Actas de las XXXIX Jornadas de Automática, Badajoz, 5-7 de Septiembre de 2018
https://doi.org/10.17979/spudc.9788497497565.1061

NAVEGACIÓN VISUAL CONTROLADA DESDE UNAS GAFAS DE REALIDAD AUMENTADA

Christian García[1], Alejandro R. Mosteo[1,2], Ana C. Murillo[1]
[1]Instituto de Investigación en Ingeniería de Aragón (I3A), Universidad de Zaragoza
[2]Centro Universitario de la Defensa, Zaragoza (CUDZ)
{731754, amosteo, acm}@unizar.es

Resumen

Este trabajo presenta un prototipo de sistema de monitorización remoto que aúna una plataforma móvil con cámara remota y unas gafas de realidad virtual. El sistema implementa un bucle de percepción-acción gracias a los sensores de posición de las gafas VR, por un lado para ofrecer una imagen correctamente posicionada al usuario, y por el otro para controlar el dispositivo remoto (en este caso un pan-tilt*) hacia el punto de visión del usuario. El sistema se ha desarrollado mediante las librerías de software libre para realidad virtual (OSVR) y robótica (ROS), lo cual hace que esta prueba de concepto sea fácil de utilizar o replicar e integrar en otros proyectos.*

Palabras clave: Realidad aumentada, monitorización remota, robótica móvil, OSVR, ROS.

1. INTRODUCCIÓN

La realidad aumentada es una de las tecnologías que más impacto puede generar hoy en día. Proporciona una vista de un entorno físico del mundo real y además aumenta o complementa esta vista con información adicional generada por ordenador. Tanto la realidad virtual como la aumentada tienen un origen común. Ambas comenzaron con el lanzamiento entre 1957 y 1962 de *Sensorama* [7]. Sin embargo, la primera experiencia pura de realidad aumentada se creó entre 1969 y 1974, donde los usuarios interactuaban entre sí por medio de siluetas que se muestran en una pantalla. Desde entonces, se ha ido desarrollando la realidad aumentada, utilizándose principalmente para trabajos en grupos de investigación, hasta que en los años 2000 se da un salto para poder usar esta herramienta a diario dentro de juegos y aplicaciones móviles, como es el caso de ARQuake [9], la primera aplicación interior/exterior de realidad aumentada en primera persona.

Uno de los beneficios estudiados en el desarrollo de herramientas de realidad aumentada dentro de un entorno de servicios, es que pueden generar una mejora medible en los indicadores clave de rendimiento (*Key Performance Indicator*)[10] relacionados con la calidad, productividad y la eficiencia en el desarrollo de dichos servicios y actividades.

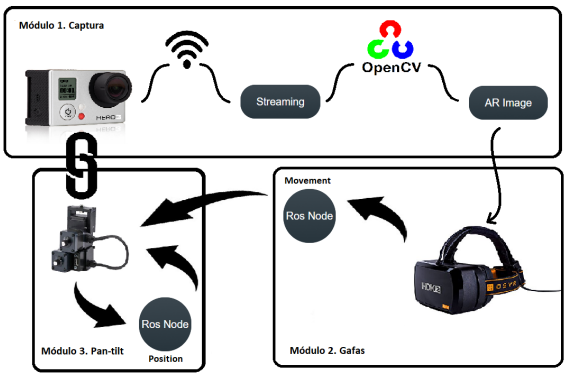


Figura 1: Arquitectura propuesta en este trabajo. Los tres módulos del prototipo se conectan vía ROS.

Por otro lado, el campo de la robótica también ha experimentado grandes avances en los últimos años, cada vez con sistemas móviles más autónomos. Esta gran evolución tanto de la robótica como de la realidad aumentada hacen atractiva la idea de aunar ambas herramientas. En particular, el objetivo general en este trabajo es juntar ambas tecnologías para mejorar los sistemas de monitorización inteligentes. Encontramos distintos trabajos juntando estas ideas, tanto en el ámbito comercial (*Robot Arena*, una plataforma de realidad aumentada para el desarrollo de videojuegos [2]), como en un ámbito de investigación (por ejemplo este modelo que busca caminos en zonas de bosque para encontrar personas perdidas [5]).

En este trabajo se presenta un prototipo de navegación visual remota controlado desde un sistema de gafas de realidad virtual/aumentada. Para ello, se propone un sistema que desde una cámara captura imágenes en directo de una escena real que queremos monitorizar, las procesa y aumenta con anotaciones automáticas, y las muestra en las gafas de realidad virtual. Además, el usuario puede controlar la navegación visual, es decir, a donde está apuntando la cámara móvil, gracias al control de un *pan-tilt* donde está montada la cámara. La Figura 1 presenta un diagrama del funcionamiento y la arquitectura del sistema diseñado e implementado.

El resto de este artículo describe la arquitectura propuesta en la Sección 2, la evaluación y detalles técnicos de integración del sistema real en la Sección 3 y discute los resultados y pasos futuros en la Sección 4.

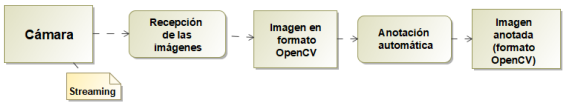


Figura 2: Flujo de acciones llevadas a cabo en el módulo de **captura y procesado de imagen** del sistema.

2. ARQUITECTURA DEL SISTEMA

Los principales módulos de este sistema, conectados y encapsulados como muestra el diagrama de la Figura 1, son los siguientes: 1) captura, envío y tratamiento de las imágenes; 2) visualización en las gafas de realidad virtual; 3) control del *pan-tilt* para poder navegar en primera persona por la escena monitorizada. Todos estos módulos se ejecutan en el ordenador central, que controla y conecta a todos los componentes.

2.1. CAPTURA Y PROCESADO DE IMAGEN

Esta sección resume el módulo que se encarga del paso de las imágenes desde la cámara móvil hasta el ordenador central, donde serán procesadas. Este primer módulo consiste en una serie de funciones encargadas de gestionar las imágenes recibidas y dejarlas preparadas para que el segundo componente de la aplicación se encargue de modificarlas para mostrarlas por las gafas de realidad aumentada utilizadas, como se resume en el diagrama de la Figura 2.

2.1.1. Captura de imágenes

En este prototipo se ha utilizado una cámara GoPro Hero3+, dispositivo ligero y manejable, con cámara RGB capaz de capturar a 60fps y WiFi. Se consideraron otras opciones, pero seleccionamos esta cámara porque nos da la posibilidad de enviar vídeo en tiempo real a través de la conexión WiFi que tiene disponible. Además, se puede integrar de forma sencilla en cualquier tipo de robot o *pan-tilt*.

La GoPro Hero3+ permite obtener vídeo en tiempo real a través de una conexión WiFi. Entre las opciones de configuración de la cámara utilizada (Tabla 1), se ha decidido utilizar el menor *fps*(*fotogramas por segundo*) posible, ya que si capturamos demasiados fotogramas por segundo, el retraso/latencia en mostrarlos anotados al usuario será demasiado alto.

Tabla 1: Posibilidades que ofrece GoPro Hero 3+ en la captura de vídeo en cuanto a frames por segundo (FPS) y resolución de la imagen (RES).

FPS	25	50	100
RES	720, 960, 1080	720, 960, 1080, WVGA	720, WCGA

2.1.2. Recepción y procesado

Una vez que la cámara empieza a transmitir vídeo, si conectamos el ordenador a la WiFi local de la cámara, podemos empezar a recibir las imágenes en una dirección IP. El módulo implementado para la recepción de imágenes inicia una captura de vídeo en la que se le indica dicha dirección IP donde están llegando las imágenes. La implementación de este módulo está basada en la biblioteca OpenCV, la cual tiene funcionalidades disponibles para este tipo de recepción/captura de vídeo:

```
cv::VideoCapture vcap;
cv::Mat image;
const std::string videoStreamAddress = \
  "http://10.5.5.9:8080/live/amba.m3u8";
if(!vcap.open(videoStreamAddress)) {
    cout << "Error ... " << endl;
}
if(!vcap.read(image)) {
    cout << "No frame" << endl;
    cv::waitKey();
}
//Procesar imagen
...
// Mostrar imagen
show_image(image);
```

Listing 1: Principales líneas del módulo para recibir el *streaming* de la cámara.

Una vez recibidos y procesados los datos enviados desde la cámara, se tienen las imágenes en una matriz en formato OpenCV sobre el que se puede añadir la información etiquetada. El campo de la realidad aumentada presenta muchas opciones para aumentar el contenido de una imagen. En este proyecto hemos optado por una opción sencilla como prueba de concepto, añadiendo información simple sobre las personas detectadas en la imagen, como se ve en la Figura 3. Para ello se utiliza un reconocedor de personas por defecto disponible en OpenCV, que permite parametrizar fácilmente los tamaños en los que detectará el objetivo. Utilizando este método, se consigue el objetivo de la aplicación de manera adecuada sin aumentar demasiado el coste computacional del sistema (el procesado de un fotograma cuesta alrededor de 0.75s). Este detector implementa el conocido algoritmo basado en HOG features para detección de personas [3]. La transformación que se ha de hacer para poder mostrar esta matriz anotada correctamente en las gafas de realidad virtual/aumentada se detalla a continuación.

Figura 3: Ejemplo de las anotaciones añadidas como ejemplo a las imágenes transmitidas desde la cámara.

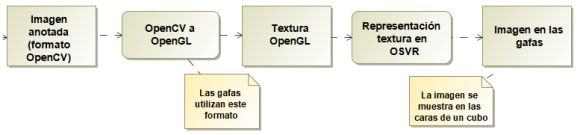


Figura 4: Flujo de acciones llevadas a cabo en el módulo de **visualización**.

2.2. VISUALIZACIÓN

Esta sección presenta las particularidades de la plataforma de realidad virtual utilizada en este proyecto, así como los módulos necesarios diseñados e implementados para poder mostrar los datos (resumidos en la Figura 4).

Open Source Virtual Reality (OSVR) [1] es una plataforma de software de código abierto para aplicaciones de Realidad Aumentada/Virtual. Esta plataforma abierta también incluye un dispositivo de gafas de realidad virtual (HDK2) [6] de bajo coste para facilitar pruebas de desarrolladores con acceso más abierto al hardware. En la Figura 5 se puede ver el dispositivo nombrado y utilizado en el trabajo. OSVR fue iniciado por expertos en juegos y realidad virtual y cuenta con el respaldo de una lista en constante crecimiento de proveedores de hardware, estudios de juegos, universidades y compañías de software. Es un sistema compatible con múltiples motores de juegos y sistemas operativos. El software OSVR se proporciona gratuitamente bajo la licencia Apache 2.0. y es mantenido por la compañia Sensics (`sensics.com`).

2.2.1. Preprocesado de datos

Las gafas de realidad virtual/aumentada utilizadas para este proyecto toman como entrada imágenes representadas como texturas de OpenGL. Este apartado detalla como se ha implementado la conversión de una imagen cargada como matriz de OpenCV a una textura de OpenGL. OpenGL dispone de funciones propias para generar una textura a partir de una imagen. Para ello, hay que establecer qué tipo de método de **interpolación** y qué tipo de **wrapping** se va a utilizar.

La **interpolación** se emplea en el caso de que, al modificar el tamaño de la imagen original, no se produzca un *matching* perfecto para cada píxel. Los métodos de interpolación de OpenGL considerados han sido:

Figura 5: Dispositivo HDK2 de OSVR utilizado en el proyecto

• GL_NEAREST: Devuelve el píxel mas cercano. Normalmente devuelve resultados algo pixelados en casos en los que la imagen original difiere en gran medida de la imagen final.

• GL_LINEAR: Devuelve la media ponderada de los 4 píxeles mas cercanos. Normalmente devuelve resultados algo mas borrosos que el método anterior.

• MIPMAP: es una copia mas pequeña de la textura que ha sido reducida y filtrada previamente, con alguna de las siguientes opciones: GL_NEAREST_MIPMAP_NEAREST (utiliza el mipmap que más se aproxima al píxel y muestrea con la interpolación del vecino mas cercano), GL_LINEAR_MIPMAP_NEAREST (Muestrea el mipmap más cercano con interpolación lineal), GL_NEAREST_MIPMAP_LINEAR (Muestrea los dos mipmaps mas cercanos interpolando al vecino mas cercano), y GL_LINEAR_MIPMAP_LINEAR (Muestrea los dos mipmaps mas cercanos con interpolación lineal).

El método GL_NEAREST es mas usado en casos de juegos en los que se busca imitar gráficos de 8 bits debido a su apariencia pixelada. Como no había una diferencia significativa en nuestro caso, se ha usado este método para evitar tener una imágenes borrosas.

El parámetro de ***wrapping*** se utiliza para elegir la opción de corrección para el caso de que alguna de las coordenadas se salga de los límites entre 0 y 1. Las posibles acciones consideradas son las siguientes:

• GL_REPEAT: La parte entera de la coordenada se ignora y se forma un patrón repetido.

• GL_MIRRORED_REPEAT: Se forma un patrón repetido, pero se produce un efecto espejo cuando la parte entera de la coordenada es impar.

• GL_CLAMP_TO_EDGE: La coordenada se ajusta entre 0 y 1.

• GL_CLAMP_TO_BORDER: Se asigna un color específico a los puntos que caen fuera de los límites.

En este caso se ha decidido utilizar el parámetro GL_CLAMP_TO_BORDER, ya que esto solo ocurre en las esquinas de la imagen debido a la transformación para su correcta visualización en las gafas, explicada a continuación.

2.2.2. Proyección en la pantalla

La textura a mostrar se proyecta en una pantalla virtual, que en este caso se ha decidido modelar como un cubo del que se usan cinco de sus caras. De este mo-

Tabla 2: Correspondencia de coordenadas de la imagen en 2D con las del cubo en 3D

Coordenadas de imagen ver Figura 6(a)	Coordenadas del cubo 3D ver Figura 6(b)
(0,0)	(-1,1,1)
(0,1)	(-1,-1,1)
(1,0)	(1,1,1)
(1,1)	(1,-1,1)
(offsetX,offsetY)	(-1,1,-1)
(1-offsetX,offsetY)	(1,1,-1)
(offsetX,1-offsetY)	(-1,-1,-1)
(1-offsetX,1-offsetY)	(1,-1,-1)

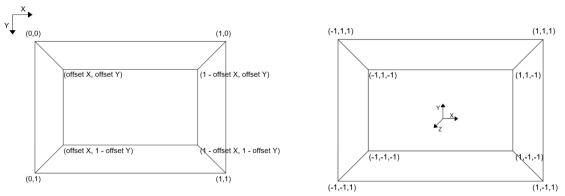
do la imagen plana no se deforma en su mayor parte y se consigue un efecto inmersivo para el usuario de manera sencilla y rápida para construir el prototipo. El usuario esta situado en el centro de este cubo, por lo que hay que transformar la imagen para poder visualizarla correctamente: dividirla en uno o varios trozos y decidir a que parte del cubo proyectar cada uno.

En el diagrama de la Figura 6(a) se ven las coordenadas empleadas para dividir la imagen en las caras deseadas. Estas coordenadas representan las zonas de la imagen que se proyectaran en cada una de las caras del cubo que rodea al usuario en la representación virtual de las gafas. La representación de dicho cubo y como se accede a cada cara (que coordenadas se utilizan para referenciar cada cara) se puede ver en el esquema de la Figura 6(b). Este esquema muestra las coordenadas que representan los 3 ejes principales del cubo, suponiendo que el origen del sistema de referencia está en el centro del cubo. Las coordenadas de las esquinas de las distintas caras en la imagen en 2D, se mapean a las correspondientes del cubo en 3D como muestra la Tabla 2. A continuación se puede ver la parte básica del código OpenGL que asigna las zonas correspondientes entre la imagen 2D (coord2f) y el cubo en 3D (vertex3f):

```
//Mapeo de la cara izquierda del cubo
glBegin(GL_POLYGON);
    glTexCoord2f(0.0,0.0);
    glVertex3f( 1.0, 1.0, 1.0);
    glTexCoord2f(offsetX,offsetY);
    glVertex3f( 1.0, 1.0, 1.0);
    glTexCoord2f(offsetX,1 offsetY);
    glVertex3f( 1.0, 1.0, 1.0);
    glTexCoord2f(0.0,1.0);
    glVertex3f( 1.0, 1.0, 1.0);
glEnd();
```

Listing 2: Ejemplo de mapeo de una de las caras del cubo con la parte de la imagen que le corresponde.

Validación de visualizaciones. En este punto se ha hecho una evaluación de las posibilidades que existen para transformar la imagen y que no se distorsione. La primera opción pasaba por simplemente mostrar la imagen completa en la cara frontal del cubo, pero los resultados no fueron los esperados ya que se perdía información en los laterales y la navegación en primera persona no se podía ejecutar como se deseaba.



(a) (b)

Figura 6: Diagrama del sistema de referencia empleado para asignar la imagen a cada una de las caras del cubo: (a) representación para dividir la imagen en varias partes; (b) representación de las caras del cubo en 3D que se visualiza en las gafas.

Figura 7: Esquema de la transformación descrita en la Figura 6 aplicado a una imagen

Tras realizar y probar los métodos con distintos usuarios, se llegó a la conclusión de mostrar la imagen en cinco caras del cubo, es decir, simulando que el usuario está situado en el centro del cubo, la cara que se sitúa detrás se ignora.

Para ajustar la distorsión de la imagen, la mayor parte de la misma se sitúa en la cara frontal del cubo, dejando así en las caras laterales un porcentaje pequeño de la imagen original. El *offset* (es decir, la distancia desde el borde de la imagen hacia el centro que se ha elegido para mostrar en los laterales del cubo) que se ha establecido para la captura de las partes laterales de la imagen es de 0.15. La Figura 7 muestra un ejemplo del resultado final proyectado en la pantalla de las gafas.

2.3. MOVIMIENTO REMOTO: PAN-TILT

Esta sección describe cómo se transmite la información de los sensores de movimiento las gafas al *pan-tilt*, para que este mueva la cámara de manera consecuente. Este proceso esta resumido en la Figura 8.

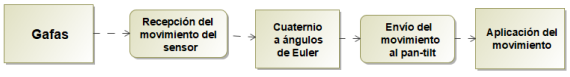


Figura 8: Flujo de acciones llevadas a cabo en el módulo de control del *pan-tilt*.

Para el prototipo de este proyecto, se ha utilizado un pan-tilt sencillo de adaptar al desarrollo, el *pan-tilt* E46 [8], mostrado en la Figura 9. Es un dispositivo pesado, por lo cual no sería práctico para una aplicación final, sin embargo, la ventaja de poder integrarlo fácilmente en el proyecto es más importante al tratarse de una prueba de concepto, sobre todo la fácil integración en el entorno Unix y ROS.

Figura 9: Pant-tilt E46-17 utilizado en el prototipo.

2.3.1. Captura de movimiento del usuario

Una vez obtenida la imagen final que se muestra en las gafas con la información adicional, el último paso para lograr el objetivo de navegar en primera persona es conseguir que la cámara de la cual vienen las imágenes se mueva a la vez que el usuario.

Para conseguir este último paso hay que obtener los valores de los sensores (IMU) de movimiento de las gafas de realidad aumentada. Estos sensores, en las gafas OSVR dan la información de rotación del dispositivo definida por un cuaternio. Sin embargo es necesario transformar la orientación obtenida a ángulos de Euler, ya que el *pan-tilt* utiliza este tipo de ángulos para su funcionamiento, realizando la transformación estándar reflejada en la Ecuación (1):

$$\begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} \arctan(\frac{2(q_0q_1+q_2q_3)}{1-2(q_1^2+q_2^2)}) \\ \arcsin(2(q_0q_2-q_3q_1)) \\ \arctan(\frac{2(q_0q_3+q_1q_2)}{1-2(q_2^2+q_3^2)}) \end{bmatrix}, \qquad (1)$$

donde q_0, q_1, q_2 y q_3 es la representación del cuaternio, se obtienen ϕ, θ y ψ como ángulos de Euler.

2.3.2. Nodo ROS de gestión del *pan-tilt*

Este último paso, gestiona el envio del movimiento del usuario, capturado en el paso anterior, al dispositivo en el que está instalada la cámara, así como la gestión de estos datos. Este paso se construye mediante la implementación de nodos de ROS (Robot Operating System) que gestionan la comunicación entre las gafas de realidad virtual con el *pan-tilt* escogido. Un nodo de ROS es un proceso que permite comunicar distintos procesos a través de *topics* y *services*. Cada uno de los procesos debe publicar y/o subscribirse al *topic* que necesita para obtener la información. La estructura de esta comunicación puede verse en la Figura 10.

El usuario puede mover la cabeza para indicar hacia donde quiere navegar por la imagen que se muestra

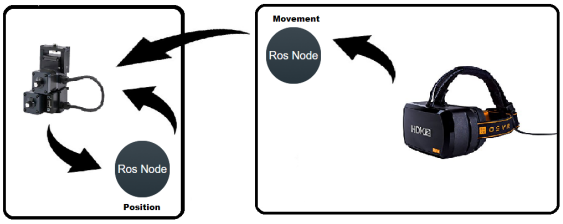


Figura 10: Estructura utilizada para comunicar el movimiento entre las gafas y el soporte de la cámara

en las gafas. Los sensores de las gafas, mediante los cálculos explicados en la Sección 2.3.1, publican el movimiento en el *topic* correspondiente, el *pan-tilt* recoge los valores y se encarga de reproducir el movimiento correspondiente para mover la cámara.

El *pan-tilt*, a su vez, para reproducir este movimiento, tiene otro hilo de comunicación. Para calcular el movimiento que tiene que aplicar, lee su orientación actual y se mueve hasta obtener la orientación final, que será la misma en la que se encuentre el usuario. Este módulo lee la posición actual en la que se encuentra el *pan-tilt* y genera el movimiento para llegar a la posición indicada desde las gafas:

```
void rotationCallback(const asr_flir_ptu_driver::State::
    ConstPtr& msg)
{
  ROS_INFO("Posicion actual: [%f][%f]", movement.state.
      position[0], movement.state.position[1]);
  asr_flir_ptu_driver::State movement_goal;
  movement_goal.state.position.push_back((movement.state.
      position[0] + msg->state.position[0])*180.0/3.141592);
  movement_goal.state.position.push_back((movement.state.
      position[1] + msg->state.position[1])*180.0/3.141592);
  movement_goal.state.velocity.push_back(1.0);
  movement_goal.state.velocity.push_back(1.0);
  ROS_INFO("Goal: [%f][%f]", movement_goal.state.position
      [0], movement_goal.state.position[1]);
  state_pub.publish(movement_goal);
```

Listing 3: Extracto del código para calcular la posición final deseada para el *pan-tilt* y su publicación en un *topic* de ROS que establece el objetivo (*movement_goal*) al que debe llegar el *pan-tilt*.

3. INTEGRACIÓN DEL SISTEMA

En esta sección se describen las componentes de validación más relevantes realizadas para cada módulo, así como las pruebas de integración realizadas.

3.1. MÓDULO 1: ANOTACIONES

En este primer módulo el objetivo era conseguir que las anotaciones automáticas añadidas a la imagen mientras se realiza el *streaming* de la cámara no causaran gran impacto en la latencia hasta las gafas. Por lo tanto, para optimizar al máximo esto dentro de las posibilidades de la cámara se han hecho diferentes pruebas de tiempos en el procesado de los fotogramas. La

Tabla 3 muestra la relación entre la calidad de los fotogramas y el tiempo empleado en procesarlos añadiendo las anotaciones.

Tabla 3: Correspondencia entre resolución de imagen (RES) con tiempo medio de procesado (T_{Proc}) para añadir las anotaciones.

RES	T_{Proc}
800 x 480 (WVGA)	0.82s
1280 x 720	1.95s
1280 x 960	2.65s
1920 x 1080	4.43s

3.1.1. Configuraciones

Ya que las diferencias visuales no son significativas, se ha optado por hacer un *streaming* con la calidad WVGA, ya que el tiempo de procesado de cada uno de los fotogramas es bastante inferior al resto. Este tiempo también depende de la máquina con la que se realicen los cálculos, lo que no influye en la decisión.

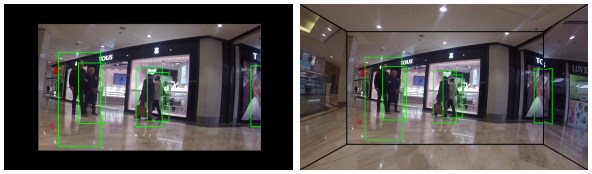
Además, el detector de OpenCv utilizado permite configurar de manera sencilla el rango de tamaño aceptable para los objetos que se quieren anotar ($maxSize$ y $minSize$ permiten indicar el tamaño máximo y mínimo del objeto a detectar, respectivamente).

En la configuración de las pruebas, se ha fijado $maxSize = Size(32, 32)$ y $minSize = Size(8, 8)$, adecuado para detectar personas en entornos de interior (donde se han realizado las pruebas). La variación de este parámetro puede hacer que el sistema obtenga más o menos falsos positivos (detecciones que no son personas) o falsos negativos (personas que no se han detectado), pero el sistema podría integrar cualquier otro detector suficientemente rápido, por lo cual este trabajo no se centra en un análisis muy detallado de la precisión del detector, su objetivo es proporcionar ejemplos de anotaciones para el prototipo.

3.2. MÓDULO 2: VISUALIZACIONES

En este módulo el objetivo ha sido conseguir mostrar el *streaming* de la cámara de manera adecuada, es decir, sin perder información de la imagen y además que la navegación en la escena pudiese ser factible. Para esto se plantearon varias posibilidades, y tras realizar varias pruebas de visualización, como se ha explicado en la Sección 2.2. La opción mas factible para mostrar la imagen en las gafas ha sido proyectarla en las distintas caras de un cubo, ya que proporciona una visualización aceptable sin añadir gran coste computacional.

Dentro de la opción de visualización elegida, hay varias posibilidades de configuración para proyectar la imagen a la pantalla de las gafas. Las consideradas (por ser rápidas y realistas) son:



(a) (b)

Figura 11: Resultados de la proyección del vídeo en las gafas: (a) Toda la imagen se proyecta en la cara frontal del cubo 3D que envuelve la escena; (b) La imagen se divide en secciones que se proyectan en la cara frontal y en las laterales del cubo.

- Mostrar toda la imagen en la cara frontal del cubo
- Mostrar gran parte de la imagen en la cara frontal, pero el resto en las caras laterales.

Como se puede observar en la Figura 11, la primera opción no da el resultado esperado, ya que se proyectan espacios en negro, por lo que en el prototipo final se ha configurado la segunda opción, detallada en la sección anterior. Además, esta opción da al usuario la sensación de estar envuelto por el entorno, lo que hace el sistema más interesante.

3.3. MÓDULO 3: MOVIMIENTO CÁMARA

En este módulo el objetivo ha sido conseguir que los movimientos de la cabeza del usuario (capturado con los sensores de las gafas) se hicieran llegar lo más rápido posible a la cámara para que cambiara el enfoque hacia la dirección de giro indicado por el usuario. Así la proyección de la imagen en las gafas cambia según el giro de la cabeza del usuario.

En la configuración de los giros se ha tenido en cuenta el caso en el que el usuario no hiciera un movimiento considerable de la cabeza (menos de 0.1 radianes). Es decir, si el usuario no movía lo suficiente las gafas como para que la escena que estaba proyectándose cambiase, no se modificaba la posición de la cámara. Esta comprobación se tiene en cuenta tanto en el giro horizontal como en el giro vertical. Con esto se consigue no añadir tiempo de procesado suplementario ya que para un movimiento pequeño no es necesario tener que llevar a cabo los envíos y cálculos de movimientos.

Se ha tenido en cuenta que el rango de las gafas es 360º tanto en la componente horizontal, como 360º grados en la componente vertical. Sin embargo, el *pan-tilt* utilizado para mover la cámara tiene un rango de movimiento de +\- 180º. Esto ha supuesto el tener que comprobar que si en las gafas se producía un giro superior al rango del *pan-tilt*, el *pan-tilt* la imagen proyectada no era la correcta.

3.4. VALIDACIÓN DE LA INTEGRACIÓN

En los experimentos de integración se ha validado la correcta ejecución de todos los módulos del prototipo. Estas pruebas han permitido verificar la correcta funcionalidad de todos ellos conectados, y verificar cuales son los componentes más críticos a optimizar para un funcionamiento más realista en el futuro. Esta validación se ha realizado en un ordenador de sobremesa equipado con Intel Core i-7, 32Gb de RAM y una tarjeta gráfica de alta gama (Nvidia GTX1070), componente esencial para el funcionamiento de las gafas de realidad aumentada.

Para entender los tiempos de ejecución del sistema y realizar un análisis preliminar de los mismos, podemos dividir los tiempos en dos bloques: tiempo de visualización por pantalla, T_{disp} (Ec. 3), y tiempo de respuesta a un movimiento del usuario T_{mov} (Ec. 3).

En cuanto al **tiempo de visualización**, sus componentes más significativas son las siguientes:

$$T_{disp} = t_{streaming} + t_{captura} + t_{anotacion} + t_{toOpenGl} + t_{display}, \quad (2)$$

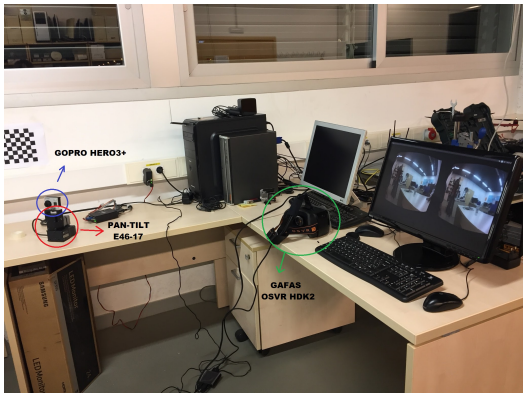
donde $t_{streaming}$ representa el tiempo que tarda en llegar el vídeo emitido desde la cámara hasta el ordenador donde se realiza el procesado, $t_{captura}$ representa el tiempo empleado en obtener cada uno de los fotogramas del *streaming*, $t_{anotacion}$ representa el tiempo necesario para generar las anotaciones del fotograma, $t_{toOpenGL}$ representa el tiempo empleado en transformar la imagen con anotaciones de una matriz de OpenCV a una textura de OpenGL para poder ser proyectada en las gafas y $t_{display}$ representa el tiempo necesario para transmitir la textura generada a las gafas para ser proyectada correctamente.

De estos tiempos, podemos considerar despreciables $t_{captura}$ y $t_{toOpenGL}$ ya que se han medido y son del orden de 1ms. El tiempo $t_{streaming}$ depende de la velocidad de la WiFi y no de nuestro sistema (sólo se podría mejorar con una versión mas rápida de la cámara y con una mejor conexión WiFi). Aunque este tiempo no se puede medir de manera sencilla, claramente es el cuello de botella en este apartado. El tiempo que depende directamente de nuestros módulos es $t_{anotacion}$, que depende de cual sea la calidad del vídeo y de con que máquina se procese. En nuestro caso, con calidad de vídeo WVGA, es una media de 0.82s por fotograma procesado.

En cuanto al **tiempo de respuesta al movimiento** del usuario, sus componentes principales son:

$$T_{mov} = t_{leerAng} + t_{toEuler} + t_{envio} + t_{exe}, \quad (3)$$

donde $t_{leerAng}$ representa el tiempo empleado en leer los movimientos que envían las gafas al sistema, $t_{toEuler}$ representa el tiempo empleado en transformar el movimiento enviado en forma de Cuaternio por las gafas a ángulos de Euler, ángulos con los que trabaja el *pan-tilt*, t_{envio} representa el tiempo necesario para que el movimiento transformado sea comunicado al *pan-tilt* y t_{exe} representa el tiempo que el *pan-tilt* emplea para reproducir el movimiento recibido y sincronizar su posición con la de las gafas.



(a)

(b)

Figura 12: Visualización del prototipo en funcionamiento. (a) Dispositivos utilizados. (b) Dispositivos en funcionamiento. El monitor muestra las dos vistas que se visualizan dentro de las gafas (en la pantalla del ojo izquierdo y derecho).

Como es de esperar, en este caso el cuello de botella también es el envío de los datos, ya que las operaciones son mínimas (lectura de un sensor que permite una frecuencia de lectura altísima, y una conversión de representación). Como en nuestro prototipo el *pan-tilt* está conectado directamente por cable al mismo ordenador que realiza el procesado, no resulta significativo. Sin embargo, en un entorno de aplicación más realista, donde esta información se enviaría de manera inalámbrica, resultaría en el cuello de botella principal (al igual que en el bloque anterior).

Por último, la Figura 12 muestra una imagen del sistema final utilizado para las pruebas de integración de este proyecto. Se pueden ver todos dispositivos utilizados en funcionamiento (cámara GoPro Hero3+, gafas OSVR HDK2, Pan-tilt E46-17) y el ordenador princi-

pal donde se realizan los cálculos. Además, para facilitar el desarrollo, la pantalla del ordenador muestra una copia de lo que se está proyectando en la pantalla de las gafas. En ella se puede observar la anotación de personas en funcionamiento. El código de este trabajo está disponible en un repositorio abierto [4].

4. CONCLUSIONES

Este trabajo ha descrito una prueba de concepto en la que se combinan distintas tecnologías mediante paquetes de *software open source* para construir un prototipo de control de un sistema de videovigilancia o monitorización remoto. Dicho sistema integra dos tecnologías clave en cada unos de sus extremos. Por un lado, una cámara móvil que puede ir embarcada en un dispositivo móvil tipo dron. Por otro lado, un visor de realidad virtual que permite una visualización más natural y poder dirigir el movimiento de la cámara remota. El hecho de que el sistema se ha desarrollado mediante las librerías de software libre OSVR y ROS, hace que esta prueba de concepto sea fácil de utilizar, replicar e integrar en otros proyectos.

Con el desarrollo de este prototipo se ha obtenido experiencia acerca de las dificultades que pueden plantearse, resumidas en dos aspectos: conflictos *software* causados por las todavía inmaduras y poco documentadas características de algunas librerías de realidad virtual, y latencias elevadas que requieren *hardware* especializado para que la experiencia de usuario sea más satisfactoria. Futuras mejoras podrían enfocarse a este último aspecto, probando dispositivos alternativos y optimizando las funcionalidades principales.

Agradecimientos

Este trabajo ha sido financiado por los proyectos UZCUD2017-TEC-06 y DPI2015-69376-R y el Gobierno de Aragón (Grupo DGA T45_17R). Los autores también quieren agradecer la colaboración de los Laboratorios Cesar (`https://www.fablabs.io/labs/laboratorioscesar`), por la cesión de parte del material utilizado en este proyecto.

English summary

VISUAL NAVIGATION FROM AN AUGMENTED REALITY HEADSET

Abstract

This work presents a prototype for remote surveillance or monitoring. It joins a mobile platform equipped with a camera and a virtual reality headset to display and control the monitoring. The system consists of a action-perception loop, which thanks to the sensors on the headset achieves two main goals: 1) displays a immersive view of the remote camera view to the user, 2) allows the user to control the remote camera (in the case of our prototype through a pan-tilt*) to point to the direction that the user head-motion indicates. The system has been developed integrated with open-source libraries for VR headsets (OSVR) and Robotics (ROS), which makes this proof of concept easy to replicate and integrate in other works.*

Keywords: Augmented reality, Remote monitoring, mobile robots, OSVR, ROS.

Referencias

[1] Y. Boger et al. OSVR: An open-source virtual reality platform for both industry and academia. In *Virtual Reality*, pages 383–384. IEEE, 2015. `https://www.osvr.org`.

[2] D. Calife, J. L. Bernardes Jr, and R. Tori. Robot arena: An augmented reality platform for game development. *Computers in Entertainment (CIE)*, 7(1):11, 2009.

[3] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *Conf. on Computer Vision and Pattern Recognition*, volume 1, pages 886–893. IEEE, 2005.

[4] C. García. Prototipo de navegación visual. TFG Ing. Informática, Universidad de Zaragoza, 2018. Disponible en `https://github.com/christianjaka94/osvr_unizar`.

[5] A. Giusti et al. A machine learning approach to visual perception of forest trails for mobile robots. *IEEE Robotics and Automation Letters*, 1(2):661–667, 2016.

[6] Headset HDK2. `http://www.osvr.org/hdk2` (último acceso Julio 2018).

[7] M. L. Heilig. Sensorama simulator, Aug. 28 1962. US Patent 3,050,870.

[8] Pan-tilt PTU-E46. `https://www.flir.com/mcs/view/?id=63554` (último acceso Julio 2018).

[9] B. Thomas et al. First person indoor/outdoor augmented reality application: ARQuake. *Personal and Ubiquitous Computing*, 6(1):75–86, Feb 2002.

[10] A. Weber and R. Thomas. Key performance indicators–measuring and managing the maintenance function. *Ivara Corporation*, 2005.