

UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR



Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

GOOGLE CARDBOARD ROBOTIC AVATAR

Autor: Borja Mauricio Fourquet Maldonado

Tutor: Francisco Saiz López

Junio 2016

GOOGLE CARDBOARD ROBOTIC AVATAR

Autor: Borja Mauricio Fourquet Maldonado
Tutor: Francisco Saiz López

Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio 2016

Resumen

El objetivo principal de este Trabajo de Fin de Grado es desarrollar un sistema distribuido heterogéneo, combinando visión remota y realidad virtual, donde el usuario pueda ver a través de los ojos de un *avatar robótico*.

El cliente consiste en unas gafas de realidad virtual, como lo son Google Cardboard. Es un artilugio de cartón con dos huecos para un par de lentes, que apuntan a una cavidad donde se ha de situar un smartphone. Una aplicación mostrará dos imágenes en la pantalla del dispositivo móvil que serán observadas a través de estas lentes por el usuario.

En la parte del servidor tenemos un par de videocámaras situadas en una estructura sobre un motor. Una parte del servidor envía los fotogramas capturados por las cámaras a través de un protocolo de multimedia en red hasta el cliente, donde cada transmisión se corresponde con un ojo del usuario.

Por otra parte, el servidor también se encargará de hacer rotar la estructura con las cámaras, controlando el motor según los movimientos de la cabeza del usuario. La aplicación envía periódicamente la rotación del dispositivo, que se calcula utilizando el acelerómetro y la brújula del smartphone, a este servidor de control.

Palabras Clave

Google Cardboard, Realidad Virtual, Redes Multimedia, Android, Servomotor, App, Transmisión de vídeo, Tiempo Real, Sensores, Sistema Distribuido

Abstract

The main goal of this Bachelor Thesis is to develop an heterogeneous distributed system, combining remote vision and virtual reality, where the user can look through the eyes of a *robotic avatar*.

The client consists of a pair of virtual reality glasses, such as Google Cardboard. It is a cardboard made gadget with two holes for a pair of lens that look to a cavity where the smartphone must be placed. An application will show two images on the mobile device's screen that will be observed through this lens by the user.

There is a couple of videocameras on the server side, assembled in a structure on a motor. A part of the server sends the frames captured by the cameras using a multimedia networking protocol to the client, where each stream corresponds to an eye.

On the other hand, the server is also responsible for rotating the structure with the cameras controlling the motor according to the movements of the user's head. The application periodically sends the rotation of the device, which is calculated using the smartphone's accelerometer and compass, to the control server.

Key words

Google Cardboard, Virtual Reality, Multimedia Networking, Android, Servomotor, App, Video Streaming, Real Time, Sensors, Distributed System

Agradecimientos

Agradecimientos

Todo list

Agradecimientos	v
Estado del arte: Historia, nacimiento y evolución.	5
Estado del arte: Conceptos básicos de la realidad virtual	7
Manual de usuario: Repositorio	51
Manual de usuario: Uso	51
Manual de usuario: Despliegue de los servidores de vídeo	51
Manual de usuario: Código arduino	51
Manual de usuario: Despliegue del servidor de control	51

Índice general

Índice de Figuras	XII
Índice de Tablas	XVI
1. Introducción	1
1.1. Motivación del proyecto	1
1.2. Objetivos y enfoque	1
1.3. Metodología y plan de trabajo	2
1.3.1. Recursos	2
1.3.2. Producción	3
1.3.3. Despliegue	3
1.3.4. Documentación	3
2. Estado del arte	5
2.1. Historia, nacimiento y evolución.	5
2.1.1. Sensorama	5
2.1.2. Headsight	5
2.1.3. Realidad virtual en los 90'	6
2.1.4. Realidad virtual en el siglo XXI	6
2.2. Conceptos básicos de la realidad virtual	7
2.2.1. Vista estereoscópica	7
2.2.2. <i>Head tracking</i>	7
3. Análisis, arquitectura y diseño del sistema	9
3.1. Análisis	9
3.1.1. Descripción del producto	9
3.1.2. Objetivos y funcionalidad	9
3.1.3. Requisitos	10
3.1.4. Tamaño y rendimiento	10
3.2. Arquitectura	10
3.2.1. Vista estática	10

3.2.2. Vista funcional	11
3.2.3. Vista dinámica	12
3.3. Diseño	14
3.3.1. Servidor de vídeo	14
3.3.2. Servidor de control	14
3.3.3. Cliente móvil	14
4. Tecnología y pruebas	17
4.1. Calidad de servicio	17
4.1.1. Servidores	17
4.1.2. Clientes	17
4.1.3. Interfaces	18
4.1.4. Pruebas significativas	18
4.1.5. Configuración final	18
4.2. Cámaras	20
4.2.1. Cámaras con objetivo ojo de pez	20
4.2.2. Cámaras con lente normal	20
4.3. Servomotor	21
5. Implementación	23
5.1. Cliente	23
5.1.1. Fundamentos de programación para Android con Android Studio	23
5.1.2. Aplicación	24
5.2. Servidores de vídeo	25
5.2.1. Configuración del servidor	25
5.3. Servidor de control	26
5.3.1. Circuito y programación de la placa Arduino	26
5.3.2. Cliente/servidor UDP	26
5.3.3. Programación del servidor	27
6. Conclusiones y trabajo futuro	29
6.1. Conclusiones	29
6.2. Trabajo futuro	30
Glosario de acrónimos	31
Bibliografía	32

A. Fragmentos de código	35
A.1. Cliente	35
A.1.1. AddressManager.java	35
A.1.2. PositionSender.java	36
A.1.3. FormActivity.java	36
A.1.4. MainActivity.java	38
A.2. Servidores de vídeo	40
A.2.1. deploy.sh	40
A.3. Servidor de control	41
A.3.1. servo_serial_read.ino	41
A.3.2. control_servo.py	42
A.3.3. udp_server.py	43
B. Montaje y funcionamiento	45
B.1. Robot	45
B.2. Gafas de Realidad Virtual	47
B.3. Aplicación Android	50
C. Manual de usuario	51
C.1. Repositorio	51
C.2. Uso	51
C.2.1. Despliegue de los servidores de vídeo	51
C.2.2. Código arduino	51
C.2.3. Despliegue del servidor de control	51
C.2.4. Instalación de la aplicación	51

Índice de Figuras

1.1. Ejemplo de aplicación para Google Cardboard. Utiliza OpenGL para renderizar los gráficos y los muestra adaptándose a cada ojo.	2
1.2. Movimiento de la cabeza alrededor del eje vertical	2
1.3. Google Cardboard: las gafas de realidad virtual a partir de unas lentes, cartón y un smartphone	3
2.1. Sensorama	6
2.2. Coordenadas polares. Azimuth o Yaw (Ψ), Pitch (Φ), Roll (θ)	7
3.1. Diagrama conceptual de la red del sistema	10
3.2. Diagrama de componentes del sistema. Se pueden apreciar los distintos componentes, las interfaces que ofrecen y requieren y el protocolo que utilizan para comunicarse entre sí.	11
3.3. Diagrama de actividad de la aplicación para Android. En ocre, las vistas de la aplicación. En azul, las interacciones del usuario. En rojo, las respuestas del sistema.	13
3.4. Diagrama de clases del servidor de control	14
3.5. Diagrama de clases de la aplicación para Android	15
4.1. Captura de cámara con lente <i>fish-eye</i>. Esta cámara de alta definición obtiene un gran ángulo de visión a coste de la marcada curvatura de la imagen.	20
4.2. Captura de cámara con lente normal. Tomada a la misma distancia. La cámara de por sí tiene menos resolución, lo cual afecta directamente a la calidad de la imagen y la lente le da <i>zoom</i> a la captura, renunciando a un amplio ángulo de visión pero sin sufrir deformaciones.	21
5.1. Circuito del Arduino controlando el servomotor. El cable rojo va a VCC (3.3V) de la placa arduino, el cable negro a GND (tierra) y el cable blanco al PWM correspondiente. Variando el valor de la señal analógica del PWM, conseguimos cambiar de sentido y de intensidad.	26
A.1. Patrón de diseño <i>Singleton</i>. La única instancia de esta clase puede ser referenciada invocando al método público y estático <i>getInstance()</i>	35
A.2. Atributos de la instancia. Estos son los datos que se almacenan en esta clase. Sus correspondientes <i>getters</i> y <i>setters</i> son de acceso público	35
A.3. Atributos de la clase.	36

A.4. Método constructor.	36
A.5. Método <i>send()</i>. Este método se encarga de enviar un mensaje, en forma de una cadena de caracteres, a través del socket UDP previamente inicializado en el constructor.	36
A.6. Definición de la clase, <i>AddressManager</i> y <i>onCreate()</i>.	36
A.7. Método <i>updateValues()</i>. Controlador del botón «Update values». Accede a los <i>EditText</i> de la vista y los almacena en el modelo (<i>AddressManager</i>)	37
A.8. Método <i>nextActivity()</i>.	37
A.9. Inicialización de objeto <i>VideoView</i>. El mismo proceso se realiza para <i>myVideoViewRight</i> y ambos comparten el mismo <i>MediaController</i>	38
A.10. Acceso a los sensores y creación del objeto <i>PositionSender</i>:	38
A.11. Hilo de envío de la posición al servidor de control	39
A.12. Método <i>onSensorChanged()</i>: Realiza el cálculo de las coordenadas polares (figura 2.2) cada vez que los sensores cambian de valor.	39
A.13. Método <i>onBackPressed()</i>: se ejecuta cuando se presiona el botón de los dispositivos Android con forma de triángulo.	40
A.14. Script de despliegue de los servidores multimedia.	40
A.15. Cabecera del código del Arduino.	41
A.16. Función auxiliar <i>myRead()</i>. Lee carácter a carácter del puerto serial hasta encontrar el valor '\n' y convierte la cadena de caracteres obtenida en un valor numérico.	41
A.17. Función <i>setup()</i> de Arduino. Esta función se ejecuta al arrancarse la placa.	41
A.18. Función <i>loop()</i> de Arduino. Esta función se ejecuta indefinidamente después de ejecutarse <i>setup()</i>	42
A.19. Definición de la clase y constructor de <i>ServoControl</i>. El ángulo inicial es $\Psi_0 = 0^\circ$. El constructor crea un objeto <i>Serial</i> en el puerto 9600 a partir de su ruta en el sistema operativo.	42
A.20. Definición de la clase y constructor de <i>ServoControl</i>.	42
A.21. Inicialización de recursos del servidor de control.	43
A.22. Bucle del servidor de control.	43
B.1. Circuito electrónico: las conexiones a la placa arduino se corresponden con el diseño en de la figura 5.1. El cable conectado por la derecha de la placa se conecta al PC a través de cualquier puerto USB.	45
B.2. Videocámaras: se encuentran unidas por una estructura de cartón fijada con cinta americana.	46
B.3. Videocámaras sobre el servomotor: la estructura con las cámaras se coloca perpendicularmente sobre una estructura circular de un par de pulgadas de diámetro, que a su vez está enganchada al engranaje al cual hace girar el servomotor.	46
B.4. Google Cardboard original:	47
B.5. Gafas de realidad virtual de plástico: estas gafas son más ergonómicas, ya que tiene gomaespuma en las zonas que hacen contacto con la cara y la cinta elástica se ajusta mejor a la cabeza.	48

B.6. Gafas de realidad virtual de plástico: la parte blanca está compuesta de unas ventosas sobre las que se adhiere el dispositivo móvil para que éste no se mueva con los movimientos de la cabeza del usuario.	48
B.7. Gafas de realidad virtual de plástico: las lentes enfocan las imágenes que se muestran en la pantalla del dispositivo móvil, adaptándolas al ojo para que el usuario pueda sentir la profundidad a partir de las dos vistas.	49
B.8. Smartphone Android: Utiliza la versión Android 5.1.1 Lollipop como sistema operativo. Resolución full HD (1920x1080p).	49
B.9. Formulario de la aplicación móvil. Permite al usuario modificar las direcciones de los tres servidores.	50
B.10. Vista principal de la aplicación móvil. Cada imagen se corresponde a una captura de una de las cámaras alojadas en el servidor. Esta disposición de las fotografías otorga al usuario visión estereoscópica, es decir, la capacidad de sentir profundidad.	50

Índice de Tablas

4.1. Pruebas significativas de la calidad de servicio	19
4.2. Relación aproximada entre el desplazamiento angular y el valor analógico de entrada del servomotor	21

1

Introducción

A little imagination, the right tools, and you can have a virtual world at your fingertips...

1.1. Motivación del proyecto

Mi motivación personal a la hora de embarcarme en este proyecto es la de crear un sistema informático heterogéneo, investigando así campos como la programación para dispositivos móviles (en este caso, para Android) y las redes multimedia.

Por otra parte, mi interés por la realidad virtual, que tanto está creciendo en cuestión de meses, me lleva a integrarla en este proyecto.

Por último, mi experiencia en las prácticas de empresa que realicé en el proyecto de emprendimiento The Haptic Eye [1] me animaron a llevar a cabo otro proyecto innovador, en tiempo real y que reúna tecnologías punteras en el estado del arte.

1.2. Objetivos y enfoque

- Investigar los distintos protocolos e implementaciones de transmisión de vídeo en tiempo real a través de la red para encontrar el más idóneo para este proyecto.
- Aprender los fundamentos de la programación de aplicaciones para dispositivos móviles, así como familiarizarme con el uso del IDE Android Studio.
- Asimilar las bases de la realidad virtual y, concretamente, de las aplicaciones para las gafas de VR *Google Cardboard* o similares (figura 1.3).
- Diseñar el sistema del producto, tanto *front-end* como *back-end*, teniendo en cuenta las características distribuidas y en tiempo real del software para crear un prototipo, a modo de prueba de concepto (POC), con las funcionalidades básicas.
- Redactar unas conclusiones del proyecto que sirvan de guía a toda aquella persona que se aventure en un proyecto similar y enumerar las posibles mejoras futuras a partir de las facilidades e impedimentos que se han dado en el desarrollo de este prototipo y teniendo

en mente el estado actual de las tecnologías implicadas, así como las perspectivas de que éstas crezcan y mejoren en el futuro inmediato.

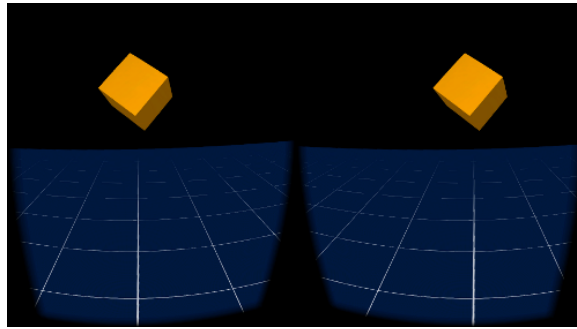


Figura 1.1: **Ejemplo de aplicación para Google Cardboard.** Utiliza OpenGL para renderizar los gráficos y los muestra adaptándose a cada ojo.

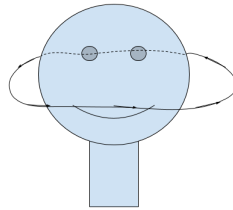


Figura 1.2: **Movimiento de la cabeza** alrededor del eje vertical

1.3. Metodología y plan de trabajo

En este apartado se enumeran las herramientas utilizadas para llevar a cabo este proyecto.

1.3.1. Recursos

Estos son los recursos físicos más significativos utilizados en este proyecto.

- PC con procesador **Intel Core i7 4790K @ 4.0GHz**
- **Arduino UNO**
- **Servomotor a 3.3V**
- **Videocámaras USB** corrientes (*webcams*) x 2

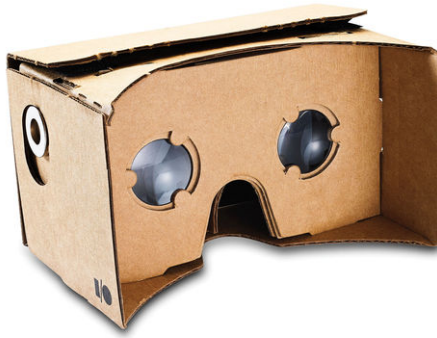


Figura 1.3: **Google Carbdboard**: las gafas de realidad virtual a partir de unas lentes, cartón y un smartphone

1.3.2. Producción

La producción de todo el software se ha llevado a cabo en el PC de sobremesa anteriormente mencionado.

- **Ubuntu 14.04**: popular sistema operativo basado en Debian.
- Repositorio en **BitBucket** con git (<https://bitbucket.org/>)
- **Android Studio** para el cliente Android del sistema
- **Eclipse Mars.2** para el cliente con JavaCV
- **Sublime Text 3** para editar archivos Bash, Makefiles y código Python.
- **Arduino IDE** para el código de la placa Arduino

1.3.3. Despliegue

El despliegue de los servidores se lleva a cabo mediante la terminal de Linux

- **Ubuntu 14.04**
- **Makefile**
- **Bash**
- **cvlc** (VLC para la terminal de Linux)

1.3.4. Documentación

La documentación se ha elaborado utilizando las siguientes herramientas:

- **TeXstudio**: editor de documentos LaTeX.
- **Creately**: aplicación web para el diseño de diagramas que puede ser utilizada en <http://creately.com/>
- **README.md** de BitBucket. Markdown (.md) es un lenguaje de marcas, mucho más simple que HTML, orientado a documentar repositorios.

2

Estado del arte

2.1. Historia, nacimiento y evolución.

Estado del arte: Historia, nacimiento y evolución.

En la década de 1930 una historia de escritor de ciencia ficción Stanley G. Weinbaum (Pygmalion's Spectacle) contiene la idea de un par de gafas que permiten al usuario experimentar un mundo de ficción a través de la holografía, el olfato, el gusto y el tacto. En retrospectiva, la experiencia Weinbaum describe para los que llevan las gafas son asombrosamente como la experiencia moderna y emergente de la realidad virtual, haciendo de él un verdadero visionario del campo.

2.1.1. Sensorama

El Sensorama era una máquina, fue uno de los primeros ejemplos conocidos de inmersión, multi-sensoriales (ahora conocido como la tecnología multimodal). Morton Heilig, su autor y que hoy en día sería considerado como un especialista "multimedia", en la década de 1950 vio el teatro como una actividad que podría abarcar todos los sentidos de una manera eficaz, elaborando así al espectador a la actividad en pantalla. El Sensorama fue capaz de mostrar estereoscópica 3-D imágenes en una vista de gran ángulo, ofrecen inclinando el cuerpo, el suministro de sonido estéreo, y también tenía pistas para la energía eólica y aromas que se activará durante la película.

2.1.2. Headsight

Es el precursor de las HMD o gafas de realidad virtual como las conocemos hoy en día. Se incorpora una pantalla de video para cada ojo y un sistema de seguimiento de movimiento magnético, que está vinculado a una cámara de circuito cerrado. El Headsight no fue realmente desarrollado para aplicaciones de realidad virtual (el término no existía entonces), pero para permitir la visualización remota de inmersión de situaciones peligrosas por los militares. movimientos de la cabeza se moverían una cámara remota, lo que permite al usuario buscar de forma natural en todo el medio ambiente. Headsight, que data de 1961, refleja esencialmente el producto que se pretende desarrollar en este TFG.

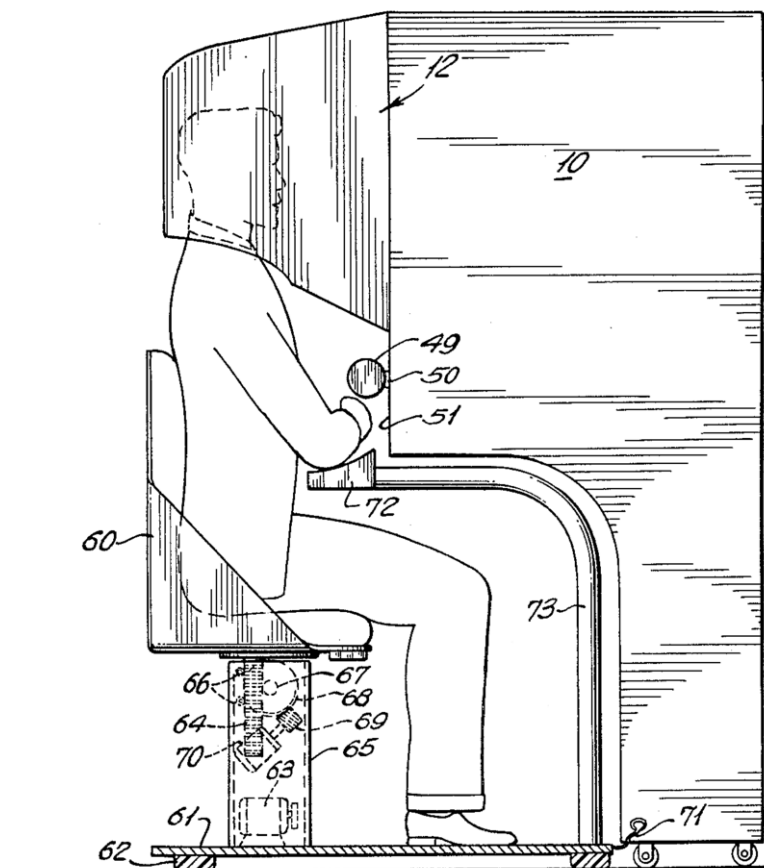


Figura 2.1: Sensorama

2.1.3. Realidad virtual en los 90'

En las dos décadas posteriores a Headsight, surgen distintos sistemas sin demasiado éxito y sin tener realmente en mente la idea de desarrollar un dispositivo de realidad virtual. Cabe destacar las gafas de realidad virtual de SEGA, Sega VR, que fueron anunciadas pero nunca llegaron a ser más que un prototipo o las Virtual Boy de Nintendo que tampoco llegaron a tener éxito porque la tecnología no lo permitía.

2.1.4. Realidad virtual en el siglo XXI

En los primeros tres lustros del siglo XXI, la realidad virtual se ha desarrollado bastante más rápido que en las décadas anteriores. La ley de Moore expresa que aproximadamente cada dos años se duplica el número de transistores en un microprocesador, lo cual afecta directamente a la potencia y el rendimiento de los componentes informáticos, haciéndolos mejorar en una razón exponencial. Por esto, lo que en cierto momento de la Historia se había visto como mera ciencia ficción, se está convirtiendo en una realidad.

Parece claro que 2016 será un año clave en la industria de la realidad virtual. Múltiples dispositivos de consumo que parecen responder finalmente a las promesas incumplidas por la realidad virtual en la década de 1990 llegarán al mercado en ese momento. Estos incluyen el pionero Oculus Rift, que fue comprada por el gigante de las redes sociales Facebook en 2014. El Oculus Rift salió al mercado este mismo año, compitiendo con los productos de Corporación de la Valve y HTC, Microsoft, así como Sony Computer Entertainment. Estos pesos pesados están seguros de ser seguido por muchas otras empresas, el mercado debe despegar como se esperaba.

2.2. Conceptos básicos de la realidad virtual

Estado del arte: Conceptos básicos de la realidad virtual

Los dos siguientes conceptos son necesarios para entender el funcionamiento del sistema que se va a desarrollar [2].

2.2.1. Vista estereoscópica

La visualización estereoscópica es un componente que separa la realidad virtual de la mayoría de los sistemas no VR. Las pantallas estereoscópicas presentan una visión diferente de la escena virtual para cada ojo, de la misma manera que los auriculares estéreo desempeñan diferentes sonidos para cada oído. La estereoscopia es una poderosa señal al cerebro de que ciertos objetos están más lejos que otros. En combinación con otras señales de profundidad, tales como el paralaje (objetos en la distancia parecen moverse más lento que objetos de cerca), líneas convergentes, y el sombreado, una pantalla estereoscópica se pueden emplear para crear efectivamente una sensación de presencia.

2.2.2. Head tracking

Significa «seguimiento de la cabeza». Quiere decir que, cuando se usa un dispositivo de realidad virtual, la imagen delante del usuario cambia cuando se mira hacia arriba, abajo y de lado a lado o el ángulo de la cabeza. Un sistema llamado 6GdL (seis grados de libertad) representa la cabeza en términos de sus ejes X, Y y Z para medir los movimientos de la cabeza hacia adelante y hacia atrás, de lado a lado y hombro con hombro. A estos atributos se les llama Yaw, Pitch y Roll (figura 2.2).

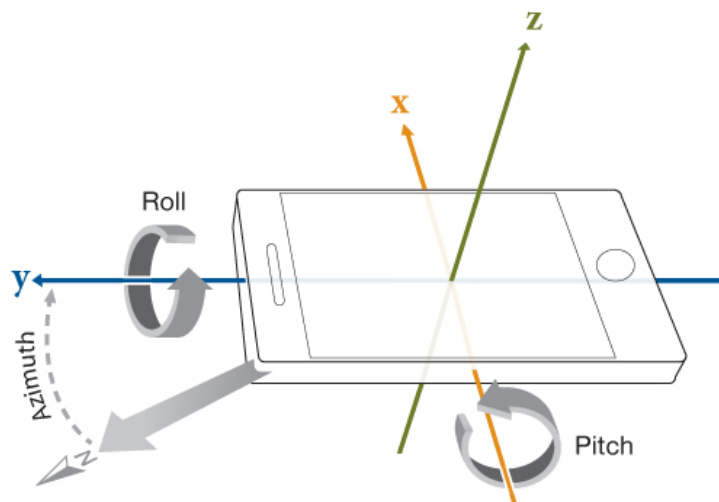


Figura 2.2: **Coordenadas polares.** Azimuth o Yaw (Ψ), Pitch (Φ), Roll (θ)

3

Análisis, arquitectura y diseño del sistema

3.1. Análisis

3.1.1. Descripción del producto

Google Cardboard Robotic Avatar es un producto que permite al usuario ver a través de los ojos de un avatar. Esto se consigue gracias a unas gafas de realidad virtual (figura 1.3) conectadas por Internet a un robot con dos cámaras por ojos. Los movimientos de la cabeza del usuario serán también transmitidos a través de la red hasta llegar al robot, que ejecuta un programa que recibe estos datos y actualiza la orientación de las cámaras en tiempo real para dar la sensación al usuario final de estar en otro cuerpo.

3.1.2. Objetivos y funcionalidad

El objetivo principal del proyecto es el de desarrollar un sistema con los siguientes tres elementos:

1. Dos **servidores de vídeo** que reciban como entrada un dispositivo cada uno (en este caso, una cámara por servidor) y como salida ofrezcan un *stream* multimedia.
2. Un **servidor de control** que reciba el posicionamiento del dispositivo de realidad virtual y asigne un valor analógico a los actuadores a partir de las coordenadas, que serán los motores sobre los que se encuentra la estructura en la que reposen las cámaras.
3. Una **aplicación para dispositivo smartphone** que establezca la conexión con los servidores. Recibirá como entrada dos *streams* de vídeo (uno por cada ojo) y como salida enviará la rotación respecto de los tres ejes dimensionales.

Sólo existe un tipo de usuario de la aplicación y, en un principio, su interacción se limitará a la siguiente:

- Alterar la posición física de las gafas de realidad virtual girando la cabeza.

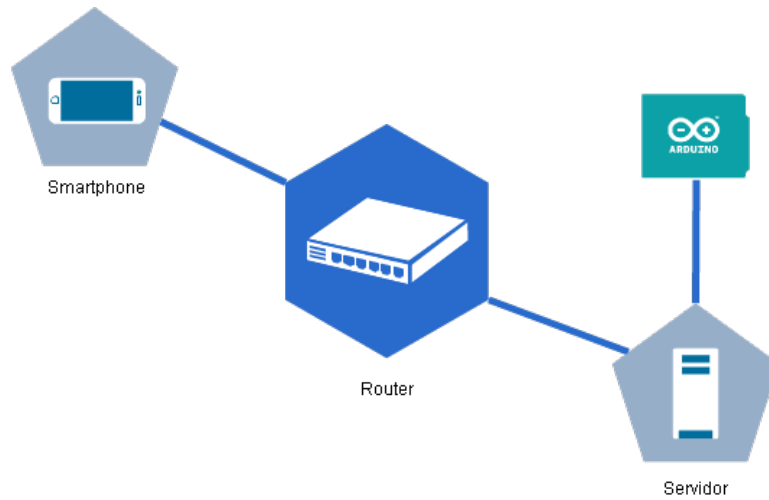


Figura 3.1: Diagrama conceptual de la red del sistema

- Modificar la dirección IP y los puertos de sendos servidores a través de la pantalla táctil del smartphone.

El sistema no debería necesitar ningún tipo de persistencia. Las conexiones se realizarían en modo *stateless*, lo que quiere decir el cliente se conecta y desconecta de los servidores arbitrariamente sin que deban almacenarse datos de sesión algunos.

3.1.3. Requisitos

1. Las imágenes se transmiten y se muestran con baja latencia.
2. Los distintos objetos comunes en las dos imágenes se pueden apreciar en tres dimensiones.
3. Los consecutivos movimientos de la cabeza son transmitidos secuencialmente y con baja latencia.
4. El usuario puede modificar la dirección IP de cada servidor.
5. El usuario puede modificar el puerto de cada servidor.

3.1.4. Tamaño y rendimiento

Esta arquitectura del software está orientada a permitir exclusivamente una conexión simultánea. Al tratarse de una aplicación en tiempo real, más conexiones podrían congestionar el tráfico en la red y saturar los recursos de los servidores. Además, la rotación de los motores sólo debería controlarse desde un solo cliente.

3.2. Arquitectura

3.2.1. Vista estática

El cliente es el componente central de este sistema heterogéneo. Éste está compuesto a su vez de otros tres componentes que se ejecutan como hilos independientes en la app. Uno de

ellos envía periódicamente la orientación del dispositivo al servidor de control en datagramas a través del protocolo de transporte UDP. Los otros dos reciben un stream de vídeo cada uno, procedentes de los servidores de vídeo a través del protocolo de aplicación RTSP. El servidor de control está conectado a su vez con el Arduino por medio de USB, al igual que las cámaras están conectadas a la máquina en la que se ejecutan los servidores de vídeos también por USB. Estas relaciones entre los componentes se pueden visualizar en la figura 3.2

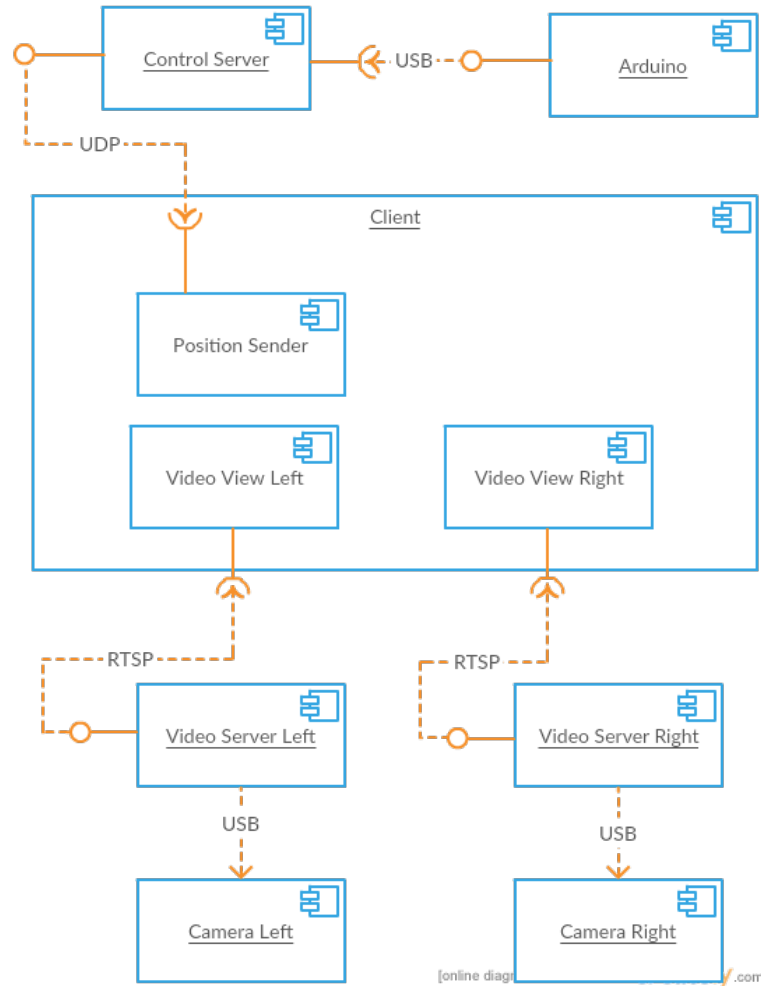


Figura 3.2: **Diagrama de componentes del sistema.** Se pueden apreciar los distintos componentes, las interfaces que ofrecen y requieren y el protocolo que utilizan para comunicarse entre sí.

3.2.2. Vista funcional

Servidor de vídeo

La funcionalidad de este componente dentro del sistema es la de proveer al cliente de dos flujos continuos de vídeo. Estos deben poder ser referenciados y accedidos a través de una dirección IP y de un puerto, bien dentro de la red local o bien a través de Internet.

Servidor de control

Este servidor también debe poder referenciarse del mismo modo que el anterior. Recibirá constantemente la posición desde el cliente que transformará en instrucciones para mover la estructura sobre la que se encuentren las cámaras.

Cliente móvil

Finalmente, el cliente se integra junto a estas partes anteriores de la siguiente forma: por un lado, ha de poder conectarse a los dos servidores de vídeo a través de sus direcciones para recibir sendos flujos de fotogramas que se mostrarán a través de la pantalla; por otro lado, ha de enviar continuamente sus coordenadas polares al servidor de control para que el Arduino pueda hacer girar el motor consecuentemente.

3.2.3. Vista dinámica

Prácticamente toda la evolución del sistema a lo largo del tiempo puede observarse en el diagrama 3.3, correspondiente al cliente. Por otro lado, los servidores pueden estar esperando a una conexión (el cliente todavía no se ha conectado) o enviando/recibiendo un flujo de datos.

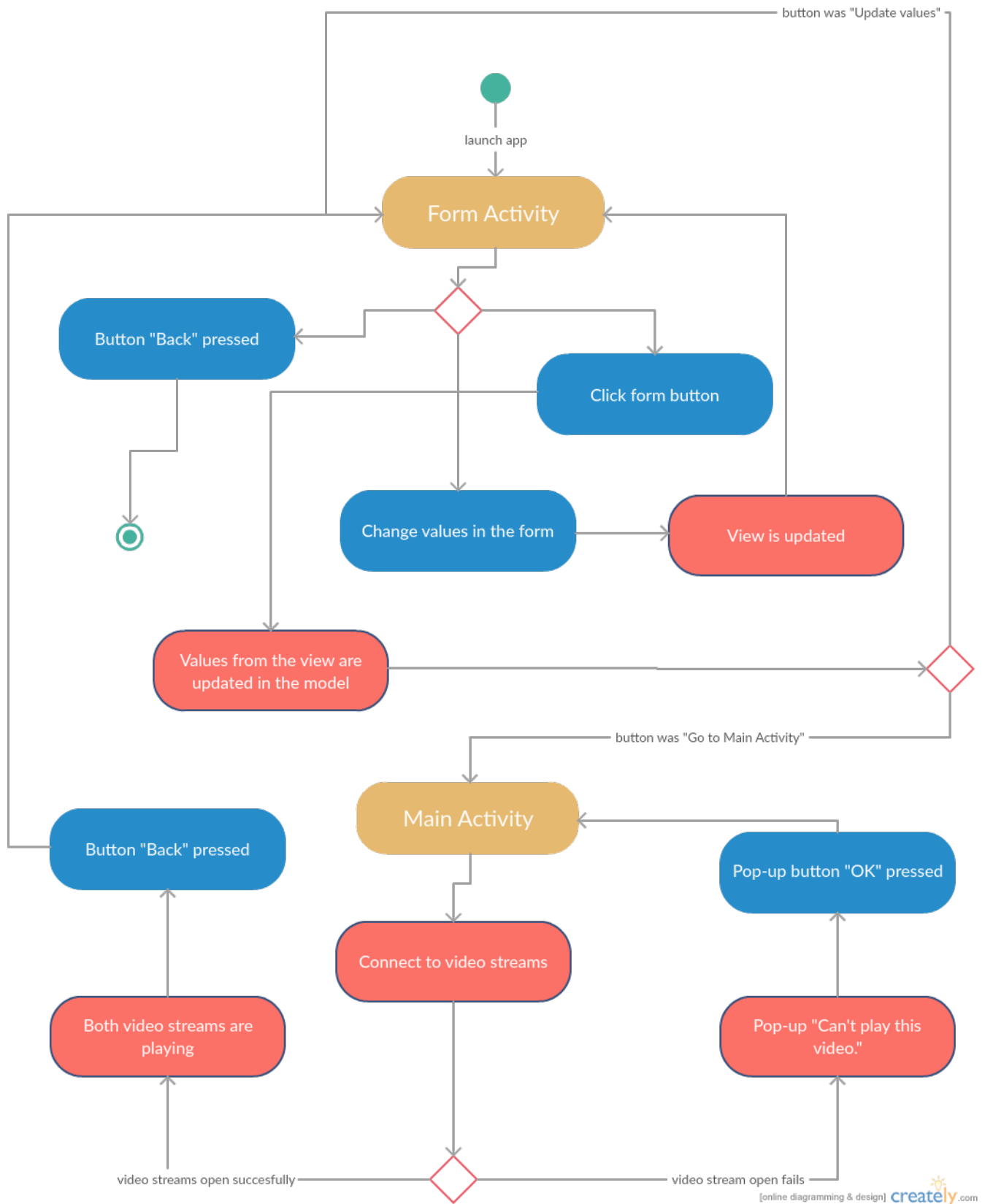


Figura 3.3: **Diagrama de actividad de la aplicación para Android.** En ocre, las vistas de la aplicación. En azul, las interacciones del usuario. En rojo, las respuestas del sistema.

3.3. Diseño

3.3.1. Servidor de vídeo

Como se explica en el apartado de implementación, esta parte del sistema no se ha llevado a cabo programando, sino utilizando una herramienta *open source*, por lo que no procede un desarrollo del diseño de los servidores de vídeo.

3.3.2. Servidor de control

El servidor de control está implementado en Python y son dos las clases más importantes, como se detalla en la figura 3.4:

- **ServoControl**: posee un objeto *Serial* correspondiente al puerto USB en el que está conectado el Arduino y un valor tipo *float* que representa el último ángulo procesado. El método *moveAngle* mueve desde el ángulo actual hasta el ángulo objetivo.
- **UDPServer**: guarda la dirección y el puerto en la cual será desplegado. El método *bind()* solicita al sistema operativo escuchar los paquetes de la interfaz de red que lleguen a cierta IP y puerto y *recvfrom()* lee datos del socket con un cierto tamaño de buffer (*bufferSize*).

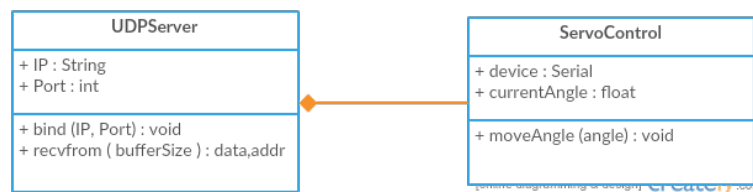


Figura 3.4: Diagrama de clases del servidor de control

3.3.3. Cliente móvil

El cliente consta principalmente de cuatro clases Java. Dos de ellas se corresponden con las dos vistas de la aplicación. Para representar esto, las clases han de extender la funcionalidad de *Android.Activity*. Estas dos han de tener acceso al *Singleton* *AddressManager*, que se encarga de gestionar las direcciones y puertos de los servidores. Por último, la clase *MainActivity* debe instanciar a la clase *PositionSender*, que se encarga de establecer un canal de comunicación UDP con el servidor de control y de enviar periódicamente un mensaje con la orientación del dispositivo móvil, obtenida a través de los sensores.

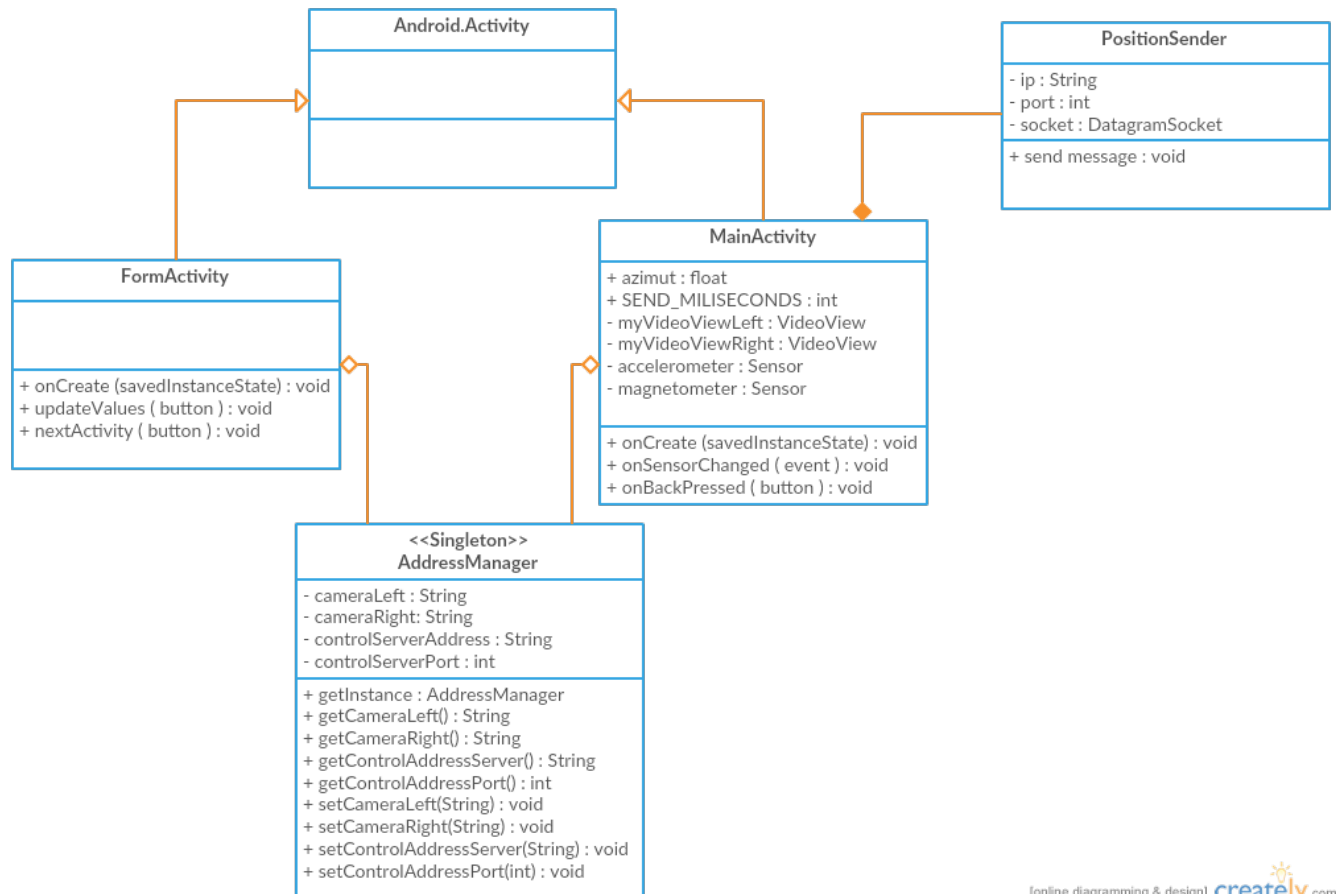


Figura 3.5: Diagrama de clases de la aplicación para Android

4

Tecnología y pruebas

4.1. Calidad de servicio

Para medir la *QoS*, se han hecho pruebas combinando distintos clientes, distintos servidores y distintos interfaces de red con el fin de hallar la mejor combinación válida para nuestro sistema.

4.1.1. Servidores

Las distintas herraminetas que se han contemplado para desplegar los servidores multimedia son las siguientes: **FFMpeg**, **FFServer**, **GStreamer** y **VLC (cvlc)**. Todas éstas se ejecutan, de manera similar, a través de la terminal de Linux y los parámetros de los streams se especifican como argumentos de los correspondientes ejecutables.

4.1.2. Clientes

Se han hecho pruebas con varios clientes para poder distinguir cuándo existía un *delay* de los *streams* como consecuencia de una congestión en la red o cuándo este retraso estaba causado por un elevado tiempo de procesamiento en la parte del cliente.

- **FPlay**: se ejecuta a través de la terminal y, aparte de mostrar el *stream*, muestra por la terminal datos acerca de éste.
- **VLC**: este famoso reproductor de multimedia también puede usarse para abrir contenido a través de una URL.
- **RTSP Player**: Es una *app* para Android que se conecta a un vídeo RTSP a través de su enlace.
- **Programa con JavaCV**: JavaCV utiliza wrappers para diversas librerías de visión por computador. Este programa de elaboración propia conecta con los servidores multimedia a través de la URL y los muestra en dos ventanas, una al lado de la otra. La idea era poder portar este código a la aplicación de Android, utilizando el archivo .jar para ARM en vez del archivo para amd64. Al probarlo, surgía un error inidentificable al abrir los *streams* y, tras bastante esfuerzo intentando solventarlo sin resultado, hubo que buscar otra solución.

4.1.3. Interfaces

- **localhost**: Pseudónimo de la dirección IP 127.0.0.1, se usa para referirse a la propia máquina. Para pruebas en las que el cliente y el servidor se encuentran ejecutándose en el mismo ordenador. La pérdida de paquetes es ínfima en este escenario.
- **Multicast Wi-Fi**: La única forma de desplegar el servidor RTSP utilizando un interfaz de red *wireless* y que pudiese ser accedido desde otras máquinas era utilizando una dirección IP multicast.
- **Unicast RJ-45**: Finalmente, el cableado a través de un cable RJ-45 al router era la mejor opción (como era de esperar). Ofrece un MTU mayor, mayor ancho de banda y menos pérdida de paquetes. Además, permite desplegar el servidor en la dirección IP correspondiente a la interfaz de red (*eth0*), lo cual hace que los servidores multimedia sean unicast, lo que quiere decir que sólo se permite una conexión a la vez.

4.1.4. Pruebas significativas

En el cuadro 4.1 pueden encontrarse las pruebas más significativas de todas las llevadas a cabo, en las cuales se comentan las prestaciones que ofrece la configuración dada.

4.1.5. Configuración final

Después de llevar a cabo este conjunto de pruebas de la calidad de servicio, la configuración del prototipo es la siguiente:

1. **Cliente**: los dos clientes de vídeo utilizados en la aplicación son los proporcionados por el SDK de Android a través de la clase **VideoView**.
2. **Servidor** de vídeo: se despliega utilizando la herramienta **clvc**, ya que permite al cliente abrir el *stream* a partir de una URL sin necesidad de un archivo SDP. Se configura en modo Unicast, asociándolo a la dirección IP de la tarjeta de red cableada por RJ-45.

Esta configuración tiene un *delay* total aproximado de 4 a 6 segundos, de los cuales 2 eran propios del servidor RTSP desplegado con **clvc**, siendo el resto retraso de proceso en el cliente.

Cuadro 4.1: Pruebas significativas de la calidad de servicio

Servidor	Cliente	Interfaz de red	Resultados
cvlc	vlc	localhost	La imagen se muestra con un delay de 2 segundos y sin apenas perder paquetes con el códec H264.
cvlc	vlc	multicast Wi-Fi	Se pierden muchos paquetes y la calidad de la imagen disminuye drásticamente. Queda descartado desplegar el servidor en una IP de la interfaz Wi-Fi y el modo multicast, ya que el objetivo es el de tener una conexión simultánea únicamente.
cvlc	vlc	unicast RJ-45	La calidad y prestaciones son muy parecidas a las mismas pruebas en local.
ffmpeg, ffserver, gstreamer	fplay	localhost	Estos tres servidores han de ser accedidos a través de un fichero SDP. Generándolo y pasándolo como argumento a fplay, se consiguen reproducir sin problemas los tres streams.
ffmpeg, ffserver, gstreamer	Programa con JavaCV (Linux)	localhost	Indicando la ruta del archivo SDP, el objeto FFMpegGrabber consigue abrir los streams. Falla cuando se intenta acceder a través de la URL
ffmpeg, ffserver, gstreamer	Programa con JavaCV (Android)	unicast RJ-45	Los recursos en Android han de ser abiertos a través de un descriptor de fichero (constante estática de la clase R), pero el constructor de FFMpegGrabber requiere una ruta en el sistema de archivos o una URL en un String o un objeto File.
cvlc	Programa con JavaCV (Android)	unicast RJ-45	La herramienta cvlc despliega un servidor RTSP que puede ser accedido a través de una URL, pero ocurre un error desconocido sin identificar en el constructor de FFMpegGrabber.

4.2. Cámaras

4.2.1. Cámaras con objetivo ojo de pez

También conocidas como *fish-eye lens*, las lentes de estas cámaras proporcionan un ángulo de visión superior al resto de tipos de lentes a cambio de una distorsión visual severa. Una vez montado todo el sistema, se ha probado a ejecutar la aplicación con las gafas de realidad virtual y conectándose a los dos servidores de vídeos alimentados por fotogramas de estas cámaras. El cerebro parece no estar preparado para sentir la profundidad a partir de dos imágenes de este tipo y, a pesar de las prestaciones que podrían haber ofrecido, las cámaras con objetivo de ojo de pez no han resultado idóneas para este proyecto.

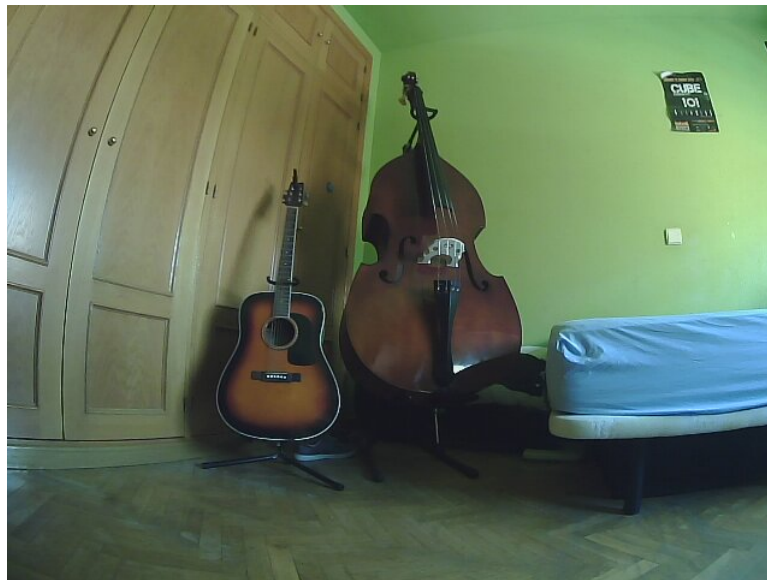


Figura 4.1: **Captura de cámara con lente *fish-eye*.** Esta cámara de alta definición obtiene un gran ángulo de visión a coste de la marcada curvatura de la imagen.

4.2.2. Cámaras con lente normal

Al final, unas webcam normales adquiridas en una tienda común de electrónica han resultado las más adecuadas. Ya que no poseen esta curvatura tan pronunciada en las imágenes, es posible captar la profundidad ya que las equivalencias entre ambas imágenes son lineales.



Figura 4.2: **Captura de cámara con lente normal.** Tomada a la misma distancia. La cámara de por sí tiene menos resolución, lo cual afecta directamente a la calidad de la imagen y la lente le da *zoom* a la captura, renunciando a un amplio ángulo de visión pero sin sufrir deformaciones.

4.3. Servomotor

Los servomotores son capaces de girar en sentido horario o antihorario con mayor o menor velocidad angular. Esto se indica mediante una señal analógica de entrada, la cual producimos en este caso mediante los PWM de la placa Arduino. En materia de código, este valor se indica con un byte sin signo, es decir, un número natural en el rango $[0, 255]$, como se explica en [3]. Así pues, después de calibrar el motor con un destornillador se ha observado el comportamiento del servomotor para distintos valores analógicos. En el cuadro 4.2 se encuentran estos resultados que posteriormente son utilizados por una clase que se encarga de enviar al Arduino los valores, según los cálculos que se detallan en el apartado de implementación, necesarios para hacer girar el motor.

Cuadro 4.2: **Relación aproximada entre el desplazamiento angular y el valor analógico de entrada del servomotor**

Incremento del ángulo ($^{\circ}$)	Valores para desplazar el servomotor a la izquierda	Valores para desplazar el servomotor a la derecha
0	90	90
45	79	99
90	73	104
135	73, 79	104, 99
180	60	117

5

Implementación

5.1. Cliente

5.1.1. Fundamentos de programación para Android con Android Studio

A continuación se explican conceptos añadidos a la programación en *Java* y básicos para entender la implementación en aplicaciones para plataformas Android, desarrollados con más detalle en [4].

- **Recursos y *R***: *R* es una clase estática existente a través de generación automática de código que asigna un número entero a los distintos recursos del proyecto (archivos de texto, imágenes, audio...). Estos recursos pueden ser referenciados desde el código *Java* mediante este identificador, que se encuentra en la clase *R* como un tipo de dato *public static int*.
- ***Activity***: la clase *Activity* se corresponde con una vista con una interfaz gráfica, un modelo con atributos de esta clase y controladores, correspondientes a los métodos. Para crear una nueva vista de la aplicación, se suele crear una clase que herede de ésta (*extends*).
- **Interfaces *handler***: en el sistema operativo Android y en la propia aplicación ocurren distintos tipos de eventos que pueden ser más o menos interesantes dependiendo de los requisitos del programa. Para poder escuchar esos eventos y hacer algo cuando ocurren, se suelen implementar interfaces Java dentro de nuestra actividad. Así, cuando el SO notifique a la actividad de una de estas señales, se ejecutará un método *handler*, cuyo nombre se indica en la interfaz implementada asociada a este suceso.
- ***Layout***: el *layout* es la distribución de los distintos elementos gráficos en la pantalla. Ésta puede indicarse de dos formas: programáticamente en alguna función de la actividad a través de diversas clases, forma que no suele ser ni habitual ni conveniente, o bien a través de un fichero XML. El SDK de Android ofrece la funcionalidad de cargar la distribución a partir del identificador de recurso de este archivo, creando internamente las clases que gestionan la distribución en la pantalla y asociándolas con la actividad.

5.1.2. Aplicación

Está compuesta por 4 clases escritas en Java, de las cuales 2 extienden a la clase `Android.Activity` y las otras 2 son clases auxiliares.

Gestor de direcciones

Esta clase implementa el patrón de diseño *Singleton* (figura A.1). Forma parte del modelo de la aplicación y almacena las direcciones y puertos de los servidores. Permite leer y escribir estos datos (figura A.2) independientemente de la actividad en la que se encuentre el usuario a través de sus métodos de acceso públicos (*getters* y *setters*).

Transmisor de posición

Esta sencilla clase auxiliar está asociada a un socket UDP que se crea en el constructor de esta misma, asociado a su vez a un puerto y una dirección IP determinadas (figura A.4). Encapsula las operaciones básicas con sockets, haciendo más sencilla la comunicación con el servidor de control a nivel de programación (figura A.5).

Formulario

Esta actividad se muestra al iniciar la app. Al iniciarse, carga su layout a partir del fichero *form.xml* y guarda la referencia a la instancia única del gestor de direcciones (figura A.6). Las direcciones de sendos servidores multimedia con los streamings de vídeo han de ser indicadas mediante la URI completa, mientras que el servidor de control ha de indicarse introduciendo la dirección IP de la máquina en la que se encuentra alojado el servidor y el puerto por separado. Los valores que se muestran en la figura B.9 son los valores por defecto de estos *textboxes* que coinciden con las direcciones en las que se despliegan los servidores en el entorno de trabajo para agilizar así las pruebas.

En el archivo XML asociado a esta actividad, también se indica el nombre de los *handlers* de los botones; es decir, la función que se ejecutará al clickar sobre cada uno de estos dos elementos. Así pues, esta clase debe implementarlos. Al presionar el botón «Update values», esta clase accede al valor de todos los *textboxes* y los almacena en el gestor anteriormente explicado (figura A.7). Al presionar el segundo botón, se abandona esta actividad y se inicia la aplicación principal, *MainActivity* (figura A.8).

Actividad principal

Como en la actividad del formulario, al crearse se guarda la referencia al *Singleton* y se carga el *layout* a partir de su correspondiente archivo XML. En éste se indica que habrá dos vistas, correspondientes a sendos *streams* de vídeo, ocupando la pantalla completa mitad y mitad.

- Seguidamente, se crean dos reproductores de vídeo que se situarán en estas dos vistas. En los constructores de estos objetos se pasan como parámetros las URI de los dos *streams*, lo que hará que los vídeos se reproduzcan tan pronto como estos dos recursos estén disponibles (figura A.9).
- Se obtienen las referencias a los sensores a través del sistema operativo, mediante los cuales se calcularán las rotaciones del dispositivo móvil. Estos son el acelerómetro y la brújula.

- Se instancia un objeto de la clase `Sender` a partir de la IP y el puerto del servidor de control almacenados en el *Singleton* (figura A.10)
- Se crea el hilo que se encargue de enviar constantemente la posición, como se muestra en la figura 2.2). A priori, la solución más obvia es crear una clase que implemente la interfaz *Runnable* y empezar un hilo mediante la clase *Thread*. Tras este intento, el sistema operativo elimina este hilo porque, al parecer, consume mucha CPU, lo cual no permite a la actividad principal refrescar la interfaz gráfica. Es decir, genera inanición. Como solución y después de investigar este contratiempo, encontré la clase *Handler* con su método de instancia *postdelayed*. Esencialmente, se inicializa este *Handler* con un objeto que implemente *Runnable* y *postdelayed* crea una alarma que se activará después de un determinado número de milisegundos pasado como parámetro. Al activarse, se ejecute el método *run()* de este handler. Con esto, creamos un objeto *Runnable* que ejecute una vez el cuerpo del bucle y finalmente vuelva a montar la alarma, como si se tratase de una recursión infinita (figura A.11).

5.2. Servidores de vídeo ---

5.2.1. Configuración del servidor

Para los servidores de vídeo, finalmente se ha utilizado el comando **cvlc**, herramienta de VLC para la terminal. Ésta ofrece una inmensidad de servicios, entre los cuales no interesa la posibilidad de desplegar servidores multimedia. La configuración de estos se indica junto con este comando en una cadena de caracteres que vendrá a definir el *pipeline* que se ejecutará para dicho servidor. Un pipeline consta, en resumen, de estos elementos:

1. Entrada (*Input Source*)
2. Operaciones intermedias (*Transcode*)
3. Salida (*Output Stream*)

Como entrada, se especifica la videocámara por su ruta dentro del sistema de ficheros y se abre en este caso con V4L2 que es una API de captura de vídeo y está integrada en el kernel de linux.

Con esta fuente, se forma el vídeo en sí. El códec de vídeo elegido ha sido el **H264**. Este formato tiene decenas y decenas de variables y parámetros. Ya que el objetivo de este proyecto no era el de realizar una tarea de optimización tan intensa, se ha recurrido a lo que se llaman *presets*, que dan valor al conjunto de parámetros del códec para ofrecer una determinada calidad y rendimiento. Se ha configurado de forma que sea lo más rápido posible y que tenga una menor latencia. Finalmente, se indica la resolución de salida para que se corresponda justo lo que ocupará en la aplicación de móvil (la mitad de la resolución de la pantalla táctil) y también se indican los FPS.

Finalmente, se indica que la salida del pipeline será un servidor RTSP, en el cual se enviarán los datos a través de RTP y cuyos metadatos de sesión se guardarán en un archivo SDP, que se encontrará en la dirección física de la propia máquina en la que se ejecute el script y en el puerto indicado.

El código completo del script se encuentra en la figura A.14

5.3. Servidor de control

5.3.1. Circuito y programación de la placa Arduino

El circuito para controlar el servomotor con nuestra placa es trivial, como se muestra en la figura 5.1. Se ha utilizado la librería [5] para controlarlo, que se carga en la memoria de la placa junto al siguiente programa codificado.

El lenguaje que utiliza el IDE de Arduino es un dialecto de C++. Un programa consta obligatoriamente de dos partes y de tantas variables y funciones auxiliares como puedan caber en la memoria de la placa. La primera parte, la función *setup()*, se ejecuta cuando ésta se conecta. La segunda, la función *loop()*, se invoca inmediatamente después y se ejecuta dentro de un bucle infinito.

- ***myRead()***: función auxiliar. Lee caracter a caracter del objeto Serial. Cuando lee el caracter '\n', convierte la cadena a un número entero y lo devuelve.
- ***setup()***: abrimos el puerto serial para recibir datos e inicializamos el objeto que se encarga de controlar el servomotor, asociándolo al GPIO 3 (PWM)
- ***loop()***: utiliza la función *myRead()* y pasa el valor leído al método *VarSpeedServo.write*, que escribe este valor en el PWM asociado al servomotor. Después, espera 500 ms y vuelve a empezar.

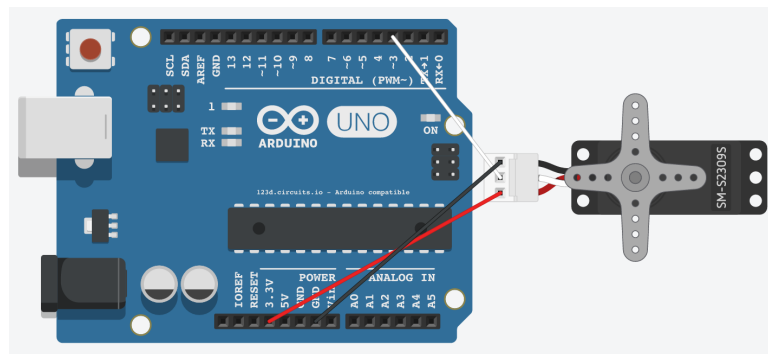


Figura 5.1: **Circuito del Arduino controlando el servomotor.** El cable rojo va a VCC (3.3V) de la placa arduino, el cable negro a GND (tierra) y el cable blanco al PWM correspondiente. Variando el valor de la señal analógica del PWM, conseguimos cambiar de sentido y de intensidad.

5.3.2. Cliente/servidor UDP

UDP es el protocolo de transporte utilizado para implementar el servidor de control. UDP es un protocolo no fiable, ya que no asegura que los paquetes lleguen a su destino ni que lleguen en orden, lo cual hace que las cabeceras sean más ligeras y, por tanto, pueda dedicarse mayor caudal del ancho de banda al contenido que se transporta. También es más sencillo que TCP en el sentido de que no necesita establecer una conexión. El servidor debe vincularse (bind) a una dirección IP y un puerto, y posteriormente los clientes pueden enviar y recibir de esta dirección sin necesidad de llevar a cabo un *handshake*.

UDP es idóneo para aplicaciones en tiempo real, ya que no nos interesa reenviar paquetes, puesto que los mensajes correspondientes a la posición del usuario sólo son válidos en el instante en el que se envía.

5.3.3. Programación del servidor

Para la implementación de este servidor, se ha elegido Python 2.7 como lenguaje de programación por los siguientes motivos:

- Es un lenguaje de muy alto nivel, potente, rápido de programar y fácil de depurar.
- El módulo **socket** proporciona un manejo de bajo nivel de los propios sockets del sistema operativo que en realidad encapsula estas funciones propias de C/C++ en UNIX.
- El módulo **serial** permite al programador acceder a cierto puerto serial para efectuar operaciones de lectura y escritura abstrayéndose del protocolo USB a bajo nivel.

ServoControl

El constructor de esta clase intenta abrir el puerto serial donde ha de encontrarse conectado el Arduino y lanza una excepción cuando la apertura falla. Después, simplemente se invoca al método *moveAngle()* que recibe un ángulo como parámetro. Una instancia de esta clase siempre guarda internamente el último ángulo recibido a través de este método (inicializándolo en 0 grados) y el ángulo que debe desplazarse se calcula mediante la diferencia con esta referencia. Es decir, el incremento del ángulo es relativo al ángulo inmediatamente anterior. Ya que el tipo de motor no es preciso a la hora de desplazar un ángulo en concreto, se ha tomado la siguiente decisión de implementación: se ha creado un diccionario en el cual las claves son un ángulo en concreto, múltiplo de 45 y de -180 a 180. Los valores, son el valor analógico que hay que pasarle al arduino para que gire, aproximadamente, esa cantidad de grados. Por tanto, el valor que se enviará a través del puerto serial será el valor para la clave que más se aproxime a la diferencia entre el nuevo ángulo y el ángulo anterior (es decir, el incremento).

UDP server

Éste es el programa que se encarga de realizar todas las tareas necesarias para desplegar el servidor de control en la máquina en la que se ejecute.

- Al ejecutarse el proceso, se crea un socket UDP.
- Se obtienen la dirección IP y el puerto que son pasados como argumentos al ejecutar el programa.
- Se vincula el socket a esta dirección a través del método *bind()*
- Se intenta crear una instancia de *ServoController*. Si hubiese un fallo, termina la aplicación indicando el error que lo produjo.
- Comienza el propio bucle del servidor:
 1. Recibe de forma bloqueante del socket y muestra los datos recibidos.
 2. Parsea el mensaje para convertirlo de una cadena de caracteres a un número en coma flotante.
 3. Llama al método *moveAngle()* de la instancia de *ServoController* para realizar el movimiento angular

6

Conclusiones y trabajo futuro

6.1. Conclusiones

Personalmente y como logro propio, se ha llevado a cabo el desarrollo de un sistema distribuido heterógeno, integrando tecnologías punteras y variadas como culmen de mis estudios en Grado en Ingeniería Informática.

El mundo de las redes multimedia y la transmisión de vídeo y audio tiene muchos detalles. A pesar de que no ha sido complejo integrar estas herramientas para desplegar un servidor de vídeo, las implementaciones de los protocolos a nivel de aplicación de transmisión de multimedia (RTSP en este caso) son complejas a bajo nivel, así como los códecs, los formatos y todos sus parámetros, variables y metadatos, lo cual tendría que tomarse en cuenta en consiguientes fases de optimización.

Por otro lado y como se ha comentado previamente, la realidad virtual se encuentra en auge, motivo por el cual han surgido y surgen nuevas librerías, herramientas, *frameworks*, etc. orientados a la creación de programas y aplicaciones de VR. Al ser tan nuevas, todavía necesitan madurar arreglando fallos y bugs, documentando mejor las APIs e implementando más funcionalidad necesaria y/o útil para el desarrollo de este tipo de software. Aún con esto, es el momento de desarrollar software de VR, ya sean aplicaciones para Cardboard o programas (videojuegos o software más comercial) para computadores de sobremesa para las nuevas plataformas que se están empezando comercializar actualmente.

JavaCV era una opción en el lado del cliente para manipular los *streams* y los frames obtenidos de estos, pero por un fallo de implementación del .jar para la arquitectura *arm* se tuvo que renunciar a todo este potencial. Aún así, OpenCV y sus versiones para *x86* y *amd64* serían una buena opción para un procesamiento previo al envío u otro proyecto relacionado.

6.2. Trabajo futuro

Se ha llevado a cabo un prototipo con la funcionalidad básica del producto descrito en un principio. A continuación se enumeran las distintas tareas que se podrían llevar a cabo para obtener una versión mejorada de éste.

- Tareas de optimización de la QoS, a saber:
 1. Desarrollar una **vista** para Android (`android.view.View`) **específica para streams RTSP**, que ofrezca mayores prestaciones tales como menor delay, mantener la sesión RTSP, sincronización de los dos flujos de vídeo, etc.
- Utilizar la API Google VR for Android [6]. Hasta hace unos meses, se llamaba Cardboard API. Actualmente también incluye soporte para DayDream VR [7], cuyo lanzamiento tendrá lugar en otoño del 2016, y está más documentada, con más ejemplos y más funcionalidad. Los vídeos se mostrarían sobre una textura de OpenGL como en la figura 1.1
- Implementar la rotación sobre los ejes X e Y (Pitch y Roll respectivamente, figura 2.2).
- **Utilizar motores paso a paso o *steppers*** para mover la estructura sobre la que se encuentren las cámaras, ya que estos permiten girar un determinado arco con precisión, dependiendo de las especificaciones del componente.
- Modelar mediante un editor de 3D y posteriormente imprimir un exoesqueleto en el que puedan acomodarse las cámaras y que proteja los motores y circuitos de los golpes, el calor, etc.
- Añadir un stream más que se corresponda con el audio, grabado en el servidor a través de un micrófono cualquiera.
- Alojar la estructura con las cámaras y los motores sobre otro robot que proporcione movilidad, como un vehículo teledirigido un dron.

Glosario de acrónimos

- **VR:** *Virtual Reality* (Realidad Virtual)
- **HMD:** *Head Mounted Display* (Gafas de Realidad Virtual)
- **POC:** *Proof Of Concept* (Prueba de concepto)
- **UDP:** *User Datagram Protocol*
- **RTSP:** *Real Time Streaming Protocol* (Protocolo de transmisión en tiempo real)
- **SDP:** *Session Description Protocol* (Protocolo de descripción de sesión)
- **URI:** *Uniform Resource Identifier* (Identificador de recursos uniforme)
- **MVC:** *Model View Controller* (Modelo Vista Controlador)
- **IP:** *Internet Protocol*
- **V4L2:** *Video4Linux v2*
- **FPS:** *Frames Per Second* (Fotogramas Por Segundo)
- **PWM:** *Pulse-Width Modulation*
- **QoS:** *Quality of Service* (Calidad de servicio)
- **API:** *Application Programming Interface*
- **GPIO:** *General Purpose Input Output*
- **SDK:** *Software Development Kit*
- **IDE:** *Integrated Development Environment*

Bibliografía

- [1] Eduardo Radío. Sarah Dobber. Borja Fourquet. The haptic eye: sistema de orientación para ciegos. <http://thehapticeye.com/>, 2015.
- [2] Robin Hollands. The virtual reality: Homebrewer's handbook. *ISBN 0 471 95871 9*, 1996.
- [3] Timothy Hirzel. Documentación de los pines pwm de la placa arduino. <https://www.arduino.cc/en/Tutorial/PWM>, 2016.
- [4] Kevin Grant and Chris Haseman. Beginning android programming. *ISBN-13: 978-0-321-95656-9, ISBN-10: 0-321-95656-7*, 2014.
- [5] Michael Margolis. Varspeedservo. <https://github.com/netlabtoolkit/VarSpeedServo>, 2009.
- [6] Google. Google vr sdk for android. https://developers.google.com/vr/android/reference_overview, 2016.
- [7] Google. Google vr: Daydream. <https://vr.google.com/daydream/>, 2016.



Fragmentos de código

A continuación se muestran los fragmentos de código más relevantes de cada uno de los componentes del sistema, con el fin de ayudar al lector a comprender las decisiones de implementación.

A.1. Cliente

A.1.1. AddressManager.java

```
private static AddressManager ourInstance = new AddressManager();

public static AddressManager getInstance() {
    return ourInstance;
}
```

Figura A.1: **Patrón de diseño *Singleton***. La única instancia de esta clase puede ser referenciada invocando al método público y estático *getInstance()*

```
private String cameraLeft = "";
private String cameraRight = "";
private String controlServerAddress = "";
private int controlServerPort = 0;
private boolean send = false;
```

Figura A.2: **Atributos de la instancia**. Estos son los datos que se almacenan en esta clase. Sus correspondientes *getters* y *setters* son de acceso público

A.1.2. PositionSender.java

```
public static final int BUFFER_LENGTH = 1024;

private String ip;
private int port;
InetAddress ipAdres;

private DatagramSocket socket;
private byte[] sendData;
```

Figura A.3: Atributos de la clase.

```
public PositionSender(String ip, int port) throws SocketException, UnknownHostException {
    this.ip = ip;
    this.port = port;
    sendData = new byte[BUFFER_LENGTH];
    socket = new DatagramSocket();
    ipAdres = InetAddress.getByName(ip);
}
```

Figura A.4: Método constructor.

```
public void send(String message) throws IOException {
    sendData = message.getBytes();
    DatagramPacket sendPacket =
        new DatagramPacket(sendData, sendData.length, ipAdres, port);
    socket.send(sendPacket);
}
```

Figura A.5: Método *send()*. Este método se encarga de enviar un mensaje, en forma de una cadena de caracteres, a través del socket UDP previamente inicializado en el constructor.

A.1.3. FormActivity.java

```
public class FormActivity extends Activity {

    AddressManager manager = AddressManager.getInstance();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.form);
        updateValues(null);
    }
}
```

Figura A.6: Definición de la clase, AddressManager y *onCreate()*.


```

public void updateValues(View button) {

    final EditText cameraLeftField =
        (EditText) findViewById(R.id.cameraLeft);
    String cameraLeft = cameraLeftField.getText().toString();

    final EditText cameraRightField =
        (EditText) findViewById(R.id.cameraRight);
    String cameraRight = cameraRightField.getText().toString();

    final EditText controlServerAddressField =
        (EditText) findViewById(R.id.controlServerAddress);
    String controlServerAddress = controlServerAddressField.getText().toString();

    final EditText controlServerPortField =
        (EditText) findViewById(R.id.controlServerPort);
    String controlServerPort = controlServerPortField.getText().toString();

    manager.setCameraLeft(cameraLeft);
    manager.setCameraRight(cameraRight);
    manager.setControlServerAddress(controlServerAddress);
    manager.setControlServerPort(Integer.parseInt(controlServerPort));

}

```

Figura A.7: Método *updateValues()*. Controlador del botón «Update values». Accede a los *EditText* de la vista y los almacena en el modelo (AddressManager)

```

public void nextActivity(View button) {
    Intent intent = new Intent(this, MainActivity.class);
    manager.setSend(true);
    finish();
    startActivity(intent);
}

```

Figura A.8: Método *nextActivity()*.

A.1.4. MainActivity.java

```

myVideoViewLeft = (VideoView)findViewById(R.id.myvideoviewLeft);

MediaController mc = new MediaController(this);
myVideoViewLeft.setMediaController(mc);
myVideoViewLeft.setVideoURI(Uri.parse(manager.getCameraLeft()));
myVideoViewLeft.requestFocus();
myVideoViewLeft.setOnPreparedListener(new MediaPlayer.OnPreparedListener() {
    public void onPrepared(MediaPlayer mp) {
        myVideoViewLeft.start();
    }
});
myVideoViewLeft.setOnCompletionListener(new MediaPlayer.OnCompletionListener() {
    @Override
    public void onCompletion(MediaPlayer mp) {
        myVideoViewLeft.stopPlayback();
        myVideoViewLeft.setVideoURI(Uri.parse(manager.getCameraLeft()));
        myVideoViewLeft.requestFocus();
        myVideoViewLeft.start();
    }
});

```

Figura A.9: Inicialización de objeto `VideoView`. El mismo proceso se realiza para `myVideoViewRight` y ambos comparten el mismo `MediaController`.

```

mSensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
accelerometer = mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
magnetometer = mSensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);

try {
    sender = new PositionSender(manager.getControlServerAddress(),
                                manager.getControlServerPort());
}
catch (Exception e) {}

```

Figura A.10: Acceso a los sensores y creación del objeto *PositionSender*:

```

final Handler handler2 = new Handler();
final Runnable r2 = new Runnable() {
    public void run() {
        try {
            String message = String.format("%.2f", Math.toDegrees(azimut));
            sender.send(message);
        } catch (IOException e) {
            e.printStackTrace();
        }
        if(manager.shouldSend())
            handler2.postDelayed(this, SEND_MILLISECONDS);
    }
};

```

Figura A.11: **Hilo de envío** de la posición al servidor de control

```

@Override
public void onSensorChanged(SensorEvent event) {

    if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER){
        System.arraycopy(event.values, 0, mGravity, 0, 3);
        mLastAccelerometerSet = true;
    }

    if (event.sensor.getType() == Sensor.TYPE_MAGNETIC_FIELD){
        System.arraycopy(event.values, 0, mGeomagnetic, 0, 3);
        mLastMagnetometerSet = true;
    }

    if (mLastAccelerometerSet && mLastMagnetometerSet) {
        float Rot[] = new float[9];
        float I[] = new float[9];

        boolean success =
            SensorManager.getRotationMatrix(Rot, null, mGravity, mGeomagnetic);
        if (success) {
            float orientation[] = new float[4];
            float outR[] = new float[9];
            SensorManager.remapCoordinateSystem(Rot,
                SensorManager.AXIS_X, SensorManager.AXIS_Z, outR);
            SensorManager.getOrientation(outR, orientation);
            azimut = orientation[0]; // orientation contains: azimut, pitch and roll
            pitch = orientation[1];
            roll = orientation[2];
        }
    }
}

```

Figura A.12: **Método *onSensorChanged()***: Realiza el cálculo de las coordenadas polares (figura 2.2) cada vez que los sensores cambian de valor.

```
@Override
public void onBackPressed() {
    Intent intent = new Intent(MainActivity.this, FormActivity.class);
    intent.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
    manager.setSend(false);
    finish();
    startActivity(intent);
}
```

Figura A.13: **Método *onBackPressed()***: se ejecuta cuando se presiona el botón de los dispositivos Android con forma de triángulo.

A.2. Servidores de vídeo

A.2.1. deploy.sh

```
#!/bin/bash

CAMERAS=(v4l2:///dev/video0 v4l2:///dev/video1)
RTP=("sdp=rtsp://:8554/" "sdp=rtsp://:8556/")
CODEC_PARAMS="preset=ultrafast,tune=zerolatency,intra-refresh,lookahead=10,keyint=15"
PARAMS="vcodec=h264,venc=x264{$CODEC_PARAMS}"
OUTPUT="width=640,height=720,fps=5"
TRANSCODE_0="#transcode{$PARAMS,$OUTPUT}:rtp${RTP[0]}"
TRANSCODE_1="#transcode{$PARAMS,$OUTPUT}:rtp${RTP[1]}"
cvlc -vvv ${CAMERAS[0]} --sout $TRANSCODE_0 >/dev/null 2>/dev/null &
cvlc -vvv ${CAMERAS[1]} --sout $TRANSCODE_1 >/dev/null 2>/dev/null &
```

Figura A.14: **Script de despliegue** de los servidores multimedia.

A.3. Servidor de control

A.3.1. servo_serial_read.ino

```
#include <VarSpeedServo.h>

VarSpeedServo myservo;  // create servo object to control a servo

const int INTERVAL = 500;
const int MIN_VALUE = 50;
const int MAX_VALUE = 150;
const int STILL_VALUE = 90;
```

Figura A.15: Cabecera del código del Arduino.

```
int myRead() {
  //return (int) Serial.read();
  String inString = "";
  while (Serial.available() > 0) {
    int inChar = Serial.read();
    if (isDigit(inChar)) {
      inString += (char)inChar;
    }
    if (inChar == '\n') {
      Serial.println(inString.toInt());
      int ret = inString.toInt();
      if (ret < MIN_VALUE || ret > MAX_VALUE) return STILL_VALUE;
      return ret;
    }
  }
  return STILL_VALUE;
  //return Serial.read();
}
```

Figura A.16: Función auxiliar *myRead()*. Lee carácter a carácter del puerto serial hasta encontrar el valor '\n' y convierte la cadena de caracteres obtenida en un valor numérico.

```
}

void setup() {
  Serial.begin(9600);
  while (!Serial) {
    ; // wait for serial port to connect. Needed for native USB port only
  }
  myservo.attach(3);  // attaches the servo on pin 3 to the servo object
}
```

Figura A.17: Función *setup()* de Arduino. Esta función se ejecuta al arrancarse la placa.

```

void loop() {
    int value = myRead();
    // myservo.write(value);
    myservo.write(value, 50, true);
    delay(INTERVAL);
}

```

Figura A.18: **Función *loop()* de Arduino.** Esta función se ejecuta indefinidamente después de ejecutarse *setup()*.

A.3.2. control_servo.py

```

device = "/dev/ttyACM0"
class ServoControl(object):
    """docstring for ServoControl"""
    currAngle = 0.0
    def __init__(self):
        super(ServoControl, self).__init__()
        self.ser = serial.Serial(device, 9600, timeout=None) # open first serial port

```

Figura A.19: **Definición de la clase y constructor de *ServoControl*.** El ángulo inicial es $\Psi_0 = 0^\circ$. El constructor crea un objeto *Serial* en el puerto 9600 a partir de su ruta en el sistema operativo.

```

def moveTo(self, angle):
    diffAngle = (angle - self.currAngle)
    if diffAngle < -180:
        diffAngle += 360
    elif diffAngle > 180:
        diffAngle -= 360
    nearestKey = self.getNearestKey(diffAngle)

    if nearestKey == None or nearestKey == 0:
        return

    print "Target angle: ", angle
    print "Current angle: ", self.currAngle
    print "NearestKey: ", nearestKey

    values = moveValues[nearestKey]
    for value in values:
        self.writeNum(value)

    self.currAngle += nearestKey
    self.currAngle = normalize_value(self.currAngle)

```

Figura A.20: **Definición de la clase y constructor de *ServoControl*.**

A.3.3. udp_server.py

```
UDPSock = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
listen_addr = (sys.argv[1], int(sys.argv[2]))
UDPSock.bind(listen_addr)

try:
    sc = ServoControl()
except Exception, e:
    print "Dispositivo no conectado"
    raise
```

Figura A.21: Inicialización de recursos del servidor de control.

```
try:
    while True:
        data,addr = UDPSock.recvfrom(1024)
        print data
        #print data.strip(),addr
        parsedAngle = int(float(data.replace(',','.')))
        sc.moveAngle(parsedAngle)
        #sc.showDebug()
except KeyboardInterrupt:
    sc.write(str(90) + "\n")
    sys.exit()
```

Figura A.22: Bucle del servidor de control.

B

Montaje y funcionamiento

B.1. Robot

Se encuentra en el lado del servidor y, funcionalmente, es una cabeza robótica con dos cámaras por ojos y con la capacidad de rotar sobre el eje Z (perpendicular al suelo).

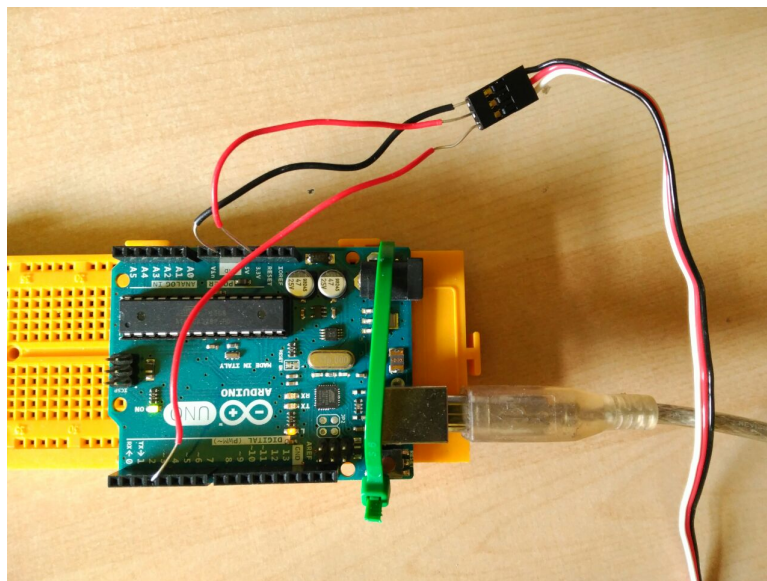


Figura B.1: **Circuito electrónico:** las conexiones a la placa arduino se corresponden con el diseño en de la figura 5.1. El cable conectado por la derecha de la placa se conecta al PC a través de cualquier puerto USB.

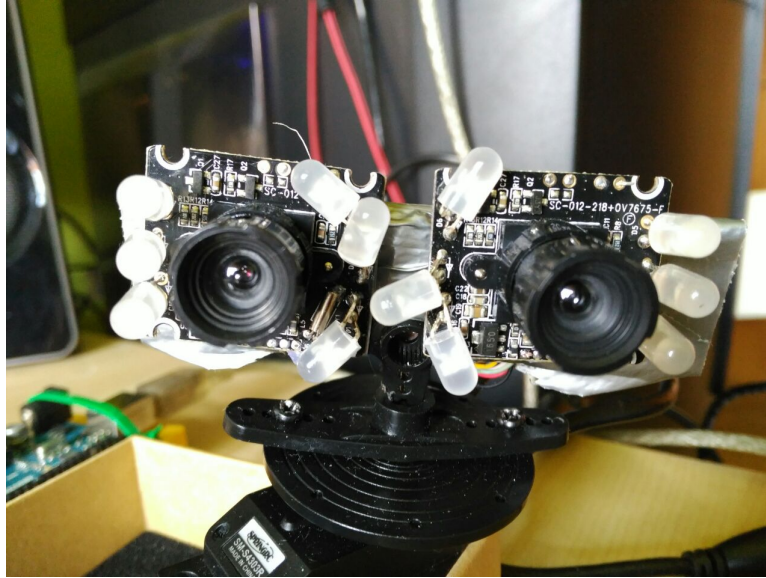


Figura B.2: **Videocámaras:** se encuentran unidas por una estructura de cartón fijada con cinta americana.



Figura B.3: **Videocámaras sobre el servomotor:** la estructura con las cámaras se coloca perpendicularmente sobre una estructura circular de un par de pulgadas de diámetro, que a su vez está enganchada al engranaje al cual hace girar el servomotor.

B.2. Gafas de Realidad Virtual

Las gafas de realidad virtual necesarias para este sistema no son más que una estructura física con dos lentes y un espacio donde introducir el móvil, de modo que el usuario pueda ver la pantalla del dispositivo a través de estas lentes.



Figura B.4: Google Cardboard original:



Figura B.5: **Gafas de realidad virtual de plástico:** estas gafas son más ergonómicas, ya que tiene gomaespuma en las zonas que hacen contacto con la cara y la cinta elástica se ajusta mejor a la cabeza.

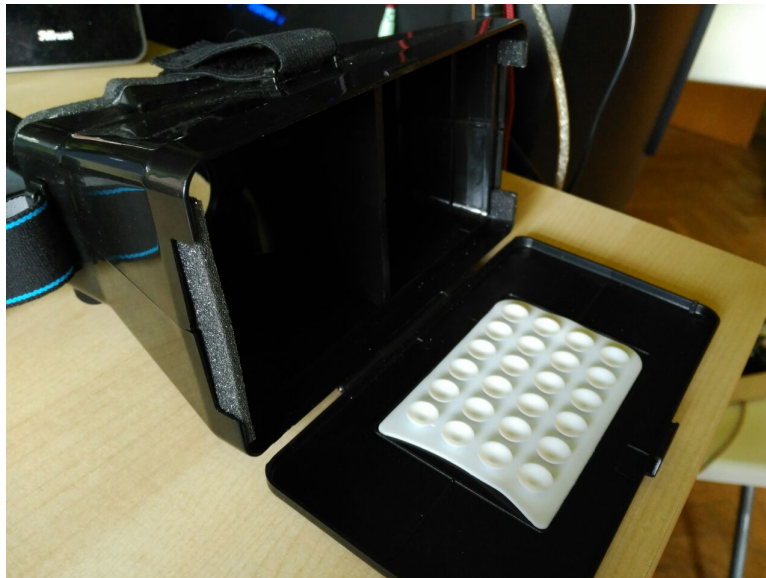


Figura B.6: **Gafas de realidad virtual de plástico:** la parte blanca está compuesta de unas ventosas sobre las que se adhiere el dispositivo móvil para que éste no se mueva con los movimientos de la cabeza del usuario.



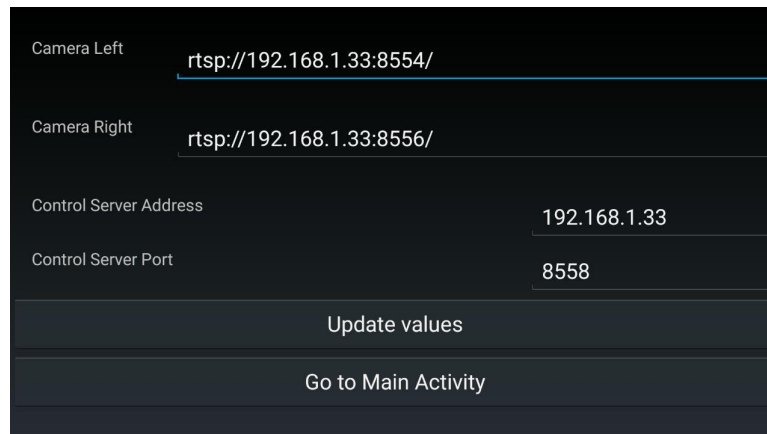
Figura B.7: **Gafas de realidad virtual de plástico:** las lentes enfocan las imágenes que se muestran en la pantalla del dispositivo móvil, adaptándolas al ojo para que el usuario pueda sentir la profundidad a partir de las dos vistas.



Figura B.8: **Smartphone Android:** Utiliza la versión Android 5.1.1 Lollipop como sistema operativo. Resolución full HD (1920x1080p).

B.3. Aplicación Android

A continuación se muestran las capturas de pantalla correspondientes a las dos vistas de la aplicación Android del cliente.



The screenshot shows a configuration screen for a mobile application. It has a dark background with white text. There are four input fields for configuration: 'Camera Left' with the value 'rtsp://192.168.1.33:8554/', 'Camera Right' with 'rtsp://192.168.1.33:8556/', 'Control Server Address' with '192.168.1.33', and 'Control Server Port' with '8558'. Below these fields are two buttons: 'Update values' and 'Go to Main Activity'.

Camera Left	rtsp://192.168.1.33:8554/
Camera Right	rtsp://192.168.1.33:8556/
Control Server Address	192.168.1.33
Control Server Port	8558
Update values	
Go to Main Activity	

Figura B.9: **Formulario de la aplicación móvil.** Permite al usuario modificar las direcciones de los tres servidores.



Figura B.10: **Vista principal de la aplicación móvil.** Cada imagen se corresponde a una captura de una de las cámaras alojadas en el servidor. Esta disposición de las fotografías otorga al usuario visión estereoscópica, es decir, la capacidad de sentir profundidad.



Manual de usuario

Manual de ayuda al usuario para utilizar el sistema.

C.1. Repositorio

Manual de usuario: Repositorio

Cómo clonar el repositorio

C.2. Uso

Manual de usuario: Uso

C.2.1. Despliegue de los servidores de vídeo

Manual de usuario: Despliegue de los servidores de vídeo

C.2.2. Código arduino

Manual de usuario: Código arduino

C.2.3. Despliegue del servidor de control

Manual de usuario: Despliegue del servidor de control

C.2.4. Instalación de la aplicación