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

Resumen

Resumen

Palabras Clave

Palabras Clave

Abstract

Abstract

Key words

Key words

Agradecimientos

Agradecimientos

Todo list

Resumen . . . III
Palabras Clave . . . III
Abstract . . . IV
Key words . . . IV
Agradecimientos . . . V
Estado del arte: Introducción . . . 5
Estado del arte: Historia, nacimiento y evolución. . . . 5
Estado del arte: Estado actual . . . 5
Estado del arte: Conceptos previos . . . 6
Referencia a gafas de VR: Descripción del producto . . . 7
Análisis: Viabilidad . . . 7
Vista funcional: Servidor de vídeo . . . 10
Vista funcional: Servidor de control . . . 10
Vista funcional: Cliente móvil . . . 10
Vista dinámica . . . 11
Diseño: Servidor de vídeo . . . 11
Diseño: Servidor de control . . . 11
Diseño: Cliente móvil . . . 11
Implementación: Fundamentos de programación para Android con Android Studio . . . 15
Implementación: Streaming multimedia, códecs y formatos de vídeo . . . 17
Implementación: Cliente/servidor UDP . . . 18
Pruebas: calidad de servicio . . . 21
Pruebas: cámaras . . . 22
Pruebas: servomotor . . . 22
Pruebas: Experimentos del sistema completo . . . 22
Pruebas: Resultados . . . 22
Conclusiones . . . 23
Trabajo futuro . . . 23

Índice general

Índice de Figuras xii

Índice de Tablas xv

1. Introducción 1

1.1. Motivación del proyecto . . . 1

1.2. Objetivos y enfoque . . . 1

1.3. Metodología y plan de trabajo . . . 2

2. Estado del arte 5

2.1. Introducción . . . 5

2.2. Historia, nacimiento y evolución. . . . 5

2.3. Estado actual . . . 5

2.4. Conceptos previos . . . 6

3. Análisis, arquitectura y diseño del sistema 7

3.1. Análisis . . . 7

3.1.1. Descripción del producto . . . 7

3.1.2. Viabilidad . . . 7

3.1.3. Objetivos y funcionalidad . . . 7

3.1.4. Requisitos . . . 8

3.1.5. Tamaño y rendimiento . . . 8

3.2. Arquitectura . . . 9

3.2.1. Vista estática . . . 9

3.2.2. Vista funcional . . . 10

3.2.3. Vista dinámica . . . 11

3.3. Diseño . . . 11

3.3.1. Servidor de vídeo . . . 11

3.3.2. Servidor de control . . . 11

3.3.3. Cliente móvil . . . 11

4. Implementación **15**

4.1. Cliente . . . 15

4.1.1. Fundamentos de programación para Android con Android Studio . . . 15

4.1.2. Aplicación . . . 15

4.2. Servidores de vídeo . . . 17

4.2.1. Streaming multimedia, códecs y formatos de vídeo . . . 17

4.2.2. Configuración del servidor . . . 17

4.3. Servidor de control . . . 18

4.3.1. Servomotor y arduino . . . 18

4.3.2. Cliente/servidor UDP . . . 18

4.3.3. Programación del arduino . . . 18

4.3.4. Programación del servidor . . . 18

5. Pruebas y resultados **21**

5.1. Calidad de servicio . . . 21

5.1.1. Servidores . . . 21

5.1.2. Clientes . . . 22

5.1.3. Interfaces . . . 22

5.1.4. Pruebas significativas (combinaciones de los 3 anteriores) . . . 22

5.2. Cámaras . . . 22

5.3. Servomotor . . . 22

5.4. Experimentos del sistema completo . . . 22

5.5. Resultados . . . 22

6. Conclusiones y trabajo futuro **23**

6.1. Conclusiones . . . 23

6.2. Trabajo futuro . . . 23

Glosario de acrónimos **25**

Bibliografía **26**

A. Fragmentos de código **29**

A.1. Cliente . . . 29

A.1.1. AddressManager.java . . . 29

A.1.2. PositionSender.java . . . 30

A.1.3. FormActivity.java . . . 30

A.1.4. MainActivity.java . . . 32

A.2. Servidores de vídeo 34
A.2.1. deploy.sh 35
A.3. Servidor de control 35
A.3.1. servo_serial_read.ino 35
A.3.2. control_servo.py 37
A.3.3. udp_server.py 38

B. Manual de usuario **39**
B.1. Repositorio 39
B.2. Uso 39
B.2.1. Despliegue de los servidores de vídeo 39
B.2.2. Código arduino 39
B.2.3. Despliegue de los servidores de vídeo 39
B.2.4. Despliegue de los servidores de vídeo 39

Índice de Figuras

1.1. **Ejemplo de aplicación para Google Cardboard**. Utiliza OpenGL para renderizar los gráficos y los muestra para adaptarse a cada ojo. 2

1.2. **Movimiento de la cabeza** alrededor del eje vertical 2

1.3. **Google Carbdoard**: las gafas de realidad virtual a partir de unas lentes, cartón y un smartphone 3

3.1. **Diagrama conceptual de la red del sistema** 8

3.2. **Coordenadas polares**. Azimuth (Ψ), Pitch (Φ), Roll (θ) 9

3.3. **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. 10

3.4. Diagrama de clases del servidor de control 11

3.5. Diagrama de clases de la aplicación para Android 12

3.6. **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

4.1. **Formulario de la aplicación móvil**. Permite al usuario modificar las direcciones de los tres servidores 16

4.2. **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. 18

A.1. **Patrón de diseño *Singleton***. La única instancia de esta clase puede ser referenciada invocando al método público y estático *getInstance()* 29

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 29

A.3. **Atributos de la clase**. 30

A.4. **Método constructor**. 30

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. 30

A.6. **Definición de la clase, AddressManager y *onCreate()***. 30

A.7. **Método *updateValues()***. Controlador del botón «Update values». Accede a los *EditText* de la vista y los almacena en el modelo (AddressManager) 31

A.8. **Método *nextActivity()*.** 31

A.9. **Inicialización de objeto VideoView**. El mismo proceso se realiza para myVideoViewRight y ambos comparten el mismo MediaController. 32

A.10. **Acceso a los sensores y creación del objeto *PositionSender***: 32

A.11. Hilo de envío de la posición al servidor de control 33

A.12. **Método *onSensorChanged()***: Realiza el cálculo de las coordenadas polares (figura 3.2) cada vez que los sensores cambian de valor. 34

A.13. **Método *onBackPressed()***: 34

A.14. Script de despliegue de los servidores multimedia 35

A.15. **Cabecera del código del Arduino** 35

A.16. **Función auxiliar *myRead()***. Lee caracter a caracter del puerto serial hasta encontrar el valor '\n' y convierte la cadena de caracteres obtenida en un valor numérico 36

A.17. **Función *setup()* de Arduino**. Esta función se ejecuta al arrancarse la placa . . 36

A.18. **Función *loop()* de Arduino**. Esta función se ejecuta indefinidamente después de ejecutarse *setup()* 36

A.19. **Definición de la clase y constructor de *ServoControl***. El ángulo inicial es $\Psi_0 = 0°$. El constructor crea un objeto *Serial* en el puerto 9600 a partir de su ruta en el sistema operativo. 37

A.20. **Definición de la clase y constructor de *ServoControl***. 37

A.21. **Inicialización de recursos del servidor de control**. 38

A.22. **Bucle del servidor de control**. 38

Índice de Tablas

1 Introducción

1.1. Motivación del proyecto

Mi motivación personal a la hora de embarcarme en este proyecto es la de crear un sistema heterogéneo, investigando así campos como la programación para dispositivos móviles (en este caso, para Android) y las redes multimedia, materias en las que era más bien inexperto antes de comenzar con este trabajo. Por otra parte, mi experiencia en las prácticas de empresa que realicé en el proyecto de emprendimiento The Haptic Eye 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

A modo de síntesis, el objetivo inicial es desarrollar un sistema que proporcionara visión remota al usuario mediante de un smartphone conectado, a través de Internet, con un robot con dos cámaras como si este último se tratase de un avatar.

El sistema consta, principalmente, de dos partes:

1. Un Google Cardboard y un teléfono móvil con Android desde el que ejecutar la aplicación. Ésta estará desarrollada utilizando la API de Google Cardboard [2]. En vez de renderizar gráficos en 3D (Figura 1.1), simplemente se reproducirán en tiempo real simultáneamente dos vídeos (uno para cada ojo).

2. Un pequeño robot con las siguientes características:

- Dos videocámaras.
- Una estructura donde alojar estas dos cámaras.
- Un motor que haga girar esta estructura sobre el eje vertical (Figura 1.2)
- Conectividad con la aplicación a través de Internet.

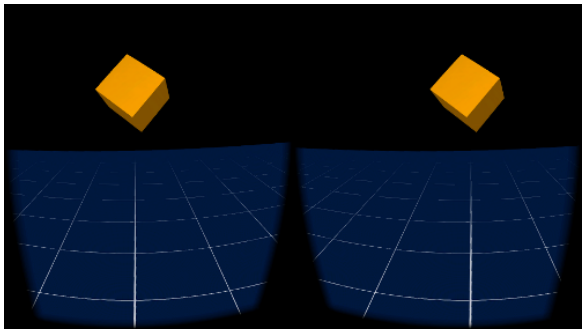

Figura 1.1: **Ejemplo de aplicación para Google Cardboard**. Utiliza OpenGL para renderizar los gráficos y los muestra para adaptarse a cada ojo.

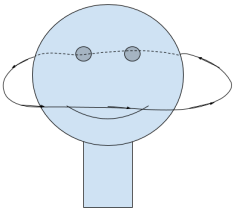

Figura 1.2: **Movimiento de la cabeza** alrededor del eje vertical

La aplicación recibirá como entrada dos streams correspondientes a las cámaras de la segunda parte y la posición en la que esta parte se encuentra (aunque en la práctica esto último quizás no haga falta). Como salida, enviará la posición en la que se encuentra la cabeza del usuario como comandos que el subsistema 2 procesará y convertirá en instrucciones para los motores que mueven su estructura.

1.3. Metodología y plan de trabajo

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

2

Estado del arte

2.1. Introducción

Estado del arte: Introducción

La realidad virtual es un concepto que existe desde hace décadas, aunque durante gran parte de este tiempo se ha visto como mera ciencia ficción. Actualmente, ya existen dispositivos diseñados específicamente para que el usuario pueda visualizar mundos virtuales en 3D.

En primavera de 2016 se concentran las fechas en las que numerosas empresas prometieron comercializar productos con esta funcionalidad, como son Oculus Rift, HTC Vive y PlayStation VR entre otros.

Por otra parte, la realidad aumentada combina en tiempo real la visión del entorno físico con elementos virtuales que añaden información a lo que uno podría ver simplemente con sus ojos. Microsoft Hololens llegará al mercado también en la primera mitad del año 2016. El producto descrito a continuación no se podría clasificar como ninguno de estos dos anteriores, aunque sí que es cierto que está fuertemente ligado a estos dos conceptos. En septiembre de 2015, Snapchat incorpora la realidad aumentada a su aplicación, pudiendo personalizar fotografías y vídeos en tiempo real con distinto contenido basado en el reconocimiento facial.

2.2. Historia, nacimiento y evolución.

Estado del arte: Historia, nacimiento y evolución.

2.3. Estado actual

Estado del arte: Estado actual

2.4. Conceptos previos

Estado del arte: Conceptos previos

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 *REFERENCIA FIGURA* conectadas a través 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.

Referencia a gafas de VR: Descripción del producto

3.1.2. Viabilidad

1

Análisis: Viabilidad

3.1.3. 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.

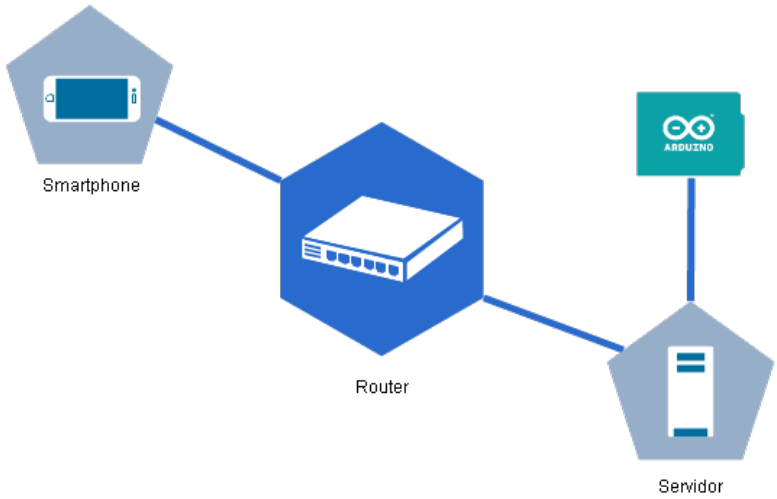

Figura 3.1: **Diagrama conceptual de la red del sistema**

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.
- 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.

3.1.4. 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.

3.1.5. 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.

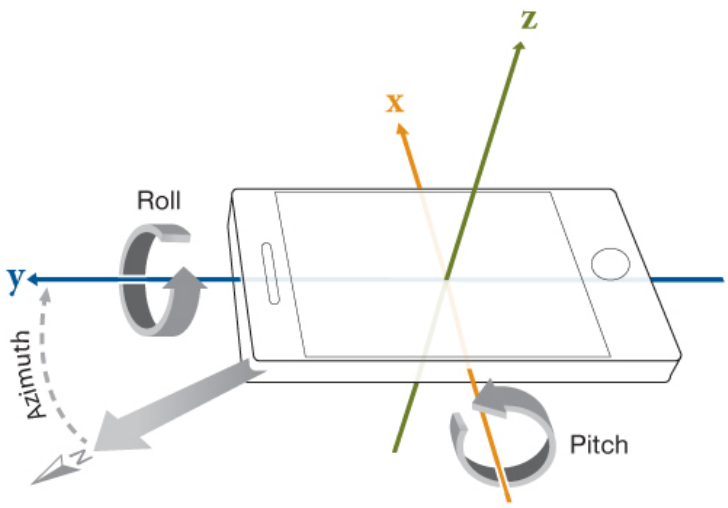

Figura 3.2: **Coordenadas polares**. Azimuth (Ψ), Pitch (Φ), Roll (θ)

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 3.3

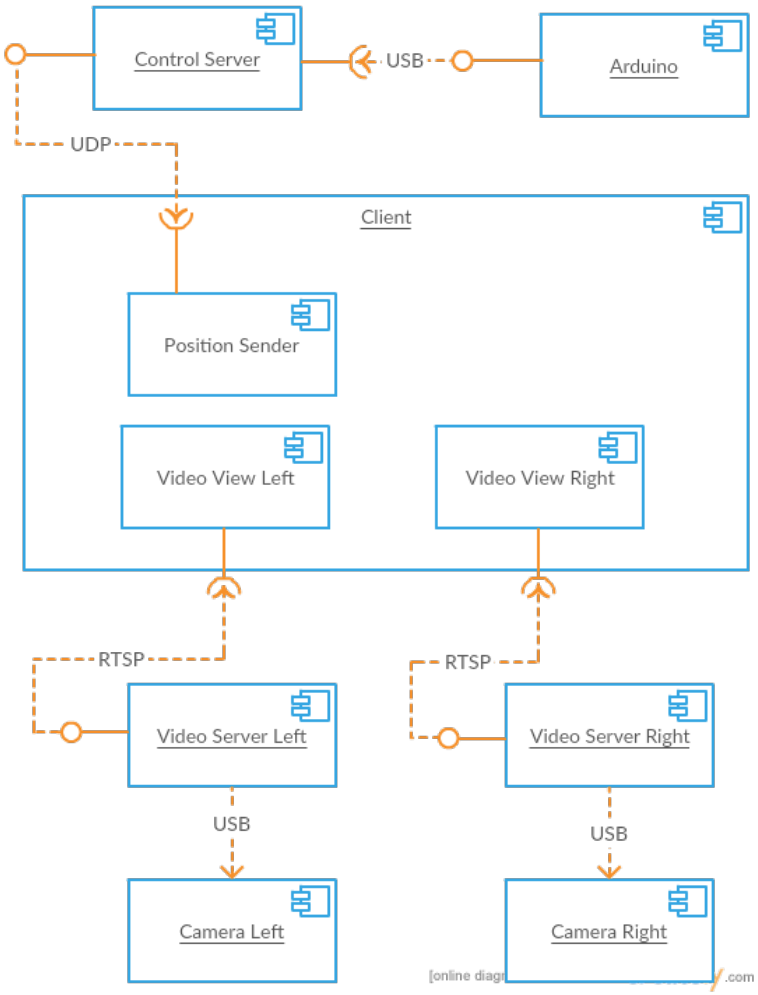

Figura 3.3: **Diagrama de componentes del sistema**. Se pueden apreciar los distintos componentes, las interfaces que ofrecen y requieren y el protocolo que utilizan para comunicarse entre sí.

3.2.2. Vista funcional

Aquí iría una explicación detallada de la funcionalidad de cada componente por separado

Servidor de vídeo

Vista funcional: Servidor de vídeo

Servidor de control

Vista funcional: Servidor de control

Cliente móvil

Vista funcional: Cliente móvil

3.2.3. Vista dinámica

Vista dinámica

Aquí se muestra cómo evoluciona el sistema con el tiempo y dados ciertos eventos.

3.3. Diseño

3.3.1. Servidor de vídeo

Diseño: Servidor de vídeo

3.3.2. Servidor de control

Diseño: Servidor de control

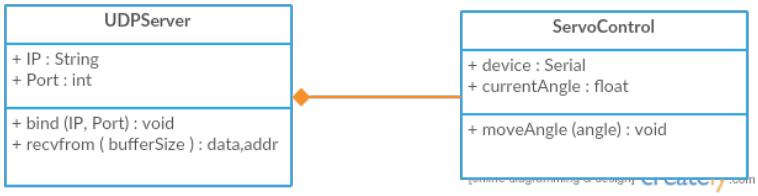

Figura 3.4: Diagrama de clases del servidor de control

3.3.3. Cliente móvil

Diseño: Cliente móvil

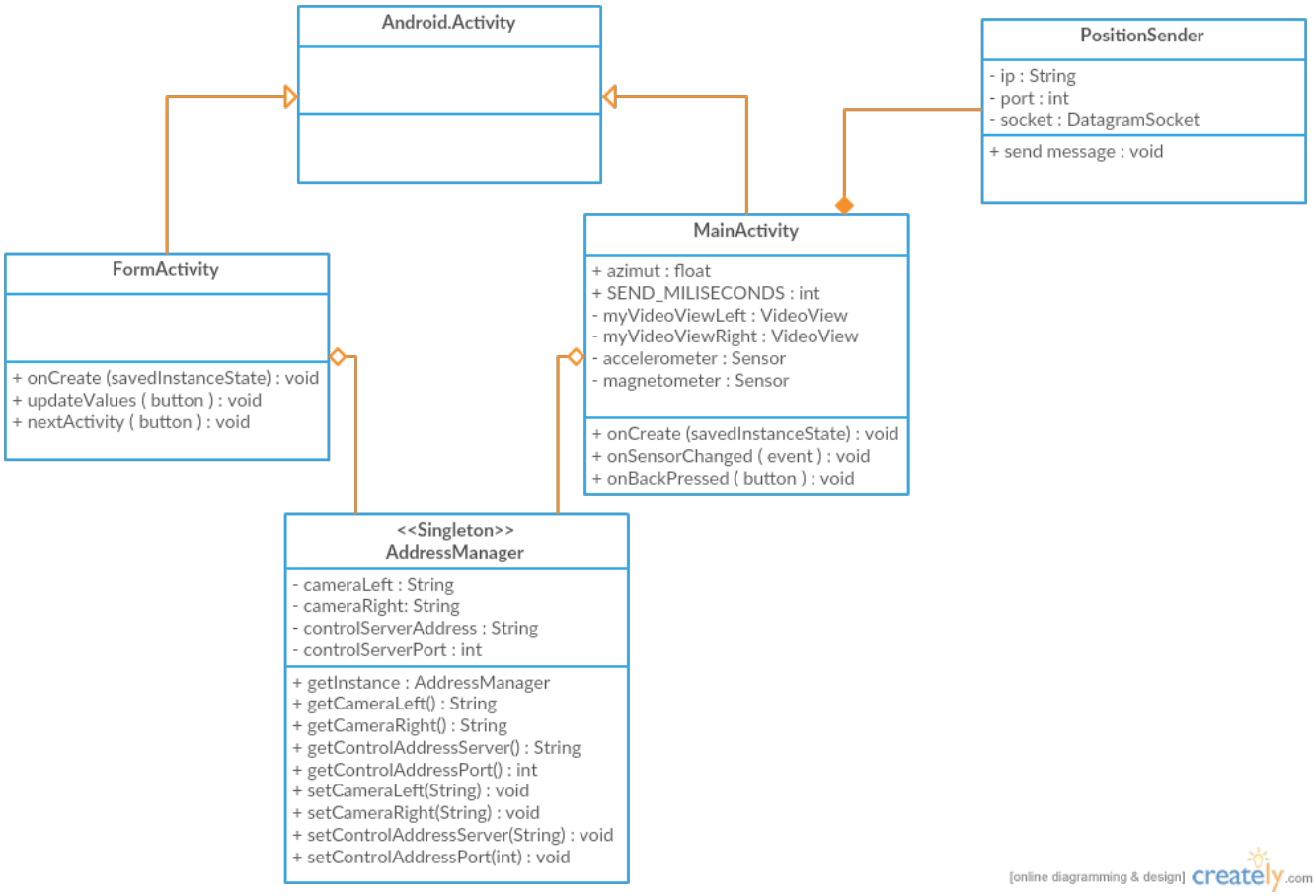

Figura 3.5: Diagrama de clases de la aplicación para Android

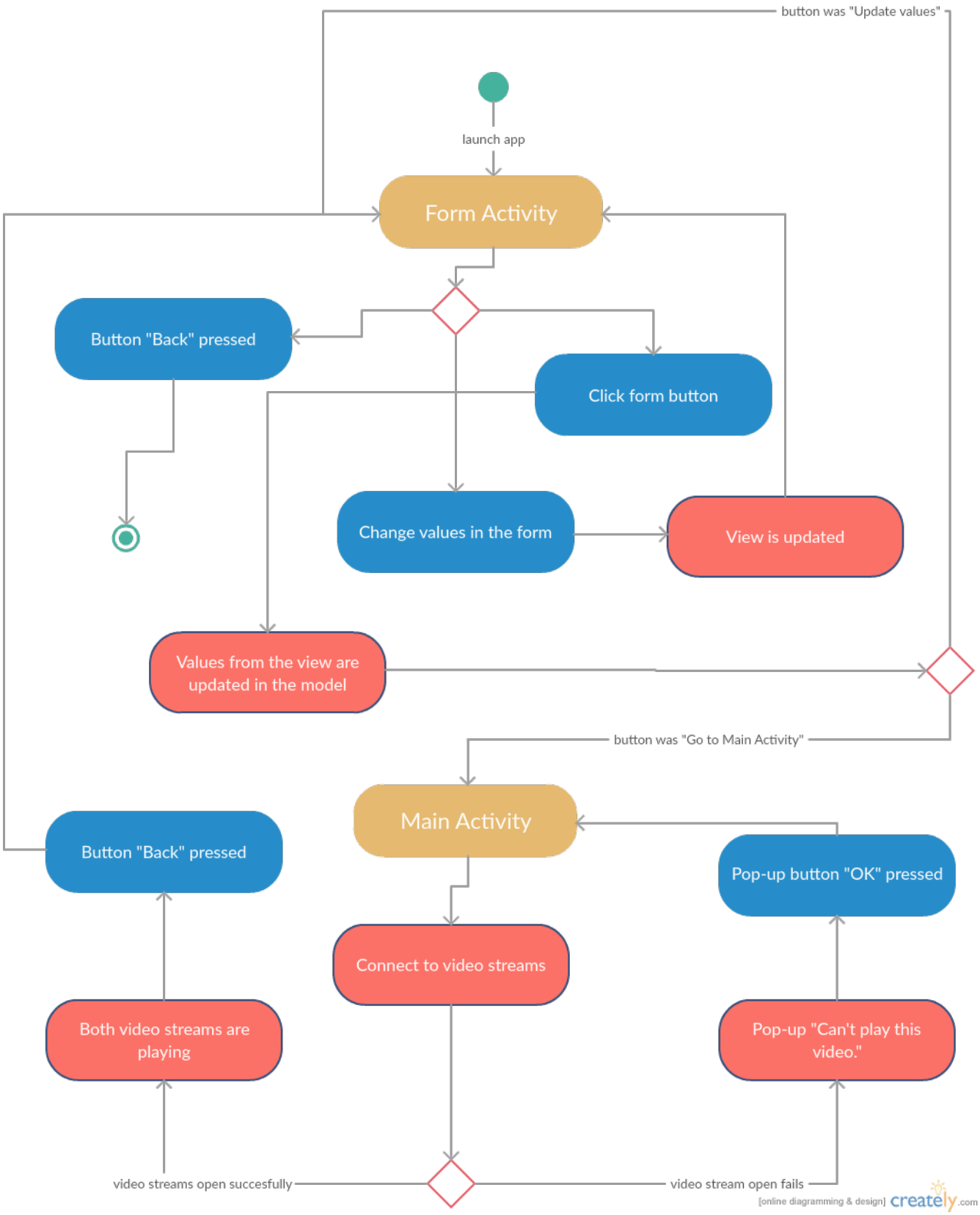

Figura 3.6: **Diagrama de actividad de la aplicación para Android**. En ocre, las vistas de la aplicación. En azul, las interacciones del usuario. En rojo, las respuestas del sistema.

4 Implementación

4.1. Cliente

4.1.1. Fundamentos de programación para Android con Android Studio

Implementación: Fundamentos de programación para Android con Android Studio

Referencias a libro de Android de la biblioteca

- Activity
- Layout
- Interfaces handler (?)

4.1.2. 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* (fragmento A.1). Forma parte del modelo de la aplicación y almacena las direcciones y puertos de los servidores. Permite leer y escribir estos datos (fragmento 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 (fragmento A.4). Encapsula las operaciones básicas con sockets, haciendo más sencilla la comunicación con el servidor de control a nivel de programación (fragmento 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 (fragmento 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 4.1 son los valores por defecto de estos *textboxs* 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 *textboxs* y los almacena en el gestor anteriormente explicado (fragmento A.7). Al presionar el segundo botón, se abandona esta actividad y se inicia la aplicación principal, *MainActivity* (fragmento A.8).

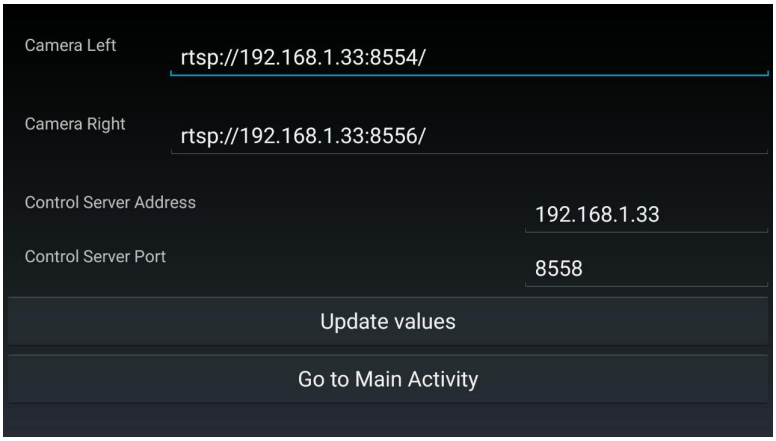

Figura 4.1: **Formulario de la aplicación móvil**. Permite al usuario modificar las direcciones de los tres servidores

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 (fragmento 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* (fragmento A.10)
- Se crea el hilo que se encargue de enviar constantemente la posición, como se muestra en la figura 3.2). 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 *Hanlder* 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 (fragmento A.11).

4.2. Servidores de vídeo

4.2.1. Streaming multimedia, códecs y formatos de vídeo

Implementación: Streaming multimedia, códecs y formatos de vídeo

4.2.2. 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 recurrrido 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 RSTP, en el cual se enviarán los datos a través de RTP y cuyos datos de sesión se guardarán en un archivo SDP, que se encontará 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 1

4.3. Servidor de control

4.3.1. Servomotor y arduino

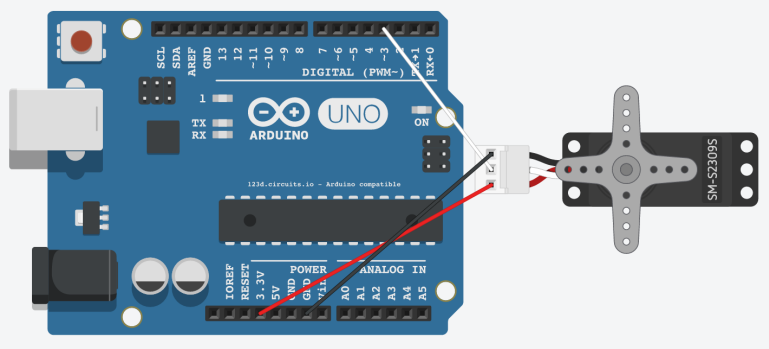

Figura 4.2: **Circuito del Arduino controlando el servomotor**. El cable rojo va a VCC (3.3V) de la placa arduino, el cable negro a GND (tierra) y el cable blanco al PWM correspondiente. Variando el valor de la señal analógica del PWM, conseguimos cambiar de sentido y de intensidad.

4.3.2. Cliente/servidor UDP

Implementación: Cliente/servidor UDP

4.3.3. Programación del arduino

Se ha utilizado la librería [1] para el control del servomotor.

4.3.4. 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.

ServoController

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

5

Pruebas y resultados

5.1. Calidad de servicio

Para medir la *QoS*, se han hecho pruebas combinando distintos clientes, distintos servidores y distintos interfaces de red.

Pruebas: calidad de servicio

5.1.1. Servidores

A continuación, se comentan las distintas implementaciones de los servidores multimedia que se han contemplado.

FFMpeg

FFServer

GStreamer

CVLC

5.1.2. Clientes

FPlay

Programa con JavaCV

App RTSP Player

VLC

5.1.3. Interfaces

localhost

Multicast Wi-Fi

Unicast RJ-45

5.1.4. Pruebas significativas (combinaciones de los 3 anteriores)

5.2. Cámaras

Pruebas: cámaras

5.3. Servomotor

Pruebas: servomotor

5.4. Experimentos del sistema completo

Pruebas: Experimentos del sistema completo

5.5. Resultados

Pruebas: Resultados

6

Conclusiones y trabajo futuro

6.1. Conclusiones

Conclusiones

Los objetivos principales de este proyecto se han visto realizados.

Personalmente, se ha llevado a cabo el desarrollo de un sistema distribuido heterógeneo, 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 demasiado 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.

Por otro lado y como se ha comentado previamente, la realidad virtual se encuentar en auge, motivo por el cual han surgido y surgen nuevas librerías, herramientas, frameworks, etc. orientados 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.

6.2. Trabajo futuro

Trabajo futuro

- 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 *REFERENCIA*. Hasta hace unos meses, se llamaba Cardboard API. Actualmente también incluye soporte para DayDream VR[2], 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, que harían que los vídeos se mostrasen como en la figura 1.1
- Implementar la rotación sobre los ejes X e Y (Pitch y Roll respectivamente, figura 3.2).
- **Utilizar motores paso a paso o *steppers*** para mover la estructura sobre la que se encuentren las cámaras.
- Añadir un stream más que se corresponda con el audio, grabado en el servidor a través de un micrófono cualquiera.

Glosario de acrónimos

- **VR**: *Virtual Reality* (Realidad Virtual)
- **UDP**: *User Datagram Protocol*
- **RTSP**: *Real Time Streaming Protocol* (Protocolo de transmisión en tiempo real)
- **URI**: *Uniform Resource Identifier* (Identificador de recursos uniforme)
- **MVC**: *Model View Controller* (Modelo Vista Controlador)
- **IP**: *Internet Protocol*1
- **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*

Bibliografía

[1] Michael Margolis. Varspeedservo. *https://github.com/netlabtoolkit/VarSpeedServo*, 2009.

[2] Google. Google vr: Daydream. *https://vr.google.com/daydream/*, 2016.

A

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 ipAddres;

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();
    ipAddres = 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, ipAddres, 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 3.2) 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()***:

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 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 caracter a caracter 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 9 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 $\Psi_0 = 0°$. 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

Manual de usuario

Manual de ayuda al usuario para utilizar el sistema.

B.1. Repositorio

B.2. Uso

B.2.1. Despliegue de los servidores de vídeo

B.2.2. Código arduino

B.2.3. Despliegue de los servidores de vídeo

B.2.4. Despliegue de los servidores de vídeo