Universidad Zaragoza
1542

TRABAJO DE FIN DE MÁSTER EN INGENIERÍA INDUSTRIAL

CREACIÓN DE UN ENTORNO VISUAL DE VISIÓN PROTÉSICA

Creation of a Virtual Enviroment of Prothesic Vision

Autora

Violeta Estepa Ramos

Dirección

Rubén Martínez-Cantín

Co-Dirección

José Jesús Guerrero Campo

Escuela de Ingeniería y Arquitectura

Año 2020

AGRADECIMIENTOS

Antes de comenzar la redacción de este Trabajo de Fin de Máster, quiero agradecer a todas las personas que han hecho que pueda lograrlo.

En primer lugar, al departamento de Informática e Ingeniería de Sistemas, en especial al grupo de investigación formado por Rubén Martínez-Cantín, Josechu Guerrero y Melani Sánchez-García, gracias por permitirme ser una pequeña parte de este grupo de investigación tan novedoso y cuyo trabajo puede conseguir tantos beneficios sociales. Agradecer en especial a Rubén, mi tutor, por estar pendiente de todos los problemas que surgían, que no han sido pocos, y más aún en las condiciones telemáticas en las que ha tenido que desarrollarse este trabajo.

También al i3A por becarme para poder desarrollar un trabajo tan excepcional de investigación y a todos mis compañeros de carrera que siempre han sido un apoyo y espero que lo sigan siendo.

Finalmente, agradecer a mis amigos y especialmente a mi familia, por apoyarme no solo en este último periodo sino en todo momento hasta llegar aquí. Sin vosotros no habría sido capaz de hacer ni una milésima parte de lo que he hecho.

Muchas gracias.

RESUMEN

CREACIÓN DE UN ENTORNO VISUAL DE VISIÓN PROTÉSICA

El uso y creación de aplicaciones tecnológicas ha experimentado un desarrollo vertiginoso en las últimas décadas y, de entre todos los nuevos elementos que usamos en nuestro día a día, estos años se ha visto desarrollada notablemente la Realidad Virtual: una forma de interactuar con un entorno distinto al real, pero que se siente como parte de él. Además de conseguir representar realidades alternativas para jugar, interactuar, observar lugares, animales u objetos que no se pueden ver en el entorno real, esta tecnología también se puede utilizar para conseguir mejorar la calidad de vida de ciertas personas.

Éste es el caso de este proyecto, en el que se trata de realizar una simulación del funcionamiento de un implante retinal de visión protésica. Estos implantes son elementos que se incorporan al ojo de personas que padecen ceguera completa, de manera que, mediante impulsos en el nervio óptico, pueden recibir información de su entorno para así mejorar su calidad de vida.

El funcionamiento de éstos se basa en la captación de imágenes mediante cámaras en el implante y transmisión de la información –previamente tratada- utilizando impulsos eléctricos que estimulan el nervio óptico. En la actualidad, los pacientes con este tipo de implante pueden llegar a percibir una *pseudo-imagen* de hasta 40 x 40 *píxels* con hasta 8 niveles de intensidad. Esto significa que la información que captan es muy reducida y tiene que tratarse adecuadamente para poder mostrarla de la forma más completa posible.

En este trabajo se va a simular un implante con un dispositivo de Realidad Virtual (VR): las ***Gafas HTC Vive Pro***, las cuales llevan integradas dos cámaras frontales, de manera que se pueda captar la imagen exterior, tratarla mediante *software* y devolver la información. De esta forma se consigue, de forma inmersiva y en tiempo real, una simulación semejante a tener el implante retinal. Como se ha mencionado, el propósito de este tipo de estudios es desarrollar el tratamiento de la imagen para que la información recibida sea la máxima o la más adecuada y, para ello, se necesitan hacer pruebas que no son viables realizar sobre pacientes reales, sino que se usan simuladores como este que se va a desarrollar en el trabajo.

El tema principal que se va a abordar en este proyecto de Fin de Máster es el estudio y análisis de las diferentes opciones para simular los implantes retinales utilizando las Gafas VR. Así, durante este TFM, se van a mostrar las alternativas para su configuración, así como los resultados obtenidos y los futuros caminos a seguir para mejorar las aplicaciones de las HTC Vive Pro en el estudio de una optimización del uso de los implantes retinales.

CONTENIDO

MEMORIA

MEMORIA

CAPÍTULO 1: INTRODUCCIÓN

En la actualidad, según datos de la Organización Mundial de la Salud (OMS, *WHO*[1]), alrededor de 285 millones de personas padecen discapacidad visual, lo que supone que un 0'7% de la población mundial tenga dificultades para reconocer su entorno. En algunos casos, esta discapacidad está generada por algunas enfermedades degenerativas que causan pérdida de visión por la degeneración de las células sensoriales de la retina.

En los últimos años se ha desarrollado una nueva tecnología: la Visión Protésica (*Prothesic Vision*). A través de prótesis retinales se otorga percepción visual a personas que sufren ceguera mediante la transformación de imágenes en impulsos al nervio óptico –estímulos a las células de la retina- que se envían a través de dicho implante (Figuras 1.1 y 1.2).

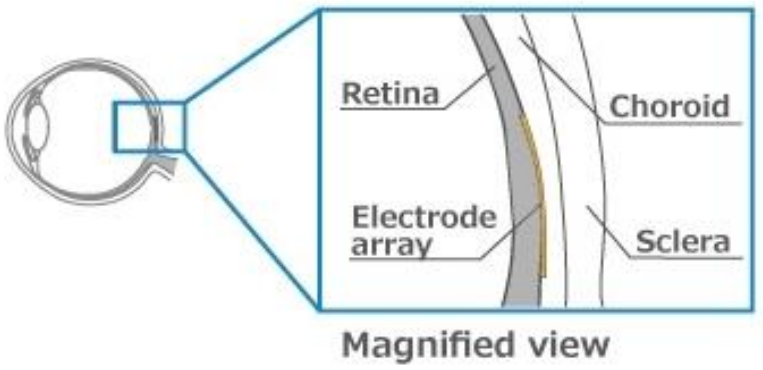


Figura 1.1: Esquema del implante subretinal (*https://www.nidek-intl.com/aboutus/artificial_sight/about_artificial_sight*)

Al utilizar estos dispositivos, los pacientes son capaces de percibir una serie de puntos de luz, denominados *fosfenos* [1], que son interpretados por el cerebro como información visual. Estos elementos son manchas luminosas que se obtienen al estimular la retina o la corteza visual de forma mecánica, eléctrica o magnética.

[1] WHO, World Health Organization

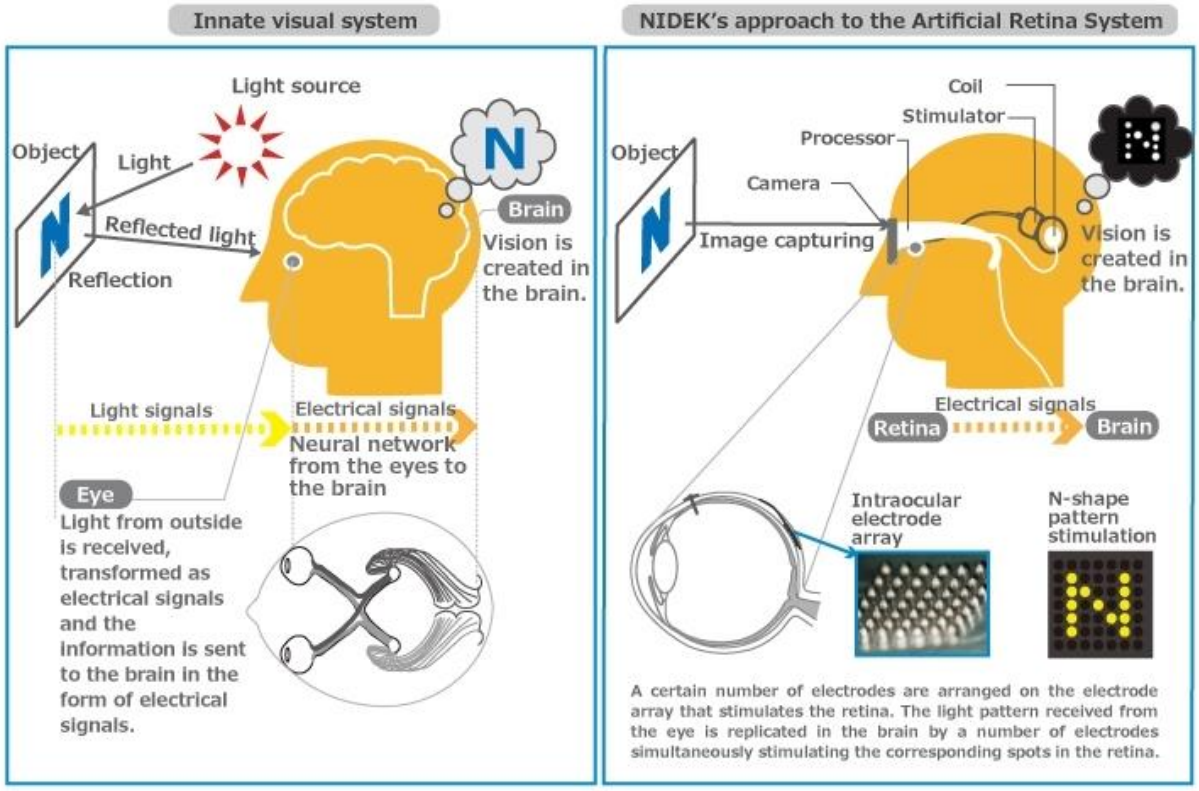


Figura 1.2: Esquema del funcionamiento del implante (*https://www.nidek-intl.com/aboutus/artificial_sight/about_artificial_sight*)

Los implantes retinales para personas con ceguera completa actuales tienen limitaciones a la hora de generar imágenes:

- Tamaño limitado: biológicamente los implantes han de tener unas dimensiones muy reducidas, lo que limita el desarrollo de estas prótesis.
- Poca resolución: se utiliza un rango de entre 60 y 1500 electrodos, lo que implica pseudo-imágenes de tamaño menor a 40 x 40 *píxels* [2].
- Estimulación limitada: solo se perciben unos determinados niveles de intensidad en los electrodos, aproximadamente 8, frente a los millones de colores de un paciente sano.
- Frecuencia de estimulación pequeña: se necesita tiempo para estimular las células retinales, así como para recuperarse y poder volver a enviar estímulos.
- Fallos de comunicación en el implante: aproximadamente el 10% de los fosfenos se consideran "muertos".

Todo esto hace que, a pesar de recibir información visual, los pacientes implantados no consiguen tener todos los datos y pierden elementos básicos en el desempeño diario como son el color, la textura, la luminosidad o las aristas y los límites de los elementos [3].

Para mejorar la percepción de los pacientes implantados, se están desarrollando técnicas de procesamiento de imagen para extraer y destacar la información relevante captada por las cámaras externas, de manera que se prueban señales específicas para: el reconocimiento de objetos; la lectura; el reconocimiento facial o la navegación, todo en el contexto de visión protésica. El procesamiento de imagen más utilizado es el de segmentación [4], el cual consiste en agrupar las regiones de la imagen basándose en el significado de éstas. En la Figura 1.3 se muestra como ejemplo el resultado al utilizar métodos de segmentación detectando siluetas y aristas del entorno mediante redes neuronales artificiales (*artificial neural networks)*.

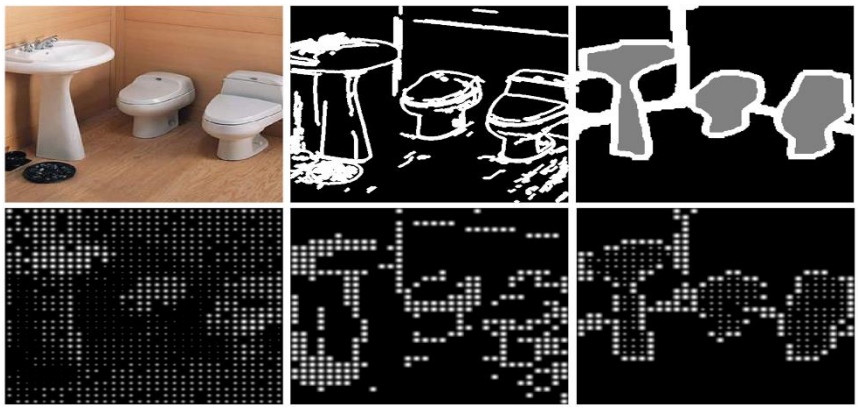


Figura 1.3: Ejemplo de generación de estímulos mediante procesamiento de imagen para SPV [4]

Sin embargo, el estudio y análisis de estas técnicas con pacientes implantados es muy complejo y costoso. Por este motivo, se están desarrollando simuladores de prótesis retinales utilizando diferentes mecanismos (pantallas de ordenador, sistemas de VR…). En este proyecto se busca crear un instrumento de laboratorio mediante dispositivos de Realidad Virtual (*VR Devices*) para llevar a cabo estas simulaciones de una manera integrada y flexible, lo cual presenta una serie de problemas como son:

- Procesar las imágenes en tiempo real: se ha de ser capaz de captar imágenes, tratarlas y mostrarlas sin que se note un retraso –*delay*-, de manera que se trabaje con la información lo más semejante posible a la que obtienen los pacientes.

- Reconstruir el efecto de los fosfenos reales: se han de convertir las imágenes en los puntos de luz que captan las personas implantadas y de una forma realista para poder trabajar sobre esa información.
- Conseguir un efecto inmersivo: representar de una forma fiel el funcionamiento de los implantes y de manera que el usuario esté inmerso en una experiencia real.

En definitiva, los problemas particulares de este proyecto se basan en conseguir representar fielmente en el simulador el funcionamiento del implante retinal. En este contexto se ha desarrollado este trabajo, el cual se ha llevado a cabo en colaboración con el departamento de Informática e Ingeniería de Sistemas de Universidad de Zaragoza, dentro del equipo de investigación de Visión y Robótica, junto con el Instituto de Investigación en Tecnología de Aragón (I3A). En dicho proyecto, se parte de un modelo de implante que genera fosfenos mediante estimulación eléctrica del nervio óptico, de manera que se forma una pseudo-imagen de 1024 fosfenos que se comportan como una imagen cuadrada de 32x32 *píxels*, aunque en este proyecto se permite cambiar dichos parámetros fácilmente.

1.1 Objetivos y Alcance

Con este trabajo se busca, mediante la utilización de gafas de Realidad Virtual (VR) provistas con dos cámaras frontales (HTC Vive Pro), ser capaces de captar las imágenes a través de las cámaras de las gafas y convertirlas a apariencia de fosfenos, para así poder mostrar lo que perciben quienes tienen el implante. Es decir, simular el comportamiento de la prótesis en tiempo real como pseudo-imágenes de fosfenos para, en un futuro, utilizar visión por computador e inteligencia artificial para conseguir estudiar y desarrollar mejoras de la información enviada y recibida sin necesidad de realizar ensayos clínicos en pacientes implantados -Figura 1.4-.

Figura 1.4: Esquema del proceso del proyecto

De esta manera, el proceso del proyecto consiste en captar las imágenes, tratarlas para que su apariencia sea semejante a la de estos 1024 fosfenos o puntos luminosos y mostrarlas por pantalla. Existe un paso intermedio entre la captación y la conversión a fosfenos: resaltar la

información relevante para mejorar los estímulos mediante técnicas de procesamiento de la imagen como las mencionadas.

Se busca crear un *software* fácilmente modificable y que se pueda mantener (robusto), para ser capaces de incorporar los diferentes procesamientos y técnicas que se desarrollen en futuras investigaciones, de manera que se mejore la información mostrada en las simulaciones y, por lo tanto, en los implantes reales. Es decir, conseguir un sistema flexible de simulación que sea portable e inmersivo. A partir de ello, los objetivos específicos para conseguir desarrollar este proyecto son:

- Comprensión de la herramienta *HTC Vive Pro:* un dispositivo de Realidad Virtual que cuenta con cámaras y pantallas propias
- Investigación para acceder a las imágenes de las cámaras duales incorporadas
- Programación de un algoritmo que las convierta en apariencia de fosfenos
- Visualización de las imágenes por pantalla del dispositivo

1.2 Metodología

A lo largo del proyecto se van a explicar las alternativas para realizar el desarrollo del *software* y, una vez obtenido el método a utilizar, se explican los procesos hasta conseguir el programa con las características requeridas. El trabajo se ha llevado a cabo siguiendo una metodología *Agile,* basada en desarrollar soluciones a través de planificación adaptativa, desarrollo evolutivo y mejora continua: se crea una aplicación ejecutable y se le añaden más funcionalidades, de manera que primero se ha buscado captar las imágenes; luego convertirlas a apariencia de fosfenos y, finalmente, mostrarlas por las pantallas integradas en el dispositivo VR. Cada una de las aplicaciones creadas se denomina MVP (*Minimum Viable Product,* Producto Mínimo Viable), en el que se prioriza el funcionamiento y, posteriormente, se añaden mejoras.

Asimismo, para reflejar el trabajo realizado en el desarrollo del proyecto se ha realizado un cronograma -Figura 1.5- en el que se enumeran las distintas actividades llevadas a cabo y la duración de las mismas, de forma orientativa.

Fase	Actividad	Duración
Estudio preliminar	Aprendizaje sobre prótesis, estudios previos, lectura de investigaciones	Prácticas previas
	Comprensión del código de conversión a fosfenos anterior en lenguaje *python*	
Puesta en marcha del sistema	Instalación de *software* requerido para ejecutar HTC Vive PRO	
Investigación de entornos de desarrollo	Unity: aprendizaje del motor gráfico (funcionamiento, tutoriales, *samples, scripts*…)	
	Aprendizaje lenguaje C# (utilizado en Unity)	
	Solución de problemas de versiones para Unity	
	Unreal: aprendizaje del motor gráfico (funcionamiento, tutoriales, *samples*, *blueprint, scripts*…)	
	Aprendizaje lenguaje C++ (utilizado en Unreal)	
	Solución de problemas de versiones para Unreal	
	SRWorks: puesta en marcha, ingeniería inversa	90 h
	HandTracking: puesta en marcha, ingeniería inversa	
	Vuforia: búsqueda de información	
Desarrollo del programa	Instalación y configuración de dependencias para trabajar con OpenVR	15 h
	Ejecución y estudio de código de los *samples* (C++)	
	Creación del código de conversión a fosfenos de una imagen existente y funciones necesarias en C++	20 h
	Realización de ingeniería inversa para llegar a captar imágenes con las cámaras de las Gafas VR	70 h
	Aprendizaje de OpenCV	10 h
	Conversión de las imágenes captadas a fosfenos	30 h
	Aprendizaje de OpenGL	10 h
	Realización de ingeniería para mostrar imágenes por las pantallas del dispositivo VR	80 h
	Mejora de la imagen (dos puntos de vista, calibración)	10 h

Figura 1.5: Cronología de desarrollo del proyecto

1.3 Estructura de la memoria

La memoria de este TFM se organiza en 6 capítulos y 2 anexos. En el capítulo 2 se realiza una introducción a la herramienta que se va a utilizar –las Gafas de VR-. En el tercer capítulo se describen las alternativas para poder realizar el proyecto en ellas, además de por qué se ha descartado cada una de ellas o la parte que se ha utilizado hasta llegar a la solución final. En el cuarto capítulo se detalla la forma de captar imágenes por las cámaras incorporadas en las Gafas de Realidad Virtual. En el capítulo 5 se explica el tratamiento de estas imágenes para pasar de una imagen distorsionada y en color a una imagen en fosfenos, similar a la información que reciben las personas implantadas. Para finalizar, en el sexto capítulo se comentan las conclusiones a las que se ha llegado con este proyecto, además de las futuras líneas de desarrollo que se generan tras este trabajo.

De forma adicional se han añadido dos anexos: Anexo A en el que se explica el funcionamiento de los archivos para conseguir ejecutar el programa y Anexo B con la programación en lenguaje C++ del programa principal que se utiliza para capturar las imágenes, tratarlas y mostrarlas en pantalla.

CAPÍTULO 2: SISTEMA DE REALIDAD VIRTUAL

En la actualidad, la Realidad Virtual es una tecnología poco estandarizada; que no tiene conocimientos, métodos y programas unificados y en la que se compite entre empresas para conseguir las mejores Gafas VR y aplicaciones. Para realizar el proyecto se utilizan las **HTC Vive PRO**, un Dispositivo de Realidad Virtual compuesto, principalmente y en lo que atañe a la aplicación de este trabajo, por dos cámaras duales frontales que captan las imágenes del exterior y una pantalla con dos lentes en la que se muestran las imágenes que el usuario percibe. Además, tiene sensores que permiten conocer la localización exacta de las gafas, así como la dirección en la que están enfocadas, de manera que se puede interactuar con un ángulo de 360º en las 3 dimensiones. En este proyecto se van a utilizar las cámaras externas para captar imágenes, tratarlas y mostrarlas por la pantalla en tiempo cuasi-real, simulando el funcionamiento del implante en los pacientes.

Descripción de la herramienta

Las HTC Vive Pro -Figura 2.1- incluyen, además de las Gafas de VR, dos controladores [I] principalmente utilizados como mandos en videojuegos y aplicaciones; y dos estaciones base [II] que se utilizan para triangular la posición de las gafas y permitir que se adecúe la imagen que se muestra por ellas a la localización del usuario y la orientación de la mirada del mismo.

Estas gafas de realidad virtual presentan características de inmersión en las que destacan tanto los gráficos como el sonido, ya que incorporan dos visores de alta resolución (HMD[2]) y un audio espacial en 3D (en el que se tiene en cuenta la orientación de las gafas en los 3 ejes). A continuación, se detallan las especificaciones de las cámaras, las pantallas y el ordenador necesario para conectarlas.

2.1 Especificaciones de las cámaras de las Gafas VR

Estas Gafas VR presentan unas cámaras duales frontales cuya principal función es la de calcular la profundidad de la sala y de los elementos en ella –a lo que denominan *Chaperone*- para así detectar los límites de la sala y poder hacer una experiencia sin peligro de impactos.

[2] HMD de las siglas *Head Mounted Display* o pantalla situada sobre la cabeza

Figura 2.1: Gafas de VR utilizadas para el proyecto *(https://www.vive.com/mx/product/vive-pro/)*

Se trata de dos cámaras de ojo de pez (*fisheye)* de 640 x 480 *píxels* cada una, con una capacidad de transferencia de vídeos sin compresión de hasta 10 Gbit/s -frente a las gafas originales de VIVE que tenían una capacidad de unos 6 Gbit/s-. Para este proyecto no supone una limitación la resolución de las imágenes de las cámaras, ya que el implante que se va a simular no va a utilizar todos los datos del entorno –se trata la imagen y se convierte en una *pseudo-imagen* de poca resolución-. La velocidad de comunicación y la frecuencia de actualización de imágenes tampoco supone una limitación ya que los implantes simulados presentan poca latencia: tienen unos tiempos de recuperación entre envío de señales (fosfenos) determinados y mayores a los que nos ofrece este dispositivo.

2.2 Especificaciones de las pantallas de las Gafas VR

El visor de las Gafas VR lo conforman dos pantallas AMOLED Dual 3.5” diagonal con una resolución de 1440 x 1600 *píxels* cada una (2280 x 1600 al combinarlos) cuyas imágenes se actualizan a una frecuencia de 90 Hz, casi un fotograma cada centésima de segundo.

Además, presentan una serie de sensores como son: sensor de gravedad, giróscopo, sensor de proximidad, sensor de distancia interpupilar *IPD* y sensor de posición de las pupilas. De esta manera, se puede variar la imagen que se muestra en función de hacia dónde se dirija la mirada -además de tener en cuenta la posición de la cabeza-.

2.3 Especificaciones usadas del ordenador

Las especificaciones de la *Workstation* –elementos del ordenador- para poder utilizar el dispositivo de Realidad Virtual han de ser similares a las utilizadas en el proyecto, que son:

- **Procesador**: Intel Core i7 - 9700K
- **Tarjeta gráfica**: NVIDIA GeForce RTX 2080 Ti
- **Memoria**: 32GB RAM
- **Sistema operativo**: Windows 10

CAPÍTULO 3: ENTORNOS DE DESARROLLO

En este capítulo se va a explicar la metodología para ser capaces de comunicarse con este dispositivo, así como las distintas alternativas que se han utilizado y la razón de que se hayan usado o no para llevar a cabo el proyecto. En la Figura 3.1 se muestra un esquema de las distintas capas de la Interfaz de Programación de Aplicaciones (API, *Application Programming Interfaces)* para las Gafas VR:

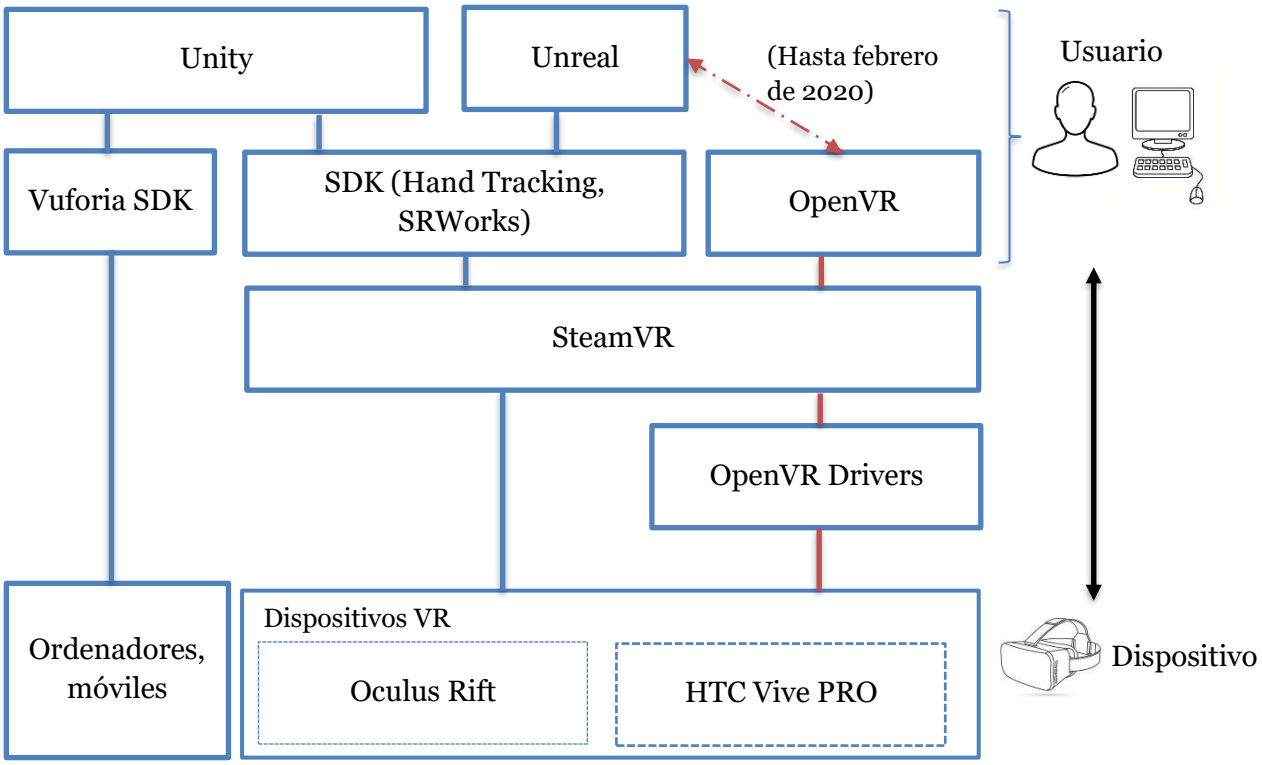


Figura 3.1: Esquema de drivers y APIs por capas para Gafas VR

En el esquema se representa, de forma jerárquica, el nivel de control (y por lo tanto de dificultad) a la hora de realizar una aplicación para los dispositivos de VR. Esto puede hacerse mediante los elementos comprendidos en la llave, mientras que lo que está fuera de ella ('SteamVR' y 'OpenVR divers') son elementos necesarios para la comunicación entre el ordenador y los dispositivos.

De entre las formas que el usuario puede utilizar se encuentran, jerárquicamente, las dos maneras de crear aplicaciones que se ejecuten en las gafas: la capa superior sería la de los motores gráficos, en los que se busca facilidad a la hora de añadir librerías, elementos y funciones frente a

conocer el funcionamiento interno de los programas, como es el caso de *Unreal* y *Unity* mediante el uso de SDK –camino en azul en la Figura 3.1-; y, bajando de nivel jerárquico, se complica la utilización a cambio de tener mayor control de los dispositivos, como es el caso de utilizar *OpenVR* –camino en rojo-. A continuación, se van a explicar las dos alternativas, la funcionalidad de cada una de ellas y, finalmente, el análisis de solución e ingeniería inversa que se ha realizado con cada una hasta llegar a la solución finalmente implementada:

3.1 Kits de Desarrollo Software (SDK, *Software Development Kit)*

Desde los creadores de las propias Gafas de VR se permitió a los desarrolladores de contenido el acceso a distintos *SDK* para poder crear aplicaciones nuevas y mejoradas para las propias Gafas VR (*https://developer.vive.com/eu/*):

- *Vive Wave SDK:* Plataforma y herramientas abiertas para desarrollar aplicaciones en cualquier Gafa de VR, no solo las de HTC.
- *Vive Sense SDK:* Acceso a los sensores de las gafas HTC con ejemplos como:
    - ***SRWorks SDK*** [5]: permite el acceso a las cámaras para la realización de aplicaciones de Realidad Mixta (XR, *Mixed Reality)*.
    - ***Hand Tracking SDK*** [6] *(*Seguimiento de la mano): mediante el uso de las cámaras frontales reconoce las manos y los gestos y las representa de forma esquemática en las pantallas de las Gafas.
    - ***Eye Tracking SDK*** *(*Seguimiento ocular): se obtiene la información del ojo de una persona, obteniendo valores que le permiten conocer hacia dónde está mirando.
    - ***Audio SDK:*** Habilita el acceso al audio de primer plano, de fondo o a una mezcla de ambos.
- *Viveport SDK:* Su funcionalidad es permitir que se publique y monetice el contenido desde diversas plataformas

De entre los 4 *SDK* de sensores que facilitan, los dos que tienen utilidad en este proyecto son SRWorks y seguimiento de la mano, por utilizar las cámaras frontales. Para ejecutar estos *Kits* se requiere un motor gráfico para funcionar, concretamente *Unity* o *Unreal*, dos ejemplos de entornos de desarrollo de juegos o aplicaciones en distintas plataformas (móviles, ordenadores, consolas, gafas de VR…). Los motores gráficos simplifican la programación para conseguir

mejores gráficos o funciones, a cambio de tener las limitaciones de los propios dispositivos. Aunque algunos estudios de desarrollo de juegos tienen los suyos propios, *Unity (unity.com)* y *Unreal (unrealengine.com)* son los más utilizados, ya que ambos destacan por ser intuitivos a la hora de utilizarlos, por ser libres y porque se adecúan a la plataforma en la que se va a implementar. A continuación, se explican sus características:

I) Unreal Engine

Permite al usuario un gran control del motor, lo que complica su utilización, pero permite que no se tenga funcionamiento inesperado de éste. Su programación se basa en diagramas de flujo, lo cual facilita el trabajo de programación, aunque su estructura es complicada y utiliza lenguaje C++, un lenguaje de bajo nivel –más cercano a la máquina-. Este motor gráfico incluye las herramientas necesarias para realizar una aplicación como son los editores de vídeo, de sonido y de código o la renderización de animaciones.

II) Unity 3D

Frente a *Unreal*, este motor se encuentra mejor documentado: permite mediante manuales y tutoriales libres conocer su uso y buscar cómo desarrollar una aplicación. Se programa con C#, un lenguaje orientado a objetos que cuenta con gran cantidad de ayudas y ejemplos para su entendimiento y utilización. En contra, no se controlan todos los aspectos de la programación y hace ejecuciones en segundo plano que pueden cambiar el funcionamiento de forma indeseada.

De este motor destaca que es capaz de trabajar con *Blender* (programa informático multiplataforma especializado en el modelado, iluminación, renderizado… de gráficos en 3D); Cinema 4D (*software* de creación de gráficos y animación) o Adobe Photoshop[3] (editor de imágenes y gráficos), entre muchos otros. Es decir, se permite la actualización automática de objetos creados con otros programas, de manera que no hace falta ni crearlos dentro de *Unity* ni volver a incorporarlos, facilitando el desarrollo gráfico de las aplicaciones y videojuegos.

Todo lo mencionado de las dos plataformas las convierte en herramientas muy versátiles para crear videojuegos multiplataforma a través de un editor visual y una programación mediante códigos de ejecución –*scripts*-. A continuación, se muestran -Figura 3.2- los distintos elementos que aparecen en *Unity* –prácticamente los mismos que en *Unreal*-:

[3] Blender: www.blender.org; Cinema 4D: www.maxon.net; Adobe Photoshop: www.adobe.com

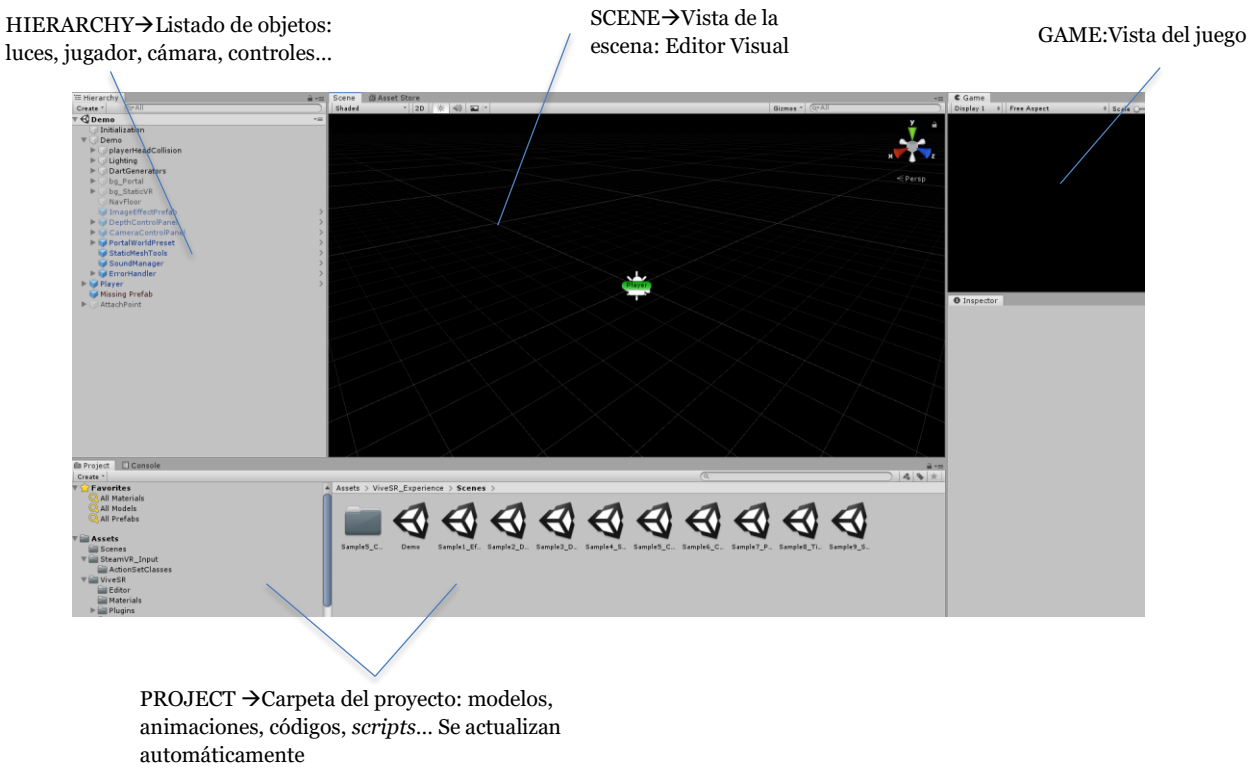


Figura 3.2: Funcionamiento del motor gráfico *Unity/ Unreal*

En la pestaña *Hierarchy* (Jerarquía) se muestran tanto los distintos objetos que hay como las dependencias entre ellos, pudiendo añadirlos, eliminarlos o editarlos de una manera intuitiva. En la pestaña *Scene* (escena) se edita el programa sin necesidad de código, moviendo elementos como el personaje, los objetos o incluso las luces. Todos estos cambios se visualizan en *Game* (juego) y todos los objetos que se utilizan y sus texturas, así como las funciones que desempeñan se encuentran en las carpetas del proyecto *Project*.

La creación de objetos, como se ha mencionado previamente, se puede realizar mediante diversos programas de edición, mientras que las funciones se programan con *scripts* en lenguaje C++ en *Unreal* y C# en *Unity*, aunque normalmente ya se incluyen predefinidos o se importan desde librerías externas.

Mediante los Kits de Desarrollo Software (SDK, *Software Developement Kit)* se obtienen las librerías y herramientas necesarias para los motores gráficos para ser capaces de acceder a las distintas aplicaciones que se quieren desarrollar. A continuación, se explican los dos SDK utilizados y mencionados:

3.1.1 SRWorks SDK

Mediante este Kit de Desarrollo *Software* se incluyen los ejemplos –*samples*- y rutinas del programa –*plugins*- que permiten, tanto para *Unity* como para *Unreal*:

- Acceso a los datos de profundidad de la escena: Módulo de profundidad (*Depth module)*
- Realización de un mapeo espacial con mallas estáticas y dinámicas (*Spatial Mapping – static and dynamic meshes)*
- Situación de objetos virtuales en el entorno real (*Placing virtual objects in the foreground or background*)
- Interacción con objetos virtuales en tiempo real y mediante interacciones manuales (*Live interactions with virtual objects and simple hand interactions*)
- Segmentación semántica mediante un módulo de Visión con Inteligencia Artificial IA (*AI Vision module for semantic segmentation*)

Es decir, utiliza las cámaras frontales de las *HTC Vive Pro* y trabaja con esas imágenes que capta, como se puede ver en las Figuras 3.3 y 3.4, en las que se aprecia que está interactuando con el entorno real al añadir elementos virtuales (un mando en la primera y el mapeo sobre las sillas y la mesa en la segunda).

Figura 3.3: Ejemplo del funcionamiento de la interacción de SRWorks (*https://developer.vive.com/resources/vive-sense/sdk/vive-srworks-sdk/*)

Figura 3.4: Ejemplo del funcionamiento del escaneo espacial de SRWorks (*https://developer.vive.com/resources/vive-sense/sdk/vive-srworks-sdk/*)

3.1.2 Hand Tracking SDK

Este SDK permite utilizar las cámaras para captar las manos del usuario –Figura 3.5 –, con lo que se hace un reconocimiento de la posición de éstas tomando datos de la localización de sus articulaciones, para así transformarlos y representar un esquema de ellas en la Realidad Virtual.

Figura 3.5: Ejemplo de funcionamiento de Hand Tracking (https://developer.vive.com/resources/vive-sense/sdk/vive-srworks-sdk/)

3.1.3 Vuforia Engine SDK

Aunque no se incluye dentro de los SDK que aporta Vive, existe otra opción para utilizar las cámaras incorporadas: una herramienta integrada en *Unity* llamada *Vuforia Engine SDK*[7]. *Vuforia* es una plataforma *software* para crear aplicaciones de Realidad Aumentada/Mixta –Figura 3.6-.

Figura 3.6: Ejemplo del funcionamiento de Vuforia *(https://developer.vuforia.com/)*

Una vez se han enumerado los distintos SDK disponibles y desarrollados por los creadores de los dispositivos VR, y mencionadas las funciones que realizan cada uno de ellos, se procede a explicar el camino alternativo al azul en la Figura 3.1, el rojo:

3.2 OpenVR API

OpenVR [8] es una *API (*Interfaz de Programación de Aplicaciones) desarrollada por *Valve*. En el esquema de la Figura 3.1 se observa una línea discontinua entre OpenVR y el motor gráfico, debido a que, hasta febrero de 2020 sí tenían comunicación entre ellas [9]. Por eso, debido a que este proyecto se desarrolló más tarde y se quiere que funcione en el futuro, no se utiliza *Unity* para esta alternativa.

Esta herramienta permite el acceso al *hardware* de Realidad Virtual sin basarse en las Gafas de VR en concreto ni, por lo tanto, en el *SDK* del vendedor o dispositivo. Esto implica que puede actualizarse independientemente del dispositivo, añadiendo robustez frente a actualizaciones de *software*. Asimismo, OpenVR es portable, con lo que la aplicación que se desarrolle con él es independiente del dispositivo a utilizar (HTC, Oculus…) y al sistema operativo en el que se ejecuta

(Windows, Linux…), implicando una mejora frente a los métodos anteriormente descritos. Esta API incluye tanto los ejemplos –*samples*- como las librerías y los códigos en lenguaje C++, de manera que se puede acceder a la información de, entre otras, una aplicación que muestra por pantalla la imagen de una de las cámaras de las Gafas VR.

3.3 Análisis de las alternativas

Una vez enumeradas las alternativas para realizar el proyecto y las funciones de cada una de ellas, se procede a explicar la versión utilizada de éstas, la puesta en marcha, el proceso de ingeniería inversa para conseguir la aplicación deseada y las razones de utilizarlo o no para la solución final:

3.3.1 SRWorks

Para este *Kit,* la versión utilizada es la última disponible en el momento de realizar el proyecto: 0.9.0.3, que incluye los modelos y archivos para ambos motores gráficos. Con esta opción se intenta, a partir de los ejemplos dados, llegar a la toma de imágenes que captan las cámaras; es decir, realizar un proceso de ingeniería inversa. No obstante, aunque se logran ejecutar los distintos ejemplos e incluso cambiar elementos de ellos, no se puede acceder a las imágenes de las cámaras directamente, de manera que no sirve para realizar el proyecto.

Además, para poder ejecutarlo en *Unity*, se necesitaba una versión de *.NET Framework*[4] anterior a la que se tenía en el ordenador con Windows 10 (4.7.2), con lo que no podía ejecutarse y daba error de librerías *dll*[5] y no podía funcionar. A parte de no poder abrir los ejemplos, este fallo implica que no es un método robusto, ya que en un futuro se quiere seguir avanzando en el proyecto y éste no puede quedarse obsoleto por las versiones del programa.

3.3.2 Hand Tracking

Mediante este SDK se intenta hacer, al igual que en el caso de *SRWorks*, un proceso de ingeniería inversa para acceder únicamente a la toma de imágenes de las cámaras. La versión utilizada es la última en el momento de realización del proyecto: la 0.9.1, que incluye los modelos y archivos para ambos motores gráficos. Con este Kit, descargable tanto para *Unity* como para *Unreal,* se obtienen los valores de localización de las articulaciones y distancias entre ellas, entre

[4] Conjunto de librerías y archivos de Windows (*www.microsoft.com*) que permiten el funcionamiento de aplicaciones y programas
[5] DLL *(Dynamic Link Lybrary*, Biblioteca de Enlace Dinámico) son conjuntos de código común (compartido entre varias aplicaciones) que permite que funcionen en un sistema operativo determinado

otros, pero no se puede acceder a las imágenes tomadas por las cámaras directamente. Es decir, internamente toma fotogramas y datos de ellas, pero no son de acceso público, con lo que no sirve este *kit*.

Además de no poder acceder a las imágenes, se presenta un problema adicional a tener en cuenta que es que esta versión requiere, para poder trabajar con *Unreal*, una versión de Visual Studio (*https://visualstudio.microsoft.com/es/*) antigua: 2017; es decir, los ejemplos no están actualizados, lo cual genera errores a la hora de editarlo y ejecutarlo.

3.3.3 Vuforia

A pesar de estar enfocado a juegos o aplicaciones en los que se une el entorno y los elementos añadidos virtualmente, *Vuforia* sólo se puede utilizar para aplicaciones en *Android, iOS* y *Windows*, por lo que no es compatible con HTC Vive y, por lo tanto, con este proyecto.

3.3.4 Motores gráficos

En cuanto a la comparación entre motores gráficos, en la realización de este proyecto se ha aprendido a usar ambos y los respectivos lenguajes de programación (C++ y C#) y se ha concluido que, para estos SDK descritos, es mejor *Unity* debido a que es más fácil su aprendizaje y utilización y, sobre todo, porque es más robusto al no tener tanto problema de versiones como presenta *Unreal,* muy a tener en cuenta para desarrollos futuros. No obstante, ninguno de los motores gráficos se ha podido emplear debido a que no se podían utilizar los *Kits* para la aplicación deseada.

3.3.5 OpenVR

Mediante los ejemplos que incluye y al tratarse de un sistema de más bajo nivel (todo se realiza mediante código), la utilización es más accesible y, a cambio, más compleja –se han de definir todos los parámetros y ejecuciones-. Asimismo, se le añade el problema de que carece completamente de documentación: prácticamente no se explica su funcionamiento ni en la página *web* ni en el mismo código.

No obstante, al tener los ejemplos y poder acceder a su programación, se utiliza como base para realizar la aplicación propia con las especificaciones deseadas en este proyecto. Finalmente, ésta va a ser la opción que se va a utilizar para desarrollar el proyecto, mediante archivos en

lenguaje C++ para los que se utilizará Visual Studio como editor de código, ya que *OpenVR* no tiene ninguna herramienta de ayuda a la programación.

Para finalizar este apartado cabe destacar que, además de haber analizado las distintas alternativas hasta conseguir la óptima, se ha aprendido a utilizar programas de creación de videojuegos (*Unity* y *Unreal)* y de edición y compilación de soluciones (*Visual Studio* 2019), además de diversos lenguajes como son C++ y C# y que todas las opciones han sido probadas, analizadas y estudiadas hasta llegar a la forma final de llevarlo a cabo.

3.4 Evaluación y pruebas de OpenVR

La herramienta finalmente utilizada ofrece, además de las librerías necesarias, una serie de archivos con ejemplos, *samples,* de entre los cuales en relación a este trabajo destacan dos:

- *Tracked Camera:* en este ejemplo se accede a las cámaras incorporadas en las Gafas VR
- *HelloVR OpenGL:* en este ejemplo se muestran en las pantallas del propio dispositivo de VR una serie de imágenes predeterminadas

Partiendo de ellos se generan otros ejemplos siguiendo una metodología *Agile*, en la que se desarrollan soluciones de forma evolutiva, con una mejora continua de las aplicaciones, lo cual se describe en los siguientes capítulos.

Para poder utilizar *OpenVR* se ha de crear un proyecto con el que se puedan compilar las soluciones y generar archivos ejecutables tanto de los ejemplos proporcionados como de los programas propios que se desarrollan. En el Anexo A se explica con detalle el procedimiento a seguir para configurar y generar el proyecto con la herramienta CMake (*cmake.org)*. De forma resumida y en lo que se refiere al proyecto, se ha utilizado la versión del programa 3.17.2 y se ha necesitado instalar Qt5 (versión 5.15.0), un entorno de trabajo orientado a objetos utilizado para desarrollar *software* que incluye las librerías y herramientas requeridas para generar este proyecto con *OpenVR*.

Asimismo, se han necesitado otras librerías adicionales para poder generar el proyecto como son las de *OpenCV* (*opencv.org*); *SDL2.dll,* una librería que tiene el propio sistema operativo Windows pero que no es capaz de encontrar en este proyecto y GLFW, una librería de OpenGL necesaria para crear texturas con las imágenes. Estos archivos deberían estar incluidos en las carpetas de *OpenVR* como librerías propias; no obstante, se han de instalar adicionalmente bien para poder generar el proyecto, bien para ejecutarlo (en el caso de SDL2).

CAPÍTULO 4: CAPTACIÓN DE IMÁGENES

En este capítulo se va a proceder a explicar, una vez mencionadas todas las alternativas estudiadas para conseguir la interacción con las gafas, el proceso de programación de la aplicación y los resultados obtenidos. Para conocer con detalle la programación de la aplicación final se recomienda la lectura del Anexo B.

4.1 Punto de partida

Como se ha mencionado, las Gafas de VR *HTC Vive Pro* llevan incorporadas dos cámaras frontales –duales- de manera que pueden captar imágenes del entorno sin la necesidad de un elemento adicional como una imagen predeterminada o una cámara externa. Para poder acceder a las imágenes de las cámaras se parte de los recursos incluidos en OpenVR, en concreto de un ejemplo: *tracked_camera_openvr*, en el que se pueden observar las imágenes que toma la cámara derecha de las Gafas VR por la pantalla del ordenador -Figura 4.1-.

Al ejecutar este ejemplo se obtiene, además del vídeo en tiempo real como sucesión de los fotogramas tomados por la cámara derecha, información sobre el tamaño de la imagen (*Frame Size*), la frecuencia de toma de imágenes (*Frame Sequence*) y la posición de la cámara en todo momento: se saben las coordenadas respecto de las dos estaciones base [2] (*Pose Value)* y las velocidades de movimiento (*Pose Velocity)* y de rotación de las Gafas (*Pose Angular Velocity)*.

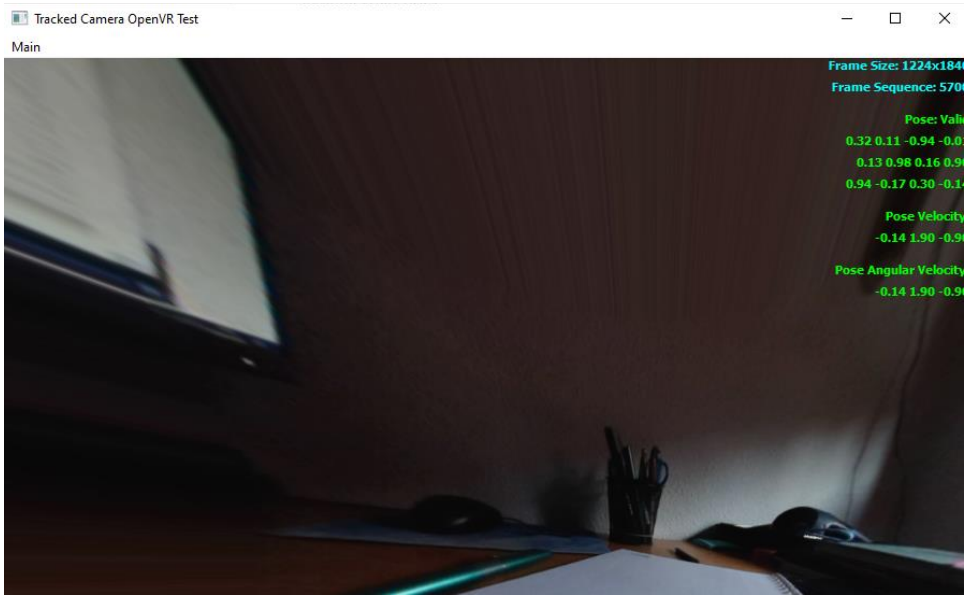


Figura 4.1: Imagen obtenida con sample tracked_camera_openvr

4.2 Experimentación

Una vez obtenida la imagen de una de las cámaras de las Gafas VR, se busca la forma de ser capaces de trabajar con estas imágenes para procesarlas previamente a mostrarlas. El ejemplo utiliza Qt5 para captar y mostrar las imágenes, pero, al querer hacer tratamiento de la imagen, se modifica el código para utilizar las imágenes con *OpenCV*[6], la biblioteca más utilizada para visión artificial. Para ello, se realiza un programa que, al ejecutarse, inicia las cámaras -como en el *sample*- y comprueba si existe algún error; posteriormente, si todo funciona de forma correcta, obtiene la imagen de las cámaras como un fotograma (*frame)* y la guarda en un *buffer* (un espacio temporal de memoria), como dato de tipo matriz de OpenCV. Finalmente, interpreta y muestra los datos de esa imagen a partir de ese *buffer,* consiguiendo representar por la pantalla del ordenador una imagen. Esta imagen se encuentra distorsionada -Figura 4.2-: los bordes sufren un efecto parecido al *zoom* y los canales de color se muestran intercambiados –de RGB a BGR[7] o viceversa: se puede apreciar que la mesa aparece en color azul, al contrario que en la Figura 4.1-.

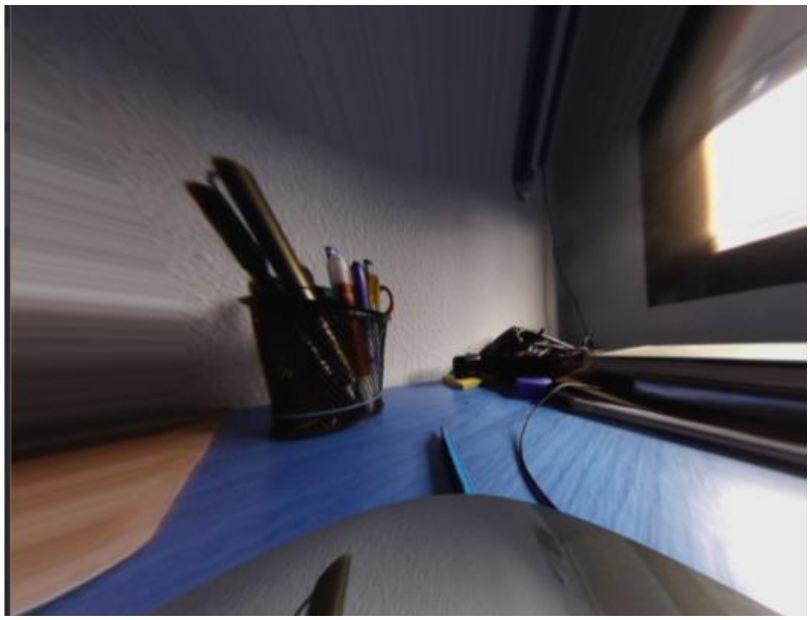


Figura 4.2: Imagen obtenida con el *software* de Valve Corporation

[6] OpenCV (Open Computer Vision) es una biblioteca libre de visión artificial desarrollada inicialmente por Intel. Sus aplicaciones son, entre otros: reconocimiento facial, de gestos y de objetos, segmentación, seguimiento… *(https://opencv.org/)*

[7] BGR son las siglas en inglés de azul (*Blue*), verde (*Green*) y rojo (*Red*), en referencia al orden de los canales de color de las imágenes.

Al intentar trabajar sobre esta imagen se descubre que el tamaño real que tiene la imagen no coincide con la dimensión que se muestra por pantalla, de forma que se realizan distintos recortes, descubriendo que la aplicación obtiene una imagen distinta a la que se muestra. La información que realmente obtiene -Figura 4.3- presenta los fotogramas de la cámara derecha sobre los de la izquierda y en forma de X –con deformación propia de las cámaras-. Es decir, el programa muestra una parte de la imagen realmente captada por las cámaras, completando la información que le falta reproduciendo los *píxels* externos.

De esta manera se empieza a trabajar a partir de estas imágenes, ya que se puede obtener un segmento con información nítida, en color y con dos puntos de vista: cámaras derecha e izquierda. El hecho de poder acceder a imágenes desde dos puntos de vista implica que en futuras aplicaciones se pueda hacer procesamiento de imagen tal como la triangulación o el cálculo de profundidad.

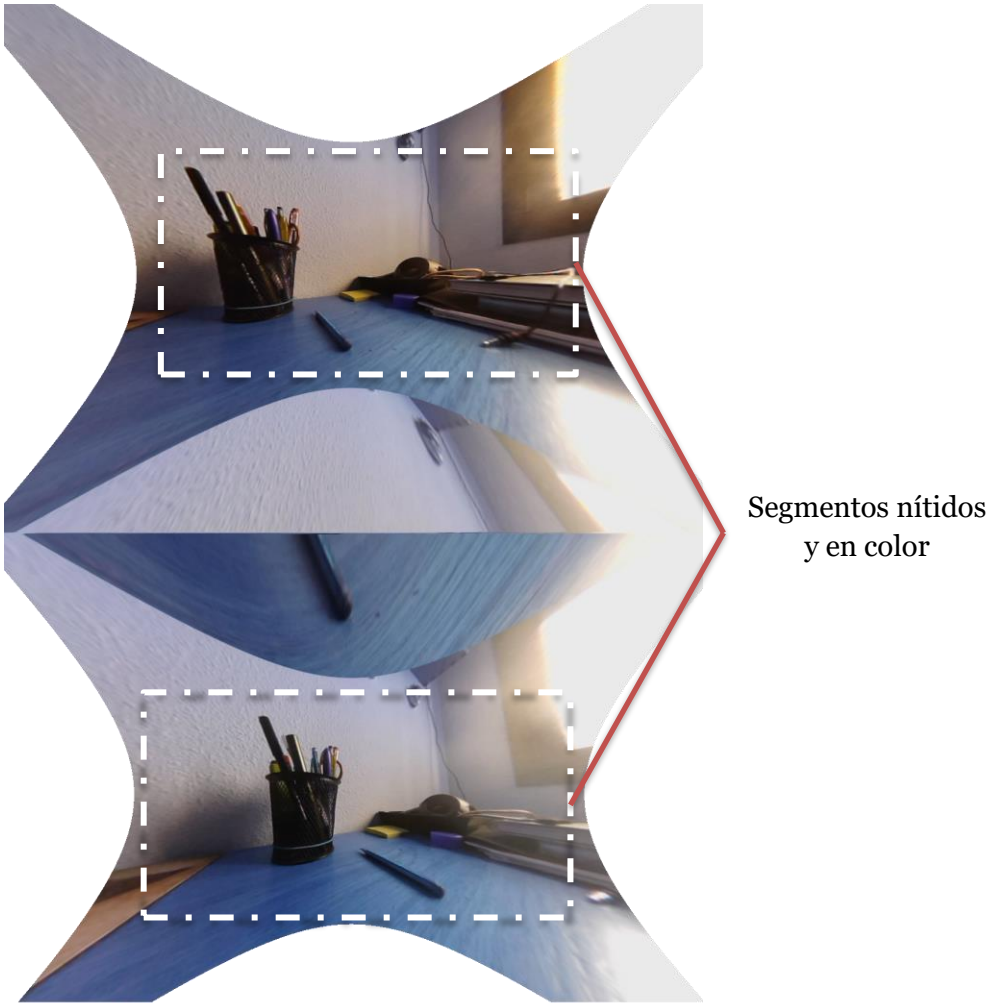


Figura 4.3: Imagen obtenida con OpenVR_Test con esquema de segmentos útiles para el proyecto

CAPÍTULO 5: CONVERSIÓN A FOSFENOS

En este capítulo se va a explicar el proceso de tratamiento de imágenes que se ha llevado a cabo en el proyecto, una vez captados los fotogramas, hasta la representación en forma de fosfenos.

El tratamiento de las imágenes captadas por las cámaras de las Gafas VR requiere que no tengan distorsión y que, preferiblemente, sean cuadradas. El tamaño de éstas no influye, pero ayuda a la hora de representar la apariencia de los fosfenos, ya que son señales similares a los puntos, por lo que es mejor utilizar imágenes cuadradas y cuya dimensión en *píxels* sea un múltiplo del número de fosfenos, para que no se pierda información o aparezca de forma no deseada en la simulación.

5.1 Obtención de imágenes nítidas

Como se ha explicado al comienzo de la memoria, la información que se envía a la prótesis es muy importante porque tiene que contener los elementos necesarios para entender el entorno: no puede desperdiciar más información de la que se pierde con el implante ni puede enviar información duplicada o que no aporte datos al paciente. De esta manera, han de recortarse los fotogramas obtenidos para tratar con imágenes que no estén ni distorsionadas ni difuminadas. Se seleccionan los segmentos centrales de la imagen porque son nítidos y se escoge un tamaño de imagen cuadrado y múltiplo del número de fosfenos.

En este proyecto se parte de que los implantes tienen 1024 fosfenos, lo que implica una imagen cuadrada de 32 x 32. Por ello, las imágenes capturadas por ambas cámaras se van a recortar en 384 *píxels* de ancho y de alto -Figuras 5.1 y 5.2-, de manera que cada fosfeno representa un conjunto de 12 x 12 *píxels*.

Todas las funciones realizadas en el proyecto tienen como variables tanto las dimensiones a las que se va a recortar la imagen como el número de fosfenos en filas y columnas o los niveles de intensidad que son capaces de reconocer los implantes. Es decir, todos los valores de dimensión del fosfeno, números de niveles de gris o tamaños son orientativos para entender con ejemplos el funcionamiento de la programación, pero pueden variar en función del implante a simular o del estudio de imagen que se quiera realizar.

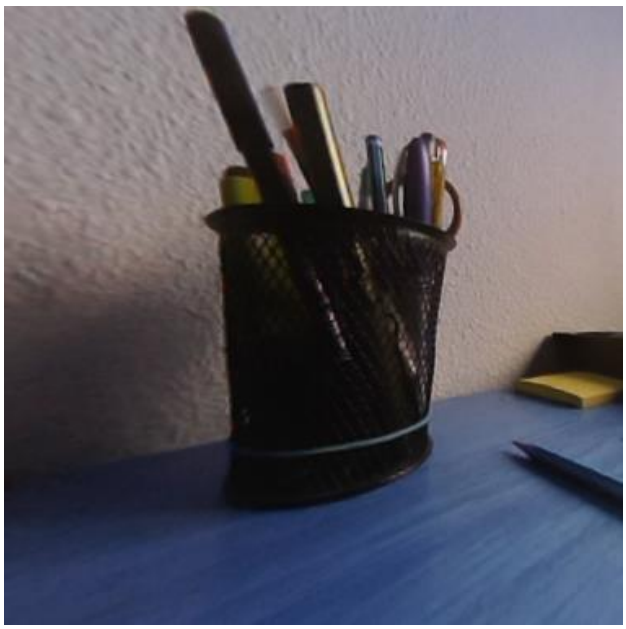


Figura 5.1: Recorte de imagen por cámara izquierda

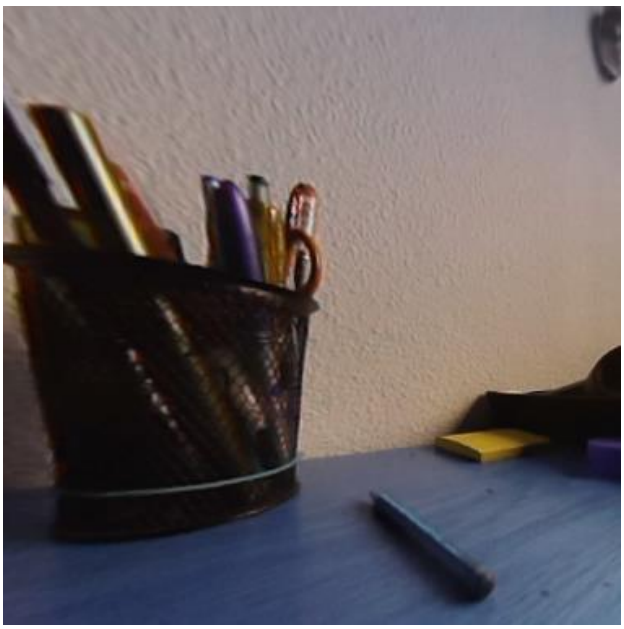


Figura 5.2: Recorte de imagen por cámara derecha

En el desarrollo de los capítulos siguientes en los que se obtiene la apariencia a fosfenos, se parte de un estudio previo [3] en el que se tiene el código en *Python,* y se transforma a lenguaje C++. Es decir, se han realizado mediante creación propia para la realización de este proyecto todas las funciones no incluidas en librerías de C++: cálculo de valor medio, cálculo de la mediana, *meshgrid* para obtener una matriz de coordenadas a partir de vectores y creación de un vector de valores aleatorios.

5.2 Conversión a un canal

Una vez se tienen las dos imágenes en formato de matriz de dimensión 384 x 384, se han de convertir de BGR (color) a BW (blanco y negro); es decir, pasar de tener tres canales de imagen a uno solo -Figura 5.3-.

5.3 Transformación a fosfenos

Una vez se ha convertido a un único canal, se ha de pasar la imagen a apariencia de fosfenos. Cada fosfeno va a agrupar un determinado número de *píxels* de la imagen en blanco y negro (en este caso un fosfeno tendrá la información de 144 *píxels*). Para entender el funcionamiento de la conversión a fosfenos, en el esquema de la Figura 5.4 se muestra cómo una imagen cuadrada de 384 x 384 *píxels* se divide en 34 x 34 fosfenos de dimensión 12 x 12.

Figura 5.3: Fotograma cámara derecha BW (una capa)

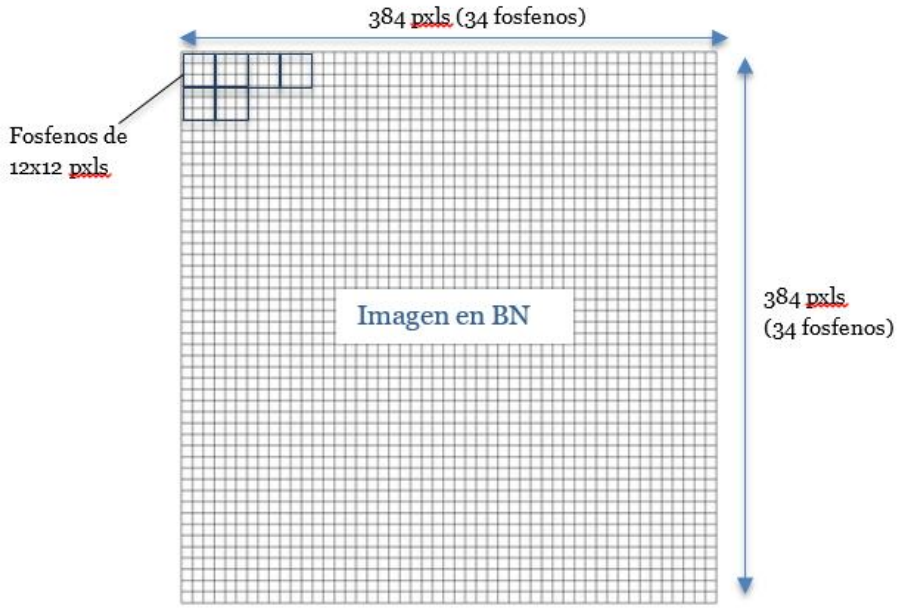


Figura 5.4: Esquema de conversión de la imagen a fosfenos

Primero, se necesita una función que genere la apariencia de los fosfenos; es decir, que en función del número de *píxels* que va a agrupar cada fosfeno (en este ejemplo 12 de ancho y 12 de alto), cree las posibles apariencias que puede tomar cada fosfeno en base a los niveles de

intensidad/ gris que es capaz de captar el implante (para este estudio 8). En el Algoritmo 1 se muestra esta creación de los fosfenos, donde se observa que, mediante una función, se introduce el número de *píxels* que va a abarcar cada fosfeno en filas (X) y columnas (Y), el número de niveles de gris o intensidad que reconoce el implante y un valor σ para mejorar la apariencia de éstos utilizando una ecuación Gaussiana.

---

función $generarAparienciaFosfenos(sizeX, sizeY, nivelesGris, sigma)$

$posX = sizeX/2$

$posY = sizeY/2$

$X, Y = meshgrid$[8] $(posX, posY)$

$localDistancia = (X - posX)^2 + (Y - posY)^2$

for $(k = 0; k < nivelesGris; k++)$

$$fosfenos = 255 \cdot e^{-\frac{localDistancia}{2\left(\sigma \cdot \frac{k+1}{nivelesGris}\right)^2}} \cdot \frac{k+1}{nivelesGris}$$

end for

devolver $fosfenos$

end función

Algoritmo 1: Generación de la apariencia de los fosfenos

En esta función se crea una matriz de 3 dimensiones: *sizeX* x *sizeY x nivelesDeGris* -Figura 5.5-, de manera que se generan todos los valores que pueden tomar el conjunto de *píxels* que conforman un fosfeno -Figura 5.6-. Es decir, se generan al principio las 8 –en este ejemplo- posibles apariencias que puede tener un fosfeno (los *píxels* que lo conforman) para, en la transformación a fosfenos de cada fotograma, no tener que calcular el valor de todos los elementos de la matriz sino sustituirlos por estos valores precalculados. Este método ahorra tiempo de

[8] La función *Meshgrid* sirve para generar dos matrices cuadrícula de coordenadas 2D con filas de longitud *y* y columnas de longitud *x* (*https://www.mathworks.com/help/matlab/ref/meshgrid.html*) para ayudar a calcular la distancia desde el centro y hacer una apariencia de 'punto'.

computación y evita retrasos (*delay*) en la muestra de las imágenes ya que sólo se calculan los posibles valores de los fosfenos una vez en toda la ejecución y no 1024 veces por cada fotograma.

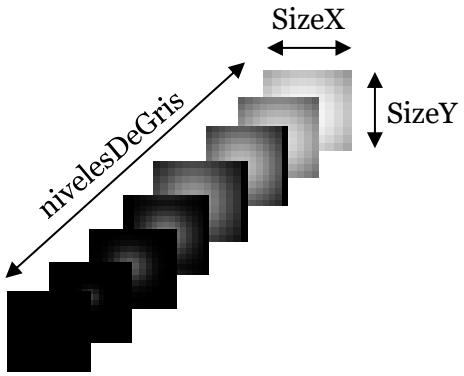


Figura 5.5: Matriz generada con el Algoritmo 1

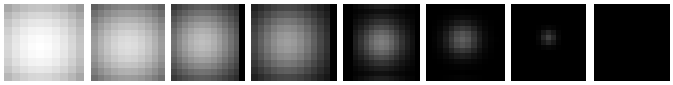


Figura 5.6: Posibles apariencias de los fosfenos con 8 niveles de intensidad, de mayor a menor

De esta manera, teniendo la imagen de un único canal (en blanco y negro) y una función que crea las posibles apariencias de los fosfenos en base a su dimensión y los niveles de intensidad, se procede a procesar la imagen. En el Algoritmo 2 se muestra cómo se recorre la matriz –imagen- a lo largo de las filas y de las columnas y se agrupan los *píxels* para convertirse en fosfenos. Posteriormente se halla el valor medio o el valor mediano del conjunto de *píxels* que conforman cada fosfeno –existen las dos funciones para elegir el mejor comportamiento-, con lo que se obtiene un valor de 0-255 (8 bits) denominado luminancia.

A partir de esto, se crea una variable llamada “Selector de luminancia” que determina qué nivel de gris aparecerá en cada fosfeno y que está directamente relacionada con la luminancia: es mayor cuanto mayor sea el valor medio o mediano de los *píxels* que conforman el fosfeno y, por lo tanto, tiene una apariencia más blanca/intensa de las mostradas en la Figura 5.5. Este proceso de calcular el valor que caracteriza cada grupo de *píxels*, llamar a la apariencia de fosfeno que le corresponde y sustituir cada valor por el correspondiente se realiza con todos los *píxels* de la matriz –imagen recortada y en blanco y negro-.

función $convertirFosfenos(imagen, nivelesGris, numeroFosX, numeroFosY, usarMediana)$

$ancho = imagen.ancho$

$alto = imagen.alto$

$stepX = ancho/numeroFosX$

$stepY = alto/numeroFosY$

$apparienciaFos = generarFosfenos(stepX, stepY, nivelesGris, sigma = 7)$

para cada fosfeno i:

if $(usarMediana)$: $luminancia = mediana(fosfeno_i)$

else: $luminancia = media(fosfeno_i)$

end if

$SelectorLuminancia = luminancia * nivelesGris / 255$

$imagen_i = apparienciaFos[SelectorLuminancia]$

end

devolver *imagen*

end función

Algoritmo 2: Conversión de la imagen en BN a fosfenos

Finalmente, debido a que el propio implante tiene fallos, se supone un 10% de fosfenos muertos: nunca se recibe la información en ese conjunto de *píxels*. Para ello, se genera un vector de números aleatorios al comienzo de la simulación, que indicarán si funcionan o no: si ese valor es menor que 0,1 ese fosfeno siempre será negro y, si es mayor, se transformará el conjunto de *píxels* a su apariencia como fosfenos. Con estos algoritmos y consideraciones se obtienen los resultados mostrados en la Figura 5.7, en la que se comparan las imágenes tomadas por las cámaras y la conversión de éstas a fosfenos. Adicionalmente, en las Figuras 5.8 y 5.9 se ven los fotogramas cuando se enfoca a una ventana y del movimiento de una mano, respectivamente.

Cabe destacar que con este proceso se pierde gran cantidad de información, por lo que una sola imagen no siempre da detalles del entorno como puede captar una persona sana, sino que se obtiene información a partir de la variación. Es decir, un solo fotograma no suele aportar información al paciente de dónde se encuentra, qué le rodea, dónde están los límites de su entorno… de manera que las personas implantadas se basan en la variación de éstos para poder reaccionar y comprender el medio.

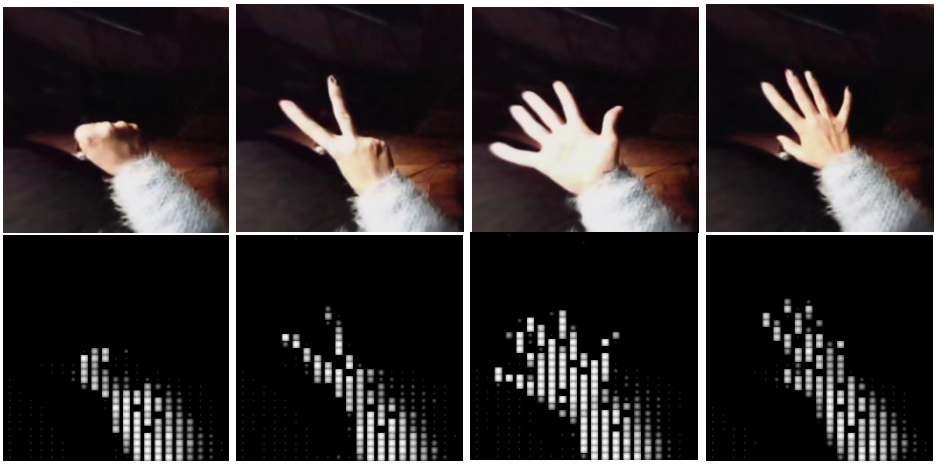


Figura 5.7: Imagen captada por las gafas y conversión a fosfenos

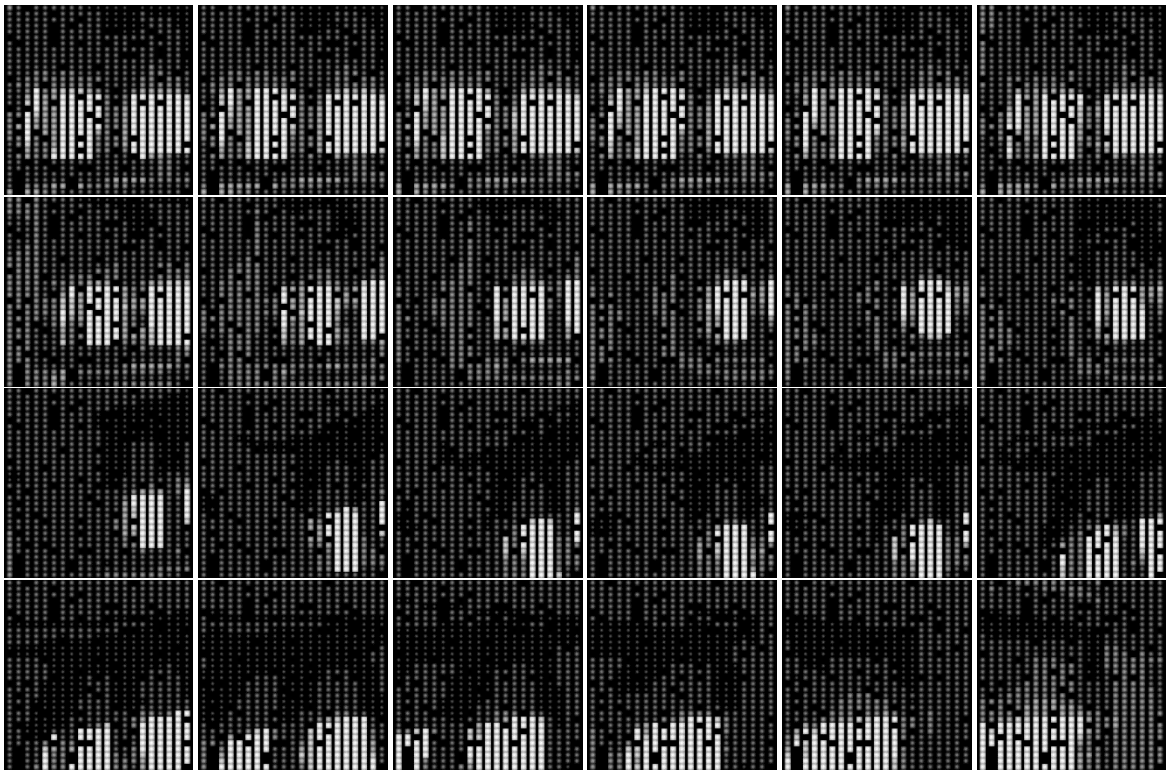


Figura 5.8: Ejemplo de la conversión a fosfenos de los fotogramas al enfocar a una ventana

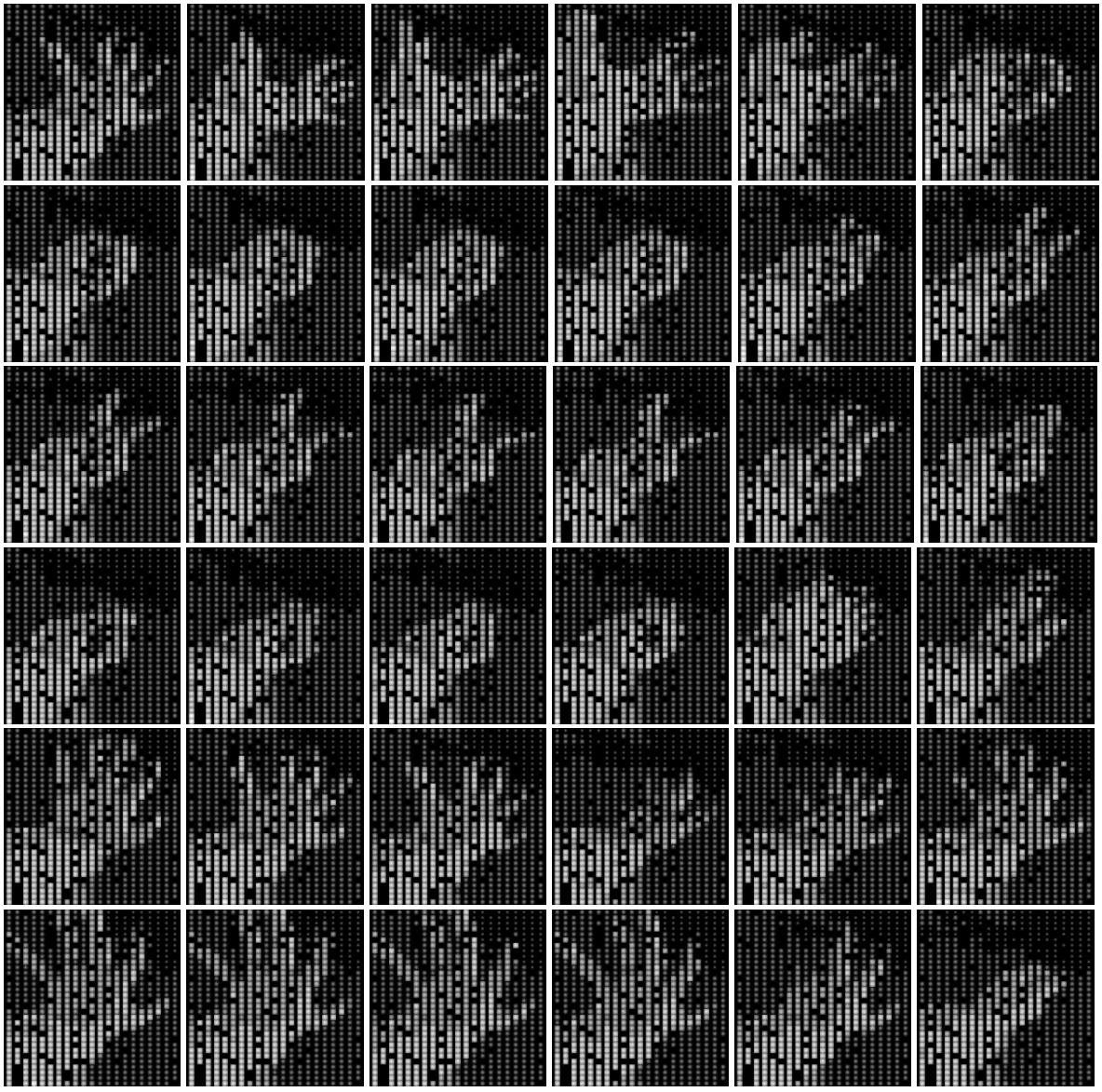


Figura 5.9*:* Ejemplo de la conversión a fosfenos de los fotogramas al enfocar a una mano

5.4 Muestra de imágenes por pantalla

La metodología de muestra de imágenes por las pantallas incorporadas en las Gafas de VR es el último aspecto para conseguir un *software* que simule las prótesis de una forma inmersiva. Para ello, se convierten las imágenes obtenidas y tratadas (del tipo de dato 'matriz de OpenCV') a 'textura de OpenGL[9]' para luego enviarla a cada pantalla -*display*- mediante capas de composición –*VRCompositor*-, de forma que, al utilizar las gafas se percibe el entorno de una manera similar a los pacientes implantados con Prótesis Visual. La información sobre la utilización de *VRCompositor* se obtuvo realizando un proceso de ingeniería inversa, debido a que no se encontraba documentado ni en OpenVR ni en las aplicaciones y ejemplos.

La forma de utilizar las capas de composición es mediante un proceso en el que se ha de convertir una imagen de tipo de dato matriz de OpenCV a textura de OpenGL, de manera que se interpolan los datos mediante un filtro y se convierte finalmente la matriz como conjunto de *píxels* en la imagen a mostrar. Posteriormente, se ha de crear un mapa MIP –*mipmap*- para que la textura principal no se muestre con todo su detalle, sino que sea capar de renderizarse más rápidamente, haciendo que no aparezcan retrasos y que el programa genere imágenes en tiempo real. Una vez obtenida la textura de OpenGL, se convierte a textura de OpenCV y se envía (*submit)* a las pantallas de las gafas mediante el VRCompositor. El esquema del proceso se muestra en la Figura 5.10:

Figura 5.10: Esquema del proceso de VRCompositor

Asimismo, mediante ensayo-error, se descubre que es necesario realizar, para que funcione correctamente el VRCompositor, una sincronización, de manera que se ha de esperar a que termine de enviar la textura a las pantallas para poder hacer otro proceso de conversión. Para ello, se han de ejecutar los comandos *glFlush* y *glFinish,* de manera que se bloquea el proceso hasta que se terminan de realizar las operaciones. Si no se utilizan estos comandos, se envían más operaciones de las que se pueden ejecutar y el programa no funciona.

[9] OpenGL (Open Graphics Library) es una interfaz de programación para gráficos 2D y 3D

Una vez realizada la inicialización de los parámetros de las cámaras y las pantallas, la captación de fotogramas y la conversión a apariencia de fosfenos y a textura, se envían las imágenes a las pantallas de las Gafas VR. En la Figura 5.11 se observa que se consigue mostrar las imágenes tanto en la pantalla del ordenador (pestaña 'IMAGEN' –ejecutada desde el propio código-) como en la pantalla del dispositivo VR (pestaña 'Vista de VR', obtenida mediante SteamVR), de manera que se demuestra que sí se obtienen las imágenes deseadas en las propias Gafas VR. Cabe destacar que, en la pestaña 'Vista de VR', se observan dos imágenes que varían mínimamente, debido a que cada ojo obtiene una imagen distinta –relativa a cada una de las cámaras-.

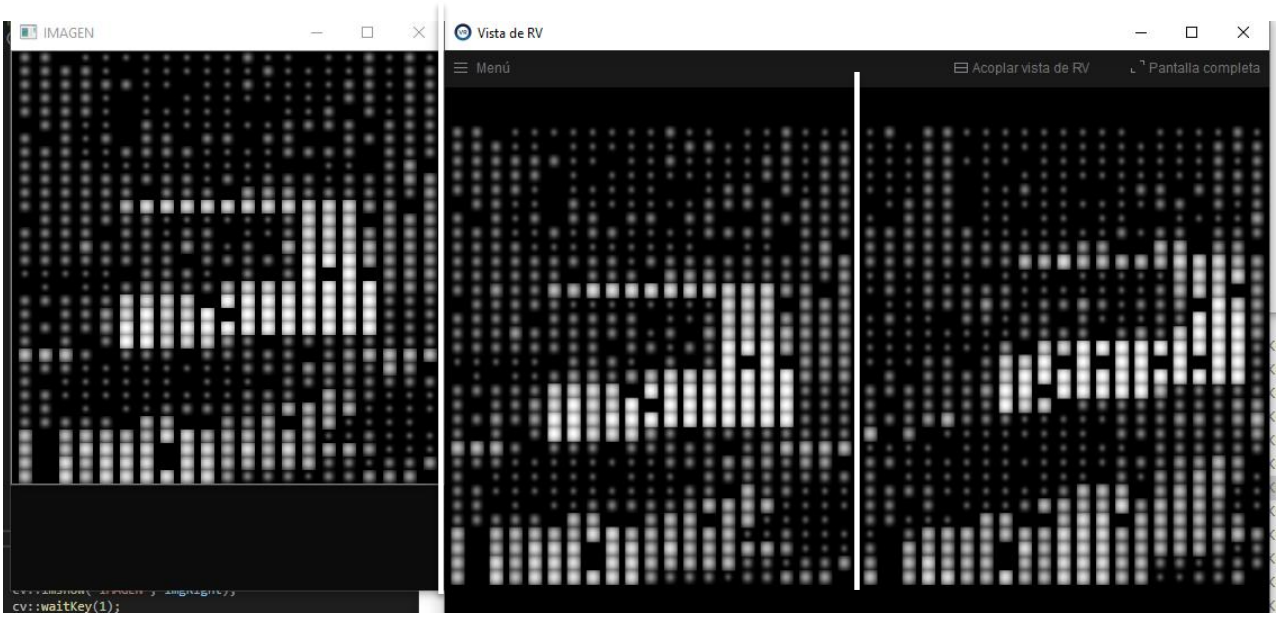


Figura 5.11: Resultado mostrado en pantalla –IMAGEN- y lo que se ve en las gafas VR –Vista de VR-.

Finalmente, se ha de considerar que la imagen en fosfenos, al ser enviada directamente a las cámaras, ocupa completamente el campo de visión de cada ojo, de manera que no se distingue con claridad la imagen: la perspectiva no es la adecuada y hay que alejarla para hacerlo realista, de manera que se procede a realizar una calibración. Para ello, se cambia el tamaño de la imagen en fosfenos sobre un fondo negro hasta conseguir que la visión de un elemento fijo –se utiliza un folio sobre una pared oscura por el contraste- con y sin el dispositivo de Realidad Virtual se encuentre prácticamente a la misma escala.

CAPÍTULO 6: CONCLUSIONES

En este capítulo se exponen las conclusiones obtenidas en el transcurso de este trabajo y se plantean las futuras líneas de investigación.

6.1 Resultados obtenidos

En el proyecto se ha llevado a cabo una investigación de métodos de desarrollo de aplicaciones ejecutables en las gafas de Realidad Virtual, utilizando distintos programas, herramientas y códigos, y haciendo un proceso de ingeniería inversa hasta llegar al modo de conseguir los objetivos establecidos.

Esta forma de conseguirlo es robusta y estable: depende de las versiones de librerías, *software* o actualizaciones, pero se encuentra, en la actualidad, más actualizada y con menor cantidad de dependencias: OpenVR es independiente del sistema operativo en el que se ejecuta y no cambia si se utiliza otro elemento de Realidad Virtual –no se enfoca en un *SDK* concreto, sino que sirve para los distintos dispositivos de Realidad Virtual-.

Además de hallar la manera de desarrollar el proyecto, se ha realizado un programa que, al ejecutarse, cumple con los objetivos de éste. Esta solución se ha llevado a cabo, como se menciona en el Capítulo 4, mediante metodología *Agile*, y más en concreto con un concepto fundamental de ésta: el Producto Viable Mínimo (MVP, *Minimum Viable Product)*, enfocado en no realizar la aplicación completa e ir solucionando los problemas, sino en desarrollar un programa mínimo (captar imágenes con las cámaras de las Gafas VR) e ir añadiendo funcionalidades por mejora continua (recortar las imágenes, convertirlas a fosfenos, enviarlas a las pantallas). Mediante este proceso se han generado distintos proyectos que se pueden encontrar anexados en la siguiente página:

https://drive.google.com/drive/folders/1spSlw69S-yqxUzJkMgYvrwd71KK8VspY?usp=sharing

en la que se encuentran los archivos C++ y los respectivos CMakeLists para poder configurar la solución. Estos archivos son, por orden de realización:

- *ImageToPhosphenes*: en el que, al ejecutarse, se introduce una imagen externa al programa y se devuelve por pantalla del ordenador su apariencia como fosfenos. Este programa no requiere de dispositivo de Realidad Virtual ni de imágenes con dimensiones concretas, sino que cualquier imagen que se introduzca se convierte a la forma en la que se percibiría con implantes.

- *CameraToPhosphenes:* en este archivo sí se necesita un dispositivo que capte imágenes, ya que se van a tomar los fotogramas de las cámaras para convertirlos en fosfenos y mostrarlos por la pantalla del ordenador. Se observa que se basa en el programa anterior y se le añade la funcionalidad de captar imágenes en tiempo real en vez de introducirlas predeterminadas. Con ello se consigue una simulación en tiempo real pero no inmersiva ya que se obtienen resultados por la pantalla del ordenador, no de las propias Gafas VR.

- *ProthesisSimulator:* ésta es la aplicación final del proyecto, ya que aúnan las funcionalidades anteriores y se añade el mostrar las imágenes por las pantallas propias del dispositivo VR, además de en la pantalla del ordenador, consiguiendo una simulación en tiempo real e inmersiva.

Dentro de los propios programas mencionados se ha utilizado este método de mejora continua, ya que, por ejemplo, al convertir a apariencia de fosfenos, primero se recortaba la imagen; una vez tenía el tamaño deseado se convertía a un canal (BN); posteriormente se generaba la apariencia de cada fosfeno; luego se convertía toda la imagen a fosfenos… Y, en el caso de captar imágenes de las cámaras, primero se conseguía mediante Qt5 y luego se utilizaba OpenCV para poder realizar el procesamiento.

Se debe resaltar que los distintos programas realizados se han ido documentando, de manera que se puede retomar el proceso en cualquier punto del desarrollo, cambiarlo y mejorarlo, para obtener optimización en el tratamiento de imagen.

6.2 Conclusiones

A partir de las imágenes mostradas a lo largo del trabajo se puede observar que se captan y se muestran los fotogramas tratados hasta convertirlos en aspecto de fosfenos en tiempo cuasi-real. De esta manera, se puede estudiar la percepción que tienen los pacientes implantados sin necesidad de que sean sujetos de prueba para posibles modificaciones, con lo que eso conlleva: dificultad para acostumbrarse, continuos cambios en la percepción, dificultad de probar con pacientes debido a que no es un implante generalizado…

Al ser una tecnología tan dinámica, la información sobre cómo realizar este proyecto era muy limitada o estaba obsoleta por falta de actualización de la información, la cual varía a una velocidad vertiginosa, de manera que la dificultad de este trabajo ha sido, principalmente, además de aprender a manejar los procesos de programación y compilación, encontrar el método para

llevarlo a cabo y realizar ingeniería inversa para conseguir acceder a los datos relevantes para este proyecto.

No obstante, finalmente se ha conseguido acceder a las cámaras y a las pantallas incorporadas y, con ello, el poder tratar y presentar las imágenes: los resultados obtenidos muestran el alcance planteado inicialmente, en el que se buscaba hacer una simulación en tiempo real y de forma inmersiva de la percepción con el implante retinal.

6.3 Líneas futuras

A partir de estos resultados, en un futuro, se buscará tratar las imágenes mediante programación, inteligencia artificial y aprendizaje automático, teniendo en cuenta aristas, profundidad o distancias, ya que al tener dos imágenes captadas desde dos puntos de vista se puede realizar la triangulación de elementos.

Es decir, al conseguir una simulación que se asemeja a los implantes y en función de parámetros propios del éste (número de fosfenos, niveles de intensidad, eficiencia del implante), se establece la base para futuras investigaciones relacionadas con el tratamiento de la imagen: el proceso intermedio entre captación y transformación a fosfenos, para mejorar la información enviada a través de los implantes y, en definitiva, la calidad de vida de los pacientes implantados.

BIBLIOGRAFÍA

[1] C. Chen, J. Suaning, Q. Morley, H.Lovell (2009) *Simulating prosthetic vision: I, Visual models of phosphenes,* Elsevier

[2] C. Chen, J. Suaning, Q. Morley, H.Lovell (2009) *Simulating prosthetic vision: II, Measuting functional capacity,* Elsevier

[3] Melani Sanchez-Garcia, Ruben Martinez-Cantin and Jose J. Guerrero (2019) *Indoor Scenes Understanding for Visual Prosthesis with Fully Convolutional Networks*. International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications

[4] Melani Sánchez-García, Rubén Martínez-Cantín and José J. Guerrero (2020) *Semantic and structural image segmentation for prosthetic vision*. PLOS One, 15(1):e0227677.

[5] Vive Developers (2020) *VIVE SRWorks SDK*. https://developer.vive.com/resources/vive-sense/sdk/vive-srworks-sdk

[6] Vive Developers (2020) *VIVE Hand Tracking SDK* https://developer.vive.com/resources/vive-sense/sdk/vive-hand-tracking-sdk

[7] Vuforia (2020). *Getting started with Vuforia in Unity*. https://library.vuforia.com/articles/Training

[8] Valve (2020). *OpenVR API*. GitHub https://github.com/ValveSoftware/openvr

[9] Windows Central (2020) *Unity drops support to OpenVR* www.windowscentral.com/unity-drops-official-support-openvr-valve-working-replacement

LISTA DE FIGURAS

ANEXOS

Anexo A: Manual de usuario

Para poder llevar a cabo este proyecto se requiere la utilización de diversos *softwares*. Para empezar, hay que descargarse de GitHub la SDK de OpenVR mediante símbolo de sistema. Para ello, ejecutar el siguiente comando en la carpeta deseada:

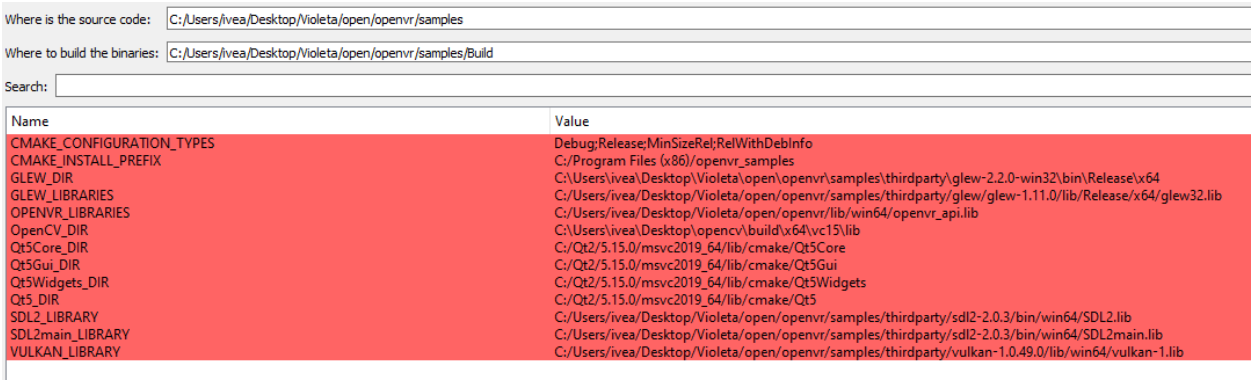
git clone https://github.com/ValveSoftware/openvr.git

Instalar CMake, un configurador de compiladores y Visual Studio 2019. Mediante CMake se van a generar los archivos para que se puedan generar ejecutables con los programas que creemos en C++. Una vez instalados CMake, VisualStudio y habiendo copiado la carpeta de OpenVR en nuestro sistema, se procede a configurar y generar el proyecto. Para ello se va a necesitar la instalación de las librerías Qt5 (versión 5.15.0) y SDL2.dll.

Una vez instalados, se configura CMake de la siguiente manera:

- Se construyen los binarios en el mismo directorio de *samples* pero en una carpeta nueva llamada *Build*
- Se completa la ruta de las librerías que faltan: bien a mano cada vez que se configure, bien cambiando el *path* (*https://www.architectryan.com/2018/03/17/add-to-the-path-on-windows-10/*)
    - Las de Qt5 se encuentran en la carpeta Qt/5.15.0/msvc2019_64/lib/cmake/Qt5 (es importante no confundirse con ninguna otra porque implicará que no se encuentren las librerías)
    - En caso de no encontrar las librerías *glew*, éstas se encuentran en la carpeta del ejemplo *thirdparty*
    - La librería *SDL2.dll* se ha de introducir en la carpeta en la que se generen los ejecutables (bin/win64)

A continuación, se muestra una captura de la apariencia del programa CMake para generar el proyecto de forma correcta:



Una vez instalado todo, se configura y se genera el proyecto para Visual Studio 2019 y sistema operativo x64. Con esto se habrán compilado los ejemplos que proporciona esta SDK. Para crear un nuevo ejemplo hay que cambiar el archivo CMakeLists.txt de la carpeta *samples* de manera que incluya la nueva carpeta del ejemplo y volver a generar la compilación con CMake.

- **Generar un nuevo ejemplo**: Se copia en la carpeta de *samples* cualquier carpeta que sirva de punto de partida y se cambia el nombre tanto de dicha carpeta como de los archivos *.cpp* y *.h* que tiene dentro. Además, en el CMakeLists.txt dentro de la carpeta *samples* se añade un Nuevo subdirectorio con el nombre de dicha carpeta.

  Dentro de la nueva carpeta se habrá copiado el CMakeLists.txt de la carpeta que servía como base, de manera que se edita este archivo cambiando los nombres de los archivos *.cpp* y *.h* en el apartado *add_executable*.

- **Editar un ejemplo**: en caso de hacer variaciones en el código, revisar que las librerías a las que se quiere acceder son las que aparecen en el CMakeLists.txt dentro de la carpeta del ejemplo, en el apartado *target_link_libraries*.

Anexo B: Programación C++

En las siguientes hojas se presentan las partes del código de programación que se ha utilizado para hacer funcionar las cámaras de las Gafas HTC Vive Pro, así como las funciones para el tratamiento y visualización de las imágenes como fosfenos. El código completo (*main.cpp)* se adjunta junto con los archivos CMakeLists.txt (el de todos los *samples* y el de *convert_phosphenes)*, necesarios para generar los archivos ejecutables en la siguiente dirección: https://drive.google.com/drive/folders/1spSlw69S-yqxUzJkMgYvrwd71KK8VspY?usp=sharing

Funciones:

- Guardado de los parámetros de la cámara

```
static bool saveCameraParams(const String& filename, Size imageSize, float
  aspectRatio, int flags,
    const Mat& cameraMatrix, const Mat& distCoeffs, double totalAvgErr) {
    FileStorage fs(filename, FileStorage::WRITE);
    if (!fs.isOpened())
        return false;

    fs << "image_width" << imageSize.width;
    fs << "image_height" << imageSize.height;

    if (flags & CALIB_FIX_ASPECT_RATIO) fs << "aspectRatio" << aspectRatio;

    fs << "camera_matrix" << cameraMatrix;
    fs << "distortion_coefficients" << distCoeffs;

    fs << "avg_reprojection_error" << totalAvgErr;

    return true;
}
```

- Generación de una función *meshgrid* que obtenga dos matrices X e Y basadas en las coordinadas contenidas en dos vectores *x* e *y*: X es una matriz en la que cada fila es una copia de x, Y es una matriz en la que cada columna es una copia de y. Ambas matrices tienen como dimensión la longitud de *x* por la longitud de *y*.

```
// MESHGRID function -> Creates a X and Y matrix
static void meshgrid(const cv::Mat& xgv, const cv::Mat& ygv,
    cv::Mat1i& X, cv::Mat1i& Y)
{
    cv::repeat(xgv.reshape(1, 1), ygv.total(), 1, X);
    cv::repeat(ygv.reshape(1, 1).t(), 1, xgv.total(), Y);
}
static void meshgridTest(const cv::Range& xgv, const cv::Range& ygv,
    cv::Mat1i& X, cv::Mat1i& Y)
{
    std::vector<int> t_x, t_y;
    for (int i = xgv.start; i <= xgv.end; i++) t_x.push_back(i);
    for (int i = ygv.start; i <= ygv.end; i++) t_y.push_back(i);
    meshgrid(cv::Mat(t_x), cv::Mat(t_y), X, Y);
}
```

- Generación las posibles apariencias de los fosfenos: se consigue una matriz de dimensiones *sizeX* x *sizeY* x *nivelesDeGris*

```
// GENERATE THE LOOK UP OF THE PHOSPHENES -> Once each photogram
cv::Mat generate_phosphenes_look_up(int size_x, int size_y, uint8_t
  gray_levels, uint8_t sigma = 7)
{
    int pos_x = size_x / 2;
    int pos_y = size_y / 2;
    cv::Mat1i x, y;
    const int sizes[] = { size_x, size_y, gray_levels };
    cv::Mat phosphenes(3, sizes, CV_8UC1);
    meshgridTest(cv::Range(1, size_x), cv::Range(1, size_y), x, y);
    cv::Mat1i localDist2(x.cols, x.rows, CV_8UC1);

    for (uint32_t i = 0; i < x.cols; i++) {
        for (uint32_t j = 0; j < x.rows; j++) {
            localDist2[i][j] = pow((x[i][j] - pos_x), 2) + pow((y[i][j] -
              pos_y), 2);
        }
    }
```

```
    for (uint32_t i = 0; i < x.cols; i++) {
        for (uint32_t j = 0; j < x.rows; j++) {
            for (uint32_t k = 0; k < gray_levels; k++) {
                phosphenes.at<uchar>(i,j,k) = 255*((exp(-localDist2[i][j] / (2
                   * pow((sigma * (k + 1) / gray_levels),2)))) * (k + 1) /
                  gray_levels);
            }
        }
    }
     return phosphenes;
}
```

- Creación de fosfenos muertos: calcula un vector de 1024 aleatorios al comienzo de la ejecución para evaluar los que no perciben ninguna señal

```
// RANDOM OF DEAD PHOSPHENES -> Creates an array: if < 0.1 then phosphene is
  dead
cv::Mat1i aleatorios(uint8_t ph_x, uint8_t ph_y) {
    cv::Mat1i ran(ph_x, ph_y);
    for (uint8_t a = 0; a < ph_x; a++) {
        for (uint8_t b = 0; b < ph_y; b++) {
            ran[a][b] = rand() % 100;
        }
    }
    return ran;
}
```

- Conversión a una imagen en fosfenos: en función de los parámetros de número de fosfenos del implante y niveles de gris, devuelve una imagen de una capa -en blanco y negro- convertida en fosfenos

```
//CONVERT INTO PHOSPHENES -> Introducing image ant parameters turns back that
  image into phosphenes
cv::Mat convert_phosphenes(cv::Mat img_array, uint8_t gray_levels = 8, uint8_t
   n_phosphenes_x = 32, uint8_t n_phosphenes_y = 32, bool use_median = TRUE,
  bool use_hex_pattern = TRUE, double dropout = 0.1, cv::Mat1i vect_aleat =
  cv::Mat1i::zeros(32, 32))
{
    uint32_t width = img_array.size[0];
    uint32_t height = img_array.size[1];
    uint32_t stepx = width / n_phosphenes_x;
    uint32_t stepy = height / n_phosphenes_y;
    bool even_row = TRUE;
    uint32_t end_i, start_i, end_j, start_j, offset;
    float luminance;
    cv::Mat img_phosphenes(height, width, CV_8UC1);

    cv::Mat phosphenes_look_up = generate_phosphenes_look_up(stepx, stepy,
   gray_levels);

  // Each phosphene processing
  for (start_i = 0; start_i < width + stepx; start_i += stepx) {
      even_row = ~even_row;
      end_i = start_i + stepx;
      if (end_i > width)
          end_i = width;
      for (start_j = 0; start_j < height + stepy; start_j += stepy) {
          if (use_hex_pattern && even_row)
              offset = stepy / 2;
          else  offset = 0;
          start_j += offset;
          end_j = start_j + stepy;
          if (end_j > height) { end_j = height; }

            // MEDIAN OR MEAN
            if ((end_i - start_i > 0) && (end_j - start_j > 0)) {
                // Using median
                if (use_median) {   //Create a vector of values of the section
                   of the image (each phosphene) and rank it from bottom to
                  top: in the middle it's the median value
                    // Create the vector
                    cv::Mat1i vector_ordenado = cv::Mat1i::zeros(stepx*stepy,
                    1);
                    uint32_t n = 0;
```

```
                    for (uint32_t a = start_i; a < end_i; a++) {
                        for (uint32_t b = start_j; b < end_j; b++) {
                            vector_ordenado[n][0] = img_array.at<uchar>(a, b);
                            n++;
                        }
                    }
                    // Rank bot-top
                    uint32_t temporal;
                    for (uint32_t i = 0; i < stepx * stepy; i++) {
                        for (uint32_t j = 0; j < stepx * stepy - 1; j++) {
                            if (vector_ordenado[j][0] > vector_ordenado[j + 1]
                    [0]) {
                                temporal = vector_ordenado[j][0];
                                vector_ordenado[j][0] = vector_ordenado[j + 1]
                    [0];
                                vector_ordenado[j + 1][0] = temporal;
                            }
                        }
                    }
                    luminance = vector_ordenado[stepx*stepy / 2][0];
                }
                else {
                    int sum = 0;
                    for (uint32_t a = start_i; a < end_i; a++) {
                        for (uint32_t b = start_j; b < end_j; b++) {
                            sum += img_array.at<uchar>(a, b);
                        }
                    }
                    luminance = sum / ((end_i - start_i) * (end_j - start_j));
                }

                uchar luminance_selector = luminance * gray_levels / 255;

                // Phosphenes dead because of the implant (always the same)
                if (vect_aleat[start_i/stepx][start_j/stepy] > dropout*100) {
                    for (uint32_t a = start_i; a < end_i; a++) {
                        for (uint32_t b = start_j; b < end_j; b++) {
                            img_phosphenes.at<uchar>(a, b) =
                    phosphenes_look_up.at<uchar>(a - start_i, b - start_j,
                    luminance_selector);
                        }
                    }
                }
            }
        }
    }
    return img_phosphenes;
}
```

En la ejecución principal se llevan a cabo los procesos siguientes:

- Inicialización de las Gafas VR

```
bool CMainApplication::Initialitation() {
    std::cout << "\nStarting OpenVR...\n";

    m_pVRTrackedCamera = vr::VRTrackedCamera();
    if (!m_pVRTrackedCamera)
    {
        std::cout << "Unable to get Tracked Camera interface.\n";
        return 1;
    }

    bool bHasCamera = false;
    vr::EVRTrackedCameraError nCameraError = m_pVRTrackedCamera->HasCamera
      (vr::k_unTrackedDeviceIndex_Hmd, &bHasCamera);
    if (nCameraError != vr::VRTrackedCameraError_None || !bHasCamera)
    {
        std::cout << "No Tracked Camera Available! " << m_pVRTrackedCamera-
          >GetCameraErrorNameFromEnum(nCameraError) << "\n";
        return 1;
    }

    // Accessing the FW description is just a further check to ensure camera
      communication is valid as expected.
    vr::ETrackedPropertyError propertyError;
    char buffer[128];
    m_pHMD->GetStringTrackedDeviceProperty(vr::k_unTrackedDeviceIndex_Hmd,
      vr::Prop_CameraFirmwareDescription_String, buffer, sizeof(buffer),
      &propertyError);
    if (propertyError != vr::TrackedProp_Success)
    {
        std::cout << "Failed to get tracked camera firmware description!\n";
        return 1;
    }
```

```
        nCameraError = m_pVRTrackedCamera->GetCameraFrameSize
          (vr::k_unTrackedDeviceIndex_Hmd,
          vr::VRTrackedCameraFrameType_Undistorted, &m_nCameraFrameWidth,
          &m_nCameraFrameHeight, &m_nCameraFrameBufferSize);
        if (nCameraError != vr::VRTrackedCameraError_None)
        {
            std::cout << "GetCameraFrameBounds() Failed!\n";
            std::cout << m_pVRTrackedCamera->GetCameraErrorNameFromEnum
              (nCameraError) << "\n";
            return 1;
        }

        image.create(m_nCameraFrameHeight, m_nCameraFrameWidth, CV_8UC4);

        m_pVRTrackedCamera->AcquireVideoStreamingService
          (vr::k_unTrackedDeviceIndex_Hmd, &m_hTrackedCamera);
        if (m_hTrackedCamera == INVALID_TRACKED_CAMERA_HANDLE)
        {
            std::cout << "AcquireVideoStreamingService() Failed!\n";
            return 1;
        }

        m_pHMD->GetRecommendedRenderTargetSize(&pnWidth, &pnHeigth);
    }
```

- Inicialización de GL

```
bool CMainApplication::BInitGL()
{
    if( m_bDebugOpenGL )
    {
        glDebugMessageCallback( (GLDEBUGPROC)DebugCallback, nullptr);
        glDebugMessageControl( GL_DONT_CARE, GL_DONT_CARE, GL_DONT_CARE, 0,
          nullptr, GL_TRUE );
        glEnable(GL_DEBUG_OUTPUT_SYNCHRONOUS);
    }

    if( !CreateAllShaders() )
        return false;

    return true;
}
```

- Inicialización de VRCompositor

```
bool CMainApplication::BInitCompositor()
{
    vr::EVRInitError peError = vr::VRInitError_None;

    if ( !vr::VRCompositor() )
    {
        printf( "Compositor initialization failed. See log file for details
          \n" );
        return false;
    }

    return true;
}
```

- Toma de imágenes por las cámaras
- Recorte de la imagen de la cámara derecha e izquierda sin distorsión

```
//Cropping image from (1840,1224) to (384,384) -> Right camera
int tpxR = 300;
int tpyR = 235;
int anchoR = 12 * 32;
int altoR = 12 * 32;
cv::Rect myROI_R(tpxR, tpyR, anchoR, altoR);
imgcopiaRight = imgcopia(myROI_R);

//Cropping image from (1840,1224) to (384,384) -> Left camera
int tpxL = 300;
int tpyL = 1200;
int anchoL = 12 * 32;
int altoL = 12 * 32;
cv::Rect myROI_L(tpxL, tpyL, anchoL, altoL);
imgcopiaLeft = imgcopia(myROI_L);
```

- Conversión a un canal (BN)

```
for (uint32_t x = 0; x < altoR; x++)
{
    for (uint32_t y = 0; y < anchoR; y++)
    {
        cv::Vec3b intensity = imgcopiaRight.at<Vec3b>(x, y);
        uint8_t promedio = (intensity.val[0] + intensity.val[1] +
          intensity.val[2]) / 3;
        intensity.val[0] = promedio;
        intensity.val[1] = promedio;
        intensity.val[2] = promedio;
        imgcopiaRight.at<Vec3b>(x, y) = intensity;
        capa_img.at<uchar>(x, y) = promedio;
    }
}
```

- Conversión a fosfenos

- Generación de texturas

```
static GLuint matToTexture(const cv::Mat& mat, GLenum minFilter, GLenum
  magFilter, GLenum wrapFilter) {
    // Generate a number for our textureID's unique handle
    GLuint textureID;
    glGenTextures(1, &textureID);

    // Bind to our texture handle
    glBindTexture(GL_TEXTURE_2D, textureID);


    // Catch silly-mistake texture interpolation method for magnification
    if (magFilter == GL_LINEAR_MIPMAP_LINEAR ||
        magFilter == GL_LINEAR_MIPMAP_NEAREST ||
        magFilter == GL_NEAREST_MIPMAP_LINEAR ||
        magFilter == GL_NEAREST_MIPMAP_NEAREST)
    {
        std::cout << "You can't use MIPMAPs for magnification - setting
          filter to GL_LINEAR" << std::endl;
        magFilter = GL_LINEAR;
    }

    // Set texture interpolation methods for minification and magnification
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, minFilter);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, magFilter);

    // Set texture clamping method
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, wrapFilter);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, wrapFilter);

    // Set incoming texture format to:
    // GL_BGR       for CV_CAP_OPENNI_BGR_IMAGE,
    // GL_LUMINANCE for CV_CAP_OPENNI_DISPARITY_MAP,
    // Work out other mappings as required ( there's a list in comments in
      main() )
    GLenum inputColourFormat = GL_BGR;
    if (mat.channels() == 1)
    {
        inputColourFormat = GL_LUMINANCE;
    }
```

```
    glTexImage2D(GL_TEXTURE_2D,     // Type of texture
        0,                  // Pyramid level (for mip-mapping) - 0 is the top
          level
        GL_RGBA8,             // Internal colour format to convert to
        mat.cols,           // Image width  i.e. 640 for Kinect in standard
          mode
        mat.rows,           // Image height i.e. 480 for Kinect in standard
          mode
        0,                  // Border width in pixels (can either be 1 or 0)
        inputColourFormat, // Input image format (i.e. GL_RGB, GL_RGBA,
          GL_BGR etc.)
        GL_UNSIGNED_BYTE,  // Image data type
        mat.ptr());         // The actual image data itself

// If we're using mipmaps then generate them. Note: This requires OpenGL 3.0
  or higher
    if (minFilter == GL_LINEAR_MIPMAP_LINEAR ||
        minFilter == GL_LINEAR_MIPMAP_NEAREST ||
        minFilter == GL_NEAREST_MIPMAP_LINEAR ||
        minFilter == GL_NEAREST_MIPMAP_NEAREST)
    {
        glGenerateMipmap(GL_TEXTURE_2D);
    }
    return textureID;
}
```

- Muestra por pantalla

```
void CMainApplication::RenderFrame()
{
    if ( m_pHMD )
    {
        ImageIntoPhosohenes();

        cv::imwrite("phosRight.png", fosfenosRight);
        std::string image_path_Right = samples::findFile("phosRight.png");
        cv::imwrite("phosLeft.png", fosfenosLeft);
        std::string image_path_Left = samples::findFile("phosLeft.png");

        cv::Mat imgRight = cv::imread(image_path_Right);
        if (imgRight.empty()) {
            std::cout << "Cannot load image: " << std::endl;
            exit(EXIT_FAILURE);
        }
        cv::Mat imgLeft = cv::imread(image_path_Left);
        if (imgLeft.empty()) {
            std::cout << "Cannot load image: " << std::endl;
            exit(EXIT_FAILURE);
        }

        // Images have to be rotate horizontally because it appears a mirror
          effect
        cv::Mat imgRightOK;
```

```
        cv::flip(imgRight, imgRightOK, 0);
        cv::Mat imgLeftOK;
        cv::flip(imgLeft, imgLeftOK, 0);

        GLuint image_tex_Left = matToTexture(imgRightOK,
          GL_LINEAR_MIPMAP_LINEAR, GL_LINEAR, GL_CLAMP);
        vr::Texture_t leftEyeTexture = { (void*)(uintptr_t)image_tex_Left,
          vr::TextureType_OpenGL, vr::ColorSpace_Gamma };
        vr::VRCompositor()->Submit(vr::Eye_Left, &leftEyeTexture );
        GLuint image_tex_Right = matToTexture(imgLeftOK,
          GL_LINEAR_MIPMAP_LINEAR, GL_LINEAR, GL_CLAMP);
        vr::Texture_t rightEyeTexture = { (void*)(uintptr_t)image_tex_Right,
          vr::TextureType_OpenGL, vr::ColorSpace_Gamma };
        vr::VRCompositor()->Submit(vr::Eye_Right, &rightEyeTexture );
        cv::imshow("IMAGEN", imgRight);
        cv::waitKey(1);
    }
```