accuracy
 */
@Override
public void onAccuracyChanged(Sensor sensor, int accuracy) {}

/**
 * Method that is triggered when the sensors are told to stop
 */
@Override
protected void onStop() {
```

```
    Parar_Sensores();

    super.onStop();
}

/**
 * Method that is triggered when the activity is finished
 */
@Override
protected void onDestroy() {
    // TODO Auto-generated method stub

    Parar_Sensores();

    super.onDestroy();
}

/**
 * Method that is triggered when the activity is paused
 */
@Override
protected void onPause() {
    // TODO Auto-generated method stub

    Parar_Sensores();

    super.onPause();
}

/**
 * Method that is triggered when the activity is restarted
 */
@Override
protected void onRestart() {
    // TODO Auto-generated method stub

    Ini_Sensores();
```

```
    super.onRestart();
}

/**
 * Method that is triggered when the activity is resumed
 */
@Override
protected void onResume() {
    super.onResume();

    Ini_Sensores();

}

//////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////
////////////////////// Sender methods ////////////////////////////
//////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////

/**
 * Method to send a message to the sender
 * @param msg
 */
public void send( String msg ){

    sender.send(msg);

}

/**
 * Method to create a sender just to send a test message to the server
 * @param ip
 */
static void sendTest(String ip){
```

```
        MessageSender sender2 = new MessageSender();


        String []args = {ip,"7800"};
        sender2.execute(args);
        sender2.sendAndTerminate("TEST MESSAGE\n");


    }
}
```

El método más interesante del código anterior es el llamado onSensorChange. Este método se activa siempre que un sensor detecta un nuevo valor. Dentro se comprueba el sensor que lo ha activado, se obtienen los valores, se muestran por pantalla y se envían al servidor gracias a la clase MessageSender.

C) Integración del servidor en Unreal Engine

Tras lograr el envío de los sensores al ordenador el siguiente paso era que estos datos pudiesen ser leídos por el motor Unreal Engine. Una primera idea fue guardar los datos recibido por el servidor de NetBeans en un archivo y que Unreal los leyese de allí. Esto fue inmediatamente descartado ya que se necesitaba un tiempo de respuesta muy bajo y tener un servidor externo a Unreal y que hubiese un paso intermedio en la recepción de datos ralentizaría mucho el proceso.

Lo más lógico fue crear una clase c++ en el motor y a aprender a usar la notación requerida para poder ser usada en un blueprint. Antes de eso se buscó información de cómo crear un servidor TCP en c++ y se realizó uno en Visual Studio gracias a el uso de Winsock, una librería de Windows para el uso de sockets TCP/IP. Una vez se probó su funcionamiento, se integró en Unreal Engine.

Tras numerosos problemas con el servidor se logró que funcionara y se mostrasen los mensajes recibidos por pantalla en la ejecución del entorno 3D. Surgió un gran problema: el servidor hacía recepción de mensajes bloqueante y no se ejecutaba un frame hasta que no se recibía un mensaje. Este problema paralizó el desarrollo del proyecto durante bastante tiempo hasta que desarrollé un servidor no bloqueante y además que fuera lanzado en otra hebra a parte del motor. Esta tarea llevó tanto tiempo porque surgieron numerosos problemas de compilación, de librerías y tardé mucho tiempo en comprender el funcionamiento de un servidor asíncrono y cómo se lanzaban hebras en Unreal Engine.

Se creó la clase serverThread que era la que se lanzaba en otra hebra para ejecutar el servidor y una clase controladorServer que es la encargada de manipular los datos recibidos por el servidor. A continuación se muestra el código de serverThread que es la que atañe más al tema de este apartado mientras que controladorServer se encarga más de la manipulación e interpretación de datos.

ServerThread.h

```
/**
This class handles the messages reception by creating a new thread where
this task will be done
*/
class SERVERTCP3_API AServerThread : public FRunnable
    {
public:
```

```
    // Sets default values for this actor's properties
    AServerThread(int port);
    ~AServerThread();

//////////////SERVER CLASS/////////////
class ServerTCP {
  private:
  int port;
  SOCKET listening;
  TArray<SOCKET> connections;
  TArray<FString> lastMessage;
  bool running;

  public:
  ServerTCP();
  bool Initialize(int port);
  void Run();
  void Stop();
  TArray<FString> getLastMessage();
};

//////////////FRUNNABLE INTERFACE////////
private:
static AServerThread* Runnable;
    ServerTCP TheServer;
    FRunnableThread* Thread;
    FThreadSafeCounter StopTaskCounter;
    int port;

public:
    virtual bool Init();
    virtual uint32 Run();
    virtual void Stop();
    void EnsureCompletion();
static void Shutdown();
    TArray<FString> getLastMessage();
    };
```

ServerThread.cpp

```
AServerThread* AServerThread::Runnable = NULL;

/*
        Constructor of the class AServerThread: it neeed the port that the server
        is going to be listening to.
        It creates the thread that runs the server
*/
    AServerThread::AServerThread(int p)
    {
        port = p;
        Thread = FRunnableThread::Create(this, TEXT("ServerTCP"), 0, TPri_BelowNormal);
        //windows default = 8mb for thread, could specify more


    }

/*
        Destructor of the class AServerThread
*/
    AServerThread::~AServerThread()
    {
        delete Thread;
        Thread = NULL;
    }

/*
        This method inititilializes the server with the port given in the constructor
*/
    bool AServerThread::Init()
    {
        return TheServer.Initialize(port);


    }

/*
        Method to start the server
*/
    uint32 AServerThread::Run()
    {
        FPlatformProcess::Sleep(0.03);
```

```
        TheServer.Run();

        return 0;
    }
/*
        Method to stop the server
*/
    void AServerThread::Stop()
    {
        StopTaskCounter.Increment();
        TheServer.Stop();
    }

/*
        This method stops the server and wait until the thread has finished
*/
    void AServerThread::EnsureCompletion()
    {
        Stop();
        Thread->WaitForCompletion();
    }

/*
        This method must be called whenever you want to finish the server.
        It stops it and make sure it finish properly
*/
    void AServerThread::Shutdown()
    {
        if (Runnable)
        {
                Runnable->EnsureCompletion();
                delete Runnable;
                Runnable = NULL;
        }
    }

/*
        It returns the las message the server received
        It return null if the server hasn't received any messages
*/
```

```
TArray<FString> AServerThread::getLastMessage()
{
   return TheServer.getLastMessage();
}

/*
   Constructor of the class ServerTCP
   Initially it's not running
*/
AServerThread::ServerTCP::ServerTCP()
{
   running = false;
}

/*
   Method to initialize the ServerTCP
   It creates all the sockets it needs with the given port
*/
bool AServerThread::ServerTCP::Initialize(int p)
{
   port = p;

   //Winsock initializing
   WSADATA wsData;
   WORD version = MAKEWORD(2, 2);
   int iniOK = WSAStartup(version, &wsData);

   if (iniOK != 0)
          return false;

   //Creation of the listening socket
   listening = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

   if (listening == INVALID_SOCKET)
          return false;

   //Add IP and port to the socket
   sockaddr_in hint;
   hint.sin_family = AF_INET;
```

```
        hint.sin_port = htons(port);
        hint.sin_addr.S_un.S_addr = INADDR_ANY;

        bind(listening, (sockaddr*)&hint, sizeof(hint));

        //Set the socket as a listening socket
        listen(listening, SOMAXCONN);

        //Set the socket to non blocking mode
        u_long iMode = 1; //Any value different from zero is non blocking
        ioctlsocket(listening, FIONBIO, &iMode);

        //Set running to true
        running = true;
        return true;
    }

/*
        This method is called after the thread is initialized and it starts
        the main loop of the server
*/
    void AServerThread::ServerTCP::Run()
    {

        //Struct with sockets
        FD_SET readfds;

        //Buffer where we receive the data
        char buffer[4096];

        //Main loop of the server
        while ( running ) {

        //Set to zero all the bits of buffer
        ZeroMemory(buffer, 4096);

        //Clean readfds
        FD_ZERO(&readfds);
```

```
//Add the listening socket to readfds
FD_SET(listening, &readfds);

//Fill readfds with the sockets
for (int i = 0; i < connections.Num(); i++) {
        FD_SET(connections[i], &readfds);
}

//Wait until there is a message
select(0, &readfds, NULL, NULL, NULL);

//If the listening socket is in readfds means there is a new connection
if (FD_ISSET(listening, &readfds))
{
        SOCKET newConnection = accept(listening, NULL, NULL);
        //If the new connection has been created succesfully
        if (newConnection != INVALID_SOCKET) {

                connections.Add(newConnection);

        }

}
//For all the connections
for (int i = 0; i < connections.Num(); i++) {

        //If the socket is in readfds means there is a new message in it
        if (FD_ISSET(connections[i], &readfds))
        {
                //Clean the buffer
                ZeroMemory(buffer, 4096);

                //Receive the message
                int retval = recv(connections[i], buffer, 4096, NULL);
//If we received something we set the attribute lastMessage to the new message
                if (retval > 0) {
```

```
                        FString msg = buffer;
                        msg.ParseIntoArrayLines(lastMessage);
                    }

                }
            }
        }
    }

/*
        Method to stop the ServerTCP
        It closes all the sockets and cleans the winsock
*/
    void AServerThread::ServerTCP::Stop()
    {
        running = false;
        closesocket(listening);
        for (int i = 0; i < connections.Num(); i++)
                closesocket(connections[i]);
        WSACleanup();
    }
/*
         Method that returns the last message the ServerTCP has received and
        set it to null afterwards
*/
    TArray<FString> AServerThread::ServerTCP::getLastMessage()
    {
        TArray<FString> msg(lastMessage);
        lastMessage.Empty();
        return msg;
    }
```

Cabe destacar que esta clase implementa la interfaz FRunnable que proporciona Unreal Engine. Esta interfaz es la encargada de lanzar hebras y los métodos más importantes son Initialize y Run. En el método Initialize de la clase se establecen e inicializan los sockets y se prepara uno para escuchar al puerto indicado. Por otro lado el método Run es el que se ejecuta en la hebra a parte y contiene el bucle principal del servidor donde se registran nuevas conexiones y se reciben mensajes.

Así, con la clase ServerThread y ControladorServer se solucionó el problema del bloqueo y del bajo framerate del espacio 3D consiguiendo altas tasas de frames por segundo, tantas como el ordenador permitía.

Además se observó que la latencia era muy buena, es decir desde que se envía un mensaje y aparecía por pantalla pasaba muy poco tiempo. Al tener estos resultados no fue necesario buscar otras métodos de envío de datos.

Los blueprints creados para inicializar el servidor e imprimir los mensajes por pantalla se muestran en las figuras 5.11 y 5.12 respectivamente:

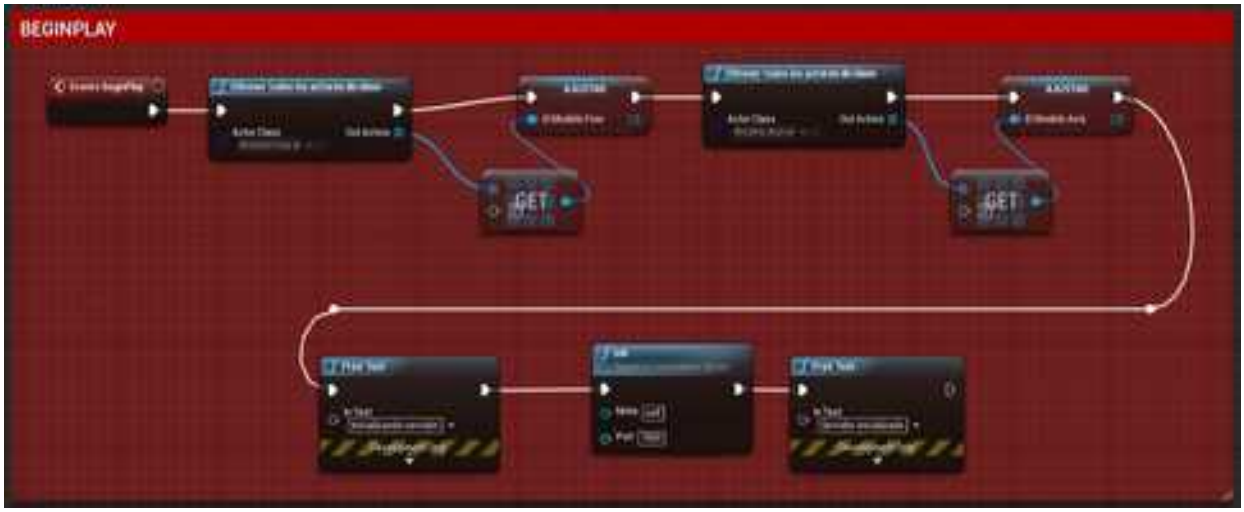

Figura 5.11: Blueprint ejecutado al principio del juego

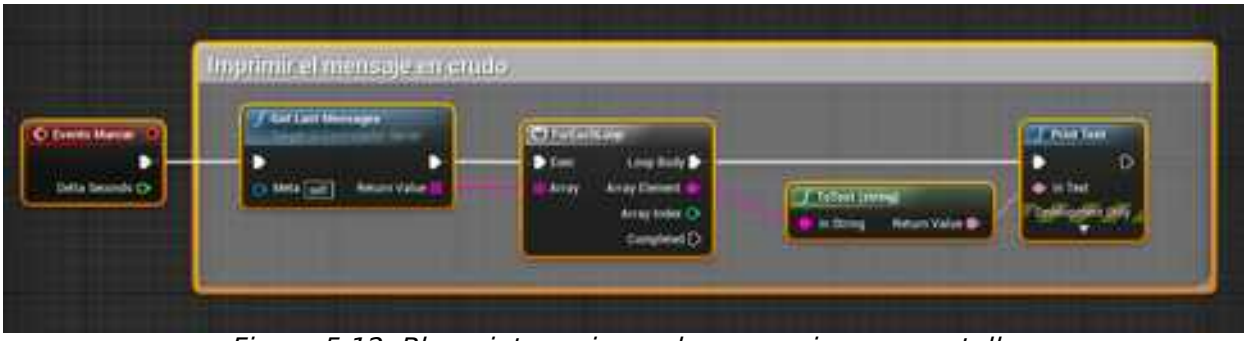

Figura 5.12: Blueprint que impre los mensajes por pantalla

Con las imágenes anteriores se puede apreciar el funcionamiento de la programación con blueprints. Algunos de los nodos usados se corresponden con funciones de la clase ControladorServer que se explicará en el siguiente apartado.

D) Primera Interacción

En esta etapa se consiguió hacer que se moviera un modelo en el entorno 3D gracias a los datos enviados por el móvil. Lo primero que se hizo fue convertir los datos recibido (en formato String) a vectores de rotación. Para ello se incluyó esa funcionalidad en la clase ControladorServer cuyo código se muestra a continuación:

AControladorServer.h

```
UCLASS()
class SERVERTCP3_API AControladorServer : public AActor
     {
     GENERATED_BODY()
public:
     // Sets default values for this actor's properties
     AControladorServer();
protected:
```

```
    void BeginPlay() override;
    void EndPlay(const EEndPlayReason::Type EndPlayReason) override;

public:
    virtual void Tick(float DeltaTime) override;
    AServerThread* theServer;


    UFUNCTION(BlueprintCallable, Category="Get")
    TArray<FString> getLastMessages();


    UFUNCTION(BlueprintCallable, Category = "Get")
    void getLastVectors(TArray<FVector> & allVecs, TArray<float> & allZooms);


    UFUNCTION(BlueprintCallable, Category = "Get")
    void getLastAxis(TArray<float> & allVal, TArray<FString> & allAxis);


    UFUNCTION(BlueprintCallable, Category = "Get")
    void getLastJSONLR(TArray<FVector> & valR, TArray<FVector> & valL, TArray<bool> &
handR, TArray<bool> & handL, bool & recibido);


    UFUNCTION(BlueprintCallable, Category = "State")
    void Init(const int port);


    UFUNCTION(BlueprintCallable, Category = "Movement")
    FRotator getNewRotation(const FVector & rads);


    UFUNCTION(BlueprintCallable, Category = "Movement")
    float getNewAxisRotation(const float & rads);
    float radsToDeg(float & rads);
};
```

AControladorServer.cpp

```
/*

       Constructor of the class AControladorServer
*/
AControladorServer::AControladorServer()
    {
    // Set this actor to call Tick() every frame.  You can turn this off to improve per-
formance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;
    }


/*

       Called when the game starts or when spawned
*/
```

```
void AControladorServer::BeginPlay()
{
   Super::BeginPlay();
}

/*
   Called when the game is finished
   It stops the server
*/
void AControladorServer::EndPlay(const EEndPlayReason::Type EndPlayReason)
{
   if (theServer) {
          theServer->Shutdown();
          delete theServer;
          theServer = nullptr
   }

   Super::EndPlay(EndPlayReason);
}
// Called every frame
void AControladorServer::Tick(float DeltaTime)
{
   Super::Tick(DeltaTime);
}
/*
   Method to get the last message the server received
   It returns an empty TArray if it hasn´t received anything
*/
TArray<FString> AControladorServer::getLastMessages()
{
   TArray<FString> msg;
   if(theServer)
          msg = theServer->getLastMessage();
   return msg;
}

/*
 Method that returns the last messages with the information of the sensors of the phone
 in a TArray of FVector and a TArray with the zoom.
 This methos is used in the "Free rotation interaction"
```

```
 The messages has to be sent with the format: <FString>: "x y z zoom\n"
 It returns a TArray<Fvector> with every "xyz" tha it's received.
 It also returns a TArray<float> with all the zooms that are received
 Each element "i" of allVecs corresponds with the same element "i" of allZooms
*/
    void AControladorServer::getLastVectors( TArray<FVector> & allVecs, TArray<float> &
allZooms)
    {
       //Declaration of all the variables
       TArray<FString> msg;
       FString ls;
       FVector *v;
       float xf, yf, zf, zoomf;
       FString xs, ys, zs,zooms, aux1, aux2;

       if (theServer) {
              //Get the last message of the server
              msg = theServer->getLastMessage();

              //If there are messages
              if(msg.Num() > 0)
                     //For each message it's transformed into a FVector
                     for (int i = 0; i < msg.Num(); i++) {
                            v = new FVector();
                            ls = msg[i];
                            //We get every value by spliting the FString
                            ls.Split(" ", &xs, &aux1);
                            aux1.Split(" ", &ys, &aux2);
                            aux2.Split(" ", &zs, &zooms);

                            xf = FCString::Atof(*xs);
                            yf = FCString::Atof(*ys);
                            zf = FCString::Atof(*zs);
                            zoomf = FCString::Atof(*zooms);

                            v->Component(0) = xf;
                            v->Component(1) = yf;
                            v->Component(2) = zf;
```

```
                        //Add the FVector and the zoom to the TArray
                        allVecs.Add(*v);
                        allZooms.Add(zoomf);
                    }
                }
            }

/*
 This method is used in the "Axis rotation interaction".
 It returns a TArray<float> with all the values of the axis Y of the sensor of the phone.
 It also returns a TArray<Fstring> with the axis

 The messages have to be sent in the format: <FString>: "Y Axis"

 Axis needs to be one of these values: "X","Y","Z","NONE"

 Each element "i" of allVal corresponds with the same element "i" of allAxis

*/
    void AControladorServer::getLastAxis(TArray<float> & allVal, TArray<FString> & allAxis)
    {
       //Declaration of the variables
       TArray<FString> msg;
       FString ls;
       float yf;
       FString Axiss, ys;

       if (theServer) {

            //Get the last message of the server
            msg = theServer->getLastMessage();

            //If we received messages
            if (msg.Num() > 0)
                //Each message it's added to the right TArray
                for (int i = 0; i < msg.Num(); i++) {
```

```
                        ls = msg[i];
                        ls.Split(" ", &Axiss, &ys);
                        yf = FCString::Atof(*ys);
                        allVal.Add(yf);
                        allAxis.Add(Axiss);
                    }
                }
            }

/*
 This method is used in the "Full interaction method".
 It returns:

 valR: TArray<FVector> where all the "xyz" values sent by the right device are stored
 valL: TArray<FVector> where all the "xyz" values sent by the left device are stored
 handR: TArray<bool> where each element "i" is true if there is a message in the element "i" of valR
 handL: TArray<bool> where each element "i" is true if there is a message in the element "i" of valL
 recibido: bool that is true if there is at least one message

 The messages needs to be sent in JSON format in the next way:
 {
   "getL":
   {
     "hand":"L",
     "x":"value_x_L",
     "y":"value_y_L",
     "z":"value_z_L"
   },
   "getR":
   {
     "hand":"R",
     "x":"value_x_R",
     "y":"value_y_R",
     "z":"value_z_R"
   }
```

```
}

*/
    void AControladorServer::getLastJSONLR(TArray<FVector>& valR, TArray<FVector>& valL,
TArray<bool> & handR, TArray<bool> & handL, bool & recibido)
    {
       TArray<FString> msgs;
       recibido = false;
       if (theServer) {
              msgs = theServer->getLastMessage();
              //If there are messages
              if (msgs.Num() > 0) {
                     //For each message
                     for (int i = 0; i < msgs.Num(); i++) {

                            FString msg = msgs[i];
                            TSharedPtr<FJsonObject> JsonObject;
              TSharedRef< TJsonReader<> > Reader = TJsonReaderFactory<>::Create(msg);

                            //Create the JSONObject from the received String
                            if (FJsonSerializer::Deserialize(Reader, JsonObject)) {

                            //Create the JSON subobjects for each device
                            auto JsonR = JsonObject->GetObjectField("getR");
                            auto JsonL = JsonObject->GetObjectField("getL");

                            //Vectors with the values from the messages
                            FVector *vecR = new FVector();
                            FVector *vecL = new FVector();

        //For each device, if it exists, we get the data and add it to the vectors
                            if (JsonR->GetStringField("hand").Equals("R")) {

                            vecR->Component(0) = JsonR->GetNumberField("x");
                            vecR->Component(1) = JsonR->GetNumberField("y");
                            vecR->Component(2) = JsonR->GetNumberField("z");

                             handR.Add(true);
```

```
                        }
                        else
    //It's importan to add values to the vectors even though we haven't receive data from a device so that we assort the indices when we return the vectors
                        handR.Add(false);

                        valR.Add(*vecR);

                        if (JsonL->GetStringField("hand").Equals("L")) {
                        vecL->Component(0) = JsonL->GetNumberField("x");
                        vecL->Component(1) = JsonL->GetNumberField("y");
                        vecL->Component(2) = JsonL->GetNumberField("z");
                        handL.Add(true);
                        }
                        else
                        handL.Add(false);

                        valL.Add(*vecL);

                        recibido = true;

                        }
                    }
                }
            }
        }

/*
    Method to init the server listening to the given port
*/
    void AControladorServer::Init(const int port)
    {
        theServer = new AServerThread(port);
    }

/*
    Method to transform an FVector of rotations into an FRotator
    Used in "Free rotation interaction"
*/
```

```
FRotator AControladorServer::getNewRotation(const FVector & rads)
{
	//Components of rads
	float rads_x = rads.Component(0);
	float rads_y = rads.Component(1);
	float rads_z = rads.Component(2);
	//Transformation to the new rotation
	FRotator rot_out;
	rot_out.Roll = radsToDeg(rads_x);
	rot_out.Pitch = -radsToDeg(rads_y);//Rotations on the device are not the same in unreal so we need to change some values
	rot_out.Yaw = -radsToDeg(rads_z);
	return rot_out;
}
/*
	Method that receives a value of radiasn/second and returns the rotation to apply to the model
	Used in "Axis rotation interaction"
*/
float AControladorServer::getNewAxisRotation(const float & rads)
{
	float r = rads;
	return radsToDeg(r);
}
/*
	This method interpolates a values of radians/second into a degree of rotation to the model
*/
float AControladorServer::radsToDeg(float & rads)
{
	//Variable to check if randias are negative
	bool negativo = false;
	if (rads < 0) {
		negativo = true;
		rads *= -1;	//We set the radians to possitive for its use in the inter-polation
	}
	//Degrees
	float deg = 0;
	//Movement filter
	if (rads > 0.5) {
```

```
                if (rads < 1)
                        deg = 1;
                else if (rads >= 1 && rads <= 4)
                        deg = 1.5 * rads - 0.5; //Interpolation function for 1 <= x <= 4
                else if (rads > 4)
                        deg = rads + 4;          //Interpolation function for 4 < x
        }
        if (negativo)           //If we received negative radians we change back the de-
grees to negative
                deg *= -1;
        return deg;
    }
```

En este código se muestran los métodos para la gestión del servidor y además funcionalidades de las siguientes versiones pero para este apartado la función que interesa es getLastVectors que transformaba el paquete de mensajes recibido en vectores de tres elementos que contenían los valores del giroscopio que se habían recibido.

Una vez decodificados estos valores se procedió a pasárselos en crudo a la rotación del modelo para ver cómo se comportaba. El resultado fue que el modelo estaba continuamente temblando debido a el ruido de los sensores y que por supuesto, no había ninguna traducción entre radianes/segundo y grados de rotación. El blueprint de esta versión se muestra en la figura 5.13.

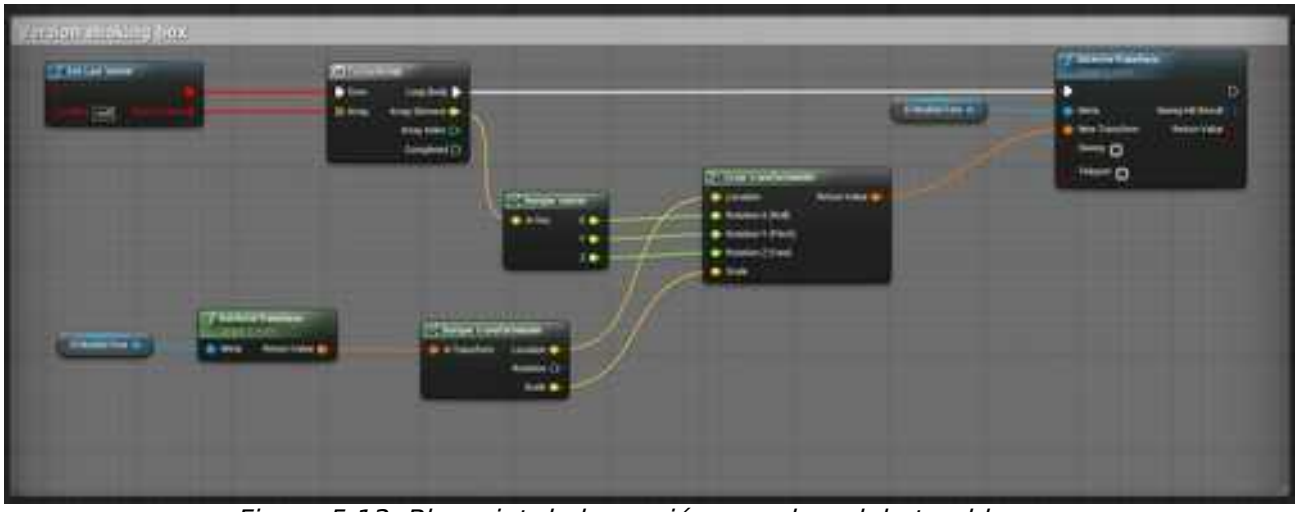

Figura 5.13: Blueprint de la versión con el modelo tembloroso

Para resolver este problema se creó la función getNewRotation que transformaba estos vectores en rotaciones válidas para el modelo. Además se dio uso al nodo AddActorLocalWorldRotation que permitía rotar el modelo de manera muy suave y solucionaba los problemas que ocasionaba el nodo setActorTransform. Además me per-

caté de que se perdían mensajes porque se recibían en paquetes de varios y solo se interpretaba el primero.

Para solucionar eso se creó una cola FIFO en el blueprint en la que almacenaba los mensajes, si no se recibe ninguno nuevo se lee de esa cola hasta que se quede vacía. Si llegan mensajes mientras la cola tiene elementos, se limpia la cola y se actualiza con los nuevos mensajes.

Por otro lado se incluyó en el envío de mensajes un valor de zoom y ahora la función getLastVectors devolvía dos vectores, uno con las rotaciones y otro con los zooms. Cada elemento del vector de rotaciones corresponde al elemento de zooms con el mismo índice. El blueprint de esta versión se muestra a continuación en la figura 5.14:

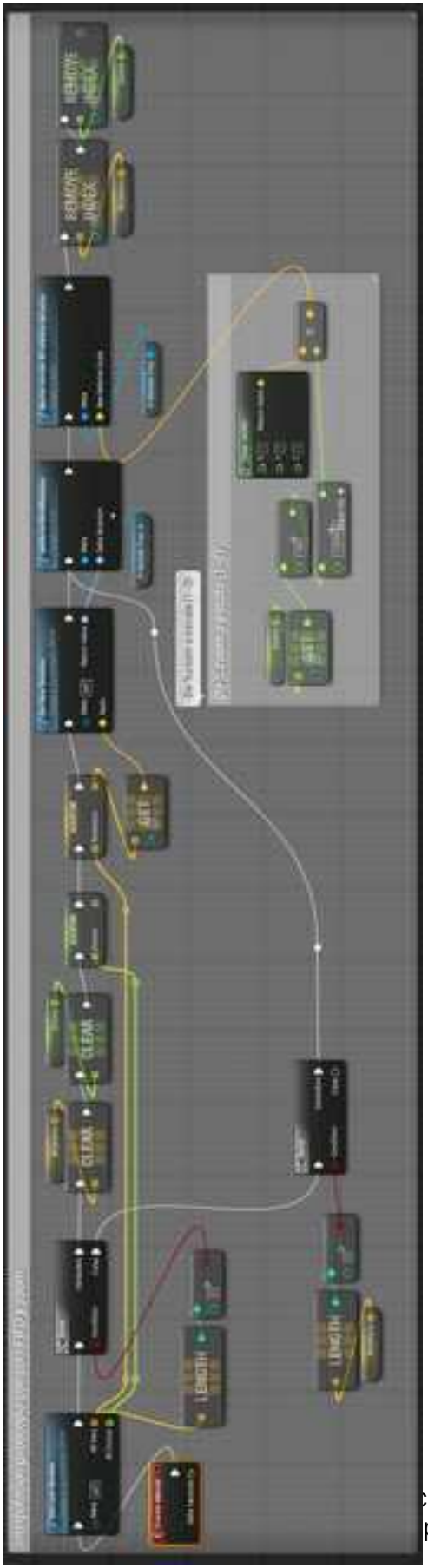

Figura 5.14: Blueprint de interacción suave y

se creó una actividad con el objetivo de ser más
cia se muestra en la figura 5.15 y consiste básica-
pulsado envía mensajes al ordenador y un slider

para cambiar el valor de zoom.

Figura 5.15: Interfaz para la interacción con rotación libre y zoom

El código de la actividad para este método se presenta a continuación:

UserActivityFree.java

```
public class UserActivityFree extends AppCompatActivity implements SensorEventListener {
```

```
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
////////////////// Activity variables /////////////////////////
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////

//IP received from the main activity
private String IP;
//Current zoom of the object
private int n_zoom;
//Textview where the current zoom is showed
TextView txtZoom;
//Sensor manager of the device
private SensorManager mSensorManager;
//Object that sends messages to the server
private MessageSender sender;
//State variable that controls whether information can be sent or not
private boolean sendingRotation;

/**
 * Actions to do when the activity starts
 * @param savedInstanceState
 */
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_user_free);

    // Access to the sensor service
    mSensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);

    //messageSender initialization
    Intent intent = getIntent();
    this.IP = intent.getStringExtra("IP");
    sender = new MessageSender();
    final String args[] = {IP,"7800"};
    sender.execute(args);
```

```
//Initialization of state and control variables
txtZoom = (TextView)findViewById(R.id.ZoomText);
n_zoom = 0;
sendingRotation = false;

/**
 * Zoom seekbar. It controls the zoom of the object by dragging the button of
 * the bar-
 */
final SeekBar ZoomBar = findViewById(R.id.ZoomBar);
ZoomBar.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener(){

    @Override
    public void onProgressChanged(SeekBar seekBar, int i, boolean b) {
        n_zoom = i;
        txtZoom.setText(String.valueOf(i)+"%");

    }

    @Override
    public void onStartTrackingTouch(SeekBar seekBar) {

    }

    @Override
    public void onStopTrackingTouch(SeekBar seekBar) {

    }
});

/**
 * ROTATION Button
 * Whenever this button is pressed and hold it will send information to the server
 */
final ImageButton RotationButton = findViewById(R.id.RotationButton);
RotationButton.setOnTouchListener(new View.OnTouchListener() {
    @Override
```

```
        public boolean onTouch(View v, MotionEvent event) {
            switch(event.getAction()) {
                case MotionEvent.ACTION_DOWN:
                    // PRESSED
                    sendingRotation = true;
                    return true; // if you want to handle the touch event
                case MotionEvent.ACTION_UP:
                    // RELEASED
                    sendingRotation = false;
                    return true; // if you want to handle the touch event
            }
            return true;
        }

    });

}

//////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////
//////////////////// Sensor methods //////////////////////////////
//////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////

/**
 * Method to start and register the sensors on the manager
 */
protected void Ini_Sensores() {

    //We only need to register the gyroscope
    mSensorManager.registerListener(this,
            mSensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE),
            SensorManager.SENSOR_DELAY_FASTEST);

}

/**
 * Method to stop listening to the sensors
```

```
     */
    private void Parar_Sensores() {

        mSensorManager.unregisterListener(this,
                mSensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE));

    }

    /**
     * Method that is triggered when the sensor's data is changed
     * @param event
     */
    @Override
    public void onSensorChanged(SensorEvent event) {
        synchronized (this) {
            Log.d("sensor", event.sensor.getName());

            switch (event.sensor.getType()) {

                case Sensor.TYPE_GYROSCOPE:
                    if(sendingRotation)
                        //Formato: Giroscopio.x Giroscopio.y Giroscopio.z Zoom
                        this.send(event.values[0] + " " + event.values[1] + " " + event.val-
ues[2]+ " " + n_zoom + "\n");

                    break;

            }
        }
    }

    /**
     * Method that is triggered when the accuracy of the sensor is changed
     * @param sensor
     * @param accuracy
     */
    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {
```

```
}

/**
 * Method that is triggered when the sensors are told to stop
 */
@Override
protected void onStop() {

    Parar_Sensores();

    super.onStop();
}

/**
 * Method that is triggered when the activity is finished
 */
@Override
protected void onDestroy() {
    // TODO Auto-generated method stub

    Parar_Sensores();

    super.onDestroy();
}

/**
 * Method that is triggered when the activity is paused
 */
@Override
protected void onPause() {
    // TODO Auto-generated method stub

    Parar_Sensores();

    super.onPause();
}
```

```
/**
 * Method that is triggered when the activity is restarted
 */
@Override
protected void onRestart() {
    // TODO Auto-generated method stub

    Ini_Sensores();

    super.onRestart();
}

/**
 * Method that is triggered when the activity is resumed
 */
@Override
protected void onResume() {
    super.onResume();

    Ini_Sensores();

}

//////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////
////////////////////// Sender methods ////////////////////////////
//////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////

/**
 * Method to send a message to the sender
 * @param msg
 */
public void send( String msg ){

    sender.send(msg);
```

```
  }
}
```

Una vez más, el método más importante es onSensorChange que envía los datos al ordenador siempre que esté pulsado el botón. Al pulsar el botón se activa una variable que permite enviar los mensajes.

Con esta versión se logró una interacción con un móvil bastante buena ya que permitía las transformaciones canónicas de rotación y escalado de manera suave y ergonómica.

E) Interacción sobre ejes aislados

Tras el éxito del anterior método se planteó la posibilidad de interactuar con el modelo eligiendo el eje de rotación y aplicándole rotaciones. Para ello se creó un nuevo tipo de modelo con tres anillos, cada uno perpendicular a un eje de coordenadas(figura 5.16)

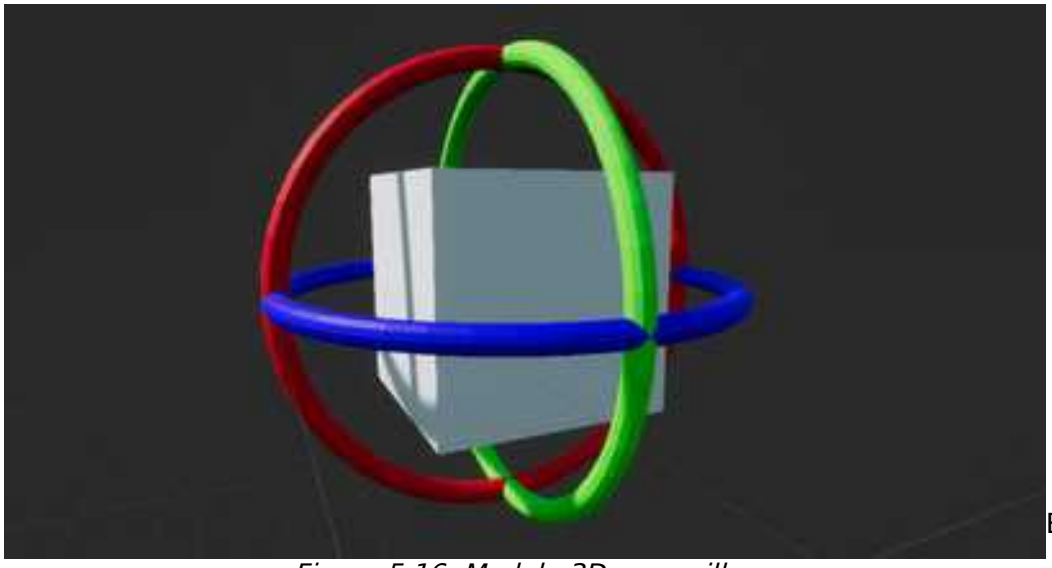

Figura 5.16: Modelo 3D con anillos

El motivo de este modelo es que se pueda ver en cada momento sobre qué eje se está rotando. Así, cuando llega información para un eje, muestra el eje correspondiente y oculta el resto. El blueprint que consigue esto se muestra a continuación en la figura 5.17:

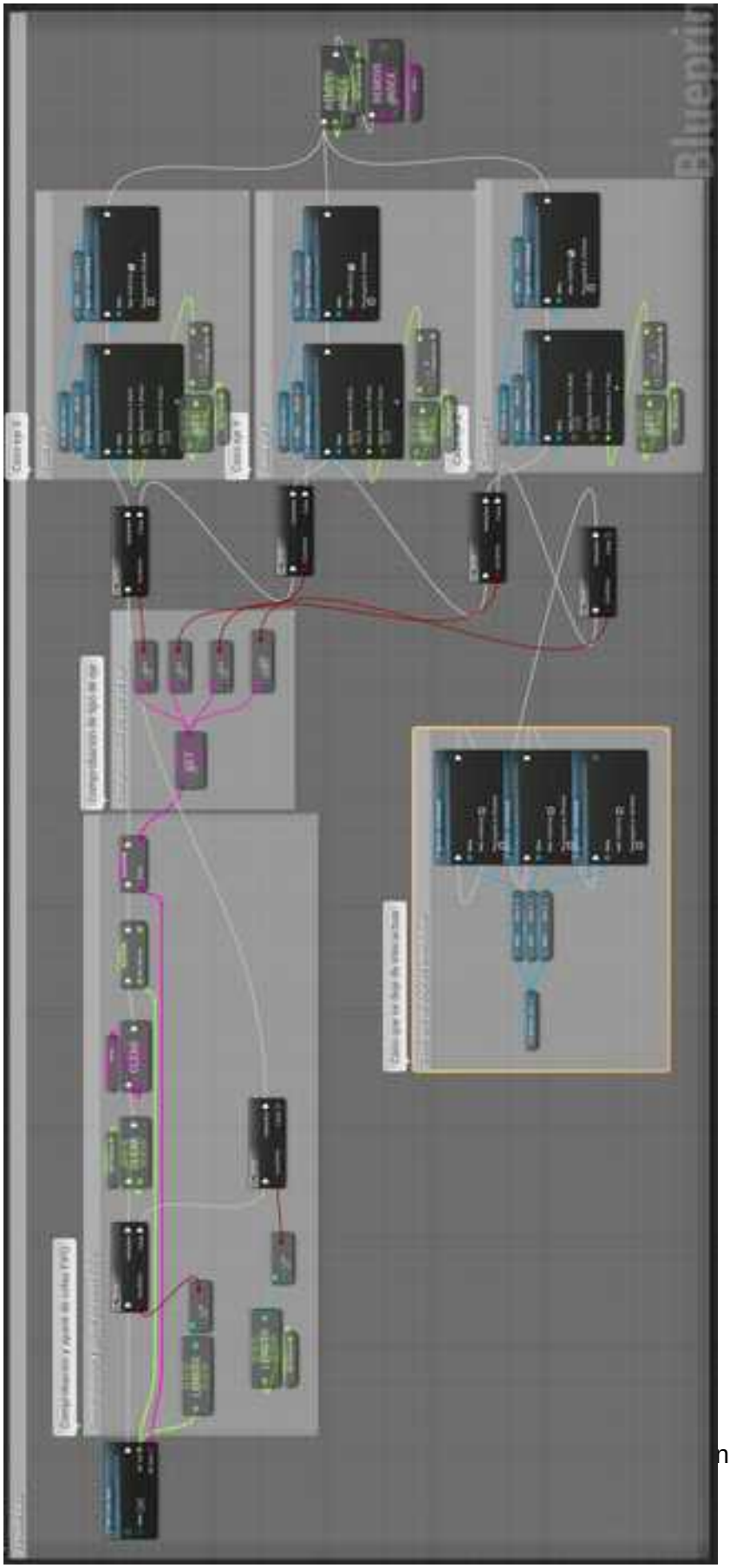

Figura 5.17

n su interfaz se muestra

UserActivityFree.java

```
/**
* This class will control the rotation of the object.
* It has 3 button, one for each axis. Whenever a button is pressed and hold
* it will send the pressed axis and the information of the gyroscope in the y
* axis.
*/
public class UserActivityAxis extends AppCompatActivity implements SensorEventListener {


  ////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////
  ////////////////// Activity variables /////////////////////////
  ////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////


  //IP received from the main activity
  private String IP;
  //Sensor manager of the device
  private SensorManager mSensorManager;
  //Object that sends messages to the server
  private MessageSender sender;
  //String where the pressed axis is stored ("X","Y","Z","NONE")
  private String sendingAxis;


  /**
   * Actions to do when the activity starts
   * @param savedInstanceState
   */
  @Override
  protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_user_axis);


      // Access to the sensor service
      mSensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);


      //messageSender initialization
      Intent intent = getIntent();
      this.IP = intent.getStringExtra("IP");
```

```
        sender = new MessageSender();
        final String args[] = {IP,"7800"};
        sender.execute(args);

        //The default axis is NONE. It will change to an actual axis when a button is
pressed
        // it will go back to NONE when the button is released.
        sendingAxis="NONE";

        /*
           AXIS BUTTONS
           When a button is pressed and hold it will change the current axis and
           information of the gyroscope will be send to the server with the axis.
         */

        final Button XAxisButton = findViewById(R.id.X_axis_button);
        XAxisButton.setOnTouchListener(new View.OnTouchListener() {
           @Override
           public boolean onTouch(View v, MotionEvent event) {
              switch(event.getAction()) {
                 case MotionEvent.ACTION_DOWN:
                    // PRESSED
                    sendingAxis = "X";
                    return true; // if you want to handle the touch event
                 case MotionEvent.ACTION_UP:
                    // RELEASED
                    sendingAxis = "NONE";
                    return true; // if you want to handle the touch event
              }
              return true;
           }

        });

        final Button YAxisButton = findViewById(R.id.Y_axis_button);
        YAxisButton.setOnTouchListener(new View.OnTouchListener() {
           @Override
           public boolean onTouch(View v, MotionEvent event) {
```

```
            switch(event.getAction()) {
                case MotionEvent.ACTION_DOWN:
                    // PRESSED
                    sendingAxis = "Y";
                    return true; // if you want to handle the touch event
                case MotionEvent.ACTION_UP:
                    // RELEASED
                    sendingAxis = "NONE";
                    return true; // if you want to handle the touch event
            }
            return true;
        }

    });

    final Button ZAxisButton = findViewById(R.id.Z_axis_button);
    ZAxisButton.setOnTouchListener(new View.OnTouchListener() {
        @Override
        public boolean onTouch(View v, MotionEvent event) {
            switch(event.getAction()) {
                case MotionEvent.ACTION_DOWN:
                    // PRESSED
                    sendingAxis = "Z";
                    return true; // if you want to handle the touch event
                case MotionEvent.ACTION_UP:
                    // RELEASED
                    sendingAxis = "NONE";
                    return true; // if you want to handle the touch event
            }
            return true;
        }

    });

}

//////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////
```

```
////////////////////// Sensor methods /////////////////////////////
////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////

/**
 * Method to start and register the sensors on the manager
 */
protected void Ini_Sensores() {

    //We only need to register the gyroscope
    mSensorManager.registerListener(this,
            mSensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE),
            SensorManager.SENSOR_DELAY_FASTEST);

}

/**
 * Method to stop listening to the sensors
 */
private void Parar_Sensores() {

    mSensorManager.unregisterListener(this,
            mSensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE));

}

/**
 * Method that is triggered when the sensor's data is changed
 * It will always send the data even if a button is not pressed because
 * the server needs to know when the buttons are not pressed. So, when
 * no button is pressed it will send NONE + Gyroscope.y
 * @param event
 */
@Override
public void onSensorChanged(SensorEvent event) {
    synchronized (this) {
        Log.d("sensor", event.sensor.getName());
```

```
        switch (event.sensor.getType()) {

            case Sensor.TYPE_GYROSCOPE:

                //Format: Axis Gyroscope.y
                this.send(sendingAxis + " " + event.values[1] + "\n");

                break;

        }
    }
}

/**
 * Method that is triggered when the accuracy of the sensor is changed
 * @param sensor
 * @param accuracy
 */
@Override
public void onAccuracyChanged(Sensor sensor, int accuracy) {

}

/**
 * Method that is triggered when the sensors are told to stop
 */
@Override
protected void onStop() {

    Parar_Sensores();

    super.onStop();
}

/**
 * Method that is triggered when the activity is finished
 */
```

```
@Override
protected void onDestroy() {
    // TODO Auto-generated method stub

    Parar_Sensores();

    super.onDestroy();
}

/**
 * Method that is triggered when the activity is paused
 */
@Override
protected void onPause() {
    // TODO Auto-generated method stub

    Parar_Sensores();

    super.onPause();
}

/**
 * Method that is triggered when the activity is restarted
 */
@Override
protected void onRestart() {
    // TODO Auto-generated method stub

    Ini_Sensores();

    super.onRestart();
}

/**
 * Method that is triggered when the activity is resumed
 */
@Override
```

```
    protected void onResume() {
        super.onResume();

        Ini_Sensores();

    }

    ////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////
    ////////////////////// Sender methods //////////////////////////
    ////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////

    /**
     * Method to send a message to the sender
     * @param msg
     */
    public void send( String msg ){

        sender.send(msg);

    }

}
```

En esta actividad el método onSensorChange envía continuamente los valores del giroscopio en el eje “Y” junto al eje seleccionado. El eje seleccionado se actualiza al pulsar uno de los tres botones de la interfaz. El motivo de que se envíen mensajes constantemente es que en este método se necesita saber también cuando no se está pulsando los botones para ocultar los anillos del modelo.

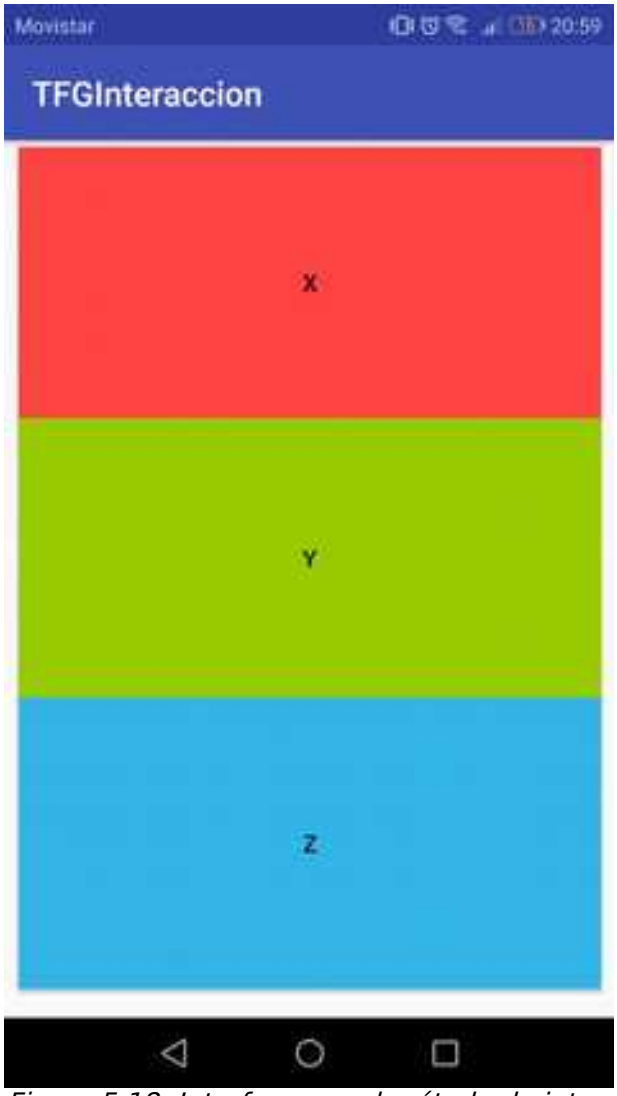

Figura 5.18: Interfaz para el método de interacción sobre ejes aislados.

F) Empaquetamiento de mensajes en JSON

Esta fase se realizó en paralelo junto a la siguiente. Su objetivo era codificar los mensajes en JSON para su envío ya que antes se estaban mandando en formato de String y las funciones para extraer los datos de ahí eran innecesariamente complejas y dependían mucho de la cantidad de datos que tenía el mensaje. Al enviar mensajes en JSON se facilitaba la exportación del sistema y le daba un acabado más profesional.

Para ello en la aplicación Android se crearon unas funciones que permiten empaquetar los datos del móvil dependiendo de si son de un móvil aislado o los dos en conjunto. Se usó la librería org.json y el código de las funciones es el siguiente:

JSONUtilities.java

```
/**
* Class that contains all the methods I use to store the information in JSON Objects
*/

public class JSONUtilities {


  // Funcion que crea un objeto JSON formado por un array formado por los datos de ambos moviles


  /**
   * Functions that create the main JSON Object by putting togheter the information of both devices
   * @param L JSON Object with the information of the L device
   * @param R JSON Object with the information of the R device
   * @return JSON Object that contains both L and R JSON objects in this format
   *
   *       {
   *           "getL": L,
   *           "getR": R
   *       }
   */
  public static JSONObject toJSONObjectEnvolvente(JSONObject L, JSONObject R){


     JSONObject ret = new JSONObject();


     try {
        ret.put("getL",L);
        ret.put("getR",R);
     } catch (JSONException e) {
        e.printStackTrace();
     }


     return ret;
```

```
  }
  /**
   * Function that create a JSON Object with the data of a device
   * @param hand Wether if the device is L or R
   * @param x value of x axis  of the sensor
   * @param y value of y axis  of the sensor
   * @param z value of z axis  of the sensor
   * @return  JSONObject that contains all the information needed of a device with the
next format:
   *       {
                "hand":"L",
                "x":"value_x_L",
                "y":"value_y_L",
                "z":"value_z_L"
            }
   */
  public static JSONObject toJSONOBjectMovil(String hand, String x, String y, String z ){
     JSONObject ret = new JSONObject();
     try {
        ret.put("hand", hand);
        ret.put("x",    x);
        ret.put("y",    y);
        ret.put("z",    z);
     } catch (JSONException e) {
        e.printStackTrace();
     }
     return ret;
  }
}
```

Para probar la funcionalidad del empaquetado en JSON se creó una actividad para la aplicación de Android que hacía uso de estas funciones y mandaba un mensaje JSON con determinados valores para poder ser decodificado en el ordenador. El código de la actividad es el siguiente:

JSONActivity.java

```
/**
* This class is only used to try my own JSON methods and send messages to the server
```

```
*/
public class JSONActivity extends AppCompatActivity {

  ////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////
  ////////////////// Activity variables /////////////////////////
  ////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////

  //Sensor manager of the device
  private MessageSender sender;
  //IP received from the main activity
  private String IP;
  //JSON Object where all the information will be stored
  private JSONObject msg_json;

  /**
   * Actions to do when the activity starts
   * @param savedInstanceState
   */
  @Override
  protected void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     setContentView(R.layout.activity_json);

     //messageSender initialization
     Intent intent = getIntent();
     this.IP = intent.getStringExtra("IP");
     sender = new MessageSender();
     final String args[] = {IP,"7800"};
     sender.execute(args);

     /**
      * SEND Button
      * When pressed it will send the JSON object in its String form
      */
     final Button SendJSONButton = findViewById(R.id.Send_JSON_button);
     SendJSONButton.setOnClickListener(new View.OnClickListener() {
```

```
        public void onClick(View v){

            send(msg_json.toString());

        }

    });

    /*
        Example of using the JSON methods with random values.
        The way I store the information in the object is the next one:

        {
            "getL":
                {
                    "hand":"L",
                    "x":"value_x_L",
                    "y":"value_y_L",
                    "z":"value_z_L"
                },
             "getR":
                {
                    "hand":"R",
                    "x":"value_x_R",
                    "y":"value_y_R",
                    "z":"value_z_R"
                }

         }
     */
    JSONObject movilL = JSONUtilities.toJSONObjectMovil("L", String.valueOf(1.1),String.-
valueOf(1.2),String.valueOf(1.3));
    JSONObject movilR = JSONUtilities.toJSONObjectMovil("R", String.valueOf(2.1),String.-
valueOf(2.2),String.valueOf(2.3));
    msg_json = JSONUtilities.toJSONObjectEnvolvente(movilL,movilR);

  }
```

```
  ////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////
  ////////////////////// Sender methods //////////////////////////
  ////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////

  /**
   * Method to send a message to the sender
   * @param msg
   */
  public void send( String msg ){

     sender.send(msg);

  }
}
```

Los datos de ambos móviles se mandan en un objeto JSON envolvente que contiene dos campos que corresponden a otros dos objetos JSON que almacenan los datos de cada móvil. El formato es el siguiente:

```
{
 "getL":
    {
       "hand":"L",
       "x":"value_x_L",
       "y":"value_y_L",
       "z":"value_z_L"
```

```
    },
  "getR":
    {
       "hand":"R",
       "x":"value_x_R",
       "y":"value_y_R",
       "z":"value_z_R"
    }

}
```

El mensaje JSON era mandado en String y era recibido por el servidor de manera correcta. El siguiente paso fue transformar el String recibido en JSON otra vez. Para ello se tuvo algunos problemas con el gestor de paquetes de Visual Studio y finalmente se optó por usar la librería que aporta Unreal Engine para la gestión de JSON que se llama FJsonObject.

Una vez lograda la decodificación de los objetos JSON se podía extraer sus valores y ser aplicados al modelo de la misma manera que se ha visto en los apartados anteriores. El código de esta funcionalidad se ha visto en el subapartado D) en el código de AControladorServer.cpp en la función getLastJSONLR. Esta función recibe un mensaje del servidor y si se ha recibido correctamente se procede a extraer primero los objetos JSON del mensaje pertenecientes a cada móvil. Para cada objeto se extraen otra vez los valores de los sensores y se añaden en los vectores correspondientes. La función devuelve cuatro vectores y un booleano; dos de los vectores contienen información de los sensores de cada móvil y los otros dos son vectores de booleanos que indican si los vectores de datos contienen datos en ese índice. El valor booleano devuelve si se han recibido mensajes en general.

El blueprint que mostraba por pantalla los valores resultantes de la extracción de datos de los objetos JSON se muestra en la figura 5.19:

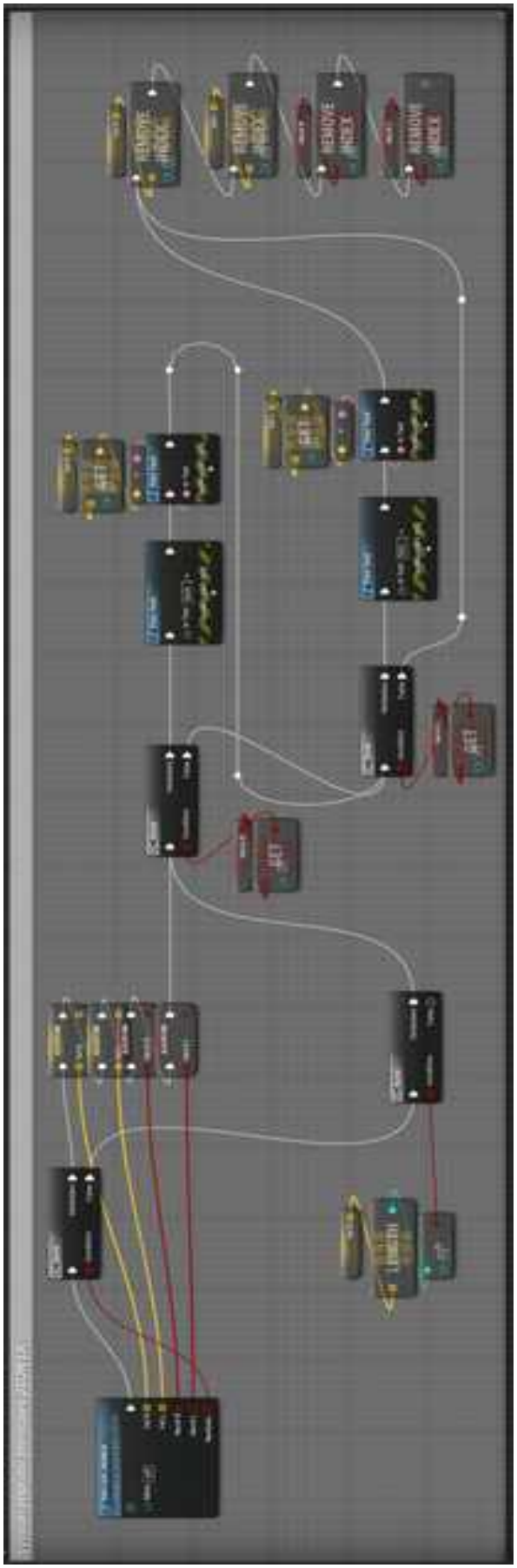

Figura 5.19: BP para imprimier los mensajes en JSON

G) Interacción completa

Este es el último y más completo método de interacción que se ha creado. En él se incluyen dos dispositivos móviles y su funcionamiento ya se ha explicado anteriormente. Si bien la interfaz es la misma que en el método de rotación libre ahora se gestionan los datos internamente de otra manera y, hay más envío de mensajes.

Al escoger este método de interacción aparece una interfaz donde se permite elegir si el dispositivo va a ser el de la derecha o el de la izquierda y se introduce la IP del dispositivo que va a recibir los mensajes. Su código es el siguiente:

ChooseLRActivity.java

```
/**
* This class lets you choose if the device is going to ve the Left device or Right de-
vice.
* The left device will send data to the right device while the right device will receive
data
* from the left device and send data to the server.
*
* Way to use this activity:
*  - It needs the IP of the server which will be indicated in the main activity
*  - If it's the left device you need to write the local IP of the right device and then
press left button (the local IP of the device itself is shown on the screen)
*  - If it's the right device you just need to press right button
*/
public class ChooseLRActivity extends AppCompatActivity {


  ////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////
  ////////////////// Activity variables /////////////////////////
  ////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////


  //IP received from the main activity
  private String IP;
  //Local IP of the device itself
  private String IP_privada;
  //Local IP of the device of the intermediary
  private String IP_inter;


  /**
   * Actions to do when the activity starts
```

```
     * @param savedInstanceState
     */

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_choose_lr);

        //Get the server IP to use it in the next activity
        Intent intent = getIntent();
        this.IP = intent.getStringExtra("IP");

        //Get the local IP of the phone
        WifiManager wm = (WifiManager) getApplicationContext().getSystemService(WIFI_SERVICE);
        this.IP_privada = Formatter.formatIpAddress(wm.getConnectionInfo().getIpAddress());

        /**
         * L Button
         * It will launch the LRActivity with an extra String called hand set to "L"
         * and the IP of the intermediary if needed
         */
        final Button LButton = findViewById(R.id.L_button);
        LButton.setOnClickListener(new View.OnClickListener() {

            public void onClick(View v){

                Intent intent = new Intent(ChooseLRActivity.this, UserActivityLR.class);

                intent.putExtra("IP", IP);
                intent.putExtra("hand", "L");
                IP_inter = ((TextView) findViewById(R.id.IP_input_inter)).getText().toString();
                intent.putExtra("IP_inter", IP_inter);
                startActivity(intent);

            }
```

```
    });

    /**
     * R Button
     * It will launch the LRActivity with an extra String called hand set to "R"
     * and the IP of the intermediary if needed
     */

    final Button RButton = findViewById(R.id.R_button);
    RButton.setOnClickListener(new View.OnClickListener() {

        public void onClick(View v){

            Intent intent = new Intent(ChooseLRActivity.this, UserActivityLR.class);

            intent.putExtra("IP", IP);
            intent.putExtra("hand", "R");
            IP_inter         =         ((TextView)         findViewById(R.id.IP_input_in-
ter)).getText().toString();
            intent.putExtra("IP_inter", IP_inter);
            startActivity(intent);

        }

    });

    //Show on the text view the local IP of the phone
    final TextView IPPrivadaVar = findViewById(R.id.IP_privada_textView_var);
    IPPrivadaVar.setText(IP_privada);


  }
}
```

Una vez configurado la interfaz anterior, se lanza una nueva actividad encargada de enviar o recibir datos dependiendo de si es el móvil de la izquierda o de la derecha. Su código es el siguiente:

UserActivityLR.java

```
public class UserActivityLR extends AppCompatActivity implements SensorEventListener {


  ////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////
  ////////////////// Activity variables /////////////////////////
  ////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////


  //IP of the server
  private String IP_server;
  //IP of the intermediary
  private String IP_inter;
  //Kind of device
  private String hand;
  //Sensor manager of the device
  private SensorManager mSensorManager;
  //Object that sends messages to the server
  private MessageSender sender;
  //Object that receives messeges from the other device (only used when it's the interme-
diary)
  private MessageReceiver receiver;
  //State variable to control if information is sended
  private boolean sendingRotation;


  /**
   * Actions to do when the activity starts
   * @param savedInstanceState
   */
  @Override
  protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_user_lr);


      // Access to the sensor service
      mSensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
```

```
        //Get the hand (L or R)
        Intent intent = getIntent();
        hand = intent.getStringExtra("hand");

        //Message sender initialization
        this.IP_server = intent.getStringExtra("IP");
        this.IP_inter = intent.getStringExtra("IP_inter");
        sender = new MessageSender();

        //Create the sender wheter if it's going to send to the server or the intermediary
        String args[] = {IP_inter,"6800"}; //Intermediary
        if( hand.equals("R")) {            //server
            args[0] = IP_server;
            args[1] = "7800";
        }

        //Checking version to see if thread pool is available and execute the sender on the
pool
        // Thread pool is neede because an AsyncTask can only be run one per execution
        if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB)
            sender.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR,args);
        else
            sender.execute(args);


        //Creation and initialization of the receiver if the device is the intermediary (R)
        if(hand.equals("R")) {
            receiver = new MessageReceiver();

            if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB)
                receiver.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR,null);
            else
                receiver.execute();
        }

        //Initially it can't send the rotation until the button is pressed
        sendingRotation = false;
```

```
    /**
     * ROTATION Button
     * When pressed it will send the rotation to the right device
     */
    final ImageButton RotationButton = findViewById(R.id.LRRotationButton);
    RotationButton.setOnTouchListener(new View.OnTouchListener() {
        @Override
        public boolean onTouch(View v, MotionEvent event) {
            switch(event.getAction()) {
                case MotionEvent.ACTION_DOWN:
                    // PRESSED
                    sendingRotation = true;
                    return true; // if you want to handle the touch event
                case MotionEvent.ACTION_UP:
                    // RELEASED
                    sendingRotation = false;
                    return true; // if you want to handle the touch event
            }
            return true;
        }

    });
}
```

```
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
////////////////////// Sensor methods //////////////////////////
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////

/**
```

```
 * Method to start and register the sensors on the manager
 */
protected void Ini_Sensores() {

    //We only need to register the gyroscope
    mSensorManager.registerListener(this,
            mSensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE),
            SensorManager.SENSOR_DELAY_FASTEST);

}

/**
 * Method to stop listening to the sensors
 */
private void Parar_Sensores() {

    mSensorManager.unregisterListener(this,
            mSensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE));

}

/**
 * Method that is triggered when the sensor's data is changed
 * In this activity the behaviour of this method changes depending on the kind of de-
vice(L or R)
 * If it's L device it will always send data to R device when the button is pressed
 * If it's R device it will send data to the server whenever receives data from L or
the button is pressed.
 * In the case of R button is pressed and is simultaneously receiving data from L it
will append R data to
 * L data and send it to the server
 * @param event
 */
@Override
public void onSensorChanged(SensorEvent event) {
    synchronized (this) {
        Log.d("sensor", event.sensor.getName());

        switch (event.sensor.getType()) {
```

```
            case Sensor.TYPE_GYROSCOPE:

                //Initialization of the JSONObject of the phone
                JSONObject local_values = JSONUtilities.toJSONOBjectMovil(hand, Float.-
toString(event.values[0]),Float.toString(event.values[1]),Float.toString(event.values[2])
);
                //Message to send
                String msg_env;
                //Message to receive
                String msg_rec = null;
                //JSONObject to receive
                JSONObject json_rec = null;

                //If the button is being pressed
                if(sendingRotation) {

                   //Intermediary case
                   if( hand.equals("R") ) {
                      //We try to receive from L
                      msg_rec = receiver.getMessage();
                      //If we have received a message from L
                      if(msg_rec != null) {
                         try {
                            json_rec = new JSONObject(msg_rec);
                         } catch (JSONException e) {
                            e.printStackTrace();
                         }
                         //We create a JSONObject with both JSONObjects from L and R and
parse it to String
                         msg_env            =            JSONUtilities.toJSONOBjectEnvol-
vente(json_rec,local_values).toString() + "\n";

                      }
                      //If we haven't receive a message from L
                      else
                         //We create a JSONObject with only R and parse it to String
                         msg_env   =   JSONUtilities.toJSONOBjectEnvolvente(null,local_val-
ues).toString() + "\n";
```

```
                }
                //Not the intermediary
                else
                    //The message will be only the L JSONObject parsed to String
                    msg_env = local_values.toString() + "\n";

                    //We send whether the R message or L+R message to the intermediary or
to the server
                    this.send(msg_env);

            }
            //The button is not pressed but we try to receive data from L anyway
            else if ( hand.equals("R") ){

                msg_rec = receiver.getMessage();
                //If we receive a message from L
                if(msg_rec != null) {

                    try {
                        json_rec = new JSONObject(msg_rec);
                    } catch (JSONException e) {
                        e.printStackTrace();
                    }
                    //We create a JSONObject with only L and parse it to String
                    msg_env             =             JSONUtilities.toJSONOBjectEnvol-
vente(json_rec,null).toString() + "\n";
                    //Send message to the server
                    this.send(msg_env);
                }

            }


            break;

        }
    }
 }
```

```
/**
 * Method that is triggered when the accuracy of the sensor is changed
 * @param sensor
 * @param accuracy
 */
@Override
public void onAccuracyChanged(Sensor sensor, int accuracy) {

}

/**
 * Method that is triggered when the sensors are told to stop
 */
@Override
protected void onStop() {

    Parar_Sensores();

    super.onStop();
}

/**
 * Method that is triggered when the activity is finished
 */
@Override
protected void onDestroy() {
    // TODO Auto-generated method stub

    Parar_Sensores();

    super.onDestroy();
}

/**
 * Method that is triggered when the activity is paused
 */
@Override
```

```
protected void onPause() {
    // TODO Auto-generated method stub

    Parar_Sensores();

    super.onPause();
}

/**
 * Method that is triggered when the activity is restarted
 */
@Override
protected void onRestart() {
    // TODO Auto-generated method stub

    Ini_Sensores();

    super.onRestart();
}

/**
 * Method that is triggered when the activity is resumed
 */
@Override
protected void onResume() {
    super.onResume();

    Ini_Sensores();

}

//////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////
////////////////////// Sender methods ////////////////////////////
//////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////
```

```
  /**
   * Method to send a message to the sender
   * @param msg
   */
  public void send( String msg ){

     sender.send(msg);

  }

}
```

En esta clase destacar el método onSensorChange que se encarga de toda la funcionalidad de envío y recepción de mensajes que se explicó junto al diagrama de comunicación de la figura 5.5. Por otro lado el móvil de la derecha hace uso de una clase encargada de recibir mensajes y que se llama MessageReceiver, su código es el siguiente:

MessageReceiver.java

```
/**
* Class in charge of creating a socket to listen if we receive data from another device
*/

public class MessageReceiver extends AsyncTask<Void, Void, Void> {

  ////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////
  ////////////////// Activity variables /////////////////////////
  ////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////

  //State variable
  private boolean running;
```

```
//Listening socket
private ServerSocket ss;
//Connection socket
private Socket s;
//Stream reader
private InputStreamReader isr;
//Buffer reader
private BufferedReader br;
//Message received
private String message;

/**
 * Constructor of MessageReceiver
 * Initially running is set to true and message is null in order to show
 * that no message has been received yet
 */
public MessageReceiver(){
    running = true;
    message = null;
    Log.i("I","Constructor socket receiver");

}
/**
 * Process to do on the thread. It will be started when the start method of the object
is called
 * @param voids Array of string that contains the information needed when the thread is
started.
 */
@Override
protected Void doInBackground(Void... voids){

    try {
        Log.i("I","Creting socket receiver");
        ss = new ServerSocket(6800);
        Log.i("I","Socket receiver created");
        s = ss.accept();
        Log.i("I","New conection in socket receiver");
        isr = new InputStreamReader(s.getInputStream());
        br = new BufferedReader(isr);
```

```
        while (running) {

            message = br.readLine();

        }

        s.close();
        ss.close();
    }
    catch (IOException e) {
        e.printStackTrace();
    }

    return null;
}

/**
 * Method that returns the last message received if so
 * @return String if a message is received and null otherwise
 */
public String getMessage(){

    String ret_msg = message;
    message = null;
    return ret_msg;

}

@Override
protected void onProgressUpdate(Void... Voids){

    super.onProgressUpdate();

}

/**
```

```
   * Method that stops the execution of the receiver
   */
  public void stop(){

     running = false;

  }

}
```

El código de esta clase ha sido reutilizado prácticamente entero del servidor de prueba que se hizo en java para NetBeans y se explicó en el apartado A). Por otro lado implementa los métodos de la interfaz AsyncTask para que pueda ser lanzada en otra hebra.

Cuando en el móvil de la derecha se creó un MessageReceiver y un Message-Sender surgió un problema y no se ejecutaban las dos clases. Tras investigar bastante descubrí que la versión de la API que estaba usando solo podía lanzar una hebra AsynTask en su propio pool de hebras por lo tanto tuve que crear otro pool de hebras donde lanzar ambas hebras y que no hubiese problemas.

Esta actividad hace uso de las funciones de JSON comentadas anteriormente por lo que Unreal Engine ya podía obtener los datos de los mensajes. El blueprint encargado de la interacción se muestra en la figura 5.20:

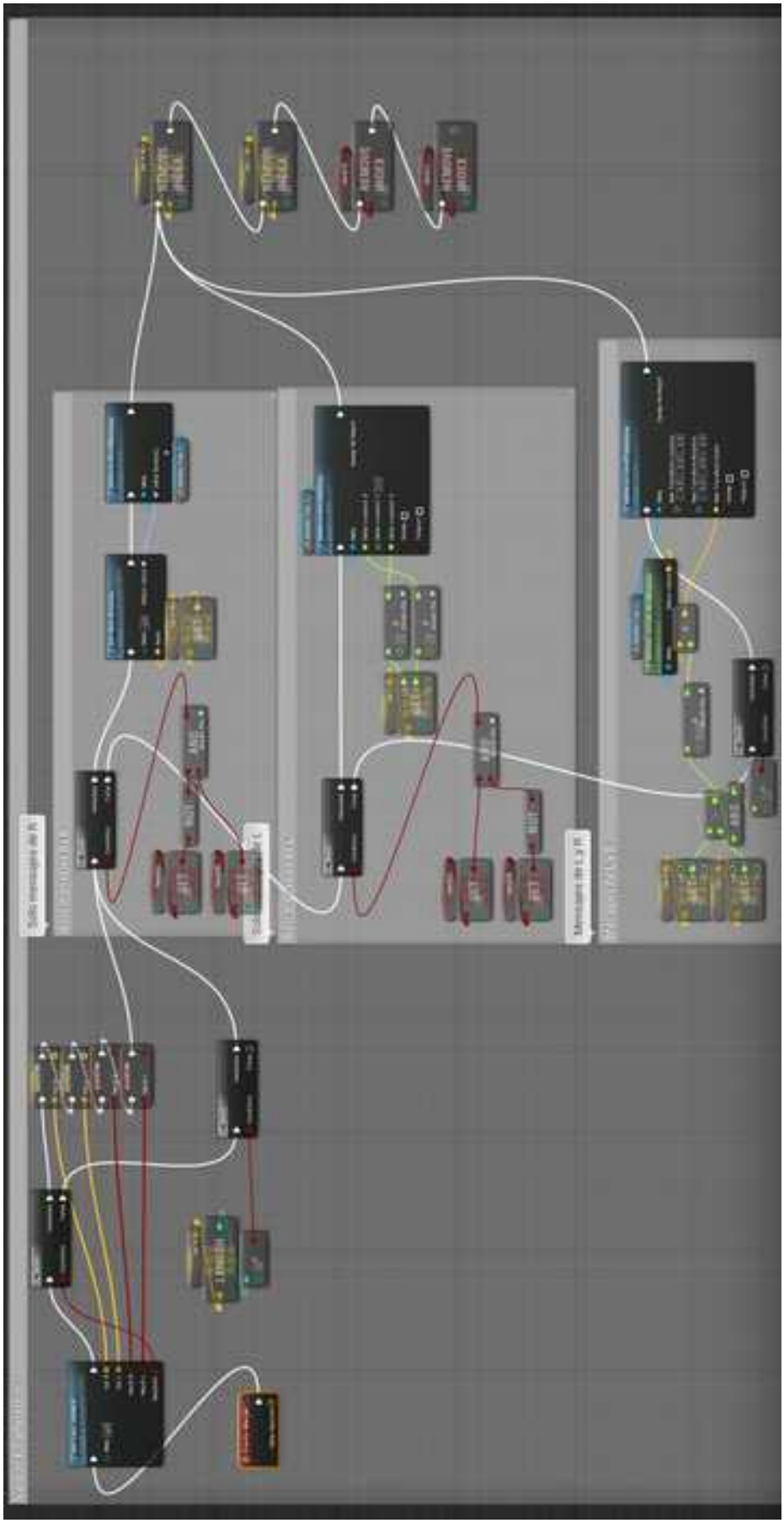

Figura 5.20

Este blueprint separa en tres casos los mensajes recibidos y los interpreta según se explicó en el apartado 5.5.2 de interpretación de datos. Las interfaces para este método de interacción se muestran en las figuras 5.6.2.11 y 5.6.2.12

Figura 5.21: Interfaz para elegir dispositivo

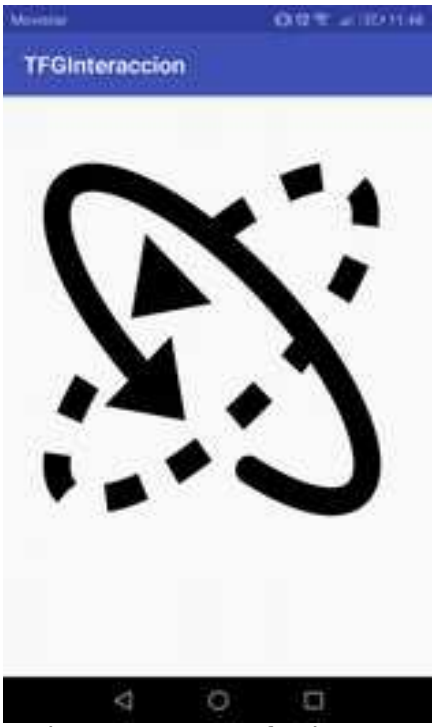

Figura 5.22: Interfaz interacción completa

Tras la implementación de este método se consiguió un interacción bastante fluida. El mayor desafío de esta parte fue crear un móvil como intermediario y que pudiese combinar los mensajes de ambos móviles para enviarlos. Una vez hecho eso se crearon métodos de interpretación similares a los creados anteriormente.

A continuación en la figura 5.23 se muestra la interfaz de la actividad principal que permite lanzar los métodos de interacción comentados. En ella se han ido incluyendo botones según se han ido desarrollando prototipos.

Figura 5.23: Actividad principal

6. Ejemplos de uso

En este apartado se muestran algunos usos de los distintos métodos de interacción mediante fotografías:

A) Rotación libre y zoom

En la ifigura 6.1 aparece el entorno 3D con el modelo en él y cómo se sostiene el móvil para poder usarlo.

Figura 6.1: Ejemplo de uso de dispositivo móvil en rotación libre

En la figura 6.2 se muestra un caso de uso en el que el usuario ha pulsado el botón y ha girado el móvil hacia la izquierda. Se puede observar que el modelo también ha girado en la misma dirección.

Figura 6.2: Ejemplo de rotación de modelo

En la figura 6.3 se muestra el caso de que el usuario desplace el slider del zoom y agranda el modelo.

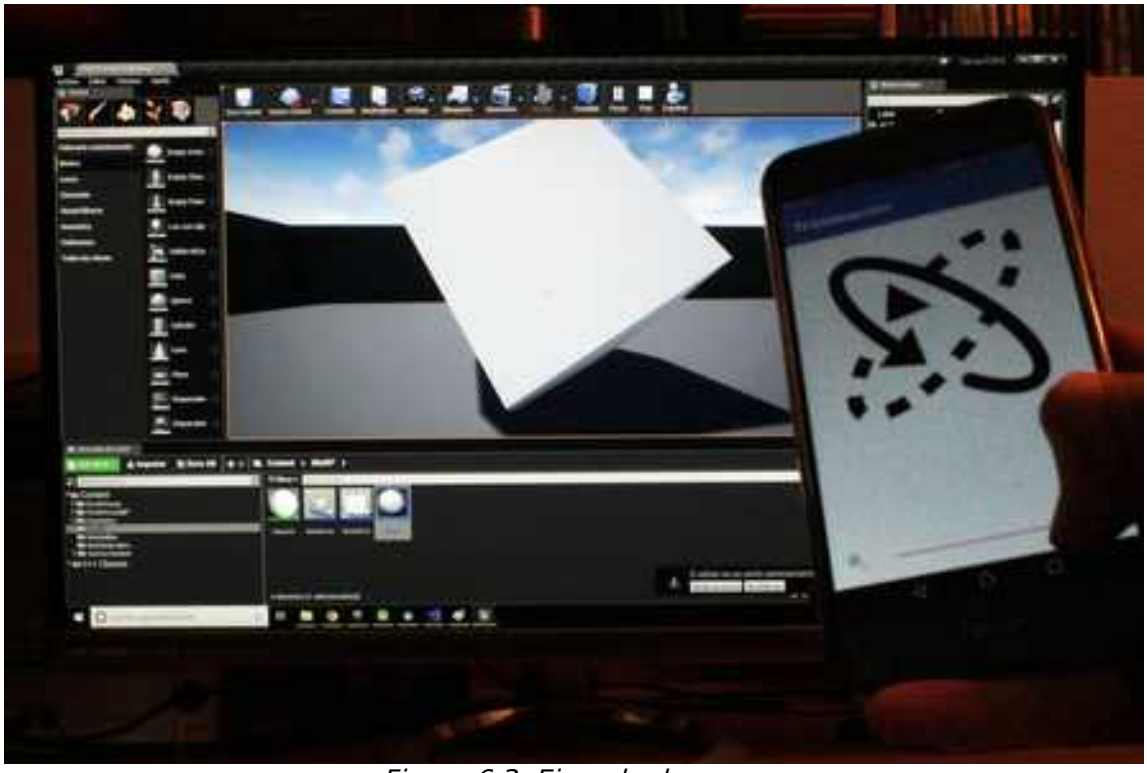

Figura 6.3: Ejemplo de zoom

B) Rotación sobre ejes

A continuación en las figuras 6.4, 6.5 y 6.6 se muestran ejemplos de uso para el método de interacción de rotación sobre ejes aislados para cada uno de los ejes. Se puede apreciar que aparece el anillo correspondiente al eje de rotación cuando lo elegimos.

Figura 6.4: Ejemplo de rotación en el eje x

Figura 6.5: Ejemplo de rotación en el eje y

Figura 6.5: Ejemplo de rotación en el eje z

C) Interacción completa

Para este método de interacción se muestra cómo se deben usar los dispositivos móviles (figura 6.6) y un ejemplo en el que se pulsan los botones de ambos móviles a la vez y se giran para hacer zoom al modelo (figura 6.7)

Figura 6.6: Modo de uso en interacción completa

Figura 6.7: Ejemplo de zoom en interacción competa

7. Conclusiones

En este apartado se analizará el trabajo realizado durante todo el proyecto, se comprobarán si se han cumplido los requisitos establecidos y se comentarán las competencias adquiridas.

Si nos remontamos a la especificación de requisitos de esta memoria podemos comprobar que se han cumplido todos los requisitos entre los que destacan haber conseguido un método de interacción en un entorno 3D a partir del uso de dos dispositivos móviles y que la interacción sea suave y ergonómica

Se han estudiado e implementado métodos de interpretación de los datos de los sensores del teléfono a datos comprensibles por Unreal Engine y aplicables a un modelo 3D. Junto a eso se han adquirido conocimientos en transmisión de datos por sockets TCP y el desarrollo de servidores. Se ha ampliado el conocimiento en programación de aplicaciones Android, el desarrollo de videojuegos en Unreal Engine y la paralelización de tareas.

En cuanto al coste económico se ha cumplido para los métodos de interacción que implican el uso de un sólo móvil ya que, como se explicó, la mayoría de personas tiene un móvil y no es necesario que gasten dinero en otros dispositivos. Para el método de interacción con dos móviles esto no se cumple y además está el problema de que ninguna persona, aunque pueda económicamente, tiene dos móviles consigo siempre.

Por lo tanto, se ha logrado un método de interacción con un móvil que cumple todos los requisitos planificados sin embargo no se ha podido implementar la transformación canónica de traslación sin complicar la interfaz como se hizo con el método de dos móviles.

8. Trabajo futuro

Este proyecto ha sentado la base en el desarrollo de métodos de interacción a partir de los sensores del móvil. Con el conocimiento que se ha adquirido y los programas

desarrollados se puede ampliar fácilmente el alcance del proyecto. Algunos posibles rumbos que podrían tomarse son:

- Creación de un método de interacción con un solo móvil que incluya las tres transformaciones nombradas sin complicar la interfaz.

- La inclusión de gafas de realidad virtual en el uso de estos métodos de interacción gracias a la simplicidad de las interfaces.

- Acceder a más sensores para que su uso combinado pueda ofrecer información más exacta de las fuerzas que se aplican a los móviles y por lo tanto que la interacción sea más precisa o que se puedan implementar otras acciones en el modelo.

- Gracias a la realidad aumentada, se podría dar uso de la cámara del móvil para que se pueda obtener una posición absoluta del móvil en el espacio físico y su orientación respecto a un punto determinado. Esto se podría hacer mostrando por la pantalla del ordenador una imagen o imprimiéndola para que la cámara la detectase y que con un algoritmo se transformara el tamaño de la imagen y su perspectiva en una posición absoluta del dispositivo móvil. Con esto se podrían captar las traslaciones del usuario por el espacio físico y cuán cerca de la imágen está para traducirlo en traslaciones en el espacio virtual.

- El uso de los métodos de interacción desarrollados en videojuegos. Se podrían desarrollar videojuegos dándole un uso al móvil similar al de los mandos de la Nintendo Wii. Incluso se podría dar uso al método de interacción con dos móviles ya que se podría crear un videojuego multijugador local y que dos personas compartan los móviles.

9. Bibliografía

- [1] Bowman, D.A., Kruijff, E., Laviola, J.J., Poupyrev I., 2004. *3D User Interfaces: Theory and Practice*
- [2] Especificaciones técinas de Nintendo Wii U. Página oficial de Nintendo. https://www.nintendo.com/wiiu/features/tech-specs
- [3] Información de Kinect. Página oficial de Xbox.

https://www.xbox.com/es-ES/xbox-one/accessories/kinect

- [4] Información de Gypsy 7 Motion Capture System. Página oficial de MetaMotion. http://metamotion.com/gypsy/gypsy-motion-capture-system.htm
- [5] Información sobre Sim-Ortho. Página oficial de OSimTech. https://ossimtech.com/en-us/Simulators
- [6] Información sobre HTC Vive. Página oficial de Vive. https://www.vive.com/us/
- [7] Información de PlayLink. Página oficial del videojuego. https://www.playstation.com/es-es/explore/ps4/games/playlink/
- [8] Información sobre AirConsole. Página oficial de AirConsole. https://www.airconsole.com/
- [9] Información sobre Adrift. Página oficial del videojuego. http://www.adr1ft.com/
- [10] Audi Media Center. *Audi launches Virtual Reality technology in dealerships.* https://www.audi-mediacenter.com/en/press-releases/audi-launches-virtual-reality-technology-in-dealerships-9270
- [11] Unreal Engine. Documentación de Unreal Engine. https://docs.unrealengine.com/en-us/
- [12] Android Studio. Documentación de Android Studio. https://developer.android.com/docs/
- [13] Blender. Documentación de Blender. https://docs.blender.org/manual/en/dev/index.html
- [14] Visual Studio. Documentación de Visual Studio. https://docs.microsoft.com/es-es/visualstudio/
- [15] Github. https://github.com/
- [16] Scrum. Página oficial de Scrum. https://www.scrum.org/

10. Apéndices

10.1 Cómo conseguir el código

El código que se ha creado para esta práctica se encuentra en mi GitHub[15] personal en el siguiente enlace:

https://github.com/adritake/TFG/

Los proyectos completos se encuentran para su descarga en mi DropBox en el siguiente enlace:

https://www.dropbox.com/sh/z8nzweny72cd0vg/AABqGMHxEeGZK1fTtn5x6H1Oa?dl=0

10.2 Manual de instalación

En esta sección se muestran los pasos que hay que seguir para instalar el software necesario para hacer funcionar esta práctica. Para la parte del ordenador necesitamos:

- Unreal Engine: Para su instalación hay que seguir los pasos indicados por su desarrollador. Se puede descargar en la página oficial de Unreal Engine en el botón de “Download”: https://www.unrealengine.com/en-US/blog

Para la parte de móvil basta con descargar la aplicación que se encuentra en mi GitHub personal en el siguiente enlace e instalarla en un dispositivo móvil con Android:

https://github.com/adritake/TFG/blob/master/Android%20part/app-debug.apk

Para instalar el proyecto en Unreal Engine es necesario descargar el siguiente proyecto y copiarlo en la carpeta de proyectos del motor (normalmente suele estar en *C:\Users\TuUsuario\Documents\Unreal Projects*). Una vez copiado, hay que abrir la carpeta del proyecto y ejecutar con Unreal Engine el archivo "*ServerTCP3.uproject*". Toda la funcionalidad del proyecto se encuentra en el blueprint llamado *Server*.

Enlace al proyecto de Unreal Engine:

https://www.dropbox.com/s/aqziv1cdtvyxome/TFGInteraccion.rar?dl=0

10.3 Manual de usuario

En este apartado se explica cómo usar el proyecto para lograr hacer funcionar los métodos de interacción desarrollados. Se asume que la aplicación está instalada y el proyecto de Unreal Engine está instalado y ejecutando. Para todos lo métodos de interacción que se explican es necesario que los móviles y el ordenador estén conectados a la misma red.

A) Interacción con rotación libre

Para este método hay que abrir el blueprint Server del proyecto de Unreal Engine y conectar el nodo "Evento Marcar" al apartado de "Interpolación giroscopio con cola FIFO y zoom" como se muestra en la figura 10.1:

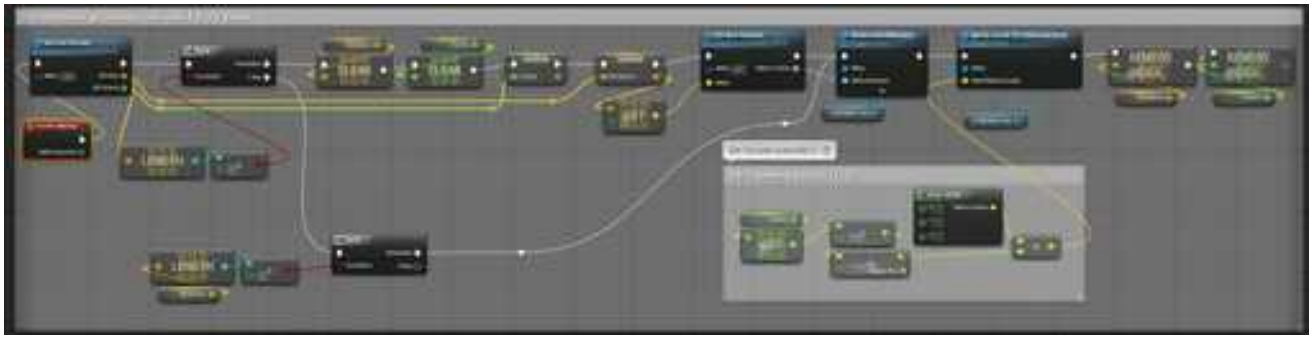

Figura 10.2.1

Una vez hecho esto, en el móvil hay que ejecutar la aplicación y aparecerá la pantalla mostrada en la figura 10.2

Figura 10.2: Pantalla principal de la aplicación

Donde aparece la IP hay que introducir la IP del ordenador. Ya está todo listo para ejecutar este método de interacción. Lo siguiente que hay que hacer es ejecutar el juego en Unreal Engine y pulsar el botón “Free” de la aplicación móvil.

Para rotar el modelo hay que pulsar el dibujo que aparece en pantalla y mover la mano. Para hacerle zoom hay que deslizar el slider que aparece en pantalla.

B) Interacción con rotación en ejes aislados

El procedimiento es parecido al anterior caso pero esta vez,en el blueprint, hay que conectar el nodo marcar al apartado “Versión Axis”(figura 10.3), escribir la IP del ordenador en el móvil, ejecutar el juego y pulsar el botón “Axis”. Ahora para rotar el modelo hay que pulsar uno de los tres botones que aparecen y mover la mano.

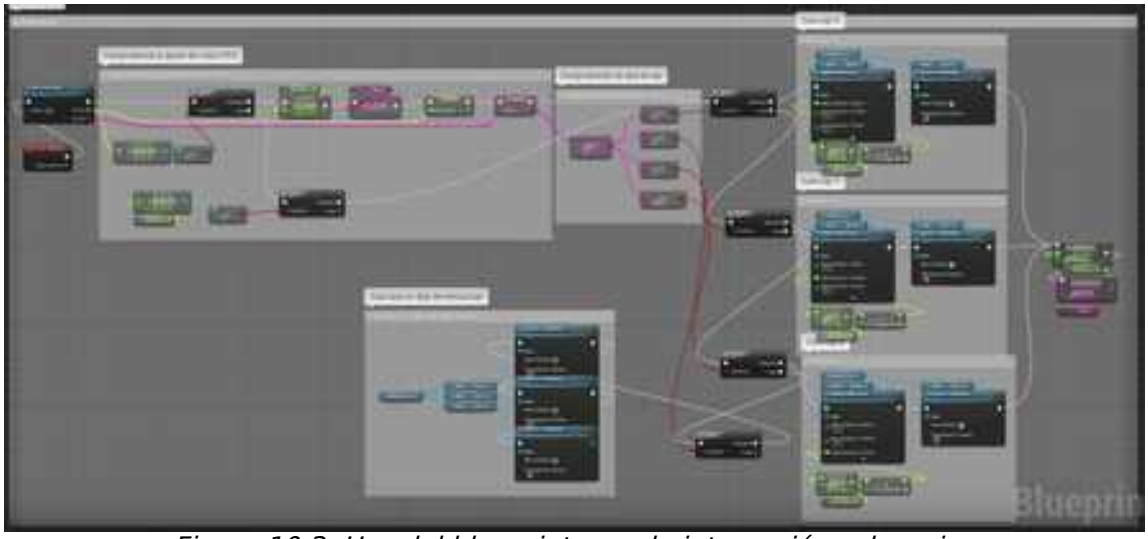

Figura 10.3: Uso del blueprint para la interacción sobre ejes

C) Interacción completa

El uso de este método requiere de un poco más de configuración previa. El primer paso es conectar el evento marcar al apartado de “Versión full interaction” tal y como se muestra en la figura 10.4:

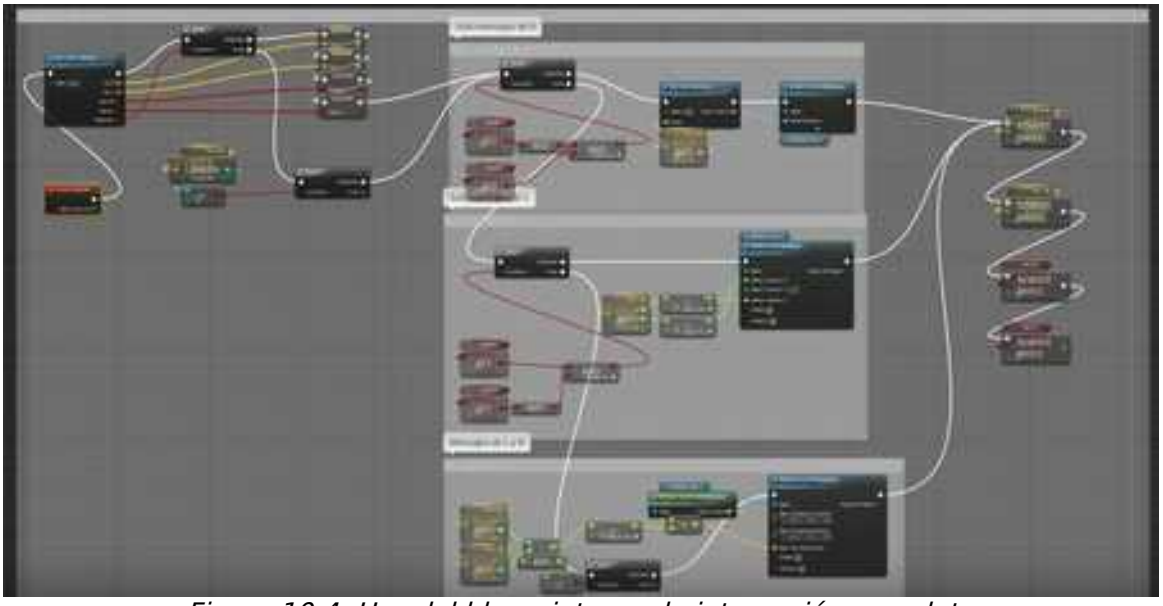

Figura 10.4: Uso del blueprint para la interacción completa

Para la parte de los móviles es necesario que dos dispositivos tengan instalada la aplicación. En el móvil de la derecha se debe escribir la IP del ordenador en la pantalla principal y pulsar el botón “L/R”. Para el móvil de la izquierda basta con pulsar el botón “L/R”. Tras pulsar este botón, en ambos móviles aparecerá la pantalla siguiente(figura 10.5):

Figura 10.5: Pantalla de elección de móvil

Tras ejecutar el juego, en la aplicación móvil hay que pulsar el botón “Right” para el caso del móvil de la derecha y para el móvil de la izquierda hay que indicar la IP del móvil de la derecha y pulsar el botón Left. En ambos móviles aparecerán una nueva pantalla con un dibujo ocupándola completamente.

Con los pasos anteriores habremos conectado los dispositivos correctamente y ya se podrá interactuar con el modelo. Para rotarlo hay que pulsar la pantalla del móvil de la derecha y girar la mano. Para trasladarlo hay que pulsar la pantalla del móvil de la izquierda y girar la mano. Para hacer zoom hay que pulsar la pantalla de ambos móviles a la vez y rotar las manos en sentido contrario.