



## Trabajo Fin de Grado

Desarrollo de una herramienta de recogida y visualización de datos relativos a la exploración de contenido en realidad virtual

Development of a data collection and visualization tool related to content exploration in virtual reality

Autor/es

Pedro José Pérez García

Director/es

Sandra Malpica Mallo

PONENTE

Ana Serrano Pacheu

Escuela de Ingeniería y Arquitectura

2021-2022

# Resumen

---

La realidad virtual (RV) es una tecnología en auge, que recientemente ha entrado al mercado a nivel de consumidor, y es accesible para la mayoría de la población. La RV tiene el potencial de cambiar el modo en que consumimos contenido, y su aplicabilidad se extiende a áreas tan diversas como el entretenimiento, la medicina, la arquitectura, el turismo, o el diseño de interiores. No obstante, todavía queda mucho por aprender sobre cómo las personas usamos la RV. Para obtener este conocimiento, es una práctica común recolectar datos de múltiples usuarios realizando diversas tareas en entornos virtuales. Esta tardea, no obstante, suele ser costosa y compleja, tanto en términos de hardware como de software.

Por este motivo, en este trabajo se ha abordado la implementación de una herramienta de recogida de datos de visualización de contenido dinámico en 360 grados en un entorno de realidad virtual. Particularmente, se ha enfocado el trabajo a la reproducción de vídeos 360 acompañados de audio ambisónico o espacializado. Este tipo de contenido no sólo permite al usuario observar una escena en todo el entorno que le rodea, sino que además ofrece sonidos direccionales que aumenten su realismo.

Esta herramienta se centra en la recogida de los datos relativos a la exploración visual del entorno, permitiendo obtener información acerca de la orientación de la cabeza y de la mirada durante toda la exploración, para su posterior procesado y obtención de métricas sobre el comportamiento humano, como un análisis topográfico de las regiones más interesantes de la escena, o la congruencia entre diversos usuarios.

En definitiva, el objetivo final de este trabajo es obtener una herramienta que permita tanto la recogida como el procesado de este tipo de información, que sea genérica y reutilizable en un amplio espectro de experimentos que impliquen estudios de usuario en entornos de RV.

Para ello, el trabajo se ha dividido en diferentes fases. En una primera fase, se estudió el entorno necesario para el proyecto. Seguidamente, se desarrolló, iterativamente, la herramienta de captura y análisis de datos. Finalmente, se llevó a cabo un pequeño estudio con usuarios reales para validar la aplicabilidad del sistema.



# Agradecimientos

---

A mis padres, Pedro y Sara, por darme el apoyo necesario durante estos últimos años, y por todo el sacrificio y el esfuerzo que lleváis haciendo más de 22 años para darme las oportunidades que no tuvisteis, espero poder devolvéroslo todo algún día.

A mis hermanos, Gonzalo y Diego, aunque a veces seáis insoportables, sois una parte enorme de quién soy. Porque me habéis acompañado literalmente desde el primer día, por todos los momentos que hemos hecho el tonto, reido y reñido juntos, gracias por estar ahí.

A Sandra, Dani y Ana; mi directora, mi *casi-co-director* y mi ponente para este TFG, por permitirme hacer este trabajo y por vuestra ayuda y guía durante el proceso, especialmente las últimas semanas, sé que ha sido un poco caótico.

A Alba, Andrés, Fernando y Daniel (Martincho para los amigos), por ser mis compañeros y amigos durante estos años de carrera, por las tardes en la biblioteca, los cafés y los nervios de antes de un examen, habéis hecho que estos años fueran más fáciles y llevaderos.

Por último, a todos los que no cabéis en las líneas de arriba ya seáis familia o amigos, porque me habéis ayudado a ser la persona que hoy escribe esto y porque como dicen en inglés, *it takes a village*, y yo no podría haber llegado aquí solo.

Gracias.



# Índice

---

<b>1. Introducción</b>	<b>13</b>
1.1. Contexto del proyecto . . . . .	13
1.2. Objetivos y alcance del proyecto . . . . .	14
1.3. Planificación y herramientas . . . . .	15
1.3.1. Planificación . . . . .	15
1.3.2. Herramientas utilizadas para el desarrollo . . . . .	16
<b>2. Estado del arte</b>	<b>19</b>
<b>3. Sistema implementado</b>	<b>21</b>
3.1. Primera funcionalidad del sistema: Cargar y reproducir clips de vídeo y audio	21
3.1.1. Componente VideoController . . . . .	21
3.2. Recogida de datos . . . . .	28
3.2.1. Componente HeadDataLogger . . . . .	28
3.2.2. Componente EyeDataLogger . . . . .	34
3.3. Recreación de <i>scanpaths</i> a partir de logs . . . . .	36
3.3.1. Componente ScanpathReplayer . . . . .	36
3.4. Generación de mapas de saliencia . . . . .	43
3.4.1. Componente SaliencyMapsGenerator . . . . .	43
<b>4. Validación experimental</b>	<b>52</b>
4.1. Experimento . . . . .	52
4.1.1. Funcionamiento . . . . .	52
4.1.2. Vídeos mostrados . . . . .	53
4.2. Resultados obtenidos . . . . .	58
<b>5. Conclusiones</b>	<b>63</b>
5.1. Conclusiones personales . . . . .	63
5.2. Trabajo futuro . . . . .	64
<b>Bibliografía</b>	<b>65</b>
<b>A. El motor Unity</b>	<b>68</b>
A.1. El editor de Unity . . . . .	69
A.2. Coordenadas tridimensionales de Unity . . . . .	71
A.3. Funciones y métodos de interés de Unity . . . . .	72
A.4. Esperas en corrutinas de Unity . . . . .	73

<b>B. Trabajo previo: Familiarización con Unity, vídeo 360 y audio ambisónico</b>	<b>74</b>
B.1. Reproduciendo vídeo 360 en Unity . . . . .	74
B.2. Audio ambisónico en Unity . . . . .	75
<b>C. Logs y archivos generados</b>	<b>78</b>
C.1. Ficheros con datos de orientación de la cabeza . . . . .	78
C.2. Ficheros con datos de orientación de la mirada . . . . .	79
C.3. Cacheando los logs en <i>ScanpathReplayer</i> y <i>SaliencyMapsGenerator</i> . . . . .	81
C.4. Archivos y ficheros generados por <i>SaliencyMapsGenerator</i> . . . . .	82
<b>D. Enunciado práctica 1 VR del <i>Master in Robotics, Graphics and Computer Vision</i></b>	<b>84</b>

# Índice de tablas

---

- 1.1. Tabla de tareas y sus IDs correspondientes con los plazos ilustrados en el diagrama de Gantt de la Figura 1.2. . . . . 16

# Índice de figuras

---

1.1. Crecimiento estimado para el sector de la realidad virtual en Estados Unidos, 2021 - 2028 . . . . .	14
1.2. Diagrama de Gantt con las tareas identificadas por su ID según la Tabla 1.1. . . . .	16
1.3. Imagen de unas gafas de realidad virtual HTC Vive Pro . . . . .	17
1.4. Imagen del dispositivo de <i>eye tracking</i> , integrado en las gafas de realidad virtual . . . . .	17
 3.1. Componente VideoController y sus propiedades, visto en el editor de Unity. . . . .	22
3.2. Pérdida de calidad de imagen al no redimensionar la <i>RenderTexture</i> correctamente. . . . .	25
3.3. Ejemplo del <i>raycast</i> (vector verde) ejecutado desde la cámara (FOV en blanco). . . . .	27
3.4. Jerarquía requerida por <i>OpenXR</i> para la cámara. . . . .	27
3.5. Diagrama de secuencia con el funcionamiento del sistema de <i>logging</i> simplificado. . . . .	29
3.6. Propiedades configurables del componente <i>HeadDataLogger</i> en su primera versión. . . . .	32
3.7. Propiedades en el editor de Unity de la segunda versión del componente <i>HeadDataLogger</i> . . . . .	34
3.8. Vista del componente <i>EyeDataLogger</i> desde el editor de Unity, con sus propiedades editables. . . . .	35
3.9. Opciones configurables que ofrece el componente <i>ScanpathReplayer</i> en el editor de Unity. . . . .	37
3.10. Formato del nombre de los ficheros con la lista de nombres de los vídeos y audios. . . . .	38

---

3.11. Correspondencia entre IDs del título de los <i>logs</i> y el orden de los vídeos de <i>VideoController</i> , validado con el contenido del fichero de texto con la lista de títulos. . . . .	38
3.12. Captura de una repetición de un <i>scanpath</i> . La confianza de la medición del <i>eye tracker</i> es alta. . . . .	42
3.13. Captura de una repetición de un <i>scanpath</i> . La confianza de la medición del <i>eye tracker</i> es baja. . . . .	42
3.14. Propiedades del componente <i>SaliencyMapsGenerator</i> vistas en el editor de Unity. . . . .	44
3.15. Funcionamiento del algoritmo IVT para detección de fijaciones, definido según Salvucci y Goldberg [1]. . . . .	46
3.16. Convolución con una Gaussiana que tiene en cuenta la distorsión en los polos [2]. . . . .	50
3.17. Mapa de fijaciones generado con <i>SaliencyMapsGenerator</i> para un único <i>frame</i> . . . . .	50
3.18. Mapa de saliencia generado con <i>SaliencyMapsGenerator</i> para un único <i>frame</i> . . . . .	51
4.1. Vídeo <i>Berlin VR360 video in 4K</i> . . . . .	53
4.2. Vídeo <i>Kodak 360 4K Video 360 Campus Tour</i> . . . . .	54
4.3. Vídeo <i>Teaser drone VR video 360 dronimages</i> . . . . .	54
4.4. Vídeo <i>VR 360 VR_ZenRockShore</i> . . . . .	55
4.5. Vídeo <i>QueAhoraSi_FLIP</i> . . . . .	55
4.6. Vídeo <i>Daniel Ricciardo's Record Pole Lap (360 Video) 2018 Monaco Grand Prix</i> . . . . .	56
4.7. Vídeo <i>Rhino 4K VR 360 Video</i> . . . . .	56
4.8. Vídeo <i>360deg RAIK Vekoma Shuttle Roller Coaster Phantasialand 360</i> . . . . .	57
4.9. Vídeo <i>360° DRONE VIDEO_4k</i> . . . . .	57
4.10. Mapas de fijaciones correspondiente a los <i>frames</i> 504 a 508 del vídeo <i>Teaser drone VR video 360 dronimages</i> , utilizando datos de <i>eye tracking</i> para la elaboración del mapa. . . . .	60
4.11. Frame 508 del vídeo <i>Teaser drone VR video 360 dronimages</i> con las fijaciones de las Figuras 4.10a, 4.10b, 4.10c, 4.10d y 4.10e superpuestas. . . . .	61
4.12. Mapas de fijaciones y de saliencia del <i>frame</i> 170 del vídeo <i>Teaser drone VR video 360 dronimages</i> . Se han calculado utilizando datos de orientación de la cabeza. Se ha resaltado la apariencia de las zonas salientes aumentando la exposición con el fin de mejorar su visibilidad. . . . .	61

4.13. Fijaciones de la Figura 4.12a superpuestas al <i>frame</i> correspondiente del vídeo, que contiene texto en la zona donde se concentran las fijaciones. . . . .	62
A.1. Logotipo de Unity desde 2021 . . . . .	68
A.2. Distribución de la interfaz del editor de Unity con sus elementos más importantes. . . . .	69
A.3. Pestaña de organización de un proyecto en el editor de Unity. . . . .	69
A.4. Inspector de Unity, mostrando los componentes <i>Transform</i> y un <i>script</i> pertenecientes a un objeto junto a sus propiedades. . . . .	71
A.5. Sistemas de coordenadas de varios motores y herramientas tridimensionales populares. . . . .	71
A.6. Sistema de coordenadas de Unity representado mediante los ejes X, Y y Z.	72
A.7. Ejemplo de una corutina en Unity, con imprecisiones temporales derivadas de otros cálculos en el <i>frame</i> (en rojo), y de que no se ejecutan en el preciso instante en el que la espera termina (en naranja). . . . .	73
B.1. Configuración básica para reproducir vídeos 360º en formato equirrectangular en Unity . . . . .	75
B.2. Distorsión en una imagen 360 al mapearse sobre una esfera con pocos triángulos en su malla poligonal. . . . .	76
B.3. Configuración básica para reproducir audio ambisónico en Unity. En algunas guías se explica que hay que activar la opción <i>Spatialize</i> , pero es únicamente necesario para el audio normal, con audio ambisónico no hay que activarla, ni es necesario tocar el <i>slider</i> de la opción <i>SpatialBlend</i> . . . . .	77
B.4. Opciones para importar un clip de audio ambisónico a Unity. . . . .	77
C.1. Aspecto de los campos de un <i>log</i> al abrirlo con Excel. . . . .	79
C.2. Formato del nombre de los <i>logs</i> de datos de orientación de la cabeza. . . . .	79
C.3. Aspecto de un <i>log</i> de datos de la mirada al abrirlo con Excel. . . . .	80
C.4. Formato del nombre de los <i>logs</i> de datos de dirección de la mirada . . . . .	81
C.5. Proceso de generación de cachés con los datos almacenados en los ficheros de <i>logging</i> . . . . .	82
C.6. Formato del nombre de las carpetas que contienen mapas de saliencia. . . . .	82
C.7. Sistema de carpetas creado dentro de <i>Outputs</i> para el procesado de <i>logs</i> de un vídeo. . . . .	83

C.8. Formato del nombre de los mapas de fijaciones o saliencia para cada <i>frame</i> de vídeo. . . . .	83
C.9. Tamaño de una fijación en los mapas generados. Ambas cubren un grado de ángulo visual. A la izquierda, resolución de 384x192. A la derecha, 3840x1920. .	83

# Glosario de términos y acrónimos

---

fps	<i>Frames per second</i> (Fotogramas por segundo).
frame	Fotograma. También hace referencia a un ciclo de ejecución del motor.
framerate	Tasa de fotogramas por segundo a la que se reproduce un vídeo.
HMD	<i>Head Mounted Display</i> (Casco de realidad virtual).
VR	<i>Virtual Reality</i> (Realidad virtual).
RV	Realidad virtual
<i>eye tracking</i>	Técnica de seguimiento de la mirada basada en monitorizar la orientación de las pupilas.
<i>debug</i>	Depuración.
<i>shader</i>	Programa con instrucciones que se ejecuta en una tarjeta gráfica.
4K	Resolución de vídeo de 3840 píxeles de ancho y 2160 de alto.
<i>logging</i>	Proceso de recogida de información y posterior volcado a un fichero o terminal de mensajes.
<i>log</i>	Fichero resultante de un proceso de <i>logging</i> .
<i>raycast</i>	Operación que permite comprobar si hay algún objeto al seguir una dirección durante una distancia determinada, partiendo de un punto concreto.
FOV	<i>Field of view</i> (Campo visual ángulo de visión, vertical u horizontal)
<i>scanpath</i>	Recorrido seguido por la mirada de una persona al observar un contenido durante un periodo de tiempo determinado.
IOC	( <i>Inter Observer Congruency</i> ) Medida que representa la dispersión de la mirada entre diferentes usuarios que observan la misma imagen. Definido en Bruckert, Lam, Christie y le Meur [3], ver <i>Abstract</i> .

# 1. Introducción

---

## 1.1. Contexto del proyecto

La realidad virtual nace en la década de 1950, como un proyecto con fines militares [4]. Desde entonces, se ha perfeccionado y refinado, siempre buscando aumentar la sensación de inmersión de los usuarios en el entorno, entendiendo como inmersión la sensación de realismo percibida por parte de los usuarios que hacen uso de esta tecnología. Con el paso de los años, la RV fue evolucionando y mejorando. Durante este proceso la informática se convirtió en una gran aliada, ya que a la hora de simular entornos se comenzaron a utilizar gráficos generados en tiempo real, que en ocasiones se complementarían con físicas simuladas ante las acciones del usuario sobre el entorno. También se apoya en la informática a la hora de gestionar otros estímulos sensoriales, como los sonoros, para los cuales se tiene el audio espacializado o ambisónico.

En los últimos años, gracias al avance del *hardware* se han podido alcanzar unas cotas de potencia suficientes como para poder gestionar en tiempo real un entorno simulado a una resolución que resulte creíble al ojo, alcanzando unos niveles de inmersión por parte de los usuarios muy elevados. Esto ha popularizado el contenido en realidad virtual enormemente en muy poco tiempo, sobre todo en la industria del entretenimiento [5], donde los videojuegos y el metraje en 360 grados (a veces 180) permiten aprovechar la mayoría de las capacidades del medio. De hecho, el valor actual del mercado de la realidad virtual se estima en 4420 millones de dólares en 2020, y se espera una tasa de crecimiento anual compuesto del 44 % hasta 2028, donde habrá multiplicado su valor hasta 84090 millones [6], un crecimiento que pocos sectores van a experimentar, como se puede ver en la Figura 1.1.

Además, la realidad virtual se utiliza en otros campos más allá del entretenimiento, cumpliendo un papel fundamental, como el entrenamiento militar o médico, o la educación, donde su uso permite reducir enormemente los costes y los riesgos al no tener que utilizar equipamientos (o pacientes) reales sino virtuales.

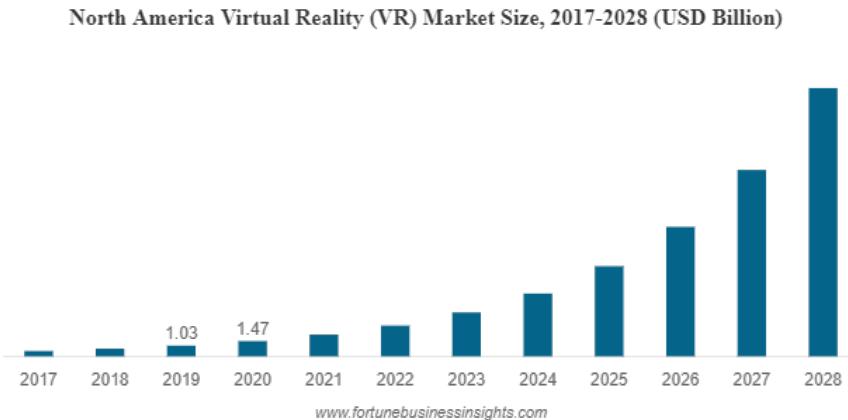


Figura 1.1: Crecimiento estimado para el sector de la realidad virtual en Estados Unidos, 2021 - 2028

En la actualidad nos encontramos en un momento clave para el sector, debido a su crecimiento y expansión, por lo que cuánto conozcamos acerca del contenido en 360 grados será crucial para el desarrollo futuro del sector. El contenido en 360 grados para RV aún tiene mucho camino por recorrer hasta alcanzar la madurez, por lo que se puede aprender mucho acerca del lenguaje visual, la forma de narrar e hilar historias mediante cortes [7], o la forma de distribuir las escenas. Por ejemplo, en el contenido audiovisual tradicional la cámara determina lo que ven los usuarios mediante diferentes tipos de planos, encuadres, composiciones y hasta colores. Un entorno en RV da a los usuarios libertad para focalizar su atención donde deseen, lo cual puede ser bueno o malo si no se sabe explotar. Entender dónde se centra la atención en un entorno que se puede explorar libremente permitirá explotar mejor las posibilidades narrativas y comunicativas del medio.

## 1.2. Objetivos y alcance del proyecto

Es en este contexto presentado donde se enmarca el Trabajo Final de Grado realizado, puesto que se confecciona una herramienta que registra el comportamiento de los usuarios al examinar escenas en RV combinadas con audio ambisónico (el audio ambisónico es aquel donde el sonido se almacena en canales diferentes permitiendo simular la espacialización de este) permitiendo extraer una serie de métricas y datos que ayuden a entender el comportamiento de los usuarios en diferentes contextos, para el posterior uso de esos datos en cualquier contexto que se deseé, como la realización de más estudios e investigación.

Con todo lo anterior, el alcance de este proyecto incluye los siguientes objetivos:

- Estudio del estado del arte en realidad virtual (Sección 2)
- Aprendizaje relacionado con las herramientas de desarrollo elegidas (Anexo B).
- Desarrollo de la herramienta de recogida de datos (Sección 3).

- Validación experimental de la herramienta mediante la elaboración de un estudio (Sección 4)

Complementariamente, al final del documento se encuentran los Anexos A (El motor Unity), B (Trabajo previo al desarrollo), C (Tratamiento de *Logs* y otros ficheros generados) y D (Enunciado de la práctica 1 de la asignatura de RV del *Master in Robotics, Graphics and Computer Vision*).

Este proyecto se ha llevado a cabo en el grupo de investigación *Graphics and Imaging Lab*, en la Universidad de Zaragoza. El trabajo del grupo se centra en los *gráficos por computador*, realizándose investigación en áreas de renderizado físicamente correcto, procesamiento de imágenes, fotografía computacional, percepción aplicada, o realidad virtual. Este proyecto se enmarca en esta última; donde el grupo ha trabajado frecuentemente con técnicas de captura de datos de usuarios, conceptos como la saliencia, el análisis del comportamiento visual, así como el uso de técnicas de aprendizaje profundo.

## 1.3. Planificación y herramientas

### 1.3.1. Planificación

Inicialmente se plantea una metodología de desarrollo iterativa, donde se van realizando tareas de desarrollo de componentes del sistema, acompañado de reuniones periódicas para discutir los avances, comentar dudas y planificar los siguientes pasos hasta la próxima reunión.

Se acordó que las reuniones para comentar los avances en el trabajo realizado fueran de carácter semanal todos los lunes, una vez se hubiera completado un pequeño proceso de recopilación de información y adaptación al entorno de desarrollo, el cual se detalla en el Anexo B.

El presente trabajo ha sido desarrollado entre los meses de octubre de 2021 y enero de 2022, donde se han llevado a cabo diferentes tareas. Se enumeran en la Tabla 1.1 junto con su número de ID:

ID	Descripción tarea
1	Familiarización con Unity3d y trabajo previo
2	Implementación componente reproductor de vídeos
3	Implementación componente de recogida de datos de la cabeza
4	Implementación de un modo repetición basado en logs
5	Implementación de selector de comportamiento deseado
6	Implementación de la detección de fijaciones y generación de mapas de saliencia
7	Integración de las gafas RV al proyecto
8	Implementación componente de recogida de datos de eye tracking
9	Validación experimental
10	Redacción de la memoria
11	Repasso final del código
12	Preparación de la entrega final

Tabla 1.1: Tabla de tareas y sus IDs correspondientes con los plazos ilustrados en el diagrama de Gantt de la Figura 1.2.

El siguiente diagrama de Gantt, mostrado en la Figura 1.2, ilustra los plazos de desarrollo del trabajo para cada tarea, identificada por su número de ID:

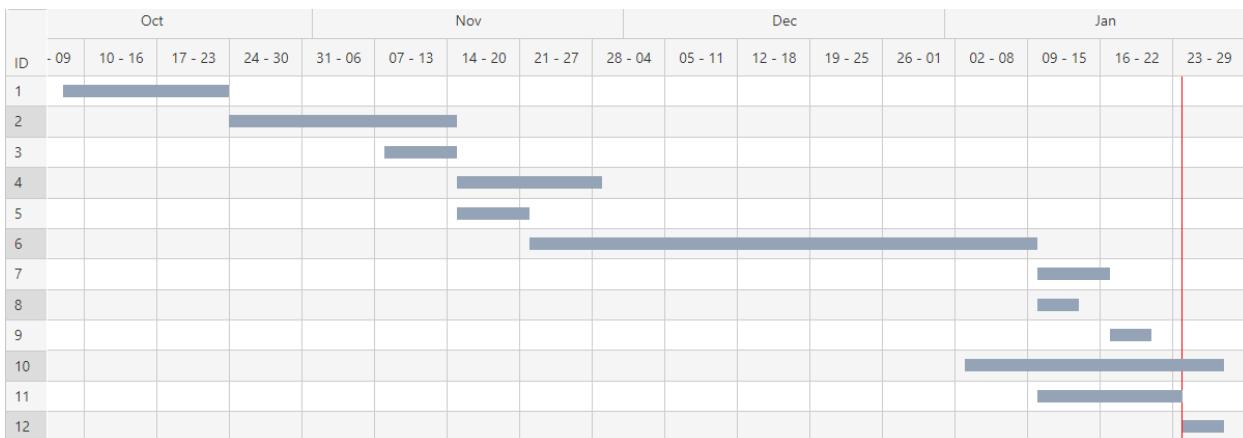


Figura 1.2: Diagrama de Gantt con las tareas identificadas por su ID según la Tabla 1.1.

### 1.3.2. Herramientas utilizadas para el desarrollo

El sistema se ha desarrollado utilizando el motor Unity3D [8], puesto que facilita la inclusión y el uso de determinadas características, como el soporte a un HMD (*Head mounted display*) de realidad virtual, o la posibilidad de reproducir vídeos en 360 grados. De esta manera, se pueden centrar los esfuerzos en programar el comportamiento del *pipeline* de recogida de datos, eliminando la necesidad de programar un reproductor de audio o vídeo de forma manual. Además, también se facilita la integración de clips de audio ambisónico, como se detalla en la sección B.2 del Anexo B. Se considera una parte fundamental del desarrollo, por lo que en el Anexo A se presenta este motor más a fondo, exponiendo su funcionamiento básico.

También se ha hecho uso de *OpenCV plus Unity* [9], un paquete para Unity que permite utilizar la mayoría de las características que ofrece OpenCV para otros lenguajes de programación. Se trata de una adaptación de la versión de C#.

Para el control de versiones se ha utilizado un repositorio de GitHub en el cual se encuentra alojado el código que compone el proyecto de Unity junto con la documentación con las instrucciones para su uso. El repositorio se puede consultar en el siguiente enlace: <https://github.com/PedroPerez14/Pipeline-VR>

Como entorno de desarrollo se ha elegido Visual Studio, por la facilidad de integración que tiene con Unity, ya que ofrece funciones como el autocompletado o el señalamiento de sintaxis propia de Unity. Se ha utilizado la versión de 2019.

Como *hardware* adicional se ha hecho uso de unas gafas de realidad virtual, unas HTC Vive Pro [10], acompañadas de un *eye tracker* fabricado por Pupil Labs [11]. Este *eye tracker* tiene una tasa de muestreo de hasta 200 Hz, dependiendo de la situación y las gafas sobre las que se utilice.



Figura 1.3: Imagen de unas gafas de realidad virtual HTC Vive Pro



Figura 1.4: Imagen del dispositivo de *eye tracking*, integrado en las gafas de realidad virtual

Para integrar el *HMD* (*Head Mounted Display*, ver Glosario de términos) con el *eye tracker* al proyecto, se ha necesitado importar al proyecto el paquete de Unity *hmd-eyes*

[12], junto con OpenXR [13], un *plugin* orientado a la programación en varios dispositivos de realidad virtual o aumentada. También se ha instalado el *plugin* de SteamVR [14], necesario para la comunicación entre algunos modelos de gafas RV y Unity.

## 2. Estado del arte

---

El sistema visual humano funciona como un filtro a la hora de centrar más atención a las regiones más interesantes del entorno. En este proceso se exhibe un comportamiento conocido como fijación visual, consistente en mantener la mirada sobre un mismo punto del espacio. Partiendo de este comportamiento, se ha tratado de explorar y entender la atención visual en numerosas disciplinas a lo largo de los años. Un concepto importante relacionado con el de atención es el de saliencia. Esta se define como la capacidad de un objeto o una región del entorno de destacar y llamar la atención de los observadores.

Una de las formas más utilizadas en los últimos años de modelar la atención humana ha sido a través de la saliencia, elaborando modelos que permitieran evaluar qué atrae al ojo humano en diferentes imágenes o escenas. En 1987, Koch y Ullman [15] presentaron un primer modelo que permitía predecir las regiones más salientes de una imagen, partiendo de un grupo de características de estas imágenes, y su primera implementación práctica fue presentada por Clark y Ferrier [16]. Con el paso de los años han ido apareciendo diferentes trabajos que proponen nuevos modelos o refinan los anteriormente existentes. En diferentes ocasiones se han definido modelos para predecir la saliencia sin la necesidad de recoger datos de usuarios, utilizando las características de las imágenes que se quieren analizar, o utilizando vídeos. También han surgido modelos que adaptaban estos resultados a contenido omnidireccional en RV, y otros que también incorporan el audio como un factor a tener en cuenta, por ejemplo Chao [17]. Hoy en día los modelos con mejor rendimiento utilizan aprendizaje profundo [18], por lo que la obtención de datos de usuarios reales es crucial [19].

La RV aún está desarrollando su propio lenguaje comunicativo, de forma que es importante comprender el máximo posible acerca de la atención humana en este medio, que a diferencia del resto, utiliza contenido en 360 grados. Este contenido presenta retos propios, diferentes a los de otro contenido audiovisual, como por ejemplo la necesidad de mantener un *framerate* elevado para evitar mareos en los usuarios [20] en el caso de los vídeos 360, lo que a su vez deriva en problemas de ancho de banda. Para mitigarlos se puede tratar de modelar qué regiones del vídeo van a ser más exploradas por los usuarios, para centrarse en la calidad de vídeo de esas regiones, como hacen Ching-Ling Fan y otros [21, 22, 23, 24].

En el caso del presente trabajo se quiere estudiar y modelar el comportamiento de usuarios reales ante vídeos 360 en RV, por lo que es necesario disponer de un método

para obtener tanto fijaciones como saliencia. Salvucci y Goldberg [1] presentan diversos métodos sobre cómo obtener fijaciones a partir de datos de un dispositivo de *eyetracking*, mientras que Le Meur y Baccino [25] detallan la forma de obtener mapas de saliencia, partiendo de las fijaciones visuales de los usuarios. Ambos documentos se han tomado como puntos de referencia para el desarrollo del presente trabajo.

# **3. Sistema implementado**

---

En este capítulo se detallan tanto las decisiones de diseño de la herramienta de recogida de datos desarrollada como los detalles de la implementación que se consideren relevantes para entender mejor el proceso de desarrollo. También se comentarán las dificultades encontradas y las soluciones que se les han ido dando. El trabajo previo al desarrollo del *pipeline*, necesario para entender el funcionamiento del motor Unity, se encuentra en el Anexo B. En él se explican los pasos a seguir para reproducir vídeo 360 y audio ambisónico en Unity.

## **3.1. Primera funcionalidad del sistema: Cargar y reproducir clips de vídeo y audio**

Una vez se concluyó el aprendizaje básico, se comenzó el desarrollo de la herramienta de recogida de datos. El primer paso sería conseguir crear un reproductor que muestre una serie de vídeos junto a sus respectivos clips de audio cargados por los usuarios. Para ello, se creó un proyecto nuevo con igual configuración que el anterior. En la única escena que contiene el proyecto se tenía como único objeto la cámara principal, junto a un nuevo objeto, llamado *VideoPlayer*. Este objeto contendría los componentes mencionados en el Anexo B, necesarios para reproducir tanto vídeo 360 como audio ambisónico. Además, se añade un *script* llamado *VideoController*, que se encarga de controlar la lógica detrás de la carga y reproducción de vídeos, así como de gestionar las múltiples opciones para aportar variedad a los experimentos a realizar.

### **3.1.1. Componente VideoController**

Este componente del reproductor de vídeos es el responsable de gestionar qué vídeos se reproducen, cuándo se reproducen y hasta cuándo se reproducen. Presenta una serie de opciones configurables, planteadas de forma que no haya necesidad de modificar los componentes *VideoPlayer* o  *AudioSource* de Unity (Ver Anexo B), centralizando así estas opciones de configuración, lo que simplifica el proceso de puesta a punto de los experimentos que se quieran realizar.

### 3. Sistema implementado

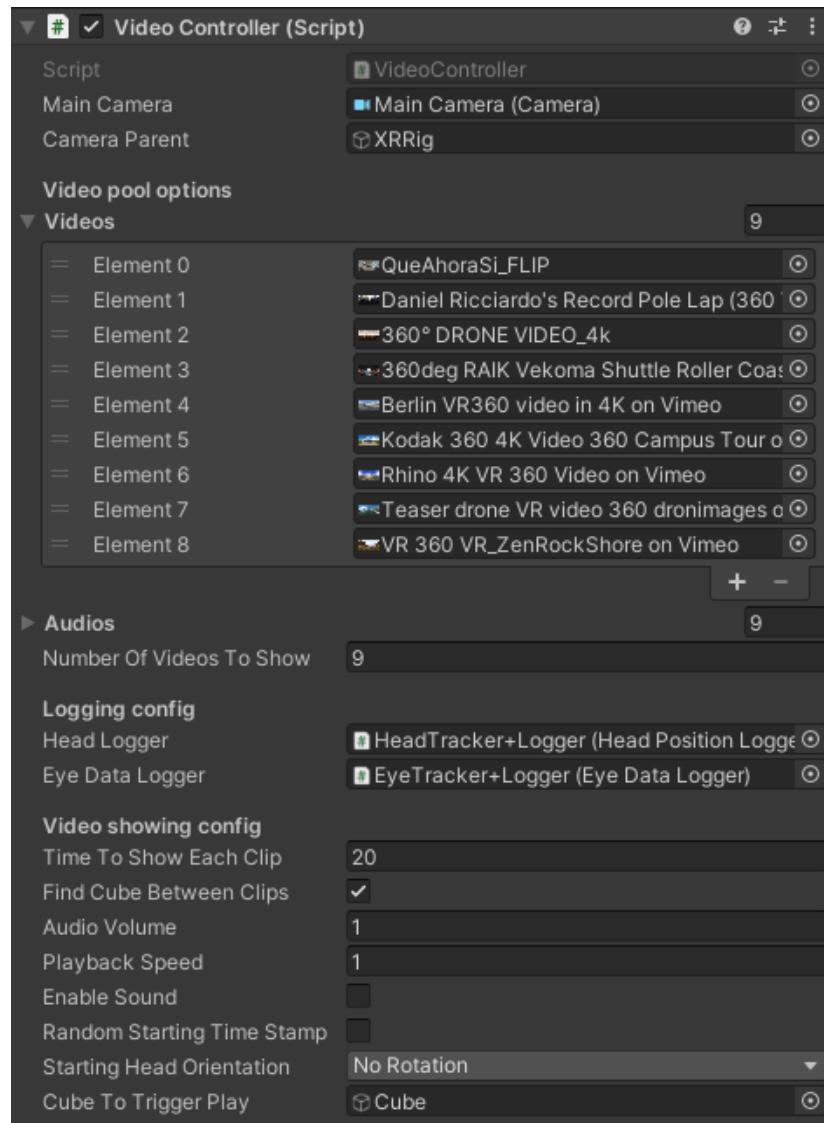


Figura 3.1: Componente VideoController y sus propiedades, visto en el editor de Unity.

Desde el editor de Unity se pueden configurar las siguientes opciones relacionadas con la reproducción de vídeos:

- Objeto que se tendrá en cuenta como punto de vista del usuario.
- Lista de clips de vídeo que pueden ser visionados por los usuarios.
- Lista de clips de audio asociados a los vídeos.
- Cantidad de clips de vídeo que se reproducirán de entre los de la lista.
- Tiempo, en segundos, que durará la visualización de cada clip.
- Opción de esperar una cantidad de tiempo entre el final de un clip y el inicio del siguiente o hacer que los usuarios busquen un objeto en la escena que aparecerá alrededor suyo (Serrano y otros [7], *Supplement*).

- Periodo de tiempo en segundos con la pantalla en negro entre dos clips de vídeo sucesivos.
- Volumen del audio asociado a esos vídeos, entre 0 y 1.
- Habilitar o deshabilitar el audio asociado a los vídeos.
- Habilitar que los vídeos se reproduzcan en un momento aleatorio, en lugar de empezar por el principio de estos.
- Elegir posición inicial de la cámara cada vez que se reproduzca un clip de vídeo entre no modificarla, cambiar la longitud a una aleatoria entre 0, 90, 180 y 270 grados, y aleatorizar la latitud y longitud por completo.
- Componente que hará funciones de *logging* relativo a la orientación de la cámara (se asume equivalente a posición de la cabeza del usuario).
- Componente que hará funciones de *logging* relativo a la mirada.

En cuanto a los clips de vídeo y de audio que se pueden asignar a este componente, hay que destacar que son dos listas independientes, y que es responsabilidad de quien lo configure hacer que la n-ésima posición de cada una de las dos listas haga referencia a un clip de audio y uno de vídeo que se correspondan entre sí, así como que las dos listas tengan el mismo número de elementos, puesto que se asume que cada vídeo viene acompañado de un audio. El motivo de que audio y vídeo se traten por separado es que Unity no puede reproducir los clips de audio ambisónico si estos vienen junto al vídeo en formatos como el *.mp4*, ya que esa pista de audio se trata como si fuera normal. Por esto, se tratan los clips de audio separados de los clips de vídeo.

A continuación se explica más acerca del funcionamiento de este componente del programa, incluyendo aquellos detalles de la implementación que se consideren relevantes.

#### Bucle de reproducción y corrutinas

Durante una las reuniones periódicas se acordó que los vídeos fueran reproducidos aleatoriamente, eligiendo tantos como le se haya indicado, de entre la lista de vídeos que se hayan cargado. Para este paso es necesario haber importado a Unity los vídeos y audios que se pretendan utilizar, y añadirlos a la lista después de ello. Además, no se puede reproducir dos veces el mismo clip de vídeo en una misma ejecución, salvo que haya sido incluido dos veces en la lista de vídeos.

Algunos parámetros tienen que ver con el control de los tiempos de reproducción de cada clip de audio o vídeo, por lo que se necesita crear una espera temporal en el bucle de reproducción. En Unity, cada vez que se llama a una función, esta se ejecuta hasta quedar completada y luego devuelve el control al método que la llamó. De esta manera, hacer un bucle simple del estilo “reproducir, esperar X segundos, pausar” no funcionaría, ya que el resto de tareas que queramos tener en ejecución, como hacer *logging* de información del

### 3. Sistema implementado

---

comportamiento visual, no funcionarían durante la espera temporal. La manera más sencilla de hacer esto en Unity es mediante el uso de corutinas, un tipo especial de función que permite devolver el control al motor y reanudar su ejecución en el siguiente *frame*, de forma que el resto de tareas que se quieran ejecutar pueden hacerlo sin verse bloqueadas por otra. Son invocadas una vez durante el cálculo de cada *frame*. Más detalles sobre las corutinas en la Sección A.4 del Anexo A.

Sabiendo esto, y que una espera temporal en Unity se considera una corutina, se diseña el esquema del bucle encargado de la reproducción de vídeos:

---

**Algoritmo 1:** Esquema del bucle principal del reproductor audiovisual.

---

```
int videoID;
for numVídeosQueReproducir veces do
    videoID = ElegirVideoSinRepetir();
    AjustarRenderTexture(videos[videoID].resolucion);
    ReproducirVideoYAudio(videoID);
    Esperar(min(tiempoQueReproducir, videos[clipID].duracion));
    Parar(videoID);
    if esperarTiempo_buscarObjeto3D then
        | EsperarTiempo(tiempoEntreVideos);
    end
    else
        | EsperarHastaQueUsuarioEncuentreObjeto3D();
    end
end
return;
```

---

En este esquema se presentan las tareas más relevantes que se llevan a cabo para cada uno de los vídeos que se reproducen. Una de ellas es ajustar la *RenderTexture* que se necesita para reproducir los vídeos, como se explica en la Sección B.1 del Anexo B. A esta se le deben dar las dimensiones correspondientes a la resolución del clip de vídeo que se vaya a reproducir, o de lo contrario se mostrará un vídeo con la resolución del anterior, pudiendo producir cierta distorsión o pérdida de calidad de imagen durante la ejecución, como se muestra a continuación en la Figura 3.2.



Figura 3.2: Pérdida de calidad de imagen al no redimensionar la *RenderTexture* correctamente.

La espera temporal correspondiente al tiempo que reproducir cada vídeo no siempre corresponde a la cantidad de segundos que especifiquen los usuarios. Esto se debe a que se contempla la posibilidad de que un vídeo sea más corto que la cantidad de tiempo durante la cual se quiere reproducir. Para tener eso en cuenta, la espera temporal corresponde al valor mínimo entre el valor especificado en la configuración y la duración del vídeo a reproducir.

Así se evita mostrar la pantalla en negro durante unos segundos hasta que termine la espera temporal. Aunque se comenta en la Sección 3.2, este intervalo temporal se utiliza al recoger datos del comportamiento visual de los usuarios, por lo que no calcularlo correctamente podría derivar en lecturas erróneas correspondientes a los usuarios contemplando una imagen completamente negra.

#### Gestión de las opciones de configuración

A la hora de comenzar la reproducción en un momento aleatorio, se tiene en cuenta el número de segundos durante los cuales se quiere reproducir, generando un instante temporal en el intervalo  $\{0 \dots [\text{final} - \min(\text{tiempoQueReproducir}, \text{videos}[\text{clipID}].\text{duracion})]\}$ . Así se garantiza que se reproduzca el vídeo durante tantos segundos como haya especificado el usuario en la configuración, o durante tanto tiempo como dure el vídeo si es demasiado corto, como se ha explicado anteriormente.

La opción de deshabilitar el audio ha sido trivial de implementar, accediendo de nuevo a las propiedades del componente  *AudioSource*  (Anexo B, Sección B.2). Para la opción de la posición de la cámara tampoco se han tenido problemas, dependiendo de la opción elegida se modifica su rotación de una forma u otra (se puede consultar el sistema de coordenadas utilizado por Unity en la Sección A.2 del Anexo A).

- **No rotation:** No se cambia nada.
- **Rotate 90 Deg Intervals:** Su rotación XYZ será  $\{0, \text{rand}(0,3) * 90, 0\}$ .
- **Fully Randomized:** Su rotación XYZ será  $\{\text{rand}(0,90), \text{rand}(-180,180), 0\}$ .

Por último, como ya se ha comentado existe una opción que permite indicar qué mecanismo se quiere utilizar para desencadenar la reproducción de un clip de vídeo en cuanto termina el anterior. Si se elige una espera temporal, se ejecuta una corutina con esa espera. En el caso de elegir la opción de buscar un objeto tridimensional, cada vez que se detiene un vídeo se muestra un cubo rojo (puede especificarse cualquier otro objeto) ubicado en un lugar aleatorio alrededor del observador, a una distancia de entre 1 y 5 unidades de Unity.

Los usuarios deben mirar hacia este cubo, de forma que se realiza un  *raycast*  (Ver glosario de términos) cada vez que se calcula un nuevo  *frame* . Este  *raycast*  utiliza la posición y orientación de la cámara, por lo que se tendrá que centrar el cubo dentro del campo visual. Si se comprueba que están mirando al cubo, se incrementa un contador de tiempo con el tiempo transcurrido desde el último  *frame*  de ejecución, hasta que se sobrepasa o iguala el valor de una décima de segundo.

Esto se hace así para evitar que se encuentre el cubo por accidente y se pierda de vista al instante, forzando a los usuarios a tener la atención centrada en algo para el momento en el que comienza el siguiente vídeo. En la Figura 3.3 mostrada a continuación se puede ver un ejemplo de cómo funciona este  *raycast* .

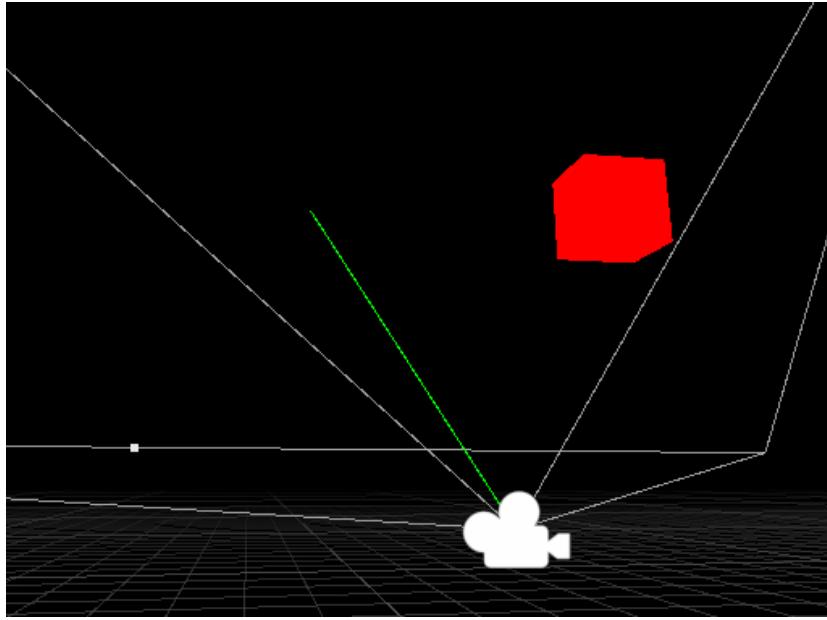


Figura 3.3: Ejemplo del *raycast* (vector verde) ejecutado desde la cámara (FOV en blanco).

#### Modificaciones en la cámara al incorporar las gafas VR

Al adaptar la cámara a las gafas de realidad virtual se necesitó cambiar la jerarquía de la escena, tal como requería OpenXR [26]. Ahora la cámara es “hija” de un objeto vacío, que a su vez es hijo de otro objeto vacío. El objeto raíz adquiere la posición de la cámara, el hijo puede presentar un *offset* para representar a la altura de la cámara en los ojos de los usuarios, y la cámara estará centrada en relación a este objeto. Además tiene un componente nuevo, *Tracked Pose Driver*. El aspecto de la jerarquía de objetos que se ha utilizado en el *pipeline* se muestra a continuación, en la Figura 3.4:

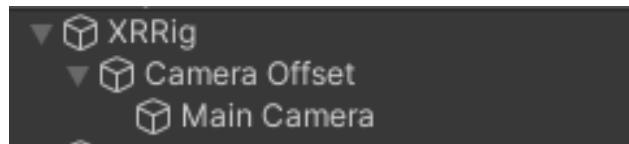


Figura 3.4: Jerarquía requerida por *OpenXR* para la cámara.

Este componente se encarga de orientar la cámara constantemente para que corresponda con la posición del usuario que lleva puestas las gafas de realidad virtual. Por tanto, no sirve rotar la cámara al principio de cada vídeo ya que este valor sería sobreescrito al instante. Se aprovecha la nueva jerarquía requerida por OpenXR y se rota el componente padre de la cámara, consiguiendo el efecto deseado.

## 3.2. Recogida de datos

Este es el propósito principal del *pipeline* desarrollado. Se han centrado los esfuerzos en recoger dos tipos de datos. En primer lugar, la información relativa a la orientación de la cabeza de los usuarios, para comprender mejor hacia qué región de las escenas se focaliza la atención. En segundo lugar, se recoge la información de la dirección de la mirada de los usuarios, permitiendo obtener mayor precisión acerca de las regiones que atraen la atención. Esto se logra gracias a dispositivos de *eye tracking*, como el que se ha utilizado durante las fases de desarrollo y validación, presentado en la Sección 1.3.2, Figura 1.4. La recogida de estos datos se realiza durante la visualización de los vídeos 360 en realidad virtual por parte de los usuarios, y son dos los componentes encargados de esta tarea, uno para cada tipo.

Almacenar esta información en ficheros debidamente etiquetados permite disponer de ella en todo momento, ya sea para calcular fijaciones, mapas de saliencia (ver Sección 3.4), o para crear *datasets* utilizables para entrenar modelos de predicción de datos relacionados con la saliencia.

Aunque sus estructuras y funcionamientos son bastante parecidos, se explican los dos componentes encargados de recolectar datos tanto de la orientación de la cabeza como de la dirección de la mirada. También se detalla qué datos se recopilan exactamente y cómo es el proceso de recogida.

### 3.2.1. Componente HeadDataLogger

El siguiente paso del desarrollo tras tener listo el reproductor de vídeos 360 es crear un componente que se encargue de recoger de forma periódica información relativa a la rotación de la cabeza de los usuarios durante la visualización de los vídeos. Para ello, se creó un *script* que se añade a un objeto de la escena de Unity, el nombre asignado a este componente es *HeadDataLogger*, igual que el del objeto al que pertenece, para mejor identificación por parte de los usuarios.

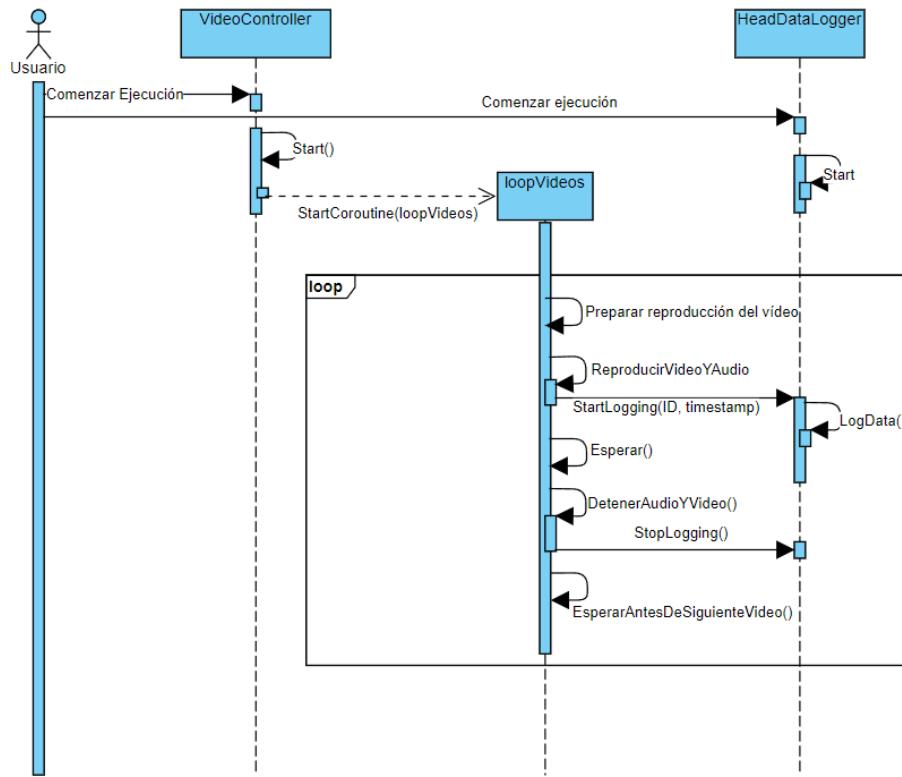


Figura 3.5: Diagrama de secuencia con el funcionamiento del sistema de *logging* simplificado.

#### Funcionamiento base y datos a recoger

Desde un principio se tuvieron claras varias ideas acerca del comportamiento que se quería implementar:

- Se muestrea información de la rotación de la cabeza del usuario de forma periódica durante la reproducción de los clips de vídeo.
- Este muestreo de datos comienza cuando el vídeo se empieza a reproducir y termina tras la cantidad de tiempo de reproducción que se le asignó en el componente dedicado a la reproducción de vídeos (Figura 3.5 y Algoritmo 1).
- La información recogida se guarda en ficheros en formato .csv.
- Se genera un fichero para cada uno de los vídeos que visualiza un usuario durante una ejecución, quedando al final tantos *logs* como vídeos haya visualizado.
- El usuario puede especificar en qué carpeta guardar los ficheros generados.
- Se puede activar o desactivar la recogida de datos antes de lanzar una ejecución, en caso de que los usuarios estén haciendo pruebas donde no necesiten recoger estos datos.

Con esta base (ver Figura 3.5), se acordó en una reunión de control que los datos a recopilar y guardar en estos ficheros serían relativos a la rotación de la cabeza del usuario, sin tener en cuenta la posición de esta. Principalmente se debe a que al tratarse de vídeos en 360 grados proyectados en la *Skybox* de la escena, un cambio en la posición del usuario no afecta en absoluto a la percepción del vídeo, ya que el cielo de la escena es una posición en el infinito. Sí que podría tener sentido recoger información de la posición de la cabeza de los usuarios en caso de querer estudiar, por ejemplo, sus reacciones a diferentes tipos de contenido a través de su postura (por ejemplo, se podría querer estudiar el lenguaje corporal al visualizar vídeos con sonidos abruptos como explosiones, golpes inesperados o sustos, frente a contenido que produzca otro tipo de estímulos más relajantes). No obstante, se ha considerado que no merece la pena debido a lo específica que sería esta aplicación, y a que la mayoría de la información útil la aporta la orientación de la cabeza y no la posición.

#### Recogida de la información

Hasta ahora se ha explicado qué datos se recogen y por qué. En la Sección C.1 del Anexo C se comenta qué aspecto tienen los ficheros generados tras recopilar esa información. Falta por explicar cómo se obtiene la información. En primer lugar se explica cómo se inicia y detiene el proceso de recogida de datos, que va sincronizado con el inicio y el final de su reproducción en el componente dedicado a ello.

Cada vez que *VideoController* inicia un vídeo, avisa a *HeadDataLogger* para que compruebe si debe iniciar la recogida de datos. A continuación, *HeadDataLogger* comprueba si en su configuración se le había indicado que debe recoger datos o no, y si ya hay otro proceso de *logging* en marcha o no, algo que no debería suceder. Si debe recoger datos y no estaba volcando información a ningún otro fichero, indica que ahora sí está recogiendo información y comienza el proceso. La forma de comenzarlo difiere de la primera versión implementada a la segunda, se explica a continuación.

---

**Algoritmo 2:** Forma notificar a *HeadDataController* desde *VideoController* para que comience el proceso de *logging*.

---

```

empezarLogging(int ID, float timestamp)
if habilitarLogging && !estaLogeando then
    estaLogeando = true;
    CrearLog(ID, timestamp);
    if primeraVersionDelComponente then
        //Algoritmo 4
        IniciarCorrutina(CorrutinaLogging);
    end
    if segundaVersionDelComponente then
        //Algoritmo 5
        ControladorEyeTracker.Suscribirse(EsperarEventoEyeTracker());
    end
    t0 = Reloj.Ahora();
    //Ver Ecuación 3.1
end

```

---

Este proceso comienza por crear el fichero *.csv* al que volcar la información, según las reglas de nombrado explicadas en la Sección C.1 del Anexo C. A continuación escribe las tres primeras líneas del fichero, donde detalla el *timestamp* de reproducción del vídeo en el momento de comenzar a recoger los datos (necesario para recrear el *scanpath* seguido por los usuarios, se explica en la Sección 3.3), la rotación de la cámara (equivalente a la cabeza del usuario que lleva las gafas de realidad virtual) en ese instante, y los nombres de cada uno de los campos que constituyen las muestras que se recogen. El componente *VideoController* se encarga de proporcionar tanto el *timestamp* del vídeo como su *ID*, necesarios para el proceso de creación del *log*.

En el caso de detener la recogida de datos, se indica que ya no se están recogiendo datos, y se detiene el proceso. La forma de detenerlo es deteniendo una corutina, o desuscribiéndose de un evento del *eye tracker*. Se explica a continuación con más detalle.

---

**Algoritmo 3:** Forma de notificar a *HeadDataController* desde *VideoController* para que comience el proceso de *logging*.

---

```

detenerLogging()
if habilitarLogging && estaLogeando then
    estaLogeando = false;
    if primeraVersionDelComponente then
        //Algoritmo 4
        DetenerCorrutina(CorrutinaLogging);
    end
    if segundaVersionDelComponente then
        //Algoritmo 5
        ControladorEyeTracker.Desuscribirse(EsperarEventoEyeTracker());
    end
end
```

---

Para generar las marcas temporales (*timestamps*) que se asignan a cada muestra se consideraron varias opciones. La primera opción era utilizar la variable *Time.time* de Unity, que contiene el tiempo en segundos desde el inicio de la ejecución de la aplicación, pero no tiene en cuenta el tiempo transcurrido durante el *frame* actual. Por este motivo se decidió descartar la opción. La segunda es la variable *Time.realtimeSinceStartup* del motor, que contiene el tiempo en segundos desde el inicio de la ejecución. No se utiliza por esto mismo, ya que se quiere que cada *timestamp* sea relativo al momento de comenzar a registrar datos. La tercera, que es la que se ha decidido implementar, es utilizar el reloj del sistema para obtener un tiempo inicial  $t_0$ , medido en el momento de comenzar a muestrear datos. Para cada muestra  $i$  se genera un segundo tiempo,  $t_i$ , de forma que el *timestamp* mostrado en cada muestra  $i$ -ésima de datos de la cabeza de los usuarios, relativo al instante de inicio del muestreo  $t_0$  del fichero final se calcula como se muestra en la Ecuación 3.1:

$$timestamp_i = t_i - t_0 \quad (3.1)$$

Ecuación 3.1: Cálculo del *timestamp* de una muestra de datos, relativo al instante inicial de la recogida de datos.

Tras calcular los *timestamps* que aparecen en los *logs*, el siguiente paso es obtener el resto de valores relativos a la orientación de la cabeza de los usuarios. Se enumeran en la Sección C.1 del Anexo C. Estos valores (rotación como ángulo euleriano, como cuaternión y vector *forward* de la cámara) se obtienen accediendo directamente al componente *Transform* de la cámara cada vez que sea necesario tomar una muestra, o se calculan a partir de los demás valores (coordenadas *uv*).

Queda por aclarar la periodicidad de las muestras tomadas. No se ha aclarado hasta ahora esta parte del funcionamiento del componente dado que durante el desarrollo han existido dos versiones con similitudes y diferencias clave entre sí, y aquí es donde se encuentran las principales diferencias entre las dos versiones implementadas. Se explican ambas a continuación, junto a por qué se ha pasado de la primera a la segunda para la versión final del *pipeline*.

#### Primera versión: Tasa de muestreo configurable

La primera versión de este componente permitía al usuario determinar la tasa de muestreo de datos, medida en milisegundos. Esta se encontraba disponible como un campo modificable en el editor de Unity, como muestra la Figura 3.6.

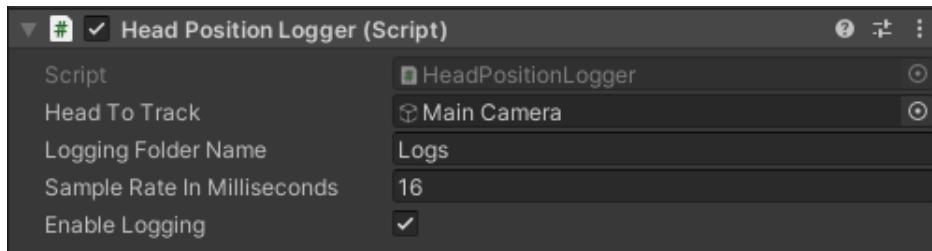


Figura 3.6: Propiedades configurables del componente *HeadDataLogger* en su primera versión.

Con este diseño, la implementación pasaba por una corutina que esperara la cantidad de tiempo requerida en las opciones de configuración. Tras la espera, toma una muestra de los datos que necesita y los escribe en el fichero de salida.

**Algoritmo 4:** Esquema de la corutina que recoge datos en la primera implementación del componente.

---

```

float timestamp_muestra = ObtenerTimestamp(); //Ver Ecuación 3.1
for true do
    if log.Existe() then
        Vector3 rot = cam.transform.euler;
        Cuaternion quat = cam.transform.rotation;
        Vector3 fwd = cam.transform.forward;
        Vector2 uv = CalcularUV(fwd); // [27]
        log.EscribirLinea(timestamp_muestra, rot, quat, fwd, uv);
    end
    else
        | Debug.Log("Error");
    end
    Esperar(tasaDeMuestreo_ms);
end

```

---

#### Versión final: Sincronizado con las muestras del *eyetracker*

Esta es la versión definitiva del componente, la primera se tuvo que modificar debido al funcionamiento del plugin de *eye tracking*. Se explica con más detalle en la Sección 3.2.2. En resumen, el *eye tracker* muestrea a una tasa fija de 120Hz o 200Hz, dependiendo de ciertas condiciones [11]. Para leer los datos muestreados, hay que establecer una conexión con la aplicación externa de PupilLabs, que se ejecuta aparte del *pipeline* en el sistema. Al establecer la conexión es necesario suscribirse a los eventos que genera el componente *GazeController* [12].

Al suscribirse a los eventos, se hace un tratamiento asíncrono de la información. Por tanto, de haber mantenido el diseño de la primera versión, el componente que registra datos de la cabeza se comportaría de manera síncrona respecto a la tasa de muestreo que se le indique mientras que el que registra datos de la mirada presentaría un comportamiento asíncrono, con una tasa de recogida de información dependiente del *hardware* utilizado.

Esto no se ha considerado apropiado, ya que si estos dos tipos de información son complementarios entre sí, la decisión más racional es hacer que la información se recopile de la manera más homogénea entre ambos componentes. Este es el motivo de que esta segunda versión del componente funcione de manera asíncrona, dependiendo de los eventos del *eyetracker* a pesar de no necesitar leer datos de la mirada. La decisión se consultó con los directores del trabajo y se determinó que era la opción más apropiada. De esta forma, el componente pasó a tener el aspecto que muestra la Figura 3.7.

### 3. Sistema implementado

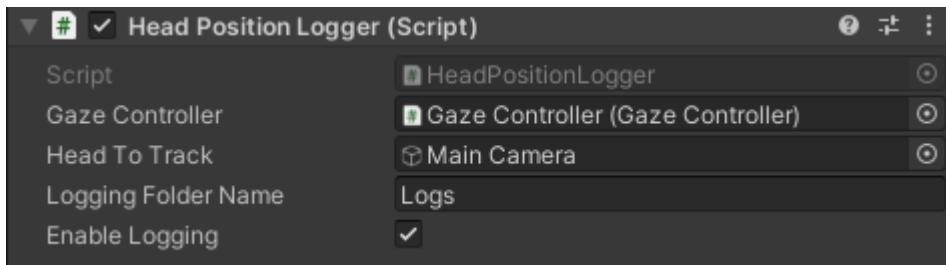


Figura 3.7: Propiedades en el editor de Unity de la segunda versión del componente *HeadDataLogger*.

Otra razón detrás de la toma de esta decisión es que al recibir ambos componentes de *logging* los eventos procedentes del *eye tracker* al mismo tiempo, en ambos tipos de *logs* aparecen el mismo número de entradas de datos, y se corresponden entre sí al tener el mismo *timestamp* (Ver Figuras C.1 y C.3).

---

**Algoritmo 5:** Esquema de la función con la que el componente se suscribe a los eventos del *eye tracker* y recoge datos en la segunda implementación.

---

```
método EsperarEventoEyeTracker(DatosEyeTracker datos)
if log.Existe() then
    Vector3 rot = cam.transform.euler;
    Cuaternion quat = cam.transform.rotation;
    Vector3 fwd = cam.transform.forward;
    Vector2 uv = CalcularUV(fwd); //Ver Sección [27]
    log.EscribirLinea(timestamp_muestra, rot, quat, fwd, uv);
end
else
    | Debug.Log("Error");
end
```

---

#### 3.2.2. Componente EyeDataLogger

Este componente está diseñado para recoger información relativa a la mirada de los usuarios, en conjunción con un dispositivo de *eye tracking* montado en las gafas de realidad virtual que se utilicen para el visionado de los vídeos. Fue de los últimos en incorporarse al *pipeline* desarrollado, aunque se detalla ahora debido a todas las similitudes que presenta con el componente de recogida de datos de la cabeza, *HeadDataLogger*. La Figura 3.8 muestra el aspecto de este componente en el editor de Unity.

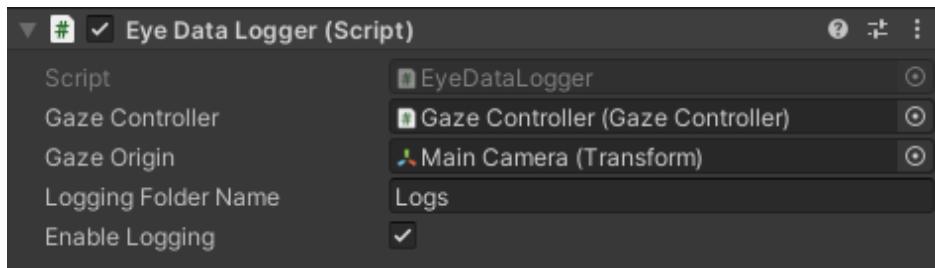


Figura 3.8: Vista del componente *EyeDataLogger* desde el editor de Unity, con sus propiedades editables.

## Funcionamiento y datos a recoger

A la hora de diseñar este componente, se decidió que el comportamiento fuera lo más parecido al de su complementario, para mantener una cierta consistencia en la generación de datos, así como en el formato de estos. Al igual que en *HeadDataLogger* se recoge la información de manera periódica, que se vuelca a un fichero en formato *.csv*. Para más información acerca de la estructura de los ficheros de *log* generados por este componente se puede consultar la Sección C.2 del Anexo C. La recogida de información sigue el mismo patrón, comienza cuando se inicia la reproducción de un vídeo y termina cuando se detiene el vídeo tras un tiempo determinado y configurable. También se genera un fichero de *log* por cada vídeo visionado por los usuarios, de forma que al final de una sesión de recogida de datos se tengan tantos ficheros como vídeos se han reproducido. Los usuarios también pueden especificar la carpeta en la que se generan estos ficheros de *log*, y hasta pueden desactivar la recogida de información de este tipo por completo si así lo desean.

## Recogida de la información

El proceso de recogida de datos es idéntico al que se expone en la Sección 3.2.1 para los datos de la cabeza. El componente encargado de la reproducción de vídeos avisa a este componente cuando va a poner en marcha un clip de vídeo, y este comprueba que deba recoger datos y que no estuviera recogiendo datos previamente (algo que no debería ocurrir pero se comprueba por si acaso). Si se cumplen ambas condiciones, comienza la recogida de datos suscribiéndose a los eventos generados por el *eye tracker*. Para finalizar la recogida de datos, se indica que ya no se están recogiendo datos y se desubscribe de esos eventos. Este comportamiento es exactamente igual al presentado en los Algoritmos 2 y 3, en la sección referente a la información de la cabeza de los usuarios. La única diferencia está en el método que se suscribe a los eventos, pues este recopila otro tipo de información. Se puede consultar su funcionamiento en el Algoritmo 6.

La generación de los *timestamps* de cada muestra sigue exactamente el mismo procedimiento mostrado en la Sección 3.2.1 para los datos de la cabeza.

#### Eventos generados por el *eye tracker*

Por último, la recogida de datos como tal se gestiona dentro del tratamiento a los eventos que llegan del *eye tracker*, en la función que se suscribe a los eventos. Esta recibe como parámetro un tipo de dato llamado *GazeData*, del cual se extraen la dirección de la mirada y la confianza de la medición. La dirección de la mirada se convierte a diferentes tipos de dato para mayor redundancia de información, y se escribe en el *log* de datos.

---

**Algoritmo 6:** Esquema de la función con la que *EyeDataLogger* se suscribe a los eventos del *eye tracker* y recoge datos de la mirada.

---

```

método RecibirDatosEyeTracker(DatosEyeTracker datos)
if log.Existe() then
    float timestamp_muestra = ObtenerTimestamp(); //Ver Figura 3.1
    Vector3 dir = ConvertirACoordsGlobales(datos.dirLocalMirada);
    Quaternion quat = Quaternion(dir);
    Vector3 rot = quat.Euler();
    Vector2 uv = CalcularUV(dir); //Ver [27]
    float conf = datos.confianza;
    log.EscribirLinea(timestamp_muestra, rot, quat, dir, uv, conf);
end
else
    | Debug.Log("Error");
end

```

---

### 3.3. Recreación de *scanpaths* a partir de logs

Una vez implementada la funcionalidad de recogida de datos, se decidió que sería muy útil tener alguna forma de volver a visualizar en cualquier momento la trayectoria seguida por un usuario al explorar una escena. Esto se podría utilizar como validación de la implementación de la recogida de datos explicada en la Sección anterior, pero más importante, se puede utilizar en un contexto real de investigación para visualizar los datos y buscar anomalías o patrones comunes entre los usuarios sobre los que se haya efectuado la recogida de datos.

#### 3.3.1. Componente ScanpathReplayer

Para esta tarea, se creó un nuevo objeto en la escena principal del proyecto, al que se ha asignado el componente encargado de implementar el comportamiento deseado, un *script*. Tanto el objeto como su componente comparten el mismo nombre, *ScanpathReplayer*. La vista del componente desde el editor de propiedades de Unity se muestra a continuación, en la Figura 3.9.

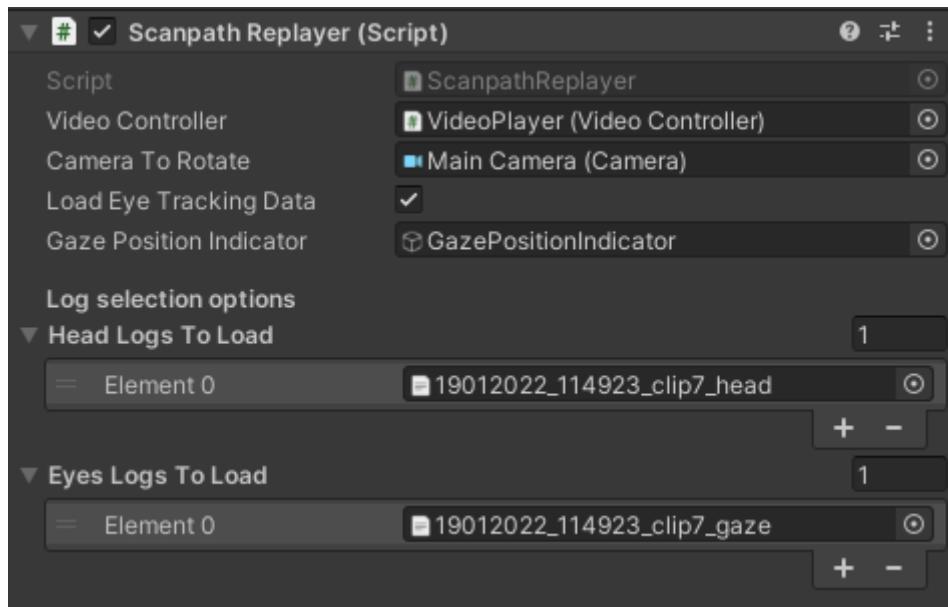


Figura 3.9: Opciones configurables que ofrece el componente *ScanpathReplayer* en el editor de Unity.

Al plantear cómo se lograría implementar un “modo repetición”, se aprovechó que ya se tenía almacenada en *logs* información referente tanto a instantes de tiempo (*timestamps*) como a la orientación (dirección de la mirada o rotación de la cabeza, dependiendo del tipo de fichero). El paso lógico fue hacer que *ScanpathReplayer* leyera estos ficheros y con esa información pudiera recrear los *scanpaths* generados por los usuarios. El comportamiento diseñado se basa en las siguientes ideas clave:

- El componente recibe una lista de *logs*, modificable desde el editor de Unity para su fácil manejo.
- Al iniciar la ejecución se procesa fichero a fichero cada uno de los *logs* que hay en la lista, reproduciendo el vídeo asociado a cada uno.
- Durante estas reproducciones, el componente lee las muestras registradas en cada una de las líneas del fichero. Para cada una, se ajusta la rotación de la cámara acorde al instante temporal actual.

Este diseño previo a la implementación es, a grandes rasgos, el que se tiene en la versión final del componente. No obstante, hay una serie de detalles que no se han tenido en cuenta, como la necesidad de interpolar entre los valores de las muestras leídas. Todo esto se discute a continuación, elaborando acerca de las decisiones tomadas.

#### Reproducción de clips de vídeo y audio desde un *timestamp* hasta otro

Una vez se han almacenado los datos de forma que sea fácil acceder a ellos (El proceso de carga de *logs* en cachés en memoria para reducir los accesos a disco se explica en la

### 3. Sistema implementado

Sección C.3 del Anexo C), el siguiente paso es poner en marcha la recreación de *scanpaths*. Para ello, el componente necesita decirle a *VideoController* que reproduzca una serie de vídeos. Esta lista de vídeos se genera extrayendo los IDs del título de cada vídeo, como se muestra en las Figuras C.2 y C.4. Así, *VideoController* reproducirá los clips de vídeo y audio que ocupen esas posiciones en su lista de vídeos. **Nota:** Cuando se esté en este modo no aplican las opciones de configuración del reproductor de vídeos tales como silenciar el audio o modificar la velocidad de reproducción de audio y vídeo.

Se debe tener especial cuidado para asegurarse de que coincidan los IDs de los clips de vídeo y audio con sus posiciones en la lista de *VideoController* en el momento de poner en marcha la recreación del *scanpath*. Por el momento, esto es responsabilidad de los usuarios, por lo que se recomienda cambiar las listas de vídeos y audios lo mínimo posible.

Para ayudar con esta tarea, se configuró el componente *HeadDataLogger* para que generase dos ficheros de texto aparte del *log* que corresponda. El primero contiene los títulos de los clips de vídeo en orden, uno en cada línea. El segundo tiene la misma estructura, pero con los títulos de los clips de audio. Los nombres de estos ficheros de texto siguen el siguiente patrón:

**ddMMyyyy\_HHmmss\_type\_clips\_list.txt**

Figura 3.10: Formato del nombre de los ficheros con la lista de nombres de los vídeos y audios.

En la Figura 3.10 aparece *type* en referencia al tipo de lista, que puede ser **audio** o **video**. Adicionalmente, la marca temporal siempre coincide con la del primer log generado durante una sesión de recogida de datos, para indicar que ese era el orden de la lista de vídeos en el instante de comenzar esa recogida de datos. Se muestra en la Figura 3.11.

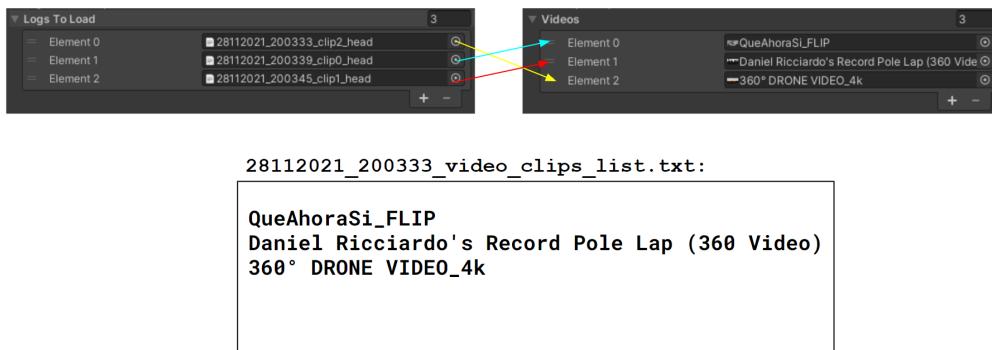


Figura 3.11: Correspondencia entre IDs del título de los *logs* y el orden de los vídeos de *VideoController*, validado con el contenido del fichero de texto con la lista de títulos.

Junto a la lista de qué clips de vídeo reproducir, se indica también a *VideoController* el instante inicial a partir del que reproducir cada vídeo, así como la cantidad de tiempo durante la que reproducir antes de parar. Estos campos se obtienen a partir de la cabecera

del fichero y del *timestamp* de la última muestra que contenga. El bucle de reproducción en *VideoController* está implementado con una corutina de Unity. Su comportamiento se resume de la siguiente manera:

---

**Algoritmo 7:** Esquema del bucle del reproductor audiovisual al recrear *scanpaths* por orden de *ScanpathReplayer*.

---

```

reproducirCola(int[] IDs, float[] timestamps_inicio, float[] duraciones)
int i=0;
for IDs.longitud() veces do
    AjustarRenderTexture(videos[IDs[i]].resolucion);
    ReproducirVideoYAudioAPartirDeTimestamp(IDs[i], timestamps_inicio[i]);
    Esperar(duraciones[i]);
    Parar(videos[IDs[i]]);
    Esperar(tiempoEntreVideos);
    i++;
end
return;

```

---

Se puede observar que en este caso sí que se utiliza duración de la espera temporal entre vídeo y vídeo con la pantalla en negro que se configura en este componente. Esta es la única propiedad de *VideoController* que influye en el funcionamiento de *ScanpathReplayer*.

### Corrutina principal

Al mismo tiempo que se indica al reproductor de vídeos que ponga en marcha la corutina encargada de reproducir la lista de vídeos, se pone en marcha otra dentro del propio *ScanpathReplayer* que, de forma similar a la del reproductor de vídeos, se encarga de indicar qué *log* es el que se está procesando en cada momento. También organiza los valores de otras variables que permiten saber qué muestras tomar como referencia para la rotación de la cámara (o dirección de la mirada).

---

**Algoritmo 8:** Esquema de la corrutina que gestiona la recreación de los *scanpaths* recogidos en *logs*.

---

```

CorrutinaRepeticionScanpaths() int logActual=0;
for logsCabezaQueCargar.longitud() veces do
    t'_0 = Reloj.Ahora();
    muestraActual = 0;
    muestraSiguiente = 1;
    repeticionEnCurso = true;
    Esperar(duraciones[logActual]);
    repeticionEnCurso = false;
    Esperar(tiempoEntreVideos);
    logActual++;
end
return;

```

---

En el esquema arriba mostrado se han introducido unas cuantas variables cuyo funcionamiento se desconoce por el momento. Antes de explicarlo, para entenderlo se debe pensar en la corutina arriba descrita como una forma de apuntar a las primeras dos mediciones de cada *log* de forma coincidente con el instante en el que se vaya a iniciar la reproducción del vídeo asociado a ese fichero. Con esas variables que se presentan a continuación se realiza una interpolación que permite reconstruir el *scanpath* seguido por un usuario.

#### Interpolando datos para recrear *scanpaths*

Un fichero de *log* cargado en las cachés tiene *timestamps* crecientes tales que el instante en el que se comienza a almacenar información es el 0. Por eso la primera muestra suele tener como *timestamp* el valor 0, o un valor diferente pero muy cercano a 0.

Ahora se quiere “deshacer” ese camino, es decir, obtener una medición temporal en el momento de comenzar la repetición que pueda equivaler al momento 0 de registrar la información ( $t_0$  en la sección 3.2.1). Para ello se genera una medición temporal que será tratada como ese instante 0 de la repetición,  $t'_0$ .

Para establecer la rotación de la cámara y la dirección de la mirada, en cada *frame* se obtiene un segundo *timestamp*,  $t'_i$ . A continuación, se obtiene el tiempo transcurrido desde el inicio de la repetición mediante la Ecuación 3.2:

$$\text{currentReplayTimestamp} = t'_i - t'_0 \quad (3.2)$$

Ecuación 3.2: Cálculo del *timestamp* relativo al momento de inicio de la repetición.

Ahora, se utilizan las variables presentadas anteriormente (Algoritmo 8), *muestraActual* y *muestraSiguiiente*. Estas dos variables apuntan en todo momento las dos muestras más cercanas al instante temporal actual. Si el intervalo temporal cada vez que se reorientan la cámara y la mirada en este modo fuera constante, bastaría con actualizar estas dos variables de manera uniforme a cada intervalo.

Como no se ha encontrado la forma de hacer esto (una espera temporal en forma de corutina no tiene tanta precisión, ver Figura A.7, Sección A.4 del Anexo A) y se ha calculado en cada *frame*, depende del *hardware*, y este tiempo puede variar.

Por ello hace falta buscar hacia adelante (el tiempo es monótonico creciente) qué muestra temporal  $j$  satisface la Ecuación 3.3:

$$\text{timestamp}_j \leq \text{currentReplayTimestamp} \leq \text{timestamp}_{j+1} \quad (3.3)$$

Ecuación 3.3: Búsqueda de las dos muestras de datos más cercanas en el tiempo al instante actual de la repetición.

Una vez encontrado, ese valor  $j$  se asigna a *muestraActual*. La variable *muestraSiguiiente* tomará el valor  $j + 1$ , correspondiente a la muestra siguiente. Como *currentReplayTi-*

$mestamp$  seguramente será un valor intermedio entre los  $timestamps$  de ambas muestras, calcula un valor  $t$  mediante la Ecuación 3.4:

$$t = \frac{currentReplayTimestamp - timestamp_{muestraActual}}{timestamp_{muestraSiguiente} - timestamp_{muestraActual}} \quad (3.4)$$

Ecuación 3.4: Cálculo del valor  $t$  para realizar la interpolación entre los valores de las muestras.

Por último, se interpola por  $t$  entre los dos valores de rotación de la cámara correspondientes a las muestras  $muestraActual$ -ésima y  $muestraSiguiente$ -ésima. Se realiza otra interpolación idéntica para la posición de la mirada.

#### Procesando información de la mirada

Se ha hablado de rotar la cámara para recrear el *scanpath* seguido por la cabeza de los usuarios. También se ha comentado que se realiza un proceso similar con la posición de la mirada, pero se han omitido deliberadamente los detalles hasta ahora.

Para recrear el *scanpath* seguido por la mirada, se tiene un objeto a una distancia fija de la cámara (10 unidades de Unity). Este objeto representa la orientación de la mirada y es la posición de este objeto la que se recalcula cuando se habla de interpolar la posición de la mirada. Se interpolan los vectores de dirección de la mirada de las muestras  $muestraActual$ -ésima y  $muestraSiguiente$ -ésima por el valor  $t$ , y el resultado se multiplica por la distancia objetivo, es decir, 10. El objeto elegido para representar la mirada de los usuarios es una esfera. Este color irá cambiando hacia rojo, en función de la confianza promedio registrada entre las muestras  $muestraActual$ -ésima y  $muestraSiguiente$ -ésima. Se pueden observar ejemplos de alta y baja confianza en las Figuras 3.12 y 3.13.

### 3. Sistema implementado



Figura 3.12: Captura de una repetición de un *scanpath*. La confianza de la medición del *eye tracker* es alta.

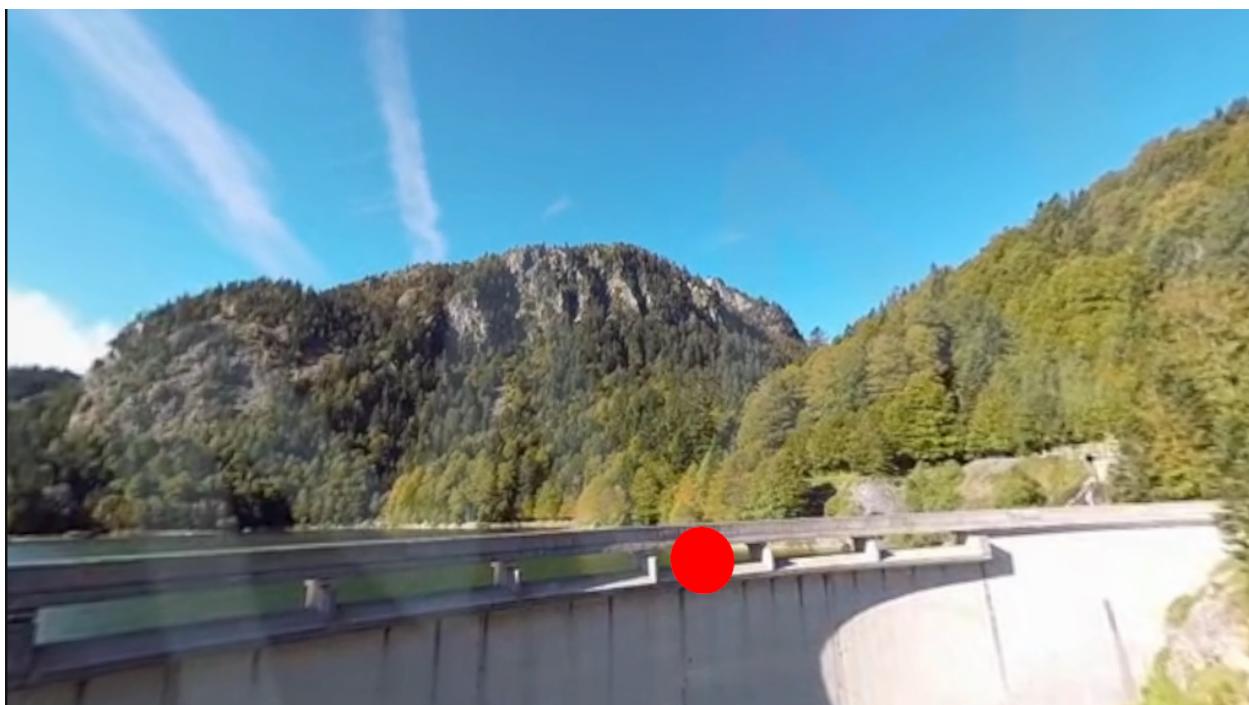


Figura 3.13: Captura de una repetición de un *scanpath*. La confianza de la medición del *eye tracker* es baja.

#### **Resumen de las propiedades y opciones configurables**

Para resumir el funcionamiento descrito para este componente, se enumeran de forma breve las opciones configurables por un usuario a la hora de recrear un *scanpath* guardado

en un fichero:

- Se puede elegir cualquier objeto como el marcador de la posición de la mirada (por defecto una esfera).
- Se puede elegir utilizar únicamente datos de la orientación de la cabeza, o añadir información de la mirada.
- Se pueden cargar múltiples *logs* en la lista. Los *scanpaths* contenidos en cada uno se irán recreando en orden.
- Hay una lista para *logs* de orientación de la cabeza y otra para los correspondientes *logs* de dirección de la mirada.

## 3.4. Generación de mapas de saliencia

La siguiente funcionalidad que se acordó implementar en el *pipeline* es la capacidad de generar mapas de saliencia. Un mapa de saliencia es una imagen con un único canal, donde las regiones con valores elevados indican alta saliencia en las mismas áreas de la imagen o vídeo que han observado los usuarios.

Los mapas de saliencia permiten entender de forma visual qué regiones del entorno atraen más la atención de la gente que lo haya observado. Esto puede resultar muy útil para tareas de investigación, por lo que se considera apropiada la inclusión de esta funcionalidad al *pipeline* desarrollado.

Como en este caso se trata de vídeos, calculan los mapas de saliencia por *frames* del vídeo. Cada *frame* de los vídeos utilizados en este *pipeline* utiliza la proyección equirrectangular (Ver [27] para más información).

Al tratarse de una funcionalidad nueva, se implementó en un componente nuevo, como se ha venido haciendo durante todo el periodo de desarrollo.

### 3.4.1. Componente SaliencyMapsGenerator

El componente en cuestión se llama *SaliencyMapsGenerator*. Se puede ver su aspecto en el editor de Unity en la Figura 3.14. Se detalla su implementación siguiendo paso por paso el proceso de obtención de los mapas de saliencia, pasando por las decisiones tomadas en diseño e implementación, junto a las dificultades encontradas y las soluciones adoptadas.

### 3. Sistema implementado

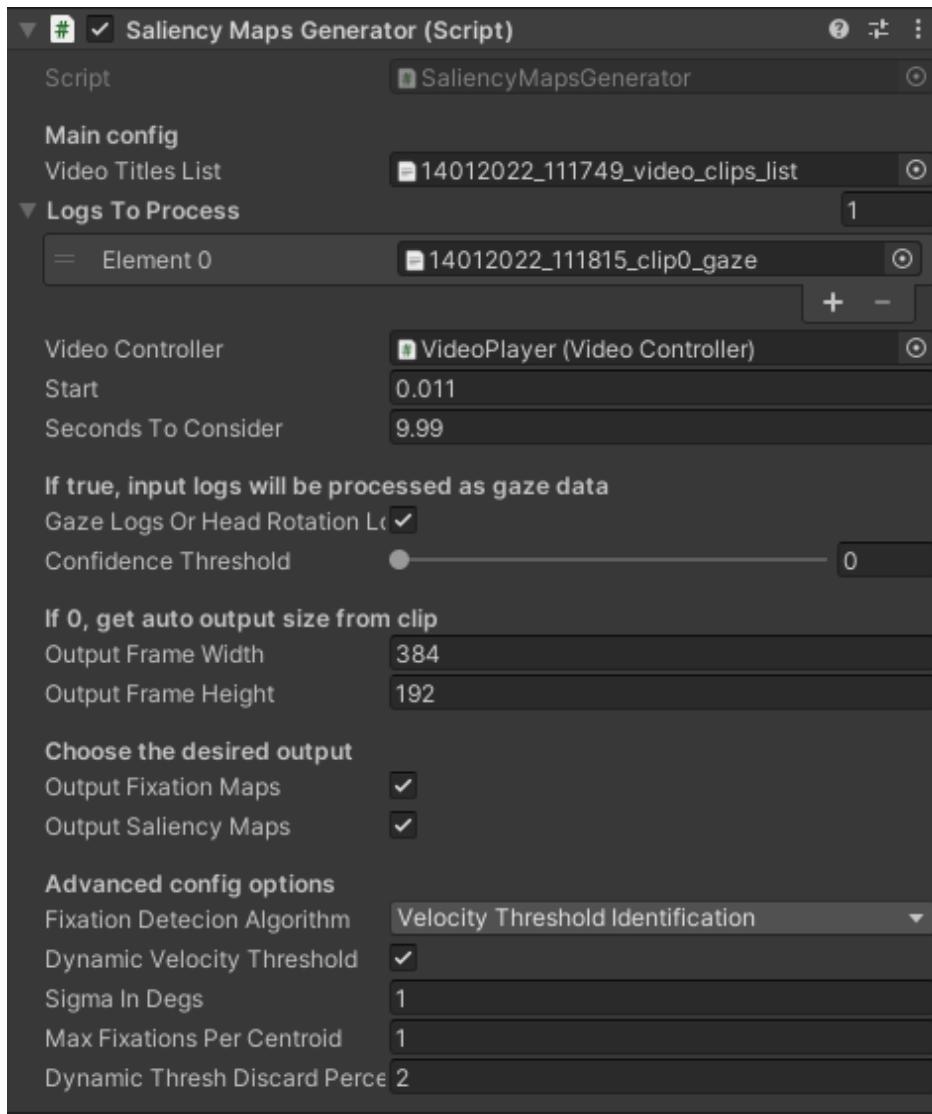


Figura 3.14: Propiedades del componente *SaliencyMapsGenerator* vistas en el editor de Unity.

Al igual que con el componente *ScanpathReplayer* (Sección 3.3.1), se decidió desde el principio que el componente funcionaría en base a los *logs* que se generan durante las recogidas de datos. Se admiten tanto de datos relativos a la orientación de la cabeza de los usuarios como de la dirección de su mirada.

Con todos los *logs* que recibe como argumento se calculan los mapas de fijaciones **promedio** de todos los usuarios para cada *frame*, y a continuación los mapas de saliencia correspondientes.

Las opciones que deben configurarse para hacer uso de este componente se describen a continuación:

- Se debe elegir un conjunto de *logs* que hagan referencia al mismo clip de vídeo.
- Se debe indicar si los logs corresponden todos con datos de la mirada o de la orientación de la cabeza. No se deben mezclar *logs* de ambos tipos.

- Se debe proporcionar el fichero con los títulos de los clips de vídeo ordenados, para obtener el título a partir del ID del nombre del primer *log* de la lista.
- Se debe proporcionar el instante inicial del vídeo a partir del cual calcular mapas de saliencia, así como el periodo de tiempo durante el cual se quieren calcular.
- Se debe especificar un umbral de confianza para descartar o aceptar muestras en caso de que los *logs* contengan datos de la mirada obtenidos con el *eye tracker*.
- Se puede especificar si se quieren o no generar las imágenes con las fijaciones en cada *frame*.
- Se puede especificar si se quieren o no generar las imágenes con la saliencia en cada *frame*.
- Se puede seleccionar el algoritmo deseado para la detección de fijaciones (por el momento solo hay uno implementado pero se soporta la opción de que haya varios).
- Se permite determinar el umbral de velocidad para calcular las fijaciones de forma dinámica, junto con el porcentaje de velocidades más altas que se descarta.
- Se pueden configurar varios parámetros para el cálculo de los mapas de saliencia.
- Se puede configurar el tamaño deseado para los mapas de saliencia y de fijaciones generados (mejor si el ancho y alto son múltiplos directos de la resolución de los *frames* del vídeo).

#### Carga de datos en cachés y operaciones previas

Al igual que con *ScanpathReplayer*, se ha decidido que los datos de los ficheros de *logging* sean cargados en memoria de la misma forma que en el componente mencionado, es decir, se crean diferentes estructuras de datos, una para cada campo que conforma las muestras recogidas en los ficheros. Los detalles de esta implementación ya se han explicado en la Sección C.3 del Anexo C.

Una vez se han cargado en memoria los datos que se van a utilizar (ver Sección C.3 del Anexo C), el siguiente paso es realizar una serie de comprobaciones sobre los parámetros especificados. Se traduce la ventana temporal de la que se desean calcular los mapas de saliencia de segundos a *frames* del clip de vídeo. Esto se consigue multiplicando el *frame-rate* por los tiempos inicial y final especificados.

Primero se comprueba que los frames de inicio y final “pertenezcan” al vídeo asociado a los *logs*, es decir, que se encuentren en el intervalo  $[0..numFramesVideo-1]$ . Seguidamente, se comprueba que la ventana temporal se encuentre completamente contenida entre el primer y el último *timestamp* de todos y cada uno de los *logs* especificados. Si alguno de estos chequeos da error, se informa por la consola al usuario y la ejecución termina.

#### Detección de fijaciones: Algoritmo IVT

El primer paso a realizar en este proceso es identificar las fijaciones visuales realizadas por los usuarios mientras observaban los vídeos. Una fijación visual se define como el mantenimiento de la mirada en una única localización. Otro concepto a tener en cuenta es el de sacada o movimiento sacádico (*saccade* en inglés). Se refiere a movimientos que presentan velocidades intermedias o altas, correspondientes a los usuarios reorientando su atención para fijarse en un punto diferente de su entorno.

Hay varios métodos para la detección de fijaciones. Estos están basados en la dispersión del área observada por los usuarios, en la velocidad de los ojos al examinar la escena, o en otros factores [1]. Como se está tratando con vídeos donde se calcula la saliencia para múltiples *frames*, no tiene sentido utilizar algoritmos que necesiten un parámetro de duración temporal de los movimientos oculares. Por este motivo se descartaron diferentes algoritmos de detección de fijaciones.

Sí que tiene sentido trabajar con algoritmos que tienen en cuenta la velocidad de la mirada en cada instante. Se decidió implementar un algoritmo de este tipo, en concreto el denominado como *I-VT* o *Velocity Threshold Identification*, cuyo funcionamiento general se muestra en la Figura 3.15.

```
I-VT (protocol, velocity threshold)
Calculate point-to-point velocities for each
point in the protocol
Label each point below velocity threshold as
a fixation point, otherwise as a saccade
point
Collapse consecutive fixation points into
fixation groups, removing saccade points
Map each fixation group to a fixation at the
centroid of its points
Return fixations
```

Figura 3.15: Funcionamiento del algoritmo IVT para detección de fijaciones, definido según Salvucci y Goldberg [1].

En este algoritmo se necesitan velocidades angulares como entrada junto con un valor de velocidad (grados por segundo) que actúa como umbral y permite clasificar los movimientos oculares como fijaciones o sacadas. Para cada fichero de *logging* se convierten las  $n$  muestras de rotación y tiempo (los *timesteps* almacenados) en  $n - 1$  datos de velocidad angular, de forma que la velocidad angular  $j$ -ésima está dada por la distancia recorrida entre las muestras  $j$ -ésima y  $j + 1$ -ésima de rotación y tiempo.

Al tratarse de velocidad angular, no sirve calcular la distancia en línea recta entre dos muestras. Se ha optado por utilizar la distancia ortodrómica, aplicando la ley del

semiverseno para facilitar los cálculos [28] [29].

$$\Delta\sigma = 2 \arcsin \sqrt{\sin^2\left(\frac{\Delta\phi}{2}\right) + \cos\phi_1 \cos\phi_2 \sin^2\left(\frac{\Delta\lambda}{2}\right)} \quad (3.5)$$

Ecuación 3.5: Cálculo de la distancia ortodrómica entre dos muestras de datos tras aplicar la ley del semiverseno.

En esta fórmula, mostrada en la Ecuación 3.5,  $\Delta\sigma$  representa la distancia entre dos puntos en la esfera, en grados.  $\Delta\phi$  simboliza la diferencia entre longitudes de las dos muestras tomadas,  $\Delta\lambda$  la diferencia en latitudes, y por último,  $\phi_1$  y  $\phi_2$  son las longitudes de las dos muestras tenidas en cuenta.

Para obtener la velocidad en grados por segundo, se divide esta distancia por el tiempo  $\Delta t$  transcurrido entre los *timestamps* de las muestras,  $t_1$  y  $t_2$  como se explica en las Ecuaciones 3.6 y 3.7.

$$\Delta t = t_2 - t_1 \quad (3.6)$$

Ecuación 3.6: Cálculo del intervalo temporal.

$$\omega = \frac{\Delta\sigma}{\Delta t} \quad (3.7)$$

Ecuación 3.7: Cálculo de la velocidad angular.

En caso de tratar información de *eye tracking*, se rechazan como fijaciones potenciales aquellas muestras con baja confianza. Esto se logra permitiendo que el usuario escoja un umbral de confianza, y se le asigna velocidad infinita a aquellas muestras de velocidad angular donde la confianza promedio de las dos muestras de datos de *eye tracking* sea inferior al umbral establecido.

También se ha añadido como opción activable del componente la posibilidad de calcular el umbral de velocidad de *I-VT* dinámicamente [7]. Este umbral se obtiene descartando el 2 % de velocidades más grandes (primero se descartan las velocidades infinitas) y posteriormente calculando el 20 % de la velocidad máxima que queda. En la implementación realizada se permite que el usuario especifique este porcentaje desde el editor de Unity.

Al tener las velocidades angulares y el umbral de velocidad, se crea una fijación por cada valor de velocidad angular que quede dentro del umbral.

Se ha decidido que los valores que constituyen una fijación son los siguientes:

- Posición en el eje X de la fijación, asumiendo que el *frame* es equirrectangular.
- Posición en el eje Y de la fijación, asumiendo que el *frame* es equirrectangular.

- *Timestamp t* del inicio de la fijación. Este será el de la primera de las dos muestras que se han utilizado para calcular la velocidad angular que se ha clasificado como fijación.
- Duración *d* de la fijación hasta su fin. Es la diferencia entre los dos *timestamps* de las dos muestras de datos que se han utilizado para calcular la velocidad angular que se ha clasificado como fijación.

Como se indica en la Figura 3.15, en cuanto se obtiene una fijación se colapsa junto con las demás que la sigan. Así, al generar la lista de fijaciones final, lo que originalmente eran *n* fijaciones consecutivas en el tiempo, se queda como una única fijación con datos:

$$X = \frac{1}{n} \cdot \sum_{i=1}^n X_i \quad (3.8)$$

Ecuación 3.8: Cálculo de la posición horizontal de la fijación centroide de otras *n* consecutivas.

$$Y = \frac{1}{n} \cdot \sum_{i=1}^n Y_i \quad (3.9)$$

Ecuación 3.9: Cálculo de la posición vertical de la fijación centroide de otras *n* consecutivas.

$$t = \min(timestamp_0, timestamp_1, timestamp_2, \dots, timestamp_n) \quad (3.10)$$

Ecuación 3.10: Cálculo del instante inicial de la fijación promedio de otras *n* consecutivas.

$$d = \max(timestamp_0, timestamp_1, timestamp_2, \dots, timestamp_n) - t \quad (3.11)$$

Ecuación 3.11: Cálculo de la duración de la fijación promedio de otras *n* consecutivas.

Es decir, se obtiene el centroide de todas las fijaciones colapsadas como indican las Ecuaciones 3.8 a 3.11. En ocasiones se colapsan demasiadas fijaciones en una sola, por lo que se obtienen menos fijaciones de las esperadas. Para solventar esto, se permite que los usuarios definan desde el editor de Unity un número límite de puntos de la mirada que pueden colapsar en una sola fijación.

## Mapas de fijaciones

Todo lo explicado hasta ahora tenía como objetivo calcular una lista de fijaciones, con sus instantes iniciales, duraciones y posiciones en la proyección equirrectangular de cada *frame*. A continuación se explica cómo se generan los mapas de saliencia a partir de estos datos.

En primer lugar, se parte de la lista que contiene todas las fijaciones en el intervalo temporal deseado. Para cada *frame* que esté dentro de este intervalo temporal especificado

se obtienen las fijaciones que comiencen, terminen o transcurran durante el *frame* actual. Para esto se utilizan el tiempo de inicio y final de cada *frame*, calculado con la ayuda del *framerate* del vídeo en cuestión.

Para obtener el mapa de fijaciones promedio entre  $N$  observadores a partir de las fijaciones de ese *frame*  $k$ -ésimo, se genera una matriz bidimensional de tanto tamaño como el *frame*, o como haya especificado el usuario. Esta matriz se inicializa a 0 para todos sus valores. Si una fijación ocurre en las coordenadas  $i, j$ , se procede como en la Ecuación 3.12:

$$fixMap_k[i, j] = fixMap_k[i, j] + \frac{1}{N} \quad (3.12)$$

Ecuación 3.12: Cálculo del valor correspondiente a la posición  $i, j$ -ésima del mapa de fijaciones.

De esta forma se obtiene un mapa de fijaciones promedio entre los  $N$  observadores.

### Mapas de saliencia y convoluciones

Para la obtención de mapas de saliencia se debe aplicar una convolución al mapa de fijaciones, utilizando un kernel Gaussiano con una desviación típica  $\sigma$  elegible por el usuario. Normalmente se escogen valores como 1 o 2 grados, puesto que en este caso la desviación típica representa grados del ángulo visual. Estos valores son comunes puesto que representan el tamaño estimado de la fóvea [25].

En lugar de aplicar una convolución con la Gaussiana, se tiene en cuenta la deformación del campo visual en la proyección equirrectangular [27], que se acentúa en los polos de la imagen [2]. Se aplica un factor de escala de  $\frac{1}{\cos \phi}$ , calculado para cada ángulo de elevación  $\phi$  (eje Y), con el fin de “estirar” horizontalmente el kernel Gaussiano isotrópico.

Para cada fila, se tiene el kernel Gaussiano  $K$ , mostrado en la Ecuación 3.13:

$$K = G_{\sigma_y} \cdot G_{\sigma_x}^T \quad (3.13)$$

Ecuación 3.13: Obtención del kernel Gaussiano, teniendo en cuenta la distorsión en los polos.

$G_{\sigma_y}$  es un vector columna que representa una Gaussiana unidimensional con una desviación estándar de  $\sigma_y$  píxeles, mientras que  $G_{\sigma_x}^T$  es un vector fila que representa una Gaussiana unidimensional con  $\sigma_x = \frac{\sigma_y}{\cos \phi}$ . De esta forma se aplica un filtro diferente a cada fila de la matriz que será el mapa de saliencia, como se muestra en la Figura 3.16.

---

```

1: procedure MODIFIEDGAUSSIAN( $F, \sigma$ )
2:    $S \leftarrow \text{zeros}(\text{num\_rows}, \text{num\_cols})$ 
3:   for  $r = 1$  to  $\text{num\_rows}$  do     $\triangleright$  Parallelizable loop
4:      $\phi \leftarrow \pi|r/\text{num\_rows} - 0.5|$ 
5:      $G_{\sigma_y} \leftarrow \text{1d\_gaussian}(\sigma, \text{kernel\_size})$ 
6:      $G_{\sigma_x} \leftarrow \text{1d\_gaussian}(\sigma/\cos\phi, \text{kernel\_size})$ 
7:      $K \leftarrow G_{\sigma_y} \cdot G_{\sigma_x}^T$ 
8:      $S_{\text{row}} \leftarrow F \circledast K \triangleright$  Optimized to only output row  $r$ 
9:      $S(r) = S_{\text{row}}$ 

```

---

Figura 3.16: Convolución con una Gaussiana que tiene en cuenta la distorsión en los polos [2].

En la implementación final se ha utilizado el paquete *OpenCV Plus Unity* para realizar las convoluciones necesarias, así como la conversión final de matriz de datos a imagen.

A continuación se muestra un mapa de fijaciones en la Figura 3.17, acompañado por su correspondiente mapa de saliencia, en la Figura 3.18. Ambos pertenecen al mismo *frame* de vídeo, utilizando datos de 13 observadores diferentes.



Figura 3.17: Mapa de fijaciones generado con *SaliencyMapsGenerator* para un único *frame*.

### **3. Sistema implementado**

---



Figura 3.18: Mapa de saliencia generado con *SaliencyMapsGenerator* para un único *frame*.

En la Sección C.4 del Anexo C se ofrecen detalles acerca de la forma que tiene *SaliencyMapsGenerator* de nombrar y organizar en directorios los mapas de saliencia y fijaciones generados.

# 4. Validación experimental

---

Para demostrar la utilidad del *pipeline* desarrollado como herramienta de recogida de datos en un entorno real de investigación se ha decidido realizar un pequeño experimento donde se recogen datos de diferentes usuarios al visualizar vídeos en 360 grados. A lo largo de esta sección se explican tanto el procedimiento seguido para los experimentos como algunos de los resultados más relevantes obtenidos.

## 4.1. Experimento

Se pidió a diferentes personas que actuaran de observadores en este experimento. Se les explicó que solo tendrían que ver unos pocos vídeos en realidad virtual. En total, se presentaron como voluntarias 13 personas, 4 mujeres y 9 hombres. Para la realización del experimento todos los usuarios debían ponerse unas gafas de realidad virtual, modelo HTC Vive Pro, equipadas con un *eye tracker* fabricado por PupilLabs, como se menciona en la Sección 1.3.2. Adicionalmente, se les pidió que permanecieran de pie durante toda la realización del experimento. Se considera suficiente el espacio del que disponían los observadores a la hora de realizar el experimento, permitiendo suficiente libertad de movimiento para explorar las escenas con libertad.

### 4.1.1. Funcionamiento

Una vez listos para comenzar el experimento, los observadores fueron informados del funcionamiento del experimento. Al comenzar, se encuentran observando una sala blanca, con imágenes de sus ojos vistos desde el *eye tracker*. Esto se hace para comprobar que se tiene visión clara de las pupilas para registrar datos. A continuación se inicia la secuencia de calibración del *eye tracker*. Aparecen una serie de puntos negros en diferentes ubicaciones del campo visual de los usuarios, de uno en uno. Se pidió a los observadores que los siguieran con la mirada según cambiaban de posición. También se les explicó que esto formaba parte del proceso de calibración previo al experimento.

Una vez terminada la calibración, se muestra en mensaje que informa de ello duran-

te unas décimas de segundo. Al cabo de medio segundo se inicia la reproducción de los vídeos. Estos vídeos no tienen sonido, puesto que en este experimento se quiere hacer énfasis en el comportamiento visual de los observadores. Cada vídeo se reproduce durante 20 segundos, puesto que se considera que el tiempo medio para explorar completamente una escena en realidad virtual es de en torno a 18 segundos [30].

Transcurrido este tiempo, el vídeo se detiene y la imagen se vuelve negra. Los observadores deben buscar un cubo rojo, ubicado de forma aleatoria en algún punto alrededor suyo. Una vez encuentran el cubo rojo, se continúa el experimento y se reproduce el siguiente vídeo. El proceso se repite para todos los vídeos que han visualizado.

### 4.1.2. Vídeos mostrados

Se ha utilizado un total de 9 vídeos para la recogida de datos, 8 fueron descargados de Internet, y 1 fue grabado utilizando una cámara propiedad del *Graphics and Imaging Lab*. Se muestran en las Figuras 4.1 a 4.9 capturas representativas del contenido de estos vídeos.



Figura 4.1: Vídeo *Berlin VR360 video in 4K*.

#### 4. Validación experimental

---



Figura 4.2: Vídeo *Kodak 360 4K Video 360 Campus Tour*.



Figura 4.3: Vídeo *Teaser drone VR video 360 dronimages*.

#### 4. Validación experimental

---



Figura 4.4: Vídeo VR 360 VR\_ZenRockShore.

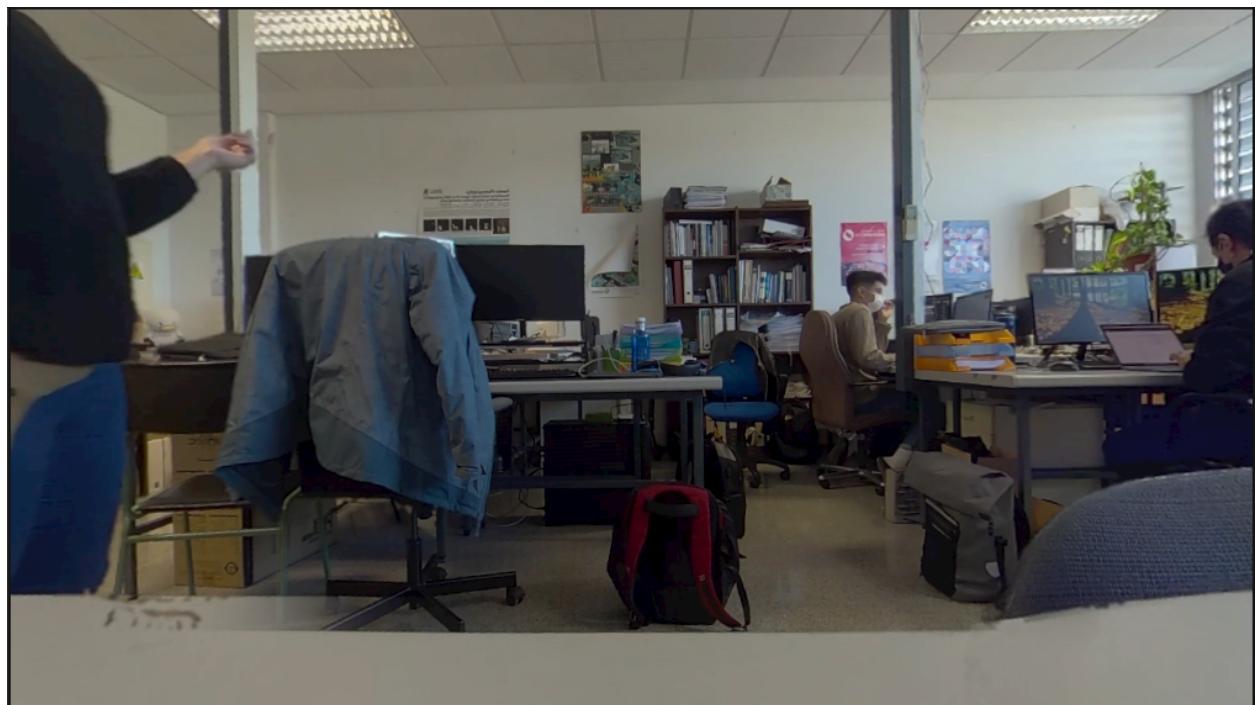


Figura 4.5: Vídeo QueAhoraSi\_FLIP.

#### 4. Validación experimental

---



Figura 4.6: Vídeo *Daniel Ricciardo's Record Pole Lap (360 Video) 2018 Monaco Grand Prix.*



Figura 4.7: Vídeo *Rhino 4K VR 360 Video.*

#### 4. Validación experimental

---



Figura 4.8: Vídeo 360deg RAIK Vekoma Shuttle Roller Coaster Phantasialand 360.



Figura 4.9: Vídeo 360° DRONE VIDEO\_4k.

## 4.2. Resultados obtenidos

Tras ejecutar el experimento de forma satisfactoria para los 13 observadores mencionados anteriormente, se ha obtenido un total de 234 ficheros de *log* con datos relativos a la exploración de los vídeos (13 usuarios, 9 vídeos y 2 *logs* por vídeo, para datos de la cabeza y de la mirada). Se han calculado los mapas de fijaciones y de saliencia para el vídeo *Teaser drone VR video 360 dronimages* utilizando los datos de los 13 usuarios, a modo de ejemplo de la capacidad del *pipeline* para soportar un estudio real con tamaños muestrales significativos. Las especificaciones del equipo en el que se ha llevado a cabo el cálculo de estos mapas de saliencia y fijaciones son las siguientes:

- **Procesador:** Intel i5 10400F @ 2.9 GHz.
- **Memoria RAM:** 16.0 GB DDR4.
- **GPU:** Nvidia GTX 1650 4GB.
- **Sistema Operativo:** Windows 10 Pro.

Se han calculado los mapas de fijaciones y saliencia utilizando los *logs* de orientación de la cabeza, en el intervalo temporal [0.1 - 19.95], expresado en segundos. No se ha utilizado el intervalo [0 - 20] debido a que a menudo la primera muestra de datos tarda unas centésimas de segundo en registrarse, y las últimas muestras de cada *log* suelen tener un *timestamp* ligeramente anterior al tiempo durante el cual se han reproducido los vídeos. Este intervalo temporal se traduce en 597 *frames*, generando tantos mapas de saliencia y de fijaciones como *frames* se han procesado, lo que se traduce un total de 1194 archivos. Los parámetros indicados a *SaliencyMapsGenerator* para el cálculo de estos mapas son los siguientes:

- **Tiempo de inicio:** 0.1 segundos (*frame* 2).
- **Tiempo a considerar:** 19.85 segundos (hasta el *frame* 597).
- **Tamaño de los mapas de saliencia:** Automático, igual a la resolución del vídeo.
- **Detección de fijaciones:** Dinámica, descartando el top 2 % de velocidades.
- **Máximo de posiciones de la mirada a colapsar en una fijación:** 1, no se colapsan.
- **Desviación estándar de la Gaussiana para la convolución:** 1º de ángulo visual.

El tiempo de cálculo para todos estos mapas de fijaciones y saliencia con datos de la orientación de la cabeza ha sido de 7 minutos y 27 segundos.

A continuación se ha repetido el proceso para calcular los mapas de saliencia correspondientes al mismo intervalo temporal del vídeo seleccionado, utilizando los *logs* de información de la mirada. Los parámetros utilizados en *SaliencyMapsGenerator* para el cálculo de estos mapas se muestran a continuación:

- **Tiempo de inicio:** 0.1 segundos (*frame* 2).
- **Tiempo a considerar:** 19.85 segundos (hasta el *frame* 597).
- **Tamaño de los mapas de saliencia:** Automático, igual a la resolución del vídeo.
- **Umbral de confianza para muestras de *eyetracking*:** Establecido en 0.8.
- **Detección de fijaciones:** Dinámica, descartando el top 5 % de velocidades.
- **Máximo de posiciones de la mirada a colapsar en una fijación:** 1, no se colapsan.
- **Desviación estándar de la Gaussiana para la convolución:** 1º de ángulo visual.

El tiempo de cálculo en esta segunda ejecución del componente ha sido de 8 minutos y 18 segundos. Este proceso también ha generado un total de 1194 archivos, 597 mapas de saliencia y 597 de fijaciones.

A continuación se muestran algunos de los mapas de saliencia y de fijaciones más destacables de los 597 *frames* para los que se han calculado. Se explican también las razones por las cuales se consideran relevantes.

## 4. Validación experimental

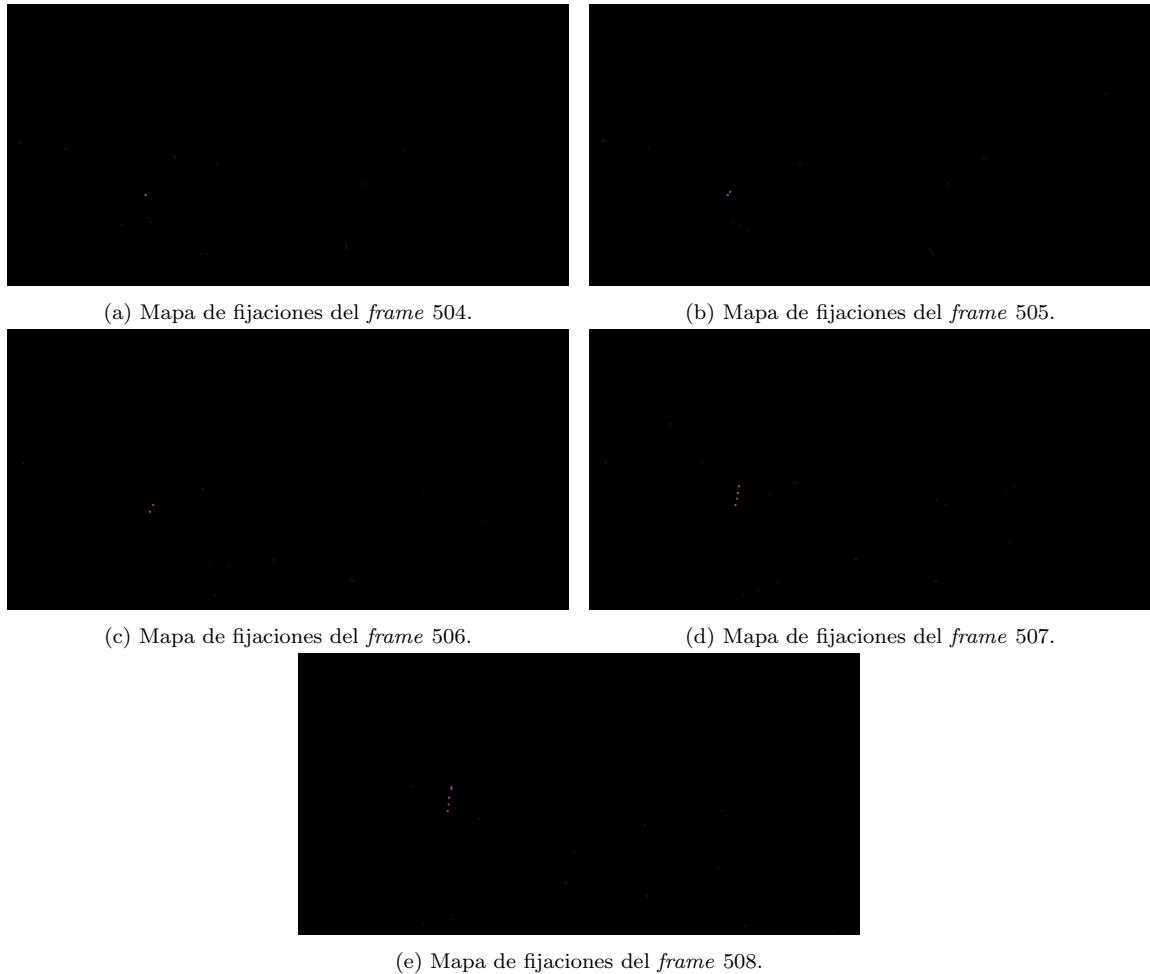


Figura 4.10: Mapas de fijaciones correspondiente a los *frames* 504 a 508 del vídeo *Teaser drone VR video 360 dronimages*, utilizando datos de *eye tracking* para la elaboración del mapa.

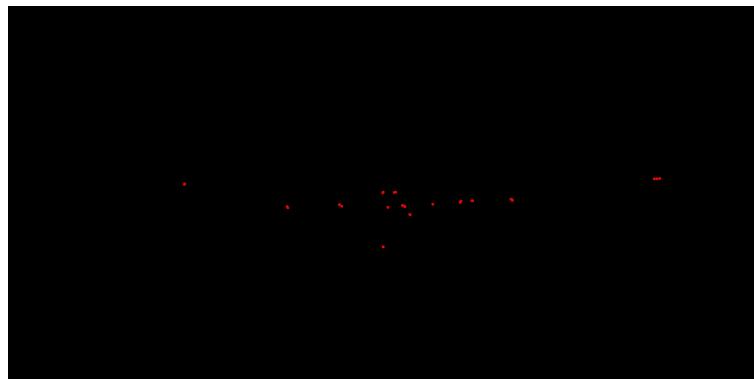
En la Figura 4.10 se muestran los mapas de fijaciones visuales para los fotogramas 504 a 508 del vídeo *Teaser drone VR video 360 dronimages*. Estos mapas de fijaciones se han calculado utilizando los datos obtenidos del proceso de *eye tracking*. Se han coloreado en morado una serie de fijaciones que se consideran interesantes, pues se puede intuir que forman una secuencia de movimiento, probablemente sean parte del *scanpath* seguido por un observador de los que realizaron el experimento. En la Figura 4.11 se visualizan las fijaciones de los 4 *frames* sobre el fotograma 508 del vídeo.

#### 4. Validación experimental

---



Figura 4.11: Frame 508 del vídeo *Teaser drone VR video 360 dronimages* con las fijaciones de las Figuras 4.10a, 4.10b, 4.10c, 4.10d y 4.10e superpuestas.



(a) Mapa de fijaciones correspondiente al *frame* 170.



(b) Mapa de saliencia correspondiente al *frame* 170.

Figura 4.12: Mapas de fijaciones y de saliencia del *frame* 170 del vídeo *Teaser drone VR video 360 dronimages*. Se han calculado utilizando datos de orientación de la cabeza. Se ha resaltado la apariencia de las zonas salientes aumentando la exposición con el fin de mejorar su visibilidad.

## 4. Validación experimental

---

En la Figura 4.12 se muestran los mapas de fijaciones y de saliencia para el mismo vídeo, en esta ocasión en el *frame* 170. Se considera relevante, dado que se puede observar que la mayoría de fijaciones se encuentran concentradas en el área central de la imagen. Al superponer las fijaciones sobre el *frame* del vídeo (Figura 4.13) se entiende mejor el por qué de esta concentración de fijaciones en esta zona central, ya que el vídeo en ese instante es un fondo plano excepto en esta región, donde hay texto. Por tanto, la tendencia natural por parte de los observadores es leer el texto al ser el único elemento que destaca:



Figura 4.13: Fijaciones de la Figura 4.12a superpuestas al *frame* correspondiente del vídeo, que contiene texto en la zona donde se concentran las fijaciones.

# 5. Conclusiones

---

Se ha implementado un *pipeline* de recogida de datos robusto y que presenta varias opciones de configuración útiles para crear diferentes tipos de experimentos, aumentando así sus posibilidades de uso. También permite calcular mapas de fijaciones y saliencia, y ver la “repetición” de cómo los observadores han visto los vídeos que se les han mostrado, para ayudar a comprender los datos recopilados.

Durante todo el proceso de desarrollo se han adquirido conocimientos de todo tipo, relacionados con el uso y funcionamiento del motor Unity, con el estado actual de la VR como tecnología, el conocimiento que se tiene sobre el comportamiento visual humano o el funcionamiento y retos específicos de entender la atención humana en VR. También se han adquirido conocimientos muy valiosos respecto a las métricas más relevantes en la investigación relacionada con la atención humana en VR, como las formas de calcular fijaciones o mapas de saliencia.

Por último, se ha puesto de manifiesto la utilidad del *pipeline* desarrollado realizando a cabo una recogida de datos con usuarios. Estos datos se han procesado con el propio *pipeline* y se han podido interpretar correctamente.

Se han cumplido los objetivos marcados al principio del desarrollo, por lo que el resultado final de este Trabajo se considera muy positivo.

## 5.1. Conclusiones personales

Desde un punto de vista más personal, se considera que el presente Trabajo es el resultado de la acumulación de conocimientos realizada durante los 4 cursos del Grado, ya que se han empleado conceptos procedentes de asignaturas como Informática Gráfica para la lógica espacial, Ingeniería del Software a la hora de planificar el comportamiento del sistema, Programación Concurrente, Visión por Computador para las convoluciones necesarias o incluso de Videojuegos, puesto que se utiliza un motor principalmente orientado a este medio. Además se extrajo de un trabajo realizado en esa asignatura la idea de generar cachés para los ficheros de *log* antes de iniciar la ejecución [31]. En general, se

siente un trabajo completo y multidisciplinar que permite ponerle el cierre a esta etapa de aprendizaje de la mejor forma posible, con unos directores excelentes que han contribuido enormemente en el resultado final.

### 5.2. Trabajo futuro

En cuanto al posible trabajo futuro, se podría mejorar la eficiencia de algunas partes del sistema paralelizando las operaciones, como por ejemplo la generación de mapas de saliencia o la generación de los cachés de datos al inicio de la ejecución. También se ha detectado que en algunos de los experimentos realizados los vídeos se congelaan durante un segundo aproximadamente, pero solo en algunos casos. Se sospecha que tiene que ver con la poca RAM disponible en el sistema ese momento, por circunstancias externas al *pipeline*. No obstante, se podría investigar utilizando las herramientas de *profiling* que ofrece Unity para ver qué está sobrecargando al sistema.

Otra línea de posible trabajo futuro sería incorporar más funcionalidades al *pipeline* como el cálculo de más métricas (Curva *Inter Observer Congruency*, ver *Glosario*) o la realización de chequeos en la capacidad visual de los observadores antes de comenzar los experimentos. Se planteó la implementación de un *Randot test* [32] [33] para este fin pero se descartó por falta de tiempo.

Por último, se considera que sería útil añadir una forma de automatizar la organización de las listas de vídeos y audios que reproducir, dado que de momento es responsabilidad del usuario colocar los vídeos en el orden correcto a la hora de recrear un *scanpath* de un observador. Esto se podría hacer a partir de los ficheros de texto que se mencionan en la Sección 3.3.1.

# Bibliografía

---

- [1] Dario Salvucci and Joseph Goldberg. Identifying fixations and saccades in eye-tracking protocols. pages 71–78, 01 2000.
- [2] Brendan John, Pallavi Raiturkar, Olivier Le Meur, and Eakta Jain. A Benchmark of Four Methods for Generating 360° Saliency Maps from Eye Tracking Data. In *Proceedings of The First IEEE International Conference on Artificial Intelligence and Virtual Reality*, Taichung, Taiwan, December 2018.
- [3] Alexandre BRUCKERT, Yat Hong LAM, Marc CHRISTIE, and Olivier LE MEUR. Deep learning for inter-observer congruency prediction. In *2019 IEEE International Conference on Image Processing (ICIP)*, pages 3766–3770, 2019.
- [4] Wikipedia Realidad Virtual, Historia. <https://es.wikipedia.org/wiki/Realidadvirtual#Historia//>. [Último acceso 2022-01-23].
- [5] Virtual Reality (VR) Market - Growth, Trends, Covid-19 Impact and Forecasts(2022-2027). <https://www.mordorintelligence.com/industry-reports/virtual-reality-market>. [Último acceso 2022-01-23].
- [6] Virtual Reality (VR) Market Size, Share and Covid-19 Impact Analysis, 2021-2028. <https://www.fortunebusinessinsights.com/industry-reports/virtual-reality-market-101378>. [Último acceso 2022-01-23].
- [7] Ana Serrano, Vincent Sitzmann, Jaime Ruiz-Borau, Gordon Wetzstein, Diego Gutierrez, and Belen Masia. Movie editing and cognitive event segmentation in virtual reality video. *ACM Transactions on Graphics (SIGGRAPH 2017)*, 36(4), 2017.
- [8] Unity website. <https://unity.com/es>. [Último acceso 2022-01-23].
- [9] OpenCV Plus Unity en la *Asset Store* de Unity. <https://assetstore.unity.com/packages/tools/integration/opencv-plus-unity-85928>. [Último acceso 2022-01-23].
- [10] Sitio web de las gafas VR HTC Vive Pro. <https://www.vive.com/eu/product/vive-pro/>. [Último acceso 2022-01-23].
- [11] Sitio web del *add on* de PupilLabs para *eyetracking*. <https://docs.pupil-labs.com/vr-ar/htc-vive/>. [Último acceso 2022-01-23].

- [12] Sitio web del *add on* de PupilLabs para *eyetracking*. <https://github.com/pupil-labs/hmd-eyes/blob/master/docs/Developer.md>. [Último acceso 2022-01-23].
- [13] Manual del *plugin* OpenXR. <https://docs.unity3d.com/Packages/com.unity.xr.openxr@1.0/manual/index.html>. [Último acceso 2022-01-23].
- [14] Unity Asset Store, página del *plugin* SteamVR. <https://assetstore.unity.com/packages/tools/integration/steamvr-plugin-32647>. [Último acceso 2022-01-23].
- [15] Christof Koch and Shimon Ullman. Shifts in selective visual attention: towards the underlying neural circuitry. *Human neurobiology*, 4 4:219–27, 1985.
- [16] J.J. Clark and N.J. Ferrier. Modal control of an attentive vision system. In *[1988 Proceedings] Second International Conference on Computer Vision*, pages 514–523, 1988.
- [17] Fang-Yi Chao, Cagri Ozcinar, Lu Zhang, Wassim Hamidouche, Olivier Deforges, and Aljosa Smolic. Towards audio-visual saliency prediction for omnidirectional video with spatial audio. In *2020 IEEE International Conference on Visual Communications and Image Processing (VCIP)*, pages 355–358, 2020.
- [18] Matthias Kümmerer, Lucas Theis, and Matthias Bethge. Deep gaze i: Boosting saliency prediction with feature maps trained on imagenet, 2015.
- [19] Ming Jiang, Shengsheng Huang, Juanyong Duan, and Qi Zhao. Salicon: Saliency in context. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1072–1080, 2015.
- [20] Zhenzhong Chen, Yiming Li, and Yingxue Zhang. Recent advances in omnidirectional video coding for virtual reality: Projection and evaluation. *Signal Process.*, 146:66–78, 2018.
- [21] Ching-Ling Fan and Cheng-Hsin Hsu. Optimizing 360° video streaming to head-mounted virtual reality. In *2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 458–459, 2018.
- [22] Ching-Ling Fan, Jean Lee, Wen-Chih Lo, Chun-Ying Huang, Kuan-Ta Chen, and Cheng-Hsin Hsu. Fixation prediction for 360° video streaming in head-mounted virtual reality. *Proceedings of the 27th Workshop on Network and Operating Systems Support for Digital Audio and Video*, 2017.
- [23] Wen-Chih Lo, Ching-Ling Fan, Shou-Cheng Yen, and Cheng-Hsin Hsu. Performance measurements of 360° video streaming to head-mounted displays over live 4g cellular networks. In *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 205–210, 2017.
- [24] Efficient 360 Video Streaming to Head-Mounted Displays in Virtual and Augment Reality. <https://nmsl.cs.nthu.edu.tw/360-video-project/>. [Último acceso 2022-01-23].

- [25] Olivier Le Meur and Thierry Baccino. Methods for comparing scanpaths and saliency maps: Strengths and weaknesses. *Behavior research methods*, 07 2012.
- [26] Unity docs - Configuring project for XR. <https://docs.unity3d.com/Manual/configuring-project-for-xr.html>. [Último acceso 2022-01-23].
- [27] Wikipedia Equirectangular projection. [https://en.wikipedia.org/wiki/Equirectangular\\_projection](https://en.wikipedia.org/wiki/Equirectangular_projection). [Último acceso 2022-01-27].
- [28] Erwan David, Jesús Gutiérrez, Antoine Coutrot, Matthieu Perreira Da Silva, and Patrick Le Callet. A dataset of head and eye movements for 360° videos. pages 432–437, 06 2018.
- [29] Wikipedia - Ley del semiverseno. [https://es.wikipedia.org/wiki/F%C3%B3rmula\\_del\\_semiverseno](https://es.wikipedia.org/wiki/F%C3%B3rmula_del_semiverseno). [Último acceso 2022-01-24].
- [30] Vincent Sitzmann, Ana Serrano, Amy Pavel, Maneesh Agrawala, Diego Gutierrez, Belen Masia, and Gordon Wetzstein. How do people explore virtual environments?, 2017.
- [31] Github - clon de doom utilizando sfml. <https://github.com/PedroPerez14/DOOM>. [Último acceso 2022-01-28].
- [32] ¿Qué es la estereopsis y cómo evaluarla? <https://optirepresentaciones.com/es/que-es-estereopsis-y-como-evaluarla/>. [Último acceso 2022-01-27].
- [33] H.W. Thimbleby, S. Inglis, and I.H. Witten. Displaying 3d images: algorithms for single-image random-dot stereograms. *Computer*, 27(10):38–48, 1994.
- [34] Guía de inicio de *Resonance Audio*. <https://resonance-audio.github.io/resonance-audio/develop/unity/getting-started>. [Último acceso 2022-01-23].
- [35] Tutorial de configuración de *Resonance Audio* en *Youtube*. [https://www.youtube.com/watch?v=SRPoy2ZAZHg&ab\\_channel=RyanZehm](https://www.youtube.com/watch?v=SRPoy2ZAZHg&ab_channel=RyanZehm). [Último acceso 2022-01-23].
- [36] Guía para desarrolladores de *Resonance Audio*. <https://resonance-audio.github.io/resonance-audio/develop/unity/developer-guide>. [Último acceso 2022-01-23].
- [37] VR Audio, differences between A format and B format. <https://postperspective.com/vr-audio-differences-format-b-format/>. [Último acceso 2022-01-23].
- [38] HOA TECHNICAL NOTES - SN3D B-FORMAT. <https://www.blueripplesound.com/b-format>. [Último acceso 2022-01-23].

# Anexo A. El motor Unity

---

Unity es un motor de desarrollo de aplicaciones multimedia, principalmente orientado a videojuegos, pero sus usos también se extienden a arquitectura, demostraciones industriales, o en menor medida, investigación. Dispone de opciones para crear proyectos en 3D y 2D, y da soporte a diferentes sistemas de realidad virtual como, por ejemplo, las gafas Oculus Rift. Como la mayoría de motores de desarrollo multipropósito, para programar el comportamiento de un objeto se necesita utilizar *scripts* que se asignan a los objetos como componentes de estos.



Figura A.1: Logotipo de Unity desde 2021

Unity permite que este código esté escrito en C#, o en otros lenguajes basados en .NET, con tal de que sean capaces de crear un archivo .dll compatible, y de que se genere manualmente código de envoltura en C# para ese archivo.

Como es natural, se ha escogido la primera opción ya que aunque no se tenían apenas conocimientos previos del lenguaje, es mucho más simple que preparar todo lo requerido por la segunda opción, además de que la curva de aprendizaje se ha visto reducida gracias a tener conocimientos previos de lenguajes similares como Java o C++.

Al tratarse de un motor completo que da soporte en tareas áreas como la reproducción de vídeos 360, la elección de utilizarlo permite ahorrar tiempo de desarrollo que habría que invertir en programar una cámara 3D, un renderizador, o soporte para audio ambisónico y realidad virtual. Después de todo, estos componentes son necesarios para la herramienta final pero se quieren centrar los esfuerzos en la implementación de una herramienta de recogida de datos, por lo que se considera conveniente utilizar un motor que brinde soporte en estas tareas.

## A.1. El editor de Unity

A lo largo de la presente memoria se van a mencionar conceptos como escena, objeto, componente y propiedad. Por ello, se considera apropiado explicar en esta sección algunas de las nociones básicas del motor, así como de su editor, puesto que el producto final está planteado para ser utilizado desde el editor de Unity. En la imagen a continuación se resaltan algunos de los elementos más interesantes de su interfaz.

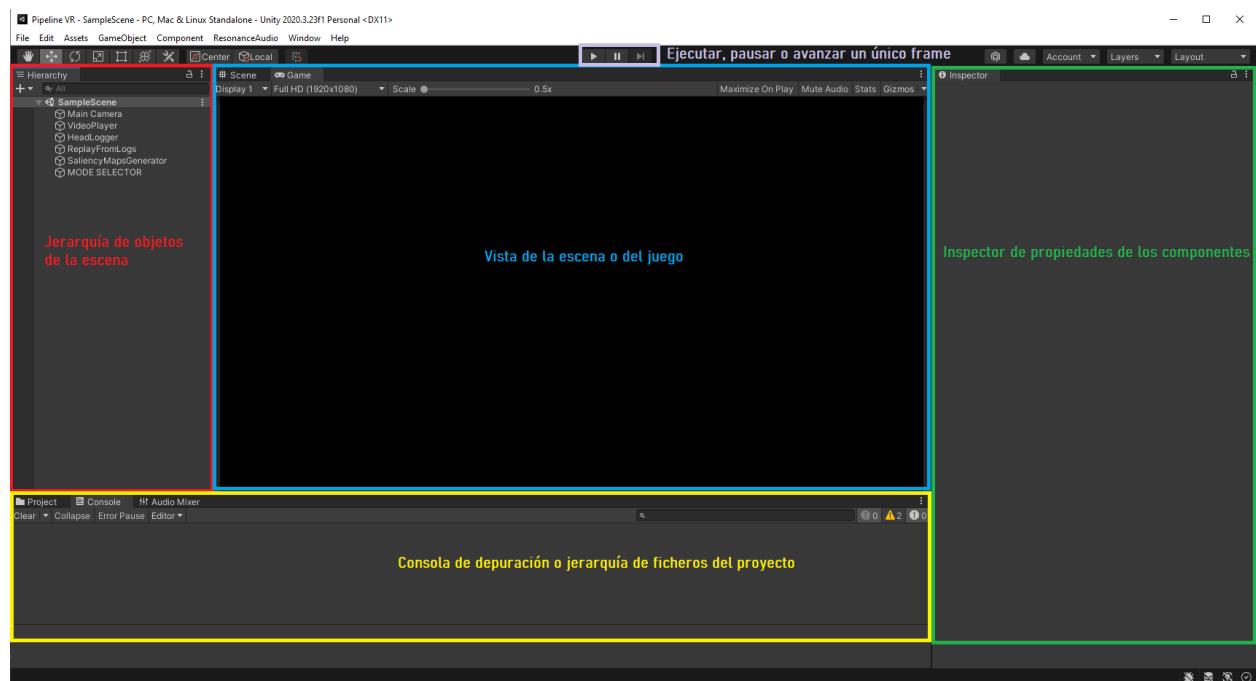


Figura A.2: Distribución de la interfaz del editor de Unity con sus elementos más importantes.

En primer lugar, en color amarillo se encuentra resaltada la pestaña que contiene la organización del proyecto, y permite navegar entre sus carpetas y archivos, así como consultar sus propiedades. Por defecto, los recursos que formen parte de un proyecto (texturas, *shaders*, *scripts*, etc.) se encuentran en una carpeta llamada *Assets*, en la raíz del proyecto. También tiene otra pestaña a su lado, la consola de mensajes. Aquí se podrán leer todos los mensajes de *debug* que se quieran generar durante la ejecución. La herramienta final informará a los usuarios de los eventos más importantes o errores por aquí.

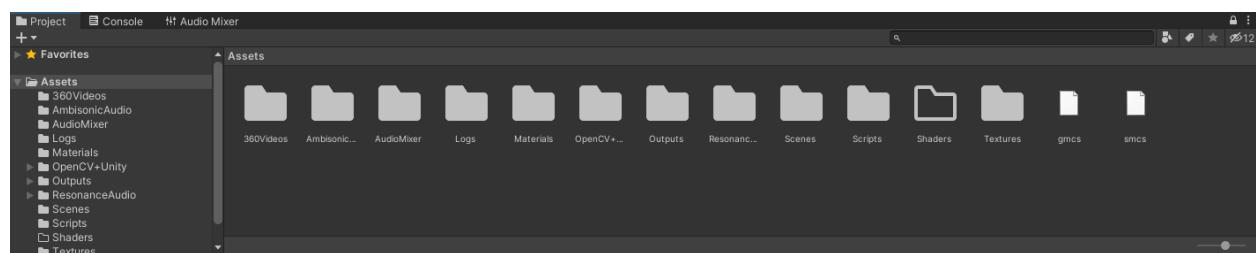


Figura A.3: Pestaña de organización de un proyecto en el editor de Unity.

A continuación, en azul se encuentran remarcadas las pestañas que contienen la vista de la escena y del juego. Una escena en Unity es un conjunto de objetos que van a interactuar entre sí en algún momento del tiempo, de forma independiente a cualquier otro objeto de cualquier otra escena que pueda tener el proyecto. La vista de escena permite visualizar y navegar por el entorno 2D o 3D que se esté creando, así como modificar de forma interactiva las propiedades (posición, escalado, rotación, etc) de los objetos que la conforman. Toda escena en Unity necesita tener por lo menos una cámara principal para renderizar y mostrar algo por pantalla, por este motivo, al crear una escena, aparecen como objetos predeterminados una cámara y una fuente de luz direccional. La vista de juego se utiliza para visualizar la ejecución del programa, desde el punto de vista de la cámara que esté renderizando en ese momento la escena.

En violeta se han resaltado los botones de lanzamiento y pausado de la aplicación. De izquierda a derecha, estos tres botones sirven para lanzar y detener la aplicación, para pausarla de forma temporal hasta que se vuelve a pulsar, y para ejecutar únicamente un *frame* si se ha pausado previamente la ejecución. Cada vez que se inicie la ejecución se cambiará automáticamente de la pestaña de la escena a la de juego, para visualizar la ejecución.

A la izquierda y en rojo aparece la pestaña de jerarquía de la escena. Ahora que se ha explicado que una escena es un conjunto de objetos que interactúan entre sí, se puede especificar que es en esta pestaña de jerarquía donde se puede consultar cuáles son esos objetos que componen la escena. También se pueden definir relaciones de padre e hijo entre los objetos, esto permite, entre otras cosas, que la posición tridimensional de un objeto hijo sea relativa a la posición del padre, en lugar de absoluta, es decir, con respecto al mundo.

Por último, al hacer clic en cualquiera de los objetos de una escena se podrán previsualizar sus componentes asociados, que son las que realmente definen su comportamiento. Un componente puede ser, entre otros, un reproductor de vídeo, un receptor de audio, o uno de entre varios comportamientos predefinidos que ofrece el motor. Aparte de esto, se pueden crear *scripts* de código C# desde esta pestaña o desde la de organización de archivos del proyecto, para asociarlos al objeto como componentes personalizados que definen un comportamiento para ese objeto. Por defecto todos los objetos tienen un componente llamado *Transform* que hace referencia a las propiedades espaciales básicas de ese objeto, como su posición, su rotación expresada en ángulos eulerianos, o su escala de tamaño. Estos componentes se pueden previsualizar en la parte resaltada en verde en la figura A.2, la pestaña *Inspector*.

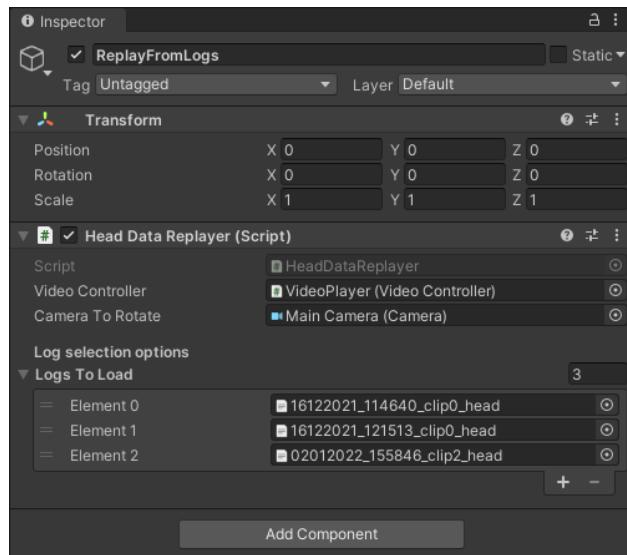


Figura A.4: Inspector de Unity, mostrando los componentes *Transform* y un *script* pertenecientes a un objeto junto a sus propiedades.

## A.2. Coordenadas tridimensionales de Unity

Ya que hablar de realidad virtual es hablar de entornos tridimensionales, se va a aclarar qué sistema tridimensional utiliza Unity, puesto que no hay un estándar fijo en la industria acerca de la organización de los ejes, de cuál es X, si Y apunta hacia arriba o hacia abajo, o de si Z debería ser X. Asumir un estándar puede dar lugar a malentendidos o errores en caso de querer utilizar o expandir la herramienta aquí desarrollada.



Figura A.5: Sistemas de coordenadas de varios motores y herramientas tridimensionales populares.

El sistema de coordenadas de Unity sigue una representación cartesiana, y en caso de ser un proyecto 3D como el que aquí se va a discutir, constará de tres ejes; X, Y y Z. La distribución de estos sigue la denominada **regla de la mano izquierda**, donde un valor positivo del eje Z sería “adelante”, una X positiva se traduce por “derecha” y una Y positiva significa “arriba”.

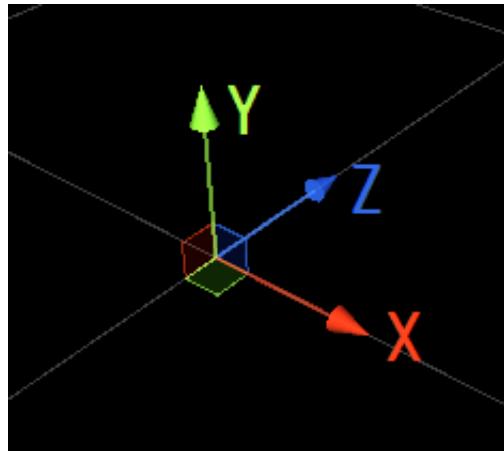


Figura A.6: Sistema de coordenadas de Unity representado mediante los ejes X, Y y Z.

### A.3. Funciones y métodos de interés de Unity

A continuación se presentan algunos métodos propios de los objetos de Unity que se han considerado interesantes, ya que debido a sus características se han tenido que usar, descartar o adaptar para la creación de los diferentes componentes de la herramienta desarrollada.

Al programar el comportamiento de un objeto en Unity, es necesario crear un *script*. Este deberá heredar de la clase *MonoBehaviour*, y suelen implementar alguno de estos dos métodos:

- **Start:** Se trata de una función de inicialización presente en todos los objetos cuando son creados, se llama una vez para cada objeto de la escena antes de empezar a calcular el primer frame. Si es muy pesada puede repercutir en retrasos en el inicio de la ejecución de la aplicación.
- **Update:** Aquí se definen las tareas que se ejecutan una vez por cada *frame*, es decir, se ejecuta tantas veces en un segundo como la tasa de refresco de la aplicación, por lo que no presenta una frecuencia fija, sino que depende de la carga de trabajo puntual y del *hardware* donde se ejecute la aplicación. Se suele usar para actualizar el aspecto visual de la escena y sus objetos.
- **FixedUpdate:** Es algo menos común, y presenta un comportamiento similar a *Update*. En este caso, aquí se deberán realizar cálculos para actualizar componentes relacionadas con las físicas, ya que tiene una frecuencia de ejecución igual a la del motor de físicas que incorpora Unity para proyectos 3D, NVidia PhysX. Aún así, no se garantiza la periodicidad de su ejecución en caso de que los cálculos de un *frame* sean demasiado pesados.

## A.4. Esperas en corrutinas de Unity

A lo largo de la memoria se menciona el uso de corrutinas para realizar la mayoría de tareas que necesitan ejecución en paralelo, o que implican una espera temporal. Una particularidad de las esperas en forma de corrutina es que no tienen en cuenta el tiempo que ha transcurrido durante el cálculo del *frame* actual hasta el momento de su invocación, por lo que si ese *frame* en concreto conlleva cálculos muy pesados, el tiempo asociado a esos cálculos no se tiene en cuenta para la espera.

Otro problema de las esperas en forma de corrutina es que debido a su comportamiento, no se despiertan en el preciso instante en el que una espera termina, sino en el *frame* siguiente a que esta termine, incurriendo en un pequeño error temporal asociado al *framerate* y al *hardware* donde se esté ejecutando. En la Figura A.7) se muestra un ejemplo de este tipo de imprecisiones. Esto no supone demasiado problema si las esperas son de un orden de magnitud elevado (segundos), pero se habrían evitado en caso de que fuera crítico garantizar esperas de tiempo exactas o de cantidades más pequeñas de tiempo.

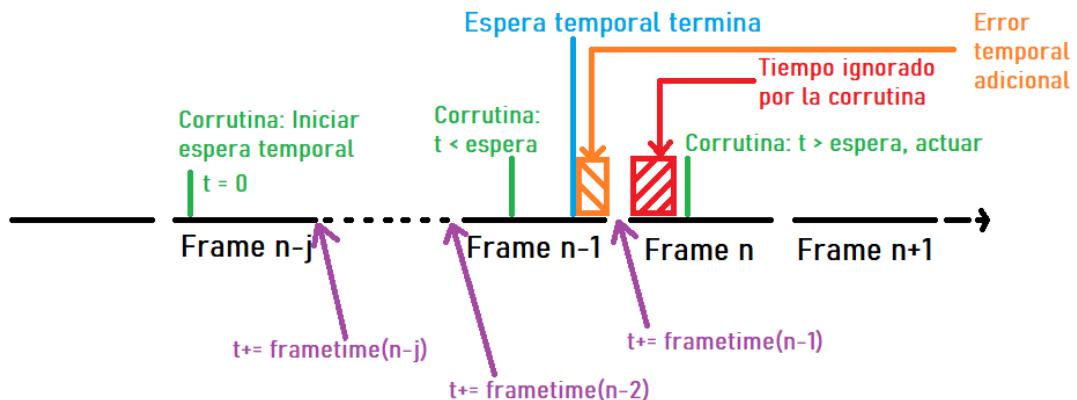


Figura A.7: Ejemplo de una corrutina en Unity, con imprecisiones temporales derivadas de otros cálculos en el *frame* (en rojo), y de que no se ejecutan en el preciso instante en el que la espera termina (en naranja).

Se han mantenido las esperas mediante corrutinas porque se considera admisible un error de unas centésimas de segundo, y porque la alternativa sería programar utilizando *Threads*, lo cual es complicado ya que Unity no permite el acceso a muchos de los objetos y propiedades desde un *thread* externo al principal del motor.

# Anexo B. Trabajo previo: Familiarización con Unity, vídeo 360 y audio ambisónico

---

Aunque no se considere trabajo directamente presente en el producto final, sin este paso previo no se podría haber hecho ninguno de los demás, por lo que se considera de suma importancia en el desarrollo. Durante este proceso de adaptación se adquirieron los conocimientos fundamentales sobre el uso del motor y el lenguaje C# que permitieron el desarrollo del producto final.

## B.1. Reproduciendo vídeo 360 en Unity

Una de las primeras tareas encomendadas fue realizar el trabajo descrito en el enunciado de la primera práctica de la asignatura *Virtual Reality* del máster en *Robotics, Graphics and Computer Vision* de la propia Universidad, y una vez realizado, buscar la forma de expandirlo haciendo que se reproduzca vídeo en 360 grados, intentando que la resolución sea la mayor posible, ya que se comentó que era un problema frecuente con Unity. Se puede encontrar el enunciado de esta práctica adjunto en el Anexo D.

La solución al problema de mostrar una imagen en 360 grados se consiguió siguiendo el enunciado, a la vez que se ganaban conocimientos sobre cómo funciona el propio Unity. Para reproducir vídeo, se realizaron los siguientes pasos:

1. Se crea un nuevo material, y se le asigna un shader de tipo *Skybox/Panoramic*.
2. Se crea una *RenderTexture* y se le da la resolución correspondiente al vídeo que se quiere reproducir.
3. En un objeto cualquiera de la escena, preferiblemente uno sin otros componentes, se crea un componente de tipo *VideoPlayer*.
4. En las opciones de este componente, se puede elegir el vídeo que sirve de entrada y hacia dónde se va a renderizar, se seleccionan para estas opciones el clip de vídeo deseado y la textura que se ha creado en el paso 2.

## B. Trabajo previo: Familiarización con Unity, vídeo 360 y audio ambisónico

5. En las opciones del material que se ha creado en el primer paso se puede elegir una textura asociada bajo la opción *Spherical (HDR)*, se elige la *RenderTexture* creada en el paso 2.
6. En el menú *Window/Rendering/Lighting/Environment* se fija como *Skybox Material* el material creado durante el primer paso.

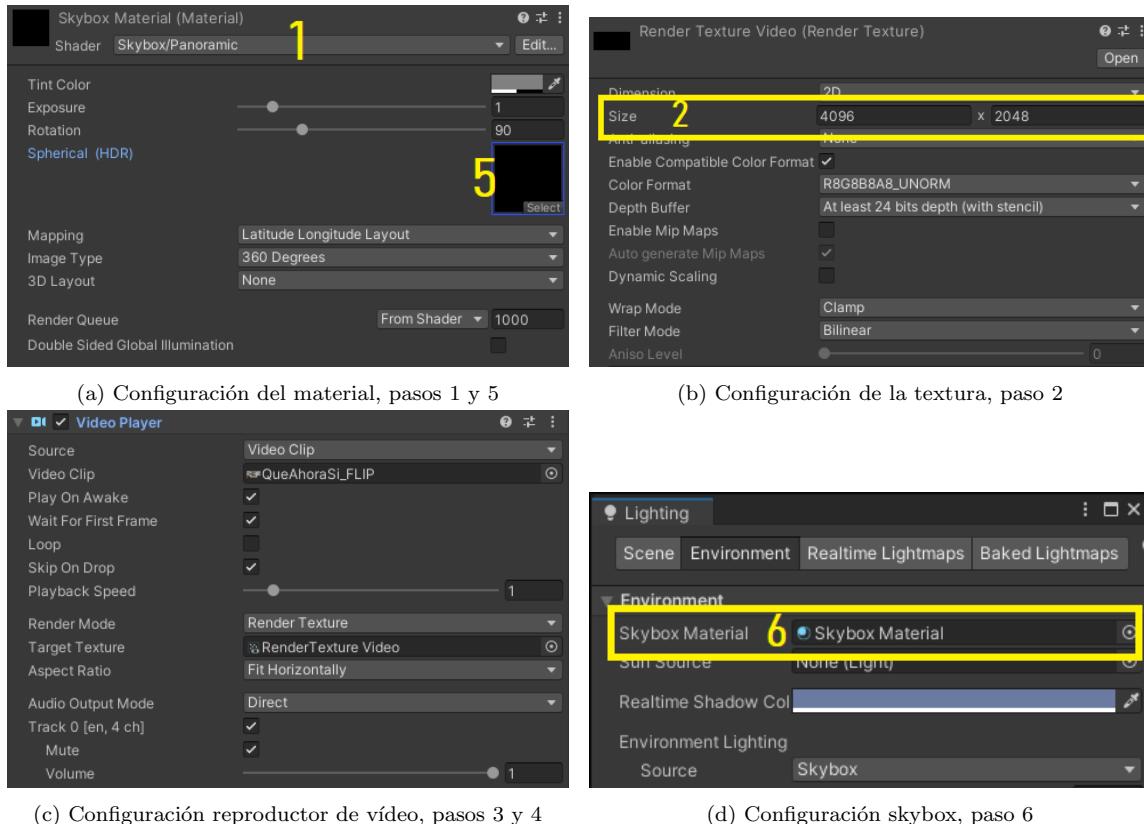


Figura B.1: Configuración básica para reproducir vídeos 360º en formato equirectangular en Unity

De esta forma, estamos consiguiendo que el vídeo se renderice en un punto en el infinito como si fuera el cielo de la escena a través de una textura, en lugar de renderizarse en una esfera unitaria tridimensional, como se hacía con las imágenes de la práctica seguida. De esta forma, se evitan pequeñas aberraciones causadas por el mapeo *uv* de los *frames* sobre la malla de triángulos que compone la esfera. Se muestra un ejemplo a continuación, en la Figura B.2.

## B.2. Audio ambisónico en Unity

El audio ambisónico es aquel donde el sonido se almacena en canales diferentes (por lo general en 4 o más) permitiendo simular la espacialización de este, es decir, que el usuario lo percibe de formas diferentes dependiendo de dónde se encuentre respecto a la fuente del sonido, ya sea a su izquierda, a su derecha, delante suyo, detrás suyo, encima, debajo,

## B. Trabajo previo: Familiarización con Unity, vídeo 360 y audio ambisónico

---

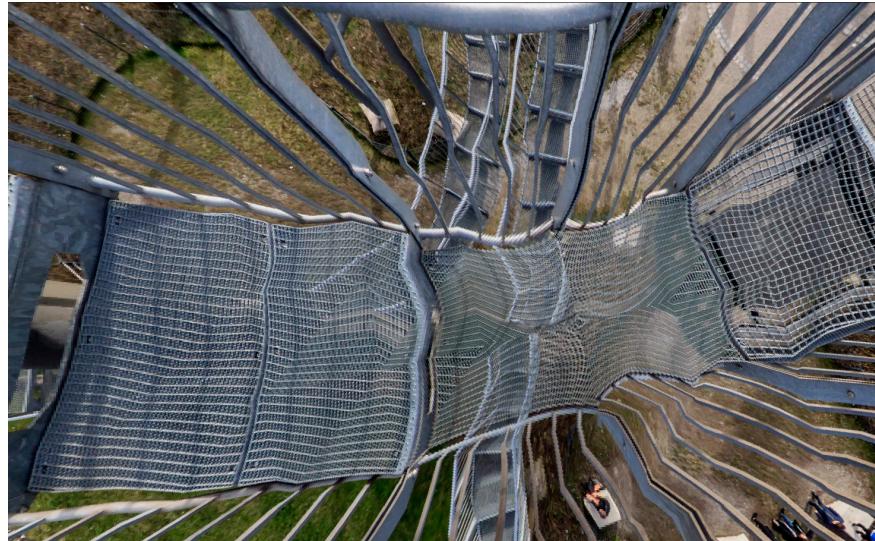


Figura B.2: Distorsión en una imagen 360 al mapearse sobre una esfera con pocos triángulos en su malla poligonal.

cerca o lejos.

Como se puede imaginar, este tipo de audio ayuda a ofrecer una experiencia mucho más inmersiva en contenidos audiovisuales en realidad virtual, donde el usuario tiene control absoluto sobre la orientación, y a veces sobre la posición desde la que se percibe el entorno a su alrededor. Este es el principal motivo por el cual se ha considerado el audio ambisónico para este trabajo.

Tratar con este tipo de audio es más complicado que con audio estándar ya que requiere calcular la posición relativa del usuario (receptor de los sonidos) respecto al componente que reproduzca el audio, para dar más peso a unos canales de audio que a otros, creando la sensación de espacialización.

A la hora de preparar el proyecto de Unity para que pudiera reproducir este tipo de audio se encontró que no se le da soporte de forma nativa. Para poder trabajar con audio ambisónico se debe utilizar uno de los varios plugins fácilmente configurables que existen para Unity. Los dos principales son *Resonance Audio*, desarrollado por Google y el que ofrece *Oculus*, *OculusSpatializer*. El primero debía descargarse aparte y el segundo viene con Unity, pero no está activado.

Se optó por instalar y configurar el primero de los dos [34] [35] [36], en caso de que el desarrollado por *Oculus* estuviera diseñado para funcionar únicamente con sus dispositivos, buscando la mayor compatibilidad posible.

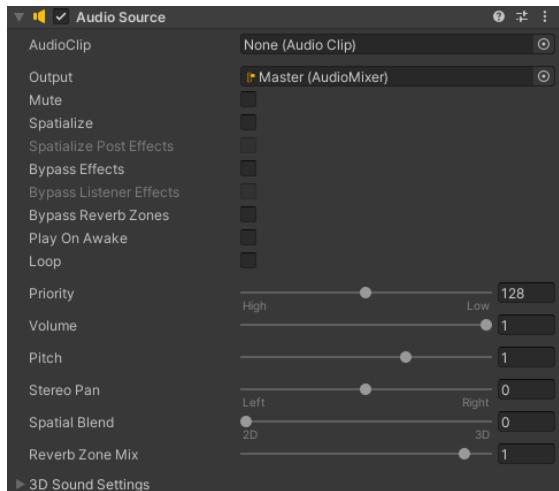
Para conseguir percibir el audio ambisónico de manera correcta, se siguieron los siguientes pasos:

1. En la cámara principal de la escena se añade un componente de tipo *AudioListener*, en este caso asumimos que la cámara representará la cabeza de los usuarios.
2. Se le puede añadir opcionalmente un componente de tipo *ResonanceAudioListener*, se ha descartado para no hacer la configuración muy dependiente del *plugin*.
3. Se crea un objeto en la escena o se elige uno ya existente, al que se le añaden dos

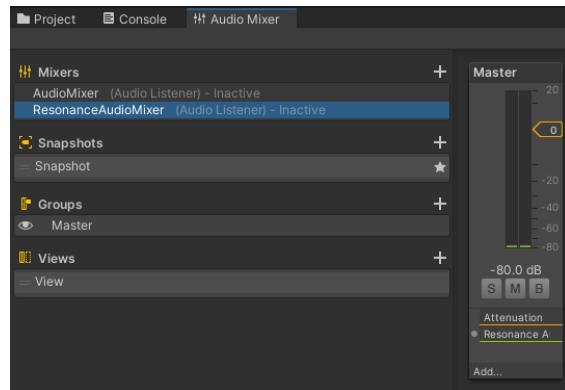
## B. Trabajo previo: Familiarización con Unity, vídeo 360 y audio ambisónico

componentes de tipo  *AudioSource* y  *Resonance AudioSource*.

4. En una carpeta del proyecto se debe crear un  *AudioMixer*, un mezclador de audio al que se le añade un  *ResonanceAudioRenderer* en la pista  *Master*.
5. Volviendo al  *AudioSource* creado en el paso 3, se configuran las opciones  *AudioClip* y  *Output*, eligiendo el audio ambisónico que se quiera reproducir para la primera, y en la segunda se deberá elegir la salida  *Master* del mezclador de audio creado en el paso anterior.



(a) Configuración del componente  *AudioSource*



(b) Configuración del mezclador de audio

Figura B.3: Configuración básica para reproducir audio ambisónico en Unity. En algunas guías se explica que hay que activar la opción  *Spatialize*, pero es únicamente necesario para el audio normal, con audio ambisónico no hay que activarla, ni es necesario tocar el  *slider* de la opción  *SpatialBlend*.

Durante este proceso de adaptación se descubrió que al importar las pistas de audio ambisónico a Unity, se debía indicar de manera explícita que estas son audio ambisónico para que el motor las trate como tal. Además, atendiendo a la documentación de Unity sobre el tema, el clip importado deberá estar en formato *.wav*, deberá tener compresión PCM con el audio codificado en formato B [37], ordenación de componentes ACN y normalización de tipo sn3d [38], por lo que se deberá tener esto en cuenta si se quieren cargar audios propios para cualquier experimento.

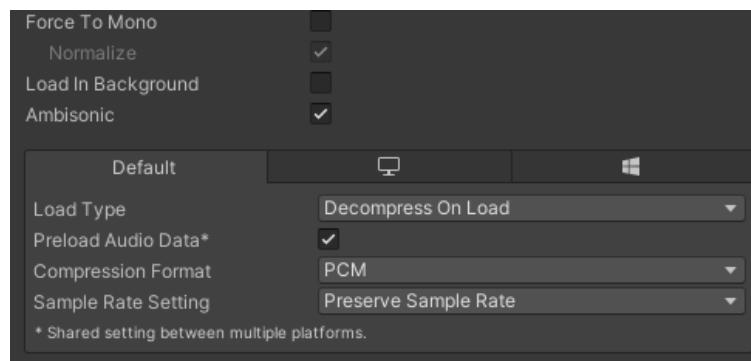


Figura B.4: Opciones para importar un clip de audio ambisónico a Unity.

# Anexo C. Logs y archivos generados

---

## C.1. Ficheros con datos de orientación de la cabeza

Los archivos *.csv* generados contienen en su primera línea el momento del tiempo del vídeo donde se comenzaron a registrar datos, expresado en segundos. La segunda línea contiene la rotación de la cámara (se considera equivalente a la cabeza del usuario) en el momento de comenzar a registrar datos, expresado como cuaternión. A partir de la cuarta línea, cada una corresponde a una muestra de la rotación de la cabeza. Cada una contiene los siguientes campos en orden, tal y como se definen en la tercera línea del fichero:

- **timestamp**: Instante temporal de esa muestra, contando desde que se ha comenzado a recoger información sobre la visualización del vídeo. Se expresa en segundos.
- **EulerRotationXYZ**: Rotación de la cabeza del usuario en ese instante del tiempo, expresado en ángulos eulerianos. Es un vector tridimensional.
- **QuaternionRot**: Rotación de la cabeza del usuario en ese instante del tiempo, expresado en cuaterniones. Es un vector de cuatro dimensiones.
- **forwardXYZ**: Vector tridimensional normalizado *forward* de la cámara (cabeza del usuario), representativo de la dirección hacia la que se está orientando en ese instante del tiempo.
- **UV\_range01**: Coordenadas UV normalizadas correspondientes a la orientación de la cabeza, proyectadas desde una imagen esférica (360) a un plano rectangular. Es un vector bidimensional. Para más información acerca de la proyección equirrectangular, se puede consultar [27].

## C. Logs y archivos generados

startingTimestamp	0			
InitialRotationQuaternion	(0.2806297,-0.06070576,-0.009054149,-0.9578517)			
timestamp	EulerRotationXYZ	QuaternionRot	forwardXYZ	UV_range01
0.00997509765625	(327.1878,7.811149,358.8591)	(0.2824239,-0.06253044,-0.009708636,-0.9572003)	(0.1142244,0.5418866,0.8326534)	(0.5216976,0.3177102)
0.010971435546875	(327.1878,7.811149,358.8591)	(0.2824239,-0.06253044,-0.009708636,-0.9572003)	(0.1142244,0.5418866,0.8326534)	(0.5216976,0.3177102)
0.019947998046875	(326.9762,8.017927,358.8445)	(0.2841804,-0.06416717,-0.0102253,-0.9565665)	(0.1169487,0.5449872,0.8302481)	(0.522272,0.3165345)
0.021942138671875	(326.9762,8.017927,358.8445)	(0.2841804,-0.06416717,-0.0102253,-0.9565665)	(0.1169487,0.5449872,0.8302481)	(0.522272,0.3165345)
0.021942138671875	(326.9762,8.017927,358.8445)	(0.2841804,-0.06416717,-0.0102253,-0.9565665)	(0.1169487,0.5449872,0.8302481)	(0.522272,0.3165345)
0.031916015625	(326.8917,8.194638,358.8643)	(0.2848583,-0.06566914,-0.01088081,-0.9562557)	(0.119394,0.5462239,0.8290865)	(0.5227629,0.3160648)
0.032912109375	(326.8917,8.194638,358.8643)	(0.2848583,-0.06566914,-0.01088081,-0.9562557)	(0.119394,0.5462239,0.8290865)	(0.5227629,0.3160648)
0.0422886474609375	(326.8117,8.396977,358.8526)	(0.285512,-0.06730734,-0.01133747,-0.9559415)	(0.1222098,0.5473917,0.8279052)	(0.523325,0.3156208)
0.053858154296875	(326.6603,8.629647,358.7983)	(0.2867834,-0.06907126,-0.01156371,-0.9554322)	(0.1253532,0.5496017,0.8259689)	(0.5239713,0.3147795)
0.055851318359375	(326.6603,8.629647,358.7983)	(0.2867834,-0.06907126,-0.01156371,-0.9554322)	(0.1253532,0.5496017,0.8259689)	(0.5239713,0.3147795)
0.056850830078125	(326.6603,8.629647,358.7983)	(0.2867834,-0.06907126,-0.01156371,-0.9554322)	(0.1253532,0.5496017,0.8259689)	(0.5239713,0.3147795)
0.06582666015625	(326.4319,8.824543,358.7543)	(0.2886931,-0.07052121,-0.01183767,-0.9547476)	(0.127825,0.5529277,0.8233661)	(0.5245126,0.3135106)
0.076796630859375	(326.1306,9.018705,358.6815)	(0.2912258,-0.0718665,-0.01192655,-0.9538766)	(0.1301569,0.5573012,0.8200455)	(0.525052,0.3118369)

Figura C.1: Aspecto de los campos de un *log* al abrirlo con Excel.

Se ha decidido que la información recogida sea redundante, debido a que dependiendo de lo que se quiera hacer con estos *logs*, puede ser más conveniente utilizar una u otra. Además, se pueden detectar errores en la lectura de los valores si al pasar de un tipo a otro (por ejemplo, de coordenadas *uv* a cuaternión) los valores no concuerdan.

El nombre de los archivos de *log* generados sigue el siguiente formato:

**ddMMyyyy\_HHmmss\_clipIDClip\_head.csv**

Figura C.2: Formato del nombre de los *logs* de datos de orientación de la cabeza.

La primera parte hace referencia a la fecha y hora del momento de su creación, mientras que *IDClip* hace referencia al número de posición que ocupa el vídeo asociado al fichero en la lista de vídeos del componente *videoController*, explicado en la Sección 3.1. La última parte indica que el fichero contiene datos de la cabeza, para diferenciarlo de los ficheros con información sobre la mirada, cuyo contenido y proceso de recogida de datos se explica en la Sección 3.2.2.

## C.2. Ficheros con datos de orientación de la mirada

Como *EyeDataLogger* centra en datos relativos a la mirada, se ha decidido que únicamente se recopilaría información relativa a la dirección de la mirada. El dispositivo de *eye tracking* montado permitía registrar también información acerca de la distancia de la mirada, es decir, la profundidad del objeto sobre el que se fija la vista. Esto se descartó ya que no tendría sentido hacerlo en un entorno 3D como el del *pipeline* donde no hay ningún objeto en la escena (el vídeo se reproduce en la *skybox*, el cielo de la escena, considerado un punto en el infinito) y todas las distancias serían infinitas. Tendría sentido haber incorporado esta información si el *pipeline* estuviera preparado para funcionar con escenas tridimensionales en lugar de vídeos.

La estructura de un fichero de *log* se compone de la primera línea, donde se escribe el *timestamp* del vídeo cuando se comienza la recogida de datos, la segunda con los nombres de cada uno de los campos que contienen las muestras de datos, y a partir de la tercera, las muestras de datos recogidos donde siguen todas la misma estructura.

Los campos que constituyen cada muestra son los siguientes:

- **timestamp:** Instante temporal de esa muestra, contando desde que se ha comenzado a recoger información sobre la visualización del vídeo. Se expresa en segundos.
- **EulerRotation:** Rotación de la mirada del usuario en ese instante del tiempo respecto de la posición de reposo de los ojos (mirando hacia adelante). Se expresa en ángulos eulerianos, leídos como un vector tridimensional en Unity.
- **QuaternionRotation:** Rotación de la mirada del usuario en ese instante del tiempo expresada en cuaterniones. Vector de cuatro dimensiones.
- **GazeDirectionWorldSpace:** Vector tridimensional unitario que representa la dirección hacia la que apunta la mirada del usuario en ese instante, expresado en el sistema de coordenadas global.
- **UV\_Gaze\_Range01:** Coordenadas UV normalizadas correspondientes a la mirada del usuario, suponiendo proyección equirrectangular en el formato de los *frames* del vídeo (Proyección equirrectangular en [27]). Es un vector bidimensional.
- **Confidence:** Medida de confianza proporcionada por el dispositivo de *eye tracking* que indica cómo de fiable es una medición realizada.

startingTimestamp		0				
timestamp	EulerRotation	QuaternionRotation	GazeDirectionWorldSpace	UV_Gaze_Range01	Confidence	
0.00997509765625	(327.9541,352.4745,0)	(-0.275427,-0.06307583,-0.01811403,0.9590794)	(-0.1110113,0.5305979,0.8403227)	(0.4790958,0.3219674)	0.9446157	
0.010971435546875	(323.4326,16.28931,0)	(-0.3105581,0.1345204,0.04444594,0.9399375)	(0.2252754,0.5957682,0.7709159)	(0.5452481,0.2968477)	0.3992594	
0.02094482421875	(324.649,14.32992,-5.233875E-07)	(-0.3012547,0.1188383,0.03787017,0.9453512)	(0.2018708,0.5785839,0.7902461)	(0.5398054,0.3036056)	0.3240006	
0.021942138671875	(345.3557,343.6698,0)	(-0.1261562,-0.1408679,-0.01810096,0.9817908)	(-0.2720384,0.2528176,0.9284816)	(0.4546383,0.4186427)	0.99	
0.022939208984375	(341.0691,351.0221,0)	(-0.1639486,-0.07720128,-0.01287123,0.9833591)	(-0.1476127,0.3244281,0.9343215)	(0.4750614,0.3948282)	0.9861793	
0.032912109375	(340.9553,351.0768,1.129014E-07)	(-0.1649312,-0.0767189,-0.01286912,0.9832326)	(-0.14662,0.326306,0.9338239)	(0.4752134,0.394196)	0.8711385	
0.032912109375	(346.6289,340.1961,2.193904E-07)	(-0.1146856,-0.1707931,-0.02001984,0.9784049)	(-0.3296177,0.2312564,0.9153539)	(0.4449892,0.4257164)	0.3060383	
0.042886474609375	(344.5601,356.0771,1.107175E-07)	(-0.1342525,-0.03391683,-0.004597758,0.9903559)	(-0.06594492,0.2662272,0.9616518)	(0.4891031,0.4142228)	0.9099834	
0.054855224609375	(333.7588,350.613,0)	(-0.2262402,-0.07968904,-0.01857442,0.9706287)	(-0.1462924,0.4421508,0.8849301)	(0.4739251,0.3542156)	0.9099834	
0.056850830078125	(346.9444,340.0671,2.191071E-07)	(-0.1119698,-0.1719493,-0.01967569,0.9785239)	(-0.3321069,0.2258968,0.9157924)	(0.4446309,0.4274688)	0.3150353	
0.057846435546875	(333.7664,350.5057,0)	(-0.2261581,-0.0805991,-0.01878089,0.9705687)	(-0.147959,0.4420314,0.8847126)	(0.4736271,0.354258)	0.8585027	
0.06582666015625	(333.0291,350.4886,-2.394911E-07)	(-0.2323951,-0.08062155,-0.01933383,0.9690815)	(-0.1472715,0.453537,0.8789853)	(0.4735794,0.350162)	0.8635027	
0.076796630859375	(343.0774,340.4913,-2.231041E-07)	(-0.1450141,-0.1675805,-0.02492936,0.974816)	(-0.3194901,0.2910795,0.9017754)	(0.4458091,0.4059856)	0.236081	
0.077794189453125	(344.9817,341.7948,-2.209918E-07)	(-0.1290391,-0.1568459,-0.02067468,0.9789386)	(-0.3017493,0.2591281,0.9174966)	(0.4494301,0.4165648)	0.1752116	
0.08776578125	(317.4626,10.25315,-8.690273E-07)	(-0.3612915,0.08327018,0.03241333,0.9281616)	(0.131155,0.6760718,0.7250691)	(0.528481,0.2636809)	1	

Figura C.3: Aspecto de un *log* de datos de la mirada al abrirlo con Excel.

Una vez más, se ha tratado de mantener cierto grado de coherencia con *HeadDataController* al seleccionar el orden en el que escribir los datos a los ficheros de salida. No obstante, se puede observar que aparece una medida nueva respecto a los *logs* de información de la cabeza. Se trata de *confidence*, la confianza de las mediciones realizadas por el dispositivo de *eye tracking*. A la hora de gestionar esta variable se contempló la posibilidad de descartar por completo aquellas mediciones que presentaran una confianza inferior a un umbral, que sería configurable por los usuarios del *pipeline*. Se descartó, debido a que se consideraba más útil generar un conjunto de datos lo menos procesado posible, dejando

la tarea de procesamiento y filtrado de los datos a otros componentes para que cualquier cálculo posterior no se vea condicionado.

De forma muy similar a *HeadDataLogger*, los ficheros generados tienen la siguiente estructura en su nombre:

ddMMyyyy\_HHmmss\_clipIDClip\_gaze.csv

Figura C.4: Formato del nombre de los *logs* de datos de dirección de la mirada

La primera parte hace referencia a la fecha y hora del momento de su creación, mientras que IDClip hace referencia al número de posición que ocupa el vídeo asociado al fichero en la lista de vídeos del componente *videoController*, explicado en la Sección 3.1. La última parte del nombre indica que contiene datos relativos a la mirada, para diferenciar estos ficheros de los generados por *HeadDataLogger*.

### C.3. Cacheando los logs en *ScanpathReplayer* y *SaliencyMapsGenerator*

Acceder a los ficheros de *log* el mínimo número de veces ayudaría a la eficiencia del *pipeline*, eliminando latencias de lectura adicionales que habría que hacer si se leyieran de disco línea por línea cada vez que se necesitara acceder a una muestra de un *log*, y podrían causar problemas en el funcionamiento de la aplicación. Este es el principal motivo detrás de esta decisión de diseño.

Al inicio de cada ejecución, Unity invoca al método *Start()* de cada uno de los componentes de la escena en cuestión. Además, no se actualiza el estado del componente (no se llama a *Update()*) hasta que se termina con *Start()*. Se ha decidido que en este método se colocaría el código encargado de generar las cachés de los ficheros, tanto para *ScanpathReplayer* como para *SaliencyMapsGenerator*. En el Anexo A, Sección A.3 se comenta que realizar tareas pesadas en el método *Start* podría repercutir en retrasos en el inicio de la aplicación. En este caso es indiferente, puesto que como máximo el tiempo de espera será de unos pocos segundos, y este modo de recreación de *scanpaths* no implica ningún tipo de interacción en tiempo real con los usuarios, por lo que no es crítico garantizar un inicio instantáneo.

Para cada uno de los ficheros, se lee todo su contenido y se almacena en memoria, separado línea por línea. A continuación, se separan los campos de cada una de las líneas y se almacenan en diferentes cachés en memoria, convertidos del tipo cadena de caracteres al tipo que corresponda a cada campo.

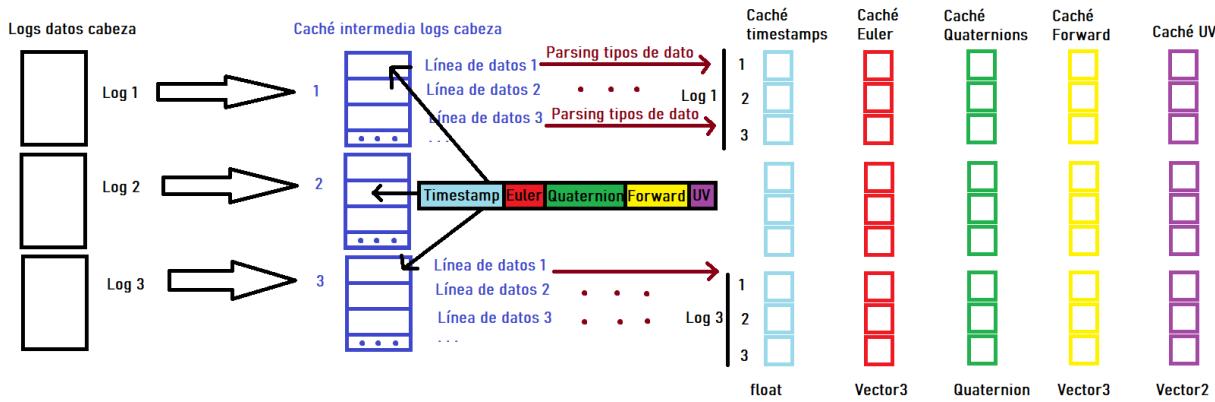


Figura C.5: Proceso de generación de cachés con los datos almacenados en los ficheros de *logging*.

La figura de encima muestra un ejemplo para los ficheros de *log* de datos de la cabeza, pero el proceso se realiza también para los *logs* de la mirada; el proceso es idéntico pero añadiendo una caché para la confianza de cada medición.

Se ha decidido que aunque no se necesiten todos los campos para este componente (por ejemplo, con tener la rotación de la cabeza en cuaterniones es suficiente para *ScanpathReplayer*), se generan las cachés de todos ellos, pensando en futuras ampliaciones del *pipeline*, ya sean propias o de los usuarios que deseen editar el código para añadir funcionalidades.

## C.4. Archivos y ficheros generados por *SaliencyMaps-Generator*

Una vez explicado en la Sección 3.4 todo el comportamiento necesario para la generación de mapas de fijaciones y de saliencia, queda comentar qué archivos genera el componente al ejecutarse y cómo los organiza.

Todo lo que genera este componente va a parar a la carpeta “Outputs” dentro de la jerarquía del proyecto de Unity. Cada ejecución genera allí una carpeta cuyo nombre sigue el siguiente formato:

nombreVideo_ddMMyyyy_HHmmss_n_users
-------------------------------------

Figura C.6: Formato del nombre de las carpetas que contienen mapas de saliencia.

Aquí, *n* hace referencia al número de *logs* de diferentes usuarios cuya información se ha

## C. Logs y archivos generados

---

utilizado para el cálculo de los mapas de fijaciones y de saliencia. Dentro de esta carpeta se encuentra otra subcarpeta, que se llama *gazeData* o *headData* dependiendo del tipo de *logs* pasados al componente en el editor de Unity. Por último, dentro de esa carpeta se encuentran otras dos carpetas más, llamadas *Fixations* y *Saliency*.

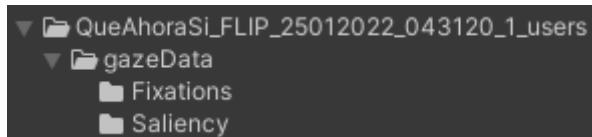


Figura C.7: Sistema de carpetas creado dentro de *Outputs* para el procesado de *logs* de un vídeo.

Los mapas de fijaciones se almacenan en la primera, y los de Saliencia en la segunda. Estos tienen el siguiente sistema de nombrado:

`{FIX/SAL}_startFrame_start_currFrame_curr_of_numFrames`

Figura C.8: Formato del nombre de los mapas de fijaciones o saliencia para cada *frame* de vídeo.

Los mapas de fijaciones tienen la extensión de archivo *.PNG* y los de saliencia *.bmp*

Para mejor legibilidad de los mapas, a la hora de generar los ficheros de imagen se pintan los píxeles de alrededor del que realmente es la fijación. La cantidad de píxeles que se pintan corresponde siempre con 1 grado de ángulo visual horizontal.

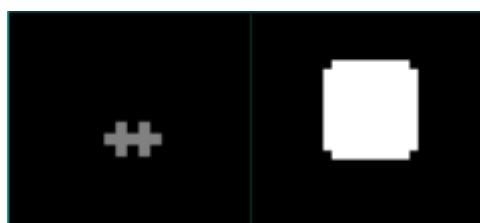


Figura C.9: Tamaño de una fijación en los mapas generados. Ambas cubren un grado de ángulo visual. A la izquierda, resolución de 384x192. A la derecha, 3840x1920.

# **Anexo D. Enunciado práctica 1 VR del Master in Robotics, Graphics and Computer Vision**

---

## **VR Lab #1: 360º content visualization**

---

### **1.1. Introduction and goals**

Virtual Reality is experiencing an unprecedented growth. On the one hand, technology is rapidly evolving, which is leading to a decrease in the price of many consumer-level VR devices (e.g., Oculus Quest, HTC Vive, Valve Index...), as well as an increase in their availability for many users. On the other hand, professional creators and practitioners are exploring new ways of presenting a wide variety of content: cinematographic content, videogames, narrative experiences, virtual tourism, or even urban design.

While for some of those applications 3D, computer-generated content is enough, for others one may want to display real, captured footage in a VR headset. How is this real footage captured? How can we visualize it in a VR headset? In this lab assignment we will learn a very common way of representing and visualizing real, captured footage (which can also be used for synthetic content). It will also serve us as an introduction to Unity, and functionalities that we will be using throughout the course.

Specifically, throughout this assignment you will learn the following:

- A short introduction to 360º content acquisition and storage.
- Visualization of 360º content in a VR headset.
- Basic use of the Unity engine, and its VR functionalities.
- Shader basics, including what is a shader and why are they useful.
- Usage of mobile phones for VR experiences.

There will be different types of tasks in this laboratory. Most of them are small, self-explanatory tasks to give you a robust **basis** for the next assignments. When a new concept was introduced, you may have some **mandatory tasks** to reinforce the learning. Mandatory tasks are preceded by [MT-X]. Finally, **optional tasks** are presented at some points, and preceded by [0T-X].

### **1.2. 360º content: The basics**

One of the key aspects of VR is that it offers an immersive environment all around the viewer: Unlike traditional media, where the content is displayed in a 2D screen in front of the

## D. Enunciado práctica 1 VR del *Master in Robotics, Graphics and Computer Vision*

---

MRGCV - Virtual Reality Lab 1: 360° content visualization

---



Fig. 1.1: Examples of equirectangular panoramas [2, 1] reprojected to a sphere. Note that the distortions disappear because the sphere adjusts itself to the whole panorama.

viewer, content in VR is displayed in all the 360 degrees that surround the user. This makes this content harder to represent, since there is much more information. One of the most common (yet not the only one) ways to represent this content are the so-called equirectangular (or 360°) panoramas.

Those are a 2D projection of a three-dimensional scene that, although distorted, has information of the whole scene, hence no information is lost (see Figure 1.1). You can think of them as if you were laying out an spherical Earth globe into a world map. In this lab, we are going to work with this kind of representation, which are very useful in VR, specially (but not only) when working with captured content. (You will learn a bit more in Lesson 4 of the course).

- Some state-of-art works have already worked with equirectangular panoramas [2, 1], and have let them publicly available. **Download some of those equirectangular panoramas**; we will need them for the next steps in the lab, and this way you will know a bit more about them.

As mentioned above, 360° panoramas are a quite interesting projection of a three-dimensional scene. Although distorted, they have information about the **whole scene**: no information is lost. So, the 3D scenario can be recovered from them. Moreover, equirectangular panoramas' inherent structure allows a **simple projection to a spherical geometry** (see Figure 1.1).

The first task on this lab assignment is to project equirectangular panoramas to a spherical geometry in Unity, in order to visualize the scene without distortions.

### 1.2.1. Introduction to Unity

Unity is a powerful, publicly available cross-platform game engine, which allows to design, build, compile and run a wide variety of games and experiences in both 2D and 3D. Working in most of the current available development platforms, Unity has a basic programming API in C#, and its engine runs over OpenGL (in Windows, Mac and Linux), Direct3D (in Windows) and OpenGL ES (in Android y iOS). It provides an huge support in shading techniques, and facilitates the design thanks to its drag-and-drop system.

To complete this assignment (and also for the next ones in this course), a complete installation of Unity is needed. This means you will need to **install Unity**. Note that Unity offers a free, non-commercial purposes license that is available for any user.

- Now, create a **new 3D project** in Unity. After the program loads, you will see a screen

MRGCV - Virtual Reality Lab 1: 360° content visualization

---

similar to the one shown in Figure 1.2.

- A - Inspector tab. The inspector tab contains the properties of the object you select in the virtual world. Here you can check and modify object's position, rotation, or scale; see attached scripts to an object, or its corresponding shaders.
- B - Project tab. It works like an common PC explorer: you can see the folder structure of your project and the files contained in each of them. **Good practices: Create folders to separate each different type of component you work with: shaders, scripts, material, etc.**
- C - Hierarchy tab. All your scene objects will be included in this hierarchy. By default, all Unity projects start with a camera and a directional light.
- D - Game tab. You will see your running application in this screen.
- E - Scene tab. This tab contains the scene itself. You can drag and manipulate objects in this scene.
- F - Console tab. By default, you can alternate your scene and your console tab. Console tab is extremely important: it will print any error message that happens during your program compilation or in running time.

Note that you can move, add, or remove any Unity tab to create your custom structure, but we recommend you to keep the default one, and add the tabs you may need.

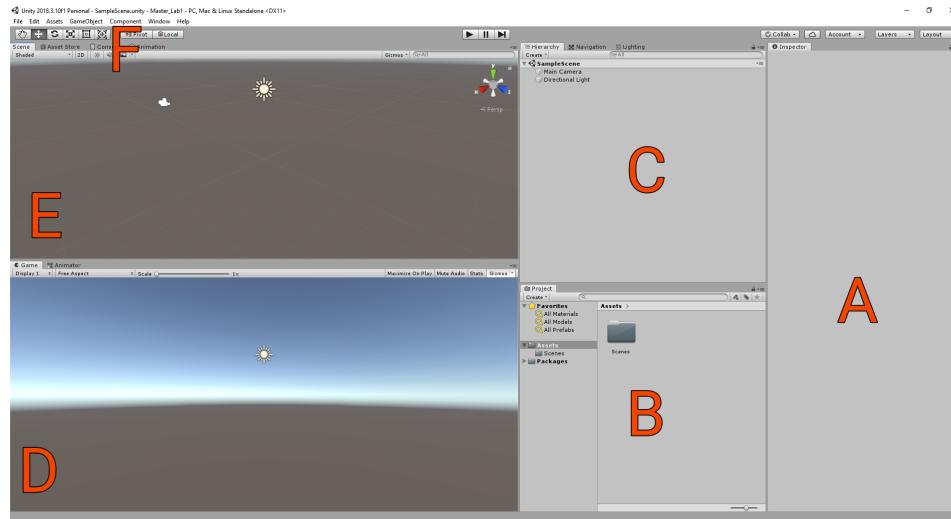


Fig. 1.2: Default Unity screen. Main document contains the explanation of each tab.

MRGCV - Virtual Reality Lab 1: 360° content visualization

---

### 1.2.2. Creating the scene

- Go to the *scene tab*, and click on the camera. The *inspector tab* now show main camera's properties. Make sure your camera's position and rotation are (0,0,0), so that it is easier to have a clear reference.

- We want to visualize 360° panoramas. As we have learned before, we will need an **spherical geometry**. Right click in the *hierarchy tab*, select *3D object* and then **Sphere**. A new sphere called "Sphere" has appeared in the scene and in the *hierarchy tab*.

- Click the sphere, go to the *inspector tab*, and change its position to (0,0,1). If you now look at the *game tab*, you should see the sphere in front of you.

- Now, we have to import the panorama we previously found to the project. Go to the *project tab*, right click and choose **Create** and then **Folder**, and rename it "Panoramas". Go to your computer's folder where you saved your panoramas, and drag them to the recently created folder.

- To project the panorama into the sphere, just drag your panorama to the **Sphere** object in the *hierarchy tab*. Unity will automatically project the panorama and you will see the sphere re-textured.

- We now have a sphere with a equirectangular panorama projected on it, but our camera is out of the sphere and we are not seeing any virtual environment (yet). Move the **Sphere** to the (0,0,0) position, and change its scale to (2,2,2). At the moment, you should see that the sphere disappears and only the background shows.

Why is this happening? By default, OpenGL renders the scene by a process called **Face Culling**. Everything in the graphic pipeline is rendered by triangles. Triangle primitives after all transformation steps have a particular facing. This is defined by the order of the three vertices that make up the triangle, as well as their apparent order on-screen. By default, OpenGL discards objects based on their apparent facing. As the **Sphere** normals are pointing to its outside, Unity will not render it by default.

- We want OpenGL to render the sphere from inside. To this, we are programming a *shader*. Go to the *project tab* and create a new folder called "Shaders". Go in, right-click, choose "Create", "Shader", and then "Image Effect Shader". Rename it so you easily identify it.

- Double-click the *shader* to see its content. First of all, change the first line to rename your *shader*. You can call it:

Shader "Hidden/Panorama"

Then, you will see a properties section.

```
Properties
{
    _MainTex ("Texture", 2D) = "white" {}
}
```

MRGCV - Virtual Reality Lab 1: 360° content visualization

---

Shaders accept **Textures** as input arguments. You can pass a shader as many textures as you want. You will then see a Subshader section, where other graphic pipeline's properties can be set.

```
SubShader
{
    // No culling or depth
    Cull Off ZWrite Off ZTest Always
```

Then, some macros are included in the sader.

```
Pass
{
    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag

    #include "UnityCG.cginc"
```

No modification should be done here. After this, we can see the definition of the data types that are passed trough the pipeline process.

```
struct appdata
{
    float4 vertex : POSITION;
    float2 uv : TEXCOORD0;
};

struct v2f
{
    float2 uv : TEXCOORD0;
    float4 vertex : SV_POSITION;
};
```

The **appdata** struct represents the mesh vertex data that is passed to the **vertex shader**. In this case, it contains information about the **position** of the vertex, and the **texture coordinates** that corresponds to that vertex. Analogously, the **v2f** struct contains the data that will be passed from the vertex shader to the **fragment shader**, which will render the final image to be displayed.

We now have two functions, which define the steps the vertex and fragmetn shader execute.

v

## D. Enunciado práctica 1 VR del *Master in Robotics, Graphics and Computer Vision*

---

MRGCV - Virtual Reality Lab 1: 360° content visualization

---

```
v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.uv = v.uv;
    return o;
}

fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);
    // just invert the colors
    col.rgb = 1 - col.rgb;
    return col;
}
ENDCG
```

In this case, the **vertex shader** just calculates in which coordinates of the screen will the vertex be projected, and assigns it a texture coordinate. On the other hand, the **fragment shader** recovers the corresponding color for a given vertex from the texture coordinates previously calculated, and inverts its color. The fragment shader will return, for each vertex, the final color it will have when displayed in the screen.

- Now, we do have a slight idea about how do shaders work. We want the sphere normals to go to the inside of the sphere instead of their actual direction. In the ‘‘SubShader’’ properties, we are going to change the culling process (as aforementioned, OpenGL render process depends on the geometry’s normals directions). **We are going to change “Cull Off” to “Cull Front”, and delete the other two options “ZWrite” and “ZTest”.** With this change, any object that is rendered with this shader attached will have its normals pointing inside. We will also **delete the inverting color line in the fragment shader**, so that the panorama final RGB is not manipulated.

- We can now save the shader and return back to Unity.
- Although the shader is ready, it is not attached to our **Sphere** yet. Select it and go to the *inspector tab*. When the panorama was attached to the sphere, a new component appeared in the tab, called as the panorama’s filename you chose before. To add our brand new shader, **drag the shader file over the component name**.

Right now, you should see part of the scene in the *game tab* (see Figure 1.3).

**[MT-0]** We are going to some different shaders. Create a copy of your current shader and add the necessary changes for each of the following modifications: (i) **swap** R, G and B channels, (ii) **grayscale** the scene, and (iii) **flip** the panorama in render time either vertically, horizontally, or both ways.

**[MT-1]** Unity allows to include text boxes in the scene. Investigate how to add a *canvas* with some texts (e.g., in the upper-left corner), and write the latitude and the longitude which the

MRGCV - Virtual Reality Lab 1: 360° content visualization

---

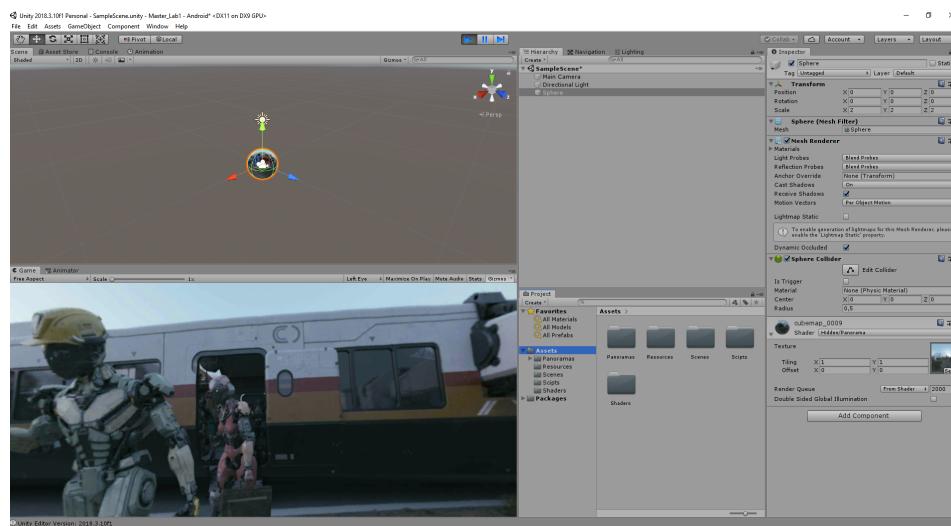


Fig. 1.3: Our project so far. We have an sphere with an equirectangular panorama projected on it, and a shader that allows it to be watched from inside.

user is looking at in each frame. Remember that your camera has an orientation property. Think which axes are you interested in.

**[OT-0]** Shaders can be used for further manipulations. Create a *shader* that adds some (a parameterizable number of) parallels and meridians (lines) over the image, creating the illusion of a spherical mesh over the scene. Note that you will need to take into account some concepts as latitude, longitude, radians...

### 1.2.3. Exploring the scene

We now have a virtual 360° scene based on a equirectangular panorama. However, if we press “Play”, we cannot rotate, hence we always see the same scene part. Although when visualizing the content with an HMD is trivial, we may want to visualize it manually when no HMD is available, or just for a quick debug of our application. Thus, we are creating a manual movement script.

- We will go to the *project tab* and create a new folder called “Scripts”. Inside it, we will right-click, select “Create” and choose “C Script”. Name it so you can identify it will be the script related to the movement.

- Open the script. By default, two functions are included but empty: Start() and Update(). The first one is a function that is executed only once, as the program begins. The latter is executed at the end of each frame. If your application runs in 60 FPS (as many videogames), it means this function will be called 60 times per second.

### MRGCV - Virtual Reality Lab 1: 360° content visualization

---

- We will use WASD keys to rotate the user, simulating a real head rotation. Add a **public** floating global variable for the **rotation speed**; the higher the value, the faster your camera will rotate. We need the program to check frame by frame if any of those keys have been pressed. We are adding two lines in the Update() function:

```
var direction = new Vector3(-Input.GetAxis("Vertical"),
                            Input.GetAxis("Horizontal"), 0.0f);
```

In this first line, we are calculating the direction of the rotation, given the input keys. If no keys are pressed, direction will be (0, 0, 0), hence no rotation will happen. As any of the inputs is pressed, the vector will change towards that direction. Note that Unity has default binding for inputs: “Horizontal” axis is bound to A-D and left-right arrows, whereas “Vertical” axis is bound to W-S and up-down arrow.

```
transform.Rotate(direction * speed * Time.deltaTime);
```

With this second line, we are **rotating** the camera in the aforecalculated direction, with the previously fixed speed. The factor *Time.deltaTime* is the inverse of the frame ratio, so it allows the rotation to be split into all the frames in a second.

- The rotation script is now ready. You can close the editor and go back to Unity. Although the script is ready, it is not attached to the camera yet. To this, drag the script to the “Main Camera” object in the *hierarchy tab*.

Now we are ready to run our experience. Press the “Play” button in the top of the Unity interface. The program will start running in the *game tab*. You can now rotate your camera with the keyboard.

[**MT-2**] Add a new script to the camera so you can rotate it with the mouse too.

[**OT-1**] Once your visualizer is ready and you can move, you can try new different content. As most of you know, Google Street View allows a 360° viewing of almost any place on Earth. Search for any source to download equirectangular panoramas from Google Street View and import them into your visualizer.

### 1.3. VR experiences on a smartphone

Until now, we have designed and implemented a 360° viewer in Unity. However, it is true that visualizing this kind of content in PC feels no realistic at all. We are now going to prepare our project to work with Android or iOS devices.

- We first need to tell Unity this application will work in VR. To that, go to the upper menu, and choose “Edit” and then “Project Settings”. A new windows will show up. Choose “Player” in the left column menu. There, look for “XR Settings” and check “Virtual Reality Supported”. Wait for some packages to be downloaded and imported. After that, just below this recently

MRGCV - Virtual Reality Lab 1: 360° content visualization

---



Fig. 1.4: Screenshot from an Android phone with the application running. Panorama taken from Google Street View, in the shore of the Ebro river.

checked box, you will see a “Virtual Reality SDK” list with no items yet. Click on the “+” button and then select “Cardboard”.

- Now, we want to export our application. Again, go to the upper menu. Choose “File” and then “Build Settings”. A new window will appear. Make sure that your scene is in the top list. If not, click the “Add Open Scenes” button. After that, change the device in the left column to your device OS. When selected, click “Switch Platform”, and wait for Unity to change necessary files.

- Now go to “Player Settings”. There, change the “Company name” and ”Product Name” to something different. After that, go down to “Other Settings” and change the “Package Name” to *company.product*, the two names you chose before. Go to “Identification - Minimum API Level” and choose an API lever equal or higher than 19, since Cardboard does not work with a lower one.

- You can now build your application. If you are using Android, a new APK will be generated. Save it wherever you want, transfer it to your device and install it. When running it, you should see a two-eyed screen ready to be held in a Cardboard setup 1.4.

The process to follow for iOS devices is quite similar: Unity already offers it as a valid device to which export your applications. If you are an iOS user but you cannot follow the process, you can use an Android emulator, such as BlueStacks<sup>1</sup>.

---

<sup>1</sup><https://www.bluestacks.com/es/index.html>

MRGCV - Virtual Reality Lab 1: 360° content visualization

---

#### **1.4. Reporting your results**

You should submit a **report** (in .PDF format) including your results for the tasks (results should be clearly labeled following the labels of the tasks they correspond to). There is no required style or fixed structure for your report, you will have to choose how to report the work you did. There are, however, some guidelines that you need to follow:

- The report should not be longer than **three pages** (*tres “carillas”, no tres páginas*), and should at least address the mandatory tasks. The report file should be named `labXX-mainReport-YYYYYY.pdf`, with XX the number of the lab, and YYYYYY your NIP.
- Do not include whole snippets of code in the report: You may submit an additional .ZIP file with the code and shaders you wrote, and indicate in the report its function with a couple sentences at most.
- Including in the report the **main difficulties, thoughts, or insights** you had through the whole laboratory, as well as its relation to the lectures of the course, will be positively evaluated.
- If there is any other part of your work that cannot fit in the report (e.g., short videos or sets of a large number of screenshots), you should submit them in a separated, supplementary .ZIP file, **but always adequately referencing it in the main report, so we are aware of the existence of that file**.
- We will not look at submitted items that have not been referenced in the main report.

In terms of evaluation, mandatory tasks can get you up to 8 points out of 10, and the 2 remaining points can be obtained through the optional tasks.

You should submit your report (and supplementary material) via Moodle, uploading them in the corresponding Moodle task. **Deadline: March 8th, 2021 @ 23:59.**

## Bibliografía

---

- [1] Jesús Gutiérrez y col. “Introducing UN Salient360! Benchmark: A platform for evaluating visual attention models for 360° contents”. En: *2018 Tenth International Conference on Quality of Multimedia Experience (QoMEX)*. IEEE. 2018, págs. 1-3.
- [2] Vincent Sitzmann y col. “How do people explore virtual environments?” En: *IEEE Transactions on Visualization and Computer Graphics* (2017).