FACULTAD DE CIENCIAS

Facultad de Ciencias

Proyecto Fin de Carrera

Desarrollo de una plataforma de comunicación para emulación de teletransportación virtual mediante realidad aumentada en dispositivos móviles

Development of a communication platform for emulating virtual teletransportation with AR on mobile devices

Para acceder al Título de

INGENIERO EN INFORMÁTICA

Autor: Borja Cano Barredo

Director: Andrés Iglesias Prieto

Junio - 2015

Contenido

Resumen

Este trabajo consiste en lograr un sistema de comunicación que permita la visualización de la forma de una persona integrada en la imagen real de nuestro entorno. El proceso de comunicación se puede dividir en dos fases.

En la fase de emisión se utilizan dos kinects y se diseña e implementa un software que permite el reconocimiento y grabación en 3D de una persona.

La recepción consiste en visualizar la imagen capturada en la primera fase de forma que parezca integrada en nuestro entorno. Para ello se utiliza un casco HMD que funciona con un smartphone, equipado con dos cámaras USB, como pantalla y unidad de control. Se implementa un software que permite la mezcla realista de las imágenes recibidas desde la fase de emisión con las de las cámaras. Para conseguirlo, este software responde a eventos como el cambio de orientación de la cabeza.

The main goal of this work is the development of a communication system integrating the visualization of a remote person within a real image of our environment. The communication process is divided into two phases.

In emitter phase, we use two Kinect devices and design and implement software code for recognition and 3D recording of a remote person's shape.

The captured image is retrieved at the receiver and seamlessly integrated into our environment. Our setting requires a HMD combined with a smartphone and equipped with two USB cameras as display and control unit. Our software provides a realistic integration of the images from the emitter within the enviroment images from the cameras. To achieve this, the software is reactive to events such as head orientation changes of the subject.

Palabras clave

Realidad Aumentada, Realidad Virtual, Kinect, Android, Estereografía.

Augmented Reality , Virtual Reality , Kinect , Android , stereography.

Capítulo 1: Introducción y objetivos del proyecto

A lo largo del tiempo, el ser humano siempre ha intentado mejorar su capacidad de comunicación. Desde el origen de los primeros lenguajes orales, este proceso de transmisión de información no ha parado de evolucionar, siendo uno de los factores clave en nuestra evolución social y tecnológica (ver Figura 1).

Actualmente, la tecnología de la comunicación sigue en continuo desarrollo intentando mejorar la comunicación a larga distancia y la calidad de la misma. Llegar a todos los lugares es uno de los principales objetivos, así como también lo es llegar con los medios y mejoras necesarios para maximizar las posibilidades de la comunicación.

Figura 1: Gafas TV estéreo de 1963

Internet ha supuesto una revolución en este sentido, llegando a causar cambios en muchos aspectos de la sociedad. No sólo es una forma rápida y eficaz de obtener información, sino que nos permite expandir las posibilidades de la comunicación entre personas, de forma que hoy día, ya no es raro establecer una comunicación a kilometros de distancia sin dejar de verse las caras.

Cada día, millones de personas hablan por videoconferencia con sus familiares, amigos o compañeros de trabajo en otros lugares del mundo. Esta tecnología permite comunicarse de una forma más cercana y sencilla, facilitando las expresiones de afecto, enfado, tristeza, y otras emociones, que en una comunicación solo por voz están muy limitadas. Sin embargo, ver a una persona con la que deseamos hablar solo parcialmente y a través de una pantalla sigue estando muy lejos de las posibilidades que ofrece una comunicación entre dos personas que se encuentran en el mismo lugar.

En la definición de “teletransportación” de wikipedia aparece el siguiente fragmento: “*En la ciencia ficción, el tele-transporte generalmente se basa en codificar información acerca de un objeto, transmitir la información a otro lugar, como a través de una señal de radio, y crear una copia del original en el punto de destino*”. A día de hoy, científicamente no se conoce ningún mecanismo mediante el cual el tele-transporte físico de objetos pueda ocurrir y no es claro que esto vaya a ser posible en el futuro. Sin embargo, sí que podemos

intentar crear una copia virtual de un objeto en otro lugar para que parezca que está junto a nosotros. Y ¿Por qué no?, también podemos intentarlo con personas.

Este trabajo intenta crear una base de comunicación usando este concepto. ¿Cómo podemos conseguir una comunicación de persona a persona, en la que estando dichas personas en diferentes lugares, se maximicen la sensación y las posibilidades de estar juntas en el mismo lugar?

En principio, no es posible transmitir la sensación de que una persona está junto a nosotros si la estamos visualizando a través de una pantalla. Tampoco si no podemos ubicarla dentro de nuestro entorno físico. Y mucho menos si lo que estamos viendo realmente es una proyección 2D de esa persona, muy lejos de la imagen tridimensional real que vemos cuando miramos a alguien que se encuentra físicamente frente a nosotros.

Estos problemas pueden solucionarse parcialmente si se usan unas gafas de realidad virtual combinadas con una cámara estereoscópica. Para poder explicar este proceso de comunicación es mejor dividirlo en tres fases (ver Figura 2):

- La primera fase trata sobre la captura de la imagen de la persona a la que se quiere tele-transportar virtualmente (emisión). Un dispositivo ubicado en el entorno físico de la persona emisora se encarga de capturar una imagen 3D (estéreo) de la misma y de separar ésta del resto de la imagen del entorno.
- La segunda fase implica la codificación y transporte de dicha imagen a través de un medio de comunicación (por ejemplo, Internet).
- Por último, la tercera fase se refiere al proceso a través del cual sería posible mezclar la imagen obtenida de la persona emisora con el entorno real (su imagen), también tridimensional, de la persona receptora. Esta fase requiere el uso de unas gafas que combinen tecnología de realidad virtual con realidad aumentada.

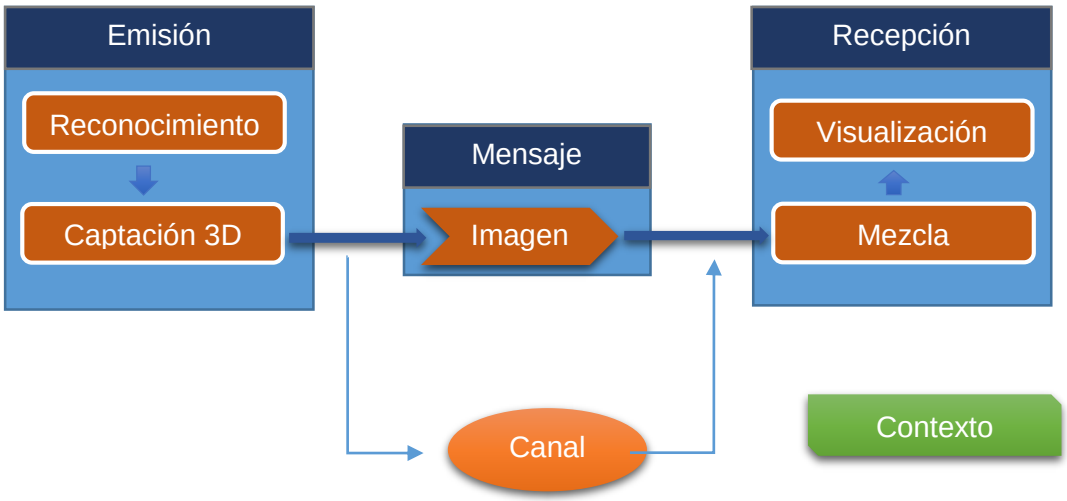

Figura 2: Fases y elementos de una comunicación

Este documento trata, por tanto, del desarrollo de un producto hardware como es un casco de realidad aumentada y realidad virtual que permita realizar la tercera fase explicada, así como todo el software que gira en torno a la idea descrita.

La estructura principal se basa varios capítulos en los que se aborda una descripción y justificación de las tecnologías empleadas tanto en la fase de emisión como en la de recepción, descripción del software o hardware desarrollado, comentarios sobre el resultado y pruebas realizadas. Existe también un capítulo dedicado a las conclusiones finales y otro a posibles mejoras o trabajos futuros.

Capítulo 2: Introducción al proceso de emisión y tecnologías empleadas

2.1 Introducción al proceso de Emisión

Es la primera etapa del proceso de comunicación. En esta fase se realiza la captura de imagen tridimensional de la persona emisora. También se separa la forma de la imagen que ocupa la persona de la del resto del entorno capturado (*foreground-background discrimination*), ya que el objetivo es ubicar virtualmente a la persona emisora en el entorno del receptor.

2.2 Objetivos

El objetivo principal de la etapa de emisión es el desarrollo de un software que permita:

1. Reconocer la forma de una persona a partir de imágenes en las que aparezca.
2. Capturar la escena mediante alguna técnica de grabación 3D.
3. Dado que forma parte de un proceso de comunicación, ambos trabajos deben realizarse con la mayor velocidad posible. El software debe ser capaz de dar salida a una sucesión de imágenes con velocidad suficiente como para obtener una señal de video.
4. Crear un servidor de comunicación al que clientes (receptores) puedan conectarse para obtener las imágenes generadas en los pasos anteriores.

2.3 Descripción y justificación de las tecnologías empleadas

En esta sección se describen algunas de las tecnologías existentes relacionadas con los objetivos de la fase de emisión. Se comentará por qué son interesantes los sensores 3D, que tipos existen, y cuáles son los más adecuados para su utilización en el proyecto. También se comentará cuáles son las posibilidades de captación de imagen 3D y que software podemos utilizar para facilitar el cumplimiento de estos objetivos.

2.3.1 El sensor 3D

El primer objetivo de la fase de emisión trata sobre el reconocimiento de la forma de la persona en la escena para, posteriormente, poder separarla del resto de la imagen. El escáner 3D es importante que el resultado del reconocimiento tenga la calidad suficiente para cumplir los objetivos del proyecto y además sea rápido.

2.3.1.1 Introducción a los sensores 3D

Un escáner 3D es un dispositivo que analiza una escena para obtener datos de su forma que permitan una posterior reconstrucción y modelado de la misma. El objetivo no es crear un modelo completo de ningún objeto, ya que implicaría un barrido de 360º alrededor del mismo o el uso de numerosos sensores, además de un tiempo de procesamiento del que no se dispone. Sin embargo, se hace indispensable su utilización para conseguir un reconocimiento de la forma de la persona que sea fiable y eficaz.

2.3.1.2 Tipo de sensores

Dentro de la clase de escáneres 3D que son capaces de funcionar sin contacto (palpación), encontramos varios tipos:

- **Time of flight:** Es un tipo de escáner que permite determinar distancias cronometrando el tiempo de ida y de vuelta de un pulso de luz (generalmente luz láser) cuando rebota contra la escena. Son muy precisos pero su principal desventaja es su tamaño y su precio. Suelen dedicarse al escaneo de objetos de gran tamaño como edificios, construcciones, superficies, etc. y es común verlos montados sobre aviones o satélites (como en modelos LIDAR para generar elevaciones digitales del terreno).
- **Triangulación:** Un haz de luz láser incide sobre la escena la cual se desea muestrear. Se usa otra cámara entonces con el objetivo de buscar el punto de incidencia. Dependiendo de lo lejos que el haz de laser impacte contra la escena, el punto aparece en diferentes coordenadas dentro del campo de visión de la cámara. Usando cálculos trigonométricos es posible determinar la posición de cada punto en el espacio. Tienen una precisión elevada pero su alcance suele ser muy limitado ya que depende directamente de la distancia entre el emisor láser y la cámara.
- **Diferencia de fase:** Funciona de forma similar al de tipo “time of flight”. En lugar de cronometrar el tiempo de ida y vuelta de un haz de luz, mide la diferencia de fase entre la luz emitida y la recibida. Son muy rápidos y de precisión intermedia, pero la presencia de luz en la escena produce un efecto de ruido en la calidad del muestreo.
- **Luz estructurada:** Los escáneres de luz estructurada proyectan un patrón de rayos de luz sobre la escena y analizan la deformación de ese patrón producida

por la geometría de ésta. Su principal ventaja es la velocidad y su capacidad para escanear objetos en movimiento en tiempo real.

- **Holografía conoscópica:** Un haz reflejado en una superficie atraviesa un cristal con dos índices de refracción, lo que da lugar a dos rayos paralelos que se hacen interferir usando una lente cilíndrica. La frecuencia de dicha interferencia determina la distancia del objeto. Se usa principalmente para la inspección de defectos en la industria del acero.
- **Luz modulada:** Emiten una luz que cicla continuamente su amplitud siguiendo un patrón sinusoidal. Para determinar la distancia viajada por la luz, una cámara determina el cambio de amplitud en el patrón una vez reflejada.
- **Escáneres pasivos:** Los escáneres pasivos, a diferencia de los activos, no emiten ninguna radiación, sino que usan la disponible en el ambiente. La mayoría de estos métodos dependen del movimiento de una cámara o de la detección de patrones similares desde dos cámaras; entonces, mediante algoritmos de triangulación estéreo, se crea un mapa 3D del entorno. Normalmente se trata de algoritmos muy complejos que necesitan un gran número de cálculos lo que se traduce en métodos lentos y poco precisos.

2.3.1.3 Sensor 3D de luz estructurada

A partir de los tipos de escáner 3D analizados es posible determinar que el más adecuado para llevar a cabo el propósito de reconocimiento de imagen de una persona, es el de tipo “luz estructurada”, ya que permite escanear objetos en movimiento en tiempo real.

Se trata de una evolución del método de escaneo por triangulación descrito anteriormente. El principal problema del método por triangulación básico o también llamado de punto único es que requiere de la exploración de los 2 ejes por separado (doble barrido), con su correspondiente retardo. El método de rendija en cambio, permite ahorrar un barrido, aunque sigue sin alcanzar la velocidad suficiente para analizar objetos en tiempo real (ver Figura 3).

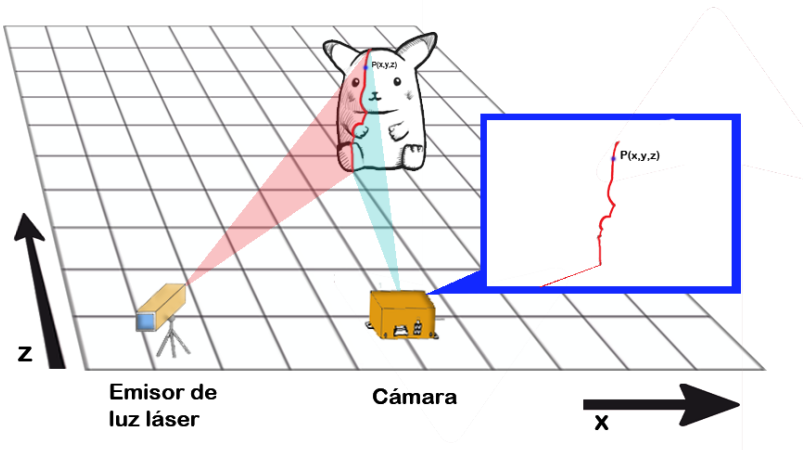

Figura 3: escáner de luz estructurada

Sin embargo, el método de luz estructurada no emite un haz de luz sino que basa su funcionamiento en la proyección de patrones bidimensionales de tipo raya o reja. Este método permite obtener un mayor número de muestras de forma simultánea (Ver Figura 4).

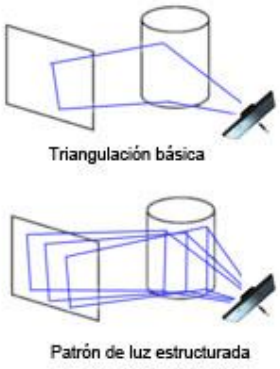

Figura 4: Ventaja del método de luz estructurada

El tipo más usado es el de patrón de rayas. La precisión de este método está limitada por la definición óptica de las rayas y la anchura de las mismas. Teniendo ésto en cuenta, se podría incrementar la precisión reduciendo la anchura de las rayas, lo que daría lugar a una mayor resolución de muestreo. Sin embargo, la resolución de la cámara es un factor límitante es este aspecto. Aún así, es la tecnología más adecuada para aplicar en imagenes en movimiento.

2.3.1.4 Sensores 3D Primesense y Microsoft Kinect

Primesense es una compañía israelí dedicada a los sensores 3D y pionera en la tecnología de luz estructurada. Esta compañía fue comprada por Apple en 2013 por 350 millones de dólares.

Anteriormente, en 2010, *Microsoft* decidió usar la tecnología de *Primesense* y su SoC (*system-on-chip*) para capturar y analizar la profundidad de los objetos. El método de *Primesense* consiste en lanzar los haces de luz mediante un láser infrarrojo y capturar sus puntos de incidencia mediante un simple sensor de imagen CMOS equipado con un filtro infrarrojo. Esta tecnología permite obtener una precisión por debajo de los 70 milímetros usando tecnología relativamente barata. Durante el desarrollo *Microsoft* denominó a este proyecto “Project Natal” y tenía como objetivo la creación una interfaz hombre-máquina sin necesidad de usar controladores físicos para juegos, como mandos, teclados o ratones. El dispositivo salió a la venta a finales de 2010 con el nombre de *Kinect* y estaba diseñado para usarse con la videoconsola *Xbox 360.*

La precisión, tamaño y bajo ratio de error de *Kinect* en relación con su precio (unos 50€), hace que sea un candidato ideal para para cumplir con el objetivo de reconocimiento de la persona.

Primesense también puso en el mercado otros sensores de características muy similares a *Kinect* como son los sensores Carmine y Capri. En 2011 salieron a la venta otros 2 sensores, está vez bajo la marca Asus, el Xtion y el Xtion Pro Live. Todos ellos tienen un funcionamiento y prestaciones similares, sin embargo, son mucho más complicados de encontrar que *Kinect*.

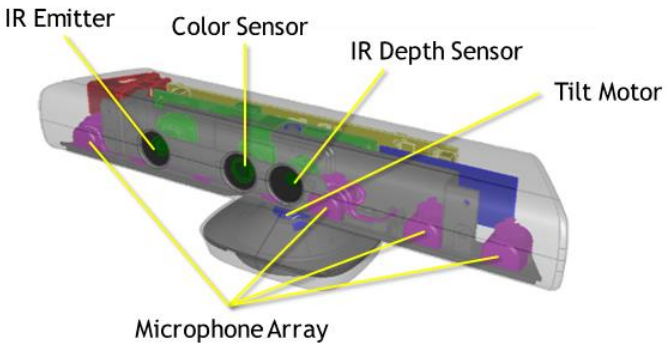

Figura 5: Componentes del sensor Kinect

Kinect es un dispositivo con forma de barra horizontal que equipa una cámara RGB, un sensor de profundidad y un micrófono (Ver Figura 5) el cual permite, mediante la ejecución del software adecuado, el reconocimiento de movimientos corporales, reconocimiento facial y análisis de voz.

2.3.2 Software para los sensores

En esta sección se describen el software utilizable para facilitar la tarea de comunicación con los sensores.

2.3.2.1 Kinect SDK y OpenNI

Una vez seleccionado el hardware para capturar la escena tridimensional se necesita una interfaz que nos permita comunicarnos y leer datos desde los sensores.

OpenNI era una organización responsable de un framework de código abierto que facilitaba esta tarea (Ver Figura 6). En 2013, tras la adquisición de *Primesense* por parte de Apple y convertirse en su subsidiaria, Apple decidió cerrar el proyecto y el 23 de Abril de 2014 su web dejo de existir, los binarios, sus fuentes, así como toda su documentación dejaron de estar disponibles de forma oficial. Sin embargo, recientemente, el equipo de structure.io ha rescatado este proyecto y tanto los binarios como la documentación pueden encontrarse en su web.

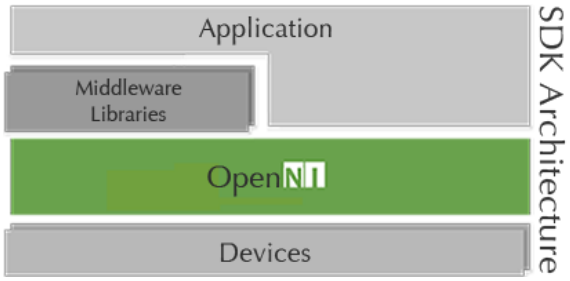

Figura 6: Arquitectura de OpenNI

OpenNI, por un lado, proporciona a los desarrolladores la habilidad para conectarse a los sensores de forma sencilla y por otro lado brinda la posibilidad de trabajar con una API unificada para todos los sensores de tipo Primesense. En el caso de *Kinect*, OpenNI añade soporte al dispositivo a través de su SDK oficial de Microsoft.

Kinect SDK es un kit de desarrollo que permite la comunicación con el sensor a bajo nivel (raw sensor streams) y también proporciona funciones de más alto nivel como el seguimiento de los movimientos del usuario (skeletal tracking)

2.3.2.2 NiTE

NiTE es un middleware que opera sobre el framework OpenNI y está desarrollado por PrimeSense. NiTE proporciona información de más alto nivel sobre la escena basándose en la información que reúne del sensor de profundidad del dispositivo. Su objetivo es proporcionar funciones que ayuden en tareas relacionadas con el desarrollo de aplicaciones NIUI (Natural Interactive User Interface), cuya misión es proporcionar una interfaz hombre-máquina basada en la captación de movimiento y no en la utilización de un teclado, un ratón o cualquier otro dispositivo físico de control. Por ello facilita funciones que ayudan a identificar gestos, movimientos de la mano o posturas corporales.

2.3.3 Estereoscopía

2.3.3.1 Introducción a la estereoscopía

La estereoscopía es una técnica capaz de crear una ilusión de profundidad mediante una imagen estereográfica. Una imagen estereográfica se forma a partir de un par de imágenes bidimensionales. Esta tecnología hace uso de los principios biológicos de la visión, ya que a partir de las pequeñas diferencias entre las imágenes, el cerebro calcula las distancias mediante técnicas de paralaje.

2.3.3.2 Paralaje y disparidad binocular

El paralaje es el cambio aparente de la posición de un objeto producido por una variación de posición del observador. Es una técnica comúnmente usada para medir grandes distancias que de otra forma no sería posible conocer, como por ejemplo, la distancia que hay entre una estrella y nosotros.

Para medir una distancia tan grande hasta un objeto muy lejano (por ejemplo un objeto llamado 'C'), es posible situarse en un punto 'A' desde el cual mirando al objeto 'C', este queda alineado con un tercer objeto 'L' mucho más lejano todavía. Si ahora cambiamos de posición y nos situamos en otro lugar 'B', el objeto deja de estar alineado con 'L'. Si 'L' es un objeto mucho más lejano que 'C', podemos suponer que las líneas que se forman entre 'A' y 'L' y entre 'B' y 'L' son paralelas (Ver Figura 7). Debido a este fenómeno se puede determinar que los ángulos que forman las líneas 'BL' y 'BC' y las líneas 'BC' y 'AC' son prácticamente iguales (ángulo 'a').

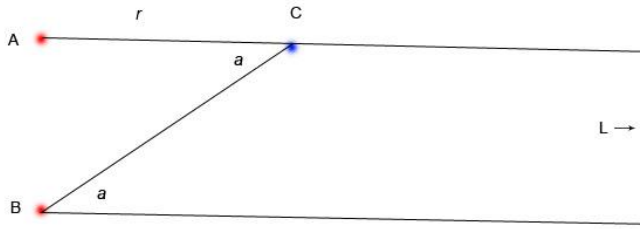

Figura 7: Calculo de distancias mediante paralaje

Como el ángulo 'a' es muy pequeño, se supone que la distancia entre 'A' y 'B' es un arco de centro en 'C' y de radio 'AC' (r en la ilustración), siendo esta la distancia que queremos obtener. En una circunferencia de radio r se cumple que el arco AB que abarca el ángulo 'a' es:

$$\mathrm{ARCO(AB)} = (a * 2 * \pi * r)/360$$

Función en la que sólo se desconoce r. El ángulo 'a' es el que representa el paralaje y es muy importante medirlo con precisión. Esto es sólo un ejemplo de cómo es posible medir distancias usando la técnica de paralaje. El cerebro emplea procesos similares para interpretar una tercera dimensión. La técnica usada por el cerebro para obtener distancias se denomina disparidad binocular.

La disparidad binocular se refiere a la diferencia entre la localización de un punto de la imagen visto con el ojo derecho y el mismo punto visto con el ojo izquierdo, es decir, a la diferencia entre los puntos de proyección en los ojos. Esa diferencia está definida por el punto en el que se cruzan las líneas rectas que parten de cada ojo y van a parar al punto del objeto.

Para saber si un punto está cerca o lejos se debe comparar con otro. La diferencia en grados entre las líneas que unen cada ojo con ambos puntos será lo que determine en que magnitud un punto está más cerca o más lejos que el otro.

Figura 8: Disparidad binocular

Como se ve en la figura 8, los ángulos d1 y d2 serán los que determinen si el coche está más cerca que el árbol y en que magnitud.

En visión por computador, para imitar este efecto, dos cámaras toman imágenes de la misma escena, pero están separadas exactamente por la distancia interpupilar humana, que usualmente está en torno a los 6.35cm.

2.3.3.3 Visualización estereoscópica

Existen varios tipos de imagen estereoscópica:

- **Libre paralela:** (también denominada side-by-side) Los ojos observan cada uno su imagen correspondiente, manteniendo sus ejes ópticos paralelos. Este método es el más similar a la visión humana, pero tiene una limitación evidente y es que la distancia entre los centros de cada imagen no puede ser superior a la distancia interpupilar, que tiene un máximo de 7,2cm. Por esta razón, no es un método usado en pantallas de televisión o cine.
- **Libre cruzada:** El efecto 3D se observa cruzando los ejes ópticos de los ojos. Aunque es posible ayudarse de un lápiz situado entre estos, no es método práctico. Sin embargo, es el único que permite observar la imagen de forma tridimensional sin la necesidad de usar gafas y para distancias entre centros de imágenes superiores a 7,2cm.
- **Anáglifo:** Se utilizan gafas con filtros de colores complementarios, típicamente rojo y azul. El observador debe colocarse unas gafas de tal manera que un filtro

queda frente a cada ojo (Ver Figura 9). De esa forma, la imagen presentada en rojo no es vista por el ojo que tiene el filtro del mismo color pero si por el otro, y viceversa.

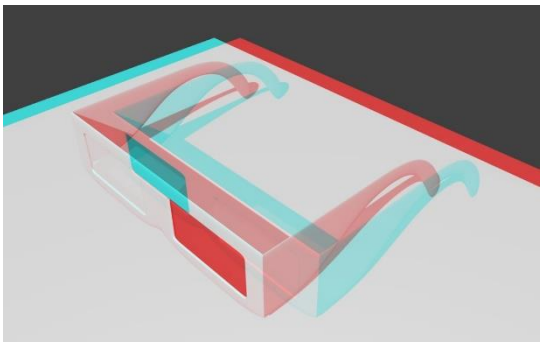

Figura 9: Gafas 3D para la visualización de anáglifos

- **Polarización:** Se utiliza luz polarizada para separar las imágenes izquierda y derecha. Es el método más usado en pantallas de cine y televisión ya que, a diferencia del anáglifo, este no altera los colores.
- **Secuencia entrelazada:** Se muestran las imágenes secuencialmente y de forma intercalada las imágenes de la derecha y de la izquierda. Requiere de unas gafas con obturadores de cristal líquido, de forma que una vez sincronizadas, cada ojo solo es capaz de ver su imagen correspondiente.
- **HMD (Head Mounted Display):** Consiste en un casco que contiene una o dos pantallas y unas lentes de forma que la imagen se visualiza directamente sobre el dispositivo. Es una variación de método side-by-side.

2.4 Conclusión

Para el trabajo desarrollado se hace uso de varias de las tecnologías y métodos expuestos. Para cumplir el objetivo del reconocimiento de la persona y llevarlo a cabo con fiabilidad y cumpliendo todos los objetivos será necesario el uso de un dispositivo *Kinect*.

Para el desarrollo del software partiremos de la base del framework OpenNI en el procesamiento de imagen.

Por último, ya que además del reconocimiento es necesario usar alguna técnica de captación que permita la posterior visualización 3D en tiempo real en el equipo receptor, la única tecnología que se adapta a los requisitos es el HMD, por lo que se requiere un dispositivo de captación side-by-side.

Sin embargo, no es posible usar por separado un dispositivo *Kinect* para ayudar al reconocimiento de la persona y otro dispositivo, como puede ser una cámara estéreo, para la captación de la imagen. Esto es debido a que la correspondencia entre pixeles en cada uno de los dispositivos es casi imposible de efectuar y mucho menos en el mismo instante de tiempo. Para solucionar el problema se utilizarán dos *Kinect* en lugar de uno, con el objetivo de usar la cámara RGB de cada dispositivo como fuente de captación estéreo.

Capítulo 3: Software Emisor o Servidor de Imágenes

3.1 Introducción al desarrollo del software

En este capítulo se describen las distintas fases de desarrollo del software que dan lugar al programa resultado que ejecuta como servidor de imágenes y cumple con los objetivos de esta etapa de la comunicación.

3.2 Análisis de Requisitos

Los requisitos funcionales son:

- A partir de los datos obtenidos de los sensores debe ser capaz de identificar la forma de una persona en la imagen de una escena si está presente en la misma.
- Debe mostrar una imagen al usuario emisor para que pueda verificar el correcto reconocimiento y funcionamiento del programa.
- La conexión con los sensores 3D debe ser automática, no debe requerir ningún conocimiento sobre la identificación y conexión de los mismos.
- Debe proporcionar funciones de servidor que permita a equipos clientes conectarse por Internet o red de área local. Los clientes reciben entonces las imágenes procesadas por el servidor.
- La velocidad de procesado y envío debe ser suficiente como para proporcionar una señal de video fluida.

Los requisitos no funcionales son:

- El soporte para la conexión de los sensores y la sincronización de los mismos será llevada a cabo mediante la utilización de la API de OpenNI.
- Para la identificación de pixeles pertenecientes a la forma de una persona en una imagen será necesaria la ayuda de algoritmos presentes en el middleware NiTE.
- La conexión de red será establecida mediante sockets a través de los cuales serán enviadas las imágenes.
- Debido a que el SDK de *Kinect* solo tiene soporte oficial para Windows, esta será la plataforma elegida para la ejecución del software servidor.

- Para mejorar la velocidad de transmisión así como la correcta visualización de las imágenes por parte del usuario emisor, la librería *OpenCV* será la utilizada para facilitar algunas compresiones y conversiones de formato de las imágenes.

3.3 Diseño

3.3.1 Resumen del diseño del servidor

Se debe diseñar un software que cumpla los requisitos señalados. El software será desarrollado en C++ utilizando la librería gráfica QT para el desarrollo de la interfaz de usuario.

El primer paso es la inicialización de los componentes. Como se ha explicado anteriormente, con el objetivo de conseguir una imagen estereográfica que posteriormente nos permita la visualización de la persona emisora en el dispositivo destino en tres dimensiones se necesitan 2 sensores, cada uno de los cuales dispone de su correspondiente cámara RGB. El software buscará de forma automática los sensores al iniciar el servidor, de tal forma que si no encuentra ninguno o sólo uno, debe informar al usuario con un mensaje de error o esperar su conexión.

Para conectarse a los sensores, primero debe inicializarse OpenNI. Si ambos dispositivos han sido conectados e inicializados de forma satisfactoria, el software estaría listo para funcionar y podemos iniciar el resto de componentes como podemos ver en el siguiente diagrama (Figura 10):

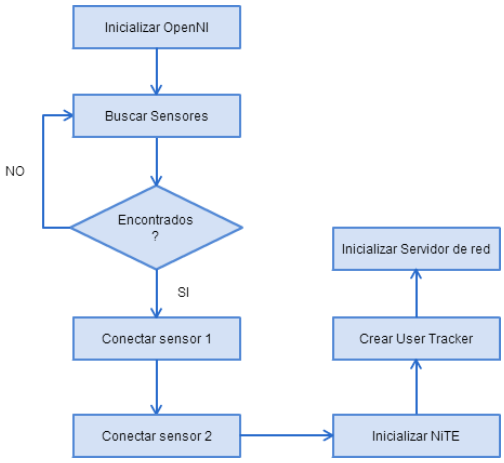

Figura 10: Inicialización del servidor

Al conectarse a cada sensor se configuran aspectos clave como la resolución y la velocidad, que en el caso de *Kinect*, se ajustan al máximo: 640x480@30fps.

Para trabajar con los datos que retornan los sensores, hay que conocer los tipos de objeto que retornan. Uno de los objetos más importantes en este sentido es el objeto *VideoFrameRef*.

Siempre que se necesite leer una imagen de alguno de los sensores, ya sea el sensor de profundidad, el sensor IR o el sensor RGB, la imagen es devuelta en un objeto de tipo *VideoFrameRef*. Este tipo de objeto tiene una serie de métodos que nos permiten obtener información relativa a la imagen como puede ser su altura, anchura o formato. El método más importante es "*getData()",* que retorna un puntero (void*) al primer pixel de la imagen. Existen cinco formatos de imagen utilizables de los cuales sólo vamos a necesitar uno: "*OniRGB888Pixel*"

OniRGB888Pixel es una estructura que contiene 3 bytes, cada uno de los cuales representa un valor RGB de color del pixel. Una vez se conoce el formato de imagen y se dispone del puntero al primer pixel se puede recorrer la imagen completa incrementando este puntero.

Otro método importante del objeto *VideoFrameRef* es "getStrideInBytes()", el cual retorna un entero que indica el número de bytes que ocupa cada fila de la imagen en memoria. Normalmente el resultado de esta función será igual al ancho en pixeles de la imagen multiplicado por el número de bytes que ocupa cada pixel, pero no siempre es así. Debido a la forma en que las imágenes son guardadas en memoria puede existir un relleno o espacio sin usar al final de cada fila, por lo que si se desea incrementar el contador desde el primer pixel de cualquier fila hasta el primer pixel de la siguiente es mejor hacerlo usando el resultado de esta función.

Como se ha explicado anteriormente, NiTE es un middleware que funciona sobre OpenNI y proporciona métodos de más alto nivel principalmente enfocados al desarrollo de aplicaciones NIUI. Una de las clases más interesantes de NiTE para este proyecto es *UserTracker*. El objeto resultante de instanciar esta clase nos permite obtener un mapa de usuarios que no es más que un array de tamaño ancho-imagen x alto-imagen x 2 bytes, donde esos 2 bytes representan un entero que contiene un "ID" de usuario. Si el ID es igual a 0 significa que en esa posición de la imagen no hay ninguna persona; en cambio, sí es distinto de 0 significa que ese pixel pertenece a la imagen de una persona. Si hay varias personas en el campo de actuación de los sensores, cada una tendrá un ID distinto.

Una vez que se dispone del objeto *VideoFrameRef* del sensor RGB y del mapa de usuarios es posible actuar de la forma indicada en la figura (ver Figura 11) para separar la imagen de la persona de la del resto de la escena.

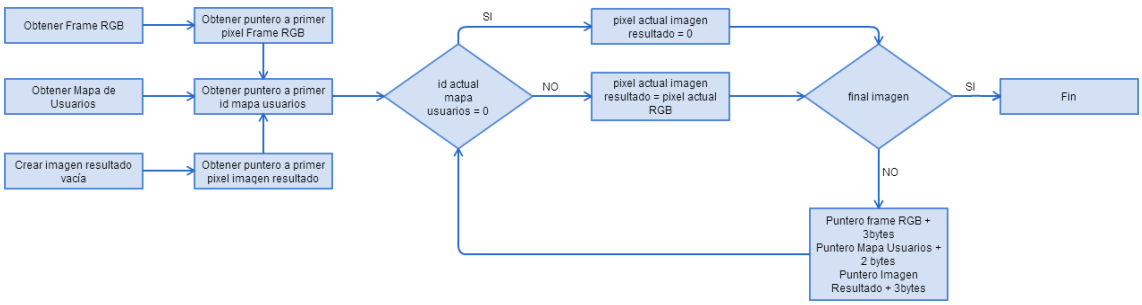

Figura 11: Procesado de la imagen

Como se ve en la imagen, se recorre la imagen obtenida del sensor RGB y el mapa de usuarios de tal manera que la posición que se está leyendo en el mapa se corresponde con la del pixel en la imagen. Si para esa posición el valor en el mapa de usuarios es 0, esto significa que el pixel que ocupa esa posición en la imagen RGB no pertenece a una

persona, y por tanto se crea una imagen resultado cuyo valor en esa posición sea 0 (color negro). Si por el contrario esa posición contiene un ID en el mapa de usuarios, escribimos el valor RGB de esa posición en la imagen resultado.

Como hay que recorrer las imágenes pixel a pixel, primero se crea un bucle que recorre las filas y luego otro que recorre las columnas de cada fila.

```
PunteroPixelRGB imagenResultado;

Para (y = 0; mientras y < altura_resultado; y++){

        PunteroPixelRGB pixelResultado =  imagenResultado + ( y * ancho_resultado);  //primera posición
        de cada fila

        Para (x = 0; mientras x < ancho_resultado; x++){

                PunteroPixelRGB pixelFuente = imagenRGB.getData() + y * imagenRGB.getStrideInBytes()
                + x;

                Copia(pixelResultado, pixelFuente, tamaño(pixelRGB));

                pixelResultado = pixelResultado + 1;

                PunteroMapa pixelMapa = mapa.getData() +  y  * mapa.getStrideInBytes() + x;

                Si (valor(userMapa) == 0 ){

                        Valor(pixelResultado)->b = 0;

                        Valor(pixelResultado)->r = 0;

                        Valor(pixelResultado)->g = 0;

                }

        }

}
```

Esta función se encarga de cambiar el valor de los pixeles que no pertenecen a una persona en color negro para un solo sensor.

El software realiza este proceso para cada uno de los dos *Kinects* necesarios para crear la imagen estéreo. Después, dependiendo del modo de envío, las imágenes se unen para ser enviadas en una sola imagen estéreo de 1280x480 pixeles o la imagen izquierda y derecha se envían por separado.

Como se ha visto, es necesario usar el mapa de usuarios que se obtiene a partir del sensor de profundidad y la imagen procedente del sensor RGB al mismo tiempo para resolver el problema. Sin embargo, el dispositivo envía las imágenes de cada sensor por separado. Esto significa que no hay garantía de que ambos sensores vayan a realizar la captura al mismo tiempo. OpenNI incluye una función que permite que la lectura de ambos sensores se realice en el mismo instante de tiempo. Para usarla es necesario inicializar ambos sensores por separado antes de la inicialización de NiTE; de otra forma, fallará. Esta función se llama *setDepthColorSyncEnabled(boolean sync).*

Un problema similar ocurre con la correspondencia entre pixeles de ambos sensores. El sensor de profundidad y el sensor RGB son dispositivos distintos que no

ocupan el mismo lugar y por tanto no captan los objetos exactamente con la misma perspectiva. Es complicado encontrar qué pixel de uno de los sensores se corresponde con el pixel del otro sensor en el mundo real. OpenNI también incluye la funcionalidad para solucionar este problema. Para ello debemos configurar el dispositivo para que ajuste la imagen entregada por el sensor de profundidad en correspondencia con las imágenes RGB, la función para llevar a cabo esta configuración en el framework es *setImageRegistrationMode(IMAGE_REGISTRATION_DEPTH_TO_COLOR).*

Con la imagen ya procesada, se continúa enviando la misma hasta el equipo cliente. Al tratarse de una comunicación en directo, no es posible implementar ninguna técnica de buffering. El envío es realizado mediante sockets utilizando el protocolo TCP. Para optimizar la velocidad de transferencia las imágenes son comprimidas en formato JPEG antes del envío y descomprimidas en destino. La librería *OpenCV* es utilizada para llevar a cabo este proceso de codificación y para mostrar al usuario la imagen en directo.

Dado que *Kinect* es capaz de procesar 30 imágenes por segundo, el programa lee y procesa las imágenes a esta velocidad. Sin embargo, solo enviará la imagen actual cuando el cliente esté listo para recibirla. Otras técnicas más complejas como el envío de las imágenes a través de protocolo UDP junto con un identificador de imagen fueron planteadas, pero no fue posible hallar ninguna técnica de descarte de imágenes eficiente en tiempo real.

El servidor entonces requiere una señal de respuesta del cliente para enviar la siguiente imagen. Mientras un hilo de ejecución se encarga de procesar y enviar las imágenes, otro permanece a la escucha de la señal de reconocimiento del cliente. Una vez que se recibe una respuesta, se procede al envío (Ver Figura 12).

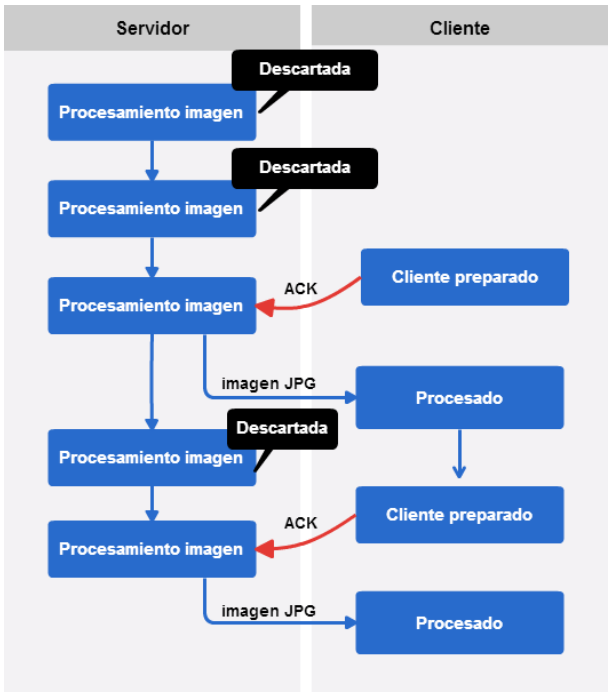

Figura 12: Comunicación síncrona

De esa forma se garantiza una comunicación en directo y un ajuste de la velocidad de transferencia en función de la velocidad de red disponible y las capacidades de procesamiento de los clientes.

Este será el núcleo de funcionamiento sobre el que girará toda la implementación del servidor.

3.3.2 Paralelismo

El procesamiento de imagen es una tarea costosa. Hay que recorrer todos los pixeles de dos imágenes de 640x480 pixeles de tamaño para obtener la imagen estéreo resultado. En cada iteración se lee la imagen RGB y el mapa de usuarios, pero podemos leer estos valores en cada sensor de forma independiente. Para optimizar la eficiencia del método es posible aprovechar las capacidades multicore de los nuevos procesadores dividiendo el procesamiento de la imagen correspondiente a cada sensor en dos hilos de ejecución paralelos. Así, la función de procesamiento principal pasa a ser la encargada de crear estos dos hilos de trabajo, cada uno ejecutando la función correspondiente a uno de los sensores (Ver tabla 1) y de esperar al resultado.

Procesamiento de imagen	
Image_processing_sensor_1(void* pParams)	Se encarga de procesar la imagen resultado del sensor 1.
Image_processing_sensor_2(void* pParams)	Se encarga de procesar la imagen resultado del sensor 2.

Tabla 1: Funciones de trabajo de los hilos de procesamiento de imagen.

Estas mejoras no tienen un gran impacto en el rendimiento resultado, ya que como se verá posteriormente, es en el cliente donde se encuentra el cuello de botella principal.

3.3.3 Multipuerto

Con el objetivo de exprimir la velocidad de la red al máximo se realiza otra aproximación para la transferencia de imágenes. Se implementa un modo multipuerto en el que la imagen de cada lado se envía a través de un socket distinto. A priori parece contraproducente ya que duplicamos la capa de control del protocolo de red, pero podemos enviar ambas imágenes de forma paralela usando dos hilos de ejecución distintos.

Así, son tres los hilos encargados de la transferencia de las imágenes, uno permanece siempre a la escucha de posibles respuestas de clientes, mientras que los otros dos envían ambas imágenes al mismo tiempo.

Serán necesarios 3 puertos para llevar a cabo este método de comunicación. El primero abrirá el socket encargado de la comunicación de control, mientras los dos restantes enviarán cada uno una de las imágenes.

En las pruebas reales veremos que esta mejora puede suponer el incremento de 1 FPS de media.

3.3.4 Falso 3D

En cine, se llama “3D real” a la grabación de una escena usando dos cámaras desde dos ángulos distintos imitando la visión binocular.

Se denomina falso 3D cuando la grabación está realizada en 2D con una sola cámara, pero posteriormente se realiza un procesado de las escenas a partir del cual se obtienen dos imágenes estéreo. Normalmente este tipo de escenas implican que artistas realicen ajustes y correcciones. Sin embargo, algunas TVs de última generación, a través de algoritmos muy complejos, son capaces de crear un falso 3D muy realista sin necesitar la intervención humana.

Uno de los pasos más complejos es el análisis de profundidad de la escena a partir del cual obtenemos una matriz de profundidades de los objetos y actores. Para la generación de esta matriz es obligatorio el análisis de varios frames al mismo tiempo, ya que será gracias al movimiento de la cámara y el cambio de posición de los objetos respecto a ella, que seremos capaces de conocer cómo de lejos o de cerca se encuentran unos respecto de otros.

Otro de los pasos más costosos en procesamiento y dificultad es el cambio de posición de estos objetos en la generación de la imagen 3D. Partiendo de la base de una imagen estéreo en espejo obtenido de la imagen 2D, y una vez obtenida la matriz de profundidad, se sabe que los objetos más cercanos deben desplazarse hacia el centro en la imagen 3D. Sin embargo, cuando se usan dos cámaras y la escena es captada desde dos perspectivas distintas, no solo varía la posición de los objetos, también estamos captando partes de ese objeto que con una sola cámara quedan fuera del rango de visión. Para conseguir un grado de realismo suficiente, la generación de estas imágenes implican la deformación de estos objetos y además deben “inventarse”, mediante técnicas de interpolación avanzadas, partes de la escena que no ha sido posible capturar. Esto da lugar a halos y anomalías en la imagen que son las que típicamente corrigen artistas, pero que en la generación automática no es posible.

Este proceso de generación artificial de imágenes estereográficas no es perfecto, pero que en sus últimas versiones consigue un grado de realismo muy importante, y que además, en películas de ciencia ficción puede crear una sensación de 3D exagerado al incrementar el efecto paralaje de los objetos en escena. Este efecto de exageración se consigue incrementando el factor de desplazamiento de los objetos cercanos respecto de los lejanos. Igualmente podríamos conseguir este efecto usando cámaras si las separamos entre sí por encima de la distancia inter-pupilar.

A modo de experimento, un modo falso 3D es implementado en el software del servidor, siendo necesaria, en este caso, la captación y el envío de una sola de las imágenes capturadas por ambos *Kinect*. Como el análisis de la escena en busca de la profundidad de un objeto es una tarea compleja y necesita del uso de varios frames, lo que implica un retardo, aprovechamos la capacidad de *Kinect* y su sensor de profundidad para facilitar la tarea.

A grandes rasgos, el método se basa en el análisis de la matriz generada por el sensor de profundidad de *Kinect* en las partes de la escena donde se encuentra una persona. Una vez tenemos la forma de la imagen que pertenece a una persona separada del resto de la escena, procesamos la matriz de profundidad en esas mismas coordenadas

y calculamos la media. Ese valor de profundidad media servirá para saber el grado de paralaje que debemos aplicar.

Por tanto, si el falso 3D está activado en el programa servidor, además del frame y su tamaño, también debemos enviar el dato de profundidad de la persona reconocida (si es el caso). De la generación de la imagen estéreo a partir de esos datos se encargará el cliente.

Este método se lleva a cabo principalmente con la misión de incrementar el frame rate, que es uno de los problemas principales de la aplicación junto con la resolución.

3.3.5 Hilos de ejecución y pipelining

Tanto el procesamiento de imagen como también el envío de las imágenes a clientes son tareas costosas y de larga duración, la interfaz gráfica debe quedar congelada a la espera de una función bloqueante y la respuesta de un posible cliente debe ser atendida lo más rápido posible. Por todas estas razones, el paralelismo en la aplicación servidor se hace imprescindible.

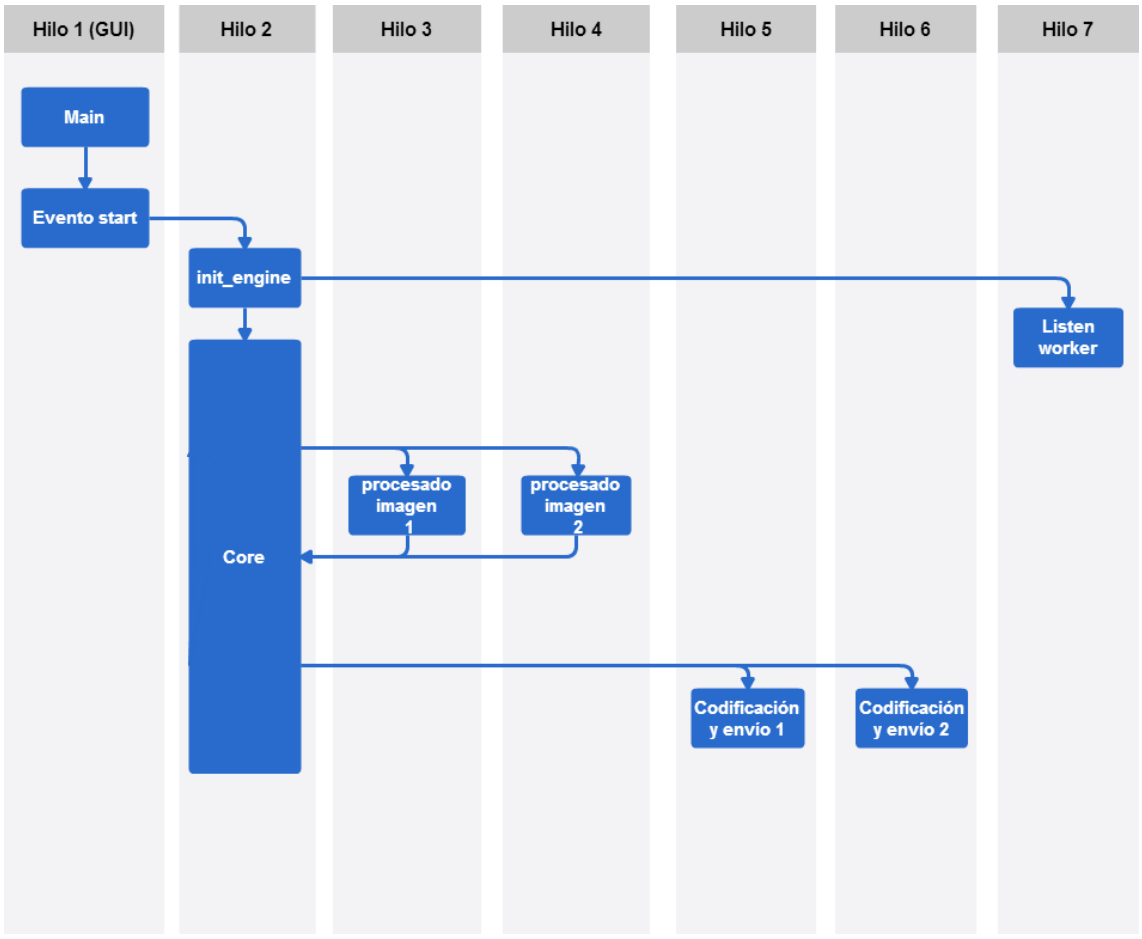

Figura 13: hilos de ejecución del servidor

Como se puede ver en la figura 13, el servidor puede tener hasta siete hilos de ejecución paralelos en el modo multipuerto.

El hilo 1 es sobre el que funciona la interfaz gráfica, de esta forma se evita el bloqueo de la misma mientras se ejecutan otras funciones.

Cuando se pulsa el botón 'start' en la UI, el manejador del evento crea un hilo (hilo 2) en el que ejecuta la función *init_engine* encargada de inicializar el framework y los sensores, y posteriormente, si todo va bien, la misma ejecuta la función *core* y crea el hilo encargado de atender las respuestas de los clientes (hilo 7).

Core a su vez crea dos hilos encargados del procesamiento de las imágenes de cada sensor (hilo 3 e hilo 4). Para evitar una desincronización entre ambos sensores, *core* espera a que ambos hilos terminen de ejecutarse antes de proseguir. Sin embargo, como veremos más tarde, esto no evita que igualmente puedan existir 7 hilos funcionando al mismo tiempo.

Una vez procesadas ambas imágenes y listas para el envío, en el modo multipuerto, *core* crea dos hilos cada uno ellos encargado de enviar una de las imágenes por un socket distinto (hilo 5 e hilo 6). *Core* no espera a que estos hilos terminen, sino que inmediatamente comienza el procesado del siguiente frame.

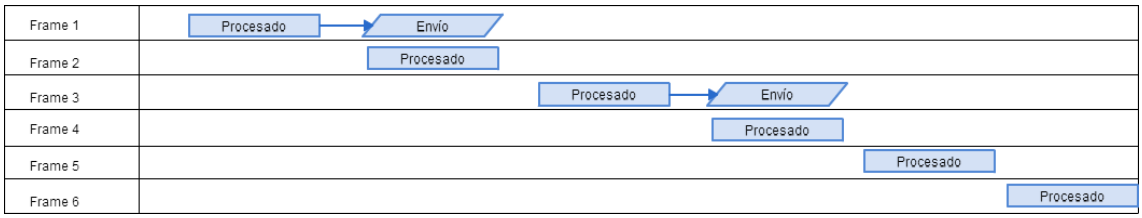

Figura 14: Pipelining de las tareas de procesado y envío

Esta imagen (Figura 14) muestra cómo la tarea de envío (cuando se produce) puede solaparse con la tarea de procesado del frame siguiente. La duración del procesado y la del envío se desconocen, pero se sabe que, inmediatamente después de crear los hilos de envío, *core* comienza una nueva iteración con un nuevo frame.

3.3.6 Estructura del proyecto

En este proyecto usamos cuatro ficheros para la implementación del servidor, además del XML que define la interfaz (ver Figura 15).

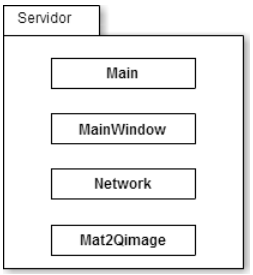

Figura 15: Paquete servidor

Main es la clase principal que inicia la generación de la interfaz gráfica instanciando la clase *MainWindow*.

MainWindow.cpp contiene todas las funciones de procesamiento de imagen y lógica de control de la aplicación. También contiene la clase *MainWindow* con los manejadores de eventos de la interfaz.

Network.cpp contiene las funciones de inicialización del servidor así como métodos que permiten el envío de las imágenes y la comunicación con los clientes.

Mat2Qimage contiene funciones auxiliares para convertir las imágenes de forma que puedan ser visualizadas correctamente en la interfaz gráfica usando la librería Qt.

3.4 Programación

3.4.1 Introducción a la programación

El lenguaje de programación utilizado es C++. Este lenguaje ha sido elegido porque es el más adecuado para trabajar con todos los frameworks elegidos. El IDE utilizado para la parte lógica de la aplicación es Visual Studio. La interfaz gráfica se realiza con QT Project.

El primer paso antes de empezar a programar es crear un nuevo proyecto e importar las librerías que se van a utilizar. La instalación tanto del SDK de *Kinect* como de OpenNI y NiTE se realizará a partir de la ejecución de sus binarios y la inclusión de sus librerías al proyecto (ver Figura 16).

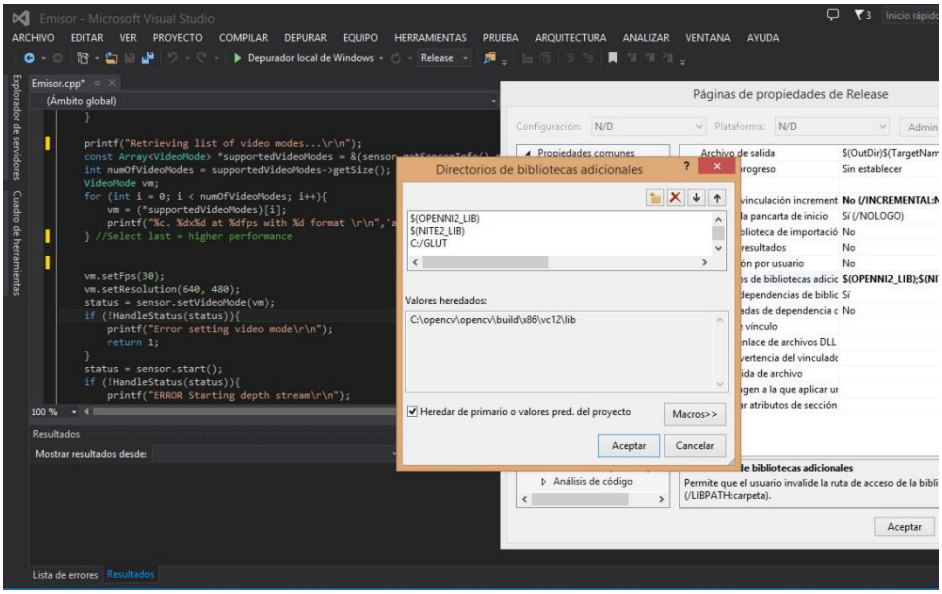

Figura 16: Importación de librerías en proyecto

3.4.2 MainWindow.cpp

Mainwindow.cpp es el fichero que contiene toda la funcionalidad relativa al procesamiento de imagen. Además, contiene la clase del mismo nombre con los manejadores de eventos de la interfaz gráfica y otras funciones para interactuar con ella.

Los métodos que implementa se pueden dividir en varios grupos (Tabla 2):

Manejador de errores	
HandleStatus(Status s)	Recibe como parámetro un objeto Status. Si s es un status de error, devuelve falso y muestra un mensaje de error por pantalla. Si no hay errores, devuelve verdadero.
Manejadores de eventos y GUI	
on_pushButton_clicked()	Maneja el evento de pulsación del botón 'start'. Comprueba que los datos introducidos sean correctos (como por ejemplo los números de puerto introducidos), y si lo son crea un nuevo hilo de ejecución que comienza la inicialización de los sensores y de OpenNI.
setTextConsole(String text)	Escribe el texto deseado en un campo de texto en la interfaz gráfica a modo de consola. Sirve para notificar al usuario de todo lo que sucede en la aplicación.
on_pushButton_2_clicked()	Se trata del evento disparado por el botón 'stop'. Modifica el valor de varios flags que detienen todos los hilos de ejecución. También detiene el servidor de red.
setFPS(double value)	Escribe en el campo FPS de la UI el valor de frames/seg actual.
setImagePreview(QImage qLeft, QImage qRight)	Recibe como parámetro las imágenes RGB de ambos sensores en el formato de imagen de QT y las muestra en la interfaz gráfica.
setState(String text)	Sirve para modificar una etiqueta que señala al usuario sobre el estado de la aplicación. Los estados pueden ser: apagado, esperando clientes o funcionando.
on_horizontalSlider_valueChanged(int value)	Recibe el evento generado al cambiar el valor de la barra de calidad. Este valor define el grado de compresión JPEG.
on_multiport_stateChanged(int state)	Esta función es llamada si la casilla multipuerto es activada. Cambia la lógica de funcionamiento de la aplicación además de habilitar dos campos de texto para introducir los puertos requeridos.
on_checkBox_stateChanged(int state)	Habilita un flag que permite la introducción de un campo de texto en las imágenes durante el procesado. Este campo de texto muestra a qué lado corresponde cada imagen.
on_checkBox_2_stateChanged(int state)	Invierte el lado en el que se muestran y se envían las imágenes.

on_fake3d_stateChanged(int state)	Habilita el modo falso 3D y desactiva el modo multipuerto, ya que no son compatibles.
Comunicación o red	
restartServer()	Reinicia el servidor de red
startServer()	Inicia el servidor de red
ListenWorker(void* pParams)	Es el método de trabajo del hilo dedicado a la escucha de respuestas del servidor. Si el cliente está listo para recibir la siguiente imagen cambia un booleano (flag) a su valor verdadero.
isClientReady()	Devuelve verdadero si el cliente está listo.
Sensores	
SetActiveSensor1()	Inicializa el sensor RGB para el dispositivo 1
SetActiveSensor2()	Inicializa el sensor RGB para el dispositivo 2
ActivateDepthSensor(Device* dev)	Inicializa el sensor de profundidad para el dispositivo especificado como parámetro
Motor	
init_Engine(void *params)	Recibe como parámetro un puntero a un objeto MainWindow. Se necesita para enviar mensajes a la consola e interactuar con la interfaz. Esta función es la encargada de llevar a cabo la inicialización de frameworks y sensores necesarios y de notificar errores que puedan surgir: - Inicializa OpenNI - Busca e inicializa ambos sensores *Kinect* - Inicializa NiTE - Instancia el objeto userTracker - Reserva el espacio para las imágenes resultado - Realiza la llamada para iniciar el servidor de red - Si todo es correcto, llama a la función core() Si se producen varias llamadas a esta función sin cerrar la aplicación tan solo iniciará el servidor de red, ya que, el resto permanecerá inicializado para ahorrar tiempo.
Core()	Core es el método que decide qué y de qué forma actúa la aplicación. - Realiza el cálculo de FPS - Inicia los hilos de ejecución que llevan a cabo del procesado de imagen principal. - Si falso 3D está activado sólo procesa y envía la imagen de uno de los sensores. Además de añadir el cálculo de profundidad. - Si multipuerto está activado ambas imágenes se envían por sockets distintos; si por el contrario, está desactivado, ambas imágenes son mezcladas en una sola imagen estéreo

	y es enviada a través del socket principal. - Codifica la imagen o las imágenes en formato JPG con la calidad seleccionada por el usuario.
send_image(t* param)	Es la función de trabajo del hilo de ejecución encargado de enviar la imagen resultado solo en el caso de que multipuerto este activo. También lleva a cabo la codificación JPEG con el objetivo de paralelizar la tarea.
Image_processing_sensor_1 (void* pParams) Image_processing_sensor_2(void* pParams)	Es la función de trabajo del hilo de ejecución encargado del procesamiento de imagen principal (Ver Figura 17). Separa la imagen de la persona de la del resto de la escena. Si es necesario realiza un análisis de la profundidad (solo en modo falso 3D)

Tabla 2: Resumen de funciones de MainWindow.cpp

Una vez las imágenes han sido procesadas se obtienen dos imágenes resultado, una de cada dispositivo *Kinect*. Ambas se envían al mismo tiempo al cliente. Para simplificar el proceso, en el modo monopuerto, se utilizan funciones de *OpenCV* para unir ambas imágenes en una sola imagen estéreo y para codificar ésta en formato JPEG. Antes de la codificación, la imagen se muestra al usuario, pero sólo se muestran las imágenes que son enviadas al cliente. De esa manera, la velocidad de reproducción de las imágenes será la misma en el servidor y en el equipo cliente. El usuario emisor podrá darse cuenta por tanto de si el cliente tiene una señal de video fluida.

```
//Get user map
nite::UserMap usersMap = usersFrame.getUserMap();

printf("Processing frame of sensor 1...\r\n");
for (unsigned int y = 0; y < (window_h - 2 * margin_y); ++y){

    OniRGB888Pixel* resultFramePixel1 = resultFrame1 + ((y + margin_y) * window_w) + margin_x;

    for (unsigned int x = 0; x < (window_w - 2 * margin_x); ++x, ++resultFramePixel1){

        //Pixel actual de la imagen RGB para el dispositivo 1
        OniRGB888Pixel* RGBPixel1 = (OniRGB888Pixel*)((char*)newFrame.getData() + ((int)(y / resizeFactor) * newFrame.getStrid
        memcpy(resultFramePixel1, RGBPixel1, sizeof(OniRGB888Pixel));
        //ID actual
        nite::UserId* userPixel1 = (nite::UserId*)((char*)usersMap.getPixels() + ((int)(y / resizeFactor) * usersMap.getStride

        //Resultado RGB para el sensor 1
        if ((int)*userPixel1 == 0){
            resultFramePixel1->b = 0;
            resultFramePixel1->g = 0;
            resultFramePixel1->r = 0;
        }else{
            if(fake3d){
                //calculamos la distancia hasta el sujeto
                if(x%10==0 && y%10==0){ //malla
                    //pixel de profundidad - en kinect, aprox de 300mm a 4000mm
                    DepthPixel* depthPixel = (DepthPixel*)((char*)depthFrame.getData()+((int)(y/resizeFactor)*depthFrame.getSt
                    counter_depth++;
                    cumsum += (float)*depthPixel;
                }
            }
        }
    }
}
depth_mean = cumsum / counter_depth;
sensor1_done = true;
```

Figura 17 Fragmento del procesamiento de imagen

3.4.3 Network.cpp

Network.cpp reúne las funciones directamente relacionadas con la red y la transferencia de imágenes. Tiene 3 funciones principales que se detallarán a continuación (Tabla 3):

StartSocket(std::string port)	Crea y retorna un socket de conexión en el puerto especificado como parámetro.
shutdownSocket(SOCKET *ClientSocket)	Cierra el socket especificado como parámetro.
SendF(vector<uchar>& buff,SOCKET *ClientSocket,int fake3d, int depth)	Envía un frame pasado como parámetro en forma de vector a través del socket pasado también como parámetro.

Tabla 3: Resumen de funciones de Network.cpp

La función de envío de la imagen consiste en un bucle que divide y envía la imagen en varias tramas. Antes de hacerlo se envía el tamaño total de la imagen para que el cliente pueda saber cuántos bytes tiene que recibir para completar una imagen (ver Figura 18). Otra opción menos eficiente sería enviar las imágenes seguidas y que el cliente busque el código de finalización hexadecimal, que en todas las imágenes JPEG es el mismo, para cada una de ellas.

Si falso 3D está activo, además de enviar el tamaño de la imagen, también enviará el valor de profundidad medio de la persona para que el cliente pueda calcular el factor de paralaje de forma más eficiente.

```
int sendF(vector<uchar>& buff, SOCKET *ClientSocket,int fake3d, int depth){
    int totalSize;
    int bytes_sent = 0;
    int total_bytes_sent = 0;
    totalSize = buff.size();
    int length = htonl(buff.size());
    //Envía el tamaño primero
    int c = send(*ClientSocket, (char*) (&length), 4, 0);
    if (c == SOCKET_ERROR){
        return 1;
    }
    //Si fake3d está activado envía el valor de profundad medio
    if(fake3d){
        int depth_n = htonl(depth);
        c = send(*ClientSocket, (char*) (&depth_n), 4, 0);
        if (c == SOCKET_ERROR){
            return 1;
        }
    }
    //Envía la imagen en varias tramas
    while (total_bytes_sent < totalSize){
        bytes_sent = send(*ClientSocket, (char*)((&buff[0]) + total_bytes_sent), totalSize, 0);
        if (bytes_sent == SOCKET_ERROR){
            return 1;
        }
        total_bytes_sent += bytes_sent;
        printf("%d bytes sent", total_bytes_sent);
    }
    return 0;
}
```

Figura 18: Función sendF encarga de enviar un frame al cliente

3.5 Interfaz Gráfica

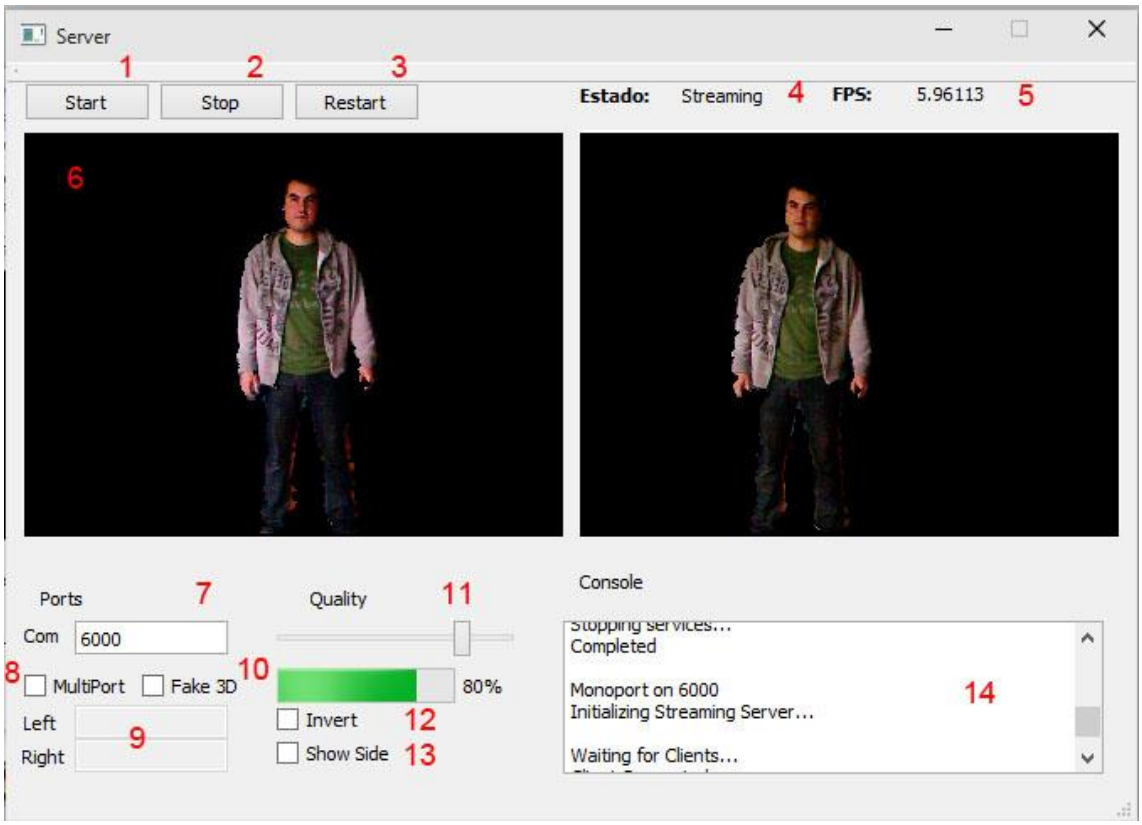

Figura 19: Interfaz gráfica del Servidor

La interfaz gráfica está realizada usando las bibliotecas de Qt. A continuación se detallan cada uno de los elementos empleados (Figura 19):

1. El botón *start* inicia las funciones del servidor. Se encarga de buscar los sensores, iniciar OpenNI, NiTE, el servidor de red y dejar el programa en estado de escucha o de espera de clientes.
2. El botón *stop* detiene las funciones de servidor. No elimina los sensores, por lo que si se vuelve a pulsar start inmediatamente después, el servidor pasará a estado de escucha mucho más rápido que la primera vez.
3. El botón *restart* reinicia el servidor de red y lo vuelve a poner en estado de escucha. Es equivalente a pulsar *stop* y después *start*.
4. Señala el estado actual de la aplicación. Puede ser apagado (Off), esperando clientes (waiting) o funcionando/transmitiendo (streaming).
5. Muestra los FPS actuales a los que la aplicación es capaz de transmitir.
6. Vista previa de la imagen tratada enviada al cliente.
7. El puerto *com* es utilizado para escuchar las respuestas de clientes y para la transferencia de imágenes en el modo monopuerto. En el modo multipuerto solo recibirá respuestas.
8. Activa el modo multipuerto.

9. Puertos necesarios para el modo multipuerto, cada uno de ellos es utilizado para crear un socket por el que se envían las imágenes (uno para cada lado).
10. Activa el modo falso 3d. No es compatible con el modo multipuerto por lo que la UI no permitirá tener activos ambos modos al mismo tiempo.
11. Ajusta el valor de compresión JPEG.
12. Cambia las imágenes de lado.
13. Introduce en la imagen una etiqueta que señala a qué lado pertenece. Esta etiqueta también será enviada a los clientes.
14. En la consola se muestra lo que va ocurriendo en la aplicación, como posibles errores o pasos. Por ejemplo, que sensores han sido encontrados.

3.6 Pruebas

Se han realizado pruebas relativas a la velocidad de procesado y también sobre la calidad de imagen y reconocimiento en diferentes escenarios y con diferentes ambientes de luz.

La velocidad de transferencia se evaluará con mayor detenimiento en la fase de pruebas de la aplicación cliente, pero desde el principio se hizo evidente que la tasa de transferencia con imágenes sin codificación era insuficiente. Debido a la diversidad de colores que puede contener la imagen (dependiendo principalmente de la ropa que lleve en ese momento la persona emisora (ver Figura 20), una compresión PNG sería menos eficiente. La tasa de transferencia pasó de ser de 3-4 fps sin compresión hasta 18-26 fps en formato JPEG. Es importante destacar que dependiendo mucho de la situación de la persona emisora en la imagen se producen muchas variaciones en la velocidad. Si la persona está cerca de la cámara y, por tanto, un alto porcentaje de la imagen pertenece a la persona, la imagen tiene un tamaño superior que si el sujeto está lejos, ya que la mayor parte de la imagen sería de un color negro homogéneo, lo que favorece enormemente la tasa de compresión JPEG.

Al tratarse de un método basado en cámaras RGB la calidad de imagen resultante depende directamente, como es obvio, de la cantidad de luz en escena, en cambio, la calidad de reconocimiento no se ve afectada por la ausencia de luz sino que puede verse beneficiada, ya que el propio sensor es el encargado de emitir la luz infrarroja con la que funciona el dispositivo.

Algunos objetos pueden alterar gravemente la calidad de reconocimiento. Objetos reflectantes, como espejos o superficies pulidas, pueden alterar la forma en la que rebotan los haces de luz infrarroja emitidos por el sensor, al igual que la presencia de niebla o humo pueden provocar un efecto de deslumbramiento.

Si el emisor entra en contacto con algunos objetos físicos también es posible que se produzcan efectos no deseados como que ese objeto pase a formar parte de la imagen de la persona. Esto puede ser interesante en algunos casos, pero sin ningún tipo de regulación o control al respecto se tomará como un defecto.

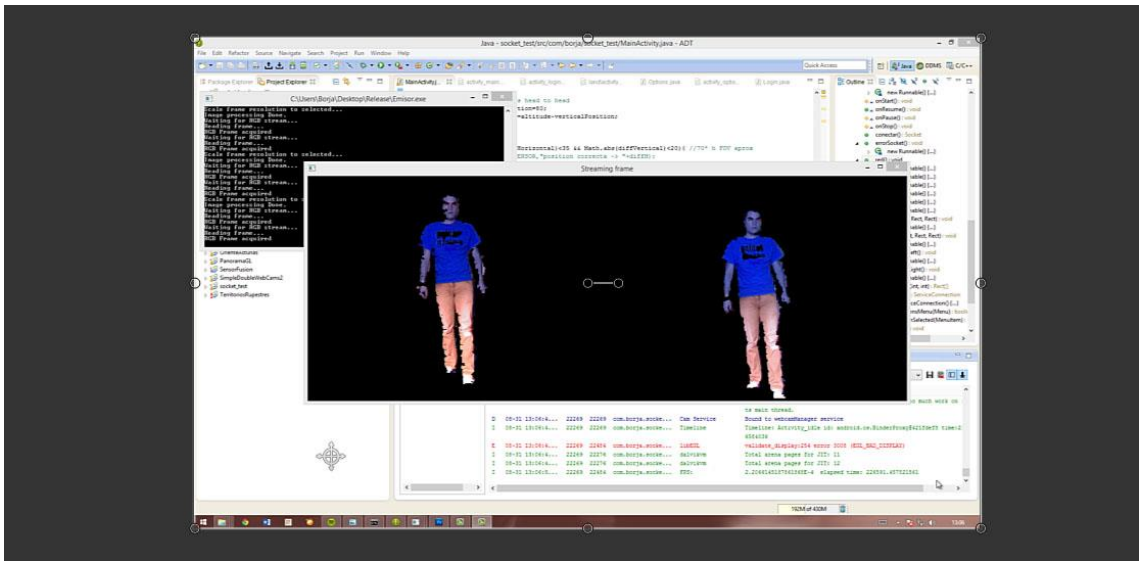

Figura 20: detalle de colores durante el reconocimiento

3.7 Conclusiones sobre el funcionamiento del servidor

La tasa de transferencia de imágenes durante las pruebas es buena: incluso por Wi-Fi se obtienen rendimientos de 26 fps sin procesamiento en el cliente, lo cual es suficiente. La velocidad de reconocimiento también es buena y la pérdida de calidad de la codificación JPEG es casi imperceptible. El principal defecto reside en la calidad con la que el sensor de profundad de *Kinect* obtiene los bordes de los objetos, en concreto en este caso, los bordes de la imagen de la persona con el resto del entorno. El corte no es todo lo limpio y definido que sería deseable sino que produce un efecto dientes de sierra. Un algoritmo de suavizado aplicado sobre la imagen podría mejorar la calidad del mismo, pero reduciría considerablemente el rendimiento.

Otro problema que perjudica gravemente la sensación de realismo en el cliente es la resolución máxima del sensor RGB de *Kinect*. Una resolución 640x480 es muy pobre para ser visualizada con un caso HMD. *Kinect* 2 podría ser una solución a este problema, ya que pasa a tener una resolución máxima de 1920x1080 pixeles, siempre y cuando se disponga de potencia suficiente en el resto de componentes.

Figura 21: Kinects capturando en estéreo

Capítulo 4: Introducción al proceso de recepción y justificación de las tecnologías empleadas

4.1 Introducción al proceso de recepción

En este capítulo se describen los objetivos más importantes del proceso de recepción, así como las tecnologías hardware y software utilizadas para completar la tarea.

Con el servidor de emisión ya completado, una persona en un lugar determinado con dos dispositivos *Kinect* y un ordenador podría iniciar el programa y dejar que un cliente se conectase y comenzase a recibir las imágenes estéreo. El proceso de recepción es más complejo que el de emisión, pues requiere de un dispositivo físico que pueda aprovechar la imagen estéreo recibida, y a la vez, que permita al usuario visualizarla sin perder de vista su entorno.

Para visualizar la imagen estéreo en 3D (imagen libre paralela o side-by-side) es necesario usar un casco HMD. Probablemente el casco HMD más famoso sea Oculus Rift, pero también hay otros cascos conocidos como los diseñados para ver cine (tales como los Sony HMZ) o el más recientemente presentado Microsoft Hololens, posiblemente el más cercano a la idea de este proyecto en su concepto.

Oculus Rift dispone de una pantalla de 7” con una resolución de 1280x800 pixeles, aunque en su versión final podría ser superior. Al colocarse el casco, el usuario puede ver la imagen a través de unas lentes especiales que permiten enfocar la pantalla a tan sólo 3 o 4 cm de distancia. Esto permite crear la sensación de estar ante una pantalla gigante.

Con un casco similar a Oculus Rift podríamos visualizar la imagen estéreo recibida desde el emisor y satisfacer la primera parte de la tarea de recepción. La segunda parte del problema consiste en permitir al usuario seguir contemplando su entorno de una forma similar a como si no llevase el casco o las gafas. Podemos solucionar este problema si construimos un casco parecido a los comentados anteriormente pero lo equipamos con un par de cámaras frontales que transmitan su imagen a la pantalla en directo. Se necesitan 2 cámaras para capturar una imagen estéreo al igual que se ha realizado con el servidor de emisión. Las cámaras estarán situadas en el frontal del casco a la altura de los ojos y a una distancia entre objetivos de 6.35cm, distancia interpupilar media (ver Figura 22).

Figura 22: Prototipo HMD durante las pruebas

El último paso consiste en mezclar ambas imágenes. La imagen de la persona emisora debe situarse sobre la imagen del entorno, de forma que quede lo más integrada posible. Hay dos técnicas de realidad aumentada que se pueden utilizar para lograr este objetivo: basada en marcadores o basada en orientación.

La técnica basada en marcadores está basada en el reconocimiento de una determinada forma en la escena que permite situar el objeto virtual con mayor precisión en el entorno; no solo el lugar está mejor definido, sino que la perspectiva también se puede determinar a partir del marcador. A priori puede parecer la técnica más interesante, pero tiene dos grandes desventajas. En primer lugar, requiere de un objeto conocido que sea reconocible por el programa y que no se confunda con el resto del entorno. Normalmente se usa una cuadricula similar a un tablero de ajedrez, como puede ser una alfombra (ver Figura 23). Pero la desventaja más importante reside en que si perdemos de vista el marcador perdemos la situación del objeto virtual, y en el caso de que la imagen sea de una persona esto es doblemente importante, ya que necesitamos un campo de visión vertical elevado. Conocer la perspectiva tampoco ayuda ya que se trata de una imagen y no de un modelo 3d, cuya captura implicaría usar un barrido en fase de emisión inviable en tiempo real a día de hoy.

Por el contrario, la técnica basada en orientación nos permite situar el objeto virtual en unas coordenadas específicas en todo momento sin necesidad de reconocer nada en el entorno. Sin embargo, la orientación puede ser imprecisa y lenta y esta es una de las razones principales por las que no suele utilizarse para este tipo de objetivos.

Figura 23: Ejemplo de RA basada en marcadores

4.2 Smartphone como dispositivo de recepción

Podemos partir de diferentes dispositivos para elaborar el casco, lo más evidente y barato puede parecer usar unas gafas 3D dedicadas a cine. Estas gafas ya incorporan la pantalla y las lentes y disponen de una entrada para la señal de video, pero habría que incorporar algún dispositivo adicional para controlar la orientación, y además, las cámaras no podrían conectarse directamente al aparato. Por otro lado, necesitaríamos un controlador externo para hacer funcionar el programa. Otro problema reside en el cableado requerido, el cual menoscaba la experiencia de usuario en términos de comodidad y naturalidad. Además de todo esto, la calidad y la resolución de este tipo de pantallas suelen ser muy limitadas.

Hoy en día casi todos los smartphones tienen pantallas de tamaño superior a las 4 pulgadas, tienen procesadores potentes, conexiones inalámbricas y disponen de algún sensor de orientación. El único problema reside en la ausencia de las cámaras frontales, ya que la gran mayoría de dispositivos sólo disponen de una cámara trasera.

Salvando esa limitación, usar como base un smartphone nos proporciona todo lo que necesitamos. Dispone de un sistema operativo y un procesador capaces de hacer un funcionar el programa cliente que necesitamos. Además, una pantalla de 5 pulgadas es suficiente para que a través de unas lentes especiales proporcione una sensación de inmersión visual (ver Figura 24), el sensor de orientación también está incluido y nos ayudará a orientar el campo de visión, y todo ello en un único dispositivo de tamaño compacto.

Figura 24: Imagen side-by-side en smartphone

El problema de las cámaras frontales puede solucionarse mediante la conexión de un par de cámaras externas. La mayoría de dispositivos disponen de un puerto USB al que es posible conectar un hub, con alimentación externa si fuera necesario, a través del cual podemos comunicarnos con un par de cámaras USB tipo Webcam.

4.3 ¿Por qué Android?

A la hora de elegir un smartphone existen varias alternativas. Teniendo en cuenta el tipo de sistema operativo, los más evidentes son los que usan Android y los que usan iOS y algo menos comunes los equipados con Windows phone.

El sistema operativo Android cumple con una serie de requisitos que lo hacen más adecuado como plataforma de recepción. Existen dispositivos con mayor potencia de procesamiento a mejor precio que la competencia y más alternativas a la hora de elegir la pantalla.

Un dispositivo iPhone no dispone de la posibilidad de conectar un hub USB OTG (USB On-The-Go) para utilizar la cámara externa. OTG es una extensión de la norma USB 2.0 que permite al smartphone actuar como host para la conexión de otros dispositivos como memorias USB, cámaras web, o ratones y teclados. En cambio, es posible encontrar facilmente hubs USB para Android con 2 puertos o más, incluso con alimentación externa.

Existe la posibilidad de obtener facilmente permisos de administrador o superusuario, lo cual se traduce en una mayor flexibilidad a la hora de afrontar desafíos como la conexión y el manejo de las cámaras externas. También es posible modificar parámetros de los controladores tanto de las cámaras como del bus USB para adaptarlo mejor a las necesidades del diseño.

Existe igualmente la posibilidad de modificar el kernel del sistema operativo para añadir módulos, controladores o librerías nativas. El kernel de Android es un kernel Linux y por tanto existe un gran número de posibilidades al respecto.

4.4 Conexión de las cámaras externas

4.4.1 Conexión y controladores

Conectar dispositivos como un teclado o un ratón a un dispositivo Android es bastante sencillo, pues son dispositivos que pueden utilizar controladores estandarizados para sus funciones básicas. Una cámara es un dispositivo más complejo, ya que puede entregar las imágenes a varias velocidades, con diferentes resoluciones y en distintos formatos, y el mapeo de las imágenes al dispositivo al que está conectado puede ser muy diferente dependiendo de la cámara usada.

La funcionalidad de los dispositivos USB está clasificada en varias clases. Cuando un dispositivo es conectado, dependiendo de su código de clase, comienza la carga de los controladores más adecuados. Existe una clase dedicada a los dispositivos de video como cámaras, sintonizadores de TV y otros dispositivos analógicos. Esta clase se denomina USB Video Class (UVC).

En Linux existe un módulo denominado UVCvideo cuya misión es proporcionar todos los componentes software necesarios para soportar dispositivos de video que cumplan la especificación UVC (UVC compliant). Este módulo va ligado a la API V4L2 (video for Linux 2), la cual implementa. Tanto el módulo UVCvideo como la API V4L2 suelen estar integrados en el kernel de Linux.

Ambos componentes serán necesarios para interactuar con las cámaras web ya que Android no dispone apenas de soporte para cámaras externas, por lo que se opta por unas que cumplan esta especificación. Como se ha dicho, tanto V4L2 como UVCvideo suelen estar incluidos en el kernel de Linux pero no ocurre así en muchos sistemas Android.

En el dispositivo de pruebas, Samsung Galaxy S3, es necesario recompilar el kernel para incorporar la funcionalidad con las siguientes opciones.

```
CONFIG_VIDEO_DEV=y
CONFIG_VIDEO_V4L2_COMMON=y
CONFIG_VIDEO_MEDIA=y
CONFIG_USB_VIDEO_CLASS=y
CONFIG_V4L_USB_DRIVERS=y
CONFIG_USB_VIDEO_CLASS_INPUT_EVDEV=y
```

Es posible conocer si se dispone de V4L2 correctamente instalado si al conectar una cámara o un nuevo dispositivo de tipo video su descriptor es añadido al directorio /dev. Por ejemplo, /dev/video0.

4.4.2 El problema del ancho de banda del bus USB

Para capturar la imagen estéreo se necesita usar dos cámaras USB externas iguales, pero este tipo de dispositivos pueden requerir una parte importante del ancho de banda disponible en el bus. La mayoría de dispositivos no van a soportar el uso de más de una Webcam al mismo tiempo (ver Figura 25).

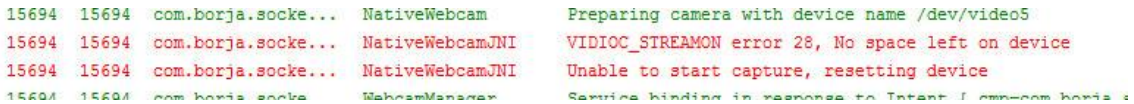

Figura 25: Error de ancho de banda insuficiente

Las Webcams compatibles con la especificación UVC pueden entregar las imágenes en varios formatos. Los formatos disponibles dependerán del modelo de cámara, pero normalmente todas son capaces de trabajar con imágenes sin compresión YUV y comprimidas JPEG (MJEG).

Cuando una cámara es inicializada, solicita al bus una parte del ancho de banda disponible. Si es concedido, ese espacio queda reservado y sólo ese dispositivo puede usarlo. Es un problema común que cuando un dispositivo se conecta al mismo bus en el que una cámara está funcionando éste no es capaz de iniciarse (ver Figura 25). En un PC esto se soluciona fácilmente conectando el dispositivo a otro puerto USB que funcione sobre otro bus, pero en un smartphone solo tenemos uno.

La solución a este problema pasa por forzar al controlador a calcular el ancho de banda real necesario (para la resolución y frecuencia seleccionada) e ignorar la cifra solicitada por la cámara. Esto se hace modificando un parámetro del controlador UVCvideo denominado *quirks*. El parámetro *quirks* sirve para ajustar el controlador para trabajar con determinadas cámaras que no cumplen la especificación UVC completamente o que tienen bugs, lo cual ocurre más frecuentemente de lo que cabría esperar. Existen 9 valores posibles del parámetro quirks, el valor 128, o 0x00000080 en hexadecimal, es el que obliga a que el controlador estime el tamaño real de ancho de banda requerido por una imagen sin compresión.

Otra acción evidente es cambiar el modo de video a MJPEG. Sin embargo, esto no será compatible con la opción anterior. No es posible calcular el ancho de banda requerido por una sucesión de imágenes MJPEG ya que cada imagen comprimida tendrá un tamaño distinto, lo que obligaría al driver a trabajar con el valor de ancho de banda por defecto de la cámara y a no funcionar con dos cámaras al mismo tiempo.

Para cambiar el valor *quirks* necesitamos permisos de superusuario en Android. Podemos ejecutar entonces la siguiente instrucción:

echo 0x00000080 > /sys/module/uvcvideo/parameters/quirks

Este cambio no funcionará después de la inicialización del controlador, por lo que es necesario conectar las cámaras después de cambiar el valor.

4.4.3 Permisos de dispositivo

Conectadas las cámaras hay que comprobar si disponen de los permisos adecuados para que una aplicación pueda usarlas. En la carpeta /dev estarán los descriptores de los dispositivos de video. En smartphones modernos que ya disponen de varias cámaras, éstos no tienen por qué ser video0 y video1. Pero la mejor forma de comprobarlo es listar los descriptores antes y después de la conexión de las cámaras.

Con la instrucción "ls –l descriptor" es posible conocer los permisos y sus propietarios (ver Figura 26). Ocurre en Android que si dos cámaras son conectadas, una de ellas puede tener permisos adecuados, pero la segunda suele ser propiedad de root. En otras ocasiones puede que el propietario sea correcto y los permisos no.

Para asegurarse de que ambas cámaras tienen propietario y permisos adecuados podemos ejecutar las siguientes instrucciones para ambos descriptores:

chmod 666 /dev/video5

chown system:system /dev/video5 o *chown system:camera /dev/video5*

Los permisos serán restablecidos cada vez que la cámara es conectada. Para que el descriptor sea creado con los permisos y propietario adecuados es posible crear una regla en el fichero /uevent.rc. No obstante, este fichero también será restablecido durante la carga del sistema operativo.

Figura 26: Permisos y propietarios de cámara

4.4.4 Kernel ramdisk

Tanto el parámetro *quirks,* que pertenece a un módulo enlazado estáticamente al kernel, como cualquier modificación realizada en el archivo uevent.rc, será eliminados durante la carga del sistema. Este fichero es directamente cargado del ramdisk del kernel el cual no es posible modificar.

Para poder hacer los cambios permanentes, la única forma de proceder es crear un script de inicio en la carpeta init.d que modifique ambos ficheros en cada uno de los inicios del sistema.

4.4.5 Estado de SELinux

Security-Enhanced Linux (SELinux) es un módulo de seguridad para el kernel que soporta políticas de seguridad para el control de acceso.

En Android este modelo de seguridad está basado en el concepto de aislamiento de procesos (sandbox) y proporciona un control de acceso obligatorio para todos los procesos, incluso los que funcionan con privilegios de superusuario. Este modelo se basa en la negación por defecto. Cualquier cosa que no esté explícitamente permitida será denegada (por ejemplo un intento de comunicación con el driver de un dispositivo desde un determinado contexto).

A partir de Android 5.0 el estado de SELinux ha pasado a ser de plena vigilancia (full enforcement) por defecto. Esto provoca que cualquier intento de comunicación con el driver de la cámara sea denegado y bloqueado (ver Figura 27).

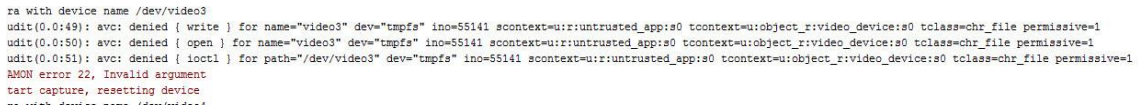

Figura 27: Bloqueo de acceso a cámara de SELinux

Para poder utilizar las cámaras a partir de Android 5.0 se debe modificar este estado a permisivo. Con privilegios de superusuario podremos hacerlo con la siguiente instrucción:

Su - c setenforce 0

O en el caso de que no estemos en el contexto adecuado:

Su --context u:r:system_app:s0 –c "setenforce 0"

Es importante señalar que aún estando en un estado permisivo continuaremos recibiendo las denegaciones de acceso por parte de SELinux aunque no se esté produciendo ningún bloqueo realmente.

4.4.6 Otros contratiempos

Algunos sistemas Android pueden experimentar la suspensión de dispositivos si en un determinado tiempo no son utilizados. Si esto ocurre, el acceso al dispositivo será rechazado. Para evitarlo es posible desactivar la auto-suspensión en el controlador USB con la siguiente instrucción:

echo -1 > /sys/module/usbcore/parameters/autosuspend

Algunos dispositivos de LG presentan un bug en el controlador USB que consiste en que, aún existiendo ancho de banda suficiente en el bus, no son capaces de reservarlo, haciendo imposible la utilización de ambas cámaras con el estándar USB 1.0. En este caso será obligatorio utilizar un hub OTG USB 2.0.

El error que presentan estos dispositivos puede comprobarse en el log dmesg:

Usb1-1.x: Not enough bandwith for altsetting 5 (2048B/frame)

El modo altsetting puede variar dependiendo de la resolución y velocidad de la cámara.

4.5 Sensor Fusión

Prácticamente todos los Smartphone Android disponen de un magnetómetro y de un acelerómetro y muchos disponen también de un giroscopio. Estos sensores se utilizan muy frecuentemente para controlar juegos o funciones avanzadas de la cámara como "photosphere" y también se está utilizando más recientemente en el desarrollo de técnicas

de navegación asistida en interiores. Todas estas funciones ayudan a ampliar las posibilidades funcionales del dispositivo.

Para conocer la orientación de un dispositivo hay que tener en cuenta que éste puede rotar sobre 3 ejes respecto de sí mismo (ver Figura 27) o respecto del mundo. Un ejemplo de giro respecto de sí mismo es el giroscopio, el cual mide velocidades angulares respecto de su posición inicial (posición de referencia). En cambio, un ejemplo de orientación cuyas coordenadas dependen del mundo es el magnetómetro, cuyo valor 0 indica el norte magnético.

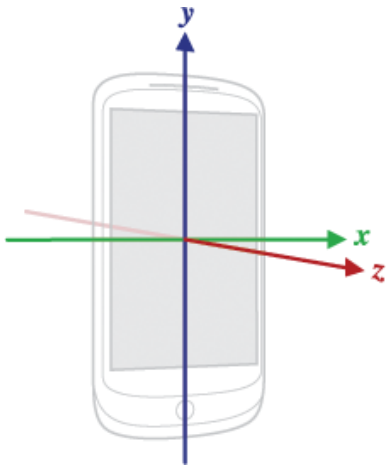

Figura 27: Ejes de orientación

Para conocer la orientación de un dispositivo con 3 ejes de libertad es necesario conocer dos vectores de referencia. En la API de Android el objeto sensor de orientación usa una modalidad de “sensor fusión” que consiste en combinar el acelerómetro y el magnetómetro. El acelerómetro es utilizado para conocer el vector de gravedad, mientras que el magnetómetro se utiliza para conocer el vector norte. El principal problema de este método es que es lento y poco eficaz.

La señal entregada por el acelerómetro puede contener ruido y hay que integrar esta señal dos veces si se quiere obtener la variación de posición. Una consecuencia de integrar la señal es que filtra el ruido; sin embargo, esto se traduce en deriva o desfase. La señal del acelerómetro se integra una vez para obtener la velocidad, lo que añade una considerable deriva, y de nuevo se integra para conocer la posición, lo que añade mucha más deriva. Además, si lo que queremos es conocer la variación de posición con el acelerómetro, tenemos que eliminar el efecto de la gravedad sobre el aparato. Esto se llama compensación de gravedad y requiere de mucha precisión. Un pequeño error de tan solo 1º en el vector de gravedad a la hora de hacer el cálculo produce una gran deriva una vez la señal ha sido integrada dos veces. Esto convierte al acelerómetro en un sensor de orientación poco preciso.

Con la señal del magnetómetro ocurre lo contrario. La señal del magnetómetro no tiene deriva pero si tiene mucho ruido (ver Figura 28) y esto ocurre por dos razones: la primera razón es que el entorno es muy ruidoso, hay muchos aparatos eléctricos y materiales que pueden alterar el escenario con fuerzas magnéticas y la segunda razón es que esta señal no se integra.

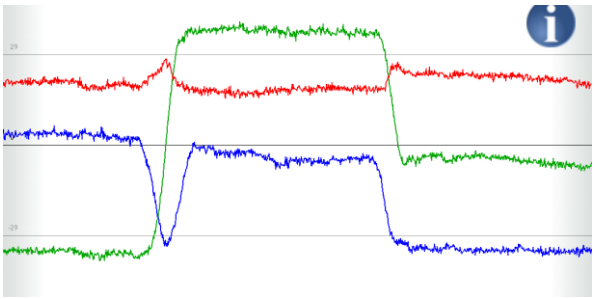

Figura 28: Señal de magnetómetro LG G2.

La forma en que la API de Google trabaja con estos sensores hace que muchos de estos efectos adversos sean minimizados (ver Figura 29); sin embargo, produce un retardo elevado en la entrega exacta del resultado.

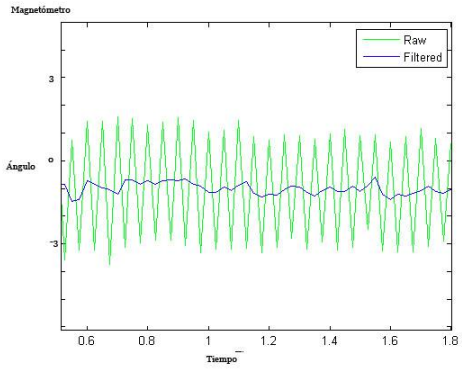

Figura 29: Filtrado del ruido del magnetómetro

El giroscopio mide la velocidad angular de rotación en un cambio de posición del dispositivo (ver Figura 30). Para obtener el ángulo de cambio, por tanto, hay que integrar en el tiempo. Esto es equivalente a multiplicar las velocidades por el intervalo de tiempo que sucede entre dos entregas de valores de la señal. Esto proporciona una señal limpia sin apenas ruido pero con cierto efecto deriva (aunque mucho menos que el acelerómetro). El giroscopio tiene entonces una buena respuesta dinámica pero añadirá un cierto desfase con el paso del tiempo, ya que el cálculo nunca es perfecto. La principal causa de deriva es debida a que es muy sensible a errores en la medición del intervalo de tiempo, por eso muchas veces el cálculo se realiza en hardware.

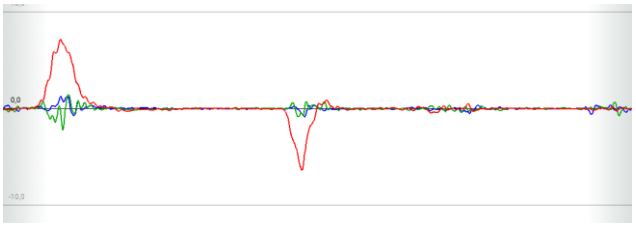

Figura 30: Señal RAW giroscopio LG G2

¿Cómo podemos entonces aprovechar las ventajas de cada uno de los sensores? Podemos usar la buena respuesta dinámica del giroscopio para obtener un giro rápido y usar el acelerómetro y el magnetómetro para corregir la deriva acumulada con el paso del tiempo (una especie de calibración temporal). O lo que es lo mismo, usar el giroscopio para intervalos cortos de tiempo y el magnetómetro/acelerómetro para intervalos más largos. Fijándose en una señal cuyo eje x simboliza el paso del tiempo, esto es como aplicar un filtro de paso bajo a la señal del acelerómetro o magnetómetro y uno de paso alto a la señal del giroscopio, antes de sumar ambas señales.

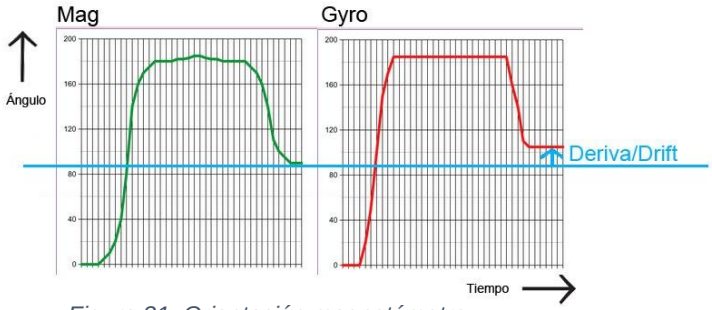

Figura 31: Orientación magnetómetro y giroscopio

En la imagen figura 31 puede verse la señal filtrada del magnetómetro para el siguiente movimiento: el dispositivo gira 180º en sentido horario en uno de los ejes. Tras un tiempo estable en esa posición se produce un giro en sentido anti-horario de 90º. A la derecha vemos la señal integrada en el tiempo del giroscopio para ese mismo movimiento. La señal del giroscopio es mucho más rápida que la del magnetómetro, la forma redondeada indicaría lentitud en la señal, que tarda en estabilizarse, pero como podemos ver, al final de la señal existe una deriva acumulada que da como resultado una orientación con unos 15º de error. Aplicando los filtros y sumando ambas señales se obtendría una señal similar a la del giroscopio pero sin errores, ya que durante la estabilización del movimiento la señal del giroscopio se iría filtrando cada vez más, provocando que el valor resultante sea el del magnetómetro (ver Figura 32).

Figura 32: Señal fusionada

4.6 OpenXC Android Webcam

OpenXC es una API que surge como alternativa al protocolo de comunicación OBD para vehículos. Mientras que OBD está basado en señales de bajo nivel como por ejemplo la posición del acelerador, OpenXC busca interpretar y entregar como salida valores de más alto nivel. Uno de los proyectos de OpenXC en desarrollo está relacionado con la conexión de una cámara web sobre un sistema operativo Android. Este trabajo está basado en la librería para Android en código nativo SimpleWebCam de *neuralassembly*. Se usará esta librería como punto de partida para la inicialización de las cámaras y será adaptada para que sea capaz de trabajar con dos cámaras en lugar de una.

Capítulo 5: Desarrollo del software de recepción

5.1 Introducción

En esta etapa del proceso de comunicación se recibe la imagen procesada en el servidor de emisión en un cliente Android y se mezcla con la imagen de las cámaras frontales. Una vez mezcladas las imágenes, se muestran en la pantalla del dispositivo en forma *side-by-side* con el objetivo de visualizarlo en un casco HMD con las lentes adecuadas. El objetivo consiste en desarrollar una aplicación que realice todas estas tareas.

En esta sección se detalla lo más importante de cada fase del desarrollo del software.

5.2 Análisis de requisitos

Los requisitos funcionales de la aplicación son:

- Debe ser capaz de conectarse a un servidor de emisión, especificando su IP y puerto/s, usando la conexión Wi-Fi del dispositivo.
- Una vez conectada al servidor debe ser capaz de recibir imágenes a través de un socket TCP codificadas en formato JPEG.
- Debe iniciar dos cámaras tipo Webcam conectadas al dispositivo por USB
- Debe mezclar la imagen de la persona recibida del servidor con la imagen procedente de las cámaras de forma que parezca integrada en la escena.
- El lugar o posición de la forma de la persona recibida en la escena se determina a partir de los valores proporcionados por los sensores de orientación
- Una vez procesada la escena debe mostrar la imagen estéreo en la pantalla del dispositivo
- La posición deseada de la persona en el entorno está controlada por el usuario mediante la introducción del valor “azimuth” de forma manual en la UI.
- Mientras se encuentre transmitiendo, el dispositivo no debe suspenderse ya que no tendremos acceso al mismo dentro del casco HMD.

Los requisitos no funcionales son:

- La conexión con el servidor se realizará mediante sockets
- La transferencia de imagen se lleva a cabo mediante una conexión usando el protocolo TCP sobre el puerto especificado.

- Para el procesamiento de las imágenes se usan funciones de la librería *OpenCV* para Android, que será cargada de forma asíncrona antes de la conexión con el servidor.
- El procesamiento de las imágenes se realiza de forma paralela a la transferencia de la imagen siguiente.
- El procesamiento de la imagen de cada lado se realiza de forma paralela.
- Debe implementar un método de orientación mejorado magnetómetro + acelerómetro + giroscopio.
- La conexión con las cámaras frontales USB se efectúa mediante una librería implementada en código nativo (JNI).

5.3 Diseño

La fase de diseño se divide en varias partes: Transferencia de imágenes, cámaras frontales, sensor fusión, procesamiento de imagen e interfaz e hilos de ejecución.

5.3.1 Transferencia de Imágenes

Para la transferencia de imágenes se elige un protocolo de comunicación síncrono. Cuando la aplicación esté lista para recibir una imagen, ésta enviará al servidor una señal de reconocimiento que será un byte cualquiera. El servidor entonces enviará la imagen procesada más reciente.

Inmediatamente después de enviar esta señal, la aplicación espera recibir el tamaño total, en bytes, de la imagen. Este dato debe ser conocido antes de empezar a recibir la imagen, ya que al tratarse de una imagen codificada en formato JPEG, su tamaño será variable.

Una vez conocido el tamaño en bytes, ya es posible recibir la imagen sin necesidad de analizar el contenido en busca del código de finalización. Se reciben los datos del socket y se almacenan en un array de bytes.

Al igual que se hizo en el servidor, usaremos *OpenCV* para descodificar la imagen en un objeto de imagen de tipo RGB888.

5.3.2 Cámaras frontales

Si las cámaras disponen de los permisos y propietario adecuados y el parámetro *quirks* del módulo UVCvideo está configurado a su valor 128, entonces es posible comenzar a usar las cámaras desde la aplicación (Ver Figura 33).

La API de Google solo contempla el uso de las cámaras nativas del dispositivo; para poder utilizar las cámaras externas desde nuestra aplicación es necesario inicializarlas usando código nativo, tal y como se haría en cualquier máquina Linux.

Como se ha comentado anteriormente, la base de este código procede de la librería OpenXC/webcam-manager, del cual se han tomado los métodos para la inicialización y mapeo de memoria de las cámaras. A continuación se describen brevemente los métodos o funciones principales y las modificaciones añadidas (Los parámetros de las funciones no se detallan, pero si se hablará de los más importantes en la descripción).

El código se divide en 4 clases:

- webcam.c: contiene los métodos JNI de comunicación con la máquina virtual java. Sus cuatro funciones son:
    - startCamera: recibe como parámetro el nombre del descriptor de dispositivo de la cámara así como el ancho y alto de la imagen.
    - stopCamera: Fuerza la llamada a los métodos de liberación de memoria de todas las clases.
    - cameraAttached: comprueba si la cámara sigue inicializada.
    - loadNextFrame: Recibe como parámetro el objeto Bitmap de Android donde se copiará la imagen recibida de la cámara.
- video_device.c: Contiene las funciones de inicialización de la cámara. Las funciones de esta clase son llamadas principalmente por el método startCamera.
    - open_device: Comprueba si el descriptor recibido existe y tiene los permisos adecuados.
    - init_device: Mediante llamadas xioctl (ioctl es una llamada del sistema en Linux que permite comunicarse con el driver de un dispositivo a través de su descriptor) se inicializa el dispositivo como tipo V4L2 de captura y streaming.
    - init_mmap: La transferencia de imágenes se realiza mediante mapeo de memoria desde el buffer de la cámara a un buffer local. Esta función se encarga inicialmente de solicitar al driver el tamaño requerido por el buffer y de reservar ese espacio en memoria. Una vez reservado, se realiza el mapeo mediante *mmap* que es una llamada POSIX cuyo cometido es mapear el buffer de un dispositivo en la memoria local asignada.
    - uninit: esta función libera el buffer local asociado al mapeo del dispositivo.
- capture.c: Contiene las funciones de captura de la imagen actual. Estos métodos son llamados principalmente por loadNextFrame.
    - process_camera: es el método principal de la clase. Recibe como parámetro un puntero al buffer local mapeado, al buffer RGB y al buffer YUV. Mediante una llamada "select" espera a que el dispositivo esté listo para la operación de entrada/salida. Después llama al método read_frame de la misma clase el cual fuerza el mapeo de la imagen de cámara mediante una llamada ioctl. En la última fase del método se llama a la

función yuyv422_to_argb de la clase yuv.c la cual convierte la imagen obtenida de la cámara en formato YUV a formato ARGB8888 que es el formato que corresponde al tipo de dato Bitmap de Android.

    - stop_camera: Libera los buffer de imagen RGB y YUV.

- yuv.c: Esta clase contiene un método nativo de conversión de imagen YUV a formato RGB con canal alfa.

    - yuyv422_to_argb: recibe como parámetro el buffer local mapeado, el buffer YUV y el buffer RGB y realiza la conversión.

Todas estas funciones están diseñadas para trabajar con una sola cámara. Aunque el descriptor de la cámara pueda especificarse como parámetro, esto sólo sirve para elegir el dispositivo de uso. Todas las variables y buffers deben duplicarse para trabajar con ambas cámaras.

La parte java de la librería contiene 3 clases principales: NativeWebcam, WebcamManager y WebcamPreview.

- NativeWebcam: Es una clase que contiene las llamadas a las funciones de código nativo y realiza la carga de la librería. Al tratarse de una librería dinámica no es posible cargar dos librerías distintas en dos objetos de la misma clase ya que una de las llamadas será ignorada. En su lugar se crearán dos clases NativeWebcam cada una empleada en controlar una de las cámaras. El código nativo será duplicado en dos librerías de forma que dependiendo del objeto instanciado de cada clase se comunique con su interfaz JNI correspondiente a la cámara seleccionada.

- WebcamManager: Es un servicio de Android al que es posible conectarse y realizar peticiones para obtener una imagen de la cámara. Se modificará esta clase para añadir la instanciación del objeto NativeWebcam y NativeWebcam2, así como los métodos para la obtención de la imagen de la segunda cámara "getFrame2()". Las peticiones getFrame() y getFrame2() realizarán entonces una llamada a sus correspondientes métodos nativos *load_next_frame* descrito anteriormente, lo cual tiene un retardo considerable hasta el retorno del Bitmap resultado. Además, el servicio contiene un solo hilo de ejecución, así que mientras atiende una petición de imagen de una de las cámaras, no podrá atender la otra. Para solucionar este problema se implementan dos tareas periódicas en hilos paralelos en el servicio que obtendrán las imágenes de las cámaras a un ritmo de 25 imágenes por segundo y las almacenarán en dos objetos de tipo Bitmap. Cuando la aplicación haga una petición de una de las imágenes de las cámaras al servicio, este solo tendrá que retornar el Bitmap más reciente.

- WebcamPreview: Contiene código relativo a la creación de una interfaz gráfica para visualizar una de las cámaras, en este caso, esta clase es eliminada.

Es importante destacar que el valor de resolución con el que inicializamos las cámaras no puede ser cualquiera y depende del modelo. Para ver una lista de formatos y resoluciones soportados podemos ejecutar el siguiente comando en Linux: v4l2-ctl –list-formats-ext.

Una vez realizadas las modificaciones, para obtener las imágenes de las cámaras, la aplicación debe conectarse al servicio WebcamManager. Con el servicio enlazado con la

actividad donde deseamos utilizar las imágenes, esta llamará a los métodos getFrame() y getFrame2() para obtener la imagen correspondiente a cada cámara.

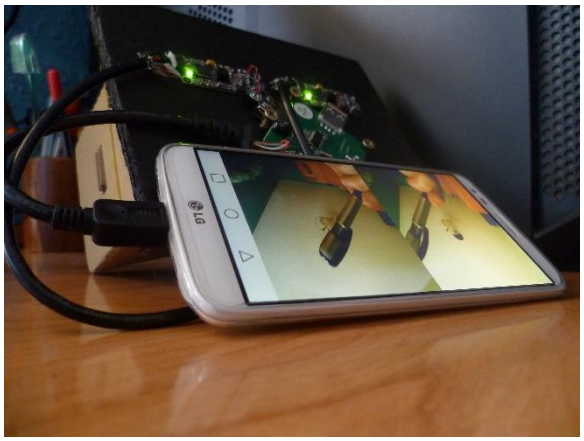

Figura 33: Detalle de conexión de cámara

5.3.3 Sensor Fusion

No existe mucha información sobre cómo implementar sensor fusión en Android. Posiblemente el mejor tutorial que existe sobre una implementación valida de código abierto sea la que Paul Lawitzki realizó para su tesis de master y que se encuentra disponible en la web “code-project.com”.

Se trata de una implementación bastante sencilla basada en aplicar un filtro pasa altas a los valores de orientación del giroscopio y un filtro pasa bajas al método de orientación común de la API de Google basada en los valores del sensor de aceleración y el magnetómetro.

El primer paso consiste en registrar los “listener” para las tres clases de sensores. La velocidad de muestreo se encuentra configurada bajo la constante SENSOR_DELAY_FASTEST que configurará la tasa de refresco de los sensores en su valor más rápido posible. Esto entrega la máxima precisión en la lectura de cada sensor pero conlleva una sobrecarga considerable en CPU. En este caso no se va a necesitar una precisión tan elevada, ya que la precisión en la superposición de las imágenes estará limitada entre otros factores por la resolución de las mismas. La tasa de refresco configurada en su valor más rápido depende de cada sensor y de cada dispositivo. Para los sensores de aceleración y magnetómetro se encuentra entorno a los 100Hz. El problema de usar el valor más rápido para el giroscopio es que su tasa de refresco puede superar los 200Hz e incluso en algunos dispositivos puede alcanzar 800Hz, con su correspondiente sobrecarga en CPU y consumo de batería. Sin embargo, hay que tener cuidado con el valor configurado, ya que, como se verá más tarde, la integración de las velocidades angulares se realiza en el mismo programa y el intervalo de tiempo se mide entre la recepción de dos eventos, lo que dependerá directamente del valor configurado. A mayor tiempo entre cálculos mayor tasa de error acumulado.

La siguiente constante de velocidad es SENSOR_DELAY_GAME, la cual suele limitar tasa de refresco a unos 50Hz para cualquier sensor. Es una tasa de refresco muy

baja para el giroscopio. Como se ha explicado anteriormente, se van a filtrar las altas frecuencias para los sensores de aceleración y magnetómetro así que no tiene sentido tener la misma tasa de refresco en los tres sensores. Desde la API 11 de Android es posible especificar un valor manualmente para los mismos. Para el giroscopio una tasa de refresco de 77hz será suficiente para el propósito del trabajo, mientras que para los demás se configurará una frecuencia en torno a los 10Hz.

La API de Android implementa los métodos para obtener los valores de orientación en grados de los sensores acelerómetro y magnetómetro. A partir de los valores recibidos en los eventos existe un método de la clase SensorManager para generar la matriz de rotación (getRotationMatrix()). Para convertir la matriz de rotación en los valores de orientación de los 3 ejes existe un método de la misma clase denominado "getOrientation", al cual se pasa como parámetro la matriz y un array de tres flotantes donde se almacenará el resultado. Dependiendo del método utilizado puede ser más interesante tener la orientación en una matriz de rotación o en los valores de orientación de los 3 ejes.

Cuando se recibe un evento del giroscopio se produce una llamada a la función "gyroFunction" con los datos del evento. Está función realiza varias tareas importantes:

- La matriz de rotación del giroscopio se inicializa como matriz identidad. Si es la primera vez que se llama a la función se tomarán como valores iniciales los obtenidos por los otros sensores. Se multiplica entonces la matriz identidad por la matriz de rotación de los sensores acelerómetro/magnetómetro (posición inicial = posición de referencia).

- Se llama a la función getRotationVectoFromGyro la cual es una copia de la misma función existente en la documentación de Android y que realiza la integración de las velocidades angulares dando como resultado un vector de rotación expresado como cuaternión. Esta función requiere conocer el intervalo de tiempo sucedido entre la entrega de datos del último giro y el actual, es decir, del intervalo de tiempo entre eventos. Así que después de obtener el resultado de la función se registra el tiempo actual para calcular el intervalo hasta el próximo evento.

- Se transforma el vector de rotación en matriz de rotación.

- Se multiplica la matriz de rotación de giroscopio obtenida en el evento anterior con la actual. Hay que tener en cuenta que es posible multiplicar matrices de rotación entre sí para representar una secuencia de rotaciones, así que de esta manera se representa la rotación total obtenida a partir de varios eventos del giroscopio.

- Con el mismo método de la case SensorManager utilizado con la matriz de rotación del magnetómetro/acelerómetro es posible transformar la matriz en los valores de orientación de los tres ejes.

Una vez se conoce la orientación en 3 ejes de los sensores acelerómetro/magnetómetro y del giroscopio se puede proceder a realizar el cálculo sensor fusión. Está tarea se realiza en un hilo temporizado que se ejecuta una vez cada 30ms. La frecuencia de corte de los filtros aplicados a los valores de los sensores depende directamente de la frecuencia de este cálculo y por eso debe ejecutarse de forma periódica. No hay una fórmula definida para obtener el valor de filtraje más adecuado.

30ms equivale a una frecuencia de cálculo de 33hz, muy superior a la tasa de refresco de imagen obtenida (FPS). Como se necesita liberar al procesador de trabajo, se

amplía este intervalo hasta los 40ms, lo que se corresponde con una tasa de actualización de 25Hz. Como se verá posteriormente en las pruebas, el coeficiente de filtraje deberá reducirse levemente para adaptarse a la nueva frecuencia.

El resultado del cálculo se almacena en un array de tres flotantes llamado "fusedOrientation". No hay que olvidar que el proceso se basa en la corrección de la deriva (drift) del giroscopio. Por eso el último paso será sobrescribir la matriz de rotación del giroscopio con los nuevos valores obtenidos.

5.3.4 Procesamiento de imagen

Con todas las imágenes en memoria y los valores de orientación en 3 ejes obtenidos en el cálculo sensor fusión, podemos proceder a la mezcla de imágenes. Las funciones de tratamiento de imágenes se realizarán apoyándose en la librería *OpenCV*.

En *OpenCV* las imágenes se almacenan en objetos *Mat*. Un objeto Mat consta de dos partes: una cabecera que contiene datos como las dimensiones, el formato y otras características de la imagen y un puntero a la matriz que contiene los valores de los pixeles. Es importante conocer que cada objeto *Mat* tiene su propia cabecera pero que dos objetos *Mat* distintos pueden tener un puntero a la misma matriz de datos de la imagen. Lo interesante de esto es que es posible crear subsecciones de una imagen completa con el objetivo, por ejemplo, de modificar sólo una parte de ella.

La carga de la librería *OpenCV* en la aplicación se puede realizar de dos maneras. Se puede incluir de forma estática, en cuyo caso todos los binarios de la librería estarían incluidos en la aplicación, o mediante inicialización asíncrona. La inicialización asíncrona accede a las librerías *OpenCV* de forma externa a la aplicación. En el caso de Android, las librerías se instalan como una aplicación aparte desde Google Play Store. La carga estática no está recomendada para producción. Por el contrario, la carga asíncrona se encarga de seleccionar las librerías más adecuadas para cada arquitectura de procesador.

Por tanto, lo primero que debe conseguir la actividad que realiza el tratamiento de imagen es cargar esta librería de forma asíncrona.

Con la librería cargada es posible comenzar a usar algunas de sus funciones. Se ha comentado que se disponen ya de las 4 imágenes y de los valores de orientación, sin embargo, la imagen estéreo procedente del servidor emisor se encuentra en un array de bytes en formato JPEG.

Podemos obtener un objeto MatOfByte a partir del array de bytes e interpretarlo después como imagen JPEG de la siguiente manera:

```
MatOfByte raw = new MatOfByte(arrayBytes);

Mat A = Highgui.imdecode(raw, 1);  //decodifica la imagen JPEG

raw.release();
```

En este punto el objeto A contiene la imagen valida en formato BGR888. *OpenCV* invierte el orden de los colores azul y rojo (RGB -> BGR). En el servidor ya se ha tenido en cuenta esta peculiaridad y se ha enviado la imagen ya en este orden de colores con el objetivo de descargar al cliente de trabajo. Si se hubiera enviado en formato RGB sería

necesario cambiar el orden antes de comenzar a tratar la imagen. La última instrucción libera el espacio reservado por el objeto raw.

La mezcla de la imagen estéreo del servidor con las imágenes de las cámaras debe realizarse de forma separada para cada lado, ya que como se ha explicado, no comparten las posiciones de los elementos. En el modo de un solo puerto es posible separar la imagen estéreo en la imagen derecha e izquierda dividiendo a lo ancho por la mitad.

```
Rect roiHalfLeft = new Rect(0, 0, 640, 480);

Rect roiHalfRight = new Rect(640, 0, 640, 480);

final Mat iLeft = new Mat(A, roiHalfLeft);

final Mat iRight = new Mat(A, roiHalfRight);
```

Al tratarse de dos imágenes de 640x480, es posible crear dos ROIs (En *OpenCV* ROI representa una parte de la imagen completa, *region of interest*) rectangulares de esa dimensión. Asignaremos cada ROI a un objeto *Mat* distinto.

En este punto, justo antes de la mezcla de imágenes, se necesita conocer los valores del sensor de orientación que determinarán como ha de mezclarse la imagen. Las imágenes son mezcladas si el sujeto emisor está dentro del campo de visión de las cámaras. Para saberlo es necesario conocer el valor azimuth y el valor altitud al que se encuentra el casco (Ver Figura 34).

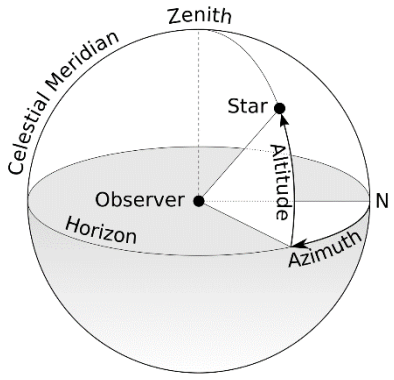

Figura 34: Sistema de coordenadas esféricas

Se supone que si una persona se encuentra frente a ti, más o menos, dependiendo de su altura, su cara estará frente a la tuya. Por eso el valor de altitud no será configurable. Este mismo valor dependerá de la distancia, pero *Kinect* no es muy flexible en este aspecto, por lo que supondrá que el sujeto emisor se encuentra a unos 3 metros. El valor azimuth en cambio será establecido en la interfaz gráfica, brindando al usuario la posibilidad de colocar la forma de la persona virtual donde quiera. Mediante la comparación de estos valores y los obtenidos por el método sensor fusion es posible determinar si el sujeto emisor está en el momento actual en escena, y si es así, en que parte de ella está.

Para determinar si está en la escena hay que tener en cuenta el campo de visión (FOV) de las cámaras, que es de unos 70º horizontal y unos 50º vertical en este caso. Este valor depende principalmente de la distancia focal de las cámaras, distancia entre la lente

y el sensor. El tamaño o escala con la que la forma de la persona entra en escena depende de lo cerca o lejos que el sujeto emisor esté de los *Kinect* y no será controlado por el programa. Entonces, el usuario introduce el punto azimuth donde desea situar al sujeto emisor alrededor de sí mismo. Se tiene en cuenta un campo de visión de 70 grados; esto supone que el sujeto estará en escena si la diferencia entre esa posición y la actual del dispositivo difiere en menos de 35 grados (ver Figura 35).

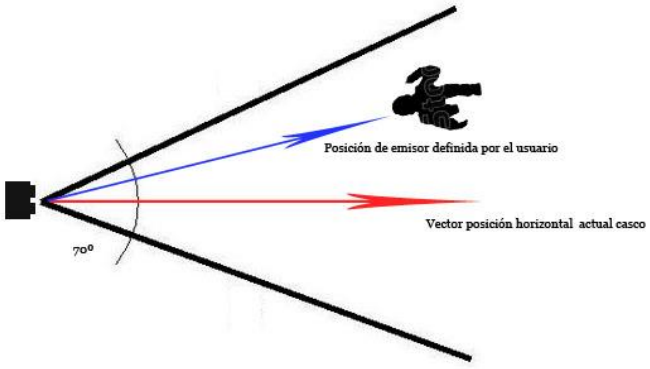

Figura 35: Colocación de la persona virtual en escena

Si la persona no está en escena, el programa carga directamente la imagen de las cámaras sin ningún tipo de procesamiento. En caso contrario, se superpone la imagen de la persona emisora sobre la imagen de las cámaras en el lugar que corresponda. Esto depende de los 3 ejes de orientación.

El valor azimuth ayuda a determinar la posición horizontal del emisor, o lo que es lo mismo, de su imagen. Esta tendrá un desfase horizontal respecto de la imagen de las cámaras antes de superponerse.

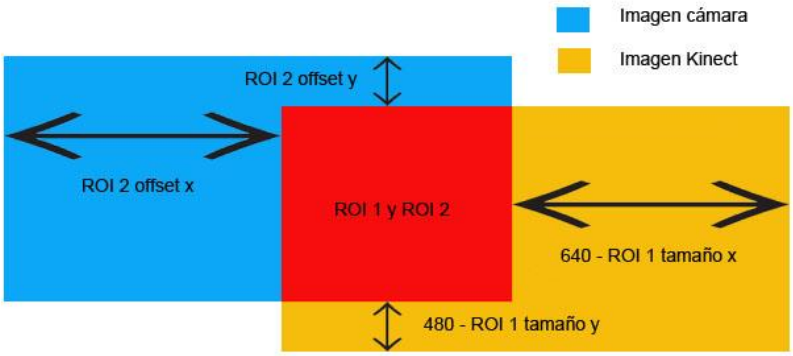

Figura 36: ROIs de superposición

Este desfase es implementado mediante la creación de un ROI de la imagen del emisor que será superpuesto sobre un ROI de la imagen de fondo. Como el primer ROI pertenece a la imagen del emisor, servirá para determinar qué parte de esa misma imagen está dentro de la escena y, por tanto, tiene que mezclarse. El segundo ROI determina la zona de la escena (cámaras) donde debe superponerse el primer ROI. Ambos ROIs deben ser iguales en tamaño (ver Figura 36).

Por supuesto, no existe una equivalencia definida entre los grados de movimiento y los pixeles de desfase. En los métodos encargados de determinar las regiones de interés se crea una variable factor de multiplicación que se encargará de realizar el ajuste (calibración). El valor de ajuste depende directamente de la distancia del sujeto emisor y la distancia de los objetos de referencia del fondo. El valor se determina mediante pruebas para una distancia al sujeto alrededor de tres metros y un fondo a 6 metros o más. Cuanto más lejos se encuentre el fondo menos sensible será el ojo humano a percibir errores en este desfase. Para la altitud ocurre lo mismo, sólo que el desfase será vertical.

El tercer eje en el casco equivale al movimiento que hacemos con la cabeza llevando la oreja hasta el hombro, más conocido como “roll”. En ese caso, aunque la escena gire, el sujeto emisor debe permanecer siempre vertical. En este caso, una función se encarga de corregir el ángulo de la imagen antes de la superposición.

La superposición de la imagen se realiza mediante una suma de las matrices de los ROIs implicados. El color negro de la imagen del emisor (fondo), al sumarse con la imagen de las cámaras equivale a dejar la segunda imagen sin variaciones, ya que su valor de color es 0. Sin embargo, es necesario crear una máscara con la forma de la imagen de la persona para poder eliminarla de la imagen de las cámaras antes de la suma. De lo contrario, se sumarían los colores de imagen de la persona con los colores de la escena de fondo de las cámaras, lo cual produciría un efecto no deseado (equivalente a una imagen con mucho brillo y semitransparente, ya que la suma de colores acerca los valores al color blanco).

Tan pronto como la imagen de uno de los lados haya sido procesada, el objeto Bitmap resultante se mostrará en la interfaz. En Android sólo el hilo de ejecución principal puede modificar la interfaz gráfica por lo que el lado izquierdo y derecho nunca se actualizarán exactamente al mismo tiempo. Esto no debería ser un problema si se consigue una tasa de refresco suficiente.

5.3.5 Hilos de ejecución

El procesamiento de imagen es una tarea exigente en tiempo de CPU, pero en este caso, al tratarse de imágenes que proceden de distintas fuentes, es posible paralelizar muchas de las tareas. Maximizar el paralelismo de las tareas es muy importante cuando se usa un smartphone como plataforma de ejecución, ya que gran parte de su potencial de procesamiento proviene del aumento en número de núcleos de la CPU. El Galaxy S3 usado en las pruebas dispone de 4 cores, pero no es raro ver dispositivos con 8 cores.

La aplicación realiza varias tareas que requieren mucho tiempo de CPU o depende de la velocidad de terceros, como la red, o la memoria.

- Transferencia de la imagen por Wi-Fi

- Calculo de la orientación Sensor Fusion
- Obtención de la imagen de cámara 1
- Obtención de la imagen de cámara 2
- Descompresión de imagen JPEG recibida desde el servidor
- Creación de la máscara de forma de persona emisora para cada una de las imágenes de cámara
- Procesamiento del giro de imagen debido a la inclinación “roll” para cada uno de los lados de la imagen estéreo.
- Suma de las matrices de imagen.

A continuación se detallan los hilos de ejecución implicados, para la modalidad monopuerto (ver Figura 37):

El hilo principal (THREAD 0) permanece descargado la mayor parte del tiempo, siendo utilizado solo para la actualización de las imágenes en la interfaz una vez han sido procesadas, de esta forma garantizamos una buena respuesta a los eventos de control provocados por el usuario.

La transferencia de imagen es una de las tareas que más retardo produce, dependiendo de la velocidad de la red. Esta tarea, junto con la descompresión JPEG y el cálculo de ROIs en función de la orientación, conformarán un hilo de proceso (THREAD 1).

Si en ese mismo hilo (THREAD 1) se detecta que la imagen del emisor está dentro del campo de visión actual, serán iniciados 2 hilos nuevos de procesamiento de imagen (THREAD 2 y 3). Si no es así, esto dos hilos solo realizarán la petición de la imagen de cámara al servicio WebcamManger.

Cuando estos hilos son lanzados, THREAD 1 inicia la recepción de la siguiente imagen a través de la red paralelamente al procesamiento de la actual. Si THREAD 1 se completa antes que THREAD 2 y 3, debe esperar a que éstos terminen, ya que podríamos perder el control sobre el orden de las imágenes.

Los THREAD 2 y 3 ejecutan a la vez pero son paralelos entre sí, por lo que pueden aprovechar la potencia de dos cores distintos si es necesario, dependiendo del gobernador de la CPU. Estos hilos realizan la parte más exigente del procesamiento de imagen; cada uno procesa la imagen de uno de los lados de la imagen estéreo, también realizan la petición al servicio WebcamManager sobre la imagen de cámara, la creación de la máscara, la rotación de la imagen en función del eje de orientación “roll” y la suma de las imágenes.

En el caso de no ser necesaria la mezcla de imágenes sólo realizan la petición al servicio WebcamManager. Está ejecución es rápida, ya que como se ha explicado anteriormente, WebcamManager ejecuta sus propios hilos para la optimización del rendimiento. Los servicios en Android funcionan en el hilo principal de la aplicación que los crea, por lo que no se considera un hilo aparte. Sin embargo, WebcamManager crea dos hilos periódicos que realizan el procesamiento de la imagen de cámara (mapeo de memoria, conversión YUV a ARGB8888, etc.) a un ritmo de 25fps. (THREADS 4 y 5).

Respecto al cálculo de la orientación, los eventos registrados por el listener ejecutan en el hilo principal (THREAD 0). El resto, aplicación de los filtros y suma de las señales, se realiza en un hilo de ejecución periódica a 25Hz (THREAD 8). Respecto a los primeros, es una buena práctica que cuando un evento desencadena una tarea larga ésta se realice en un hilo aparte. Cada evento de orientación iniciará un hilo para realizar los cálculos pertinentes, siendo el más exigente el del giroscopio, que ejecuta a una frecuencia de unos 80Hz y debe realizar la integración de velocidades angulares y varias multiplicaciones de matrices.

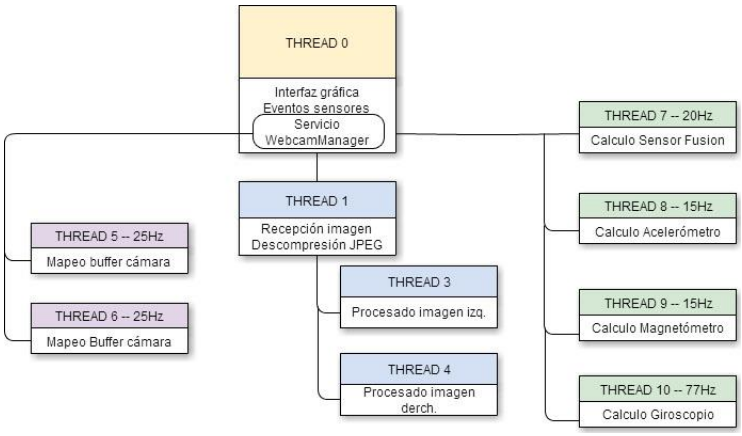

Figura 37: Diagrama de hilos de ejecución en la modalidad monopuerto

En el modo multipuerto el paralelismo es mayor. Se disponen de dos nuevos puertos para crear dos sockets de conexión distintos por lo que se transmiten la imagen izquierda y derecha por separado. En este caso THREAD 1 solo se encarga de responder al servidor en cada iteración con la petición de nueva imagen y el cálculo de ROIs como tareas importantes. Se crean dos nuevos hilos cada uno encargado de recibir una de las imágenes por su socket, además la descompresión JPEG también se paraleliza. Cuando ambos hilos acaben esa tarea se lanzan otros dos encargados del procesamiento de la imagen, como antes (ver Figura 38).

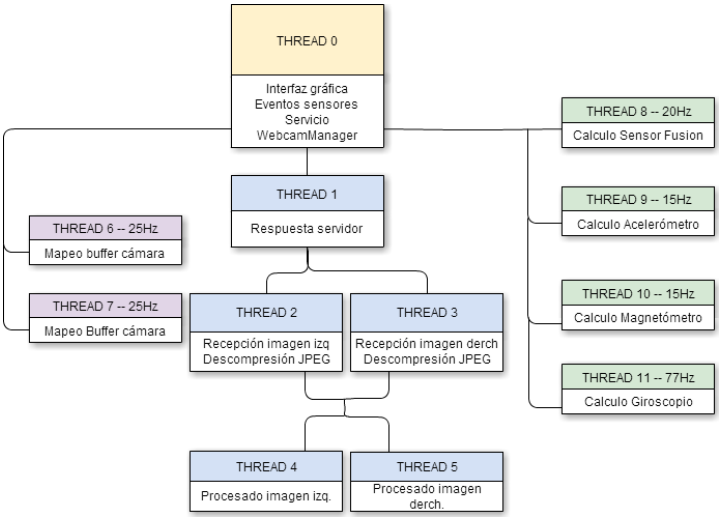

Figura 38: Diagrama de hilos de ejecución en la modalidad multipuerto

5.4 Programación

En esta sección se describen las clases y métodos principales desarrollados en la implementación de la aplicación cliente.

5.4.1 Estructura

En la siguiente imagen (Figura 39) se puede ver la estructura de clases de la aplicación. El desarrollo y las modificaciones llevadas a cabo en OpenXC.webcam forman una librería externa al paquete principal, de esa forma se favorece la modularidad.

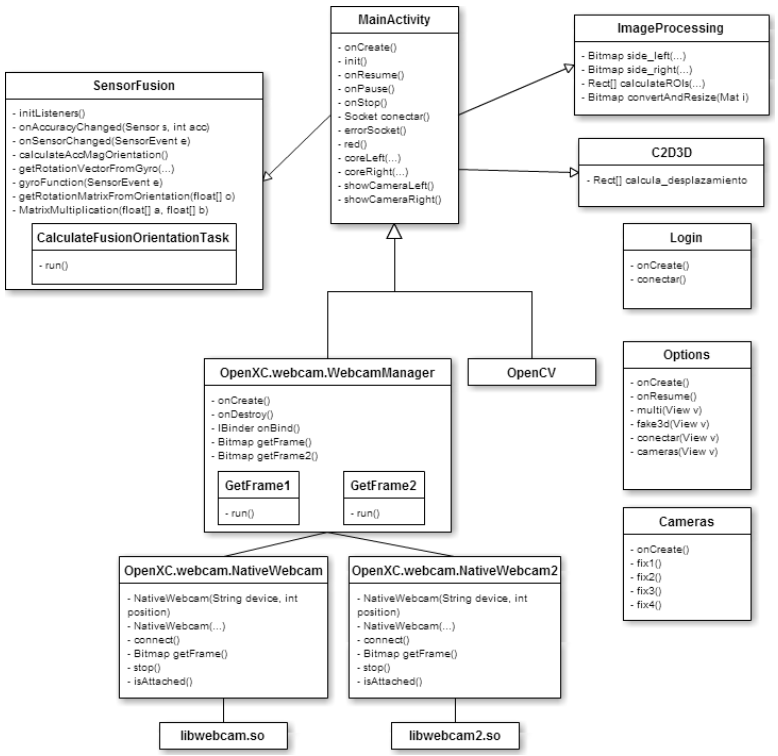

Figura 39: Diagrama de clases de la aplicación

5.4.2 Cámaras frontales

El servicio de cámaras frontales se compone de 3 clases java (WebcamManager, NativeWebcam y NativeWebcam2) además de las librerías en código nativo (libwebcam.so y libwebcam2.so).

Para compilar el código nativo de las librerías de acceso a las cámaras es necesario instalar Android NDK (Native-development-kit) en el equipo donde se desea compilar. NDK es un conjunto de herramientas que permiten compilar código nativo como C o C++ para

dispositivos Android. Al ejecutar el comando ndk-build en el directorio del proyecto se generan las librerías .so que serán cargadas dinámicamente mediante la instrucción System.loadLibrary(libname) en la inicialización del servicio. Cada objeto NativeWebcam cargará una de las librerías nativas y pasará como parámetro el objeto Bitmap donde desea mapear la imagen, así como el descriptor de la cámara y la resolución deseada.

El servicio estará funcionando mientras haya algún cliente conectado. Es importante desconectarse del servicio siempre antes de cerrar la aplicación, de lo contrario podría permanecer en ejecución.

La clase WebcamManager crea los objetos NativeWebCam y los hilos de ejecución periódicos que realizan una petición de imagen a cada cámara 25 veces por segundo. También contiene los métodos públicos del servicio que entregarán las imágenes a la actividad de proceso de imagen principal bajo demanda.

5.4.3 Resumen de métodos de la actividad MainActivity

En la siguiente tabla se resumen los métodos implementados en la actividad MainActivity (ver Tabla 4). Los parámetros de los métodos no son detallados pero si se comentarán los más importantes en la descripción. MainActivity implementa la lógica de control principal de la aplicación, la comunicación con el servidor y la visualización de la imagen estéreo.

onManagerConnected() de la clase interna BaseloaderCallback	Confirma si la librería *OpenCV* se ha cargado correctamente de forma asíncrona. En caso afirmativo llama al método conectar() y al método init() de MainActivity.
onCreate(…)	Carga la interfaz gráfica (layout) y configura la vista a pantalla completa. Inicializa las variables de imagen y la matriz de rotación del giroscopio. También inicia el hilo periódico de cálculo de sensor fusión.
onStart(…)	Inicia la conexión con el servicio WebcamManager.
onResume(…)	Recibe los parámetros IP y Puerto de la actividad anterior, así como la posición horizontal azimuth y otras variables de control.
onStop(…)	Detiene el bucle del hilo de transferencia de imagen y por consiguiente todos los demás. Desconecta el servicio WebcamManager y detiene los hilos de los sensores.
conectar(…)	Crea un socket de conexión con el servidor especificado e inicializa un objeto DataInputStream y otro DataOutputStream. Si multipuerto está activo crea dos sockets de conexión más con objetos DataInputStream.
errorSocket(…)	Esta función es llamada por la actividad cuando hay un error de conexión con el servidor. Intenta restablecer la conexión con el servidor reiniciado los métodos correspondientes.
red()	Inicia el hilo de ejecución principal. Recibe la imagen desde el servidor y responde a este cuando está listo

	para recibir la siguiente. Decide si la imagen ha de procesarse o si hay que mostrar directamente la imagen de cámara en función de la orientación. Dependiendo de esta decisión inicia los hilos de ejecución correspondientes. - Si multipuerto está activo inicia 2 hilos de transferencia independientes. - Si falso 3D está activo llama a una función de cálculo de desplazamiento diferente.
procesa_izquierda(…)	Realiza la mezcla de las imágenes de servidor y cámara en función de los ROIs creados a partir de los valores de orientación. Solo para el lado izquierdo. - Usa métodos estáticos de la clase ImageProcessing que se verán más tarde.
procesa_derecha(…)	Realiza la mezcla de las imágenes de servidor y cámara en función de los ROIs creados a partir de los valores de orientación. Sólo para el lado derecho. - Usa métodos de la clase ImageProcessing.
muestra_cámara_derecha(…)	Muestra la imagen de cámara actual por pantalla para el lado derecho. Realiza la petición de imagen al servicio WebcamManager.
muestra_camara_izquierda(…)	Muestra la imagen de cámara actual por pantalla para el lado izquierdo.
onServiceConnected(…) de la clase ServiceConnection	Este método es llamado por el sistema cuando la conexión con el servicio WebcamManager se ha efectuado correctamente. Modifica una variable de control para que el resto de métodos puedan saber que está enlazado y pueden empezar.
onServiceDisconnected(…) de la clase ServiceConnection	Este método es llamado cuando la conexión con el servicio WebcamManager se pierde debido a algún error.

Tabla 4: Resumen de métodos de MainActivity

5.4.4 Resumen de métodos de la clase ImageProcessing

ImageProcessing contiene los métodos de procesado de imagen (ver Tabla 5) como son los de mezcla de la imagen de cámara con la recibida desde el servidor, o el cálculo de ROIs implicados en el mismo proceso.

side_left(…)	Recibe la imagen de cámara, la imagen del emisor y los ROIs de las zonas implicadas. También recibe un booleano con el objetivo de conocer si debe efectuar el ajuste “roll” en la imagen y los valores de orientación necesarios. A partir de todas las variables recibidas realiza la mezcla de imágenes tal y como se detalla de forma más amplia en la sección 5.3.4.

	Devuelve un objeto Bitmap ARGB8888 con la imagen mezclada.
side_right(…)	Realiza el mismo proceso que side_left pero para la imagen derecha.
Calculate_ROIs(…)	Recibe los valores de desfase de la imagen del emisor respecto de la imagen de las cámaras y a partir de ellos calcula los ROIs de las zonas de mezcla para ambas imágenes. Devuelve dos ROIs, el ROI de la imagen de cámara donde la imagen del emisor será superpuesta y el ROI de la imagen del emisor que señala que zona de esta será superpuesta sobre la primera.
convertAndResize(…)	Si se utilizan las cámaras bajo el estándar USB 1.X solo será posible utilizarlas hasta una resolución de 640x360 por lo que está función realiza el cambio de tamaño a 640x480. Utilizando USB 2.0 esta función solo se encarga de eliminar el canal alpha de las imágenes cuando sea necesario.

Tabla 5: Métodos de la case ImageProcessing

5.4.5 Resumen métodos C2D3D

C2D3D es una clase dedicada exclusivamente al modo de funcionamiento “falso 3d”. Este modo requiere un proceso de cálculo y mezcla de las imágenes muy distinto al del modo principal. Por eso, el método de cálculo de los ROIs implicados en la mezcla debe crear, también, un ROI alternativo para la imagen creada artificialmente (ver Figura 40) y que formará parte de la imagen estéreo. A continuación se detalla dicho método de cálculo (ver Tabla 6)

calcula_desplazamiento(…)	Este método sólo es utilizado en el modo falso 3D. Recibe como parámetro los valores de desplazamiento en función de la orientación de igual forma que la función calculate_ROIs de la clase ImageProcessing, pero además recibe el valor de profundidad media de la persona capturada por *Kinect* y un entero que indica el factor de paralaje. Esta función devuelve cuatro ROIs en lugar de dos, de forma que durante la creación de la segunda imagen estéreo los dos nuevos ROIs nos ayuden a crear un efecto espejo respecto de la primera. La separación de las imágenes de la persona respecto del centro de la imagen estéreo se calculará en función del valor de profundidad y se

	podrá ajustar mediante el factor de paralaje que el usuario debe introducir en las opciones de la sesión.

Tabla 6: Método de cálculo de ROIs de C2D3D

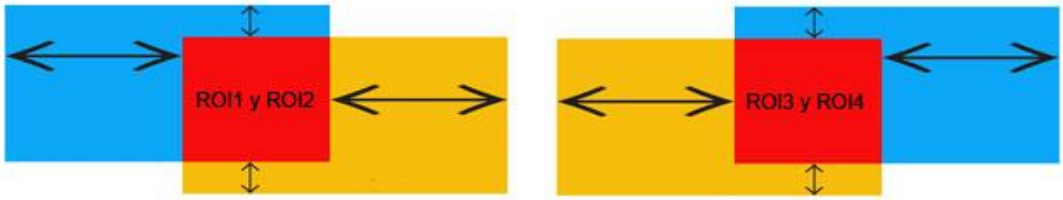

Figura 40: ROIs del modo falso 3D

5.4.6 Resumen de métodos la clase SensorFusion

A continuación se describe brevemente los métodos de la clase SensorFusion dedicada al cálculo de la orientación del dispositivo (ver Tabla 7).

initListeners()	Registra los listeners de los sensores. Se establecen las frecuencias de muestreo.
onAccuracyChanged(…)	En este caso no hace nada pero es necesario implementarlo
onSensorChanged(…)	Recibe los resultados de los sensores (evento). Inicia los hilos de proceso correspondientes en función del tipo de sensor.
calculateAccMagOrientation()	Genera la matriz de rotación a partir de los resultados del magnetómetro y del acelerómetro. Genera los valores de orientación en 3 ejes a partir de la matriz de rotación
getRotationVectorFromGyro(...)	Realiza la integración de las velocidades angulares del giroscopio
gyroFunction(…)	Acumula los movimientos del giroscopio en forma de matriz de rotación
getRotationMatrixFromOrientation(…)	Convierte los valores de orientación de los ejes en una matriz de rotación
MatrixMultiplication(…)	Multiplica dos matrices
calculateFusedOrientation(…)	Realiza la suma de señales de orientación acelerómetro/magnetómetro y giroscopio con el coeficiente de filtraje especificado.

Tabla 7: Métodos de la clase SensorFusion

5.4.7 Proceso de mezcla

Mediante los datos de orientación la función calculate_ROIs genera para cada lado de la imagen estéreo una región de interés en la imagen de la cámara frontal y otra región distinta en la imagen del emisor procedente del servidor.

La primera misión es conseguir la imagen de cámara. Se realiza la petición del Bitmap al servicio WebcamManager y se convierte a objeto Mat (Figura 41).

```
//obtiene la imagen de cámara
Mat cRight=new Mat();
Utils.bitmapToMat(mWebcamManager.getFrame2(), cRight);
```

Figura 41: Obtención de la imagen de cámara

La instrucción bitmapToMat convierte la imagen procedente de WebcamManager en objeto de tipo Mat y lo almacena en cRight.

Las imágenes procedentes de la cámara son de tipo RGBA8888, es decir, tienen canal de transparencia o canal alfa, mientras que las imágenes de servidor no tienen. Para poder superponer las imágenes, ambas deben ser del mismo tipo, así que el siguiente paso es eliminar el canal alfa.

```
Imgproc.cvtColor(cImage, cImage, Imgproc.COLOR_RGBA2RGB);
```

Ahora es el momento de rotar la imagen del servidor si la función roll está activada. La variable roll almacena el grado de inclinación de la cabeza en grados. Si la cabeza está nivelada (línea formada por los dos ojos paralela al suelo) roll será 0. Si inclinamos la cabeza en sentido anti horario roll será positivo y al contrario, negativo. La instrucción warpAffine se encarga de transformar la matriz en función del valor de giro (ver Figura 42), es una tarea exigente en tiempo de CPU.

```
//Roll
int roll = (int)(fusedOrientation[1]*180/Math.PI); //negative is clockwise
if (rollEnabled && roll != 0){
    Point center = new Point(320,240);
    double angle = -roll;
    double scale = 1.0;

    Mat mapMatrix = Imgproc.getRotationMatrix2D(center, angle, scale);
    Imgproc.warpAffine(iRight, iRight, mapMatrix, iRight.size(), Imgproc.INTER_LINEAR);
}
```

Figura 42: Código de implementación del giro "roll"

Las ROIs de orientación son secciones cuadradas que nos permiten especificar qué partes de la imagen van a ser afectadas por las funciones, pero siempre son regiones cuadradas o rectangulares. No es posible superponer una región cuadrada de la imagen del servidor sobre la imagen de cámara porque, pese a que el fondo es negro y los colores

de fondo de la imagen de la cámara no se verían afectados por el color negro (valor 0) en una operación de suma, tenemos que eliminar la forma irregular que ocupará la imagen de la persona emisora en la imagen de cámara. De lo contrario, los colores de la imagen de cámara que ocupan la zona donde será superpuesta la forma irregular de la persona y los propios colores de la persona serían sumados.

Para tratar con formas irregulares es necesario crear una máscara. En este caso es sencillo puesto que *Kinect* ya se encargó, en el proceso de emisión, de separar la forma de la persona. Una máscara es un matriz en la cual las posiciones de interés contienen un valor 1 mientras que el resto contienen 0.

En este caso la máscara se crea por discriminación de colores. Es posible crear una máscara escaneando los valores de la imagen de forma que si superan un cierto umbral tendrá un valor 0 en la máscara y si no, tendrán un valor 1. La forma de la persona puede contener muchos colores, dependiendo de la ropa que lleve el sujeto emisor, por ejemplo, pero el fondo siempre es negro. En RGB separar un color de otro es complicado, ya que colores muy próximos pueden ser muy distantes en su valor RGB y viceversa. Para simplificar el proceso primero se convierte la imagen a escala de grises (ver Figura 43).

```
//  Crea la mascara para la forma humana y su inversa
Mat maskRight = new Mat();
Mat tmpr = new Mat();
//  Selecciona ROI para superposición
Mat iRightROI = new Mat(iRight, roiSrc2);

// Convierte la imagen a escala de grises para discriminar mejor el color negro
Imgproc.cvtColor(iRightROI,tmpr,Imgproc.COLOR_RGB2GRAY);

//umbral para el color negro (background)
Imgproc.threshold(tmpr, maskRight, 1, 255, Imgproc.THRESH_BINARY);
tmpr.release();
```

Figura 43: Creación de la máscara de fondo

De esta manera tenemos la máscara de fondo. Ahora podemos invertir la matriz para seleccionar la forma de la persona (ver Figura 44).

```
//Se invierte la matriz para crear la mascara de la persona
Mat inv_mask_r = new Mat();
Core.bitwise_not(maskRight, inv_mask_r);
maskRight.release();
```

Figura 44: Inversión de la máscara para la selección de la forma de la persona

Para eliminar la forma que ocupará la persona de la imagen de la cámara se efectúa una operación AND consigo misma, pero usando la máscara para seleccionar sólo la forma de la persona como afectada por la operación (Figura 45).

```
Mat cRightSVGA_roi = new Mat(cRightSVGA,roiSrc1);
Mat r_black = new Mat();
Core.bitwise_and(cRightSVGA_roi, cRightSVGA_roi, r_black, inv_mask_r);
inv_mask_r.release();

//Se copia la roi tratada a su zona de origen
r_black.copyTo(cRightSVGA_roi);
```

Figura 45: Eliminación de la forma de la persona en la imagen de cámaras

El objeto Mat r_black contiene la ROI cuadrada con la forma de la persona eliminada. Para realizar estos cambios en la imagen global de cámara se copia a la zona que ocupaba en origen (Figura 45). El último paso consiste en sumar ambas ROIs (ver Figura 46).

```
// Suma de imagenes
Core.add(cRightSVGA_roi, iRightROI, cRightSVGA_roi);
iRight.release();
iRightROI.release();
cRightSVGA_roi.release();
```

Figura 46: Suma de las matrices de imagen

cRightSVGA_roi es la ROI cuadrada de la imagen de cámara donde la ROI cuadrada de la imagen del servidor (iRightROI) tiene que superponerse. iRightROI contiene la forma de la persona y un fondo negro mientras que cRightSVGA_roi contiene el fondo de color y la forma de la persona en color negro. El resultado se almacena en la misma cRightSVGA_roi para que los cambios afecten a la imagen de cámara completa, esa será la imagen resultado. Ahora es posible convertir el objeto a Android Bitmap y mostrarlo en la pantalla (Ver Figura 47).

```
Utils.matToBitmap(cRightSVGA, bmr);
cRightSVGA.release();

finishedRight=true;

runOnUiThread(new Runnable() {

    @Override
    public void run() {
        ir.setImageBitmap(bmr);  //mostramos la imagen
    }
});
```

Figura 47: Visualización de la imagen procesada en el ImageView correspondiente a su lado

5.5 Interfaz Gráfica

La aplicación constará de 4 actividades. Una primera actividad de conexión, una de configuración de la sesión, otra de soporte a la configuración del sistema y la cuarta será la actividad principal o de visión estéreo.

Figura 48: icono de la aplicación

En la primera actividad se mostrará un logo de la aplicación a modo de presentación y dos campos de texto para introducir la IP del servidor de emisión y el puerto principal. Si se opta por el modo de un solo puerto, este será el que se utilice para responder al servidor y recibir la imagen estéreo (ver Figura 49).

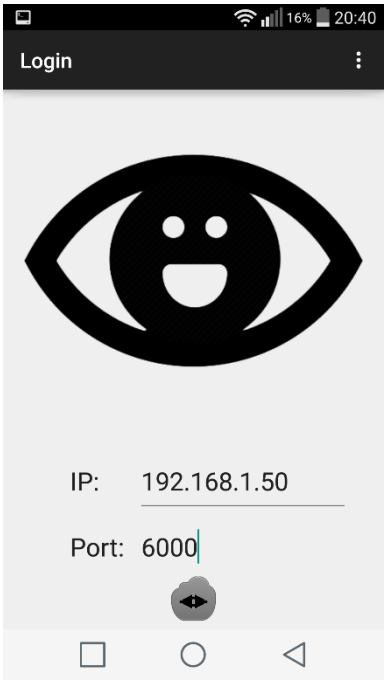

Figura 49: Actividad principal de la aplicación

En la segunda actividad se configuran aspectos de la sesión como la posición del emisor en el entorno (valor azimuth) en grados. Por ejemplo, un valor de 90º colocaría la imagen de la persona recibida al norte (ver Figura 50).

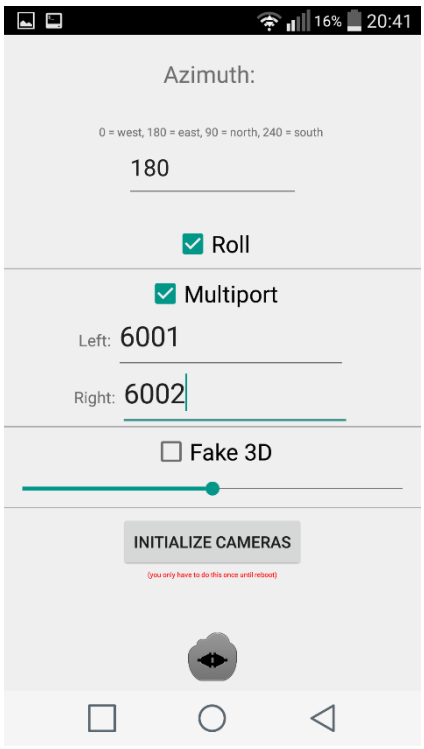

Figura 50: Actividad de configuración de la sesión

Un checkbox permite la activación o desactivación del eje “roll” en el procesamiento de imagen, ya que la tarea de giro efectuada utilizando la instrucción warpAffine de *OpenCV* es muy costosa en tiempo de CPU. También se encuentran las opciones de activación y configuración del modo multipuerto, que obligará a introducir dos puertos más si se activa, y el falso 3D, que incluye una barra para configurar el factor de paralaje. El botón “initialize

cameras” abre una nueva actividad que ayuda a configurar el sistema para la correcta utilización de las cámaras.

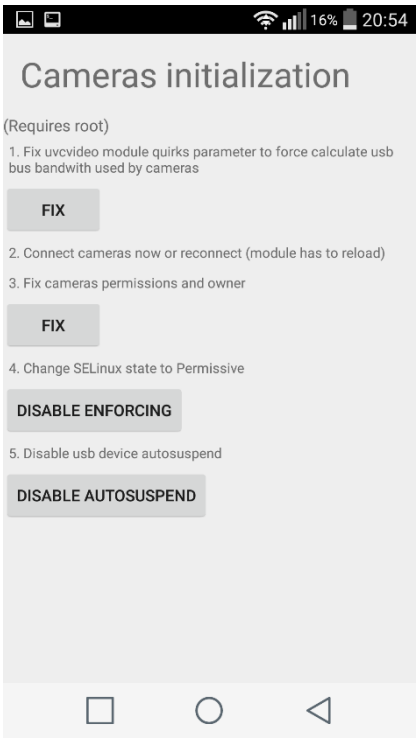

Figura 51: Actividad de ayuda a la configuración del sistema

El primer botón corrige el valor del parámetro *quirks* en el módulo *UVCvideo* para que, como se ha explicado anteriormente, sea posible utilizar ambas cámaras al mismo tiempo. El segundo botón corrige los permisos y el propietario de los descriptores de las cámaras en el caso de que sean incorrectos. Los dos botones restantes sirven para cambiar el estado de SELinux a permisivo y deshabilitar la auto-suspensión de dispositivos USB respectivamente. Estas instrucciones requieren de autorización superusuario, por lo que la autorización será solicitada al pulsar cualquiera de los botones.

La tercera actividad se compone de 2 objetos ImageView de Android para mostrar las imágenes procesadas, cada uno ocupando la mitad de la pantalla. Se han ocultado todos los elementos como la barra de notificaciones o barra de actividades para mostrar las imágenes a pantalla completa (ver Figura 43). Como el uso principal de esta actividad es dentro del casco, su orientación estará forzada en horizontal (landscape).

Figura 52: Actividad de visualización estéreo

5.6 Pruebas

Las pruebas de la aplicación se han realizado sobre un dispositivo Samsung Galaxy S3. Se trata de un dispositivo que en su día formaba parte de la gama alta de Samsung pero que fue lanzado a la venta en mayo de 2012 y está lejos de ser un dispositivo puntero en potencia hoy día. Se han realizado pruebas de estabilidad y velocidad de la aplicación en este dispositivo.

Lo primero que se observa es que la imagen no es del todo fluida sino que sufre cierta inestabilidad produciendo algunos saltos de imagen en determinados momentos. La tasa de refresco de las imágenes está lejos de los 24fps ideales pero por encima del umbral de usabilidad. Es de esperar que en un dispositivo de última generación si sea posible acercarse a esas cifras.

La estabilidad es buena si bien hay que tener en cuenta que parte del proceso implica una transferencia de imagen de 1280x640 pixeles sin buffering, con lo cual pueden producirse algunos errores de los que la aplicación debería ser capaz de recuperarse mediante una reconexión. A menudo estos fallos están causados por una pérdida de sincronización entre el servidor y la aplicación que provoca que la trama en la que se envía el tamaño de la imagen siguiente se pierda.

Respecto a la orientación, la velocidad de muestreo es suficiente para superponer las imágenes con precisión sin comprometer el rendimiento. No se ha experimentado ningún problema al respecto, sin embargo, las cámaras externas si suponen un problema en cuanto al gasto de energía del dispositivo. El hub USB utilizado en las pruebas dispone de una entrada para alimentación externa, pero no es nada práctico usarla, sobre todo cuando una de las ventajas de usar está modalidad de casco HMD es que no necesite cables. Aun así es capaz de aguantar más de una hora de uso con la batería cargada al 100%.

Sobre la calidad de la imagen resultado se discutirá más en las conclusiones finales.

Capítulo 6: Requisitos de uso

6.1 Servidor de Emisión

Los requisitos del servidor de Emisión son:

- Sistema operativo Windows XP o superior
- Dos dispositivos *Kinect* conectados por USB
- Conexión de red
- Dual core a 2.4Ghz o superior, no es exigente en RAM.
- No es necesario tener instalado *Kinect* SDK.

6.2 Cliente de recepción

Los requisitos de la aplicación cliente son:

- Smartphone Android Quad core 1.4Ghz o superior
- Pantalla de 4,7” – 5,5” Resolución 640x1280 o superior
- Android 2.3 o superior
- El sistema debe disponer de permisos de superusuario (rooted) para cambiar los permisos de cámara
- Algunos dispositivos no proporcionan energía suficiente en el bus USB para iniciar dos cámaras

Capítulo 7: Conclusiones

La realidad virtual jugará un papel muy importante en el futuro próximo. Es muy complicado explicar con palabras la inmersión visual que esta tecnología puede lograr, y como puede ayudar al usuario a experimentar sensaciones de realismo en mundos virtuales que nada tienen que ver con la experiencia de 3D ficticio de una pantalla.

No es una tecnología nueva, pero es partir ahora cuando disponemos de los medios para comenzar a llegar a la gran mayoría de las personas, no solo con dispositivos de alto coste, y de alcanzar niveles de experiencia realmente espectaculares o que llamen la atención de todo el mundo.

El primer peldaño en la escalada de éxito de esta tecnología lo constituirá el lograr que se consolide como sistema de visualización de juegos preferido. Las posibilidades que ofrece en este campo son muy superiores al uso de cualquier otra tecnología conocida hasta ahora. Por ejemplo, el poder modificar el ángulo de la cámara moviendo la cabeza en un simulador de vuelo cambia totalmente las posibilidades de control y realismo. Pero es, sobretodo, la sensación de profundad que ofrece la visión estéreo lo que supone la verdadera diferencia en la experiencia, y que, como hemos dicho anteriormente, no se puede explicar con palabras.

Pero la realidad virtual abre un abanico de posibilidades enorme. Los planes de incorporar realidad virtual en las comunicaciones se hacen evidentes con la compra de Oculus Rift por parte de Facebook. Pero desconocemos totalmente que camino o aproximación seguirán para conseguirlo.

Sin embargo, en mi opinión, cualquier tecnología de realidad virtual conocida hasta ahora tiene un gran factor limitante fuera del campo los juegos, o incluso en ellos. Esto es el aislamiento total con la realidad: el no poder moverte realmente en el mundo real, verte las manos, o tener a la vista tu dispositivo de control, como el teclado por ejemplo. Recientemente, *Hololens* ha puesto fin a algunas de estas limitaciones; sin embargo, poco sabemos de cómo funciona en un escenario real, que precio tendrá, o cuando aparecerá realmente.

En este trabajo se ha llevado a cabo otra aproximación al problema muy distinta. El uso de dos cámaras conectadas a un smartphone como solución a la perdida de la relación visual con el entorno, es una idea de bajo coste y realizada con tecnología existente, que permite ampliar las posibilidades de la realidad virtual al mezclar objetos o formas reales con otras ficticias.

La aplicación de esta solución a una comunicación de persona a persona tampoco se ha visto hasta el momento y aunque el realismo alcanzado en este proyecto está lejos de ser perfecto, se espera que pueda servir como experimento base para futuras evoluciones.

Los principales problemas se deben a limitaciones técnicas, como por ejemplo, la baja resolución de entrega de la imagen de *Kinect* o que la red o la potencia de los dispositivos pueden soportar. Una vez que el usuario se ha puesto el casco HMD el tamaño de la pantalla se ve aumentado drásticamente, esto hace que la resolución se convierta en un aspecto crítico. Con la resolución actual (1280x480) y un tamaño de pantalla equivalente a unas 50”, los pixeles se aprecian en la imagen y perjudican en gran medida la sensación de realismo.

Otro de los problemas más graves es la capacidad de *Kinect* para reconocer los bordes de los objetos. La precisión es pobre y eso se traduce en artefactos no deseados y dientes de sierra que no transmiten realismo en el corte de la imagen de la persona superpuesta sobre el entorno.

A pesar de los problemas, se ha demostrado que la idea funciona, la sensación de profundidad en el entorno del receptor sigue existiendo aun llevando un dispositivo entre sus ojos y la realidad. Además, la superposición de la persona en la escena también conserva la sensación de volumen que una imagen natural brinda a nuestros ojos, y es sobre todo apreciable cuando, por ejemplo, la persona que se encuentra emitiendo extiende el brazo hacía la cámara, o mejor dicho, hacía nosotros.

Capítulo 8: Posibles mejoras y otras tecnologías

En este capítulo se van a comentar algunas ideas y otras tecnologías que pueden ayudar a evolucionar y mejorar el realismo del proyecto.

Una de las mejoras más evidentes en emisión es la utilización de *Kinect* 2. *Kinect* 2, además de mejorar la precisión en el reconocimiento es capaz de entregar una imagen RGB con resolución de 1920x1080 pixeles. No se puede olvidar, sin embargo, que estaríamos tratando entonces con el problema de procesar y enviar una imagen estéreo de 3840x1080 pixeles. En cualquier caso, también sería posible utilizar resoluciones intermedias para adecuarlo a las capacidades del hardware existente.

Otro problema evidente en fase de emisión es que estamos captando la imagen de la persona desde una sola perspectiva o punto de vista. Aunque el receptor se desplace hacia los lados, siempre tendrá al emisor mirando en la misma dirección. Una solución a este problema podría ser la utilización de una cadena de dispositivos *Kinect* y realizar una interpolación espacial entre imágenes. Este método es utilizado a menudo en televisión multi-vista o de punto de vista libre (ver Figura 53).

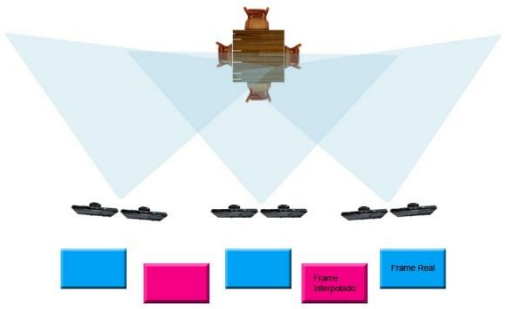

Figura 53: Posición de las cámaras en la técnica multi-vista

Esta técnica de interpolación también es utilizada en el proyecto de Google “JUMP” y su cámara (en colaboración con GoPro) con tecnología de captación de 360º (ver Figura 54).

Figura 54: Interpolación de imágenes en Google Jump

Tampoco podemos girar alrededor de la persona, pero esto es algo que se escapa fuera de la idea de este proyecto ya que implicaría el uso de tecnología no disponible en

este momento. Una tecnología que incorporase dicha funcionalidad requeriría de la creación de un modelo 3D y su texturizado en tiempo real. Con varios *Kinects* situados alrededor de la persona podríamos realizar el modelo y con la cámara RGB su texturizado, pero conlleva un retardo muy elevado para un modelo de comunicación en directo.

No se han realizado experimentos utilizando Wi-Fi 802.11ac pero es posible que produzca alguna mejora en la velocidad de envío de las imágenes.

En recepción, muchas de las posibles mejoras coinciden con las de la fase de emisión. Por ejemplo, la resolución de pantalla y de las cámaras son aspectos clave, sin embargo, en este caso solucionar el problema es más complejo. Encontrar cámaras con resoluciones Full HD es sencillo, al igual que un smartphone que equipe una pantalla de la misma resolución. El problema en este caso reside en la limitación del ancho de banda del bus USB. Hay muy pocos dispositivos que incorporen dos buses USB distintos, y en el caso de que lo hagan, suelen tratarse de tablets. En el caso de utilizar un puerto USB 3.0 son las cámaras o los hubs OTG los que no suelen cumplir el estándar. USB-C puede abrir nuevas posibilidades en este aspecto.

Otro problema en la integración de la persona con el entorno surge con los objetos y sus diferentes situaciones. Si un objeto está más cerca que la situación de la persona virtual en el entorno, ese objeto debería recortarse y no superponer la imagen de la persona sobre el mismo. Sin esta funcionalidad, el producto está condenado a usarse en entornos vacíos o con pocos objetos. Cualquier solución a este problema lleva implícita la necesidad de analizar las diferentes profundidades de la escena. Necesitaríamos un dispositivo similar a *Kinect*, también en recepción, para llevarlo a cabo. Sin embargo, *Kinect* es grande, pesado y lleva cables. Existen nuevas tecnologías que realizan análisis de profundidad de la escena y son relativamente portables. Una de estas tecnologías es Hololens, de la que sabemos muy poco sobre la base de su funcionamiento, y la que más se parece a nuestro dispositivo de pruebas sería Project Tango de Google.

La base hardware de Project Tango es una Tablet de 7” con una cámara RGB trasera, un sensor de profundidad y un sensor de movimiento. No es capaz de capturar imágenes estéreo pero es capaz de calcular los valores de profundidad de una escena. El sensor de movimiento integrado ayuda a situar el dispositivo en el entorno. Con el giroscopio y sensores tradicionales podemos conocer la orientación, pero Project Tango va un paso más allá facilitando la tarea de situar al sujeto en el espacio, por ejemplo, para conocer cuando nos acercamos o nos alejamos de algo.

Apéndice A: Construcción del casco HMD

Existen muchas alternativas de gafas VR en el mercado, algunas muy baratas, como Google Cardboard y las que siguen su modelo. Desgraciadamente ninguna contempla la posibilidad de añadir dos cámaras estéreo.

Para este proyecto se ha creado desde cero unas gafas artesanales completas para un Galaxy S3 y una tapa trasera universal para usar con Cardboard u otro tipo de gafa.

El material de la carcasa está formado por placas de espuma forradas de cartulina negra para mejorar la iluminación. La estructura tiene la forma semicircular de la cabeza por la parte de las lentes mientras que la parte trasera es plana y es donde se montan ambas cámaras.

En su interior y a una distancia de 3cm aproximadamente de la carcasa trasera se encuentras las lentes. He usado una lentes asféricas (lente con una forma similar a una porción de esfera, aunque no sea estrictamente esférica) de 5 aumentos para enfocar la pantalla y separadas entre sí a una distancia (entre centros) de 6.4 cm (ver Figura 55).

Figura 55: Vista de las gafas

He creado esta carcasa para usar un Galaxy S3 como dispositivo receptor, así que las medidas son las adecuadas para este dispositivo (ver Figura 56).

Figura 56: Gafas con Galaxy S3

Para no limitar la experiencia a un solo dispositivo las cámaras son montadas sobre una tapa aparte que irá adherida a las gafas que se deseen usar mediante velcro (ver Figura 57).

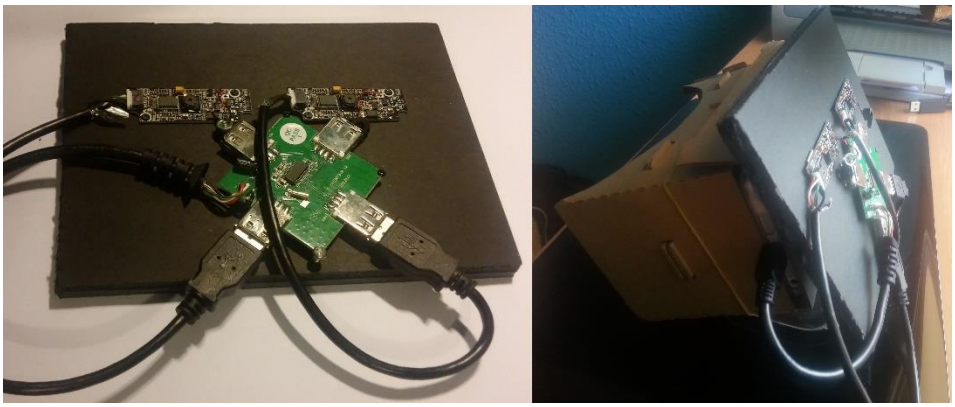

Figura 57: Tapa de cámaras y cámaras adheridas a Cardboard

Esta tapa usa el doble de grosor que el resto de la estructura de las gafas para añadir resistencia. Sobre ella van montadas ambas cámaras y el hub OTG.

Una cinta elástica para sujetar el casco a la cabeza completa el diseño.

Apéndice B: Imágenes del resultado