

E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

Sistema de gestión de cámaras en realidad virtual para Unreal Engine.



Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor: Álvaro Lerga Armendáriz

Directores: Oscar Ardaiz Villanueva,
Amalia Ortiz Nicolás

Pamplona, 6 de septiembre 2021

Agradecimientos

A mi familia por haberme respaldado y dado ánimos durante toda la carrera.

A mis directores del Trabajo de Fin de Grado por la ayuda, indicaciones y sugerencias realizadas para mejorar tanto la versión final del proyecto como la redacción de la memoria y su presentación.

A mis amigos y compañeros de la carrera por su apoyo y ayuda.

Agradezco a la empresa Dr. Platypus & MS. Wombat por prestarnos material para realizar el proyecto y aportar comentarios útiles sobre conceptos de la grabación de una película, funcionamiento de la industria y correcciones del mismo.

A Marta Oroz Yoldi por ayudarme con los modelados utilizados en el proyecto.

A Ander Oyarzun por colaborar conmigo en este trabajo.

A Josune Sorbet por ayudarme a grabar los videos, motivarme a llegar a horas decentes al laboratorio y por la siguiente frase.

Gracias a todos vosotros el proyecto ha llegado a lo que es ahora.

Contenido

Agradecimientos	2
Palabras clave	4
Motivación	5
Introducción	5
Objetivos	6
Descripción general de proyecto	7
Industria cinematográfica y VR	9
Storyboard y su importancia	12
Características de un storyboard	13
Materiales	14
Hardware	14
Oculus Rift	14
Wacom One	15
Software	16
Unreal Engine 4.26	16
Entorno fbx con el que se han realizado las pruebas	20
Tareas Realizadas	21
Investigación sobre las cámaras virtuales.	21
Aplicaciones de ejemplo de realidad virtual en Unreal.	23
Pruebas con las funcionalidades de VR.	24
Salida de la cámara a una ventana emergente.	25
Creación del personaje director	26
Conexión multijugador	27
Implementación	29
Introducción	29
Personaje VR (Camarista)	32
Movimiento	32
Punteros	33
Acciones	33
Cámaras	36

Personaje Director	39
Movimiento	39
Puntero	40
Acciones	41
Indicador	45
Conexión multijugador	46
Problemas durante el desarrollo	49
Problemas con la creación de las cámaras	49
Problemas con el uso de la tableta	50
Resultados	51
Pruebas descartadas	52
Una cámara con todas las PCSceneCaptureComponent.	52
Posibles introducciones futuras	53
Conclusiones	54
Manual de uso	55
Camarista	55
Controlador Izquierdo	55
Controlador derecho	56
Director	56
Teclado	56
Ratón	57
Tableta	57
Referencias	58
Ilustraciones Implementación	60
Figuras	60

Resumen del Proyecto

Dado que la Realidad Virtual es un campo emergente, que se está empleando para mejorar la planificación y ejecución de actividades al poder recrearlas o simularlas, la intención del trabajo es aplicar Realidad Virtual al proceso de creación de storyboards y animáticas.

El trabajo consiste en desarrollar una aplicación para el control de cámaras de una escena desarrollada en Unreal Engine. Esta aplicación facilita el proceso de diseñar el sistema de cámaras de la escena ya que permite no sólo gestionar las cámaras si no también la colaboración entre distintos miembros del equipo.

Así, gracias a esta herramienta una persona puede sumergirse en el mundo 3D a través de las gafas de Realidad Virtual y visualizar la salida de las cámaras, es decir, su campo de visión, y trasladarlo a una ventana diferente en tiempo real, tanto en ejecución como en el editor del proyecto. Cuando se modifica el objetivo de la cámara, dicho cambio se refleja en la ventana correspondiente. Dichas cámaras se pueden mover, rotar y eliminar, así como obtener información de las mismas. Simultáneamente, otra persona puede colaborar con la persona que está dentro del escenario virtual utilizando tanto el teclado y ratón como una tableta gráfica para ofrecerle indicaciones como: señalizar mediante indicadores para la posición de las cámaras, resaltar un cámara, dibujar en el espacio 3D para establecer referencias visuales, etc.

Palabras clave

- Realidad Virtual
- StoryTelling
- StoryBoard
- Unreal Engine
- cámaras virtuales
- C++
- SceneCaptureComponent2D

Motivación

La principal motivación para realizar este proyecto era facilitar las tareas de creación del storyboard. Una empresa se puso en contacto con nosotros y nos preguntó si era posible realizar una aplicación que permitiera emplear la Realidad Virtual para la planificación de una película de animación, en concreto, para la parte del desarrollo del StoryBoard. Creímos que era una buena idea y que la Realidad Virtual puede ayudar mucho en la creación de este tipo de producciones.

Además, personalmente había trabajado poco con el motor gráfico Unreal Engine por lo que quería adquirir experiencia e investigar sobre las posibilidades del mismo. Mi experiencia con la Realidad Virtual anterior a realizar el proyecto era prácticamente inexistente, más allá de haberla probado alguna vez, por lo que también tenía curiosidad por ver cómo eran las sensaciones y la experiencia.

Introducción

El storyboard es una parte muy importante durante la creación de una producción cinematográfica. Sirve para esquematizar las escenas permitiendo resumir en dibujos los diferentes planos de una escena. Esto permite que se facilite la organización de las secuencias y los planos de una película sin la necesidad de haber grabado nada previamente.

La llegada de los dispositivos VR permiten tener una visión más precisa y profunda de muchos campos, el de la animación no es una excepción.

Teniendo estas dos ideas en cuenta, en este trabajo se implementará el uso de las gafas VR en el desarrollo del documento storyboard y que así este pueda aportar más información y detalles útiles durante la grabación de la producción. Principalmente estaría dirigido a su uso en producciones de animación teniendo como entorno de la realidad virtual un escenario que formaría parte de la producción de tal forma que las gafas VR permitirían tomar decisiones más precisas sobre los elementos del escenario, los personajes que participen en la escena y sus movimientos.

Objetivos

Los objetivos iniciales del proyecto eran:

- Poder gestionar las cámaras de una escena. La primera idea consistía en moverlas y rotarlas.
- Obtener la salida de las cámaras en una ventana emergente, para poder dar indicaciones de los resultados.

Una vez desarrollados estos dos puntos y con el feedback recibido de las pruebas que se realizaron, comenzamos a pensar cómo se podría mejorar la experiencia de uso. Tanto para facilitar su utilización como para introducir nuevas funcionalidades con la intención de completar la gestión de las cámaras.

Creímos que otros objetivos a tener en cuenta serían:

- No sólo poder rotar y mover las cámaras sino crearlas y eliminarlas.
- Poder crear indicaciones dentro de la escena para definir mejor la colocación de las cámaras, más allá de escuchar a la persona que está viendo las ventanas y el punto de vista de la persona con las gafas VR.

Al finalizar el proyecto, los objetivos que se plantearon fueron superados y se incluyeron más opciones de las que en un principio se plantearon.

Descripción general de proyecto

El proyecto consiste en desarrollar un sistema de gestión de cámaras que permita a los creadores de animación planificar con mayor precisión los planos que se deben incluir en el storyboard.

La persona que desempeña esta función se llamaría camarista.

La gestión de las cámaras requiere crearlas, mover su posición, rotarlas en cualquiera de sus ejes y eliminarlas. El camarista se colocaría las gafas VR de tal forma que se encontraría en una representación del escenario que se usaría en la película o videojuego. De esta manera sería más consciente de las distancias, la disposición de los objetos...

Para poder ver con precisión el enfoque de la cámara y el plano que esta está capturando se puede visualizar lo que la cámara está capturando en una ventana emergente. Esta ventana muestra en tiempo real todo lo que enfoca la cámara. Por lo tanto, en caso de que la cámara se rote, mueva o elimine el resultado del nuevo plano se verá reflejado en la ventana.

El proyecto dispone de cuatro ventanas para desempeñar esta tarea. El camarista puede seleccionar la ventana, de entre las 4 existentes, en la que plasmar lo que la cámara está capturando. Además de estas 4 ventanas existe una quinta que incluye la visión de la persona que lleva puestas las gafas.

En paralelo, otra persona es la encargada de revisar los planos de las cámaras. Esta persona al poder ver los planos de las cámaras, y sin tener puestas las gafas VR, tiene una visión más clara de la escena por lo que puede darle indicaciones a la persona con las gafas.

Para ello, además de encargarse de revisar las ventanas, también dispone del control de un personaje en la escena. El camarista que lleva las gafas VR puede ver dicho personaje y así recibir indicaciones visuales y más precisas que simples descripciones del entorno. Esta persona realiza las funciones de un director.

Las indicaciones de las que dispone son:

- Crear puntero láser. Con este puntero el director puede señalizar en tiempo real en el escenario y así facilitar las indicaciones que le dé al camarista.
- la posibilidad de colocar marcadores para indicar posiciones de las cámaras y eliminarlos.
- Señalar una cámara para indicar al camarista la cámara en cuestión de la que se está hablando.

- Dibujar en el escenario para dar indicaciones sobre el movimiento o rotación de las cámaras. El color de estas indicaciones puede ser seleccionado por el director.

En caso de que exista en el escenario una cámara seleccionada, los indicadores que se creen incluirán un texto con el nombre de la cámara seleccionada. De esta forma, el camarista sabrá qué cámara debe ir en dicha posición, aunque el director cambie la cámara seleccionada. Así se puede asegurar la posición específica de cada cámara en la escena.

Un ejemplo de estas funcionalidades sería un supuesto en el que ocurre lo siguiente:

- El camarista, usando las gafas VR, crea varias cámaras en la escena, seleccionando las ventanas en las que se quiere visualizar cada una de ellas.
- El director puede observar la escena desde la tableta, el punto de vista del camarista en una ventana de ejecución y los planos de las cámaras.
- El director selecciona las cámaras y coloca los indicadores en las posiciones en las que quiere que se encuentren cada una de estas. Esta tarea puede llevarlas a cabo desde la tableta o el ordenador.
- El camarista colocaría las cámaras en sus correspondientes posiciones mediante las gafas VR.
- Una vez están colocadas, el director, mediante el puntero láser y la opción de dibujar, indica el ángulo y rotación de las cámaras para enfocar el plano como desea que se muestre en el storyboard.
- El camarista rota las cámaras para que estas cumplan las indicaciones del director.

El sistema de gestión de cámaras está desarrollado para el motor gráfico Unreal Engine 4. Para ser exactos es la versión 4.26 del motor gráfico desarrollado por la empresa Epic Games. La implementación de los comandos está desarrollada en C++ en el entorno visual Studio 2019.

Industria cinematográfica y VR

La industria cinematográfica es una de las más populares del entretenimiento. Los números obtenidos de la suma de los beneficios de la venta en taquilla y los beneficios obtenidos por la venta de la industria para el consumo en casa, es decir, de películas y servicios como: Netflix, HBO, Amazon Prime son únicamente superados por la industria del videojuego.

En 2019 las ventas en taquilla sobrepasaron los \$42 mil millones de dólares de recaudación y el entretenimiento en casa alcanzó la cifra de los 55.9 mil millones de dólares, estos datos superan ampliamente a otras industrias del mundo del entretenimiento como la música que el mismo año recaudó 20 mil millones de dólares.[9]

Dentro de la industria cinematográfica, la animación es uno de los sectores que está más interesado en introducir las nuevas tecnologías en su metodología de trabajo. Se trata de un sector que, en un corto espacio de tiempo, en los últimos 2 o 3 años, ha experimentado un cambio radical abrazando las nuevas tecnologías como motor para mejorar la calidad y rapidez en la creación de sus producciones. [10]

Muchos estudios están introduciendo inteligencia artificial para poder agilizar el flujo de trabajo. Existen softwares como Houdini[11] que permiten crear animaciones y a partir de esas realizar modificaciones lo que facilita la creación de las animaciones, obteniendo nuevas animaciones con una menor inversión de tiempo y de trabajo.

Otra introducción realizada por la animación es el uso de Machine Learning. Los estudios de animación emplean esta herramienta de la inteligencia artificial para retocar y mejorar la calidad de las fotografías, re escalar animaciones...

Se espera que el Machine Learning también permita grabar animaciones de actores reales sin la necesidad de utilizar los trajes de captura de movimiento, que actualmente son necesarios para recopilar los movimientos y expresiones faciales de los actores. En este campo se han realizado grandes avances como los MetaHumans desarrollados por Epic Games para Unreal Engine 5.[10]

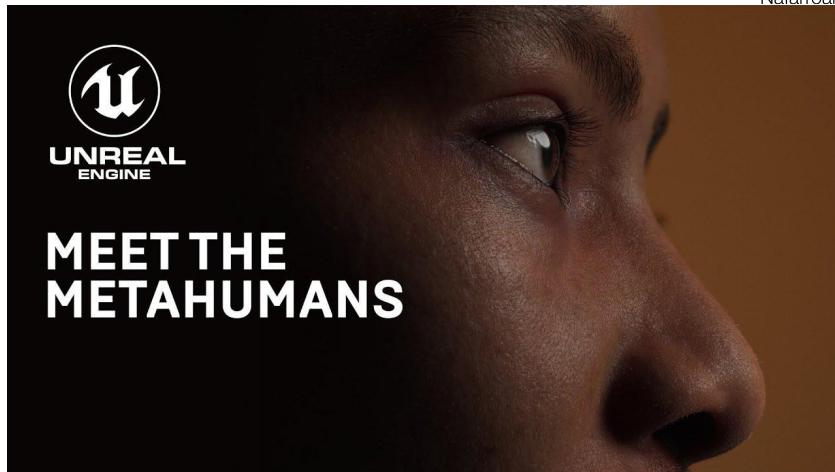


Figura 1 Portada del video de presentación de MetaHumans[12]

A pesar de todos estos avances realizados y las ventajas que conlleva realizar una película o un corto de animación frente a una producción live-action, una de las principales desventajas de las películas de animación es la imposibilidad de tomar un punto de vista en primera persona del entorno en el que se desarrolla una película. Una solución para este problema es el uso de la realidad virtual.

La realidad virtual ya se ha introducido en el mundo cinematográfico para hacer más inmersiva la experiencia del espectador. Se puede considerar un paso natural para los estudios de animación, que en no mucho tiempo han pasado de ser siempre en 2D a dar un salto al 3D permitiendo al espectador observar más en detalle los personajes y el mundo. Con la realidad virtual pueden experimentar vivir en él.

El uso de la realidad virtual durante el desarrollo de las películas de animación permite que un animador entienda mejor la localización espacial de los objetos y personajes. Esto hace que el animador sea capaz de realizar con mayor detalle todas las tareas relacionadas con la localización como pueden ser: la colocación en el espacio de los personajes, las poses de los mismos, comprender mejor las distancias, perspectivas y mejorarlas posturas de ciertas partes de los personajes o su movimiento durante la animación.

Por culpa del covid-19 muchas películas y series live-action se han visto obligadas a parar su producción, aunque las animaciones no se han visto tan perjudicadas, lo que las ha beneficiado frente a las live-action, han sufrido los efectos negativos de no poder reunirse. Ambos sectores de la industria han visto en las nuevas tecnologías una forma de paliar las consecuencias que el coronavirus ha causado

a la industria. Y la realidad virtual se puede usar como una forma de planificar la grabación de las películas tanto de animación como live-action.

Un uso que se ha dado para la realidad virtual en las grandes producciones ha sido la creación de cámaras virtuales. Un ejemplo son las películas de la saga Star Wars. Durante el desarrollo de la segunda trilogía de Star Wars y la película de Solo: A Star Wars Story George Lucas empleó las cámaras virtuales para crear batallas de naves más realistas.

Otro ejemplo de las cámaras virtuales empleadas en películas fue el remake de El Rey León. Para que los trabajadores de la película pudieran experimentar el entorno de África, leones en África fueron filmados con cámaras de realidad virtual y reproducidos en el set de Estados Unidos.[11]

Storyboard y su importancia

El storyboard es una parte crucial en el planteamiento y organización de una animación, una película...

Se trata de un conjunto de viñetas en orden secuencial que permiten previsualizar una escena o una animación de manera visual sin la necesidad de realizar una grabación o crear la animación. De esta forma, se pueden tomar decisiones sobre cómo se van a desarrollar las diferentes escenas de acuerdo a una narrativa previamente redactada.

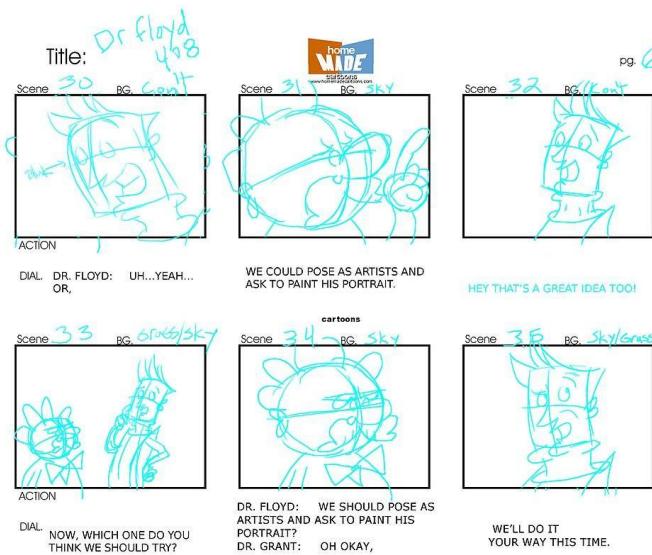


Figura 2 Ejemplo StoryBoard: [Storyboard_for_The_Radio_Adventures_of_Dr._Floyd\[14\]](#)

En cada viñeta se refleja el ángulo de la cámara, el enfoque, los diálogos de los personajes, los movimientos de los mismos, el tipo de plano...

Toda esta información sirve para poder representar y visualizar mejor la idea que se quiere llevar a cabo.

Entre las razones por las que se utilizan los storyboards encontramos las siguientes:

- Se puede utilizar como una base sobre la que realizar cambios y permitir que las personas implicadas en la toma de decisiones, como el director o el productor, decidan si es necesario realizar modificaciones. El storyboard se emplea ya que en ocasiones es difícil transmitir una idea sin una visualización de la misma.

A veces, un guion o una narrativa parece buena sobre el papel, pero no se sostiene al desarrollarlo. El storyboard permite realizar y ver los cambios que se deben hacer en un guion si así fuera necesario.

- También puede emplearse como una guía para los cámaras, animadores y otras personas sobre cómo se desarrolla una escena y el cometido que cada persona implicada debe realizar. Con la información que está anotada en cada viñeta y la representación visual es mucho más fácil tener una visión clara sobre el resultado final que se quiere alcanzar.

Características de un storyboard

- Viñetas o recuadros. Estas imágenes suelen corresponder a un plano en específico de la grabación.
- Acompañando a cada viñeta aparece indicada información sobre ella:
 - Número de escena
 - Identificación de la escena
 - Número de plano dentro de la escena
 - Descripción del audio
 - Observaciones técnicas
 - Entre una viñeta y otra se indica cómo se produce la transición.

Entre los diferentes tipos de transición encontramos:

- por corte directo, es decir, una viñeta sigue a la anterior.
- Por movimiento de cámara: lo que implica que para pasar de una viñeta a la siguiente se emplean planos que muestran el movimiento en la grabación.
- Por disolvencia entre una imagen y otra. Consiste en pasar de un plano a otro mediante imágenes intermedias en las que unas se sobreponen los dos planos. Mientras el segundo plano aparece el anterior se desvanece.

Materiales

Hardware

Oculus Rift

Aspectos y características técnicas[1]

- Pantalla OLED 2160×1200 px.
- Tasa de refresco de 90 fps.
- Ángulo de visión: 110º.
- Área de seguimiento: 1,5 x 3,3 m.
- Sensores de funcionamiento: Acelerómetro, giroscopio, magnetómetro y sistema posicional 360 grados.
- Peso: 470g

Requisitos Oculus Rift

- Tarjeta gráfica: NVIDIA GTX 970 / AMD R9 290 equivalente o superior
- Procesador: Intel i5-4590 equivalente o superior
- Memoria: 8GB+ RAM
- Salida: Salida compatible HDMI 1.3
- Entrada: 3 puertos USB 3.0 más 1 puerto USB 2.0
- Sistema operativo: Windows 7 SP1 64 bit o superior



Figura 1 GafasVR:Oculus Rift

Wacom One

Características técnicas[2]

- **Dimensiones:** 22,5 x 35,7 x 1,4 centímetros.
- **Peso:** 1 Kg
- **Pantalla:** 13.3 pulgadas.
- **Superficie:** Lámina mate.
- **Resolución máxima:** 1920 x 1080 píxeles.
- **Gama de color:** 72 % de NTSC.
- **Tecnología de la pantalla:** AHVA
- **Sistema operativo:** Windows 7.
- **Sistemas operativos compatibles:** iOS, Android, Windows.
- **Software incluidos:** Bamboo Paper, Clip Studio Paint Pro y Adobe Premiere Rush.
- **Función multitáctil:** No
- **Conectividad:** Puerto USB y HDMI.
- **Tipo de lápiz:** Lápiz de Wacom One.



Figura 2 Tableta: Tableta Gráfica Wacom One.

Software

Unreal Engine 4.26

Unreal Engine es un motor gráfico desarrollado por Tim Sweeney, fundador de la empresa Epic Games. Fue utilizado por primera vez por dicha compañía durante el desarrollo del juego homónimo lanzado en 1998. Se caracteriza por estar escrito en C++. En un principio fue creado con la intención de ser utilizado para juegos en primera persona, no obstante, ha sido empleado en múltiples desarrollos no relacionados con el género de los First-Person Shooter.



Figura 3 Logo Unreal Engine[3]



Figura 4 Logo Epic Games[4]

Enfocado en juegos 3D ha conseguido ganar fama también en proyectos no relacionados con los videojuegos, dentro de la misma plataforma anuncian el motor gráfico como una herramienta para videojuegos, películas y eventos televisivos, arquitectura, ingeniería y construcción y en la industria del automóvil y la manufactura de otros productos. Hoy en día, es de los motores gráficos más utilizados. Generalmente es elegido debido a la alta calidad de los gráficos.

En el caso de la industria cinematográfica, Unreal engine ha tomado importancia en las creaciones que requieren animación y efectos especiales. El motor gráfico permite

desde la creación de assets para películas de animación hasta el renderizado de entornos de grandes producciones como la serie de Disney *The Mandalorian*[5] o *WestWorld*. La producción de HBO en su tercera temporada implementa tecnologías del motor gráfico. Otra película que implementó tecnología de Unreal Engine para sus efectos especiales fue *Ford vs Ferrari* (*Le Mans '66*).



Figura 5 *The Mandalorian* usando Unreal Engine para los VFX[6]



Courtesy of John P. Johnson/HBO

Figura 6 WestWorld[7]

A pesar de los logros obtenidos en la industria del cine en los últimos años, donde verdaderamente destaca es en la industria del videojuego. Desde la creación de su primer videojuego y a lo largo de sus 5 generaciones, Unreal Engine ha sido el motor gráfico elegido para el desarrollo de una gran cantidad de videojuegos. Entre los puntos positivos que se pueden destacar del motor gráfico para la industria del videojuego son tanto la posibilidad de programar en C++ la totalidad de los proyectos como la posibilidad de emplear Blueprints. Los Blueprints son una herramienta creada por unreal que permite que los desarrolladores creen objetos, personajes y funcionalidades, en otras palabras, puedan desarrollar videojuegos sin la necesidad de programar el código. Esto se consigue gracias a herramientas con interfaces gráficas que facilitan estas tareas. La programación, las funciones, clases y variables se sustituyen por cuadrados que las representan relacionando unas con otras mediante líneas. Esto permite que un usuario del motor gráfico no necesite conocimientos de programación en C++ siempre que comprenda la lógica y unos básicos conceptos de programación. En el caso del diseño de los personajes y objetos, para añadir componentes es necesario emplear llamadas a funciones. Estas funciones permiten crear y añadir las componentes a la jerarquía de un personaje. En cambio, mediante Blueprints simplemente se pueden arrastrar y añadir en el lugar que se desee y organizar la jerarquía. Para editar ocurre lo mismo, se puede modificar su posición, rotación y tamaño mediante un editor gráfico. Todo esto permite que todos los cambios sean más visuales y por lo tanto, fáciles de realizar. Por el contrario, utilizando código implica modificar numéricamente los vectores de posición, rotación... sin saber el resultado hasta la compilación y ejecución del proyecto.

La generación actual de Unreal Engine es la 4º Generación. Fue lanzada al mercado el 19 de marzo de 2014 accesible al público mediante una suscripción mensual. En marzo de 2015 Epic Games cambió su sistema de financiación y lo lanzó gratuitamente para el público, con pagos dependiendo de las ganancias obtenidas de su uso.

La versión de Unreal Engine empleada para realización del Trabajo de Fin Estudios es la 4.26.1 lanzada el 2 de febrero de 2021.



Figura 7 Icono de Unreal Engine 4.26.1 del Launcher de Epic Games

Entorno fbx con el que se han realizado las pruebas

Durante el desarrollo del proyecto se ha empleado un escenario para realizar las pruebas. El entorno fue prestado para el proyecto por parte de la empresa de animación Dr. Platypus & Ms. Wombat. Se trata de una representación del despacho Oval.



Figura 8 Logo de Dr Platypus and Ms Wombat[8]

El entorno se ha utilizado para realizar pruebas sobre los cambios en las cámaras y tener referencias sobre las que comprobar los movimientos y rotaciones de las mismas.

En relación a la parte más técnica, se trata de un archivo fbx que se introdujo en la escena común actor vacío con un conjunto de modelados adheridos a él para así poder mejorar su colocación ya que cualquier: movimiento, rotación y escalación del actor se traslada a cada uno de los modelados.



Figura 9 Escenario Vista Frontal

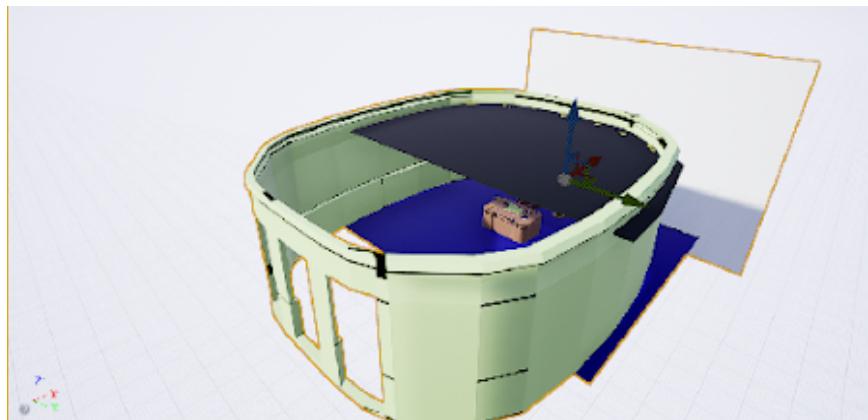


Figura 10 Escenario Vista Picada

Tareas Realizadas

Investigación sobre las cámaras virtuales.

Dado que el proyecto consiste en un sistema de gestión de cámaras virtuales, el primer paso fue investigar cómo se podía crear una cámara dentro de una escena. El objetivo no era encontrar una forma de generar una cámara que fuera similar a la de un jugador sino una forma de extraer el punto de vista de una cámara.

En unreal existen herramientas que permitían realizar esta tarea ya que es usada no sólo para esto, sino que también en otras tareas como la creación de cámaras de seguridad en videojuegos. En una cámara de seguridad lo que se observa a través de ella se transfiere a una pantalla.



Figura 11 Cámara de Seguridad en Unreal Engine[15]

Estas herramientas eran: la componente SceneComponent2D y la clase TextureRenderTarget2D.

Una componente es un objeto de Unreal Engine se fija a un actor de tal forma que actúa como parte de este, como un subobjeto. Las componentes son los objetos que como su nombre indica componen o forman un actor de Unreal Engine.

La clase actor es la clase base de Unreal Engine, es todo objeto que puede ser colocado o creado en una escena. Los actores son clases vacías si no contienen componentes. Las

diferentes componentes le otorgan al actor todas sus características como: forma, color, sonido, colisiones, una cámara, habilidades...

La componente `SceneCaptureComponent2D` permite obtener el punto de vista de un objeto y colocarlo sobre una textura de tipo `TextureRenderTarget2D`. A partir de esta textura se puede crear un material. Los materiales se emplean para simular el material del que está hecho un objeto. Pueden ser: oro, madera, hierro... Al poder transformar el contenido de la textura `TextureRenderTarget2D` en un material se consigue que el objeto que contenga dicho material refleje lo que la componente `SceneCaptureComponent2D`.

Cualquier movimiento que se realice con el objeto que contiene dicha componente se verá reflejado en el material, como, por ejemplo: si la cámara rota, se mueve etc... Además, si algún evento ocurre en la escena y modifica el punto de vista del objeto, por ejemplo: otro objeto se mueve dentro del campo de visión, también se refleja en el material. Por lo tanto, la idea de simular una cámara dentro de la propia escena ya está definida.

Aplicaciones de ejemplo de realidad virtual en Unreal.

Unreal Engine permite utilizar la tecnología VR. Tanto para usar el editor de proyectos, lo que permite crear un proyecto por completo utilizando gafas VR, como para ejecutar un proyecto. La opción para ejecutar un proyecto con VR se denomina VR Preview. Al seleccionar esta opción el jugador puede utilizar las gafas VR para ver el escenario.

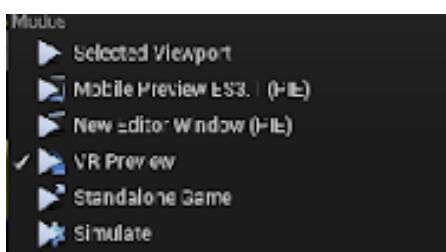


Figura 12 Modos de Ejecución

El comportamiento de las gafas ya está implementado por defecto, esto significa que los movimientos que el jugador realice con la cabeza para mirar a su alrededor no necesitan ser implementados. Esta función permite que el jugador vea el escenario en realidad virtual. Además de poder ver el escenario en las gafas VR, el proyecto también crea una pantalla emergente en la que se muestra el punto de vista de las gafas. Esta ventana facilita parte del objetivo del trabajo ya que permite al director ver el punto de vista del camarista, y de esta manera tiene más información acerca de la situación de todo el escenario. El director puede ver como el camarista está respondiendo a sus indicaciones y modificarlas si fuera necesario.

A pesar de que el movimiento de las gafas esté implementado, si el actor al que el jugador controla no tiene implementado que la cámara rote según la rotación de las gafas el jugador verá limitado su campo de visión al de la propia cámara.

Los controladores no están implementados durante la ejecución. Para implementar los mandos es necesario programar tanto el movimiento como las acciones que se realizan con los botones.

Para implementar estas dos tareas se utilizó un ejemplo de los controladores ya creados.[16] Estas se simplifican gracias a que unreal pone a disposición de los usuarios

funciones y clases para el movimiento de los controladores y tiene mapeados los botones y ejes de un gran número de periféricos. En esta lista de mapas se incluyen tanto los ejes y botones del teclado y ratón como los de los controladores de las Oculus Rift.

Una vez se habían implementado estas funcionalidades se les otorgó a los controladores de unos modelados de manos, incluidos en el ejemplo, y unos punteros láser que facilitaran el apuntado a los objetos de la escena.

Pruebas con las funcionalidades de VR.

Para probar el funcionamiento de las manos VR introduce unas funciones básicas al personaje controlado por el camarista cómo mover objetos, eliminarlos y crearlos.

Salida de la cámara a una ventana emergente.

Sabiendo cómo se podía utilizar SceneCaptureComponent2D y UTextureRenderTarget2D para conseguir simular las cámaras, el siguiente paso era saber cómo trasladar la salida capturada por SceneComponent2D a las ventanas emergentes. Para conseguir este objetivo se utilizaron plugins. De acuerdo con el diccionario de Cambridge, un plug-in es todo programa que permite a un programa de mayor tamaño mejore su funcionamiento. Esta mejora puede deberse a un funcionamiento más rápido o porque añada nuevas herramientas.

Unreal Engine permite tanto instalar plugins ya creados por la comunidad de usuarios como crear plugins propios.

Para crear los plugins Unreal Engine ofrece diferentes plantillas enfocadas a que un plugin realice una tarea concreta. Una de esas plantillas se llama “Editor Standalone Window”. Este tipo de plugin permite crear un botón que se adhiere a la barra de herramientas y cuando se pulsa abre una ventana nueva. Ya existía un ejemplo[17] de cómo crear una ventana emergente que refleja el contenido de una SceneCaptureComponent2D, a través de uno de estos plugins. El plugin estaba diseñado para la versión 4.21 de Unreal Engine. El plugin está compuesto por las clases por defecto incluidas en un plugin de tipo “Editor Standalone Window” y además una clase denominada PCSceneCaptureComponent2D. Esta última clase tiene como clase padre a la clase SceneCaptureComponent2D, y de esta forma se pueda conseguir que en la ventana emergente del plugin se observe lo que la componente captura en la escena.

PCSceneComponent2D al ser una clase de tipo componente, al igual que SceneComponent2D, se puede agregar a un actor como una componente más del actor.

Una vez entendido tanto el funcionamiento del plugin de tipo “Editor Standalone Window” como el uso de la clase PCSceneCaptureComponent2D se modificó el plugin para que fuera compatible con la versión 4.26.1

Para obtener las 4 ventanas se duplicó el plugin que acabamos de detallar realizando ciertos cambios. Principalmente se realizaron cambios en la clase PCSceneCaptureComponent2D. Se cambió el nombre de la clase y todas las referencias del plugin a ella añadiendo el nombre del plugin para que fuera distintivo en cada uno, con esto se consiguió que cada plugin tuviera su propia clase PCSceneCaptureComponent2D. Una clase diferente en cada plugin permite que no

existen colisiones entre los plugin. Un ejemplo de dichas colisiones es que al asignar una componente a un actor el resultado se refleje en más de una ventana.

Se realizaron otras modificaciones como el texto de los botones o el nombre de las ventanas para facilitar el uso del proyecto a los usuarios.

Creación del personaje director

Una vez terminadas las funcionalidades del camarista, se procedió a crear el personaje que sería manejado por el director. Para el jugador controlado por el director de la escena hice varias pruebas de movimiento con los diferentes tipos de actores que existen en Unreal Engine. El objetivo de las pruebas era comprobar qué clase era la mejor para utilizar como base del personaje. Principalmente las pruebas se basan en analizar los movimientos y las funcionalidades multijugador del actor.

En primer lugar, se probó con el actor “default pawn”(peón por defecto). La clase peón es una subclase de la clase actor. Es la subclase básica de la clase actor preparada para actuar como un personaje en la escena y ser controlada. Este control puede ser un jugador o una IA.

La clase “default pawn” dispone de un sistema de movimiento ya implementado. Este movimiento consiste en un movimiento mediante las teclas w,a,s,d para el movimiento del personaje y rotar la cámara del jugador usando el ratón. El movimiento era el apropiado para el proyecto ya que permitía mover al personaje tanto con el ratón y teclado como usando la tableta.

Al colocar la ventana del director en la tableta la propia ventana permite un joystick virtual táctil, que puede ser controlado con el lápiz de la tableta. El resto de la pantalla táctil permite rotar al personaje.

El movimiento de la tableta era muy errático ya que la tableta era muy sensible y hacía que el actor rotase de forma incontrolada. El principal problema por el que se descartó el uso de este actor fue la respuesta al intentar la comunicación entre este actor y el que controlaba el camarista con las gafas VR.

Se hicieron pruebas con otro tipo de actores de la clase peón. Otro peón que se probó fue el spectator pawn. Esta clase tiene las mismas habilidades que el default pawn pero añade nuevas funcionalidades dirigidas a como su nombre indica para ser empleado como espectador. Dado que el objetivo del director es tener una vista general de la escena era una opción que podía ser acertada. No obstante, los resultados obtenidos para la interacción eran similares al del default pawn.

Finalmente, utilizando un proyecto ejemplo[18] con conexión entre los actores se eligió la clase character como base para el director.

La clase `character` es una subclase de la clase `peón` además de componentes para su movimiento y animación incluye implementaciones básicas de un sistema de conexión multijugador. La clase también está orientada a ser controlada por un jugador y preparada para que los personajes puedan caminar, correr, nadar y volar.

Tomando el `character` del proyecto ejemplo se creó un personaje al que se le agregaron las funcionalidades del director.

Conexión multijugador

Además de poder elegir el tipo de ejecución que el jugador desea, como se ha explicado en el apartado sobre los tipos de ejecución, también permite elegir el número de jugadores y la ejecución para cada uno de ellos.

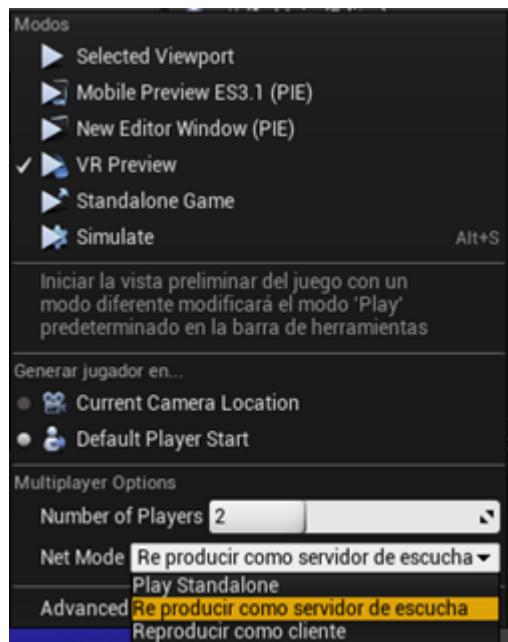


Figura 13 Opciones Multijugador

Como se puede observar en la imagen Unreal Engine permite elegir entre tres tipos de conexión.

- **Play Standalone:** El proyecto crea dos ejecuciones por separado. No existe ningún tipo de conexión entre los dos jugadores, no existe un servidor al que se conecten y ninguno de ellos puede actuar como servidor.
- **Reproducir como servidor de escucha:** Uno de los jugadores actúa como cliente y como servidor. El otro cliente se conecta al que ejerce de servidor.
- **Reproducir como cliente:** Los jugadores se crean como clientes. Se lanza un servidor dedicado al que los jugadores se conectan. El servidor se ejecuta fuera de las escenas.

Network	Description
Mode	
Standalone	The game is running as a server that does not accept connections from remote clients. Any players participating in the game are strictly local players. This mode is used for single-player and local multiplayer games. It will run both server-side logic and client-side logic as appropriate for the local players.
Client	The game is running as a client that is connected to a server in a network multiplayer session. It will not run any server-side logic.
Listen Server	The game is running as a server hosting a network multiplayer session. It accepts connections from remote clients and has local players directly on the server. This mode is often used for casual cooperative and competitive multiplayer.
Dedicated Server	The game is running as a server hosting a network multiplayer session. It accepts connections from remote clients, but has no local players, so it discards graphics, sound, input, and other player-oriented features in order to run more efficiently. This mode is often used for games requiring more persistent, secure, or large-scale multiplayer.

Figura 14 Tipo de Red en Ejecución

De acuerdo con documentación del motor gráfico, cualquier cambio que se produzca en el servidor es comunicado a todas las entidades que actúan como clientes. Conociendo esta información, y tras realizar pruebas con los tres tipos de ejecución, se eligió como modo de la conexión la segunda de las tres opciones.

Como se refleja en la tabla en la entrada Listen Server o servidor de escucha es la opción que se encuentra orientada para juegos multijugador a pequeña escala orientados a acción cooperativas o competitivas. En el caso de nuestro proyecto únicamente existen dos jugadores y el jugador VR se encuentra ya en la escena en el momento de iniciar la ejecución por lo que consideramos que era la opción que mejor se ajustaba a las necesidades de mismo.

Al elegir reproducir como servidor de escucha el personaje del camarista ejerce las funciones de servidor además de las de cliente. De esta manera, la conexión entre desde el camarista al director se encuentra ya generada. Los cambios en las cámaras y los movimientos del personaje movido por las Oculus Rift son visibles por el usuario que se encarga de la dirección de la escena.

Para realizar la conexión inversa, es decir transmitir los cambios del director al camarista, se pueden utilizar diferentes herramientas.

Para realizar la implementación mediante C++ se utilizan los siguientes conceptos:

- Replicación de variables
- Replicación de movimiento
- RPCs(Remote Procedural Calls)
- Replicación de componentes

Implementación

Introducción

Un proyecto de Unreal Engine implementado en C++ implica que todos los objetos que realicen una acción o interacción en la escena son clases que contienen el código de dicha función.

Tal y como se ha explicado la clase básica se denomina actor, la cual puede crearse, destruirse, y ser modificada....

Las subclases de la clase actor son clases que tienen como clase padre a esta, por lo que incluyen las funciones y atributos de la misma. Las dos subclases de actor más importantes son la clase pawn(peón) y la clase character. Además, de las clases actor encontramos las clases componente que son aquellas que en conjunto forman a un actor.

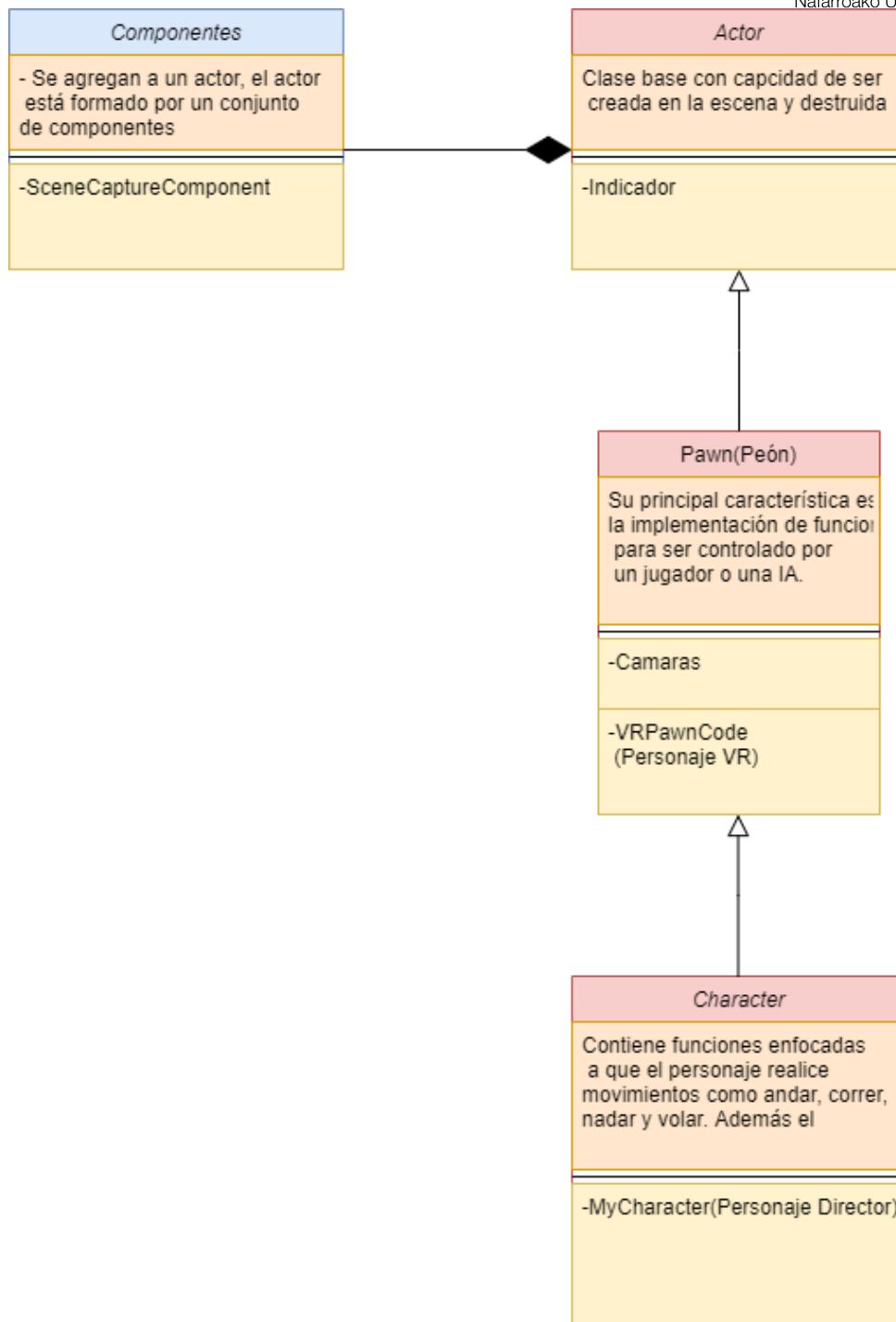


Figura 15 Diagrama de clases de Unreal Engine

La figura anterior no refleja como tal un diagrama de clases ya que no se muestran los atributos y métodos de cada clase, sin embargo, muestra la jerarquía y composición de las clases de Unreal Engine. En el apartado naranja encontramos una escueta explicación sobre la clase y que la diferencia de las otras cuatro clases de la figura. El apartado inferior, en color amarillo, se encuentran las clases del proyecto que han empleado dicha clase como clase padre.

La clase actor consta de tres funciones base.

- Un constructor: Esta función se utiliza para la creación del actor. Se compila al abrir el proyecto. Es utilizada para crear y añadirlas componentes al actor. Algunas acciones únicamente pueden ser realizadas en él. Las características implementadas en el constructor no dependen de la ejecución del proyecto de Unreal Engine, es decir que los cambios en el constructor se pueden comprobar sin la necesidad de lanzar la escena.
- BeginPlay(): Es la función que se ejecuta cuando un actor aparece en la escena. Únicamente es llamada una vez. Suele ser utilizada para configurar parámetros que requieren de elemento de la ejecución y que no es necesario completarlos en el constructor.
- Tick(): Se ejecuta en cada frame del programa. Se emplea para llamar a las funciones que se deben ejecutar en función de los eventos en la escena.

Cuando el programador de un proyecto quiere crear una clase, Unreal Engine ofrece la posibilidad de tomar como clase padre una de las clases que ya existen en el proyecto. Estas clases pueden ser clases base del motor gráfico o clases previamente creadas por el usuario.

De esta forma cualquier modificación que se quiera realizar sobre una clase actor debe ser realizada mediante instrucciones de C++.

Personaje VR (Camarista)



Figura 16 Personaje Camarista

Para la creación del personaje se eligió crear una subclase de la clase pawn(peón). Esta clase, denominada en el proyecto VRPawnCode, será la que implemente tanto las acciones del personaje controlado por las Oculus Rift como su apariencia en la escena. Entiéndase por apariencia al modelado, materiales, colisiones... Para implementar el movimiento de las manos y su modelado seguí un tutorial.[2] En la creación de la estructura básica del personaje, que he utilizado del tutorial intervienen tres funciones: `CreateComponents`, `CreateHandController`, `CreateHandMesh`

`CreateComponents`: Se encarga de crear las componentes que conformaran el personaje VR, además configura la posición y rotación de las mismas. No es necesario ya que estas acciones podrían realizarse directamente en el constructor de la propia clase, pero aún en una única función toda la creación del personaje. Entre las funciones que llama encontramos `CreateHandController` encargada del propio movimiento de cada mano. Se llama a esta función dos veces, una para cada controlador.

Movimiento

El movimiento está implementado en la función `CreateHandController`. En esta función se asigna el movimiento de los controladores de las Oculus Rift. La función utiliza las clases `FXRMotionControllerBase`, que es la clase básica de implementación de movimiento de un `motionController`, y `UMotionControllerComponent`, clase componente que permite añadir a un personaje este tipo de movimientos.

`CreateHandController` toma tres parámetros de entrada:

- La componente del personaje que va a ser componente padre del controlador. Esto implica que acciones que se realicen sobre la componente padre como rotación y movimiento también afectan a la componente hija. En este caso se toma como padre la componente raíz del personaje, por lo tanto, si la componente raíz se mueve o elimina, ocurre lo mismo con la componente UMotionControllerComponent.
- Texto de la clase FName para nombrar e identificar a la componente en la jerarquía de componentes del personaje
- El id del identificador del controlador que se está usando.

Una vez tiene implementado el movimiento, llama a la clase CreateHandMesh que asigna el modelado de la mano dependiendo si es izquierda o derecha.

Punteros

Una vez implementadas los controladores y el movimiento de la cámara, introduce unos punteros laser para mejorar la precisión de las acciones del jugador. Para la creación de los punteros se utilizó la función DrawDebugLine de Unreal. Esta clase permite dibujar líneas en la escena que son visibles durante la ejecución del programa. Entre los parámetros de la función encontramos:

- La variable que identifica la escena
- El punto de comienzo de la línea
- El punto final de la línea
- El color de la línea
- La persistencia: Se trata de una variable booleana que indica si una vez dibujada la línea esta se mantiene en la escena o desaparece tras un tiempo.
- Tiempo de vida: Tiempo que la línea permanece en la escena en el caso de ser dibujada
- Grosor de la línea: Variable de tipo float que indica el grosor de la línea dibujada.

A parte de DrawDebugLine también se utiliza la función LineTraceSingleByChannel. Esta función permite saber si existe un objeto entre dos puntos. Con esta clase podemos saber si en el rayo del puntero existe algún objeto y cual es.

Acciones

Introducción

Los actores que pueden ser controlados por un jugador tienen definida la función SetupPlayerInputComponent. Esta función recibe como parámetro de entrada la instancia de la clase UInputComponent que monitoriza los inputs del jugador.

UInputComponent dispone de las funciones **BindAction** y **BindAxis**. Ambas sirven para ejecutar funciones cuando se recibe un input. BindAction dispone los siguientes parámetros de entrada:

- **ActionName:** Es un parámetro de tipo FName. Similar a un String, registra el nombre de la acción.
- **KeyEvent:** Recibe valores del enumerado EInputEvent, con valores {IE_Pressed = 0, IE_Released = 1}. Sirve para saber si la función se llama cuando se presiona o se suelta un botón o una tecla.
- La clase que realiza la función.
- La función que debe ejecutarse cuando la acción es lanzada.

Una vez se ha definido la función BindAction es necesario enlazarla al proyecto mediante los ajustes de proyecto de Unreal.



Figura 17 Ajustes de Proyecto: Mapa de Botones

En los ajustes de proyecto se crea un nuevo mapeo de una función, se le da como nombre el ActionName de la llamada de BindAction y se asigna uno o más botones a los que se relaciona. Como se observa en la figura 5, se pueden usar varios niveles de mapeo ya que se pueden utilizar combinaciones de teclas. Por lo tanto, puede existir

una acción que se ejecute cuando se presiona una tecla y otra acción diferente cuando se presiona esta tecla sumada a una tecla especial. Estas teclas especiales son las que se ven en la figura 5: Shift, Ctrl, Alt, cmd.

De esta forma, cuando se presione o suelte el botón establecido en los ajustes, se consulta la BindAction que concuerde con su ActionName y con el KeyEvent, y se ejecuta la función correspondiente.

La función BindAxis es similar a BindAction pero, se utiliza para inputs no binarios, es decir, para inputs con más de dos valores, por ejemplo, el valor de un input que sea un eje. Las tres diferencias principales son:

- No recibe como parámetro el KeyEvent.
- La función a la que llama debe estar definida con un parámetro de entrada correspondiente al valor de los ejes.
- En los ajustes del proyecto en lugar de permitir combinaciones de teclas permite establecer un valor entero que multiplicará el valor recibido como input.

Las acciones que se pueden realizar con el personaje utilizando BindAction son:

CREAR UNA CÁMARA

Con el objetivo de crear una cámara utilizamos la función SpawnActor que permite establecer la clase que se va a crear, así como: su posición, rotación y otros parámetros de spawn. Estos parámetros incluyen: las colisiones, el dueño del objetivo (útil para los permisos)...

Utilizamos un BindAction, de tal manera que si se pulsa el botón se llama a una función que tiene como parámetro de entrada la posición final del DrawDebugLine de la mano izquierda. La función consulta la variable contcam, una variable de tipo entero que almacena el valor de la próxima ventana a la que se va a asignar una cámara. El entero contcam toma valores de 1 a 4 y dependiendo del valor se utiliza la función SpawnActor con la clase cámara correspondiente tomando como posición de spawn la posición final del DrawDebugLine.

ROTAR UNA CÁMARA

Para rotar las cámaras es necesario que la función LineTraceSingleByChannel de una de las manos del jugador detecte una cámara como el objeto al que se está apuntando, de esta forma, se puede almacenar en una variable actor la cámara en cuestión. Registraremos el input del jugador utilizando tres funciones BindAction, una para cada eje en el que se puede girar. En caso de que se presione el botón corresponde, se modifica la rotación de la cámara añadiendo al eje el valor de una variable de tipo

float llamada *giro*. Para incrementar el ángulo de giro se llama a la función *SetActorRotation* del actor devuelto por *LineTraceSingleByChannel*. Al comienzo de la ejecución, la variable *giro* toma valor 1, usando una cuarta *BindAction*, si el jugador pulsa un botón puede modificar el valor de giro de 1 a -1 y viceversa. Esta acción, como es razonable, cambia el sentido de rotación de la cámara.

MOVER UNA CÁMARA

Al igual que en el caso de la rotación el jugador debe estar apuntando a una cámara, en este caso, con su mano izquierda. Capturando mediante *LineTraceSingleByChannel* dicha cámara, se transforma su posición actual por la del valor final del *DrawDebugLine*. Para únicamente afectar a la posición y no a la rotación, se almacena la rotación antes del cambio y se le asigna a la cámara después de que se produzca. En caso de que no se tenga esto en cuenta, es posible que la rotación de la cámara vuelva a la que tiene la cámara por defecto. La cámara modifica su posición mientras el jugador la esté apuntando y pulsando un botón simultáneamente.

ELIMINAR UNA CÁMARA

Similar a las dos acciones anteriores si el usuario apunta a una cámara y pulsa un botón, se llama a una función que mediante el método *DestroyActor* de la propia cámara la elimina por completo de la escena.

Cámaras

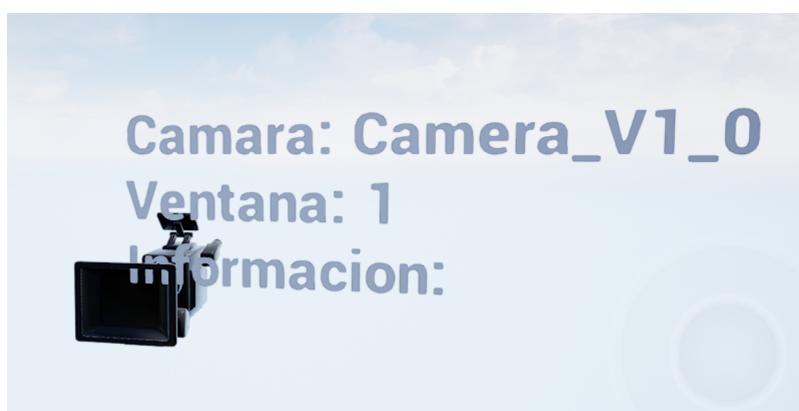


Figura 18 Ejemplo de cámara

Las cámaras empleadas en la escena también son clases C++. Estas clases son tipo Pawn, sin embargo, podrían ser clases actor ya que no son controladas ni por el jugador ni una IA. La elección fue simplemente por preferencia personal.

En un primer momento la implementación de las cámaras estaba incluida en una única clase llamada **Camera**. En esta clase se implementaban todas las componentes necesarias para el funcionamiento correcto.

La clase Camera incluye la componente raíz del actor, la componente que se encarga del modelado de la cámara y un texto que contiene la información de la cámara.

La información que se incluye en el texto es:

- El nombre de identificación de la cámara en la escena.
- La ventana a la que está asignada la cámara
- Información técnica de la cámara

No obstante, existía un problema con la creación de la componente PCSceneCaptureComponent2D. En el apartado 7.4 se explica que se crearon clases diferentes de PCSceneCaptureComponent2D para cada plugin. Esto permite nuevas funciones, como, por ejemplo, que el camarista elija qué ventana quiere que le corresponda a una cámara en concreto.

La componente PCSceneCaptureComponent2D, al igual que el resto, solo puede ser creada en el constructor de la clase. Uno de los inconvenientes de Unreal Engine es que no permite incluir parámetros de entrada en los constructores de los actores. A pesar de que existen formas de introducir parámetros de entrada al crear un actor, estos valores se tienen en cuenta en la función BeginPlay(). Debido a esto no se podía elegir la ventana, ya que cuando se llamaba a la función BeginPlay, con la ventana elegida como parámetro, o ya existía una componente PCSceneCaptureComponent2D creada o ya no se podía crear.

Para remediar este problema se introdujo una jerarquía de clases.

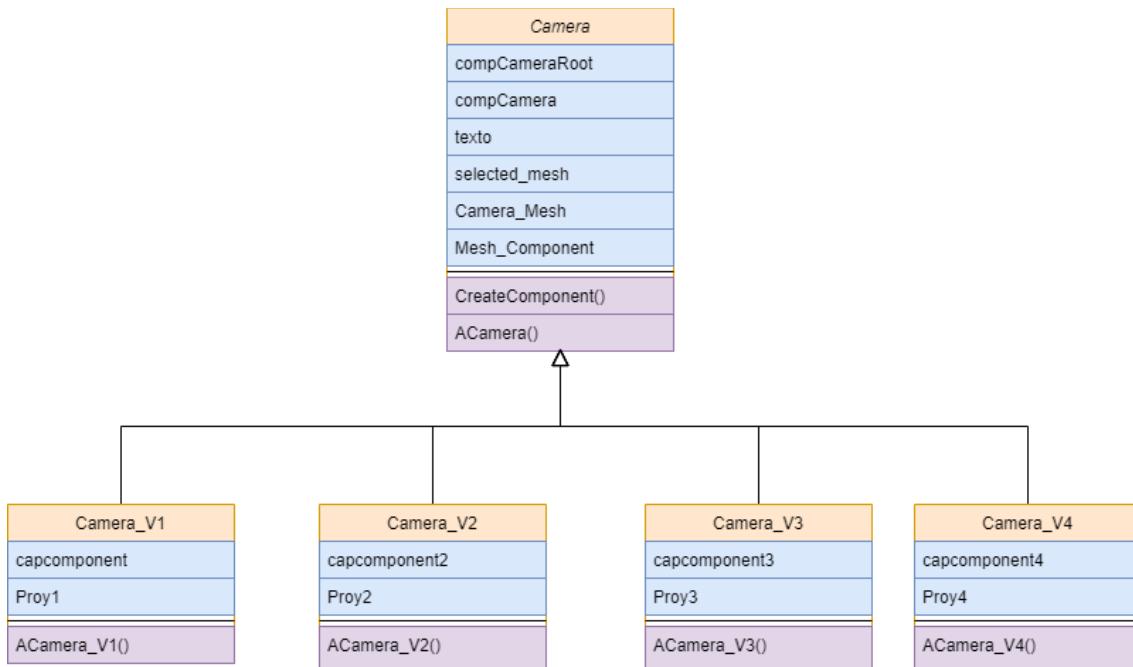


Figura 19 Jerarquía de las clases Camera

De esta forma, todas las componentes no relacionadas con la visualización en las ventanas están implementadas en la clase Camera. En la clase camera se añaden el modelado, las colisiones y la componente de texto.

En el constructor de las clases que heredan de Camera se crea la componente SceneCaptureComponent2D correspondiente y se le asigna la textura que necesita la componente. Además, las clases hijas son las que inicializan el texto de la cámara para que tenga la información de la ventana correcta.

Las cámaras pueden ser creadas durante la ejecución, pero también pueden ser creadas desde el editor. Esta opción permite ejecutar el proyecto con cámaras ya creadas en caso de que se desee.

Las cámaras han sido programadas para poder replicarse lo que permite que se modifiquen en proyectos multijugador.

Personaje Director

Como ya hemos mencionado la clase que implementa el personaje controlado por el director es una clase `character`. Para implementar el movimiento del actor utilicé un tutorial que mostraba como crear un actor con la capacidad de volar.[3]



Figura 20 Personaje Director

Movimiento

Para la implementación del movimiento se utilizan cinco funciones `BindAxis`. Tres de las `BindAxis` son para el movimiento del personaje en los tres ejes, y otros dos para la rotación.

En el caso de las funciones del movimiento tiene asociados dos botones, uno multiplicado por un entero unitario positivo y otro negativo. De esta forma, si se pulsa el botón multiplicado por uno, el personaje se mueve en el sentido positivo del eje. En el caso de los botones multiplicados por el entero negativo se mueven en el sentido negativo del eje. Si en lugar de utilizar un botón para enlazar una `BindAxis` se utiliza un periférico que ya tiene registrados unos ejes, no es necesario especificar cuál es el sentido positivo y cuál es el negativo, excepto que se quiera cambiar el sentido definido por defecto. Por ejemplo, al utilizar el ratón del ordenador, Unreal Engine tiene registrado que existen dos ejes x e y. En el eje x tiene configurado que mover el ratón hacia la derecha es un input positivo y hacerlo hacia la izquierda es un input negativo. Si el jugador mueve el ratón hacia arriba es un input positivo y hacia abajo negativo.

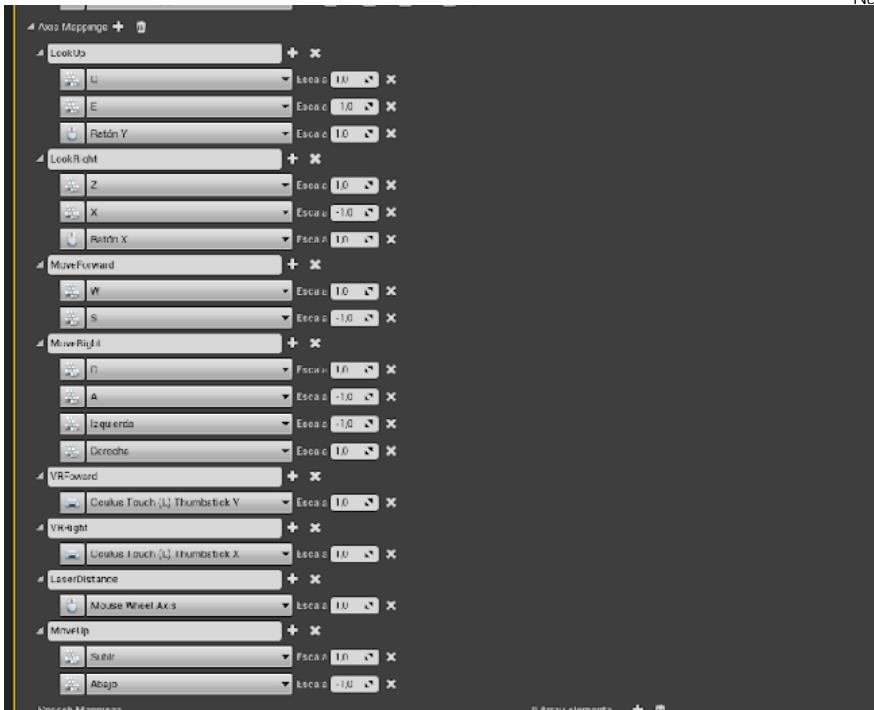


Figura 21 Axis Mappings

La implementación de la rotación es similar. Únicamente se utilizan dos `BindAxis` ya que no interesa que el personaje rote sobre su eje X. El objetivo de la rotación es que el jugador puede mirar a su alrededor en la escena.

Las funciones que se ejecutan cuando se accionan los botones utilizan la función `AddMovementInput`, método implementado por los actores de tipo Pawn. Esta función recibe como parámetro el vector dirección sobre el que se quiere aplicar el movimiento y el valor input del eje.

Puntero

La implementación del rayo láser del personaje director es similar al creado para cada una de las manos VR del camarista con dos diferencias:

- Mediante una `BindAction` se puede activar o desactivar.
- El valor que se pasa como parámetro de final del rayo a la función `DrawDebugLine` depende del valor recibido por un input de un `BindAxis`. Esto implica que se puede regular la distancia a la que se encuentra el punto final. Esta habilidad es útil ya que, este punto es el que se toma como coordenada para colocar el resto de los marcadores.

Acciones

Para llevar a cabo cada una de las acciones disponibles para el director, se tienen que cumplir una serie de condiciones. Como si de un diagrama de estados se tratara las han implementado de tal forma que para que algunas de ellas se puedan realizar se bloquean otras, de esta manera el usuario puede realizar una acción sin miedo a equivocarse y realizar otra diferente.

Para explicar mejor esta relación tenemos el siguiente diagrama de estados.

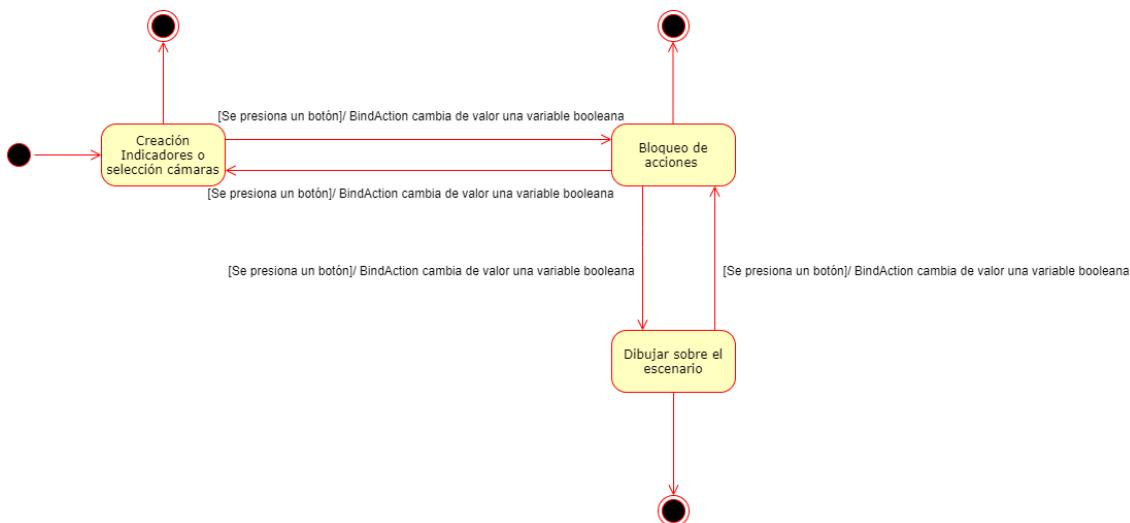


Figura 22 Diagrama de Estados Director

El director puede crear indicadores en la escena y resaltar una cámara para dar una indicación al camarista. Si el director pulsa un botón bloquea estas acciones por lo que únicamente puede utilizar el puntero. Esta situación de bloqueo es el punto intermedio entre los dos tipos de acciones: dibujar y colocar marcadores.

En este punto si el director quiere dibujar debe pulsar un botón para pasar a ese estado, si pulsa el primer botón vuelve al estado inicial de colocar marcadores.

Durante todo este proceso el director puede modificar tanto el tamaño del puntero como si quiere que sea visible en cualquier momento o únicamente cuando realice una acción.

Seleccionar una cámara

Con la intención de dar indicaciones más precisas sobre que cámara se está hablando en concreto, hemos implementado la posibilidad de seleccionar una cámara. Al hacerlo esta cambia de color a un color verde. Para seleccionar una cámara el director hace click con el botón izquierdo del ratón o hace click con la tableta apuntando con el puntero a una cámara en concreto. Esta acción se implementa mediante una `bindAction`. La función asociada a la `BindAction` cambia el modelado de la cámara a un modelado de color verde. En el caso en el que hubiera otra cámara seleccionada, la cámara que estaba seleccionada vuelve a cambiar al modelado original. Gracias a esto en la escena no hay dos cámaras seleccionadas en la escena al mismo tiempo y no se produzcan confusiones. Si el jugador selecciona una cámara que ya se encontraba resaltada esta vuelve a su modelado original.

Para modificar el modelado he implementado dos funciones diferentes. Una llamada `change_mesh` y otra `reset_mesh`.

La función `reset_mesh` recibe como parámetro de entrada un actor, que es una cámara, y le asigna el modelado por defecto de la cámara.

Para poder utilizar esta función con el actor adecuado utilzo una variable de la clase `actor` que almacena la última cámara seleccionada.

La función `change_mesh` recibe como parámetro de entrada un actor y cambia el modelado contrario, entre el seleccionado y el por defecto, al que tienen en ese momento.

Creación de Indicador

Cuando el director pulsa un botón o en su defecto hace click sobre la tableta se crea un indicador en la escena en el punto final del puntero. Para crear el actor en la escena se llama a la función `SpawnActor`. Como ya hemos explicado en la creación de las cámaras `SpawnActor` permite instancia un actor de una clase en la escena con unos parámetros como la posición, la rotación y parámetros de colisión.

No obstante, no es la única función que se utiliza para esta tarea. Existe una segunda un segundo caso en el que se crear un indicador. El objetivo principal de los indicadores es marcar posiciones en las que colocar una cámara. Para implementar esta idea los indicadores están implementados con un texto. En el caso en el que una cámara este seleccionada en la escena, al crear el indicador este lleva un texto con el nombre de la cámara seleccionada.

La función utilizada para crear el indicador con texto no es `SpawnActor`, sino `BeginDeferredActorSpawnFromClass`. Esta clase permite crear un actor, pasar parámetros de entrada que se asignan en la función `BeginPlay` y `FinishSpawningActor` para finalizar la creación hacerla efectiva. Se emplea esta función para poder pasar como parámetro el nombre de la cámara seleccionada.

Eliminación de Indicador

Para eliminar un indicador se emplea la misma función que para eliminar una cámara. Si el jugador hace click sobre un indicador ya creado este se elimina.

Bloqueo de acciones

Como se muestra en la Figura 8 el bloqueo de las acciones sirve de punto intermedio y como método de seguridad para no realizar acciones sin querer. Como se muestra en la imagen se implementa mediante una `BindAction`.

Dibujar en el escenario

Para dibujar en el escenario se utiliza la función `DrawDebugLine` que empleamos para los punteros. En este caso, se cambia el parámetro de permanencia a true, de esta manera lo que se dibuja se mantiene en el escenario. Los límites de la línea que se dibujada la he implementado utilizando el valor final del puntero. Dado que el escenario se actualiza cada frame, la forma en la que he programado el recorrido de la línea es tomar el valor final del puntero y dibujar una línea entre este punto y el punto final de la anterior iteración.

Mediante otra `BindAction` he programado que si se presiona una tecla se cambie de color entre cinco colores.

Para esto he empleado un switch que en función de la variable `actualColor`, que cambia mediante la `BindAction`, llama a `DrawDebugLine`. Dentro de una misma línea no se puede cambiar de color, ya que lo he programado para que únicamente actualice el color cuando se presiona el botón y el usuario no está dibujando.

Borrar Dibujado

Para borrar lo dibujado en el escenario utilice la función `FlushPersistentDebugLines`. La cual borra todas las líneas persistentes dibujadas utilizando `DrawDebugLine`.

Indicador



Figura 23 Indicador con la componente de texto

El indicador se encuentra codificado en la clase actor del mismo nombre. Se trata de un actor que sólo implementa un texto, una variable de tipo FName llamada camara y el modelado. En el modelado se incluyen la malla de colisiones del actor.

Estas dos componentes se crean en la función CreateComponent, que es llamado en el constructor. Una vez ejecutado el constructor, lo primero que comprueba en la función BeginPlay es si la variable cámara ha sido modificada por el director durante la llamada a BeginDeferredActorSpawnFromClass, si se añade el contenido de la variable al texto. Por el contrario, si no ha sido modificada se eliminan la componente texto del actor.

Conexión multijugador

Como se ha explicado en apartados anteriores, para la implementación del sistema de multijugador del proyecto se han utilizado los conceptos de replicación de variables, RPCs (Remote Procedural Calls), replicación de movimiento.

En Unreal Engine para que cualquier actor que se encuentre en la escena pueda modificarse para varios jugadores debe ser replicable. Si el actor es replicable cualquier cambio que sufra será visible para todos los jugadores. Cualquier actor tienen un método `SetReplicates`. Esta función recibe como parámetro el valor booleano para modificar el valor de la variable `bReplicates` del actor. Si la variable toma el valor true, el actor es replicable. También se puede configurar la replicación del movimiento con el método `SetReplicateMovement` que permite que se replique el movimiento.

Tras realizar varias pruebas decidí que lo mejor para asegurarme que los cambios realizados en los actores tuvieran el efecto correcto en los dos jugadores era permitir la replicación de todos los actores en la escena. La replicación de los actores permite que se repliquen estos cambios, pero únicamente se trata de una característica necesaria pero no suficiente.

Para hacer efectiva la conexión multijugador entre los dos jugadores he utilizado RPCs[4]. Las RPCs son que se llaman localmente, pero se ejecutan remotamente en otra máquina. De esta forma, el cliente puede llamar a una función que se ejecute en el servidor y los cambios sean visible en este último.

En Unreal Engine la declaración de una función de este tipo se hace utilizando los parámetros de UFUNCTION. Unreal permite utilizar palabras clave en la declaración de las funciones, que se colocan entre los paréntesis de la palabra UFUNCTION. Si no se emplean no es necesario colocar la cabecera en la declaración de la función.

Un ejemplo de esto sería la siguiente declaración:

```
UFUNCTION(Server, Reliable)
void CreateIndicator(FVector position);
```

Figura 24 Declaración de función RPC

En este caso la función está definida e implementada en el personaje director, ya que se trata de la creación de un indicador, sin embargo, se ejecuta en el servidor que como sabemos es el personaje VR.

Dependiendo de donde se desee realizar la ejecución del código existen tres posibles palabras clave: Client, Server, NetMultiCast.

La palabra Reliable que acompaña a la Server en la figura es empleada para asegurarnos de que la función se ejecuta correctamente en la máquina remota.

Otros requisitos necesarios para que se pueda emplear una RPC son:

- La función debe ser llamada desde actores.
- Los actores deben estar replicados.
- Si se quiere ejecutar un código en un cliente desde el servidor sólo se ejecutará en el cliente y no en el servidor.

Durante la implementación de los personajes hemos visto que uno de los parámetros que se puede establecer en las funciones dispuestas para el spawn de los actores era el dueño del actor creado. En estas dos tablas de la documentación de Unreal [4] se muestra cómo afecta el dueño de un actor a la hora de ejecutar una función en una máquina remota.

Función RPC invocada desde el servidor

Actor ownership	Not replicated	NetMulticast	Server	Client
Client-owned actor	Runs on server	Runs on server and all clients	Runs on server	Runs on actor's owning client
Server-owned actor	Runs on server	Runs on server and all clients	Runs on server	Runs on server
Unowned actor	Runs on server	Runs on server and all clients	Runs on server	Runs on server

Figura 25 RPC invocada desde Servidor

Función RPC invocada desde el cliente

Actor ownership	Not replicated	NetMulticast	Server	Client
Owned by invoking client	Runs on invoking client	Runs on invoking client	Runs on server	Runs on invoking client
Owned by a different client	Runs on invoking client	Runs on invoking client	Dropped	Runs on invoking client
Server-owned actor	Runs on invoking client	Runs on invoking client	Dropped	Runs on invoking client
Unowned actor	Runs on invoking client	Runs on invoking client	Dropped	Runs on invoking client

Figura 26 RPC invocada desde Cliente

Cabe recordar que en el proyecto el personaje con las gafas VR es el que actúa tanto como servidor como cliente y el personaje del director únicamente actúa como cliente.

Sabiendo esto, como se puede observar en las dos tablas la comunicación del servidor al cliente es más fácil de implementar que la comunicación inversa.

Para los cambios realizados por el servidor no es necesario llamar a funciones RPC que sean ejecutadas por el cliente.

En el caso del cliente se emplean funciones RPC para los cambios que realiza en el escenario y que se quiere que sean visibles para el servidor.

En concreto se emplea para la creación de los indicadores, el dibujo en el escenario, la selección de las cámaras y poder visualizar el rayo del director en la escena por parte del camarista. El movimiento del propio personaje no necesita de funciones de RPC, como hemos comentado, al ser un personaje de tipo character contiene implementado características multijugador por defecto.

En la función CrearIndicador no se necesita ningún cambio más, sin embargo, en las funciones que en las que interviene se modifica alguna componente como en change_mesh o reset_mesh, en las que se modifica una variable de la cámara, es necesario modificar la declaración de la variable para que esta pueda ser modificada por la función RPC.

La implementación necesaria consiste en primer lugar, incluir en la declaración de la variable la palabra clave Replicated. En el caso de las variables no se emplea la palabra UFUNCTION para definirla, en su lugar se emplea UPROPERTY.

```
UPROPERTY(Replicated)
UStaticMeshComponent* CameraMesh;
```

Cuando una variable se define como Replicated es necesario implementar la función GetLifetimeReplicatedProps[6]. Esta función mantiene un control de todas las variables que han sido replicadas, es un método propio de la clase actor. Si un actor implementa una variable replicada es obligatorio que sobrescriba este método.

Problemas durante el desarrollo

En el transcurso de la investigación y el desarrollo del proyecto he sufrido ciertos problemas, este apartado está destinado a explicar esos problemas y, si ha sido posible, como se han superado.

Problemas con la creación de las cámaras

La creación de las cámaras virtuales y su traslación a las ventanas emergentes ha consumido una parte significante del tiempo de realización del trabajo. Uno de los problemas que han surgido durante el desarrollo de esta parte tenía que ver con la implementación y el uso de la clase `PCSceneCaptureComponent2D` del plugin empleado.

Una vez realizadas las pruebas con el Blueprint, intenté implementar las cámaras en C++. Un problema que surgió es que en un Blueprint se podía añadir la componente del plugin arrastrando la componente a su lugar en el actor, sin embargo, desde código no conseguía añadir la componente al actor. Esto se debía a que no podía incluir el propio plugin en el código a través de los include. Como no reconocía el plugin tampoco reconocía la clase `PCSceneCaptureComponent2D` creada en él. Intenté solucionar el problema de diferentes formas y busqué información sobre cómo acceder a código que se encontraba en un plugin. La solución final fue crear clases que tuvieran como clase padre a las clases `PCSceneCaptureComponent2D`. Estas clases hijas estaban creadas en la carpeta donde se encontraba el resto de los scripts, y gracias a ello, sí eran reconocidas por las clases que utilizaban la componente en su código. Estas clases recibían el nombre de `PCScenCaptureComonent2DCode` y la variación del número de ventana para que fueran distintas entre ellas. Esto permitió implementar las cámaras mediante código y crear cámaras tanto en el editor como en tiempo de ejecución que pudieran mostrar el resultado en las ventanas emergentes.

Problemas con el uso de la tableta

Al probar la tableta para el personaje del director surgió un problema, únicamente con acercar el lápiz de la tableta a la pantalla, ya lo reconocía como un input y la tableta hacía que el personaje rotara de tal forma que no era fácil de controlar el movimiento. Como esto impedía que se pudiera dibujar o colocar marcadores de manera cómoda introduce una acción para bloquear la rotación del personaje. El jugador puede pulsar una tecla y de esta manera bloquea la rotación. Esto solucionaba el problema de la sensibilidad de la tableta, no obstante, supuso otro problema diferente. EL punto final del puntero del personaje depende de la rotación, ya que para conseguir este punto se toma el vector forward del personaje. Esto implica que si se bloqueaba la pantalla independientemente de donde se colocara el lápiz en la pantalla el puntero apuntaría en el mismo sitio, enfrente del personaje, y, por lo tanto, los indicadores, y los dibujos también. Para solucionar este problema pensé en como transforma una coordenada en 2D que, es la que se le puede introducir a la tableta a través del lápiz, a una 3D. Tras probar diferentes funciones utilicé `DeprojectMousePositionToWorld`.



Figura 28 Apuntado del director con rotación bloqueada

Resultados

En los apartados anteriores se ha detallado tanto la descripción como la implementación de las tareas que componen este proyecto. Dado que se trata de un proyecto destinado a realizar una tarea basada en la visualización de una escena, considero que la mejor forma de mostrar los resultados obtenidos es mediante imágenes que muestren las posibilidades que ofrece el proyecto en su estado actual.

Comencemos por mostrar las posibilidades que ofrece el control del personaje camarista.

Camarista

Como función principal el camarista puede colocar cámaras.



Figura 29 Generación de cámara

Como se puede ver en la imagen la cámara se crea en la escena con el nombre de la cámara, la ventana en la que se encuentra reflejada su salida e información de la

cámara. En la demo no se ha colocado la información. No obstante, Unreal permite obtener información del plano como el aspecto de ratio (altura y anchura de la imagen), la rotación de la cámara o el FOV a través del método `GetCameraView` de la componente `SCaptureComponent2D`.

En la escena se han colocado unos planos en los que también se refleja la salida de la cámara. Se han colocado para que el camarista pueda comprobar los planos, pueden estar tanto fijos en una posición como que acompañen al camarista en su movimiento. Si se quiere se pueden eliminar.

El plano generado por la cámara se observa en su correspondiente ventana, en este caso la ventana número 1.

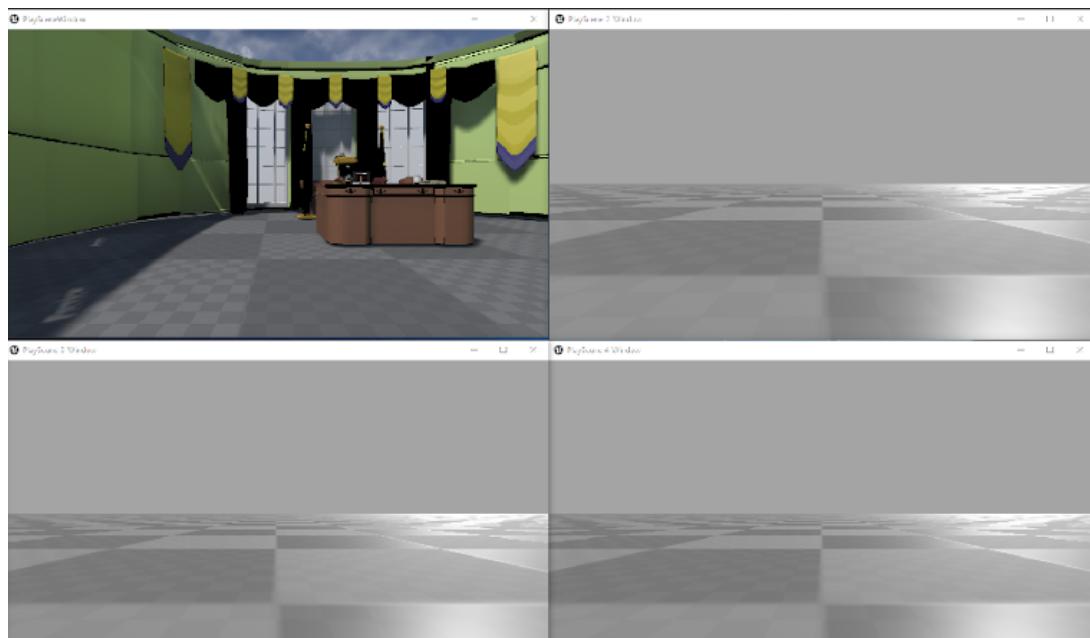


Figura 30 Ventana1 activada

Como se puede observar la primera ventana muestra el plano mientras las otras tres se encuentran desactivadas, lo que se ve en ellas es la salida por defecto cuando no existe ninguna cámara creada.

La cámara puede ser rotada en cualquiera de sus tres ejes y esto se ve reflejado, en el plano de la ventana.



Figura 31 Rotación de la cámara1 sobre su eje Z

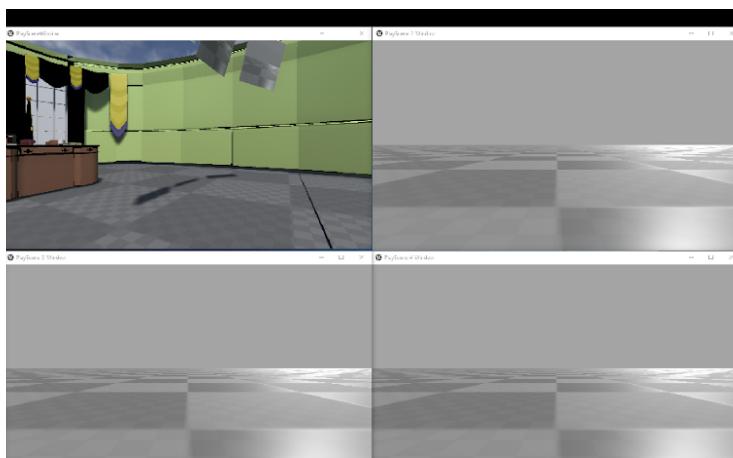


Figura 32 Ventana1 con rotación sobre Eje Z

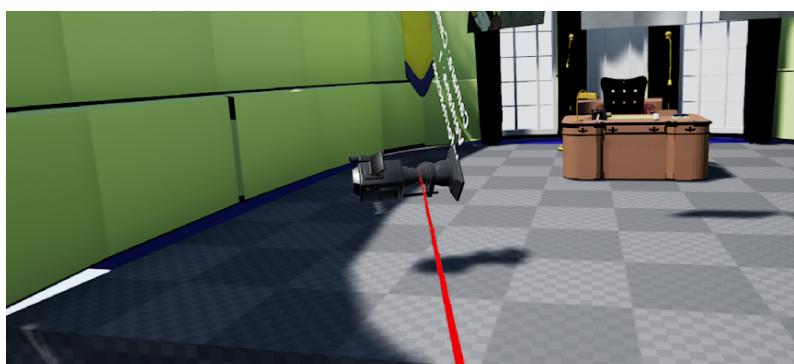


Figura 33 Cámara con rotación sobre Eje Y

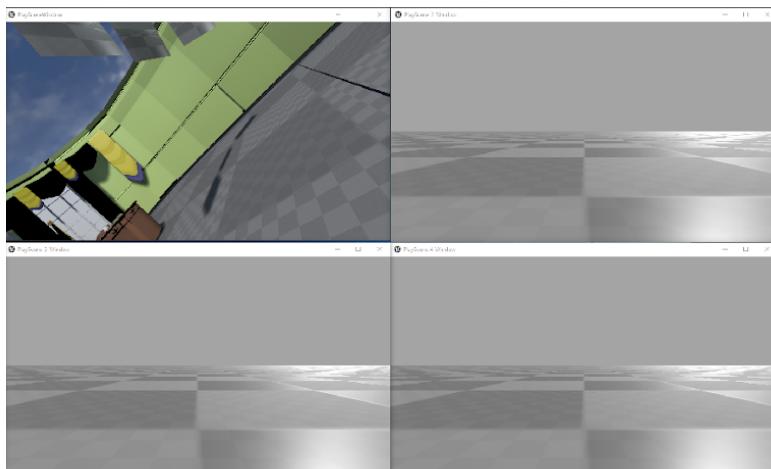


Figura 34 Ventana1 con rotación sobre Eje Y



Figura 35 Cámara1 con rotación sobre su eje X

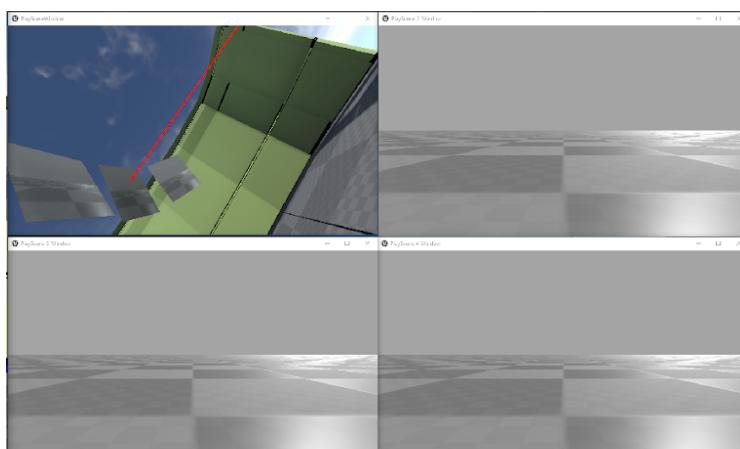


Figura 36 Ventana1 con rotación sobre Eje X

Si se crea una segunda cámara por defecto se crea en la siguiente ventana

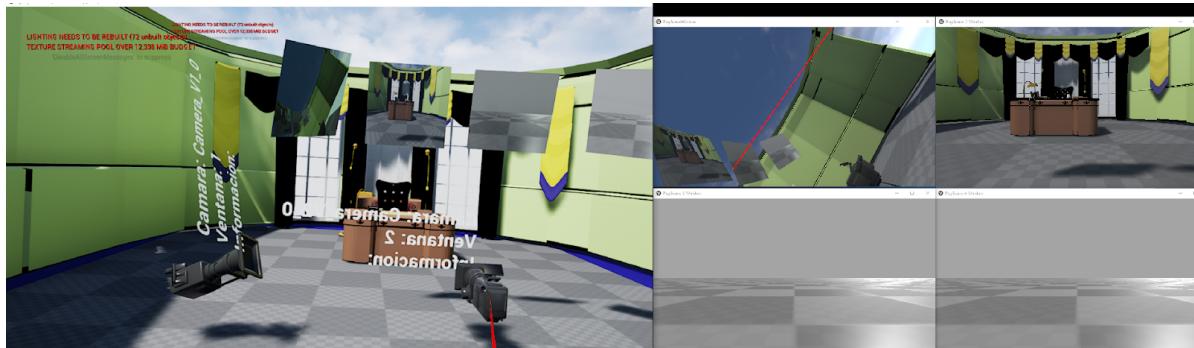


Figura 37 Creación de una segunda cámara

Si el usuario lo desea puede seleccionar en qué ventana crea una cámara. Por ejemplo, en la siguiente imagen se puede observar como el camarista salta la tercera ventana para colocar un cámara en la cuarta ventana.

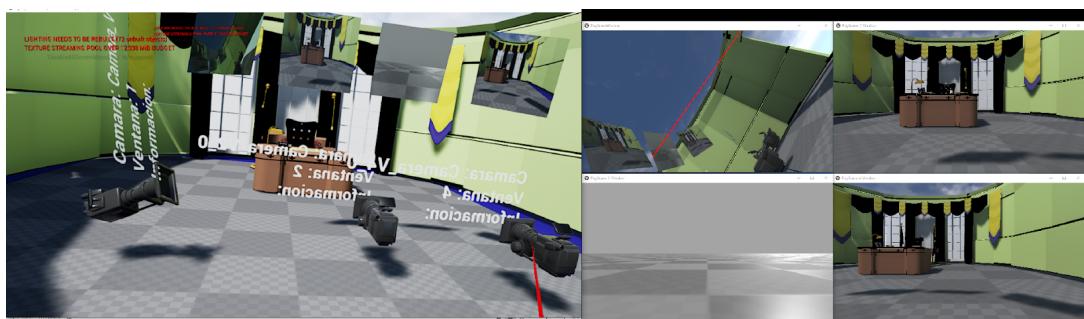


Figura 38 Ejemplo el camarista elige ventana

Si el usuario crea una cámara y le asigna una ventana en la que ya se estaba reflejando el plano de una cámara, la ventana muestra el plano de la última cámara asignada.

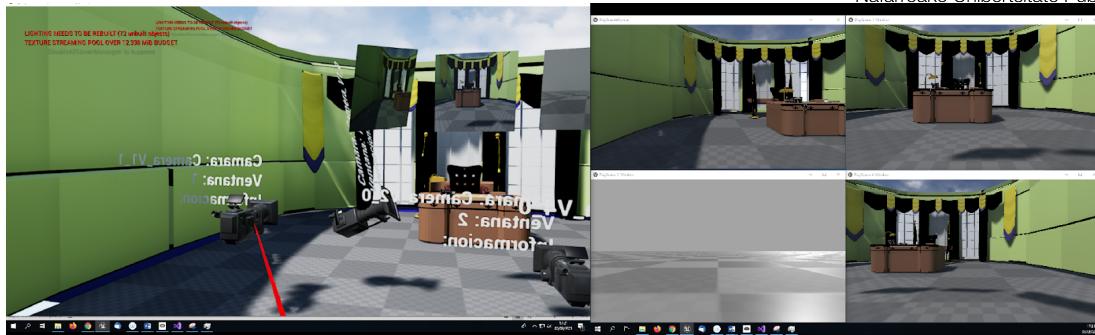


Figura 39 Asignando una cámara a una ventana no vacía

Como se puede apreciar la ventana 1 muestra el plano de la cámara que se acaba de crear y no el de la que se encuentra rotada.

En caso de eliminar una cámara cuyo plano se observa en una ventana y existen otras cámaras asignadas a esta, la ventana vuelve a reflejar la salida de la cámara creada más recientemente.



Figura 40 Cámara eliminada

Director

El director dispone de un puntero que puede activar y desactivar.

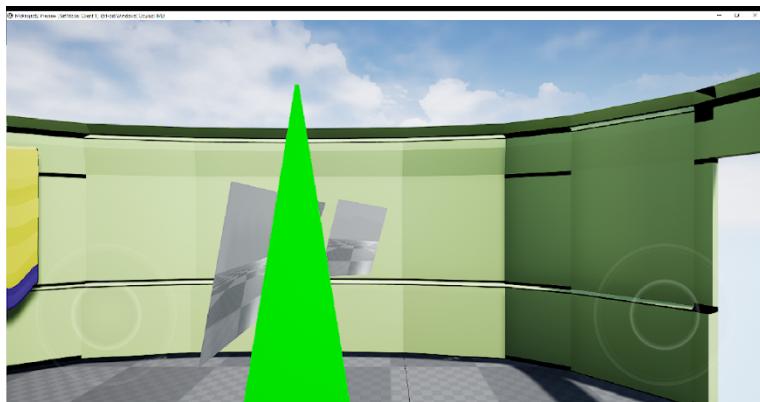


Figura 41 Director con puntero activado



Figura 42 Director con puntero desactivado

Como se muestra en la figura 41 si el director se encuentra con la rotación del personaje bloqueada no se toma el vector forward como dirección para colocar los indicadores, seleccionar cámaras...

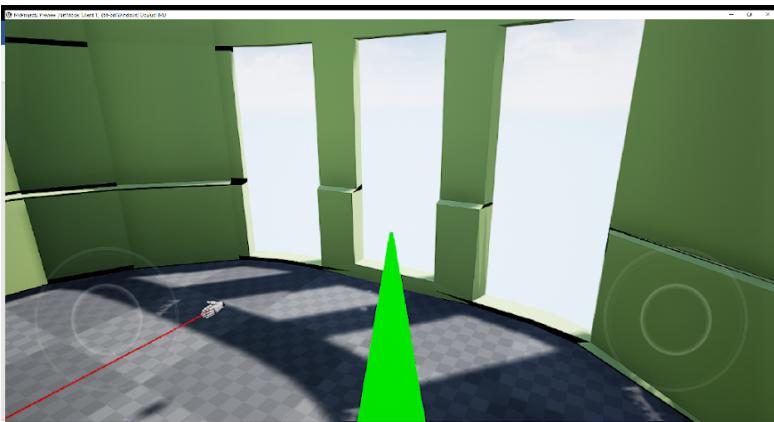


Figura 43 Director con la rotación desbloqueada

El director puede modificar la distancia del puntero utilizando el ratón como se observa en la siguiente figura.

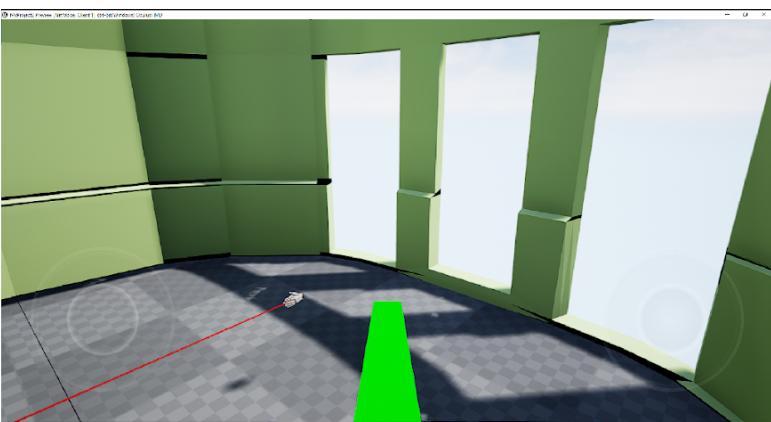


Figura 44 Modificación de la distancia del puntero del director

El director puede colocar indicadores en la escena. Estos indicadores pueden ser eliminados por el mismo.

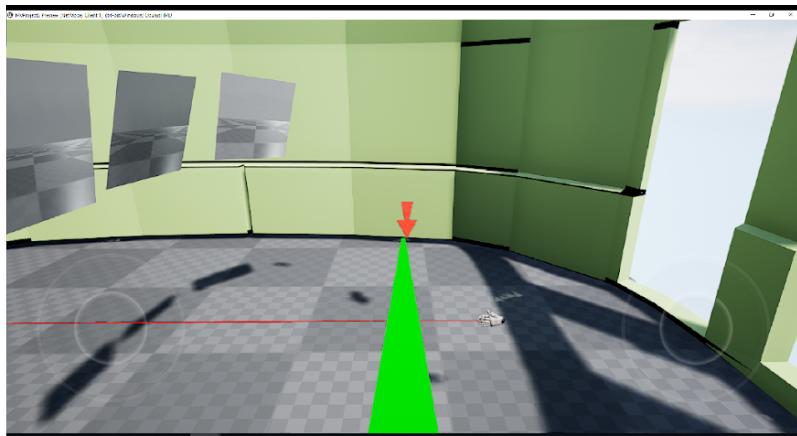


Figura 45 Director coloca un indicador

Todas las acciones realizadas por el director son visibles para el camarista.

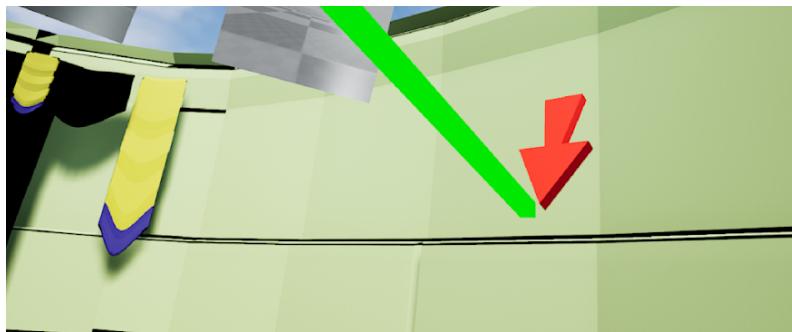


Figura 46 Vista del indicador por el camarista

El director puede dibujar sobre la escena en 5 colores diferentes

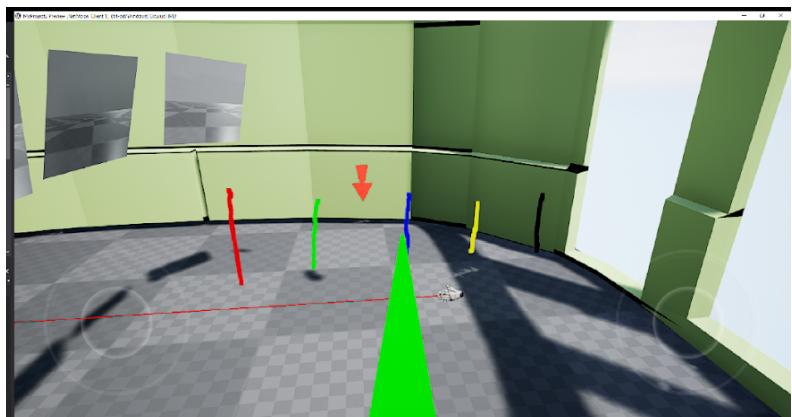


Figura 47 El director dibuja en la escena

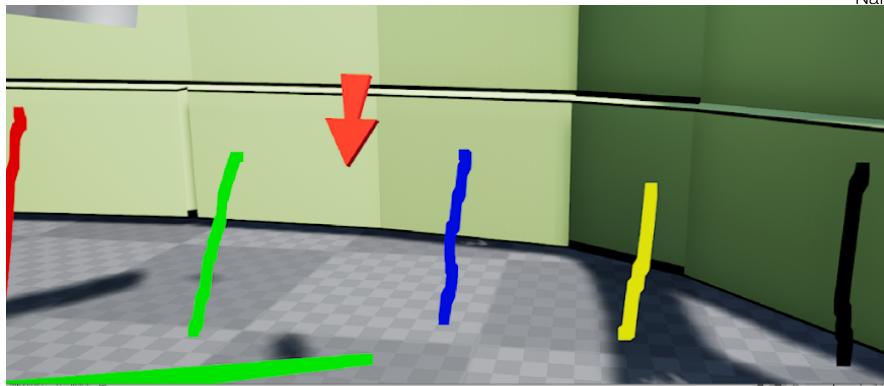


Figura 48 Visión de los dibujos por el camarista

Si el camarista crea una cámara el director puede seleccionarla para indicar de que cámara está hablando.

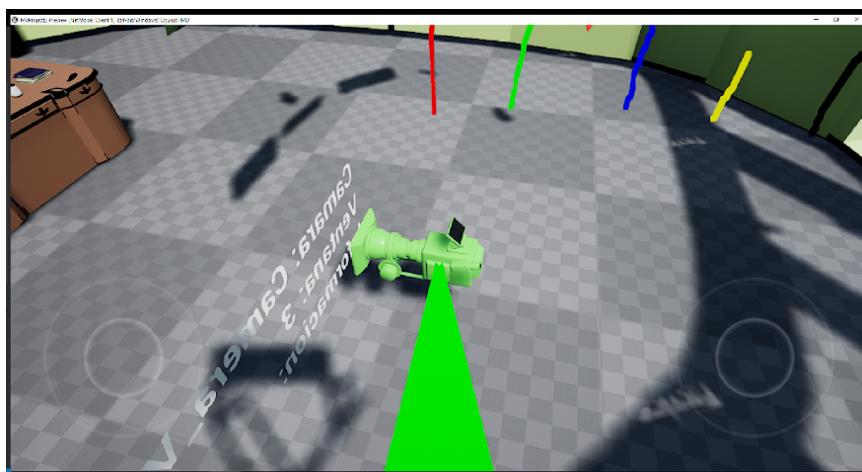


Figura 49 Director selecciona una cámara



Figura 50 El camarista ve la cámara seleccionada

Si en la escena hay una cámara seleccionada y el director crea un indicador, este incluye el nombre de la cámara seleccionada en ese momento, con esto se sabe qué cámara debe colocarse en un indicador.

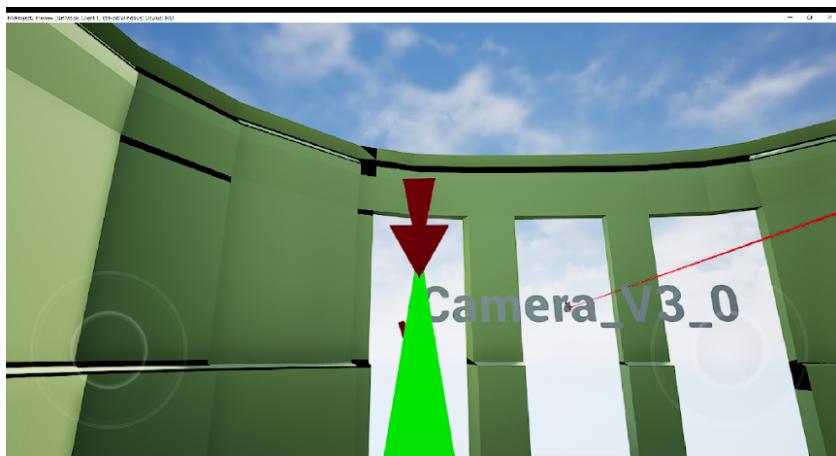


Figura 51 El Director crea un indicador con el nombre de una cámara

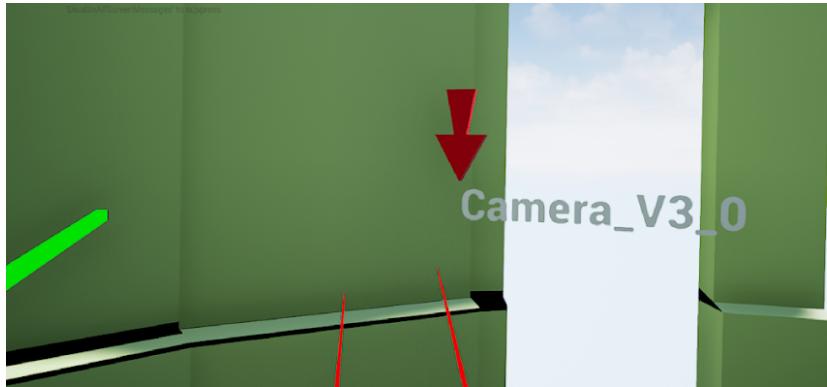


Figura 52 Indicador con texto desde el punto de vista del camarista

Estas son las acciones que se pueden realizar en el proyecto, a pesar de haberse probado otras opciones e intentado introducir nuevas funcionalidades, algunas ideas se han rechazado porque no eran viables o menos eficientes que las utilizadas en el trabajo.

Pruebas descartadas

Una cámara con todas las PCSceneCaptureComponent.

Antes de decidir implementar la jerarquía de cámaras explicada anteriormente, con la intención de poder elegir la ventana de cada cámara o, en su defecto, que se instanciaran de forma secuencial idee una forma de hacer esta misma tarea con una única cámara. La intención era crearla utilizando `BeginDeferredSpawnActor` y pasarle como parámetro de entrada la ventana correspondiente. Tal y como he explicado anteriormente, dado que las componentes se crean en el constructor y los parámetros de entrada se reciben en el `BeginPlay` para cuando se recibía a que ventana debía ir, la cámara ya tenía creada todas sus componentes. Para intentar remediar esto, cree una componente `PCSceneCaptureComponent2DCode` correspondiente a cada ventana. Una vez se llegaba a ejecutar que ventana le correspondía dejaba esa asignada y eliminaba el resto. Descarté esta implementación ya que, aunque eliminara las componentes en el `BeginPlay`, al crear la cámara modificaba todas las ventanas. Este resultado no era estético y podía llevar a equivocaciones al director del proyecto al controlar las cámaras.

Posibles introducciones futuras

Conforme se realizaba el proyecto y al realizar las pruebas en cada fase se han ido anotando posibles acciones que no se encuentran implementadas pero que podrían mejorar la experiencia y funcionalidad.

Estas funcionalidades son las siguientes:

- **Poder restablecer el contenido del RenderTarget2D al color de restablecido.** Actualmente si una ventana no tiene asignada ninguna cámara no retoma el valor por defecto, el color negro, si no que mantiene el último frame capturado por la última cámara que quede asignada a dicha ventana.
- **Introducir más opciones para el control de cámaras** (rotación, cambio de velocidad en los movimientos)
- **Poder modificar la cámara que refleja una ventana cuando una cámara sea seleccionada.** La cámara que se ve reflejada en una ventana es la última cámara que ha sido asignada. Si selecciona o se mueve una cámara en concreto estaría bien que la ventana de dicha cámara reflejara lo que esta cámara está mostrando.
- **Eliminar un marcador cuando la cámara correspondiente se encuentra en su posición.** Dado que la función de los indicadores es señalar la posición a la que debe ir una cámara, creo que una buena introducción en el proyecto sería que si un indicador está señalando la posición a la que debe ir una cámara y la cámara se posiciona en ese punto, el indicador se borre. De esta forma cumple su función y no es necesario que la borre el director.
- **Modificar el zoom de las cámaras.**
- **Crear interfaces de usuario.** Tanto el director como el camarista tienen un gran número de funciones que pueden realizar, además en el caso del director
- **Permitir cargar escenas ya creadas.** Para poder retomar una escena si ya existían cámaras y marcadores colocados en el escenario. A su vez, sería interesante poder guardar las escenas.
- **Poder borrar una línea en concreto o por color.** Es decir, la intención es crear más opciones de dibujado para el director. Actualmente, cuando se borra el dibujo que se encuentra en la escena se borra en su totalidad. Creo que poder borrar un solo trazo o los trazos de un color en concreto podrían ayudar a facilitar las indicaciones del director.
- **Modificar los punteros de las manos del camraista.**

Conclusiones

Llevar a cabo este proyecto me ha aportado mucho a nivel tanto personal como profesional.

Entre los puntos positivos destacaría que me ha permitido ver la profesión desde un nuevo punto de vista, realizando un trabajo más costoso y extenso que una simple práctica. En mi opinión, quizás equivocada, creo que realizar este proyecto es una situación más semejante a un posible trabajo como programador, ya que, he tenido que gestionar y administrar el proyecto dividiéndolo en subtareas a corto plazo por mí mismo sin tener una fecha límite para cada una de ellas. Otro punto que refuerza esta idea, es que he sido yo el que tomaba las decisiones finales sobre la implementación del proyecto por lo que ante las adversidades sufridas he tenido que investigar y barajar las posibles soluciones para cada problema y elegir la que me parecía las más correcta. Considero que ha sido un trabajo muy entretenido y gratificante cada vez que conseguía implementar una nueva acción que funcionaba correctamente.

Por el contrario, como puntos negativos, en algunas ocasiones ha sido frustrante. He realizado pruebas y he investigado sobre ciertos puntos del proyecto sin poder avanzar, lo que por momentos he visto como una pérdida de tiempo y que al ver los resultados finales puede parecer que el esfuerzo ha sido inferior al llevado a cabo. Pienso que algunas de las tareas podrían haberse realizado en menos tiempo y de manera más eficaz.

En relación a los objetivos del proyecto y apartados más técnicos, me agrada a ver cumplido los objetivos marcados. Como he dicho en el apartado de motivación, tenía poca experiencia con la realidad virtual y con Unreal Engine y nunca había programado en C++, creo que desarrollar este proyecto me ha permitido adquirir muchos conocimientos que podré utilizar en un futuro. Como anécdotas de la evolución, al principio del semestre, o cuando introduce el movimiento, me mareaba al utilizar la realidad virtual y ahora mismo puedo utilizarla sin problemas.

Manual de uso

Con la intención de facilitar el uso de la aplicación y el entendimiento de las acciones de las que se disponen he redactado este manual.

Camarista

El camarista dispone de todos los inputs necesarios en los mandos de las gafas Oculus Rift.

Oculus Touch Controllers



Figura 53 Controladores Oculus Rift[22]

Controlador Izquierdo

- **Botón Y:** Permite crear una cámara
- **Botón X:** Apuntando una cámara, permite rotarla sobre su eje Z
- **ThumbStick:** Permite mover al personaje por la escena.
- **Presionar ThumbStick:** Apuntando a una cámara, elimina dicha cámara
- **Botón Trigger:** Apuntando a una cámara, mientras pulse el botón podrá mover la cámara de posición. Puede mover la cámara mientras se mueve el personaje.

- **Botón Grip:** Permite pasar de ventana que se asignará la siguiente cámara creada. Cada vez que se presiona se pasa a la siguiente ventana de la 1 a la 4, cuando llega a la última vuelve a la ventana número 1.

Controlador derecho

- **Botón A:** Apuntando una cámara, permite rotar la sobre su eje X
- **Botón B:** Apuntando una cámara, permite rotar la sobre su eje Y
- **Presionar ThumbStick:** Cambia el sentido de rotación de las cámaras sobre los tres ejes.

Director

El director puede interactuar con la escena usando la tableta o el teclado y el ratón. Para intercambiar entre los modos explicados en la implementación Figura 8 es y el movimiento necesario emplear el teclado.

Teclado

Movimiento

- **Tecla W:** Permite al jugador avanzar en la dirección en la que está mirando.
- **Tecla S:** Permite al usuario retrocede en la dirección en la que está mirando.
- **Tecla A:** Permite al jugador moverse a la izquierda.
- **Flecha izquierda:** Permite al jugador moverse a la izquierda.
- **Tecla D:** Permite al jugador moverse a la derecha.
- **Flecha derecha:** Permite al jugador moverse a la der
echa.
- **Flecha arriba:** Permite al jugador moverse hacia arriba.
- **Flecha abajo:** Permite al jugador moverse hacia abajo.

Acciones

- **Tecla F:** Cambia el modo en el que se encuentra el jugador. Si el jugador está colocando marcadores o seleccionando cámaras y presiona la tecla F bloque esa posibilidad. Si vuelve a ser presionada permite colocar marcadores y seleccionar cámaras.
- **Tecla C:** Si el jugador tiene bloqueado la posibilidad de colocar marcadores y presiona la tecla C pasa al modo de dibujar en la escena.
- **Tecla O:** Al pulsar el botón se intercambia el color de dibujado. Existen cinco colores: rojo, verde, amarillo, azul y negro. Para poder cambiar de color el jugador no debe estar dibujando en ese momento, pero debe encontrarse con el modo de dibujado seleccionado. El color por defecto es el rojo, cada vez que se cambie al modo de dibujado este color será el que se encuentre seleccionado.

- **Tecla B:** Elimina todo lo que se encuentre dibujado en el escenario
- **Tecla R:** Bloquear pantalla. Si se presiona la tecla R se bloquea la rotación del personaje. No interfiere en ninguna otra acción.
- **Tecla Y:** Sirve para hacer el rayo siempre visible. Si se activa el rayo se mantiene en pantalla siendo visible tanto para el director como para el camarista. En el caso en el que se vuelva a presionar se desactiva, haciendo que el rayo sólo sea visible cuando se crea un indicador, se señala una cámara, dibuja en el escenario o se mantiene pulsado el botón izquierdo del ratón.

Ratón

- **Botón izquierdo:** Botón utilizado para realizar todas las acciones del escenario. Es el botón empleado para crear los indicadores, señalar las cámaras y dibujar en el escenario. Si el puntero no está bloqueado se activa cuando se presiona y se mantiene visible mientras siga presionado.
- **Botón central:** Realiza la misma tarea que la tecla Y del teclado.
- **Rueda del ratón:** Modifica la distancia del rayo para colocar marcadores o dibujar a distinta profundidad.

Tableta

Hace las veces de ratón.

- **Tocar con el lápiz:** Actúa como hacer click con el botón izquierdo del ratón.
- **Pulsar el botón del lápiz:** Si se mantiene el botón con el lápiz apuntando a la tableta a una corta distancia tiene el mismo efecto que tocar la pantalla con el lápiz. Si se mantiene pulsado es similar a mantener pulsado el click izquierdo del ratón.

Referencias

- [1] *Oculus Rift* características, requisitos, precios Y COMPRA. Mundo Virtual. (n.d.). <http://mundo-virtual.com/gafas-realidad-virtual/oculus-rift/>.
- [2] *Wacom one: Análisis, OPINIONES, CARACTERÍSTICAS y Precio.* Tableta Gráfica: Comparativa y Guía de compra. (2020, April 8). <https://tabletagrafica.es/wacom/one/>.
- [3] *Unreal engine branding guidelines and trademark usage.* Unreal Engine. (n.d.). <https://www.unrealengine.com/en-US/branding>.
- [4] *Home.* Epic Games. (n.d.). <https://www.epicgames.com/site/es-ES/home>.
- [5] Santamaría, P. (2020, February 21). *El Gran Secreto de LOS Escenarios de The Mandalorian.* El Output. <https://eloutput.com/noticias/cultura-geek/the-mandalorian-vfx-unreal-engine/>.
- [6] Farris, J. (2020, February 20). *Forging new paths for filmmakers on the mandalorian.* Unreal Engine. <https://www.unrealengine.com/en-US/blog/forging-new-paths-for-filmmakers-on-the-mandalorian>.
- [7] *HBO's Westworld turns to Unreal engine for in-camera visual effects.* Unreal Engine. (n.d.). <https://www.unrealengine.com/en-US/spotlights/hbo-s-westworld-turns-to-unreal-engine-for-in-camera-visual-effects>.
- [8] *Inicio Dr. Platypus & MS. WOMBAT.* Inicio Dr. Platypus & Ms. Wombat. (n.d.). <https://www.drplatypusandmswombat.com/inicio>.
- [9] Richter, F. (2020, September 22). *Infographic: Gaming: The most lucrative entertainment industry by far.* Statista Infographics. <https://www.statista.com/chart/22392/global-revenue-of-selected-entertainment-industry-sectors/>.
- [10] Bizofan. (2021, August 31). *The future of the animation industry.* Business of Animation. <https://businessofanimation.com/the-future-of-the-animation-industry/>.
- [11] *Fully procedural.* SideFX. (n.d.). <https://www.sidefx.com/products/houdini/>.
- [12] UnrealDevelopmentKit. (2021, February 10). *Meet the METAHUMANS: Free Sample now available | Unreal Engine.* YouTube. <https://www.youtube.com/watch?v=6mAF5dWZXcI>.
- [13] 17/02/2021, T. W., & Watson, T. (2021, February 17). *How virtual reality movies are keeping the industry going during the crisis.* Skywell Software. <https://skywell.software/blog/virtual-reality-movies-the-future-of-filmmaking/>.

- [14] Wikimedia Foundation. (2021, June 6). *Storyboard*. Wikipedia.
<https://en.wikipedia.org/wiki/Storyboard>.
- [15] Juarez Ariño, M. (2019, March 25). *[Ue4][Bp]Security cameras guide*. Mauro Juarez Ariño.
<https://maurojuareza.wordpress.com/2019/03/23/ue4bpsecurity-cameras/>.
- [16] Krishnappa, M. (2018, December 23). *Using UE4 and C++ for VR DEVELOPMENT*. Game Development Articles and Tutorials.
<https://xrdeveloper.net/playlists/using-ue4-and-c-for-vr-development/>.
- [17] Batname. (n.d.). *Batname/UE4MultiWindow*. GitHub.
<https://github.com/Batname/UE4MultiWindow>.
- [18] Awforsythe, A. (2020). *awforsythe/Repsi: Basic example project Demonstrating replication in UE4*. GitHub.
<https://github.com/awforsythe/Repsi/>.
- [19] *Rpcs*. Unreal Engine Documentation. (n.d.).
<https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/Networking/Actors/RPCs/>.
- [20] de Lima, E. S. (n.d.). *Multiplayer and network communication in unreal engine*. Edirlei Soares de Lima.
https://edirlei.com/aulas/dp/DP_Lecture_05_Multiplayer_Unreal_Engine_2018.html.
- [21] *Aactor::getlifetimereplicatedprops*. Unreal Engine Documentation. (n.d.). <https://docs.unrealengine.com/4.27/en-US/API/Runtime/Engine/GameFramework/AActor/GetLifetimeReplicatedProps/>.
- [22] <https://developer.oculus.com/blog/tech-note-touch-button-mapping-best-practices/>. realovirtual.com. (n.d.).
<https://www.realovirtual.com/foro/topic/12812/que-tal-los-oculus-touche>.

Ilustraciones Implementación

Figuras

Figura 1 Portada del video de presentación de MetaHumans[12] 12Figura 2 Ejemplo StoryBoard: Storyboard_for_The_Radio_Adventures_of_Dr._Floyd[14] 14Figura 3 Logo Unreal Engine[3]18Figura 4 Logo Epic Games[4] 18Figura 5The Mandalorian usando Unreal Engine para los VFX[6] 19Figura 6 WestWorld[7] 20Figura 7 Icono de Unreal Engine 4.26.1 del Launcher de Epic Games 21Figura 8 Logo de Dr Platypus and Ms Wombat[8] 22Figura 9 Escenario Vista Frontal 22Figura 10 Escenario Vista Picada 23Figura 11 Cámara de Seguridad en Unreal Engine[15] 24Figura 12 Modos de Ejecución 26Figura 13 Opciones Multijugador 30Figura 14 Tipo de Red en Ejecución 32Figura 15 Diagrama de clases de Unreal Engine 34Figura 16 Personaje Camarista 36Figura 17 Ajustes de Proyecto: Mapa de Botones 38Figura 18 Ejemplo de cámara 40Figura 19 Jerarquía de las clases Camera 42Figura 20 Personaje Director 43Figura 21 Axis Mappings 44Figura 22 Diagrama de Estados Director 45Figura 23 Indicador con la componente de texto 49Figura 24 Declaración de función RPC 50Figura 25 RPC invocada desde Servidor 52Figura 26 RPC invocada desde Cliente 53Figura 27 Declaración de propiedad replicada **¡Error! Marcador no definido.**Figura 28 Apuntado del director con rotación bloqueada 56Figura 29 Generación de cámara 56Figura 30 Ventana1 activada 57Figura 31 Rotación de la cámara1 sobre su eje Z 58Figura 32 Ventana1 con rotación sobre Eje Z 58Figura 33 Cámara con rotación sobre Eje Y 58Figura 34 Ventana1 con rotación sobre Eje Y 59Figura 35 Cámara1 con rotación sobre su eje X 59Figura 36 Ventana1 con rotación sobre Eje X 59Figura 37 Creación de una segunda cámara 60Figura 38 Ejemplo el camarista elige ventana 60Figura 39 Asignando una cámara a una ventana no vacía 61Figura 40 Cámara eliminada 61Figura 41 Director con puntero activado 62Figura 42 Director con puntero desactivado 62Figura 43 Director con la rotación desbloqueada 63Figura 44 Modificación de la distancia del puntero del director 63Figura 45 Director coloca un indicador 64Figura 46 Vista del indicador por el camarista 64Figura 47 El director dibuja en la escena 64Figura 48 Visión de los dibujos por el camarista 65Figura 49 Director selecciona una cámara 65Figura 50 El camarista ve la cámara seleccionada66Figura 51 El Director crea un indicador con el nombre de una cámara 66Figura 52 Indicador con texto desde el punto de vista del camarista 67Figura 53 Controladores Oculus Rift[22]

