20 / 21

Escuela de Doctorado
y Estudios de Posgrado
Universidad de La Laguna

Desarrollo de “Ready Player Lab” con Unreal Engine 4

TRABAJO DE FIN DE MÁSTER:

“LABORATORIO DE QUÍMICA GAMIFICADO EN REALIDAD VIRTUAL”
“GAMIFIED CHEMISTRY LABORATORY IN VIRTUAL REALITY”

AUTOR:
MANUEL HERNÁNDEZ PADRÓN
TUTOR:
JORGE MARTÍN GUTIÉRREZ

D. **Jorge Martín Gutiérrez**, con N.I.F. 28949460G profesor Titular de Universidad adscrito al departamento Técnicas y Proyectos en Ingeniería y Arquitectura de la Universidad de La Laguna, como tutor:

CERTIFICA

que la presente memoria titulada:

"Laboratorio de química gamificado en realidad virtual."

ha sido realizada bajo su dirección por D. **Manuel Hernández Padrón**, con N.I.F. 54108688S.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 07/09/2021.

AGRADECIMIENTOS

Este proyecto no hubiera sido posible sin la ayuda ni guía de mi tutor, Jorge Martín, y de todo el profesorado del máster de Desarrollo de Videojuegos.

Especial mención a Jesús Torres por todas las horas dedicadas a la resolución de problemas.

A mis amigos, que siempre me han animado.

A Alexis Plasencia, por sus ilustraciones.

A Almudena del Barrio, por el cariño y apoyo incondicional.

RESUMEN

Este proyecto describe el desarrollo de un videojuego ambientado en el área de la química y las ciencias en general. Tiene como objetivo enseñar al jugador aspectos científicos teóricos y sobre todo prácticos a través del entretenimiento. Está enfocado para el uso de la realidad virtual, puesto que es el medio idóneo para desarrollar situaciones de aprendizaje mediante experiencias interactivas, donde se persigue la presencia del usuario a través de diferentes estrategias inmersión. Está desarrollado con el motor de videojuegos Unreal Engine 4 junto con el headset de Windows Mixed Reality de Acer.

Palabras clave: videojuegos, realidad virtual, química, laboratorio, unreal engine 4, WMR.

ABSTRACT

This project describes the development of a videogame about chemistry and sciences. The main goal is to teach the player scientific knowledge through entertainment. This project has been designed to be played with virtual reality because it is considered the best media to develop learning situations through interactive experiences. Creating a feeling of presence, using immersion as the mechanism by which we achieve it, is what this project is all about. It is powered by the Unreal Engine 4 and Acer's Windows Mixed Reality headset.

Keywords: videogames, virtual reality, chemistry, lab, unreal engine 4, WMR.


READY
PLAYER
LAB

GLOSARIO

2D – 2 Dimensiones

3D – 3 Dimensiones

AI – *Artificial Intelligent*

AnimBP – *Animation Blueprint*

b – *Boolean*

BPc – *Blueprint Child*

BPi – *Blueprint Interface*

BPP – *Blueprint Parent*

BS – *Blend Space*

CPU – *Central Processing Unit*

DLSS – *Deep Learning Super Sampling*

DoF – *Degree of Freedom*

E – *Enumeration*

EPI – Equipo de Protección Individual

fps – *Frames Per Second*

Gb – *GigaByte*

GM – *Game Mode*

GHz – *GigaHertz*

GS – *Game State*

HDR – *High Dynamic Range*

HMD – *Head Mounted Display*

LED – *Light Emitting Diode*

MSAA – *Multisample Anti-Aliasing*

NavVR – *Navigation Virtual Reality*

PC – *Player Controller*

POI – *Point of Interest*

PPE – *Personal Protection Equipment*

RAM – *Random Access Memory*

RGB – *Red Green Blue*

RT – *Render Target*

UE4 – *Unreal Engine 4*

UI – *User Interface*

UMG – *Unreal Motion Graphics*

USD – *United States Dollar*

UV – Ejes U y V de texturas 2D

VR – *Virtual Reality*

W – *Widget*

Wc – *Widget Child*

WMR – *Windows Mixed Reality*

XR – *Extended Reality*

ÍNDICE

Capítulo I. Introducción. 1

I.1 Introducción a la VR: presencia a través de la inmersión. **1**

I.2 VR, educación y entrenamiento. **1**

I.3 Antecedentes. **3**

I.4 *Hardware*. **4**

I.5 Consideraciones para el diseño de videojuegos en VR. **6**

I.6 *Motion Sickness*. **8**

I.7 Planificación del proyecto. **8**

I.8 Preparación de UE4 para el desarrollo en VR. **9**

I.9 Optimización para la VR. **10**

I.10 Control de versiones: *Perforce*. **13**

Capítulo II. Diseño y desarrollo. 14

II.1 Diseño inicial y creación del mapa de navegación 3D. 14

II.2 Sistema de locomoción por teletransporte. 59

II.3 Sistema de giro gradual o *snap-turning*. 65

II.4 Sistema de interacción con el mundo virtual. 67

II.4.1 Importación y animación de las manos virtuales. 67

II.4.2 Creación de objetos interactuables. 69

II.4.3 Configuración de las manos para la interacción con objetos. 71

II.4.4 Adición de *feedback* visual y háptico a la interacción. 74

II.5 Diseño y desarrollo del nivel tutorial. 76

II.5.1 Arte de la escena tutorial. 76

II.5.2 *Mainloop* del nivel tutorial. 78

II.5.3 Programación y animaciones del *bot* guía del tutorial: *LabBot*. 85

II.6 Diseño y desarrollo del nivel principal: *Lobby*. 87

II.6.1 Arte de la escena principal. 87

II.6.2 *Mainloop* de la escena principal. 89

II.6.3 *Chemical Dance Station*. 90

II.6.4 Estación fullereno C_{60}. 92

II.6.5 Estaciones de viaje y de salida del juego. 92

II.7 Diseño y desarrollo del laboratorio de química. 94

II.7.1 Arte de la escena de laboratorio. 94

II.7.2 Tarea 1: Equiparse con los *EPIs*. 95
II.7.3 Tarea 2: Encontrar elementos de seguridad. 98
II.7.4 Tarea 3: Seleccionar material de laboratorio. 99
II.7.5 Sistema de puntuación. 102
II.8 Otros aspectos de interés. 103
II.8.1 Audio. 103
II.8.2 *Lightmaps*. 104
II.8.3 *Nvidia DLSS.* 104
II.8.4 Desarrollo de una herramienta para borrar *assets* residuales. 105
Capítulo III. Resumen y Conclusiones. 107
Bibliografía. 109
Bibliografía complementaria. 112

Capítulo I. Introducción.

I.1 Introducción a la VR: presencia a través de la inmersión.

La realidad virtual, o VR, es un medio de inmersión hacia un mundo simulado, permitiendo al usuario ver, escuchar e interactuar con un entorno totalmente ficticio. En el mundo del desarrollo en VR, cuando el usuario está completamente dentro de esta experiencia, se llega a lo que se conoce como inmersión.

Asimismo, igual de importante es la presencia, término que hace referencia a la sensación del usuario de estar físicamente presente en el mundo virtual, respondiendo a los estímulos del entorno. En definitiva, el desarrollo en VR trata sobre esto.

En principio puede parecer que los términos presencia e inmersión son similares, pero lo cierto es que no. Se puede pensar en la presencia como el objetivo del desarrollo en VR, es decir, las sensaciones que se intentan despertar en el usuario; mientras que la inmersión es más bien el mecanismo para lograrlo.

La presencia no es algo fácil de conseguir, ya que existen numerosos factores en contra. Los usuarios pueden sentir el HMD en su cara, pueden golpearse con el mundo real por accidente y pueden escuchar sonidos del exterior. Además, las resoluciones de imágenes no suelen ser lo suficiente altas (efecto "*screen door*"), se hace difícil leer textos pequeños, etc. Sin embargo, y a pesar de todas estas limitaciones, la VR puede crear la sensación de presencia; a veces por el simple hecho de que se pueden realizar experiencias que no ocurren en otros medios. Se pueden diseñar maneras de explorar y aprender que no son posibles o son muy costosas mediante otros medios. Es el camino para conectar con gente y lugares que de otra forma sería imposible o muy difícil.

Hay que resaltar también que la realidad virtual es un medio que todavía se encuentra en su infancia. Actualmente se encuentran las primeras generaciones de consumidores de VR. [1]–[3]

I.2 VR, educación y entrenamiento.

La realidad virtual comenzó sus primeros pasos en la educación en 1929, cuando *Edwin Link* creó *Link Trainning* para entrenar a los pilotos de aeronaves usando un simulador. La combinación de la inmersión y la interacción hace de la VR una poderosa herramienta para la educación, el aprendizaje y la exploración. La VR es capaz de otorgar un entendimiento mucho

más experimental y concreto que otros medios habituales. Mientras dichos medios comunican ideas, la VR comunica directamente experiencias.

La educación tradicional está centrada en comunicar hechos a los estudiantes. Pero los hechos, por sí solos, no tienen el suficiente contexto. La VR les permite, en cambio, trabajar directamente con materiales y representaciones de ideas. Mientras exploran aprenden, desarrollando habilidades y convirtiendo ideas abstractas en experiencias. Se hace necesario resaltar que la educación en VR y la tradicional no son algo que se coloque en diferentes peldaños, sino que deberían ser consideradas como piezas de un mismo puzle. Al fin y al cabo, teoría y práctica siempre han ido de la mano para logar un completo aprendizaje.

El uso de la VR en educación debería ser fácil de usar, inmersivo, con gancho y lleno de significado, promoviendo en el usuario diferentes maneras de aprender mediante la interacción, la exploración y el descubrimiento.

Ahora bien, para que un videojuego tenga utilidad educativa debería, primero, despertar la curiosidad en el usuario; segundo, permitirle experimentar sin miedo a equivocarse o ser juzgado; y tercero, generarle placer activando los sistemas de recompensa del cerebro. Esto haría de un videojuego un completo mecanismo de aprendizaje.

Pero la realidad es que apenas se han visto triunfar a videojuegos educativos, es decir, videojuegos que se diseñaron con tal propósito. Un posible motivo, es que, hasta ahora, todos los videojuegos educativos tienen un cometido similar, educar en materias teóricas, pero de una manera más didáctica. Sin embargo, no son exitosos. Fracasan a la hora de hacer un videojuego entretenido. Esto, por consecuencia, hace que el videojuego también fracase en la tarea de educar. El problema es que son un complemento a la educación tradicional. Fracasan, bien por un mal diseño, bien por una temática poco atractiva.

¿Qué hacer entonces para hacer que funcione un videojuego educativo? Lo más importante, por encima de todo, es entender al jugador. En lugar de que el propósito sea educar y luego entretener, se debería invertir la fórmula, entretener y luego educar. Esto es algo que muchos videojuegos ya hacen casi que sin querer. Por lo tanto, existen videojuegos que educan y que son exitosos, pero no son educativos, ya que su propósito no es ese. La conclusión, por tanto, es que no deberían existir los videojuegos educativos, es decir, videojuegos cuya principal misión es la de educar. El principal objetivo de un videojuego debería ser el de entretener.

También hay que resaltar que el objetivo de este proyecto no es el de ser un simulador de laboratorio químico, sin embargo, sí persigue el aprendizaje del usuario. Si entre los objetivos

de un videojuego está educar, para que no fracase, debería utilizar el entretenimiento como herramienta para lograr dicho objetivo. En el caso concreto de la VR, además, persiguiendo la presencia a través de la inmersión [1].

I.3 Antecedentes.

A lo largo de toda la revisión bibliográfica se han encontrado numerosos e interesantes proyectos que guardan similitud con este. Todos tienen en común el uso de tecnologías como la de la realidad virtual para aplicaciones de carácter científico-técnico. En este apartado se comentan algunos ejemplos de interés.

Uno de los casos más exitosos es *HoloLAB Champions*, un videojuego de realidad virtual dirigido a estudiantes que quieran interactuar y competir de forma segura sobre química. Está desarrollado por *Institute of Education Sciences* en EEUU [4].

El segundo de ellos es una aplicación de realidad virtual hecha en *Unity* y dirigida a las *Oculus Quest* que intenta demostrar la importancia del uso de los guantes de laboratorio como introducción a la química práctica. Es un proyecto de colaboración entre diferentes universidades de EEUU –*University of Southern California, Boise State University,* y *California State University*– donde colaboran departamentos como el de química, ciencias de la computación e ingeniería [5].

Imágenes del proyecto en Unity sobre la importancia del uso de guantes de laboratorio.

Otro proyecto de interés es una aplicación de realidad virtual hecha con *UnityMol* y *ChimeraX* para *HTC Vive* que intenta mejorar el entendimiento de la química de la coordinación y los orbitales moleculares. Es un proyecto de colaboración entre *University of California* y *Pacific Northwest National Laboratory*, donde colaboran los departamentos de química, bioquímica y ciencias físicas y computacionales [6].

Orbital Battleship es otro proyecto de videojuego multijugador en realidad virtual donde se compite por acertar en cuestiones científicas. Está desarrollado en *Unreal Engine 4* para *Oculus*. Fue desarrollado en la universidad *ITMO* de San Petersburgo [7].

El siguiente proyecto de interés se llama *NC State VR Organic Chemistry Labs* y trata sobre el uso de la realidad virtual en vídeos sobre prácticas en laboratorios de química lanzados como apoyo al aprendizaje práctico de la materia [8].

Otro proyecto de interés es una aplicación de realidad virtual cuyo objetivo es el de generar una experiencia inmersiva en el aprendizaje de la espectroscopía de infrarrojos. Fue desarrollado en *North Carolina State University* [9].

I.4 *Hardware*.

La manera de llegar a la inmersión antes mencionada es a través de un casco o gafas de realidad virtual, también llamado HMD o *headset*. Este artilugio es el que muestra el mundo virtual y monitorea el movimiento de la cabeza del usuario, creando la ilusión de movimiento en el espacio. Algunos HMDs incluyen auriculares para favorecer aún más la inmersión a través de lo que se conoce como *Spatialized Audio*.

Existen diferentes tipos de HMDs. Se usa el término grados de libertad *o* DoF para marcar una diferencia. De este modo existen HMDs con 3 DoF y con 6 DoF. En los del primer tipo solo se puede monitorear la rotación del usuario, el grado en el que se inclina hacia un lado –balanceo o *roll*–, el grado en el que se inclina hacia adelante –cabeceo o *pitch*–, o el grado en el que se gira hacia los lados – *yaw*–. Ejemplos de estos HMDs son *OculusGo* y *Samsung Gear*.

HMD Oculus Go *HMD Samsung Gear*

Por otro lado, existen los dispositivos de 6 DoF. Los tres DoF extra se deben a que también monitorean la posición del usuario, arriba y abajo –*up and down*–, de lado a lado –*side to side*–, y hacia adelante y hacia atrás –*forward and backward*–. La mayoría de los primeros 6 DoF *headsets* necesitaban *hardware* extra para esto, como cámaras externas o estaciones base. Actualmente las cámaras se incorporan en el propio HMD –lo que se conoce como *inside-out tracking*–. Ejemplos de estos HMDs son: *HTC Vive Cosmos*, *Oculus Rift S, HTC Vive Focus, Oculus Quest* y *Windows Mixed Reality.*

HMD HTC VIVE Cosmos

HMD HTC VIVE Focus

HMD Oculus Rift S

La mayoría de los HMDs también traen consigo un par de mandos o controladores –también llamados *controllers* o *imput devices*– para interactuar con el mundo.

HMD HTC Oculus Quest

Para el desarrollo de este proyecto se hará uso del *set* de HMD y *controllores* de *Windows Mixed Reality* (WMR, de marca *Acer*).

HMD WMR de ACER

En la realidad virtual usualmente se utiliza el término háptico para referirse a las sensaciones físicas del *hardware* de VR, como la vibración de los mandos al acercarse a un objeto interactuable. Esto fomenta la inmersión junto con el *feedback* audiovisual [1][2][3].

El resto del equipo de desarrollo está compuesto por: Portátil *Gaming OMEN by HP* –procesador *Intel(R) Core(TM)* i7-9750H *CPU* 2.60GHz y 2.59 *GHz*, 16 *Gb* de *RAM*, tarjeta gráfica *NVIDIA GeForce RTX 2060*– y un micrófono *Blue Yeti* [10].

I.5 Consideraciones para el diseño de videojuegos en VR.

El comportamiento de los objetos visuales del proyecto es algo de lo que preocuparse. Las sensaciones necesitan emparejarse a las expectativas del usuario. Usar por ejemplo el *spatialized audio* es muy importante, ya que hará que los objetos tengan un sonido localizado.

Por otro lado, la latencia hace referencia a la velocidad con la que una aplicación de VR responde visualmente a las acciones del usuario, fundamental para la inmersión. Actualmente una óptima latencia estaría alrededor de 20 milisegundos. En el apartado de optimización se verán qué aspectos técnicos se deben ajustar para ello. Algunos ejemplos son el uso del plugin DLSS o la acción de activar/desactivar ciertas opciones de UE4.

En VR, ante la duda de elegir entre calidad de imagen y fps o *framerate*, siempre hay que decantarse por lo segundo. Si la experiencia no se desarrolla de forma cómoda y suave debido a una mala latencia, por impresionantes que sean los detalles de las texturas, se podría romper la inmersión, o aún peor, llegar al *Motion Sickness*. Este es un concepto que hace referencia al malestar físico del usuario.

Es importante, también, asegurarse de que las interacciones con el mundo tienen sentido. Por ejemplo, si se coloca en la escena un objeto que tiene aspecto de ser interactuable y el usuario no puede cogerlo, se perderá inmersión. Es mejor colocar objetos no interactuables fuera del área de juego o bien cambiar su apariencia.

La manera en la que se representan las manos también es de importancia. Si solo se representan las manos, la lógica natural del usuario sería la de coger objetos y moverlos. Si en lugar de manos se representan los mandos, herramientas, armas u otros, se sugiere al usuario interactuar de forma diferente con el mundo. Los usuarios tratarán de hacer aquellas cosas que parece que pueden hacer.

Hay que resaltar que cuánto más inmersiva se vuelve una experiencia más fácil es de romper dicha inmersión. La inmersión no requiere que el mundo virtual sea un espejo del mundo real a la perfección, sino más bien que la experiencia sea consistente consigo misma.

Otro aspecto para considerar es la conciencia que el usuario tiene sobre su propio cuerpo. Es peligroso para la inmersión añadir elementos que hagan al usuario perder la concepción natural que tiene de sentir las partes de su cuerpo a pesar de no verlas. Esto se conoce como propiocepción. Si se representan partes del cuerpo del usuario que no encajan con este sentido, se perderá la inmersión. Por eso es desaconsejable representar más allá de las manos del usuario en realidad virtual. Representar los brazos puede ser un error, ya que no hay información acerca de que están haciendo en el mundo real para llevarlo al mundo virtual. Pasa lo mismo con las manos muy realistas, pueden hacer sentir incomodidad al usuario si no se emparejan a la perfección con sus manos reales. Por todo ello es mejor representar solo las manos estilizadas –en *cartoon* o robóticas–.

Es importante no tomar nunca el control de la cabeza o de la visión del usuario. Hacer esto podría despertar en el usuario *motion sickness* además de romper la inmersión por completo. Si el usuario mueve la cabeza, la cámara debe moverse también, incluso si el juego está pausado o cargando. Asimismo, igual de importante es no acelerar la cámara. Para implementar el movimiento en VR se suele optar por el teletransporte o el movimiento continuo suave, nunca acelerado. Tampoco se debe manipular la profundidad de campo o usar el *motion blur*.

También es aconsejable regular la intensidad de las luces y los colores. Las luces fuertes, los destellos intensos o parpadeos son un camino directo a la *motion sickness*, al igual que cuando no se cuida la escala. Algunos usuarios son propensos al mareo si la escala del mundo no la sienten correcta. Hay que asegurarse de que los objetos en el mundo son escalados acorde a las expectativas que sugiere la experiencia. En UE4, por defecto, una unidad de *unreal* o UU equivale a un centímetro.

Como desarrollador, también es importante tener en cuenta las acciones físicas que se le piden al usuario, como caminar, extender los brazos o agacharse. Ya que podrían tener accidentes con el mundo real. Es importante tener marcada el área de juego si se usan acciones de movimiento real en la escena. Además, la mayoría de HMDs usan cables, por lo que el usuario podría quedar atrapado entre los mismos si se gira demasiadas veces.

Por último, hay que destacar el hecho de ser consciente de cómo de inmersiva se quiere que sea una experiencia VR y hacer el diseño acorde a tal propósito. No todas las aplicaciones VR necesitan la misma inmersión. Por ejemplo, si se está diseñando una simple herramienta de ingeniería como el desmonte por piezas de un motor, es más importante centrarse en la

habilidad de manipular el modelo fácilmente y no tanto en hacer creer que el usuario se encuentra en otro lugar [1]–[3].

I.6 *Motion Sickness*.

Los seres humanos emplean mucho tiempo en aprender a caminar. Es, en realidad, una tarea de complicada coordinación. Con el tiempo se logra sin ni siquiera concentrarse en ello. Esto es gracias a una estructura en el oído interno llamada sistema vestibular, usada para coordinar movimientos y mantener el equilibrio. Este sistema es extremadamente sensible y trabaja en conjunto con la vista y la propiocepción para entender cómo es el movimiento.

Esto crea un problema cuando la información visual le dice al cuerpo que se está moviendo, pero ese movimiento no puede sentirse en el sistema vestibular. Esto provoca lo que se llama *motion sickness* o cinetosis. Es una sensación de mareo y náuseas similar a cuando se viaja en un barco, por ejemplo. Un buen videojuego de VR se basa en simular el movimiento del usuario evitando la *motion sickness*. La causa más común de *motion sickness* es un pobre diseño de la locomoción. La segunda causa más común es la mala latencia.

Además de la *motion sickness* también hay que resaltar la fatiga visual. Los ojos utilizan músculos para orientarse y focalizar objetos, los cuales también pueden quedar fatigados. Los síntomas pueden incluir dolor de cabeza, fatiga o visión borrosa. Esta fatiga puede ser causada principalmente por dos factores. Uno de ellos es el *flickering* o parpadeo causado por una mala latencia. El otro es el conflicto entre la distancia fija a la que los ojos deben enfocar para ver la pantalla del HMD y las distancias cambiantes a las que se necesitan adaptar para focalizar los objetos del juego en profundidad. Esto se denomina conflicto de acomodación de vergencia. Puede gestionarse manteniendo los objetos importantes en el mundo virtual, los que requieren enfoque, a aproximadamente 1 m de distancia como mínimo. De esta forma la adaptación del ojo y la vergencia son similares, evitando así la fatiga [1]–[3].

I.7 Planificación del proyecto.

Antes de comenzar con el desarrollo es importante tener un objetivo claro para el proyecto. ¿Qué se intenta hacer? ¿Para qué? Esclarecerlo ayudará a que el resultado final no sea algo difuso. Cuánto más se avance en el desarrollo del proyecto más costoso y difícil se vuelven los cambios. Otra pregunta de vital importancia sería, ¿encaja el proyecto para la realidad virtual? En este caso la respuesta es claramente sí. El proyecto funciona mejor en VR que en un medio tradicional. En este caso se está tratando de diseñar un videojuego en realidad

virtual cuyo objetivo principal es el aprendizaje del usuario, pero no a cualquier precio, solo a través del entretenimiento. Este aprendizaje está relacionado con competencias útiles para la vida real, relacionadas con el conocimiento teórico y sobre todo práctico de la disciplina química. Es por ello que la VR es el medio perfecto para ello.

Una buena planificación tiene en cuenta los costes. Aunque este proyecto está orientado a ser evaluado técnicamente para el máster de Desarrollo de Videojuegos, en un futuro podría evolucionar a un proyecto más profesional. Hay que tener en cuenta que UE4 es gratis para descargar y usar y, además, para un uso comercial los términos y condiciones son razonables. Al descargar el motor hay que aceptar unos acuerdos de licencia. Si como desarrollador el juego o aplicación genera más de 1 millón de USD, es obligatorio pagar en calidad de royalties un 5% de las ganancias en bruto [11].

I.8 Preparación de UE4 para el desarrollo en VR.

Para este proyecto se utilizará el motor de videojuegos *Unreal Engine* 4, concretamente la versión 4.26.1 (posteriormente actualizado a 4.26.2). Hay que instalar el *Epic Games Launcher* (con cuenta de usuario) y posteriormente la última versión del motor. También hay que instalar el software del HMD, en este caso, el de *Windows Mixed Reality*. Los mandos se conectan por bluetooth al ordenador. Una vez instalado se puede navegar en portal VR de WMR donde se pueden acceder a diferentes aplicaciones y juegos. A través de este portal se configura el nivel respecto al suelo y el área de juego. También existe la opción de juego en estático.

Pantalla setup de WMR: trazar área de juego o jugar en el sitio.

Como posteriormente se utilizará el plugin en UE4 de *OpenXR* para el flujo de datos entre el motor y la aplicación de WMR, hay que instalar la aplicación específica de *OpenXR* para

WMR. No se usarán otras aplicaciones para este fin, como por ejemplo *SteamVR*. También se usará el plugin de *Nvidia* DLSS.

UE4 ofrece un proyecto base para facilitar los primeros pasos en el desarrollo de un proyecto de realidad virtual que ofrece aspectos básicos como diferentes tipos de locomoción o interacción con objetos. Sin embargo, en este caso se comenzará desde cero creando un proyecto en blanco llamado "*ReadyPlayerLab_v0*".

A la hora de crear el proyecto hay que seleccionar el *hardware target*. El proyecto se desarrollará para aplicación de escritorio, *Windows 64 Bit*. En cuanto a los *graphics target* se seleccionó *maximum quality*. Así pues, será un proyecto en blanco basado en *blueprints* sin contenido inicial.

Además de las herramientas que ofrece UE4 para el desarrollo del proyecto, también se han utilizado otras relacionadas con el modelado 3D. Principalmente *Blender*, aunque también en menor medida *Autodesk Maya* y *Autodesk 3ds Max*. Las principales tareas realizadas fueron las correspondientes al arreglo de los mapas UV y el *smoothing group* de diferentes mallas de objetos 3D importados al proyecto, así como la incorporación de texturas o la exportación final en formato *fbx* [12]–[14]. En cuanto a la edición de imagen se ha utilizado principalmente *Adobe Photoshop* y *Canva*, donde se ha creado y diseñado elementos de la interfaz de usuario y otros elementos gráficos del juego [15], [16]. Para la grabación y edición de audio se ha utilizado *Audacity*, *Voicemod* y *Voicemeeter*. Con *Voicemeeter* se pudo mezclar audio y combinar canales, uno para conectar el micrófono y otro para un programa de audio, combinando una entrada *hardware* y una entrada virtual. *Voicemod* por su parte es el programa de audio modulador de voz que permitió añadir efectos a diferentes grabaciones. Por último, *Audacity* permitió no solo grabar, sino mejorar calidad de los audios, así como exportar en formato *wav* [17]–[19]. Finalmente, para la grabación y edición de vídeo se utilizó *OBS* y la aplicación de *Fotos* de *Windows 10*. El primero se utilizó para grabar la pantalla y el segundo para hacer ediciones sencillas y muy básicas de los vídeos. También permitieron la exportación de archivos *mp4* [20], [21].

I.9 Optimización para la VR.

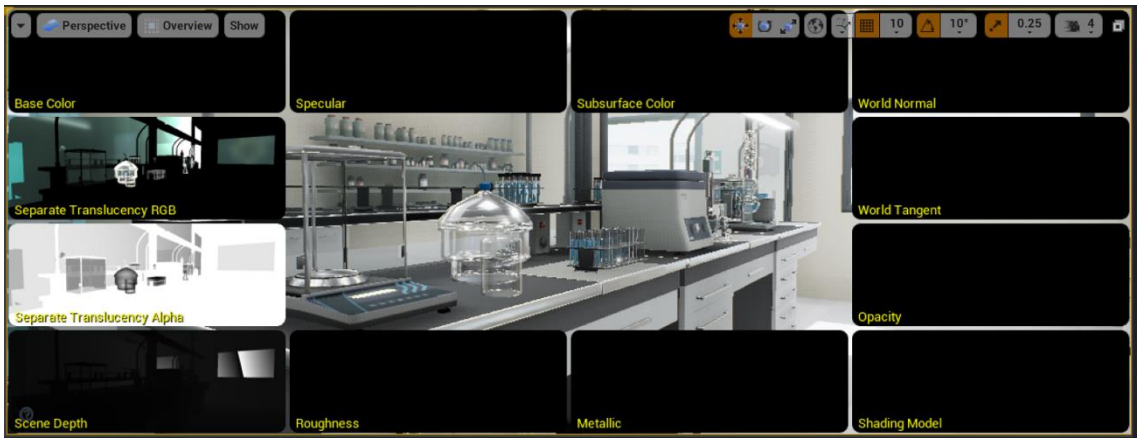
Es necesario modificar la configuración del motor para obtener el mejor rendimiento posible en el proyecto de VR [22], [23]. En *project settings* hay que modificar ciertos aspectos.

El primero de ellos es activar *Instanced Stereo*. Antes de la versión 4.11 de UE4, el motor simplemente ejecutaba todo el proceso de renderizado dos veces, una para cada ojo. Esto no es

muy eficiente, ya que la única diferencia real entre las dos vistas era un pequeño cambio en la ubicación del ojo que lo miraba. El *Instanced Stereo* optimiza este proceso de renderizado al permitir que la escena se renderice de una sola pasada. La vista renderizada se entrega al *hardware* de video junto con la información que necesita para ajustar la vista para cada ojo.

Lo segundo es activar *Round Robin Occlusion*. El motor de *Unreal* elige qué objetos dibujar a través de un proceso llamado *Culling*. Para ello utiliza diferentes métodos en orden de complejidad: *Distance Culling*, *View Frustum Culling*, *Precomputed Visibility* y *Dynamic Occlusion*. Este último se encarga de comprobar en tiempo real si un objeto en la escena está siendo bloqueado por otro actor. Es una tarea costosa, por lo que solo se hace para objetos que no han pasado por los anteriores procesos de *culling* menos costosos. Concretamente para los proyectos de VR, *Unreal* ofrece un método optimizado de *Dynamic Occlusion* llamado *Round Robin Occlusion*. Este último se encarga de realizar la oclusión para un ojo por *frame*, alternándolos, en lugar del cálculo completo que realiza el método por defecto. Por tanto, no se trata de una mejora del algoritmo de oclusión, sino de las llamadas que se realizan a dicho proceso. Esto ahorra una considerable cantidad de tiempo, especialmente en escenas con muchos objetos. Funciona bien porque las vistas de renderizado para cada ojo son casi idénticas.

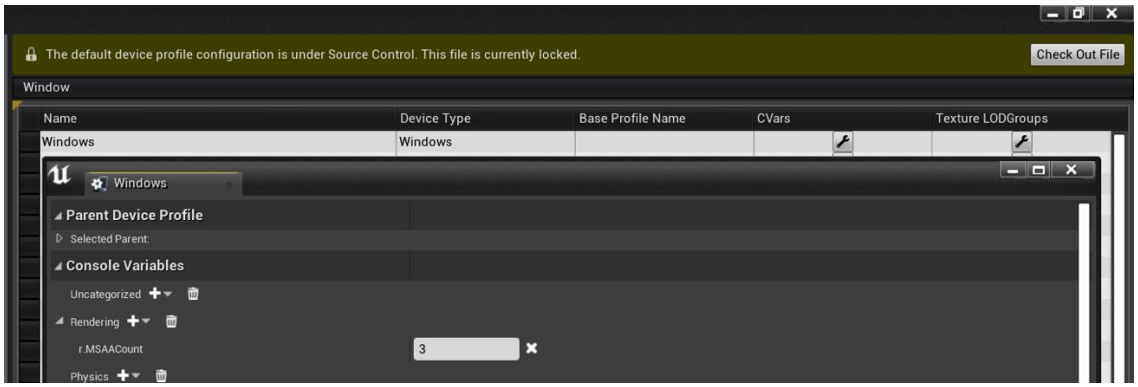
Lo tercero es elegir el método de renderizado *Forward Shading*. Entendiendo el proceso de *shading* como el de aplicar luz a una geometría, aparecen dos opciones: *forward shading* y *deferred shading*. La primera opción es la más apropiada para este caso. Mediante esta opción, cada objeto geométrico en la escena es sombreado a la par que renderizado, y se comprueba cada luz de la escena para ver cómo debería afectarle. Por tanto, si se tienen muchos objetos y luces en la escena, esto se traduce en muchas operaciones. Es por esto por lo que la mayor parte de las luces de los objetos tienden a ser precalculadas – *baked*– en *static lightmaps*. Por otra parte, el *deferred shading* utiliza una serie de imágenes que contienen información sobre los materiales, la profundidad de cada píxel y otros factores que afectan a la iluminación de la escena. Por tanto, el sombreado solo se ejecuta una vez después de que se haya ensamblado toda esta información. A esta colección de *buffers* se le llama *Geometric Buffer –G-Buffer* – y a su proceso de construcción *base pass*.



Contenido del G-Buffer de la escena de laboratorio.

Esto es más eficiente y rápido para escenas con muchas luces dinámicas. También permite el uso de efectos visuales como el *ambient occclusion*. El problema está en el manejo de esta información. Es difícil desactivar aspectos individuales del proceso de renderizado. Sin embargo, algunos de estos procesos son costosos de ejecutar eficientemente en VR. Son calculados para pantalla plana y no suelen encajar con la visión de VR. Es por esto por lo que, a partir de la versión 4.14, *Unreal* incorpora un tipo de *Forward Shading* especial para proyectos de VR. Sin embargo, también añadió una mejora para el otro sistema.

Lo siguiente es elegir un método *anti-aliasing* MSAA personalizado. Cuando el proceso de renderizado dibuja la escena, se está mostrando una rejilla de píxeles –*picture elemets*– para colorear. Esto se convierte en un problema cuando un objeto en la escena 3D solo llena parcialmente un píxel en el espacio 2D en el que se va a dibujar. El renderizado tiene que decidir si el píxel debe ser llenado con el color del objeto o el color del fondo. En la práctica esto significa la aparición de bordes irregulares, lo que se conoce como *aliasing*. La manera de resolverlo es a través de un proceso llamado *anti-aliasing*. Existen diferentes métodos y técnicas, aunque todos tienen el efecto de suavizar los bordes. Esto es especialmente importante en VR ya que las resoluciones de los HMDs siguen siendo limitadas. El método *Multisampling Anti-Aliasing* solo está disponible para *forward shading.* Hay que ir a la herramienta *Device Profiles Window* dentro de *Window* / *Developer Tools* y luego seleccionar *CVars* en la fila *Windows*. Después hay que abrir *Console Variables* / *Rendering*. Desde aquí se pueden ver todas las variables de consola relacionadas con el renderizado que se están especificando actualmente. Al hacer clic en *+* se puede añadir un valor a *r.MSAACount*. Por defecto suele estar en 4, pero reducirlo a 3 reducirá ligeramente la calidad del *anti-aliasing* pero también acelerará el proceso.



Ajuste del MSAA.

Hay que guardar los cambios. Estos serán escritos en un nuevo archivo de configuración en el directorio *Configs* del proyecto llamado *DefaultDeviceProfiles.ini*.

Por último, hay que activar la casilla de *Start in VR*, ya que el proyecto solo funcionará en VR. También hay que desactivar la opción *Ambient Occlusion Static Fraction*. Este es un efecto de sombreado costoso y que se calcula para el espacio de pantalla, por lo que puede no lucir bien en VR [1]–[3]. A modo de resumen ver la siguiente tabla:

Project Settings	*Engine*	*Rendering*	*VR*	*Instanced Stereo*	*TRUE*
				Round Robin Occlusion	*TRUE*
			Forward Renderer	*Forward Shading*	*TRUE*
			Default Settings	*Anti-Aliasing Method*	*MSAA*
				Ambient Occlusion Static Fraction	*FALSE*
	Project	*Description*	*Settings*	*Start In VR*	*TRUE*

Resumen de ajustes optimizados para el proyecto en VR.

I.10 Control de versiones: *Perforce*.

El proyecto está disponible en el *Perforce* del máster dentro de *//D3D2021/Unreal Projects/alu0100617900/ReadyPlayerLab*. Como el proyecto está desarrollado enteramente en *blueprints* solo contiene las carpetas *Config*, *Content* y el archivo *uproject*. También se añadió la carpeta de *Build* con el ejecutable [24].

Capítulo II. Diseño y desarrollo.

El proyecto se divide en tres niveles. El primero es el nivel correspondiente al tutorial del juego. El segundo es el *lobby* o salón principal desde el cual se accede al resto de niveles. Este nivel es más que un mero tránsito del jugador, ya que también incorpora elementos de interacción. El tercero es el nivel correspondiente al laboratorio de química. Se comenzó desarrollando el *lobby* por ser el eje principal de juego. Dentro de este nivel/escena el usuario puede moverse mediante el teletransporte o también utilizando un mapa virtual de navegación 3D [25].

II.1 Diseño inicial y creación del mapa de navegación 3D.

En este apartado se describe el desarrollo de las bases del proyecto y se describe al detalle el desarrollo de un sistema de navegación basado en un mapa de previsualización y teletransportación. Para comenzar hay por migrar los *assets* del "*Studio Showcase*" del *Marketplace* de *Unreal*. Este es el escenario elegido para el *lobby.* El destino será un proyecto vacío que contendrá el proyecto final. Seguidamente se desarrollará la lógica del jugador. Después se construirá el mapa interactivo de navegación 3D y, cuando esté listo, se agregarán las manos de realidad virtual. Para todo se harán uso tanto de los *Blueprints* como de los UMG.

Este sistema de navegación permite al usuario teletransportarse a ubicaciones preestablecidas alrededor del escenario utilizando un *widget* 3D UMG – *Unreal Motion Graphics UI Designer*– controlado por los controladores de movimiento en VR. Se construirá esta mecánica en dos formas, el "*TopMode*" y el "*PreviewMode*". La primera consistirá en una vista desde lo alto de la escena con marcadores y lugares a los que se quiere teletransportar a un usuario. Este simplemente tendrá que apuntar y hacer clic en el marcador al que desea viajar. Mediante la segunda forma, se apagará el modo de vista previa y se animará la cámara al punto de vista de la nueva ubicación. El usuario podrá recorrer el mapa a través de los puntos y presionar el botón de viaje a la ubicación para teletransportarse.

Después de que el usuario inicie el juego comenzará el modo de realidad virtual y el *pawn* VR se generará. El jugador VR podrá generar el mapa 3D frente a él presionando y sosteniendo ambos gatillos. Mientras se sostiene el gatillo se controla la ubicación del mapa en el espacio de la escena. Cuanto más cerca estén las manos entre sí, más lejos estará el mapa del usuario. Cuando el usuario suelte el gatillo, el mapa permanecerá en su lugar para que el usuario

pueda señalar la ubicación o el botón al que le gustaría viajar o interactuar presionando el gatillo para ello.

Se comienza creando un proyecto vacío llamado "*ReadyPlayerLab*". Luego se migran los *assets* necesarios sobre los que construir el proyecto. Se eligió el proyecto "*Virtual Studio"* porque tiene un diseño interior y arquitectónico agradable y profesional, perfecto para la realidad virtual.

Dentro de "*Content"* se crea una nueva carpeta para almacenar todos los *assets* del mapa de navegación 3D llamada "*NavVR*". Dentro se crean otras carpetas: "*GamePlay*", "*Player*" e "*Interface*". "*GamePlay*" contendrá una nueva *blueprint class* de tipo *GameModeBase* llamada "*GM_StudioShowcase*". Se crea otra de tipo *GameState* llamada "*GS_StudioShowcase*". También se crea un *PlayerController* llamado "*PC_StudioShowcase*". En la carpeta "*Player*" se colocarán los *pawns* necesarios. "*BPP_Pawn*" es la *blueprint class* padre de tipo *Pawn*. A partir de esta se crea una *blueprint class* hija llamada "*BPc_VRPawn*".

En las clases jerárquicas en *c++* o en *blueprint* cualquier cosa agregada a *BPP_Pawn* será heredada por la clase hija. Si hay una variable, función o evento en el *blueprint* padre, los *blueprints* hijos también lo tendrán. La ventaja es que se pueden manejar desde el padre. Por último, en la carpeta "*Interface*" se crea una *blueprint interface* llamada "*BPi_StudioShowcase*".

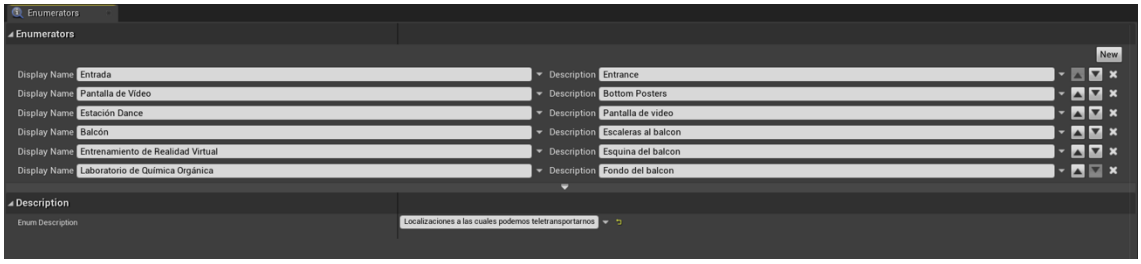
Dentro de *NavVR / GamePlay*, en "*GM_StudioShowcase*" se establece la *Game State Class* a "*GS_StudioShowcase*", la *Player Controller Class* a "*PC_StudioShowcase*", y la *Default Pawn Class* a "*BPc_VRPawn*".

Hay que recordar que existen numerosas formas mediante las cuales los *blueprints* pueden comunicarse, tanto directas como indirectas. Una de estas formas es a través de las *Blueprint Interfaces*. Con ellas, cada *blueprint* del proyecto puede suscribirse – implementar la interfaz– a los eventos de las mismas y recibir notificaciones y/o datos. Además, cada *blueprint* puede llamar a los eventos para comunicarse con otros *blueprints* que estén escuchando. Esto

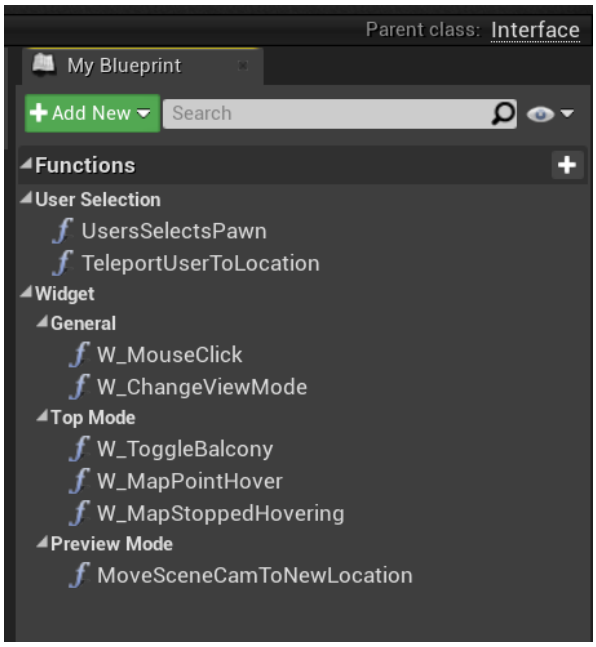
tiene la ventaja de que ningún objeto necesita ser consciente del otro, por lo que se desacopla su lógica entre sí y se facilita el código.

Se continuó creando un *asset* del tipo *Enumeration*. Esta es simplemente una lista ordenada para ayudar a realizar todo tipo de tareas como: organizar etapas de un nivel, dar una lista de categorías a un botón de la interfaz de usuario, usar el string del texto como etiqueta, etc. En este caso se utiliza para crear una lista de ubicaciones a las que el usuario podrá teletransportarse llamada "*EMapPoint*". Dentro van las seis localizaciones posibles:

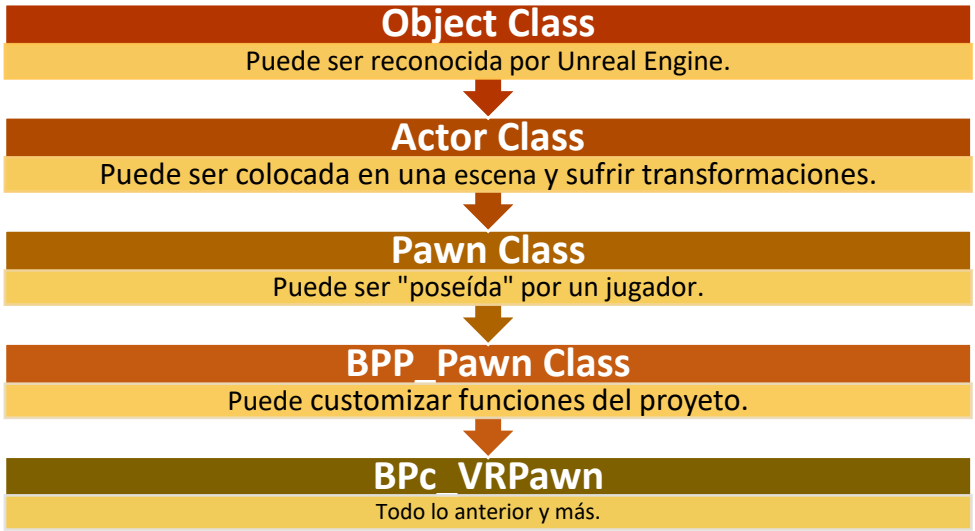
- *Entrada*
- *Pantalla de Vídeo*
- *Estación Dance*
- *Laboratorio de química*
- *Balcón*
- *Entrenamiento para la realidad virtual (tutorial)*



Se usarán las *blueprint interfaces* para llamar directamente a las funciones en los *blueprints* en las que están implementadas o en las que están escuchando. Por ejemplo, si hay quince *blueprints* donde todos están implementando la misma interfaz y se usa el nodo *Get All Actors* y se llama a la función usando un *For Each Loop*, todos ellos ejecutarán esa función. Si esa no es la intención, es importante filtrar las llamadas de alguna manera; tanto antes de realizar la llamada como inmediatamente después de que el *blueprint* reciba la llamada. De esta forma se evita el uso innecesario de memoria. Cuando se produce un solapamiento ente el *collider* de la mano y otro actor, si dicho actor implementa la interfaz en específico, se llama a una función para emparejarla a la correspondiente mano *VR*. De esta forma, primero se usa indirectamente la interfaz, y solo después se llama a la función que contiene. En *NavVR / Interface / BPi_StudioShowcase* se crean las funciones que se usarán para la navegación en la escena, clasificadas según su categoría. Usar las *blueprints interfaces* en el proyecto facilita el desarrollo y beneficia a la reusabilidad y la lectura del código.



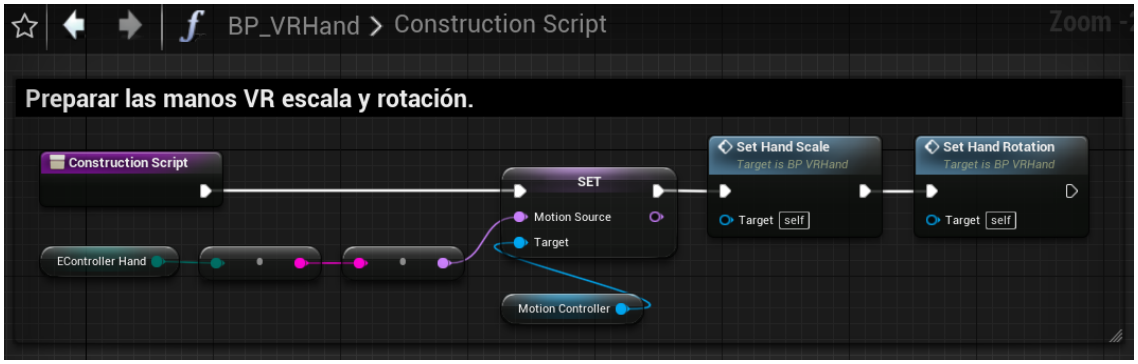
El siguiente paso fue crear el *VRPawn* que aparece cuando el jugador inicia el juego. *BPP_Pawn* tiene como *parent class* a *Pawn*, y *BPc_VRPawn* a *BPP_Pawn*. *BPc_VRPawn* es la *child class* de *BPP_Pawn*, que a su vez es la *child class* de *Pawn*, que a su vez es la *child class* de Actor, que a su vez es la *child class* de *Object*. De esta forma, *BPc_VRPawn* heredará todo lo que contengan sus *parent classes*. También tendrá la capacidad de ser reconocido por UE4, ser posicionado en una escena y ser transformado, ser manejado por un jugador, customizar las funciones del proyecto, etc.

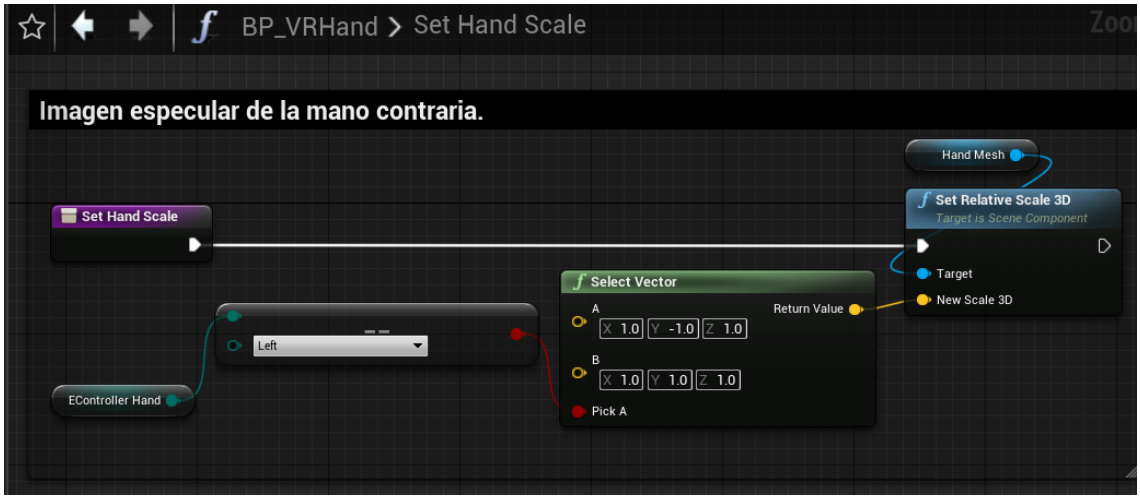


Dentro de *BPc_VRPawn* se creó el evento llamado "*PreapreVREnvironment*" el cual contiene algunas funciones para preparar al equipo para trabajar con cualquier HMD que utilice el usuario. Inicialmente se crearon funciones como "*IdentifyHMD*", "*Set Floor for HMD*" y "*Enable HMD*". Estas funciones se encargaban de identificar el tipo de HMD conectado por el

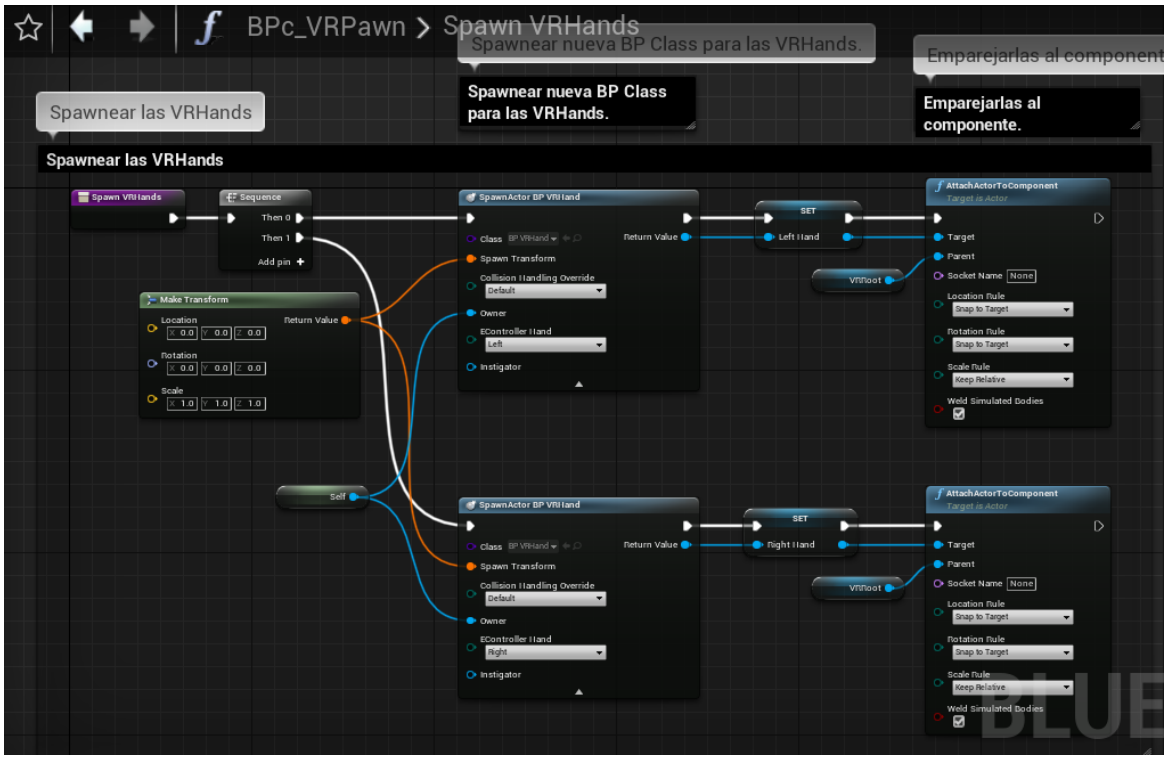
usuario, establecer el nivel del suelo y activar el modo VR respectivamente. Con cambios posteriores del proyecto todas estas funciones se descartaron y quedaron en desuso, puesto que el *plugin* de *OpenXR* se encarga de estas tareas.

Por lo tanto, a este evento solo le queda la función "*SpawnVRHands*", encargada de hacer aparecer las manos de VR. Durante este paso se crea una nueva *BP class* y luego hay que emparejarla a un componente, tanto para la mano izquierda como para la derecha. Dentro *NavVR/Player* se crea una nueva carpeta llamada *Motion Controllers*. Dentro se crea una nueva *blueprint class* de tipo actor llamada "*BP_VRHand*". Se le añade un componente *Motion Controller*. Se crea una nueva variable de tipo *EControllerHand*. *Unreal Engine* ya la incorpora, por lo que no es necesario crearla, pero sí se necesita activar las pestañas *Instance Editable* y *Expose on Spawn* para así poder cambiar los valores desde el *pawn*. Hay que abrir el *Construction Script* y arrastrar esta variable y convertirla a *string*. Se arrastra el componente *Motion controller* unido a un nodo *Set the Motion Source*, donde el *target* es el componente y la *Motion Source* la misma variable *enum*. Posteriormente en este documento se desarrollará el mecanismo por el cual se implementan las manos que trae *Unreal* junto con las animaciones. Es necesario ajustar la rotación y la escala con las siguientes funciones.



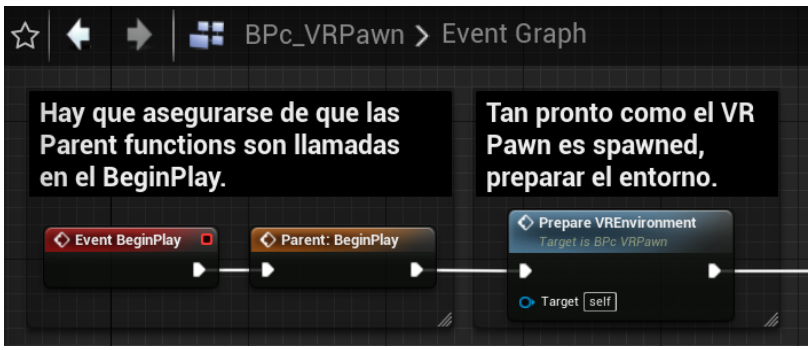


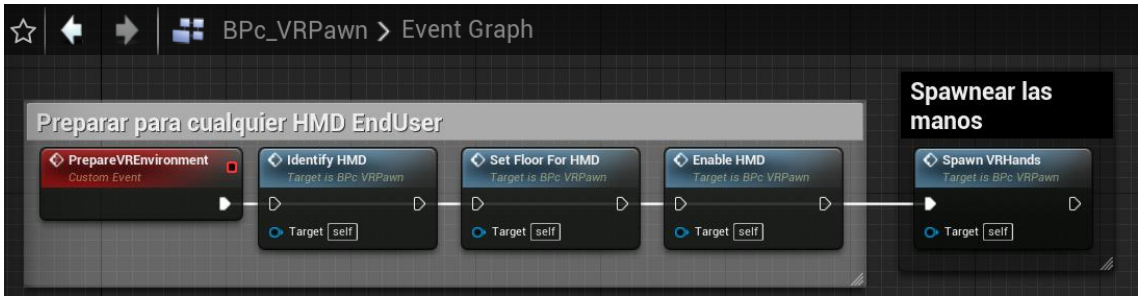
Regresando al *VRPawn*, primero se añaden los componentes que se emparejan con las manos. Hay que añadir un componente *scene* llamado "*VRRoot*". Luego se añade un componente *camera*. Para el *spawn* de los actores se utiliza el nodo *Spawn Actor from Class*, donde la clase es *BP_VRHand*. Se lleva la pestaña *spawn transform* a un *Make Transform* y *Owner* a *Self* (como referencia a sí mismo). Hay que proporcionar a este nuevo *spawned actor* una variable llamada "*LeftHand*". Esta variable va unida a un nodo *AttachActorToComponent* donde *Parent* es el componente *VRRoot* que hay arrastrar.



Se establece *Rotation Rule* y *Location Rule* a *snap to target*. Se hace lo mismo para la mano derecha. Se establece la pestaña de *EControllerHand* a *Right* y se promociona a otra variable llamada "*RightHand*".

Volviendo al *Event Graph* del *BPc_VRPawn* hubo que asegurarse de que el evento *PreapreVREnvironment* es llamado al inicio, llamado en el *Event Begin Play*. Es importante saber que cualquier cosa en el *Begin Play* del *parent blueprint* ha de ser llamada también en el *Begin Play* de las *child classes*, de otra forma no será llamada. Esto se resuelve haciendo clic derecho en el *Event Begin Play* y añadiendo *Add Call to Parent Function*.

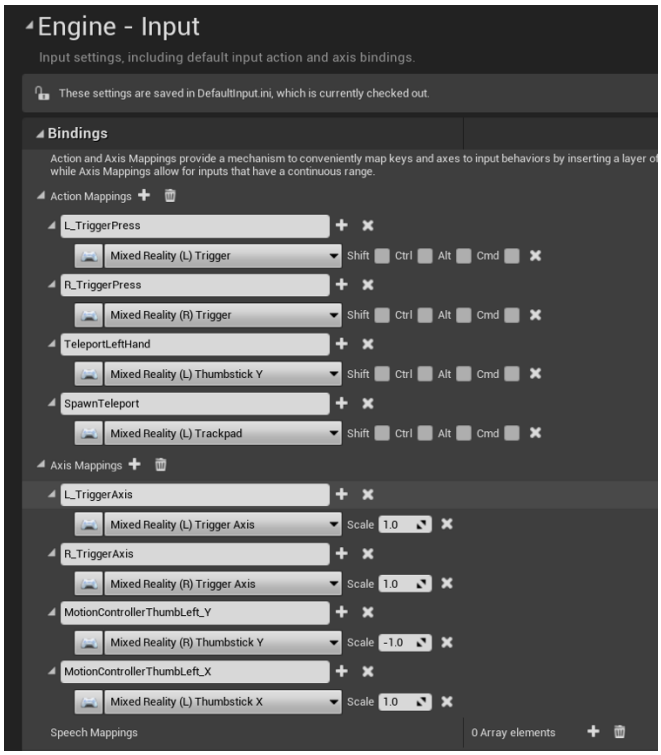




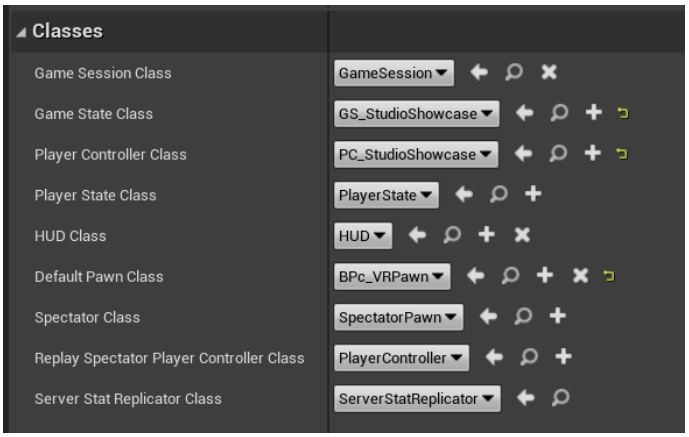
Lo siguiente fue desarrollar una manera en la que el usuario interactúe con la escena usando los *Motion Controllers*. Para ello se crean eventos de *Inputs Action/Axis Mapping* que contendrán diferentes funcionalidades. Es importante revisar las asignaciones de entrada o *Inputs Mapping* en la documentación oficial de los fabricantes del HMD en concreto. 17. Esta información es relevante para configurar la *key* de cada *action input* y *axis input*. De esta manera, los *inputs* VR estarían configurados para usar los gatillos – o *keys* similares– para diferentes *controllers*. Al iniciar el juego, y una vez conectados los *controllers* y el HMD, se deberían crear los *inputs* correspondientes. Estos *inputs* se configuran desde *Project Settings / Inputs*.

No está de más resaltar la diferencia entre *Action Mapping* y *Axis Mapping*. En el primer caso se habla de *inputs on/off*. Cuando se pulsa la tecla se llama a un evento y cuando se suelta también. En el segundo caso se habla de *inputs* donde la intensidad con la que se accione cuenta,

como un gatillo, un joystick, etc. Una cantidad entre 0 y 1, donde 0 es no accionar y 1 es accionar al máximo.



El siguiente paso fue establecer correctamente el *Game Mode* y todas sus opciones:



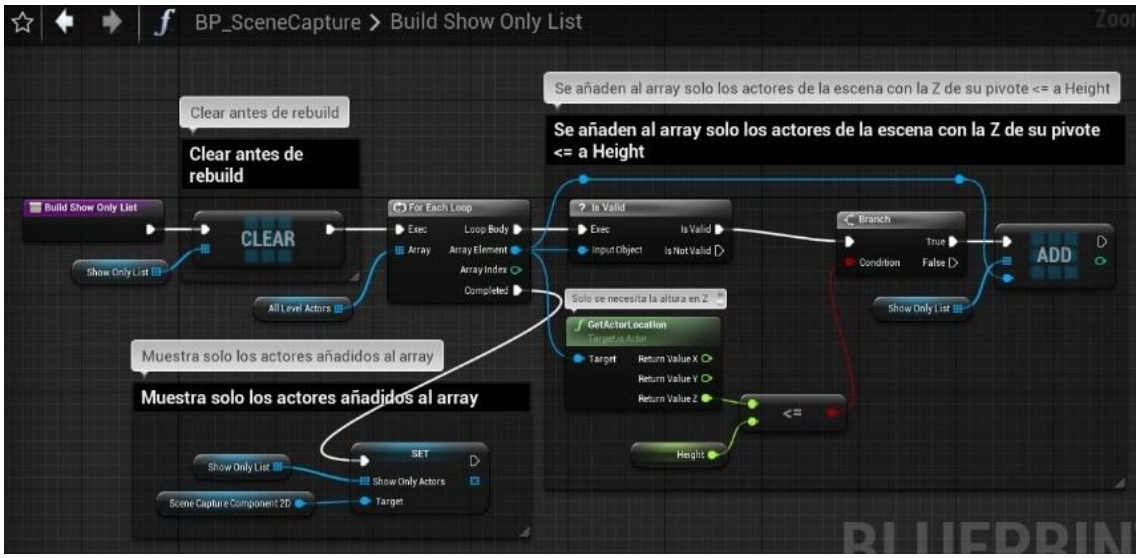
Para que el usuario pueda navegar por el nivel una de las opciones es desplegar el mapa de navegación donde se muestra una vista en directo de la escena, y donde además podrá seleccionar el punto al cual le interesa teletransportarse. El *Top Mode* en el mapa de navegación facilita al usuario ubicarse en las diferentes localizaciones en la escena, mientras que el *Preview Mode* le muestra una vista en primera persona de la localización antes de teletransportarse.

La construcción de un mapa de navegación comenzó creando un *scene capture component* con una vista en directo de la escena desde lo alto. Para ello se guarda la imagen en

un *render target* que luego se usará como *background* del mapa. En *NavVR / GamePlay / Actors* se crea un nuevo *blueprint class* de tipo *Actor* llamado "*BP_SceneCapture*". Lo primero es añadir el *Scene Capture Component 2D*. Este componente grabará la escena y lo usará para crear una textura que se usará en el *Widget*. La manera de hacerlo es mediante el uso de un *render target*. En *Details/Scene Capture* se selecciona *Texture Target*. Se crea y se guarda en una carpeta llamada "*Textures*" en *NavVR*. Se pone como nombre "*RT_SceneCapture*". También se necesita cambiar el *capture source*, para ser capaces de verlo en el *HMD*. Hay que cambiarlo a *Scene Color (HDR) in RGB, SceneDepth in A*. Luego se arrastra el actor a la escena y se coloca en lo alto. Obviamente, el techo de la escena tapará la vista general de la misma. Por ello, hay modificar el *Scene Capture Component* para que solo muestre actores debajo de una cierta altura. Dentro de *BP_SceneCapture* se crea una nueva variable de tipo *Float* llamada "*Height*". Se crea una nueva función llamada "*GetAllLevelActors*". Es importante recordar que si se obtienen todas las clases de tipo actor se tendrá una referencia de todo lo que en la escena se deriva de esa clase. Esto incluye *static meshes*, *blueprints*, luces, etc. Hacer una búsqueda de todo el nivel es algo extremadamente costoso. Por este motivo solo se hará una vez al inicio del juego. Para ello se promociona el valor devuelto a una variable llamada "*All Level Actors*". Posteriormente se probó esta lógica en el *Construction Script*. Hay que incluir un *Clear* del *array* antes de establecerlo de nuevo. *Clear* borra todos los datos dentro del *array* al que está conectado. Reestablece el *array* y elimina todos sus índices.

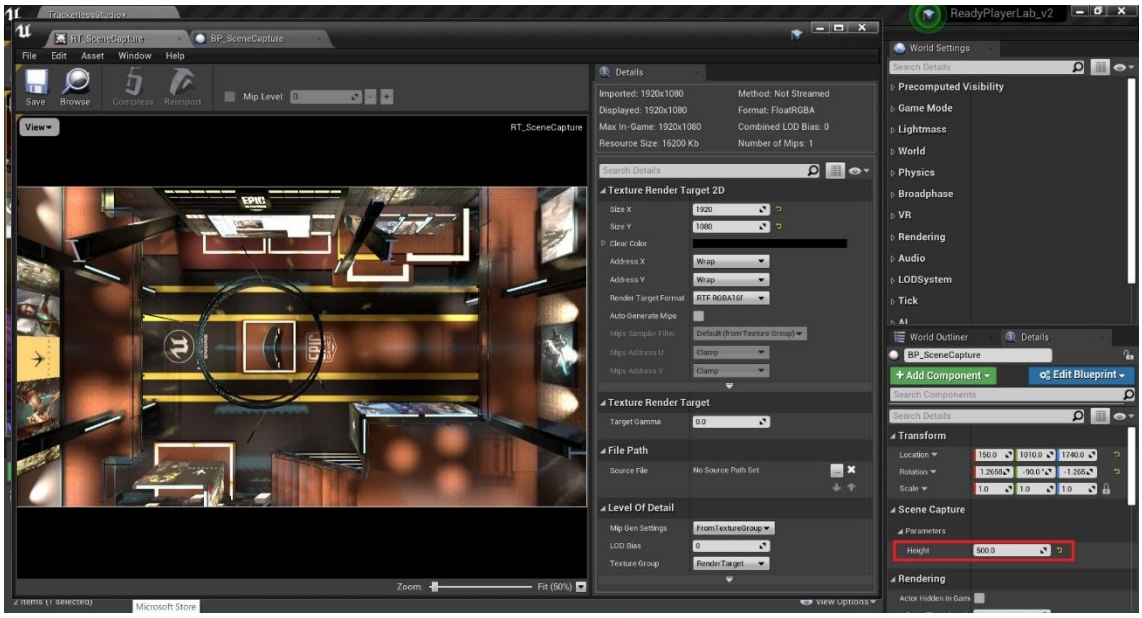
También hay que crear otra función llamada "*BuildShowOnlyList*". Hay que duplicar la variable *AllLevelActors*, llamada "*ShowOnlyList*". Se crea una referencia de ésta y de nuevo empieza con un *Clear*. Se crea una referencia de *AllLevelActors* y conecta a un nodo *For Each Loop*. Se irá iterando a través de cada referencia en el *array*, primero comprobando si es válido y luego dándole la localización del actor. Hay que recordar también que cuando el código está en funcionamiento a veces se elimina información para ayudar a la performance de la aplicación. Muchas veces pasa esto con variables que no son válidas o están vacías, o su referencia no está. Si esto ocurriera el juego colapsaría. Por ello, y para minimizar este riesgo, se ejecuta el nodo

"*?IsValid*". Se comprueba así si el objeto es válido o no y se ejecutan dos piezas de código para las dos posibilidades. Volviendo al *blueprint*, si la localización en Z o lo que es lo mismo, la altura, es menor o igual a la variable *Height* se añadirá a la lista. Si está por encima de la variable, no se añadirá y será ocultado para la cámara. Hay que arrastrar una referencia al *Scene Capture Component* y añadir un nodo *Set* de *Show Only List* unido a la pestaña *Completed* del *For Each Loop*. Luego hay que conectar una referencia a *Show Only List* en *Show Only Actors*. Por último, hay que hacer que la variable *Height* sea visible y editable para poder modificarla desde la escena.



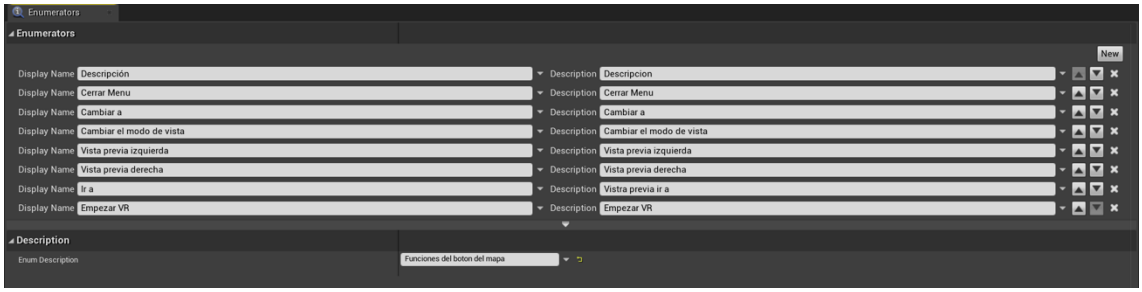
Hay que llevar las funciones *GetAllLevelActors* y *BuildShowOnlyList* al *Construction Script*. El último ajuste es cambiar el *Primitive Render Mode* a *Use Show Only List*. Luego se vuelve a abrir *RT_Scene Cpature* para ver cómo a medida que incrementa el valor de *Height*, se muestran más actores en la textura. Un valor óptimo es de 500. También hay que cambiar el tamaño de la textura a 1920x1080. Cuanto mayor sea esta resolución, más costoso será su renderizado, ya que lo hace cada *frame*. Por eso es importante dejar la resolución adecuada.

De esta forma se ha desarrollado un mecanismo para la creación de una textura 2D que representa la imagen en directo de la escena y que se usará como base para la visualización por parte del usuario del mapa de navegación.



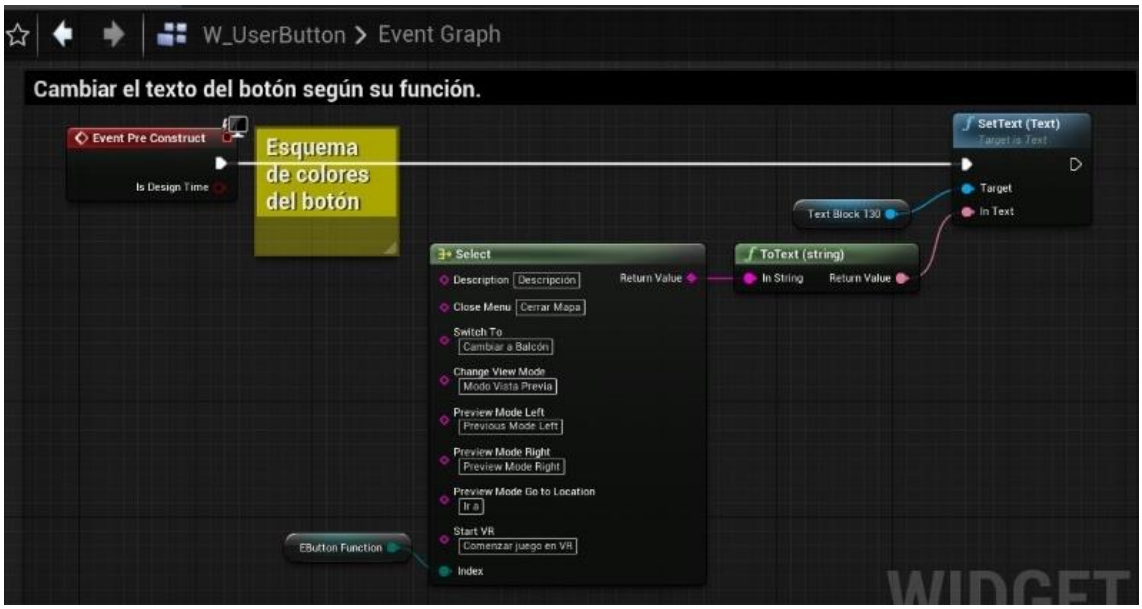
Lo siguiente fue construir el *Widget* del mapa de navegación y usar este *scene capture render target* para la visualización del mismo. Se usó el *scene capture render target* como fondo del mismo. Se creó la UI con ítems reutilizables y reprogramables, incluyendo botones personalizados. Además, se utilizaron variables públicas, por lo que se puede alterar su lógica con otros widgets.

Para desarrollar esto primero se crea una nueva carpeta dentro de *NavVR / UMG* llamada "*NavigationMenu*" y un nuevo *Widget* llamado "*W_TopMode*". Hay que abrirlo y reemplazar el *Canvas Panel* por un *Horizontal Box*. Luego hay que colocar una imagen dentro y seleccionar *RT_SceneCapture*. Posteriormente se crean los botones que se usarán a través del mapa. Dentro de *UMG* se crea una carpeta llamada "*Reusable*", y dentro un nuevo *widget* llamado "*Wc_MainButton*". Para este mapa solo se necesitará un botón que se reutilizará una y otra vez. De nuevo se reemplaza el *canvas panel* por un *horizontal box*, y dentro se coloca un botón ocupando todo el espacio. Hay que añadirle un evento *On Clicked*. Lo siguiente es crear una *Enumeration* llamada "*EButtonFunction*". Esta servirá para hacer una lista de las posibles funciones que tendrá el botón en el mapa.



Volviendo a *Wc_MainButton* se crea una nueva variable llamada "*EButtonFunction*" del tipo "*EButtonFunction*" conectada a un *Print String* para asegurarse de que el botón funciona. Luego se crea otro *widget* en *Reusable* llamado "*W_UserButton*". Se reemplaza el *Canvas Panel* por un *Size Box*. El propósito de este *widget* es contener el texto de los botones en el mapa. De esta forma se tiene un solo *asset* reutilizable que informa al usuario de la función de un cierto botón. Dentro de la *Size Box* se añade un *Overlay* para poder tener múltiples hijos en la jerarquía. Dentro se coloca una imagen para el fondo y un *text block*. Por último, se coloca el botón *Wc_MainButton*. Es importante que dentro de la jerarquía *UMG* este aparezca al final para que pueda recibir el clic correctamente por parte del usuario y cumpla su función.

Dentro del *Designer* de del *W_UserButton* se marca la pestaña *Is Variable* del botón. En el *Event Graph* se añade el nodo *Event Pre Construct*. Se añade también la misma variable *EButtonFunction*. Con ella se puede cambiar el texto según el valor de la enumeración. Se usa el nodo *Select* para establecer el texto con ayuda del nodo *To Text (from string)*.



Se necesita informar al *Main Button* cuál es el valor del *EButtonFunction* en el *Event Construct*, por lo que se establece el valor dentro de *Wc_MainButton*.

Volviendo a *W_TopMode* y dentro del *Horizontal Box* se coloca un *Overlay* y dentro de este el "*Image_LiveMap*" utilizando *Wrap With*. Aquí es donde se coloca también el *canvas panel* que contendrá los puntos del mapa. También se coloca una *Vertical Box* para una barra lateral de botones. Se establece el *canvas panel* a *fill* y *la vertical box* a *auto*. Esto debería hacer que el *canvas panel* se ajuste al máximo al espacio y que la *vertical box* solo ocupe el espacio que necesite. Luego se busca el botón para el usuario en la *Palette* y lo se arrastrará tres veces. Hay que cambiar el valor de *EButtonFunction* al que se desea desde el panel *Details* del botón. Por último, se añade un nuevo botón para *Description*.

De esta forma se han creado varios *widgets* y se ha empezado a construir la jerarquía *UMG* para el mapa de navegación. También se les ha otorgado a los botones una variable editable, por lo que cada uno producirá diferentes resultados al ser pulsados. Lo siguiente fue habilitar la aparición del mapa de navegación desde el *VR Pawn* usando los *motion controllers*.

Para ello hay que añadir la habilidad para hacer aparecer el *widget* del mapa de navegación cuando el usuario presione ambos gatillos de los *motion controllers*. También hay que añadir la posibilidad de reescalar el mapa y cerrarlo. Para desarrollar esto se hará necesario crear un nuevo *3D map widget* que contendrá el mapa y su lógica. Se crearán algunas funciones en el *VR Pawn* para hacer *spawn* y manipular el mapa con los *motion cotrollers* usando datos que se recibirán de algunas ecuaciones simples. Una será para la localización 3D del punto medio entre las dos manos para colocar el mapa. Otra será la distancia entre las dos manos para determinar el tamaño y validez del mapa.

También hay que crear un nuevo *blueprint* de la clase *Actor* llamado "*BP_3DMap*". Se le añade un componente *Widget* y se le llama "*Navigation Menu*" y se selecciona la clase *W_TopMode*. En el *Viewport* se puede ver ahora el mapa. En los botones aparece "*TextBlock*" debido a que la *Preconstruct Function* dentro de los botones aun no se ha iniciado. Se cambia el

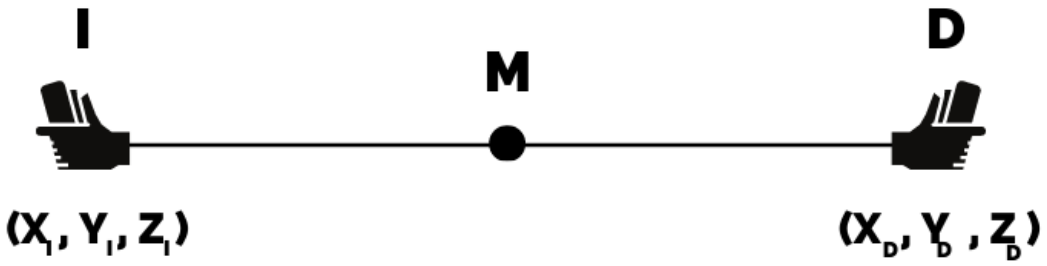
tamaño a 1920x1080, esto le dará suficientes píxeles para tener una alta resolución. Para verlo en escena a un tamaño más pequeño se puede reducir la escala 10 veces hasta 0,1 en el *Transform*.

Luego hubo que construir la lógica para que el usuario pueda hacer aparecer el mapa y manipularlo. Para ello es necesario que se pulsen ambos *triggers*. Cuando se suelta uno de los *triggers*, el mapa se queda dónde está; y si los *motions controllers* no están muy cerca, permanecerá visible.

Esto requiere abrir *BPc_VRPawn* y crear dos nuevas variables booleanas llamadas: "*bLeftHandPressed*" y "*bRightHandPressed*". Se hace un *set* de ambas variables después de pulsar y soltar los gatillos. Luego se añade un *Branch* justo después para saber si el gatillo opuesto está presionado. Si es *true* es que ambos están presionados, y si es *false* es porque su opuesto no está siendo presionado. Se crea una nueva función llamada "*SpawnMap*" y un nuevo evento llamado "*Start Map Timer*" que será llamado después de que el usuario mantenga presionado ambos gatillos. En la función *SpawnMap* se hace *spawn* el *3DMap* desde el *Transform* del *VRRoot* y se promociona a una variable.

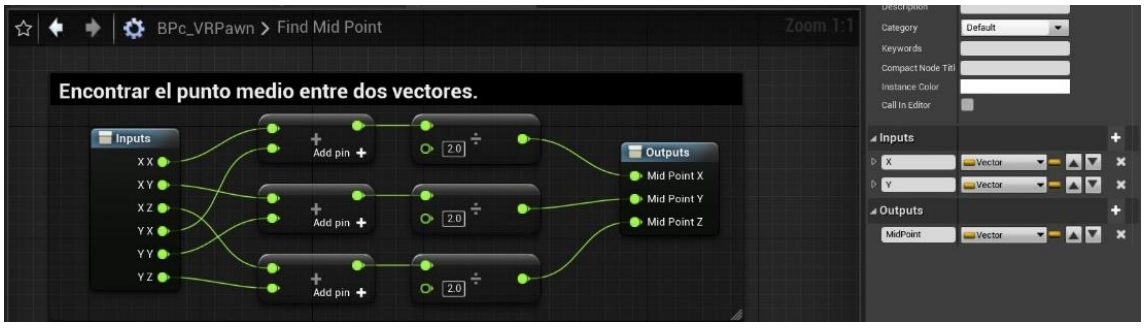
Cuando el *Start Map Timer* es llamado, primero se comprueba si el *BP_3DMap widget* es válido, ya que se quiere que aparezca solo una vez. Si no es válido, aparecerá el mapa, y si es válido, se continúa. Se crea otra función llamada "*AdjustMapTransform*". Esta es la función que tendrá el control del mapa y actualizará, en cada *frame*, la posición del punto medio entre los dos *controllers* y la rotación para mirar siempre hacia el *HMD*.

Se sabe que los *controllers* llevan asociados en todo momento un vector de posición. Se puede llamar a estos vectores $I(x_I, y_I, z_I)$ y $D(x_D, y_D, x_D)$ para el *controller* de la mano izquierda y derecha respectivamente. Para encontrar el vector correspondiente al punto medio, llamado M, hay que considerar el segmento ID que une ambos vectores. Por consiguiente, el segmento IM y el segmento MD han de medir lo mismo. Entonces el vector del punto M sería:



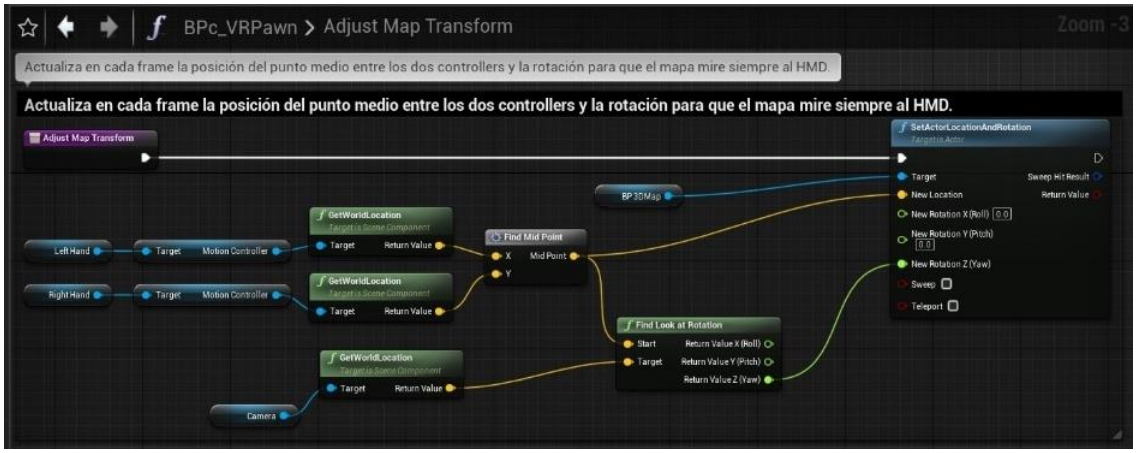
$$M(\frac{x_I + x_D}{2} + \frac{y_I + y_D}{2} + \frac{z_I + z_D}{2})$$

Al pasar este razonamiento a *blueprints*, para encontrar el vector del punto medio se crea un *macro* llamado "*FindMidPoint*". Se necesitan dos *vector inputs* (I y D) y un *vector output* (M). Para poder hacer los cálculos se necesita activar la opción *Split* del *pin* que representa cada vector.

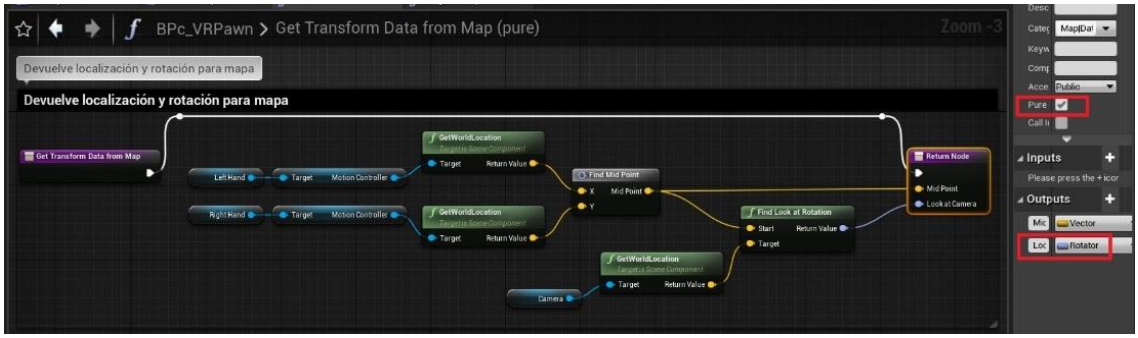


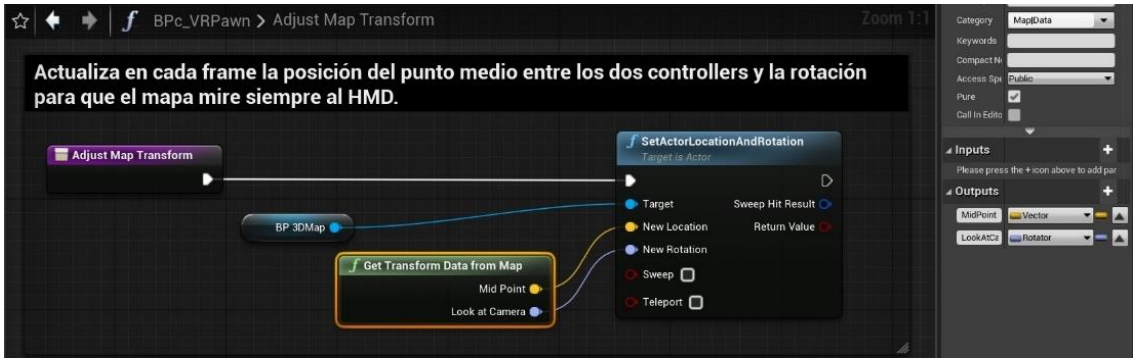
Así se obtuvo un *macro* que proporciona el punto medio entre dos vectores en un espacio tridimensional. Dentro de la función *Adjust Map Transform* se obtienen los valores de localización de ambos *controllers* y se le pasa a la función *Find Mid Point*. Se lleva el *output* al

nodo *Set Actor Location And Rotation* de la variable *BP_3DMap*. Para la rotación se usa el nodo *Find Look at Rotation*. Se utiliza esta nueva localización del punto medio unida al pin *Start*, y para el *Target* se utiliza el nodo *Get World Location* del componente *Camera*. La rotación en el eje Z queda entonces unida al pin de rotación en Z del nodo *Set Actor Location and Rotation*.



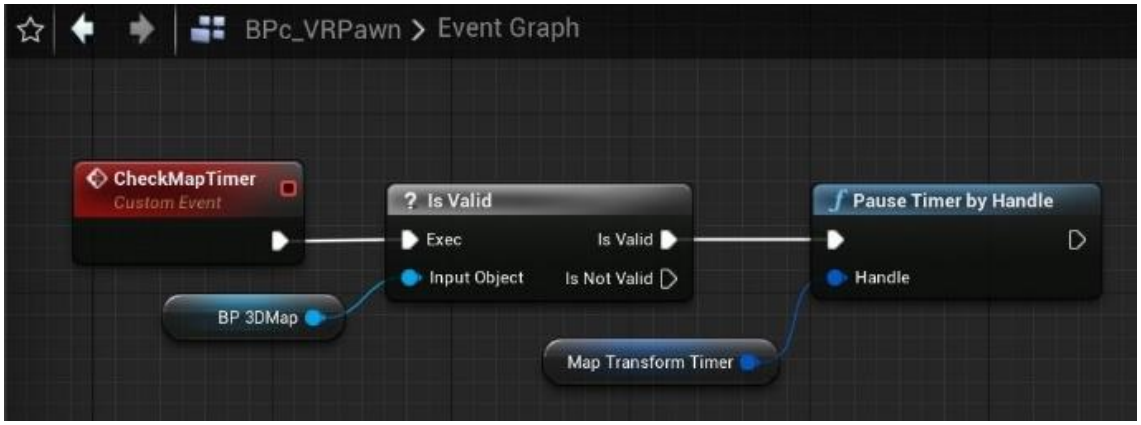
Se pueden colapsar todos estos nodos sobre rotación y localización en una nueva función llamada "*Get Transform Data From Map*". Se deshace el *Split* de la rotación y se cambia el *output* a una variable llamada "*LookAtTransform*" de tipo *rotator*. Por último, se marca la casilla *Pure* del panel *Details*. Sobre esto último, hay que recordar unos conceptos. Una función normal, es una función que se ejecutará siempre que se conecten sus nodos al hilo de ejecución. Por otra parte, una función pura es una función que se ejecutará cuando se le solicite datos.

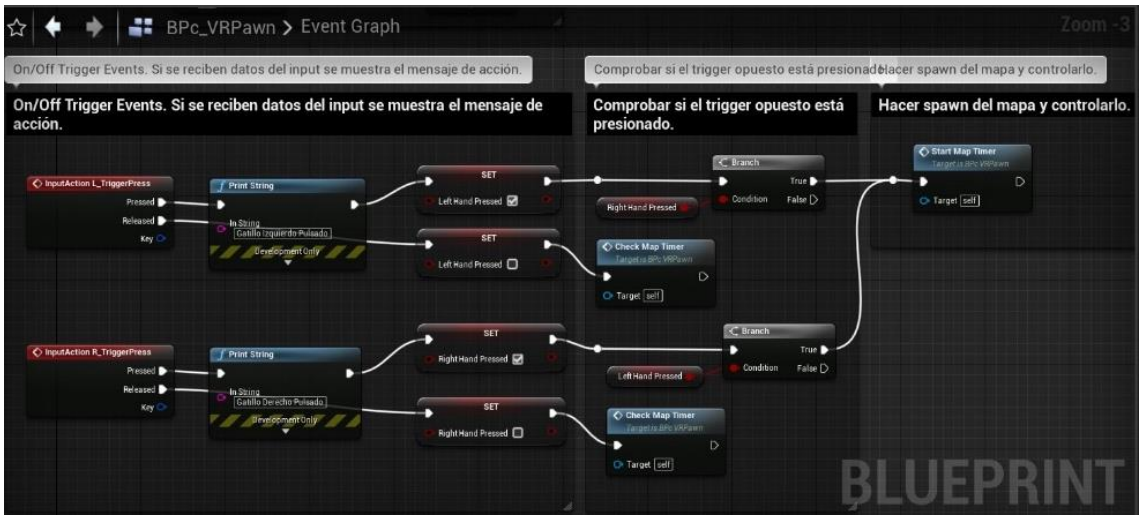




Volviendo al evento *Start Map Timer*, hay que preguntar si el *BP_3DMap* es válido. Si es válido se ejecutará el nodo *Set Timer by Event*. Esto llamará a un evento y, si se quisiera, se podría hacer en bucle hasta pausarlo. Este nodo es como una especie de *Delay* con tiempo programable que delega en alguna acción por evento o por función, y que puede ser en bucle o no dependiendo de si se marca la casilla *looping*. Se establece el tiempo en 0,02 segundos. Desde la casilla *Event,* se coloca el nodo *Create Event*, el cual dará una lista de todos los eventos o funciones utilizados que pueden trabajar con un *timer*. Se selecciona *Adjust Map Transform* y se le proporciona este *Timer* a una variable del tipo *Timer Handle* llamada "*MapTransformTimer*", por lo que se podrá pausar cuando se necesite.

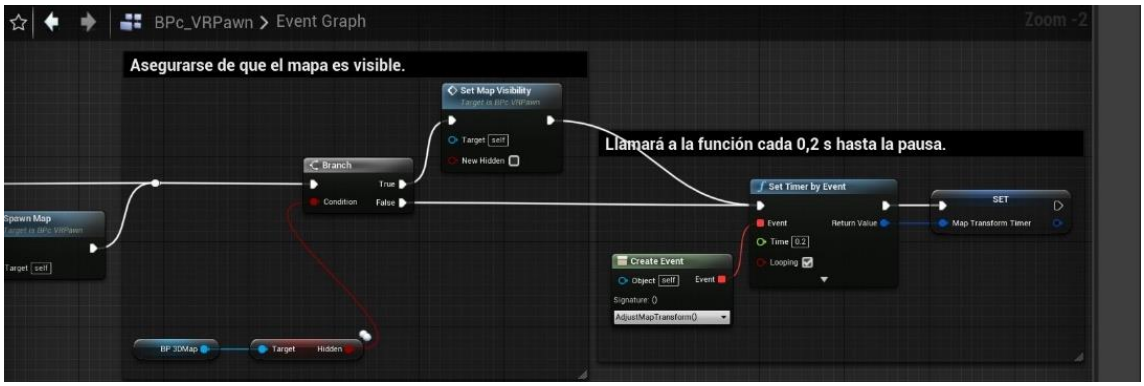
Se une el *Start Map Timer* en las pestañas *True* de los nodos *Branch* de los *Inputs Events* de los *Controllers* después de pulsar el *trigger*. También se necesita un nuevo evento para comprobar esto y pausar el *Timer* cuando se libera el *trigger*, llamado "*Check Map Timer*". Después de otra comprobación de la validez del mapa se añade un nodo *Pause Timer by Handle*. En la pestaña *Handle* se conecta una referencia a la variable *MapTransformTimer*. Ahora, se conecta el *Check Map Timer* al *Released Input* de los nodos *Input Action Press*.



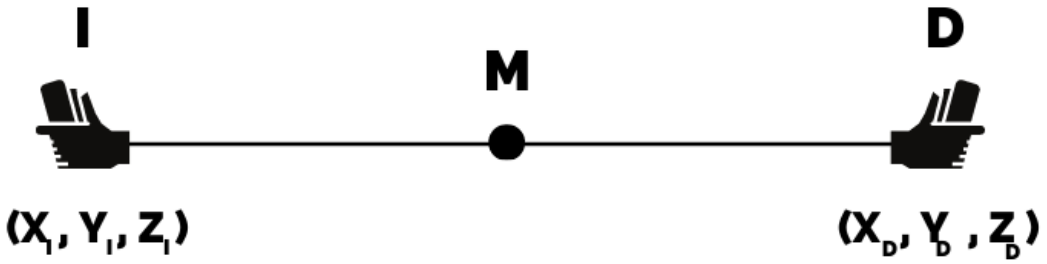


Luego se añadió la lógica para determinar cuando el mapa debe ser visible o no. Cuando el *motion controller spawnea* el mapa, primero lo *spawneará* y luego comprobará si está oculto. Si está oculto, lo muestra. Se crea una nueva función llamada "*Set Map Visibility*" y un *input* de tipo booleano llamado "*New Hidden*".

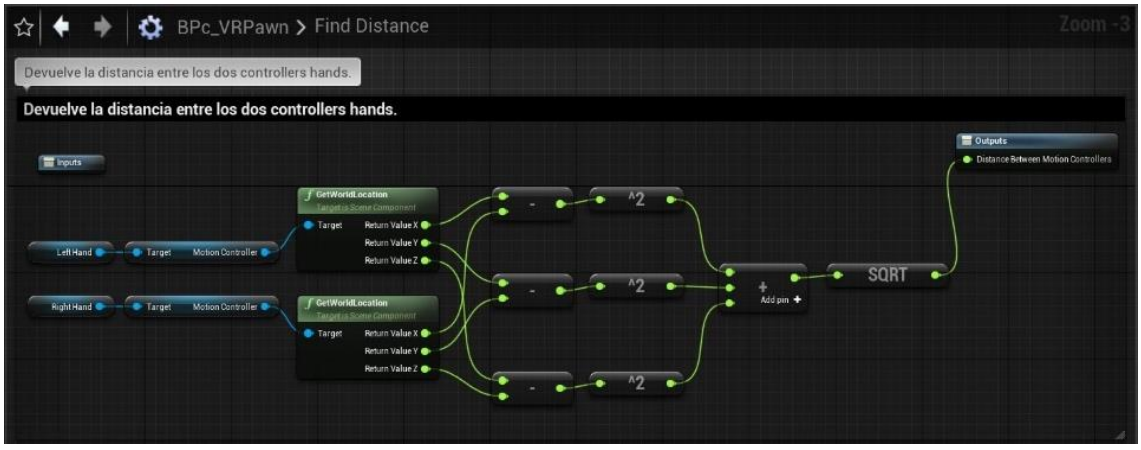
Por tanto, en el *Event Start Map Timer*, después de comprobar si es válido y de *spawnear* el mapa, hay que asegurarse de que el mapa es visible. Si la variable *bHidden* es *False*, significa que no está oculto, y se procede con el código. Si es *True* significa que está oculto, se muestra estableciendo a *false* la variable *New Hidden* en el nodo *Set Map Visibility* y se procede con el resto del código.



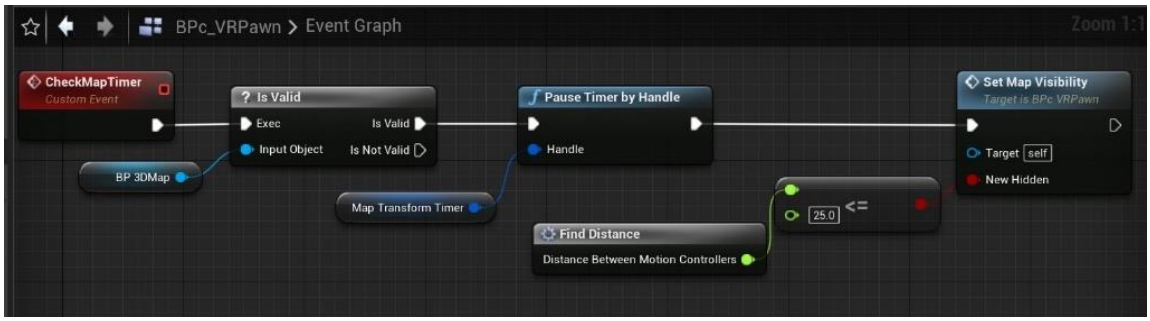
Después de soltar el *trigger* se oculta el mapa si las manos están muy cercas la una de la otra. Esto también puede usarse para cerrar el mapa rápidamente. Para determinar esto, hay que crear un nuevo *Macro* llamado "*FindDistance*". Para calcular la distancia entre dos vectores de posición, es necesario calcular la raíz cuadrada de la suma de los cuadrados de la diferencia entre cada componente, es decir, hay que calcular el módulo del vector ID:



$$Distancia = |ID| = \sqrt{(x_D - x_I)^2 + (y_D - y_I)^2 + (z_D - z_I)^2}$$



El valor resultante de distancia se conecta al *output*, que es una variable de tipo *float* llamada "*Distance Between Motion Controllers*". Para la distancia se establece 25 como valor umbral para cerrar el mapa. Si la distancia es menor o igual a 25 cuando se deja de pulsar el *trigger* se cerrará el mapa.



Por último, se añade un desplazamiento en la ubicación del *BP_3DMap*. Dentro del actor, hay que añadir el componente *Scene* llamado "*Scene_Widgets*". Se coloca dentro el *Navigation Menu*. Se cea una nueva función llamada "*Set Relative Location on Widgets*" con un *input* llamado "*Distance*" de tipo *float*. Esta función establece la localización del *Scene Component*. Solo se quiere desplazar el actor en el eje x, por lo que se desplazará solo hacia adelante o hacia atrás según la distancia de los *motion controllers*. Se añade un nodo *Map Range Clamped* al *input Distance* de la función. Básicamente, este nodo permite mapear o reasignar un número desde un rango de valores a otro, conservando siempre su misma proporción. La diferencia con el nodo *Map Range Unclamped* es que éste, a diferencia del anterior, sí puede sobrepasar los límites mínimo y máximo de los *out range*. En este caso, se ha impuesto como límites de distancia entre los *motion controllers* un mínimo de 0 y un máximo de 200 (en unidades de *Unreal*, centímetros). Estos valores son "traducidos" a desplazamiento del actor. Cuanto más cercano a 0, más lejano debería verse el actor, y viceversa.

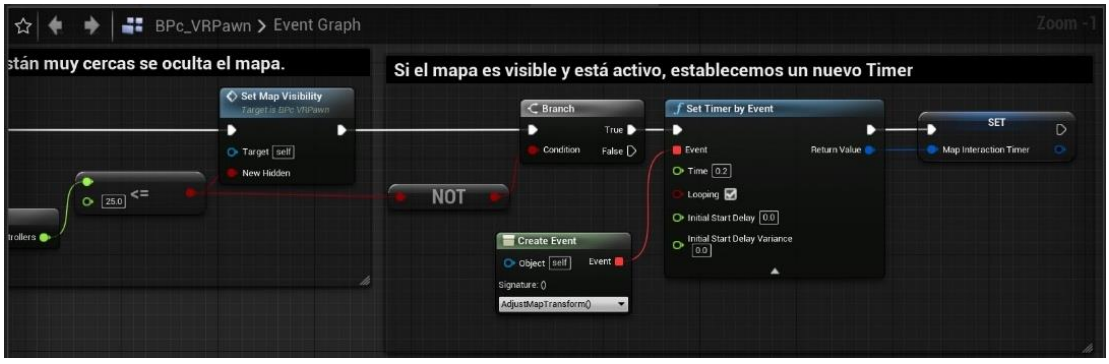
Por último, se llama a esta función dentro de la función *Adjust Map Transform* y se le pasa la distancia.

De esta forma se ha habilitado el *spawn*, el cambio de tamaño y el cierre del mapa de navegación usando los *motion controllers*. Lo siguiente fue añadir las mecánicas para interactuar con el mapa para establecer diferentes opciones y el teletransporte.

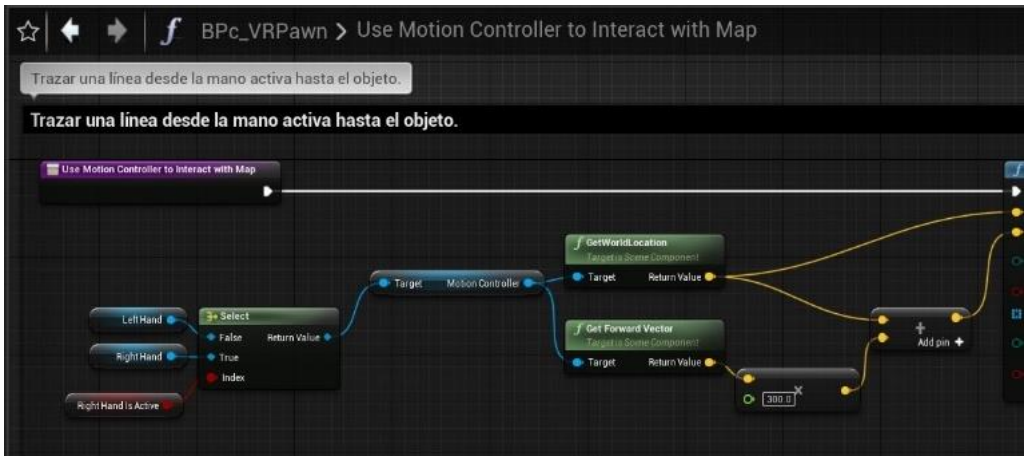
El *VR Player* necesita un *Widget Interaction Component* mediante el cual poder recibir datos de los *motion controllers* e interactuar con el mapa de navegación. Una vez detectados los *motion controllers*, se constriyó la lógica para presionar un botón en el mapa.

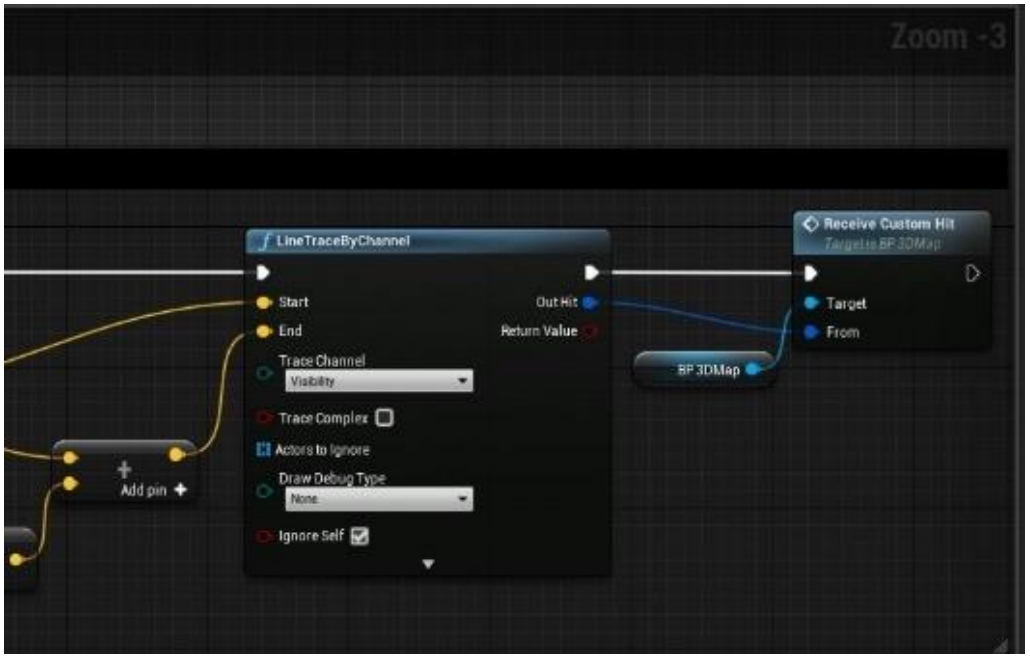
Para ello se abre el *BP_3DMap* y se le añade un *Widget Interaction Component*. Este es un componente que permite interactuar con un *Widget Component*. Esta clase permite simular un puntero láser que al apuntar a un *Widget* envía señales como si un puntero de ratón estuviera sobre él. También se puede decir al componente que simule un *Key Press*, como *Left Mouse*, para simular un clic de ratón. A este *Widget Interaction Component* se le cambia la *Interaction Source*, en el panel *Interaction*, a *Custom*. Además, se marca la pestaña *Show Debug* en el panel *Debugging*. Se crea una nueva función llamada "*Receive Custom Hit*" con un *Input* de tipo *Hit Result* llamado "*From Controller*".

Dentro de *BPc_VRPawn*, al final del *Check Map Timer Event*, se añade un *Branch* para comprobar si el mapa es visible y está activo. Si lo está, se establecerá un nuevo *Timer* y se enviará el *Custom Hit* al *blueprint* del mapa. A este nuevo *Timer* hay que configurarlo exactamente como el anterior y proporcionarle una variable de tipo *Timer Handle* llamada "*Map Interaction Timer*".



El evento al que se llama es una nueva función de nombre "*Use Motion Controller To Interact With Map*". Se usa un nodo *Line Trace By Channel* para proporcionar los datos que se necesitan enviar al *Widget Interaction Component*. Se establecen los datos del nodo según la mano que esté activa, es decir, según cuál de las manos ha pulsado más recientemente el *trigger*. Para ello se usa un nodo *Select* y las referencias a los componentes *Left Hand* y *Right Hand*. La información de la actividad la proporciona la variable booleana *RightHandIsActive*. De ahí se obtiene el *World Location* y el *Forward Vector* de la mano activa. De este modo se traza una línea desde la mano hasta el objeto mapa. Luego hay que llamar a la función *Receive Custom Hit* del *BP_3DMap*.

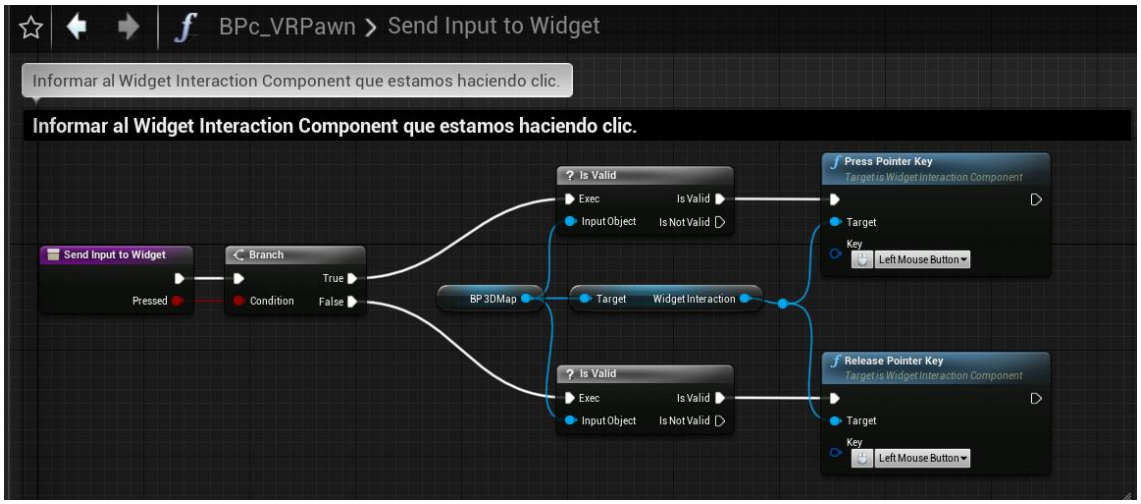




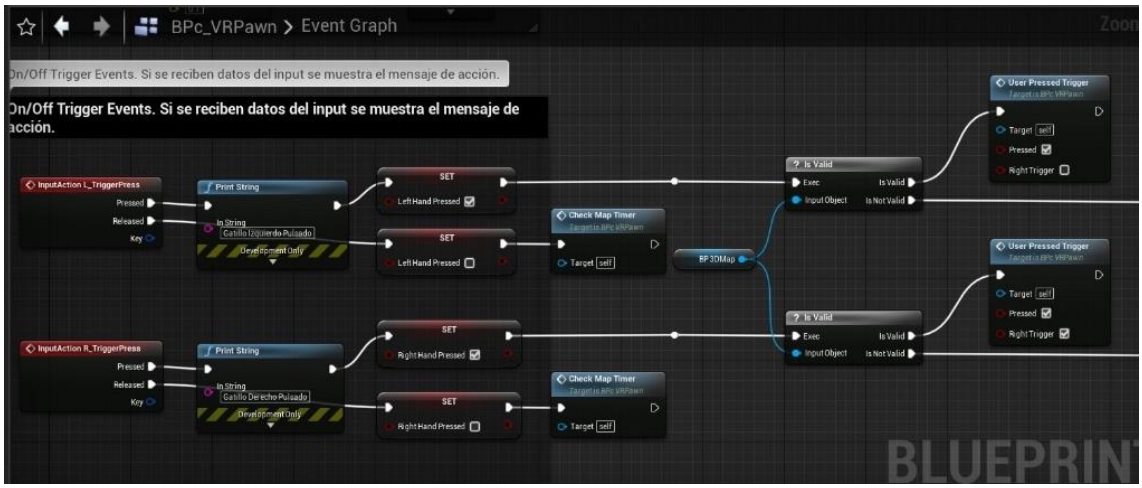
Se necesita otra función para los eventos *Input Action Trigger Press*. Por ello se crea una nueva función llamada "*User Press Trigger*" con dos inputs booleanos: "*Pressed*" y "*RightTrigger*". Cuando el *trigger* es presionado se establece la mano que lo ha activado como

activa. Luego la lógica transcurrirá por sí misma en la función *Use Motion Controller To Interact With Map*.

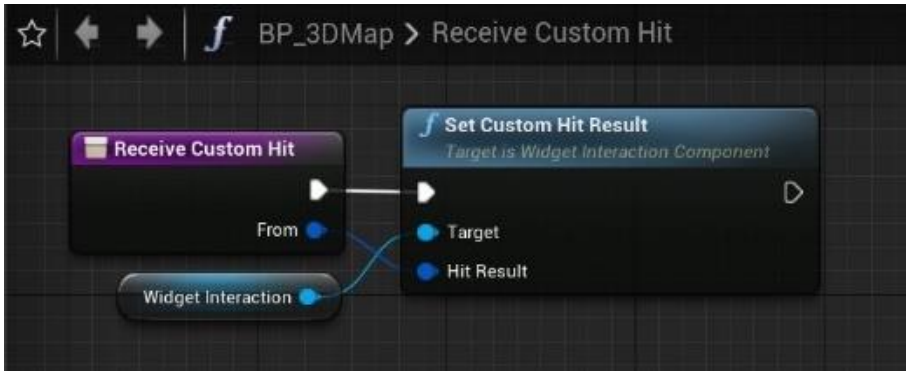
Se necesita otra función para informar al *Widget Interaction Component* que el usuario está haciendo clic. Por ello se crea otra función llamada "*Send Input To Widget*" con un *input* booleano. Se conecta a un *Branch*, si es *true*, se añade un *Press Pointer Key* desde el *Widget Interaction Component.* Si es *false*, se añade un *Release Pointer Key*. Se establece ambas *keys* como *left mouse button*. Luego se añade esta función dentro de la función *User Press Trigger*. En *true* se marca la pestaña *Pressed*, y en *false* no.

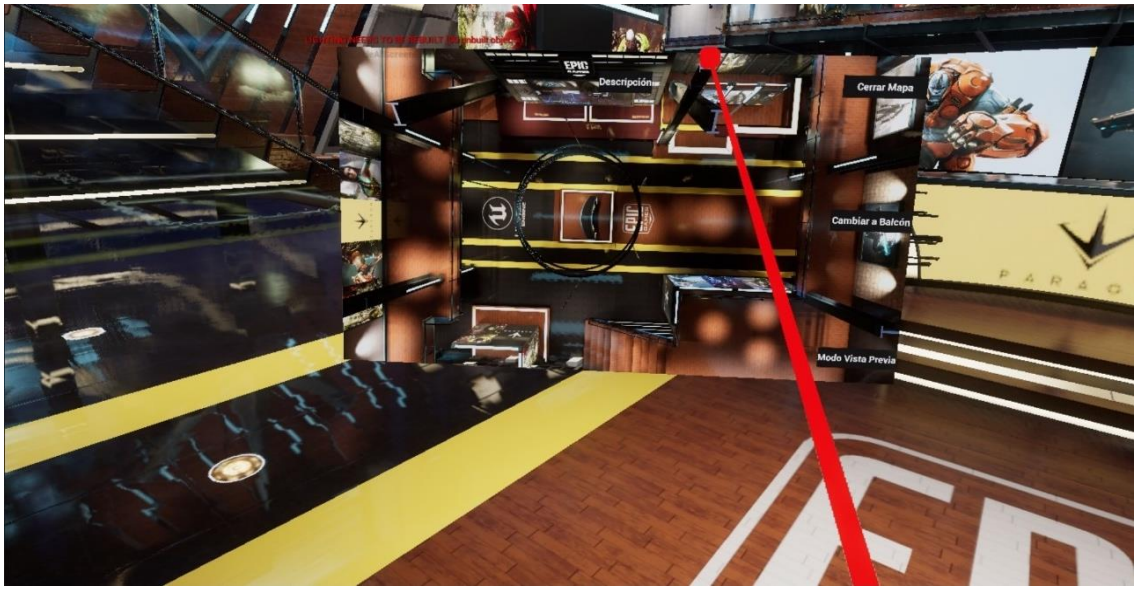


Se vuelve ahora a los *Input Action Events* en el *Event Graph* del *BPc_VRPawn*. Después de presionar el *trigger*, se establece su respectiva variable a *true*. También se añade una comprobación de validez del *BP_3DMap* antes de añadir la función *User Pressed Trigger*.



Volviendo a *BP_3DMap* se añade el nodo *Set Custom Hit Result* antes de probar el resultado. Al testearlo, después de hacer *spawn* el mapa, se puede ver la *debug line* desde el *motion controller* activo.





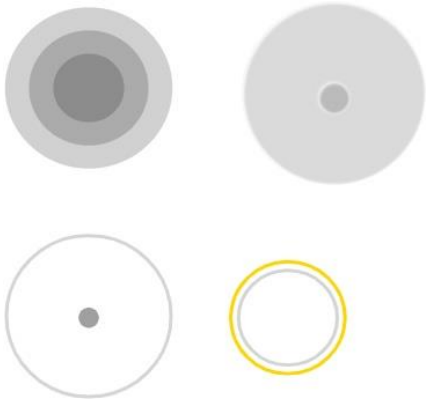
Posteriormente se añadió un cursor y se eliminó la *debug line* desactivando la casilla *Show Debug* del *Widget Interaction Component*. Para ello es necesario crear un nuevo *Widget*

llamado "*W_PlayerCursor*" con una *Size Box* como *Root*. Se añade una imagen a la jerarquía, que de momento será un simple cuadrado gris. Se añade este *widget* a la *Scene Widget Component* dentro de *BP_3DMap* y se le llama "*UserCursor*". Esta parte es muy importante. Hay que asegurarse de que el cursor no tiene colisiones, de lo contrario se recibirá el *hit* del *pawn* y no del mapa como se pretende. Para ello, en la pestaña *Collision* dentro del menú *Details* se establece *Collision Presets* como *No Collision*. No se quiere ver el cursor a menos que se esté apuntando al mapa. Por ello, dentro de la función *Recieve Custom Hit*, desde el *input From Controller,* se añade un nodo *Break Hit Result* y un *Branch*. La condición será que, si el *Hit* es con el *Navigation Menu*, el cursor será visible. De lo contrario, no lo será. Si es *true*, se quiere que se mueva a lo largo del movimiento del usuario. Se añade un nuevo nodo *Set World Location And Rotation* para el cursor, donde la localización de éste será la localización del *Impact Point* y la rotación en x se obtendrá de *Impact Normal*.

Para representar los puntos de interés del mapa − en adelante POI−, los puntos de teletransporte, se usó la aplicación Adobe Photoshop. También podría haberse usado cualquier otro software similar como *Adobe Ilustrator* para realizar gráficos vectoriales. Estas imágenes se exportaron como archivos de formato *png* que luego se incorporaron al proyecto. Después se construyó un nuevo *UMG Widget* que contiene los gráficos de estos POI, así como una nueva enumeración para que cuando el jugador seleccione un *POI Widget* se teletransporte a un lugar concreto. Posteriormente se creó un nuevo *Blueprint Actor Class* que actúa como un localizador de la posición en el mapa.

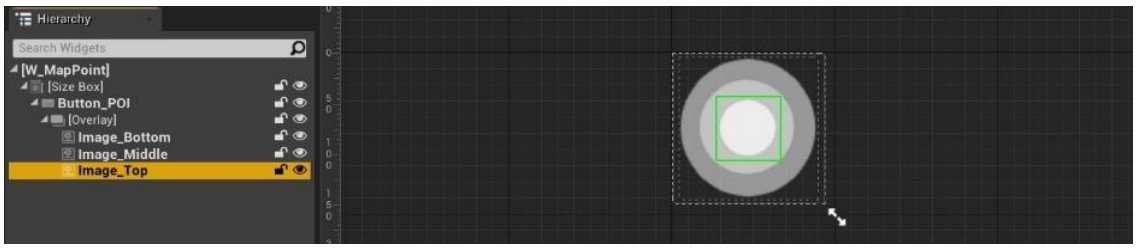
Se comenzó desarrollando la *UI* en Photoshop. Se diseñaron tanto los botones como el cursor. Para esta tarea hay que asegurarse de que los diseños tengan nombres identificables, como: "*POI_CircleBottom*", "*POI_CircleMiddle*" y "*POI_CircleTop*" para las tres capas de cada

botón; y "*UserCursor*" para el cursor. Para guardar estos archivos se crea una nueva carpeta llamada "*Images_UI*" dentro de la carpeta *Textures*.



Diseños para el cursor y la animación de clic.

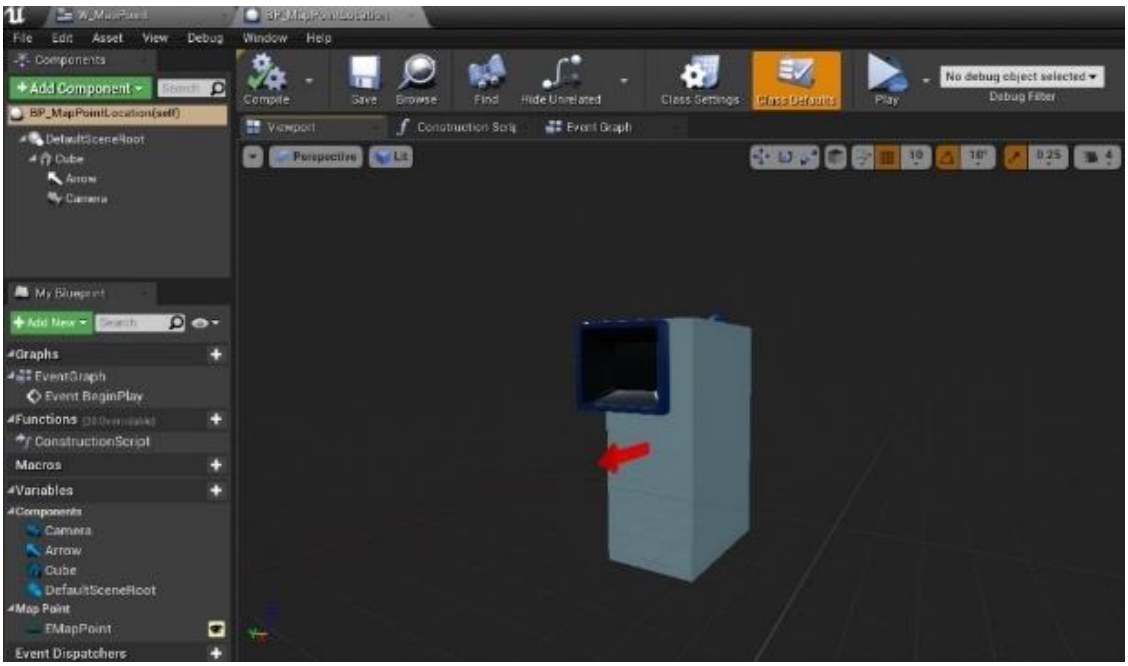
También se crea un nuevo *widget* en la carpeta *Reusable* llamado *W_MapPoint*. Se reemplaza el *Canvas Panel* por un *Size Box* y se le coloca un botón. Dentro de este se coloca un *Overlay* con las tres imágenes.



Luego se crea una variable de tipo *EMapPoint* del mismo nombre con *Instance Editable*. El mapa puede dividirse en dos zonas: planta alta, donde se encuentra el balcón, y planta baja. Por ello se crea una variable booleana llamada "*bIsBalcony*" para saber si el punto en concreto es un *Balcony Point* o no, es decir, si está en la planta alta o no. También se necesita otra variable *boolena* llamada "*bIsActive*" para determinar si ese punto en concreto debe ser visible o no. Cuando el usuario seleccione la opción *Balcony*, solo aparecerán en el mapa los botones de la planta alta, por lo que estos estarán activos y los de la planta baja inactivos. Cuando el usuario cambie la vista a la planta baja, sucederá lo contrario.

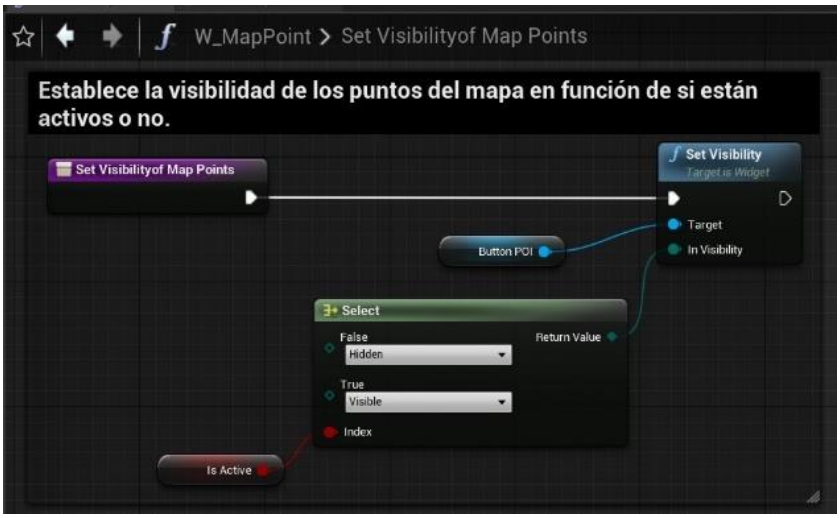
Luego se crea otro *blueprint* de tipo actor llamado "*BP_MapPointLocation*". Se distribuyen estos actores por todo el mapa, ya que otorgan las localizaciones de los lugares a los que poder teletransportarse. Este *blueprint* consiste en un cubo sin colisión con un componente *Arrow* para saber la cara hacia la que mira. También se puede incluir una cámara para simular

una vista previa. Hay que asegurarse de añadir la variable *EMapPoint* con *Instance Editable*. Además, en el *Event Begin Play* se destruyen los cubos, ya que solo se necesita saber dónde se localizan los puntos del mapa en el *Widget*. Se distribuyen en el mapa con el valor correspondiente a su enumeración *EMapPoint*. Hay que asegurarse de colocar los actores con el pivote en el suelo, ya que la lógica usará este punto como *transform* para el teletransporte.



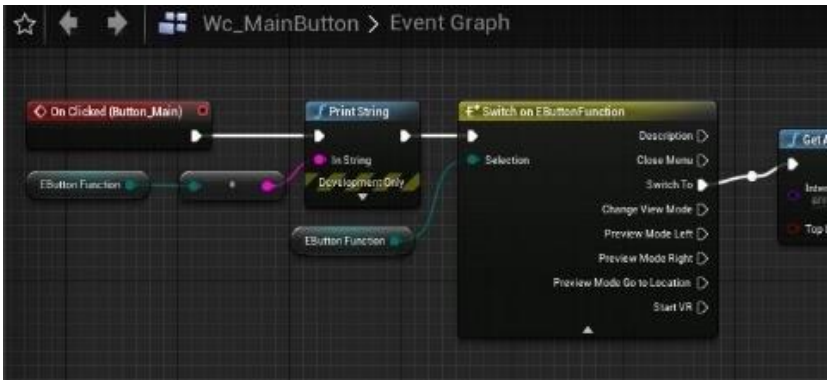
Posteriormente, al volver a *W_TopMode*, fue necesario colocar los *W_MapPoints* en el *Canvas Panel*.

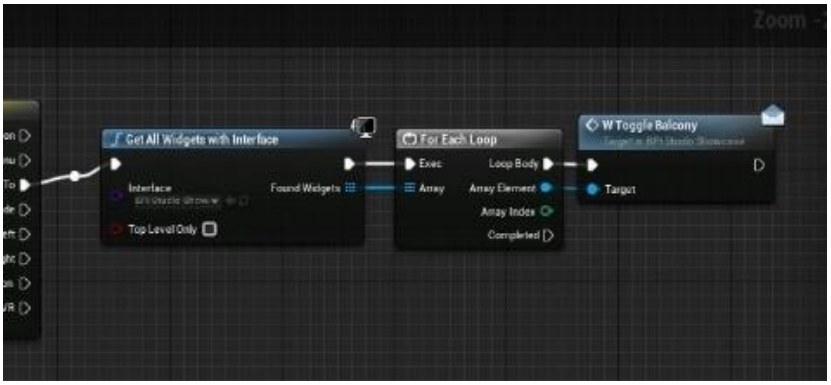
Por defecto, el *W_TopView* muestra los puntos de interés de la planta baja -o *ground*-. El resto de los puntos de interés de la planta alta, o *balconyPoints*, están inactivos y ocultos al inicio. Dentro del *Graph* de *W_MapPoint* hay que seguir la siguiente lógica para establecer el valor de *bIsActive*. Está lógica está en el *Event PreConstruct* de *W_MapPoint*, donde se ha creado una función llamada "*ToggleVisibilityOnActive*" para englobar parte de la lógica.



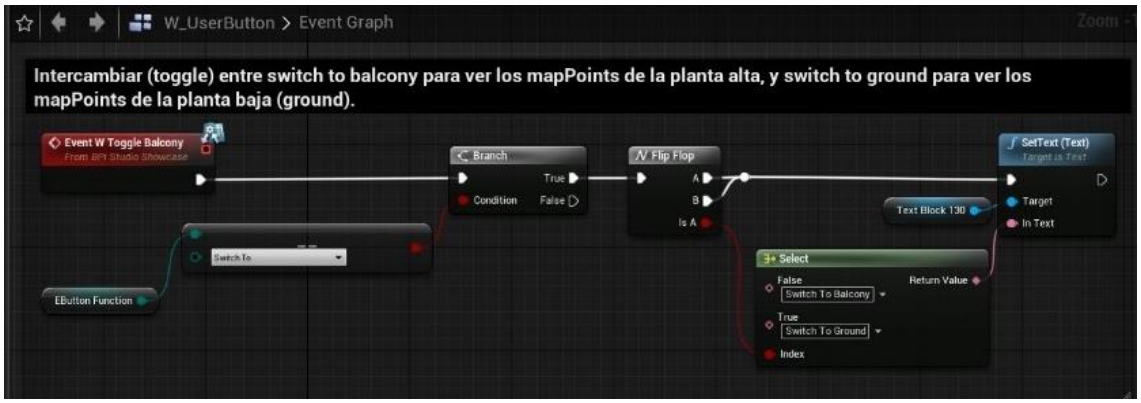
De esta forma, cualquier punto de interés del mapa en el *W_TopMode* que tenga la variable *bIsBalcony* en *true*, debería estar oculto.

Luego hubo que establecer la primera acción del *Button*. Para ello en *Wc_MainButton* hay que añadir un nodo *Switch on* desde la variable *EButtonFunction*. Desde el pin *Switch To* se crea un nodo *Get All Widgets with Interface* para obtener todos los *widgets* con la interfaz *Bpi_StudioShowcase*. Luego se añade una llamada usando *For Each Loop* a *W_ToggleBalcony*. Básicamente según la función del botón, se muestran unos puntos del mapa u otros. Para ello, hay que incluir en el array los actores con una altura (*height*) igual o inferior a los del suelo (*ground*), o incluir además los que tengan una altura igual o inferior a los puntos que están en la segunda planta (*balcony*).



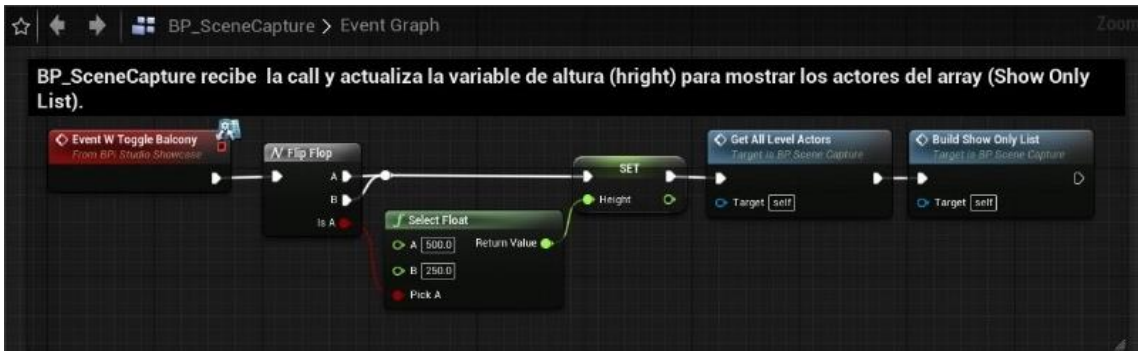


Hay que recordar que para implementar una interfaz en un *blueprint* solo hay que añadirla en *Class Settings*. Dentro de *W_UserButton* se añade el evento *W_ToggleBalcony* además de en otros *widgets* donde se necesita actualizar algo cuando se presiona el botón. Los puntos de interés del mapa necesitan ocultarse o mostrarse. Por ello dentro de *W_MapPoint*, se añade la lógica anterior. Por otro lado, *W_UserButton* necesita intercambiar (*toggle*) entre *Switch To Balcony* para ver los puntos de interés del mapa de la planta alta (*balcony*), y *switch to ground* para ver los puntos de interés del mapa de la planta baja (*ground*).



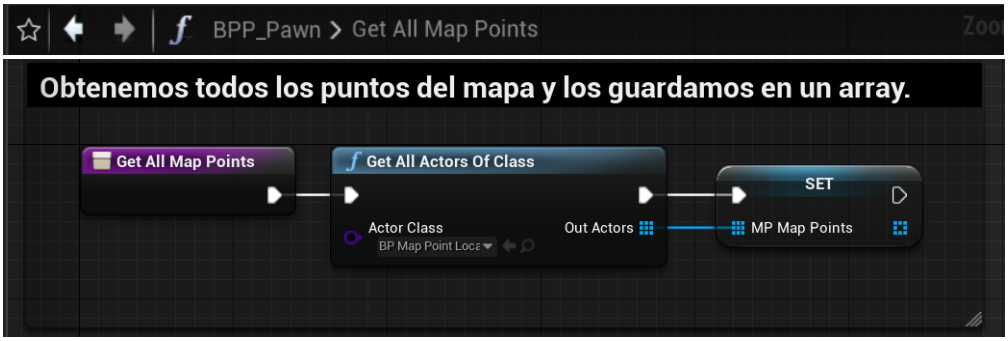
Además, *W_TopMode* necesita convertir una *Widget Interface Call* a una *Actor Interface Call*:

Así es como *BP_SceneCpature* puede recibir la *call* y actualizar la variable *height* para la altura y el *array* de actores *ShowOnlyList.*



De esta forma se colocaron los puntos de interés en el mapa junto con la lógica para poder interactuar con el mismo. Lo siguiente fue desarrollar la lógica para el teletransporte.

Para teletransportar al usuario de una localización a otra se usa un *fade* o desvanecimiento para reducir el *motion sickness*. Al empezar el desarrollo en la clase *BPP_Pawn* las instancias de las clases hijas también pueden compartir esta lógica. Para ello se implementa la interfaz *BPi_StudioShowcase* y se añade el evento *Teleport User To Location*. Se crea una nueva función llamada "*Get All Map Points*" de donde se obtienen todos los actores de la clase *BP_Map_Point_Location*. Se incluyen todos los puntos del mapa de los cuales se quiere obtener su *transform* y se guardan como una variable.

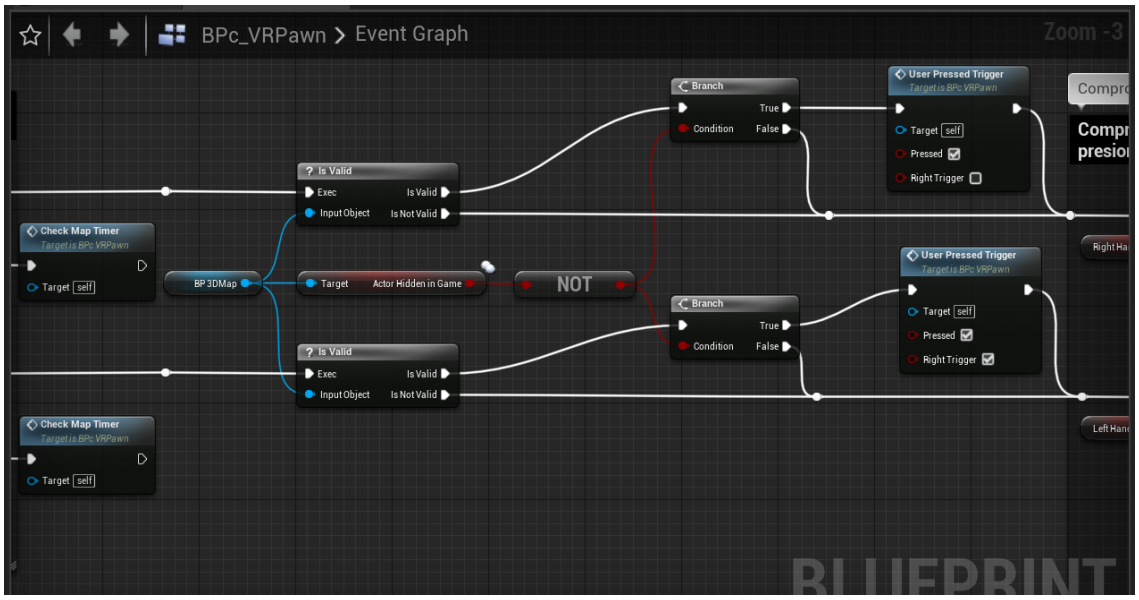


Luego se añade esta función al *Event Begin Play*. Dentro del evento *Teleport User To Location* se comprueba si el valor del array de puntos es igual al valor requerido por la función y se guarda el valor en una variable llamada "*TeleportToTransform*" de tipo *Transform*. Se crea otro evento llamado "*Teleport User To New Map Point*" y se le llama al completarse la ejecución del *For Each Loop*. Este evento se desarrollará dentro del *VR_Pawn*. Dentro de *BPP_VRPawn* se establece la rotación y posición del *VRRoot* a las de la variable *TeleportToTransform*.

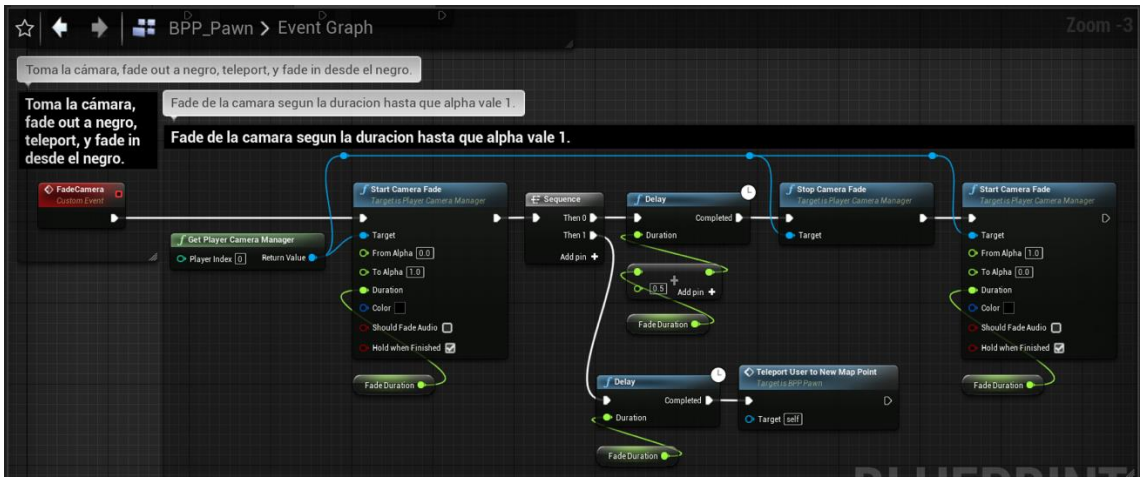
De nuevo en *BPP_Pawn*, se crea un nuevo evento llamado "*Fade Camera*". Dentro de *W_MapPoint* se añade un evento *On Clicked* para llamar al evento *Teleport User To Location* del *BPP_Pawn*.

De nuevo en *BPc_VRPawn* hay que crear un nuevo evento llamado "*User Has Teleported*". Cuando el usuario se haya teletransportado, se pausa el *Map Interaction Timer* y se oculta el mapa.

Ahora hay que modificar la lógica del *spawn* del mapa. Después de comprobar si la variable *BP_3DMap* es válida, se comprueba si el actor mapa es visible. Si lo es, se llama al evento *User Press Trigger*, y si no, se comprueba si el otro *trigger* también está pulsado.

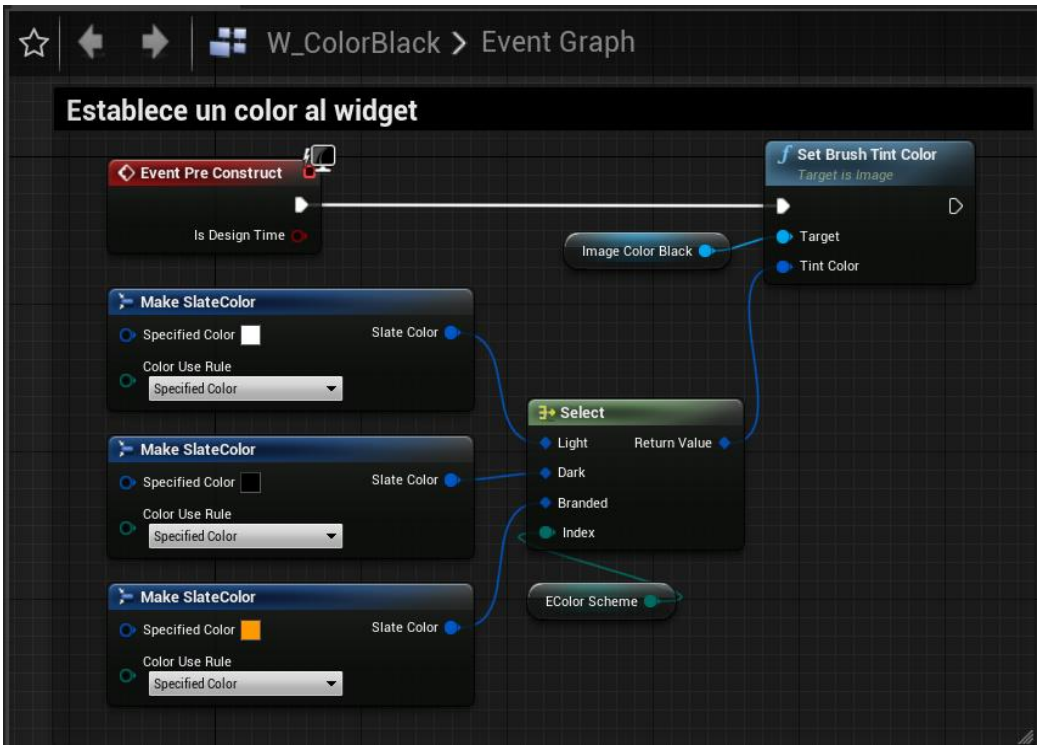


Lo siguiente que se quiere hacer es que el cursor esté oculto y que no aparezca hasta que se interactúa con el mapa. Se abre el *BP_3DMap*, se selecciona *User Cursor* y se desactiva *Visible* dentro de *Rendering*. Lo siguiente es desarrollar la lógica del *fade* de la cámara. Se añade el evento *Fade Camera* al *For Each Loop* del *Teleport User To Location*. Luego dentro del evento *Fade* se utiliza el nodo *Start Camera Fade*. Se utilizará también un nodo *Sequence* para construir la lógica del *fade out* y *fade in* con un *delay* añadido.

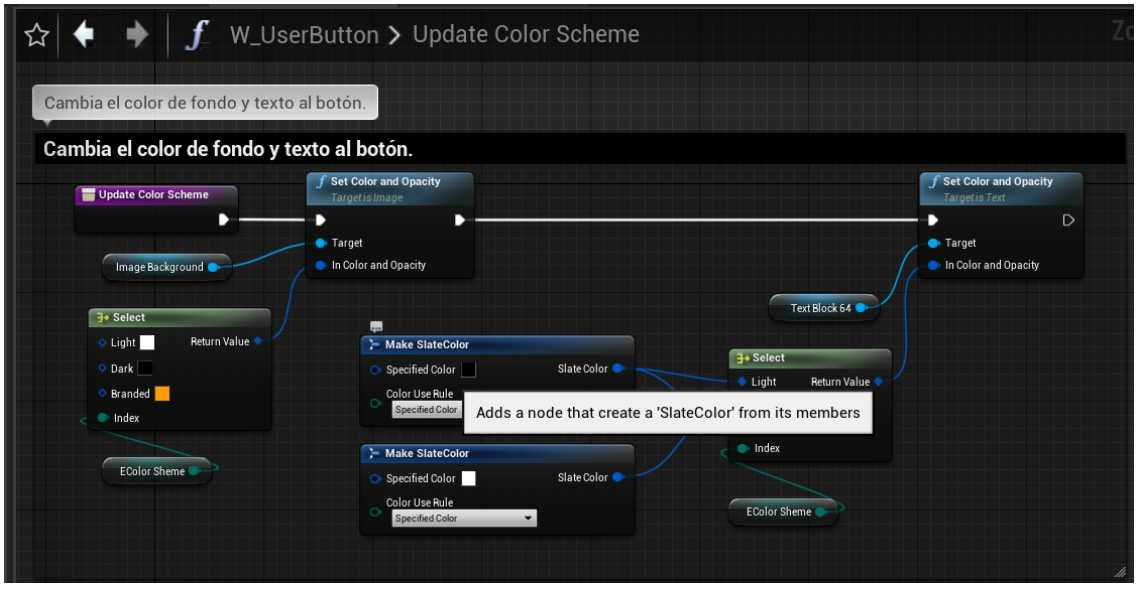


El siguiente objetivo es desarrollar aún más la interfaz de usuario para los widgets apariencia del mapa, planificando el nuevo modo de vista previa y combinándolo con el *Top Mode* ya hecho.

Para ello se crea un nuevo widget llamado "*W_ColorBlock*" que simplemente contendrá una imagen para contrastar con los botones del mapa. Se crea un nuevo *Enum* llamado "*EColorScheme*" con tres modos de colores: *light* (luminoso, blanco), *dark* (oscuro, negro) y *branded* (amarillo).

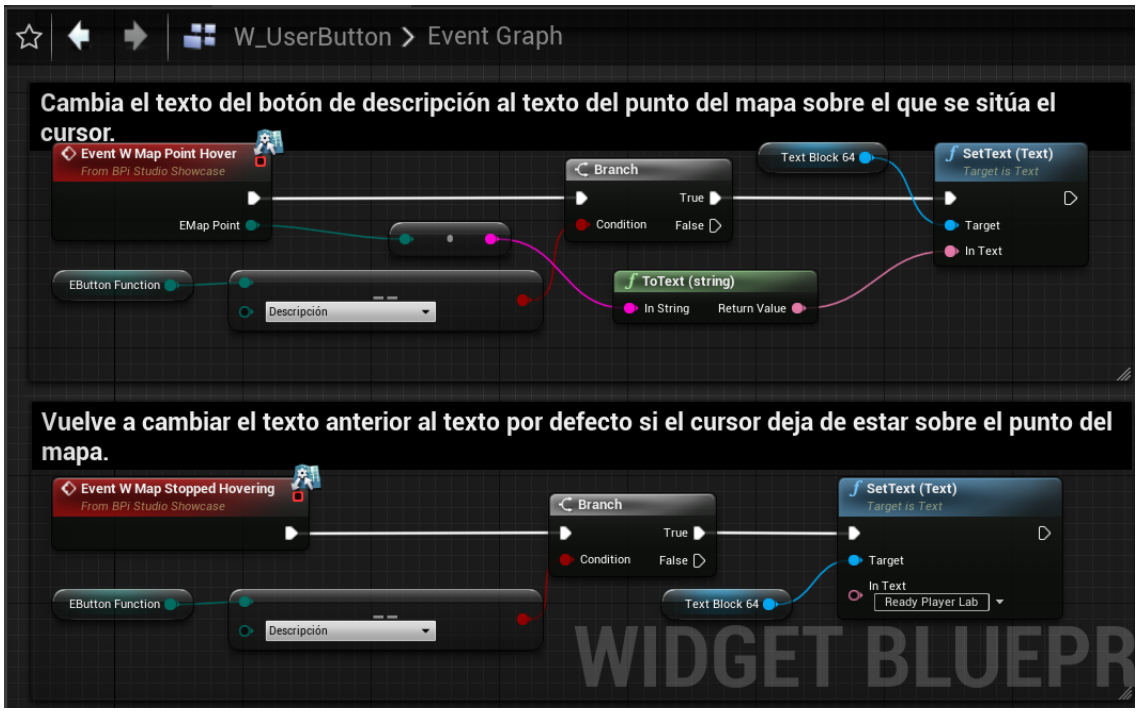


Luego, en el *W_TopMode*, se añade este nuevo widget a cada botón. Se abre el *W_UserButton* y se añade la misma variable. Se crea una nueva función llamada "*Update Color Scheme*" para cambiar el color del fondo y el color del texto del botón a la vez que se cambia el color del *W_ColorBlock*.



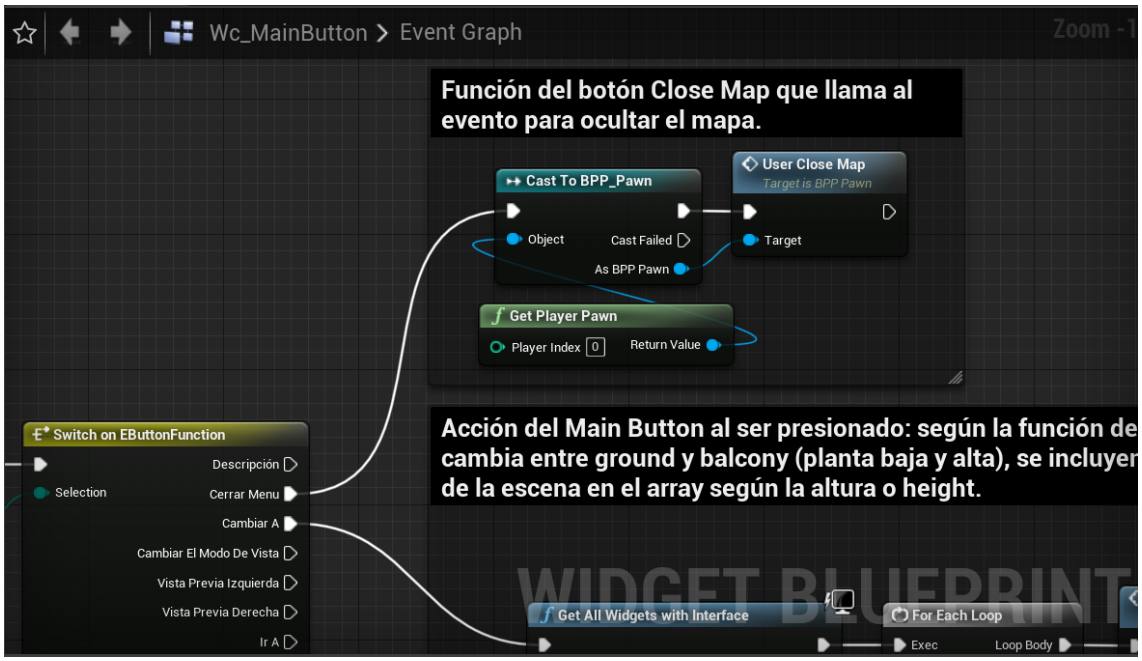
Luego hubo que construir la lógica que permite al usuario saber la localización que se corresponde con cada punto del mapa al pasar con el cursor por encima. Para hacerlo se añade una nueva función a la interfaz para el *Unhover Event* llamado "*W_MapStoppedHovering*". Dentro de *W_MapPoint* se selecciona el botón y se crea un evento *On Hovered* y *On Unhovered*.

Dentro de *W_UserButton* se añaden los eventos desde la interfaz. Esto solo afecta al botón si el valor de su función es *Description*, cambiando entre el texto del botón al texto de localización del punto del mapa y el texto por defecto "*Ready Player Lab*".



Lo siguiente fue establecer la lógica del botón para cerrar el mapa. Para ello, dentro de *BPP_Pawn* se añade un *custom event* llamado "*User Close Map*". Dentro de *BPc_VRPawn* se añade este evento. Cuando es llamado, se pausan ambos *timers* y se oculta el mapa.

Se llamará a este evento desde *Wc_MainButton*. Se añade esta lógica a la función del botón "*Close Menu*".

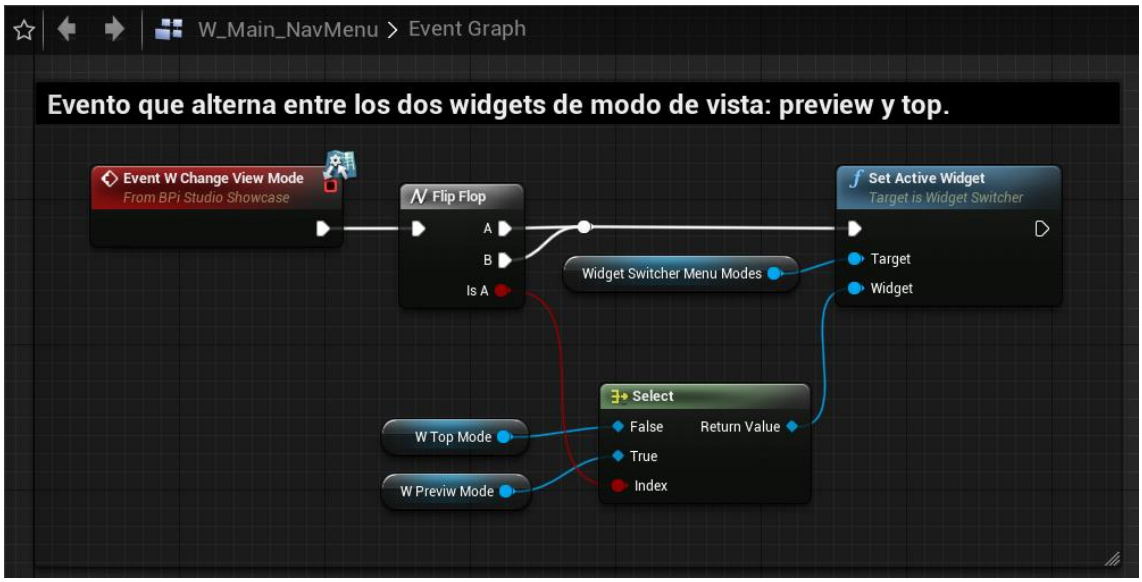


Lo siguiente fue cambiar la textura del cursor. Para ello, dentro de *W_PlayerCursor*, en *Appereance*, se selecciona la imagen png del cursor. Por defecto, los 3D *widgets* tienen el *blend mode* en *mask*. Para mejorar el aspecto hay que establecerlo a *Transparency*, dentro del *BP_3DMap / User Cursor*.

Lo siguiente fue crear el *Preview Mode* o modo de vista previa. Se necesitaron otros dos nuevos *widgets*: "*W_Main_NavMenu*" y "*W_PreviewMode*". Se empezó desarrollando el segundo. Se copia y se pegan los *assets* del *top mode* y se elimina lo que no hace falta, como los

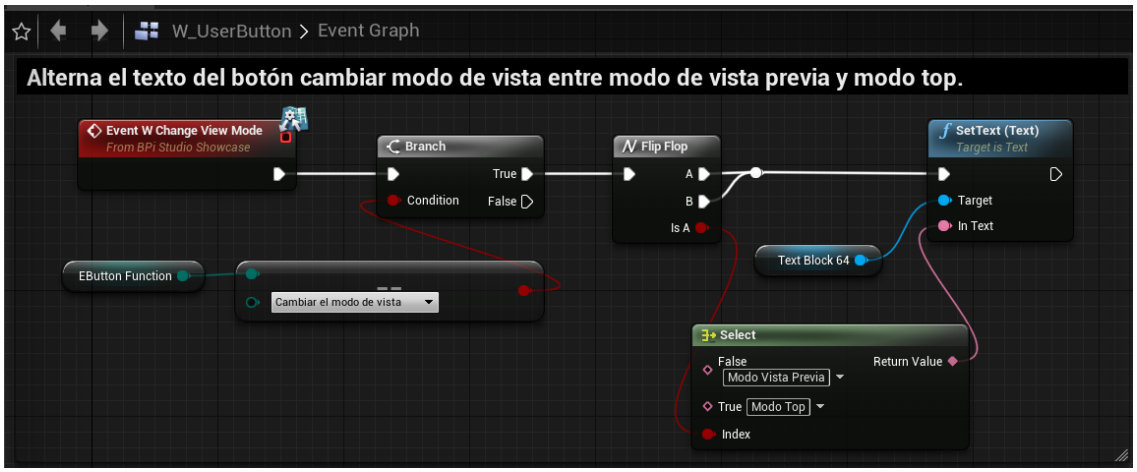
MapPoints. Luego se hace un *wrap* del *Descriptiion Button* y del *Color Block* con un vertical box. Se distribuyen más copias de este botón por el mapa. Se establece a cada botón su *EButtonFunction* correcto. Se establece al gusto el *EColorScheme* de los *Users Buttons*, de tal forma que queden con la apariencia deseada. Se elimina el botón *SwitchToBalcony*, ya que no hace falta aquí. Se añaden otros dos botones con la funcionalidad de *PreviewModeRight* y *PreviewModeLeft,* los que se encargarán de viajar entre los *EMapPoints* del escenario en modo vista previa.

Luego de desarrolló el *W_MainNavMenu*. Para ello, se añade un *WidgetSwitcher* llamado *WidgetSwitcher_MenuModes* que contiene ambos widgets, el *top mode* – que es el que aparece por defecto– y el *preview mode*. Dentro del *Graph*, el evento *WChangeViewMode* se encarga de intercambiar ambos *widgets* de modo de vista.

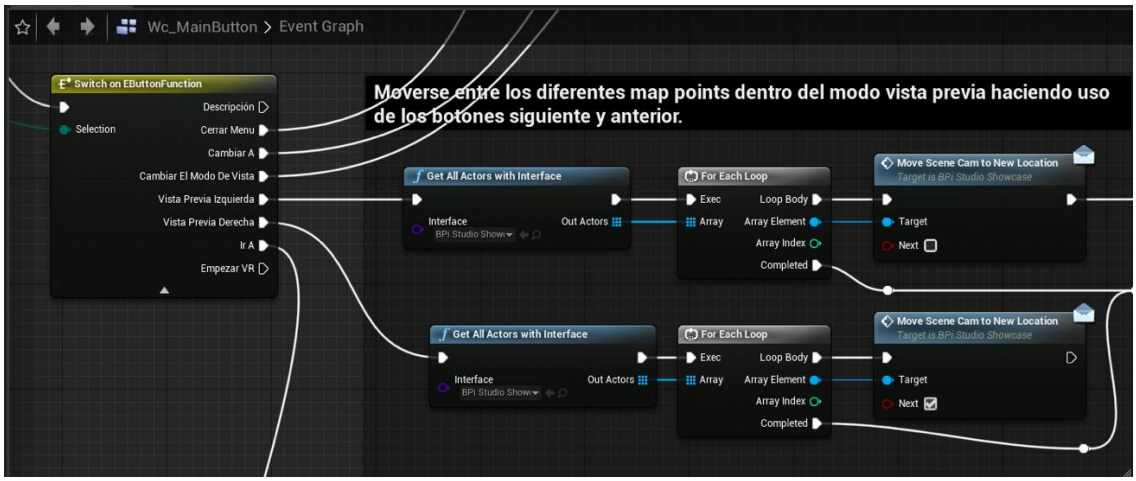


Luego hubo que volver al *BP_3DMap* y cambiar el *widget* que hace *spawn*. En lugar de *W_TopMode*, se pone *W_Main_NavMenu*. También hay que añadir la *interface call* al *reusable Main Button*.

Lo siguiente fue alternar el texto entre *PreviewMode* o *TopMode* dentro del botón según el modo de vista en el que se encuentre el usuario, tal y como se hizo anteriormente con otros botones.

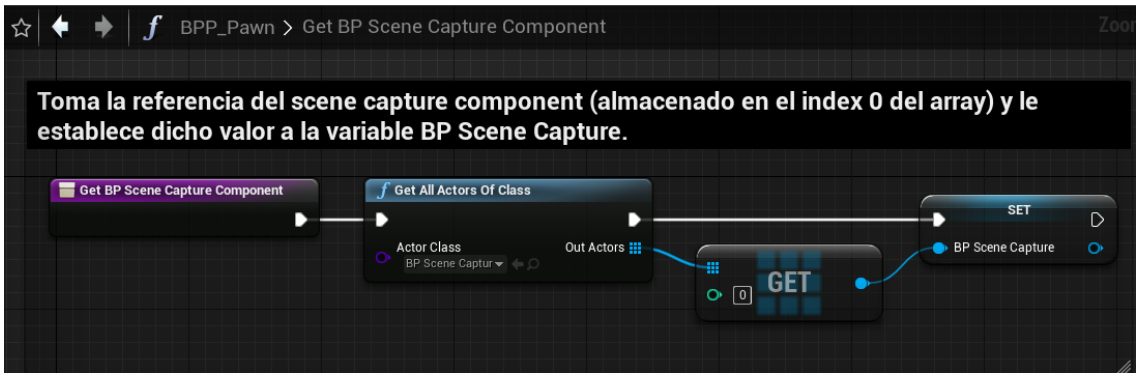


Lo siguiente fue añadir funcionalidad a los botones *Left* y *Right* para moverse por los *MapPoints* de la escena. Para ello, dentro de *BPi_StudioShowcase*, en la función *Move Scene Cam To New Location*, se añade un *bool input* llamado *Next*. Estos botones no intercambiarán entre widgets, solo necesitan comunicarse con los actores. Por lo que dentro de *Wc_MainButton* añadimos la funcionalidad, donde si *Next* es *true* se moverá a la derecha y viceversa.



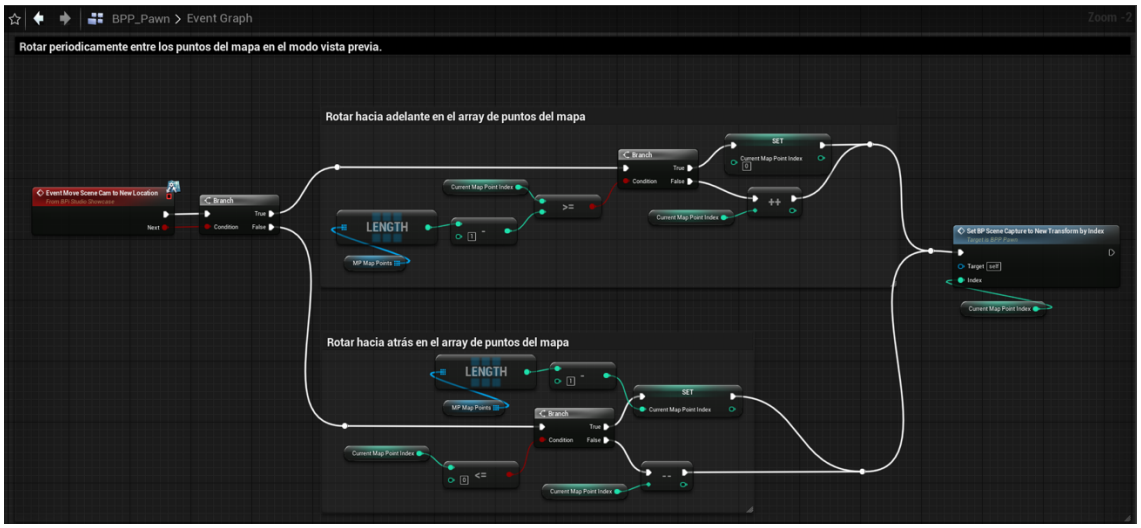
Lo siguiente fue comunicarse con el *Parent Pawn* el cual a su vez se comunica con *BP_SceneCapture*. Antes seguir con el desarrollo de la función *MoveSceneCamToNewLocation* se creó un nuevo evento en *BP_SceneCapture* llamado *SendCamToNewLocation* con un *input* de tipo *Transform* llamado *NewTransform*. Se arrastra el *SceneCaptureCoponent2D* y se añade un nodo *SetTransform*.

Luego se desarrolló una manera mediante la cual hacer posible que el *pawn* pudiera llamar a este evento. Para ello, dentro de *BPP_Pawn* se añade una nueva función para obtener una referencia del *BP_SceneCapture* llamada *GetBPSceneCapture*, la cual se añade al *Event Begin Play*. Dentro de esta función se obtienen todos los actores de su clase, que como solo hay uno, simplemente se obtiene el índice cero del *array* y se promociona a una variable.

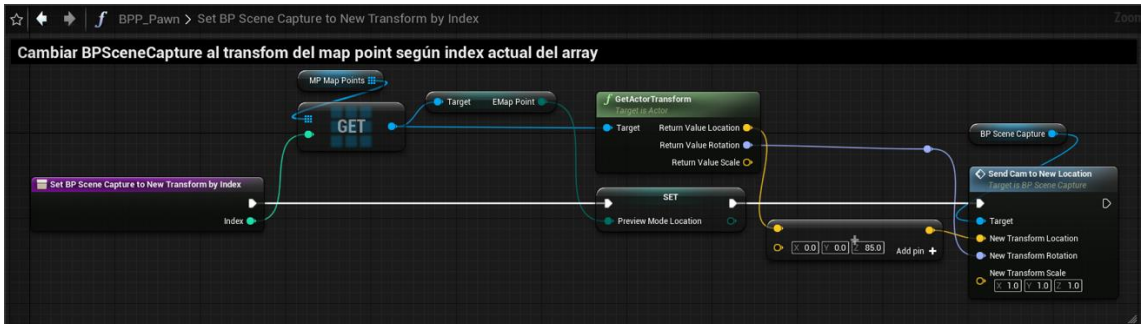


Se crea también la interface *event MoveSceneCamToNewLocation*. Cuando se llama a este evento se recorre el *array* de puntos del mapa y se obtiene el *transform* del punto anterior o siguiente. Se envía este *transform* al *BP_SceneCapture* para teletransportar al usuario. Lo primero es preguntar si la variable *Next* es *true* o *false* para rotar hacia delante o hacia atrás entre los puntos del mapa, aumentando o disminuyendo el índice del array. Si es *true*, se pregunta si el *CurrentMapPointIndexArray* es mayor o igual a la longitud del *array* menos uno. Si lo es, se establece el valor a cero, y si no lo es se añade uno al valor del índice del *array*. Sucede

exactamente lo mismo, pero al revés, para el otro caso. Si *Next* es false, se pregunta si el *CurrentMapPointIndexArray* es menor o igual a la longitud del *array*. Si lo es, se establece el valor del índice al de la longitud del array menos uno, y si no lo es se resta uno al valor del índice.



Después de establecer el valor del índice uno arriba o uno abajo, se obtiene el valor del índice actual del *array*, se obtiene el *transform* del actor –y su desplazamiento en z para que la cámara no esté en el suelo– y se añaden estos datos al evento *SendCamToNewLocation* dentro de *BP_SceneCapture*.

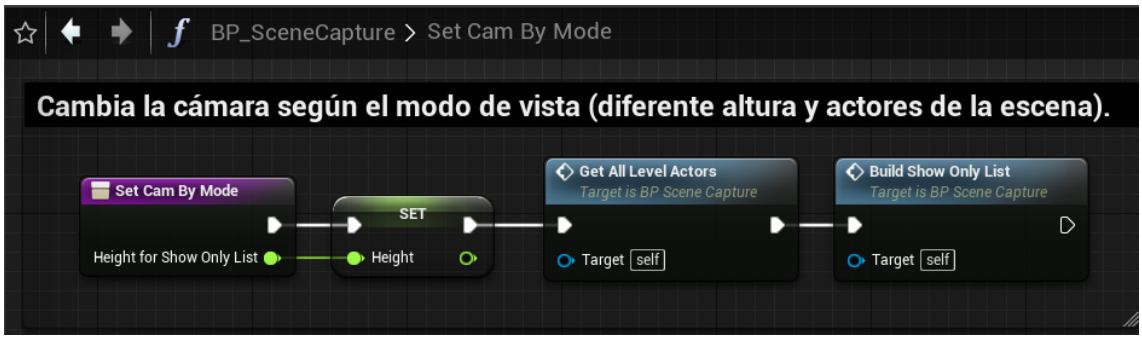


Tener múltiples formas de navegar por la escena le brinda al usuario elegir la que mejor se ajusta a su gusto. Una vez hecho esto, lo siguiente fue añadir movimiento entre las diferentes localizaciones del mapa en el modo de vista previa. Esto se hizo añadiendo animaciones de cámara al *scene capture component*, otorgando más dinamismo al *preview mode*. Para ello, en *BP_SceneCapture*, se guarda el *transform* actual para saber la localización a la que regresar cuando se regrese al modo de vista *top*.

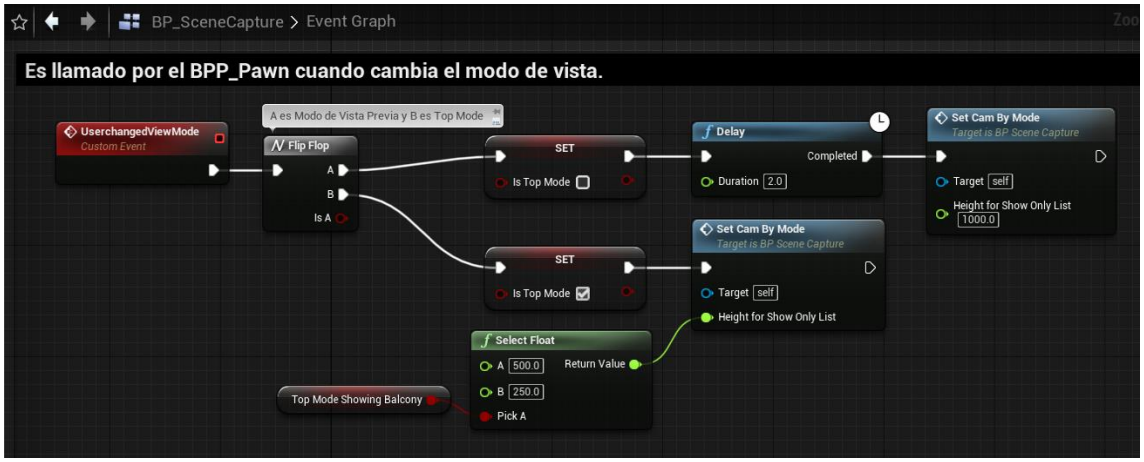
Luego, en *BPP_Pawn*, se crea un evento llamado *UserChangedWiewMode*. Antes de trabajar en él, se establece la *call* en el *reusable button*. En *Wc_MainButton*, en el *ChangedViewmode*, después del *completed execution* del *ForEachLoop* se hace un *cast* al *BPP_Pawn* usando una referencia al *player pawn* y se llama a este nuevo evento *UserChangedWiewmode.*

Luego en *BP_SceneCapture* se añade un nuevo evento llamado *UserChangedWiewMode* y se llama a este evento desde el *BPP_Pawn*. Dentro de *BP_SceneCapture* se necesita saber si el balcón está activo – si los puntos del mapa de la planta superior son visibles– para así poder volver a configurarlo cuando se vuelva al *top mode* desde el *preview mode*. Para ello se crea un nuevo *bool* y se establece su valor desde evento *WToggleBalcony*. El nuevo *UserChangeViewmode* hace lo mismo, pero para diferentes valores de altura.

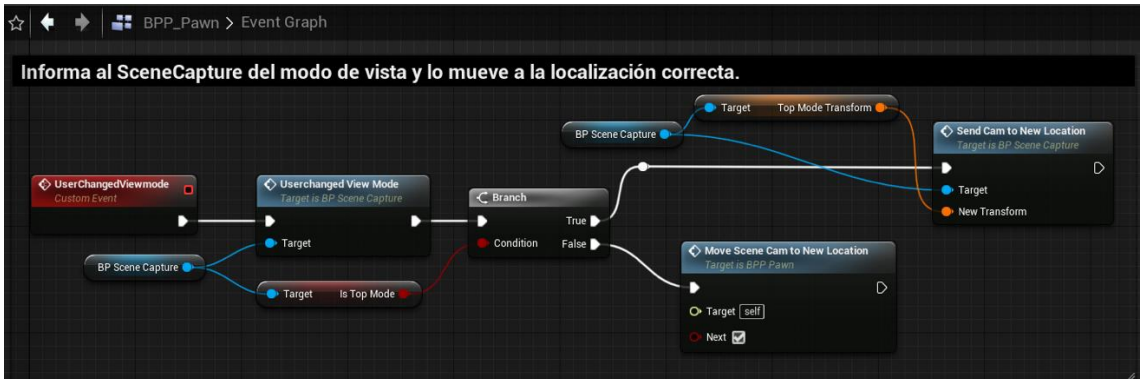
Se colapsa las funciones *GetAllLevelActors* y *BuildShowOnlyList* en una nueva función llamada *SetCamByMode*.



Se necesita también otra variable para saber en qué modo de vista se está actualmente durante el juego. Se crea una variable booleana llamada *IsTopMode*.

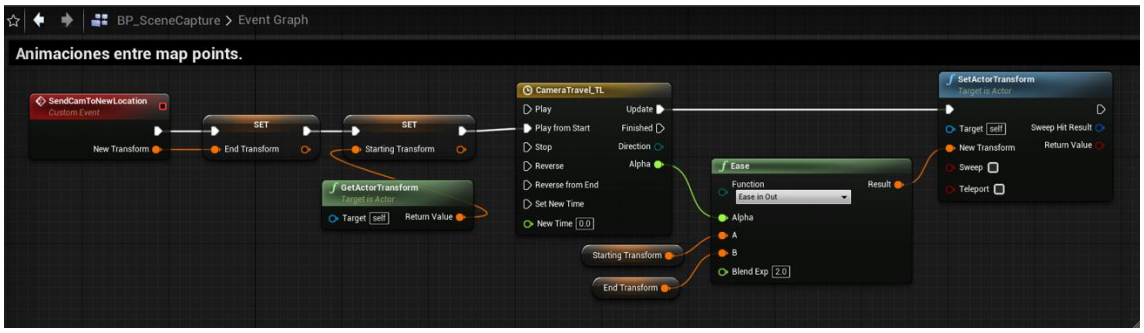


De nuevo dentro de *BPP_Pawn*, en *UserChangedViewmode,* se coloca un *brunch* al final. Si no se está en el *top mode* se llama a la función *MoveSceneCamToNewLocation*. De esta manera cuando el modo de vista cambia a modo de vista previa le comunicará a la cámara moverse al siguiente punto. Por otro lado, si se vuelve al *top mode* se regresa al *transform* que se guardó anteriormente.



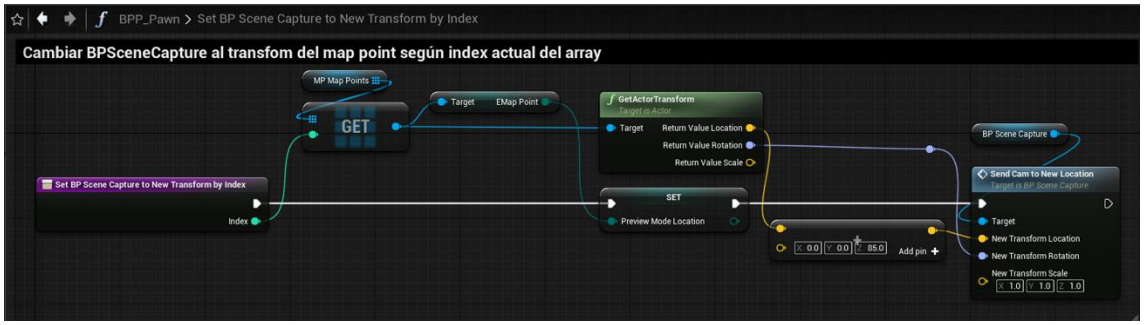
Lo siguiente fue animar la cámara. Para ello, dentro de *BP_SceneCapture*, se crean dos nuevas variables de tipo *transform* llamadas *StartingTransform* y *EndTransform*. La idea es interpolar el *transform* de la cámara desde el actual hasta el que se requiere usando un *Timeline*.

Se crea el *track* y se establece el nuevo valor del *transform* haciendo uso de un nodo *Ease* con la función *Ease In out* para un movimiento suavizado.

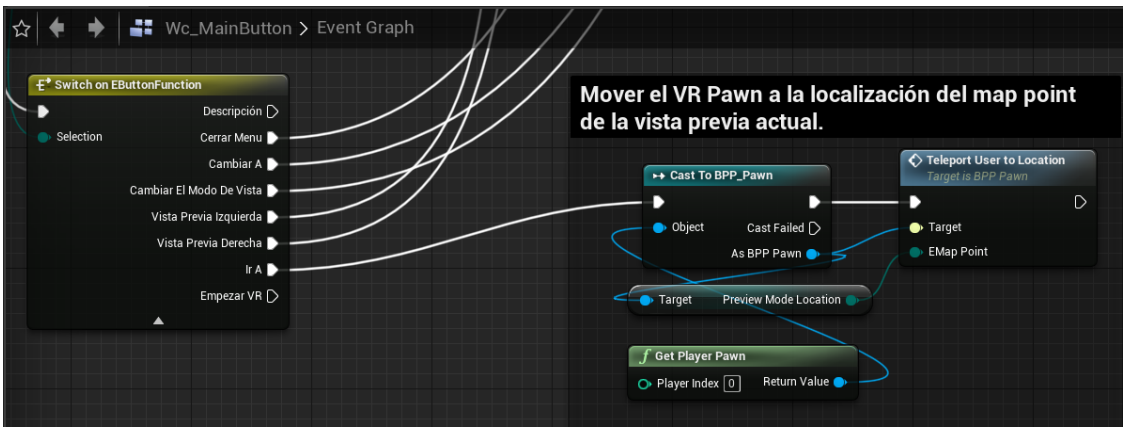


Aquí surgió un pequeño problema, y era que el techo del escenario era visible desde la animación *TopMode* a *PreviewMode* y viceversa. Para resolverlo, en *BP_SceneCapture*, dentro del evento *UserchangedViewMode* se añade un *delay* de 2 segundos para cambiar la altura.

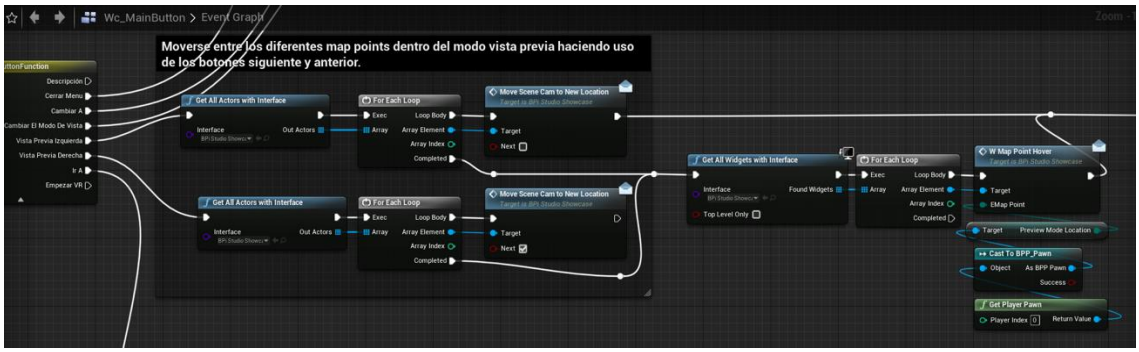
Lo siguiente fue desarrollar la lógica del botón *GoToLocation*. Para ello se necesita saber la posición del punto del mapa actual en el que se encuentra el usuario. Se crea una variable de tipo *EMapPoint* dentro de *BPP_Pawn* llamada *PreviewModeLocation*. En la función *SetBPSceneCaptureToNewTransformByIndex*, cuando se obtiene el índice del array también se obtiene el *EMap_Point* y se establece el valor a la nueva variable.



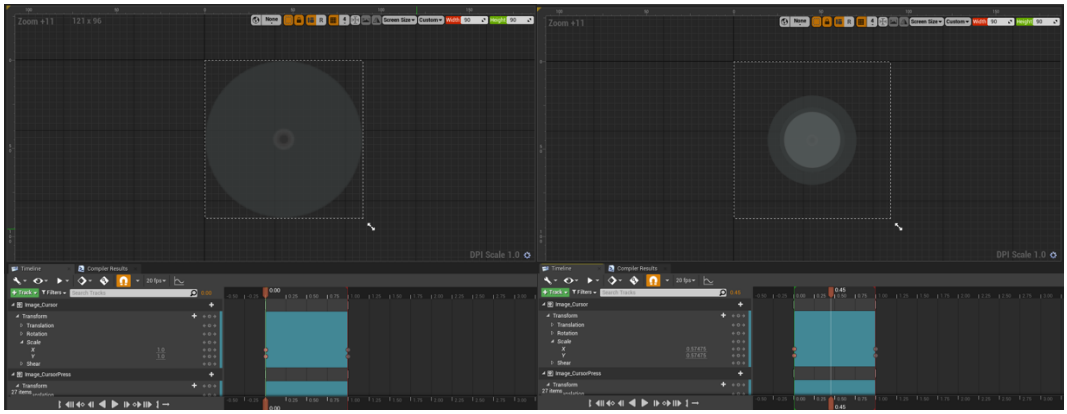
Luego, en *Wc_MainButton*, se añade funcionalidad al botón *GoToLocation*. Se hace se obtiene la variable *EMap_Point del BPP_Pawn* y se llama a la función *TeleportUserToLocation*.



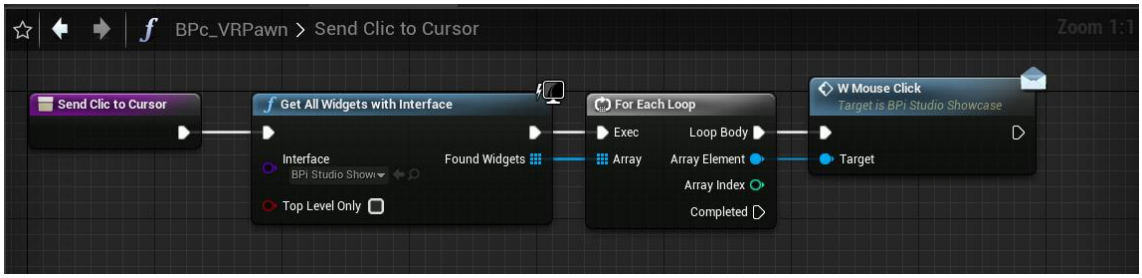
Luego, tal y como se desarrolló la lógica para los eventos *hover* sobre cambiar el texto descriptivo de los botones, se hizo lo mismo para que cada punto del mapa mostrara su localización. Para ello, dentro de *Wc_MainButton,* al final de la lógica de los botones vista previa izquierda y derecha – completados ambos *ForEachLoop*– se añade un nodo *GetAllWidgetsWith Interface* para obtener todos los *widgets* con la interfaz *BPi_StudioShowcase*. Estos se guardan en un *array* que se recorre con un nuevo *ForEachLoop* y se llama al evento WMapPointHover. Por último, se conecta la variable *EMapPointLocation* del *BPP_Pawn* – con *PureCast*– a dicha función.



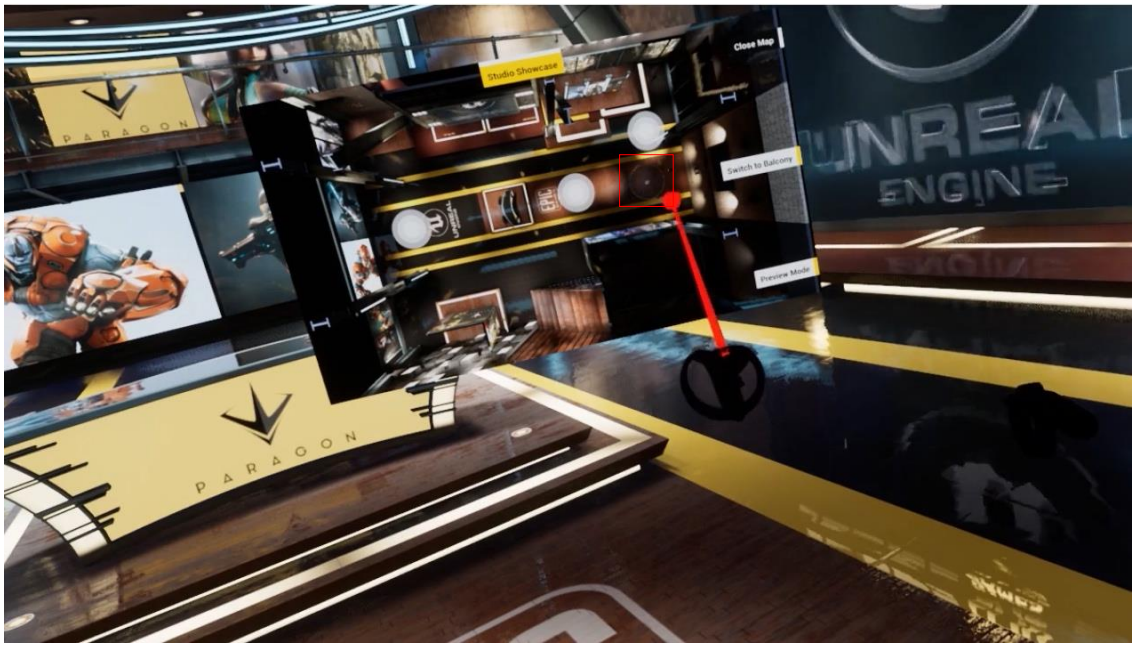
Lo siguiente fue añadir una animación al cursor para cuando el usuario haga clic en el mapa. Para ello, en *W_PlayerCursor* se añade una animación llamada *clic* donde se modifica las escalas de las imágenes entre 0 y 1. Para reproducir la animación se necesita añadir el evento *WMouseClic* junto con el nodo *PlayAnimation* en el *Event Graph*.



En *BPc_VRPawn* se crea una nueva función llamada *SendClicToCursor*. Se obtienen todos los *widgets* que implementan la interfaz y se llama al evento *WMouseClic*. Luego se añade esta función al final de la función *UserPressedTrigger*.



Aquí apareció un problema. Si se activa la línea *debug* en el *widget interaction component* en *BP_3DMap* el cursor aparece desplazado del centro de la línea de interacción.

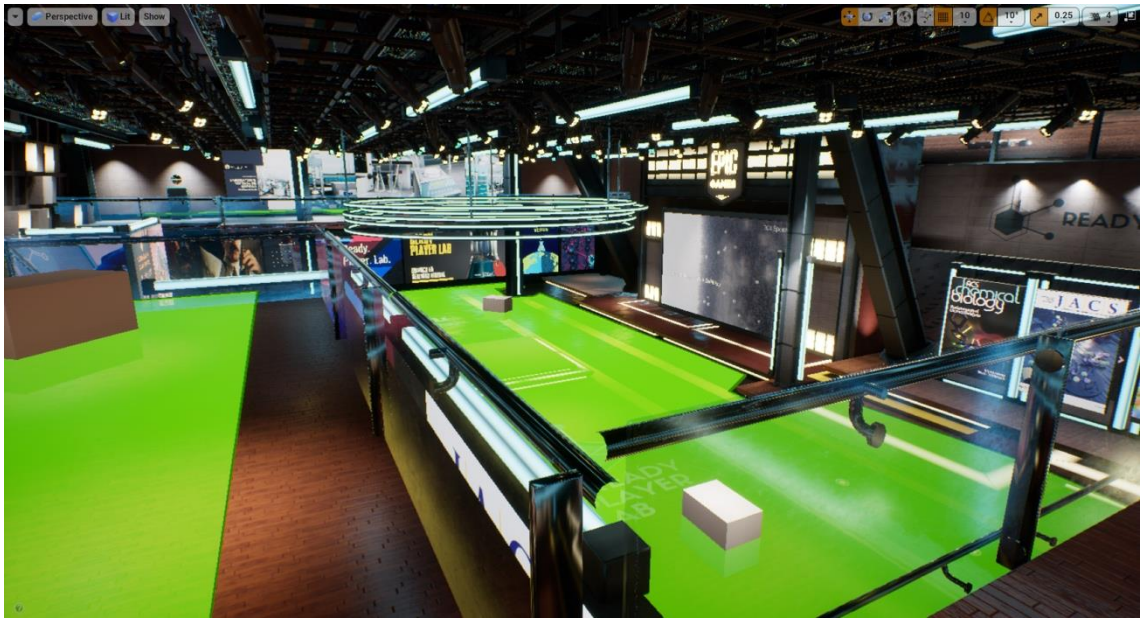


Para arreglarlo se creó un nuevo *scene component* llamado *CursorRoot* y se colocó el *UserCursor* dentro del mismo. Luego se movió el cursor manualmente a donde debería estar [25].

Hasta aquí de desarrolla con mucho detalle las etapas relacionadas con sentar las bases del proyecto y la creación de un mapa de navegación 3D. En las siguientes etapas la descripción del desarrollo será más generalizada y enfocada en la idea que se quiere desarrollar, puesto que no se pretende extender este documento más de lo necesario.

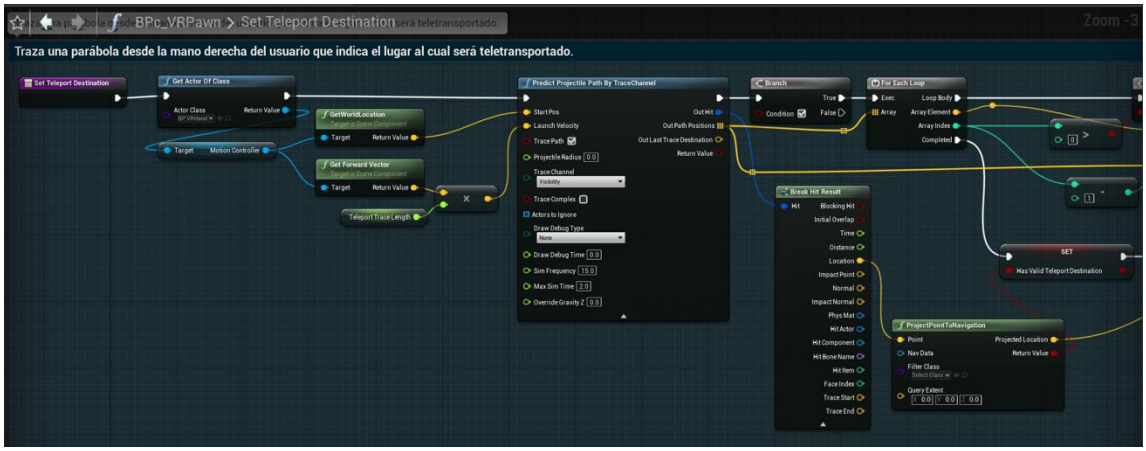
II.2 Sistema de locomoción por teletransporte.

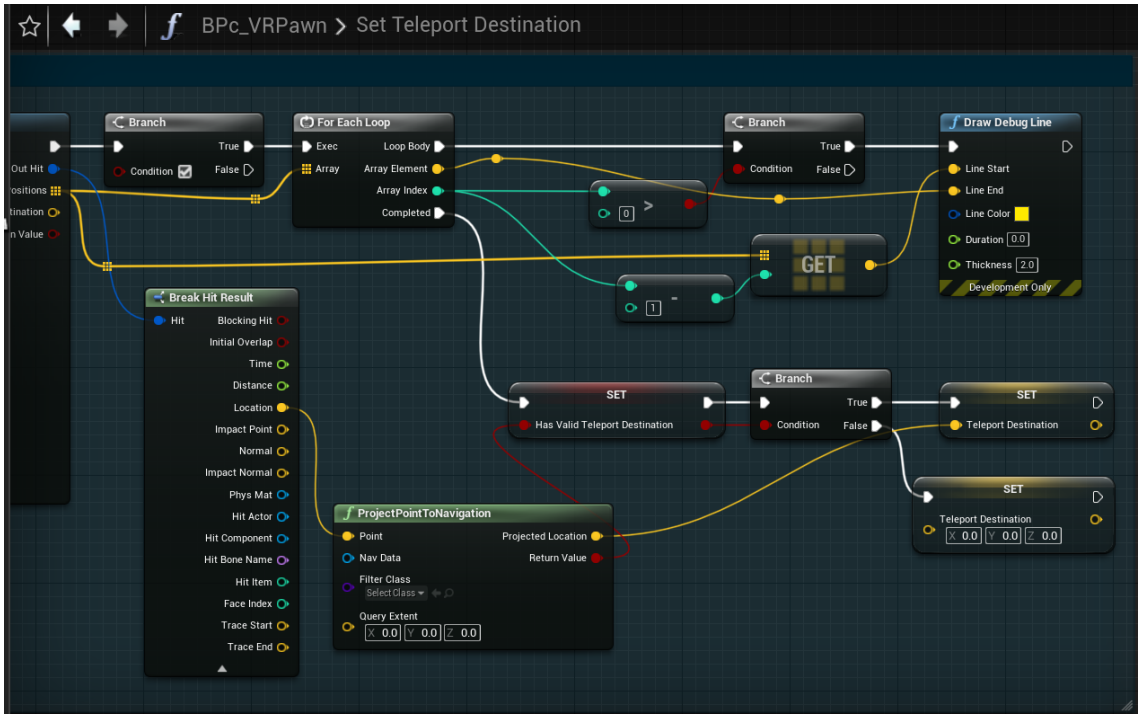
En este apartado se desarrolla un sistema de navegación por la escena alternativo al mapa virtual 3D. Como se mencionó en la introducción, uno de los sistemas de locomoción para la VR más comunes es el teletransporte. Lo primero que se necesita para ello hacer uso de la *navigation mesh* o *navmesh*. Este sistema proporciona una serie de superficies en el nivel que le dicen al motor a qué lugares está permitido teletransportarse. Se añade a la escena a través de *NavMesh Bounds Volumes*. La tecla *P* permite visualizar estos actores en la escena. Para excluir áreas navegación de la *navmesh* se utilizan *NavModifier Volumes*. Al añadir estos actores se crea por defecto otro actor llamado *ReCastNavMesh-Default*. Una de las propiedades más relevantes es *Agent Radius*, que controla la distancia a la cual la *navmesh* se acerca a los obstáculos. Se encontró un valor óptimo es 35. En este proyecto se trabajará dinámicamente con la *navmesh*, activándola y desactivándola a placer. Para ello dentro *Project Settings / Navigation Mesh / Runtime / Runtime Generation* se selecciona el modo *Dynamic*.



Lo siguiente fue trazar una línea desde el *motion controller* izquierdo con el objetivo de que el usuario pueda señalar hacia dónde quiere ir. Para ello, dentro del *BPc_VRPawn* se crea una función llamada *SetTeleportDestination*. Esta utiliza el *eventTick* para trazar una línea *debug* desde la localización del *controller* derecho hasta la *navmesh*, donde se proyecta una esfera

debug. Trazar líneas rectas presenta un problema, y es que el usuario no puede alcanzar lugares que están más altos de su punto de visión. Para solucionarlo hubo que sustituir el trazo recto por uno parabólico usando el nodo *Predict Projectile Path By TraceChannel*. Este nodo devuelve un array de puntos que describen el trazo de una parábola. Se usaron esos puntos para dibujar un indicador de posición que le indica al usuario el lugar donde será teletransportado. El input usado para ejecutar esta acción es el *thumbstick* izquierdo del *controller*.

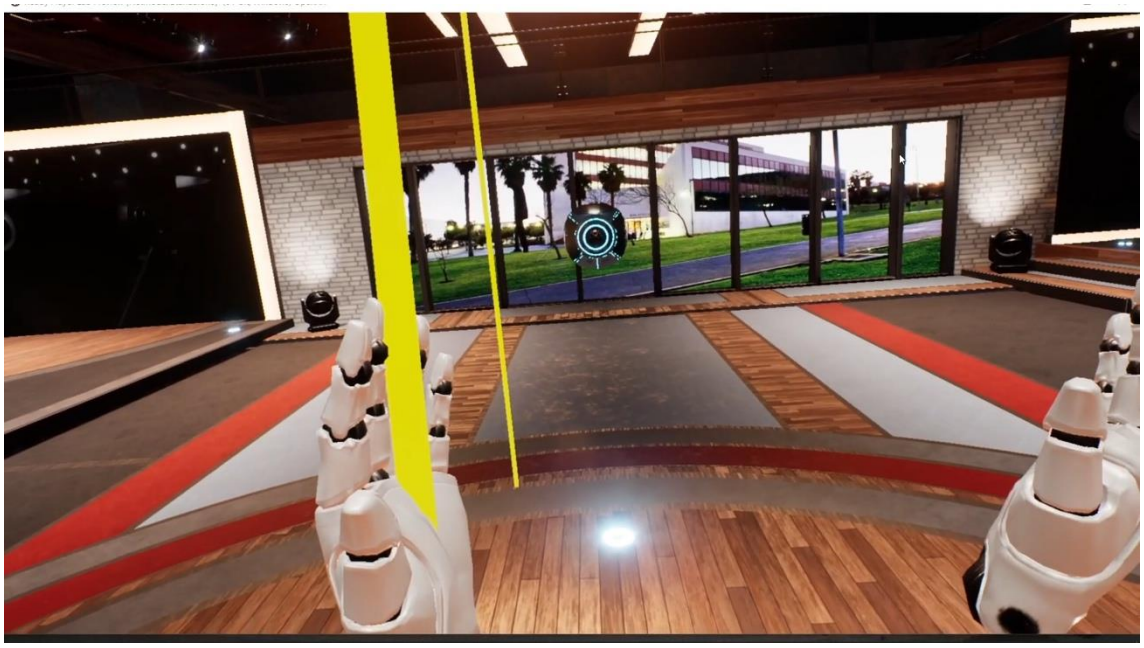




Como inciso, no hay que olvidar que los nodos que trazan líneas *debug* solo son visibles en las *builds debug* y *development*, no en las *shipping*. La manera actual de dibujar este tiro parabólico es usando una colección de puntos *debug* en el *out path positions*. En lugar de eso se podrían guardar estas posiciones en una variable y luego usar un *spline component* en el *pawn*

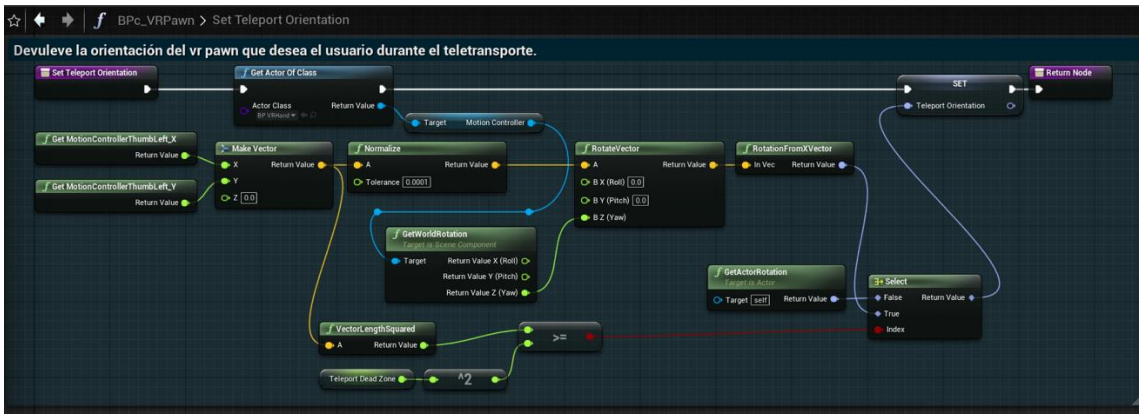
y emparejarlo con una *mesh*, o bien importar desde la *VR Template*. Esto podría realizarse en una futura ampliación del proyecto.

Para continuar hay que guardar la localización que obtenemos del *trace* en una variable de tipo vector llamada *TeleportDest*. También se hace uso de una variable booleana llamada *bHasValidTeleportDest* para preguntar si la localización es válida antes de almacenarla. Esto se hace en cada *frame* mediante el evento *tick* durante el *teleport*. Luego hay que crear el evento *input action* en *BPc_VRPawn*. Esto se mapea en el eje Y del *thumbstick* izquierdo como *action input*. También hay que desarrollar la manera en la que el usuario pueda especificar la dirección hacia la que quiere orientarse. Esto se mapea en los ejes X e Y del *thumbstick* derecho como *axis inputs*. Hay que recordar que *Unreal* maneja el sistema de *inputs* mediante dos tipos de mapeados, *action mappings* y *axis mappings*. Los primeros son eventos discretos, mientras que los segundos proporcionan información continua sobre un *input* analógico como como un *joystick* o un *trackpad*.



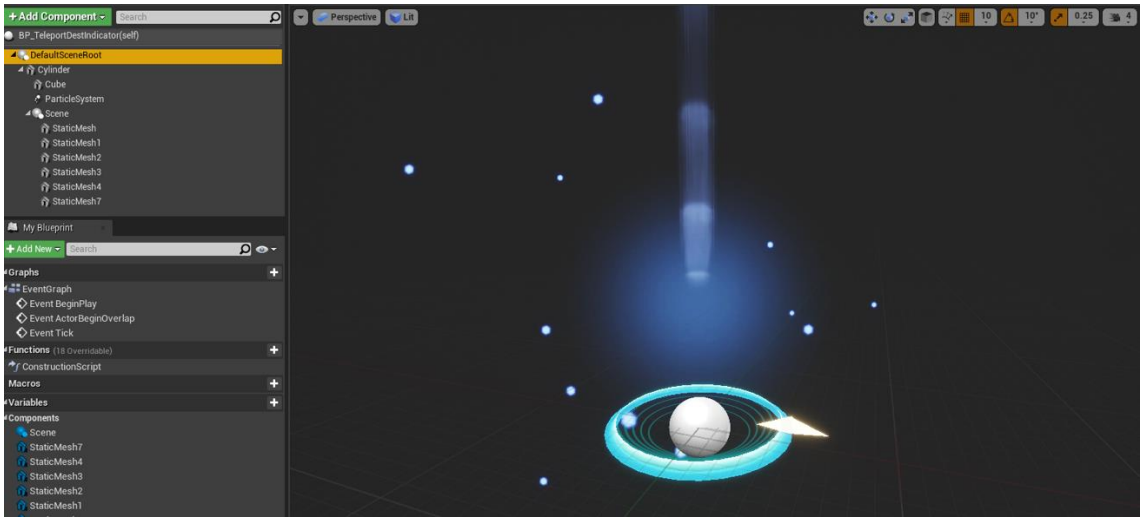
Para orientar al usuario en la teletransportación se hace uso de una función llamada *SetTeleportOrientation*. A partir de los valores de los inputs creados *MotionControllerThumbLeft* X e Y se crea un vector que luego hay que normalizar, es decir, escalarlo a una magnitud de 1 –vector unitario–. Muchas operaciones matemáticas con vectores de magnitud arbitrarias devuelven resultados incorrectos, por eso hay que usar vectores unitarios. Luego hay que rotar ese vector hacia donde el usuario tiene intención de dirigirse usando un nodo *RotateVector*. El resultado se guarda en una variable de tipo *rotator* llamada *TeleportOrientation*, que es lo que

devuelve esta función. Esta función es añadida finalmente en el *event tick* junto con *SetTeleportDestination*.

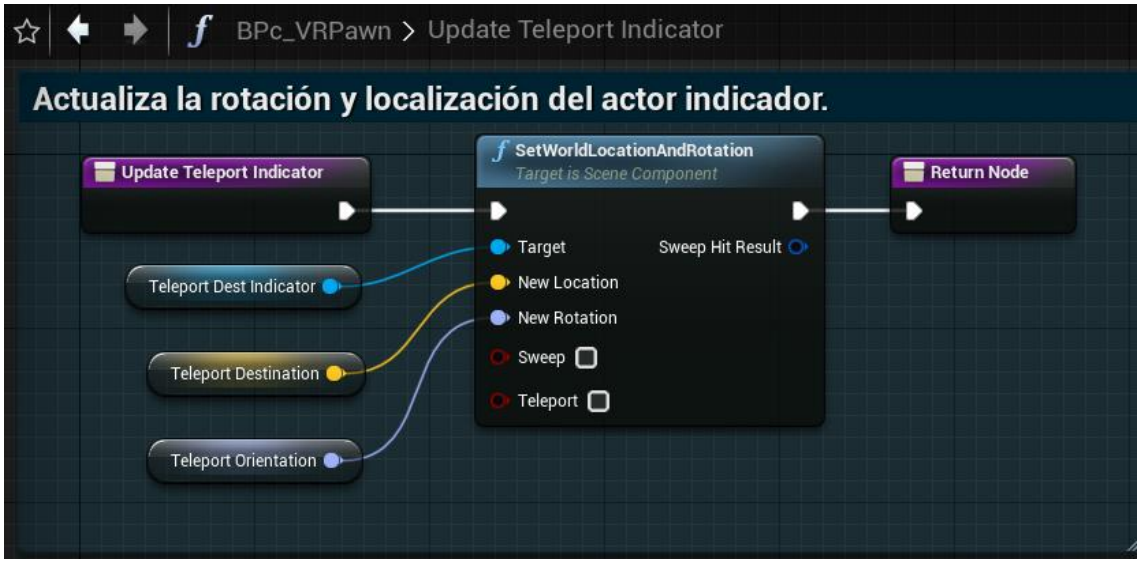


La función donde se ejecuta el teletransporte teniendo en cuenta localización y rotación quedaría así:

Lo siguiente fue crear un indicador de posición y rotación para el teletransporte. Para ello se creó el *BP_TeleportDestIndicator*, un *blueprint* de tipo actor. Consta de algunas *static meshes* con efectos de partículas, haciendo uso de materiales emisivos. Su objetivo es similar a la esfera *debug* que muestra la localización del teletransporte. También lleva consigo una malla en forma de flecha para que muestre al usuario la rotación indicada.

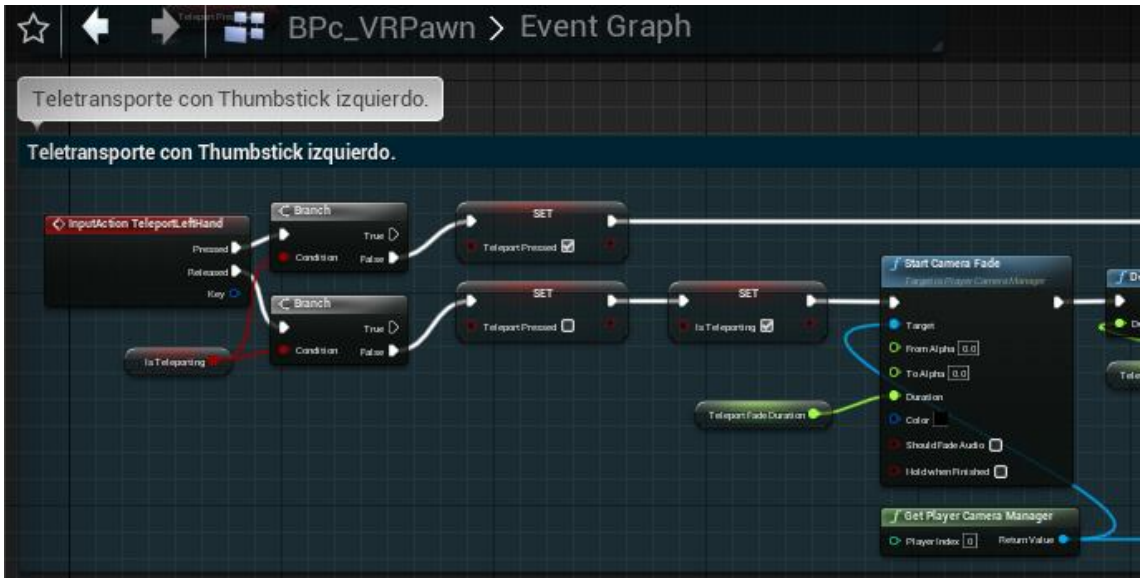


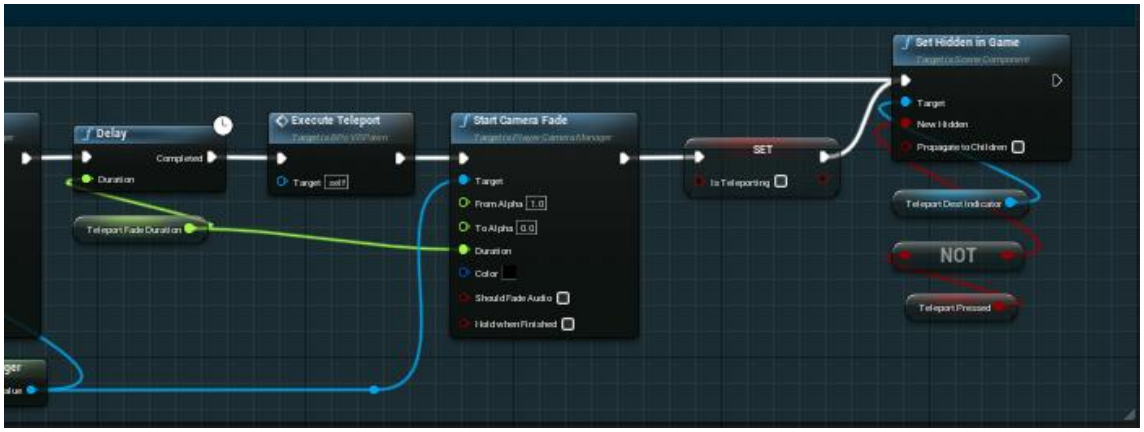
Para que funcionara hubo que añadir este indicador al *BPc_VRPawn.* Para ello, hay que añadirle el actor indicador como un *ChildActorComponent*. Luego se crea una función llamada *UpdateTeleportIndicator*.



Así es como con estas tres funciones se crea un sistema básico de teletransporte mediante navegación en la *navmesh*.

Lo que queda de este apartado es explicar qué aspectos se ha pulido y perfeccionado en este sistema. Ya se han mencionado los *fade out* y *fade in* de la cámara durante la acción. Otro aspecto para comentar es el hecho de mostrar la interfaz gráfica del teletransporte solo cuando el usuario desee usarlo. Así es como aparece la variable *bTeleportPressed*. Esta variable se vuelve *true* solo cuando el usuario pulsa o mantiene pulsada la acción del teletransporte y se vuelve *false* al soltar. También se necesita de otra variable para que la acción no se ejecute si está en progreso. Para ello se usa otra variable *bIsTeleporting*. Durante el cambio de posición y rotación se añade un *fade* a la cámara para reducir la sensación de *motion sickness*.





También se creó una *deadzone* para el input. Para ello se usa la variable *TeleportDeadzone* con un valor de 0,7. Esto significa que el usuario tiene que mover el *thumbstick* con porcentaje igual o mayor al 70% del máximo para que el *input* desarrolle la acción de orientar el *pawn* del usuario.

Como resumen del desarrollo de este apartado, cabe destacar el hecho de que este sistema de locomoción está unido a la *navmesh*, por lo que el usuario no podrá teletransportarse

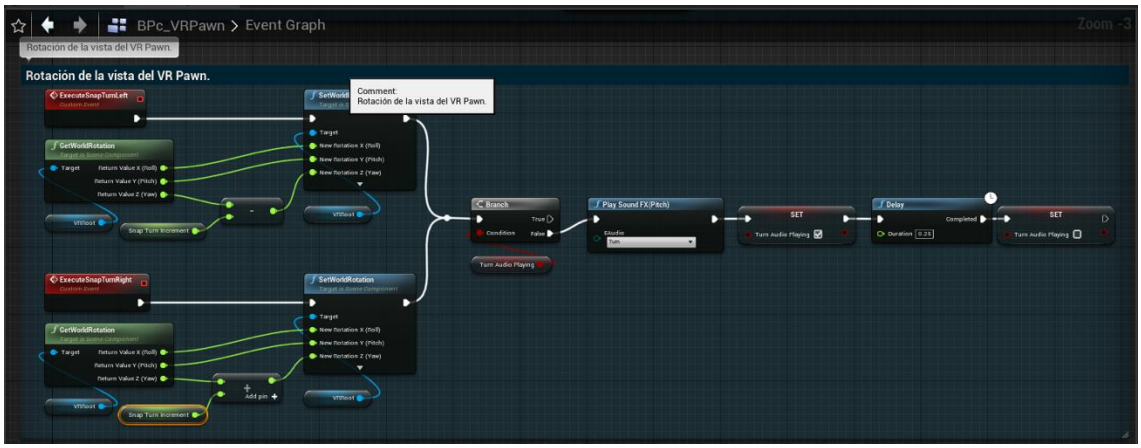
a localizaciones no permitidas por esta. También cabe destacar el uso de un trazo parabólico en lugar de una línea recta por el simple hecho de permitir al usuario acceder a localizaciones de mayor altura. También se incluyó un actor que muestra la localización y rotación de la acción del teletransporte que desea realizar el usuario. El *input deadzone* y el *camera fade* tienen como objetivo hacer más confortable la acción.

Este sistema es un método efectivo para moverse en VR, no intenta representar un movimiento fiel a la realidad. Intenta simplemente ser un método práctico que no despierte el *motion sickness* en el usuario. Por contra se pierde algo de inmersión [1].

II.3 Sistema de giro gradual o *snap-turning*.

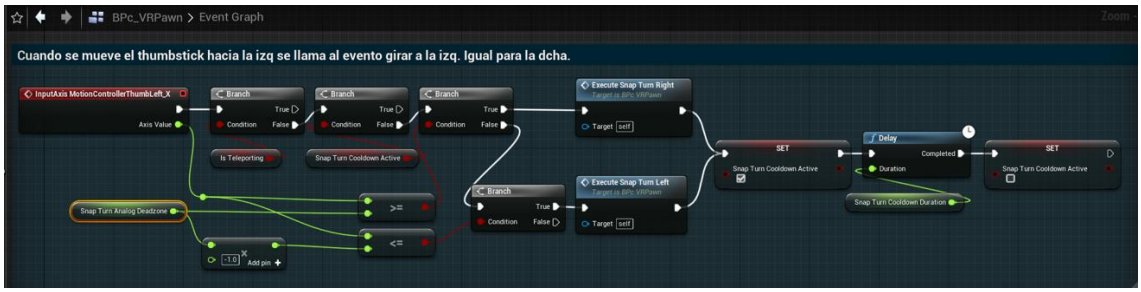
Otro aspecto importante que se necesita implementar trata sobre otorgar al usuario la capacidad de cambiar su orientación sin tener que hacerlo en el mundo real. Es una mecánica práctica y muy útil. La mejor manera para implementarlo es hacer que este giro no se haga suavemente sino mediante pequeños saltos en el ángulo de la orientación. Como se trata de una nueva mecánica hay que añadir su *input* correspondiente, que será *InputAxis Motion Controller ThumbLeftX*.

Se empieza por crear dos eventos nuevos, uno para girar a la izquierda y otro para girar a la derecha. Posteriormente hay que condicionar la llamada a estos eventos. Básicamente estos eventos se encargan de cambiar la rotación del *VRRoot* del *BPc_VRPawn* en un cierto grado almacenado en la variable *SnapTurnIncrement*. Este incremento será positivo para girar a la derecha y negativo para la izquierda. El incremento es de 30 grados sexagesimales, por lo que para dar una vuelta completa haría falta llamar al mismo evento de giro seis veces.



Esta mecánica no puede ejecutarse de cualquier manera. La primera y más evidente condición es que el usuario no se encuentre en medio de la acción de teletransporte. Para ello se hace uso de la variable *bIsTeleporting*, la cual ha de ser *false*. La segunda es crear un *cooldown* o tiempo para volver a activar esta mecánica después de haberla usado, de lo contrario, el usuario podría llamar a este evento una vez por *frame* lo que se traduce en un resultado desagradable y propenso a la *motion sickness*. Por ello se hacen uso de la variable booleana *bSnapTurnCooldownActive* y la variable *float SnapTurnCooldownDuration*. La primera es *false* por defecto, se hace *true* al activar esta mecánica y vuelve a *false* después de 0.2 segundos, que es el valor de la segunda. De esta forma solo se puede ejecutar esta mecánica si *bSnapTurnCooldownActive* es false.

Por otra parte, al igual que se hizo con el teletransporte, también se añade una *deadzone* a la acción. Para ello se usa la variable *SnapTurnAnalogDeadzone* con un valor de 0.8. Este valor es el grado con el que ha de ser activado el *thumbstick* izquierdo de la acción para que sea reconocido, es decir, mayor o igual al 80% de su máximo. Como el valor del eje varía entre -1 y 1, si el valor recogido está en el intervalo [-1, -0.8] el *VRPawn* girará 30 grados a la izquierda. De igual forma si se recoge un valor perteneciente al intervalo [0.8, 1] el *VRPawn* girará 30 grados a la derecha [1].

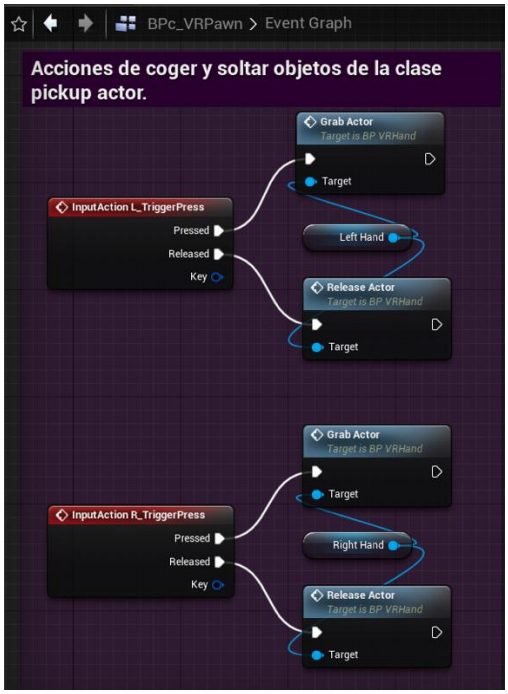


II.4 Sistema de interacción con el mundo virtual.

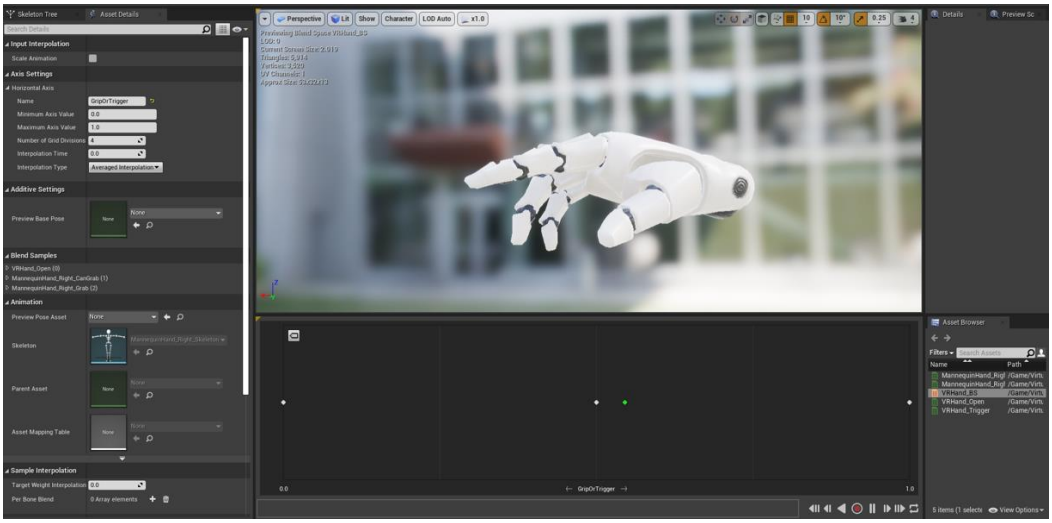
En este mundo virtual el usuario tiene la capacidad moverse por la escena, ya sea haciendo uso del mapa de navegación 3D o bien haciendo uso del teletransporte mediante la *navmesh*. También puede girar en su posición al igual que, como se verá más adelante, hacer *spawn* de un portal de teletransporte hacia el nivel principal o *lobby* desde cualquier lugar. Además de esto, tiene la habilidad de poder coger y soltar ciertos objetos. Para ello necesitará hacer uso de los *motions controllers* representados como manos virtuales. Pulsando el *trigger* de un *controller* podrá coger un objeto si este es interactuable. Al soltarlo, soltará también el objeto [1], [25].

II.4.1 Importación y animación de las manos virtuales.

En el comienzo del desarrollo de este proyecto se describió la creación de los *motions controllers* representados con sus *debug meshes*. En este apartado se detalla cómo fue su reemplazo por la malla de la mano de *Unreal*. En este proyecto se utilizan las de la *VR Template*. Para ello, se migran las animaciones de *CanGrab, Grab* y *Open* junto con materiales, texturas y mallas. Luego se añaden a los *motion controllers* dentro de *BP_VRHand*. En el *construction script* se configura el tipo, escala y rotación de cada una. Lo siguiente fue cambiar la postura o animación de las manos según la situación. Para empezar, se crean dos funciones llamadas *GrabActor* y *ReleaseActor*, dedicadas a contener la lógica para sujetar y soltar objetos interactuables. Las *inputs action mappings* son añadidas en *BPc_VRPawn* con las llamadas a sus respectivas funciones.

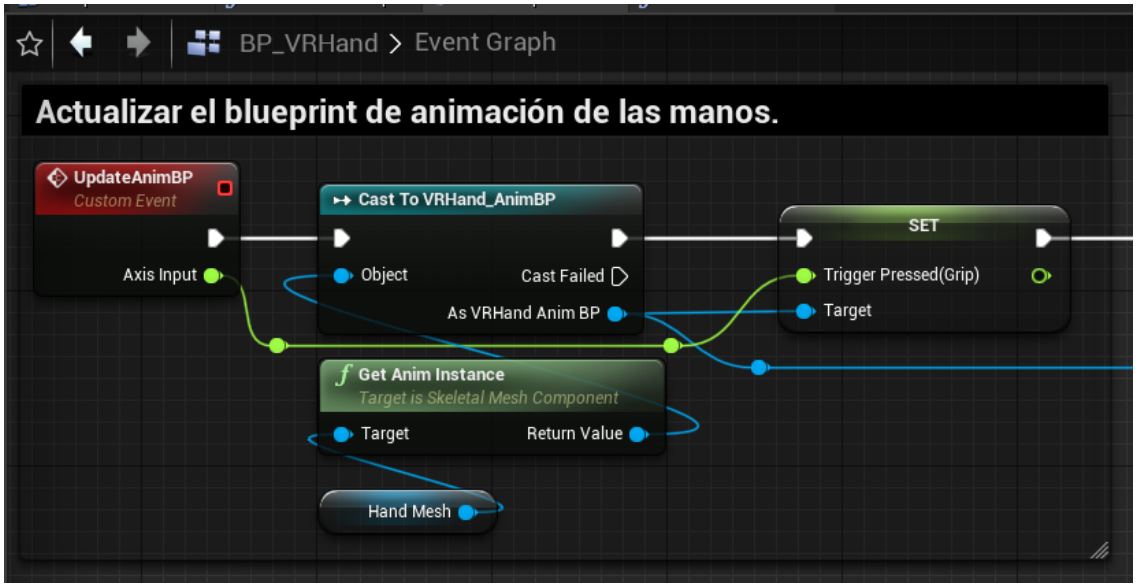


Lo siguiente fue implementar las animaciones para las manos. Las animaciones son tres: una con la mano cerrada para cuando se coge un objeto, otra con la mano abierta por defecto, y otra con la mano a medio cerrar para cuando la mano del usuario se encuentra cerca de un objeto interactuable. Esta última es una forma de dar *feedback* al usuario para que sepa con qué objetos se puede, al menos, interactuar. Se crea entonces un *animation blueprint* llamado *VRHand_AnimBP* con el *target skeleton* de *MannequinHand_Right_Skeleton*. Luego se crea un *Blend Space 1D* llamado *VRHand_BS* para unir las diferentes poses con suavidad. El eje horizontal se corresponde con los valores del axis *trigger* entre 0 y 1. Para cero la animación correspondiente es *Open*, para 0.5 *CanGrab* y para 1 *Grab*. Para el resto de los valores se interpola linealmente.

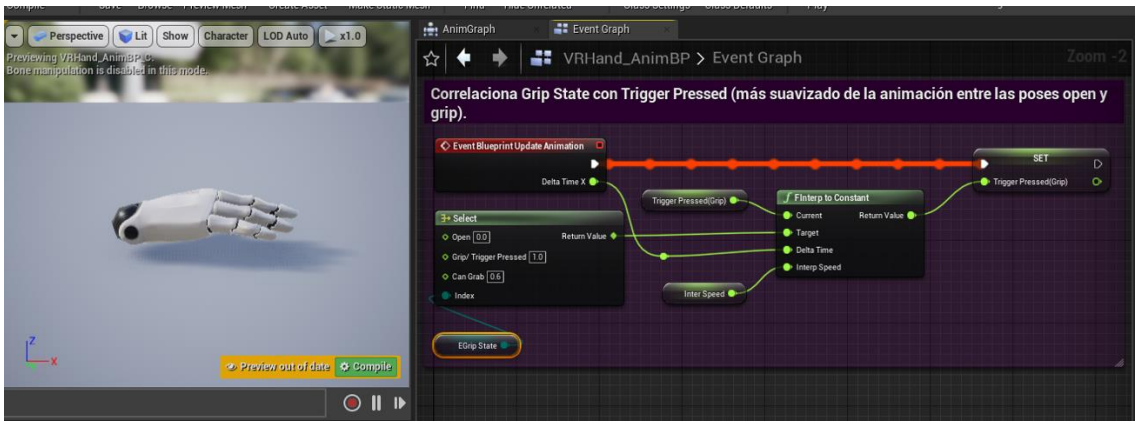


Luego se usa este *blend space* en el *animation blueprint*. Esta es una poderosa herramienta que permite controlar la manera en la que se reproducen las animaciones en una *skeletal mesh*. Se divide en dos secciones principales, el *Anim Graph* y el *Event Graph*. El primero toma los *animations inputs* y los procesa para calcular la pose de la malla en cada *frame*. El segundo sirve para procesar los datos de las animaciones y decidir cuáles se van a reproducir. Se lleva el *blend space* al grafo y se crea la variable *TriggerPressed(Grip)*. Esta variable tiene rango de entre 0 y 1.

Para conectar el *animation blueprint* a la mano hay que seleccionar la *Anim Class* dentro del Actor. El evento *UpdateAnimBP* empareja el valor del *axis input* con la variable *TriggerPressed(Grip)*.



Se crea otra variable de tipo enumeración llamada *EGripState* que contiene las tres poses o estados de las manos. Dentro del *Event Graph* del *VRHand_AnimBP* se correlaciona los valores del *axis trigger* con las poses de la mano. Además, con el nodo *FInterToConstant*, se suaviza el cambio de pose.



II.4.2 Creación de objetos interactuables.

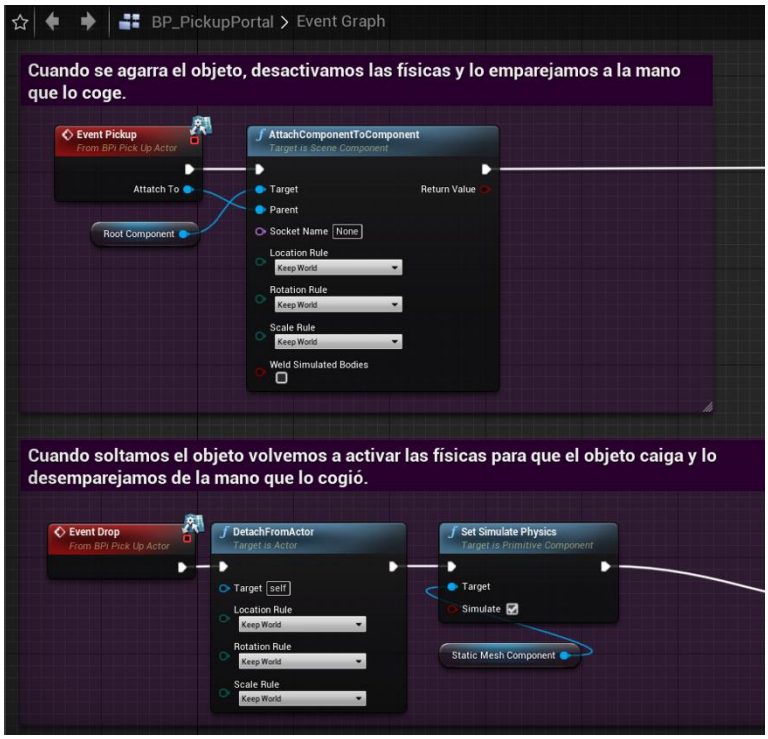
Dentro de este proyecto existen muchos objetos que son interactuables, todos ellos tienen una estructura similar. Se pueden dividir en dos grupos según se puedan coger o no. Los primeros comparten las siguientes características:

- ✓ Clase: *StaticMeshActor.*

- ✓ *Simulate Physics: true.*
- ✓ *Simulation Generates Hit Events: true.*
- ✓ *Generate Overlap Events: true.*
- ✓ *Collision Presets: PhysicsActor.*
- ✓ *Collision Can Ever Affect Navigation: false.*

Estos objetos responden naturalmente a las físicas, pero no bloquean la *navmesh*. Son trece objetos en total: dos portales de navegación entre escenas, un portal de salida, ocho objetos de laboratorio, un fullereno y un cubo. La principal diferencia con los objetos que no pueden ser cogidos pero que sí tienen funcionalidad es que éstos no responden a las físicas. Son tres objetos en total: un botón y dos objetos de laboratorio. Para que un objeto tenga la habilidad de ser cogido se utiliza una *blueprint interface* llamada *BPi_PickUpActor*. La primera función que contiene se llama *PickUp* que incluye un *input* de tipo *SceneComponent* llamado *AttachTo*. La segunda se llama *Drop*.

Esta interfaz es aplicada a los *blueprints* de los diferentes actores. De esta forma se pueden implementar las funciones que se han creado en ella. Se crean dos eventos *PickUp* y *Drop* donde se añade funcionalidad específica a cada objeto. En general los objetos que son cogidos en el evento *PickUp* utilizan el nodo *AttachToComponent* para emparejar el objeto a la mano que lo coge dejando de simular las físicas. Por su parte, el evento *Drop* utiliza el nodo *DetachFromActor* para desemparejar el objeto donde este recupera la simulación de las físicas. Posteriormente se comentarán estos objetos más detalladamente.

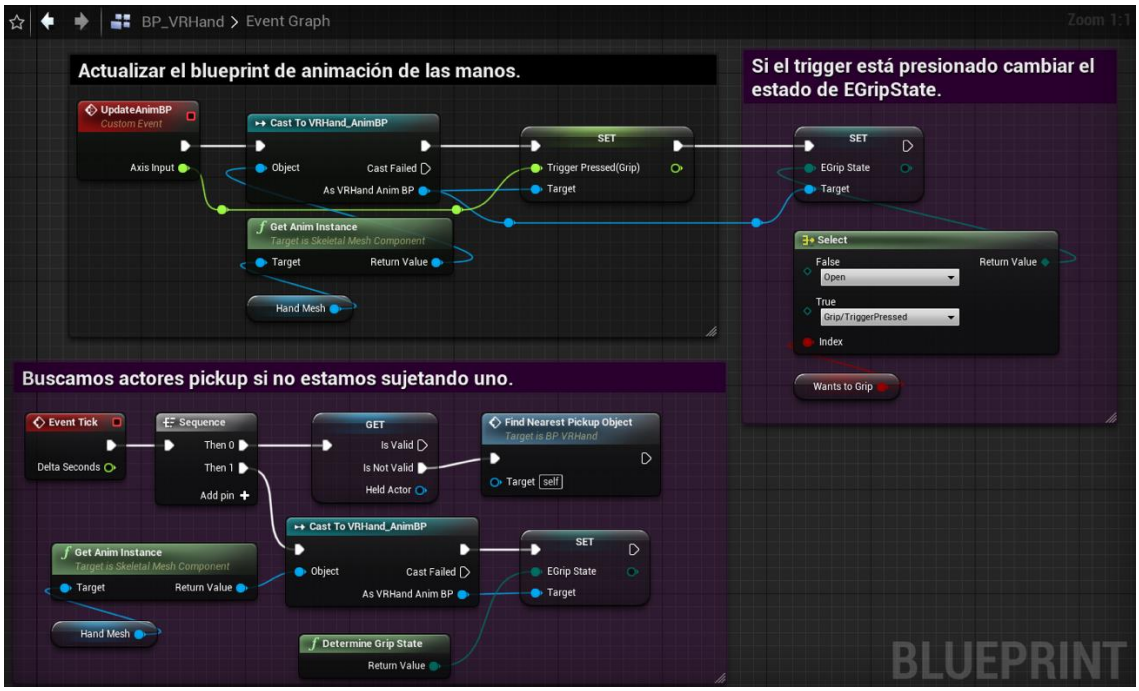


II.4.3 Configuración de las manos para la interacción con objetos.

Lo primero que hubo que hacer fue crear una función para encontrar el objeto interactuable más cercano a la mano. Para ello, en *BP_VRHand* se crea la función *FindNearestPickUpObject* con *Acces Specifier* en *Private*. Esto significa que la función no puede ser llamada desde fuera de la clase, ya que es ésta la encargada de realizar todas las operaciones. Esta función encuentra el objeto interactuable más cercano iterando en un *for each loop* de entre todos los actores con la interfaz *BPi_PickUpActor*. De entre los actores que están lo suficientemente cerca para ser cogidos, ignorando los demás, devuelve el más cercano y lo guarda en una variable para ser usado por otros métodos. Devuelve *null* si no se encuentra ningún actor. La variable *GrabRadius* es el radio de acción o distancia a la cual es posible interactuar con el objeto, y se ha establecido en 32. Ahora bien, comprobar distancias es algo costoso, más aún si esta función es llamada en el *event tick*. Por eso, y porque realmente no importa el valor sino la comparación con el radio de acción, se comparan los cuadrados de los valores, ya que es menos costoso. Para ello se calcula el cuadrado del módulo del vector resultante de la resta entre los vectores de posición del objeto y de la mano. Este valor se guarda en una variable local llamada *LocalCurrentActorDistSquared*. Sólo si este valor es menor al cuadrado del radio de acción el usuario podrá interactuar con el objeto. La razón por la que se crea una variable local es porque será usada si existe más de un objeto interactuable en el radio de acción, y de esta forma no se pierde tiempo recalculando la distancia.

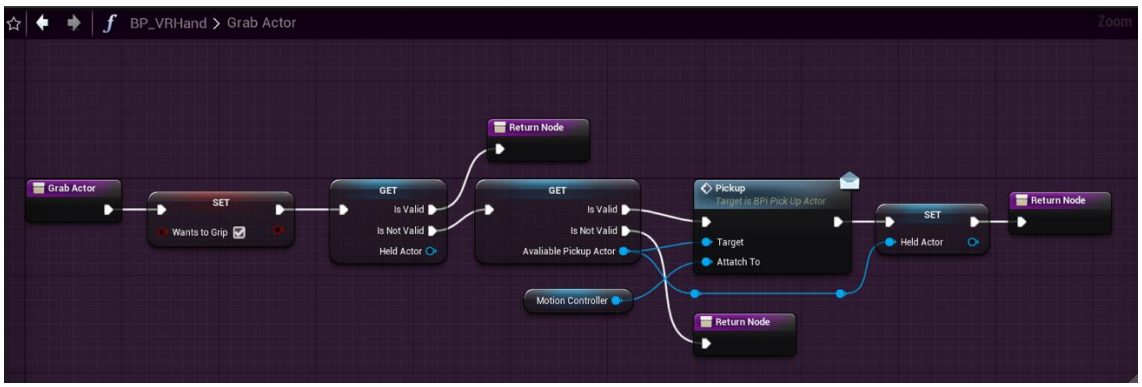
Cabe recordar que una variable local es una variable que solo existe dentro de la función en la que ha sido declarada, no puede ser leída fuera de la función. Por esto se usa este tipo de variable para almacenar el valor de la distancia. Además, estos valores se reinician cada vez que se ejecuta la función. Si el actor pasa el primer *brunch*, es un actor que puede interactuar, ahora bien, lo siguiente es saber si es el actor más cercano. Para ello se crea otra variable local tipo *float* llamada *ClosestRange* con un valor por defecto de 10000. Posteriormente se comparan ambas variables locales, si la primera es menor a la segunda, dicho actor es el más cercano. Por último, se guarda el actor en una variable para que sea usado por el método *PickUp*.

Luego se llama a la función *FindNearestPickUpObject* desde el *event tick*. Dentro de *BP_VRHand* hay que actualizar la animación de la mano solo después de encontrar actores que son interactuables. Luego se crea una variable llamada *HeldActor* de tipo *Actor-ObjectReference* para comprobar si ya se está interactuando con un objeto.

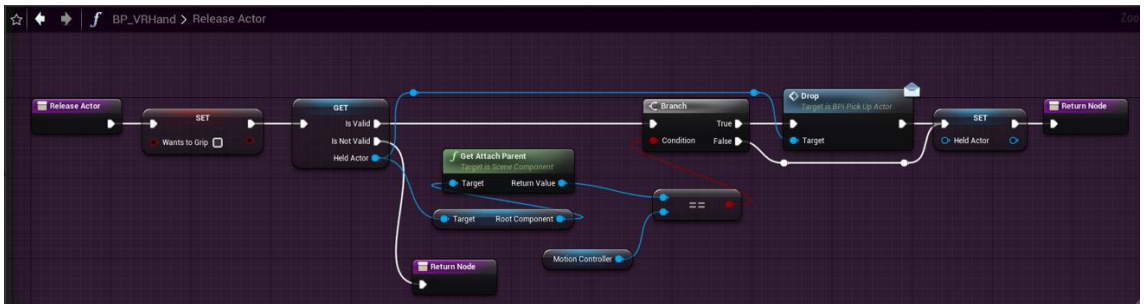


Una vez que se ha encontrado el objeto hay que programar la interacción dentro de la función *GrabActor*. Cuando se llama a la función *GrabActor*, se establece la variable booleana

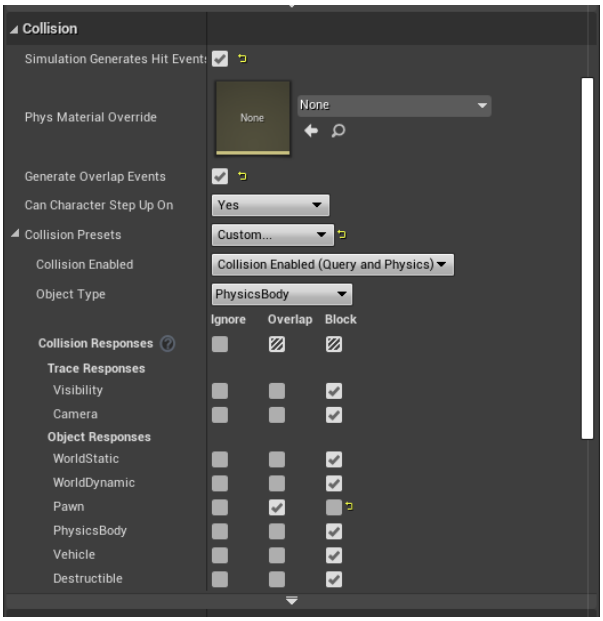
WantToGrip a *true* –marca el estado de animación de la mano–, luego se comprueba si ya se está sujetando un objeto. Si es así no se hace nada. Si no lo es se comprueba que el objeto encontrado en el *EventTick* puede ser cogido. Si no es posible, no ocurre nada. Si lo es, se envía un mensaje a través de la interfaz con la referencia a los *motion controllers* con el objeto que debe ser emparejado y almacenado en la variable *HeldActor*.



Por otra parte, cuando se llama a la función *ReleaseActor* se establece *WantToGrip* a *false* y se comprueba si se está sujetando algo. Si no, no ocurre nada. De lo contrario se comprueba que el objeto sigue emparejado, ya que el objeto podría haber sido sujetado con la otra mano. Si es así, se suelta el objeto y se actualiza la variable *HeldActor*.



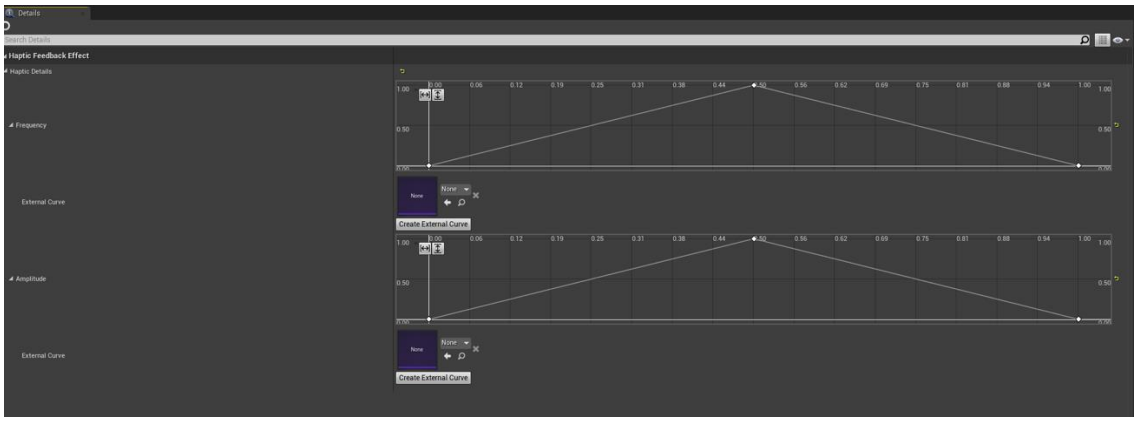
A algunos objetos se les ha editado la colisión para evitar que choquen con la cápsula de colisión del *VR Pawn*. Para ello se selecciona el *Static Mesh Component* y se modifica el *Collision Presets* de *PhysicsActor* a *Custom*, seleccionando la casilla de *Overlap* en *Pawn*. De esta manera se siguen detectando las colisiones con el *pawn* pero sin problemas a la hora de coger dicho objeto.



II.4.4 Adición de *feedback* visual y háptico a la interacción.

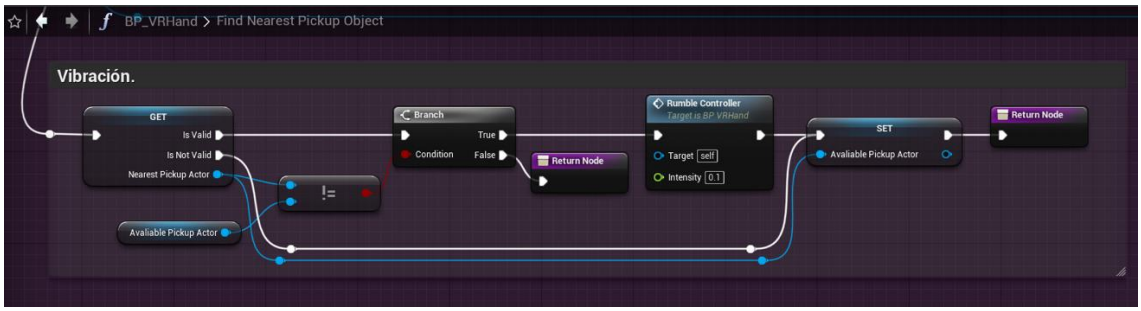
Vista las imágenes anteriores cabe destacar algunos aspectos que aparecen y sin embargo están relacionados con este apartado del desarrollo. Es importante que el usuario sepa cuando puede interactuar con un objeto. Por ello en *EGripState* se creó el estado *CanGrab*, una pose de la mano intermedia entre *Open* y *Grip/TriggerPressed*. La idea es que la mano adopte automáticamente esta pose cuando el usuario se acerca a un objeto interactuable a una distancia menor o igual al radio de interacción ya visto. El estado de la mano se maneja con la función *DetermineGripState* del *BP_VRHand*, según si *HeldActor* sea válido o no – se lleve objeto o no–, y en caso de que no, si existe un objeto interactuable cerca o no.

Por otro lado, a la par que la postura de la mano cambia a *CanGrab* cuando hay un objeto interactuable cerca y no se está sujetando ninguno, se reproduce un efecto háptico o de vibración en el mando de la mano correspondiente. Para crear este efecto háptico hay que crear un *Haptic Feedback Effect Curve* –en este caso se importó el ya existente del *VR Template*–.



Una vez creado el efecto háptico lo siguiente es crear un evento para reproducirlo en *BP_VRHand* llamado *RumbleController*.

Este evento será llamado en la función *FindNearestPickUpObject*, una vez se haya completado el *ForEachLoop* [1]–[3], [25].



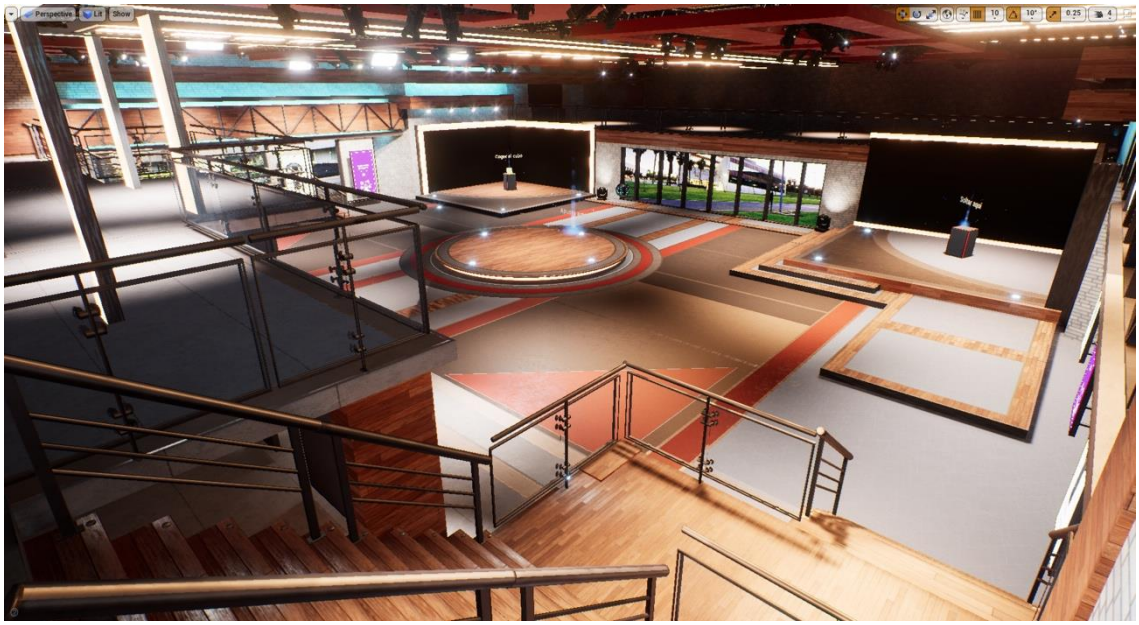
II.5 Diseño y desarrollo del nivel tutorial.

ReadyPlayerLab es un videojuego en el que puedes realizar múltiples tareas relacionadas con la química. No todas las tareas están enfocadas en aprender o desarrollar algún concepto en sí, algunas simplemente son solo pasatiempos. En cualquier caso, antes de llegar a la *lobby* o escena principal del juego, se ha diseñado una escena preparatoria para el uso y entrenamiento de la realidad virtual, inspirado en el tutorial de *steamVR*.

De esta forma el *player* hará *spawn* en este nivel y tendrá que seguir las indicaciones de un *bot* para aprender a dominar las mecánicas básicas del juego. Son las mismas mecánicas ya desarrolladas como abrir y cerrar las manos, girar, teletransportarse e interactuar con objetos.

II.5.1 Arte de la escena tutorial.

De nuevo se ha elegido otro mapa del proyecto *Virtual Studio* del *Marketplace* de *UE4*. Se han modificado ciertos aspectos del arte de la escena para darle una ambientación más acorde al proyecto. Para ello se han utilizado diferentes recursos como sonidos, imágenes y vídeos. Algunas imágenes se corresponden con localizaciones de la ULL, al igual que uno de los dos vídeos promocionales. El otro de archivo de vídeo se corresponde con *Tools for Traveling Through a New Reality* vídeo publicado en la plataforma Youtube por Windows.



También se incluyen diseños propios, como el de los *controllers* y sus acciones y otros diseños sobre la titulación.


Para moverse
y girar
Thumbstick izq
Trackpad Izq
Para viajar
con el portal
Opciones WMR
Lenovo
Left
Controller

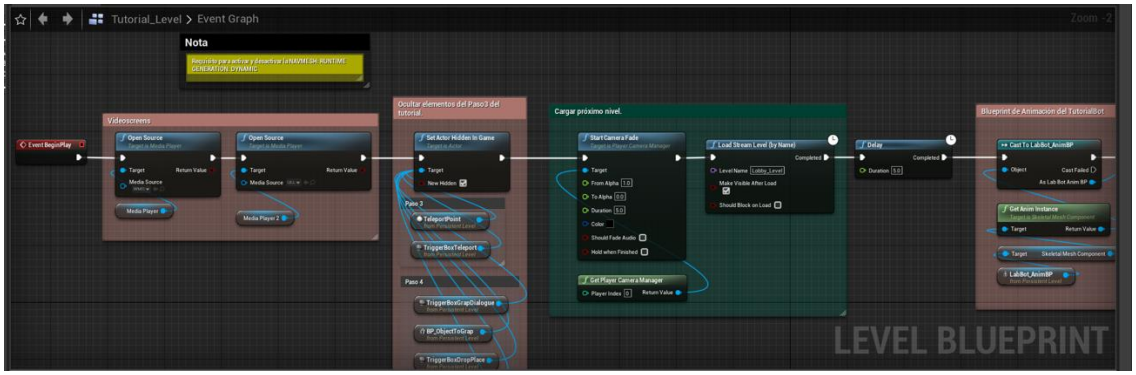

Trackpad Dcho
Thumbstick
Dcho
Para
orientarte
Opciones WMR
Lenovo
Right
Controller

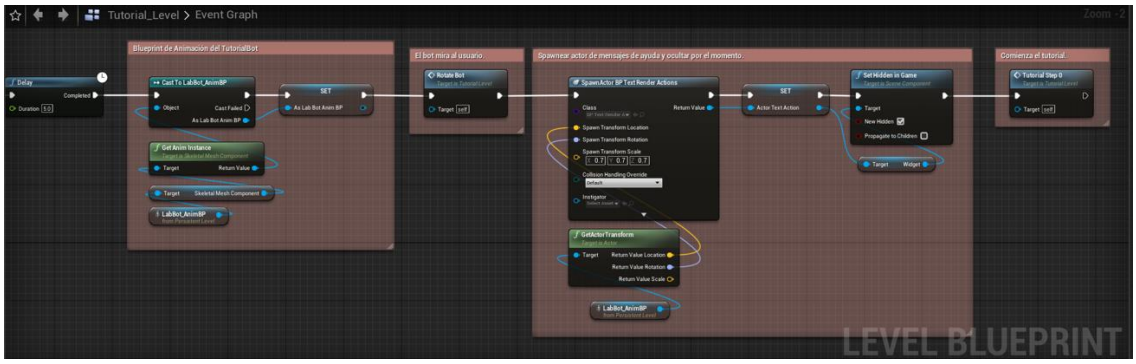

Gatillo Derecho
Para coger y
soltar objetos
Grip Derecho
Right
Controller


Grip Izquierdo
Gatillo Izquierdo
Para coger y
soltar objetos
Left
Controller

II.5.2 *Mainloop* del nivel tutorial.

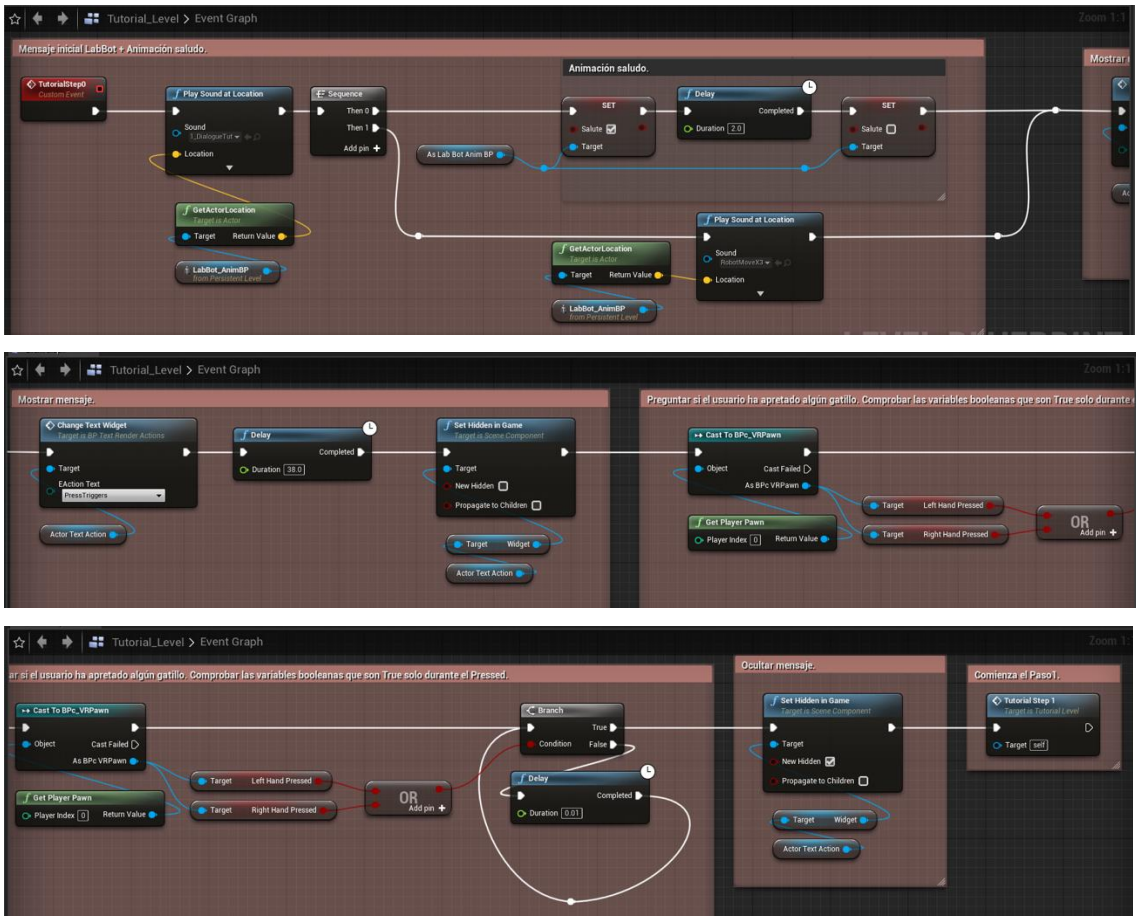
La misión del usuario en este nivel es simplemente seguir al pie de la letra una serie de instrucciones que le muestran cómo funcionan las mecánicas del juego. La mayor parte de esta lógica está desarrollada en el *Level Blueprint* distribuida en seis diferentes eventos. Para empezar, se desactiva la *dyamic navmesh*, ya que no se desea que el usuario abandone el área de *spawn* al comienzo. Como se incluyen archivos de vídeo, hay que utilizar el nodo *OpenSource* para reproducirlos. En el *Event Begin Play* se tienen estos nodos junto con un *fade* de cámara inicial hasta llamar al evento *TutorialStep0*, que es el evento correspondiente a la acción de abrir y cerrar las manos. Más adelante se desarrollará la programación de *LabBot*, el *bot* de que dirige el nivel del tutorial, pidiendo al usuario que realice acciones. Como se hará uso del *blueprint* de animación de este personaje, se hace un *cast* al *AnimBP* del mismo y se guarda en una variable para futuros usos. También aparece una función llamada *RotateBot* encargada de hacer que el *bot* mire siempre al usuario. Por último, hay un *spawn* del actor que representa mensajes de ayuda durante el tutorial.



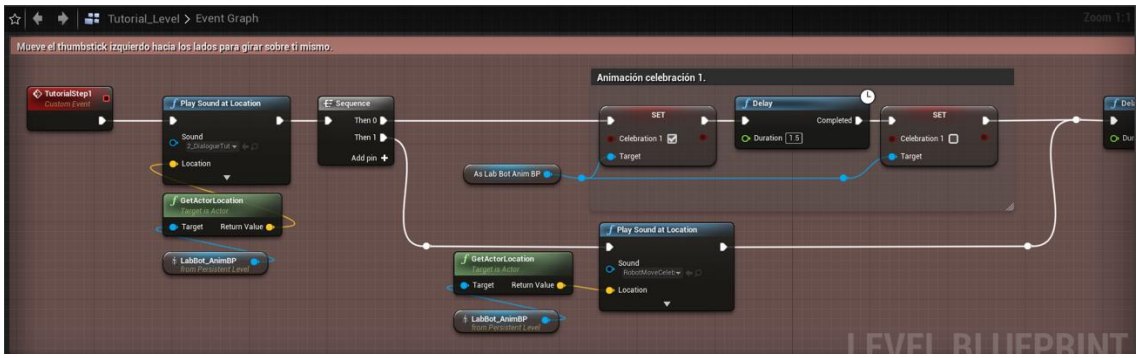


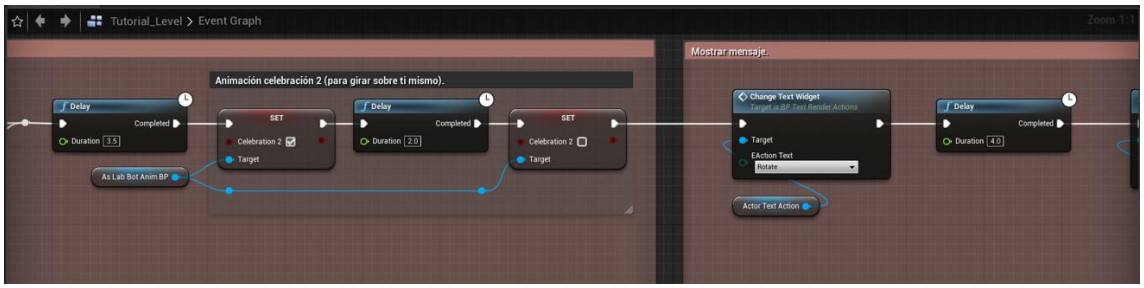
El evento *TutorialStep0* empieza con la reproducción del sonido del mensaje inicial del *bot* coordinada con una animación de saludo y sonidos para la misma. También se hace visible el mensaje de texto que resume la acción que debe ejecutar el usuario. Posterior a la explicación se entre en un bucle donde se pregunta continuamente si el usuario a ejecutado la acción de pulsar el gatillo. Para ello se aprovecha la variable del *VRPawn bLeftHandPressed* o

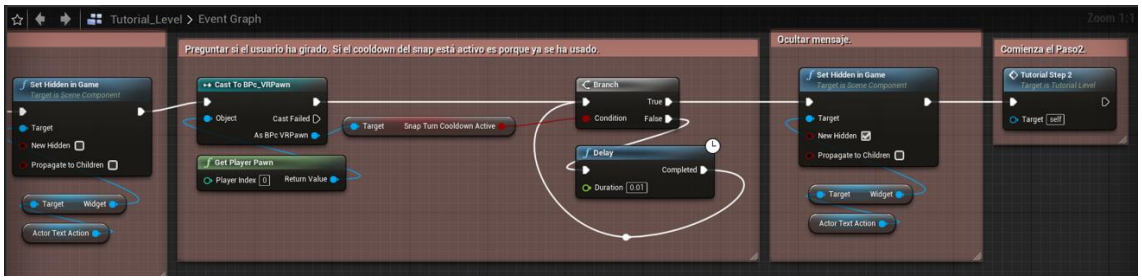
bRightHandPressed. Cuando lo hace, esta variable se vuelve true, se sale del bucle, y se prosigue con el siguiente evento.



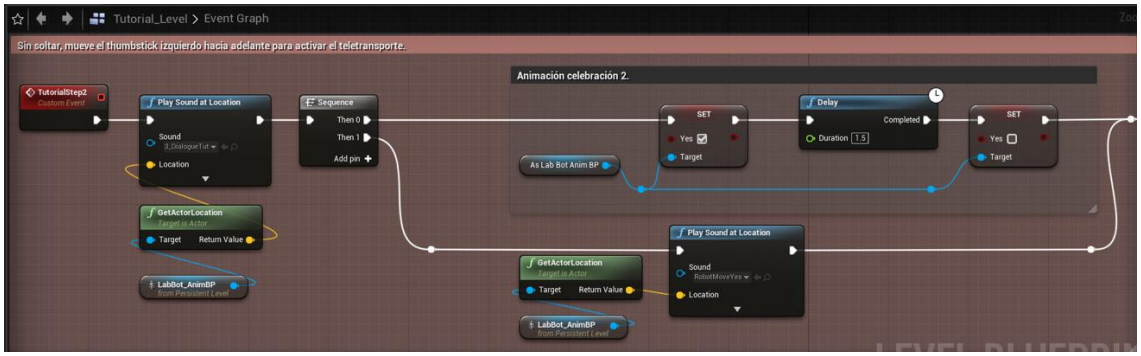
El evento *tutorialStep1* tiene como fin que el usuario utilice la mecánica de giro. Sigue el mismo esquema que el evento anterior con mensaje inicial del *bot* más animaciones y sonidos, actualización del mensaje de acción y bucle para preguntar si el usuario ha ejecutado la acción. Para ello se utiliza la variable *SnapTurnCooldownActive* del *VRPawn*.

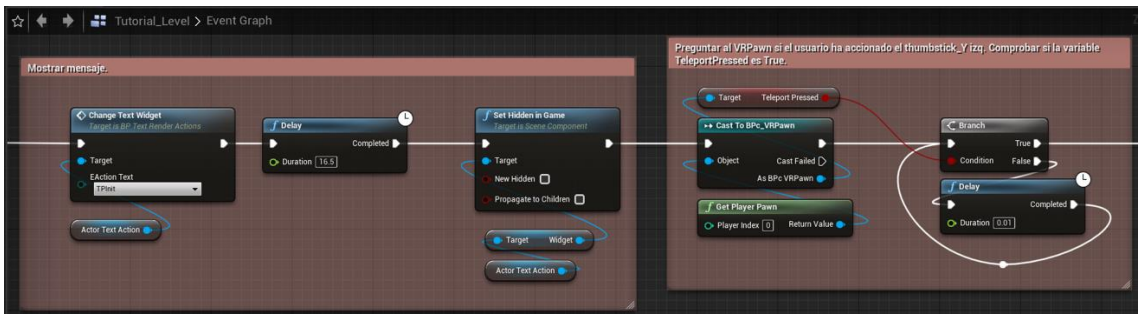


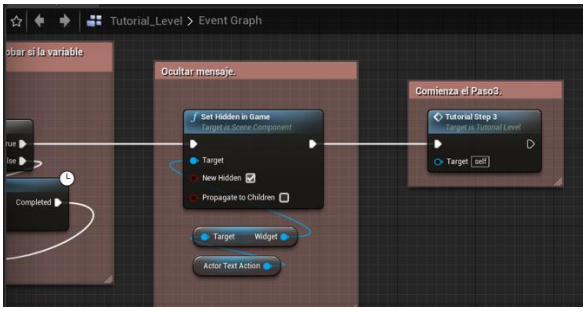




TutorialStep2 es el evento encargado del inicio del teletransporte, es decir, mover el *thumbstick* hacia delante para calcular la localización de llegada. En este caso se comprueba si la variable *teleportPressed* del *VRPawn* es *true*.

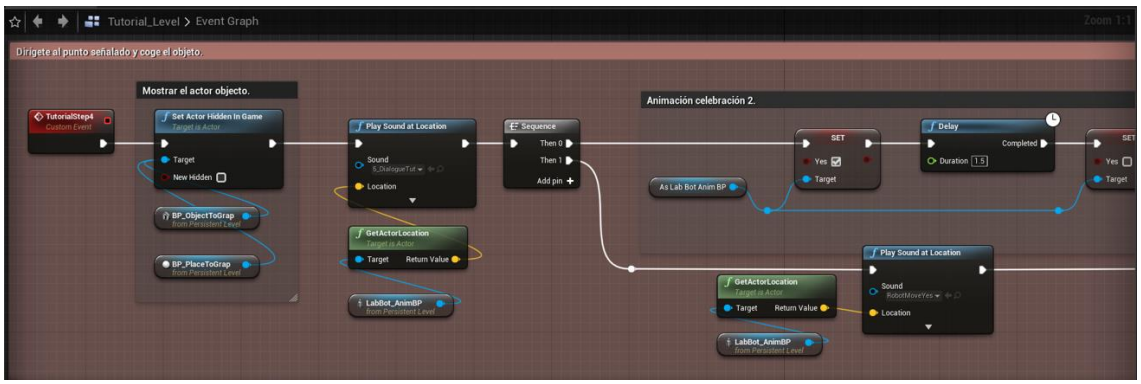


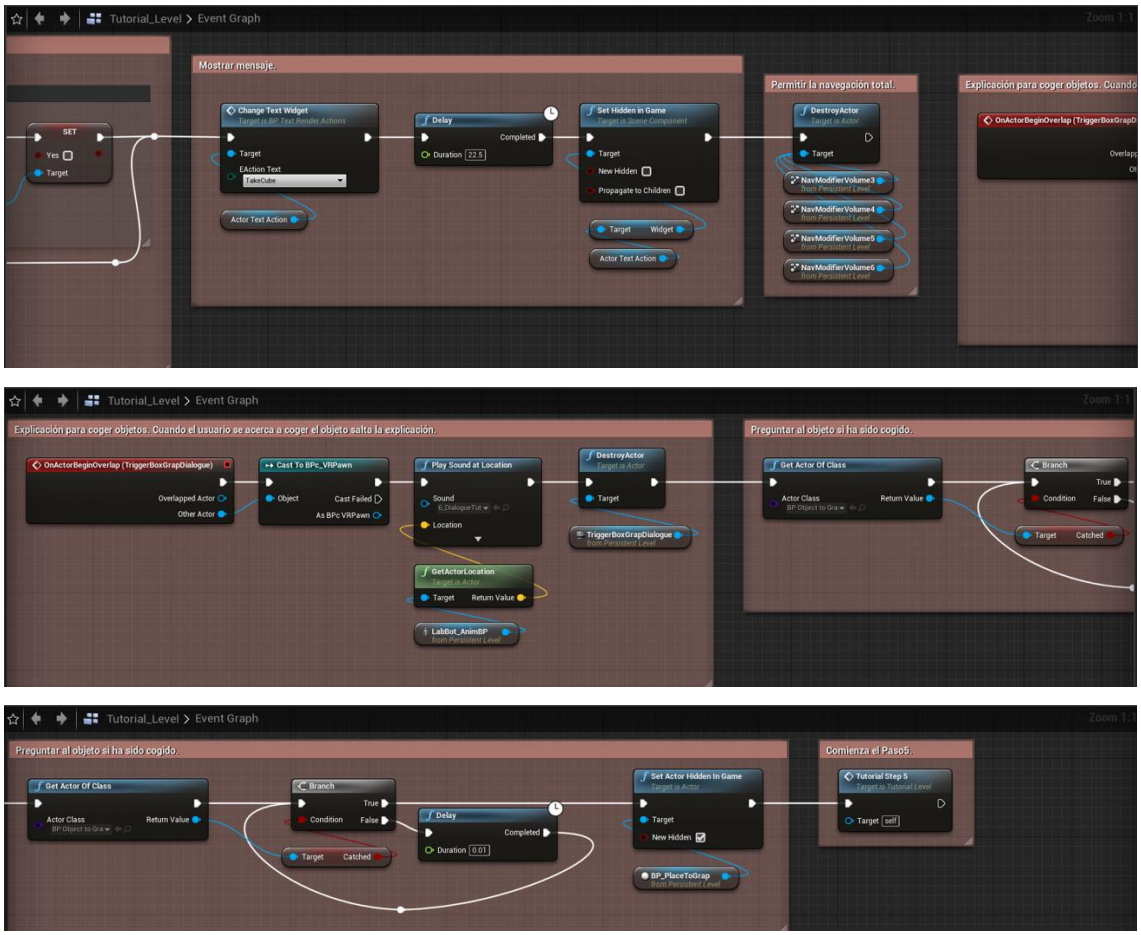




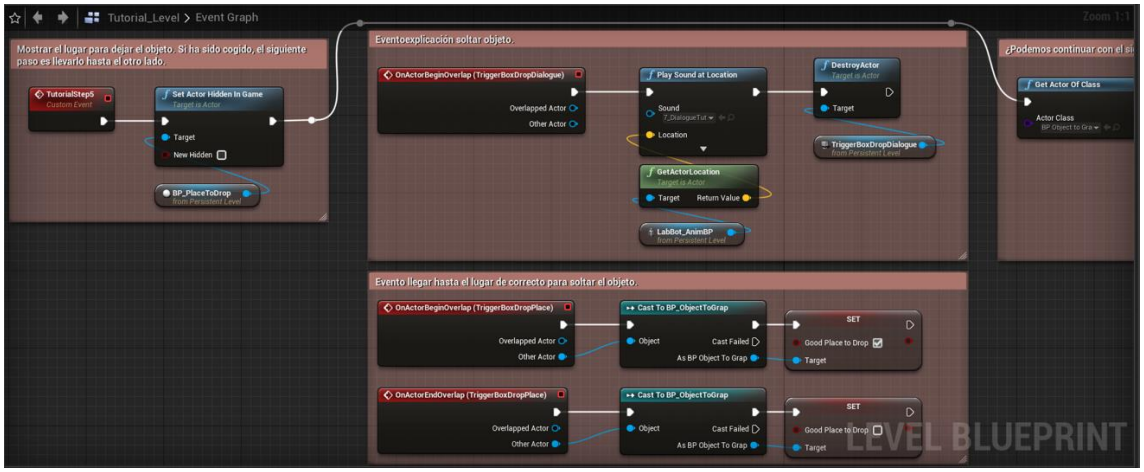
TutorialStep3 se encarga de comprobar que se ha confirmado el teletransporte. En este caso el usuario solo puede teletransportarse a un lugar objetivo de la escena, ya que es el único sitio activado de la *navmesh*. Aparece también un efecto de partículas y un texto para guiarle. Cuando este ha finalizado el teletransporte, un *triggerbox* en la zona activa el siguiente evento.

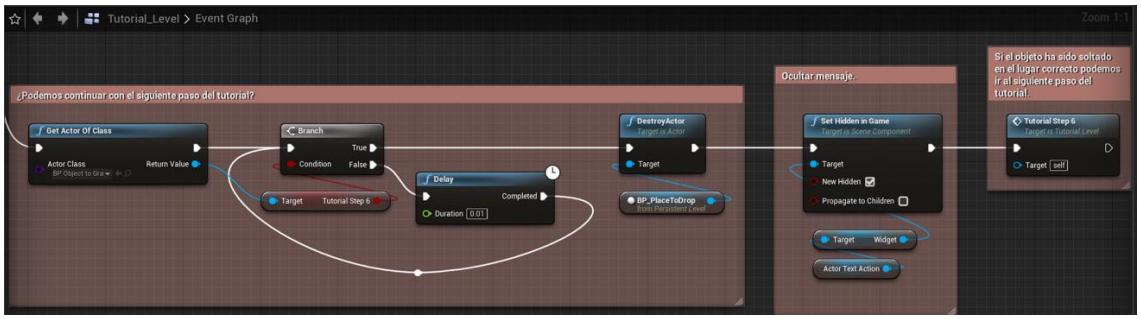
TutorialStep4 es el evento que muestra cómo coger un objeto. Para ello se pide al usuario que se dirija a la zona objetivo y agarre un objeto. Un *triggerbox* de la zona detecta si el usuario ha entrado en la misma para añadir una aclaración extra.



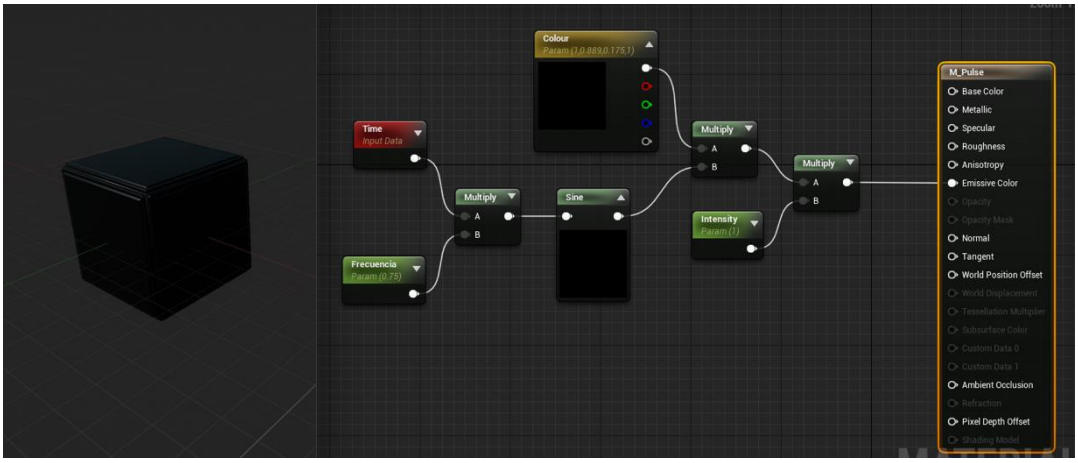


De igual forma en *TutorialStep5* se muestra como soltarlo en el lugar correcto. Para ello se hace uso de una variable booleana en el *blueprint* del propio objeto *BP_ObjectToGrap*. Este objeto tiene la estructura base típica de los objetos interactuables con los eventos *PickUp* y *Drop*. Si el objeto entra en la zona adecuada la variable *GoodPlaceToDrop* se vuelve *true*. Si posteriormente el objeto es soltado con esta variable en *true*, se prosigue al siguiente evento. En caso contrario el objeto vuelve a su lugar original.

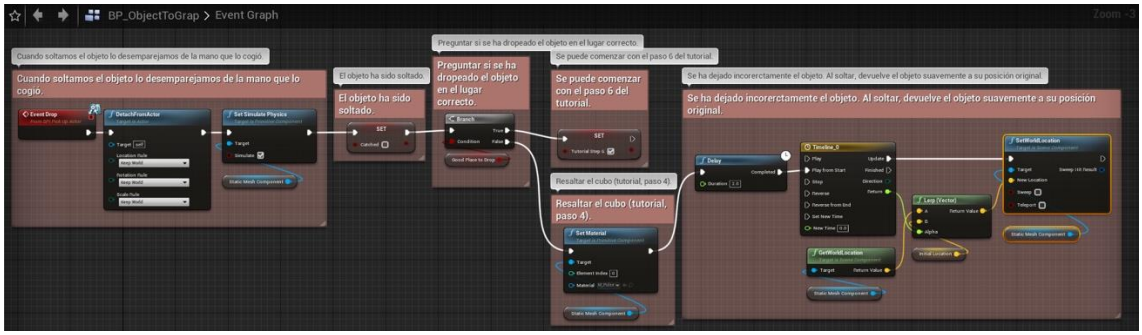




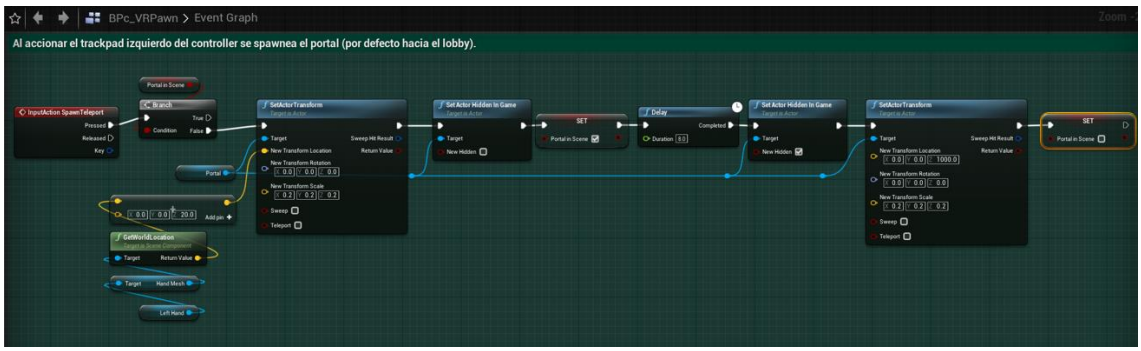
La programación desde el lado del objeto *BP_ObjectToGrap*, que es un cubo, consta de tres eventos. En el *BeginPlay* se guarda la posición original del objeto en una variable. De por sí, el objeto tiene un material emisivo pulsante para llamar la atención del usuario llamado *M_Pulse*.



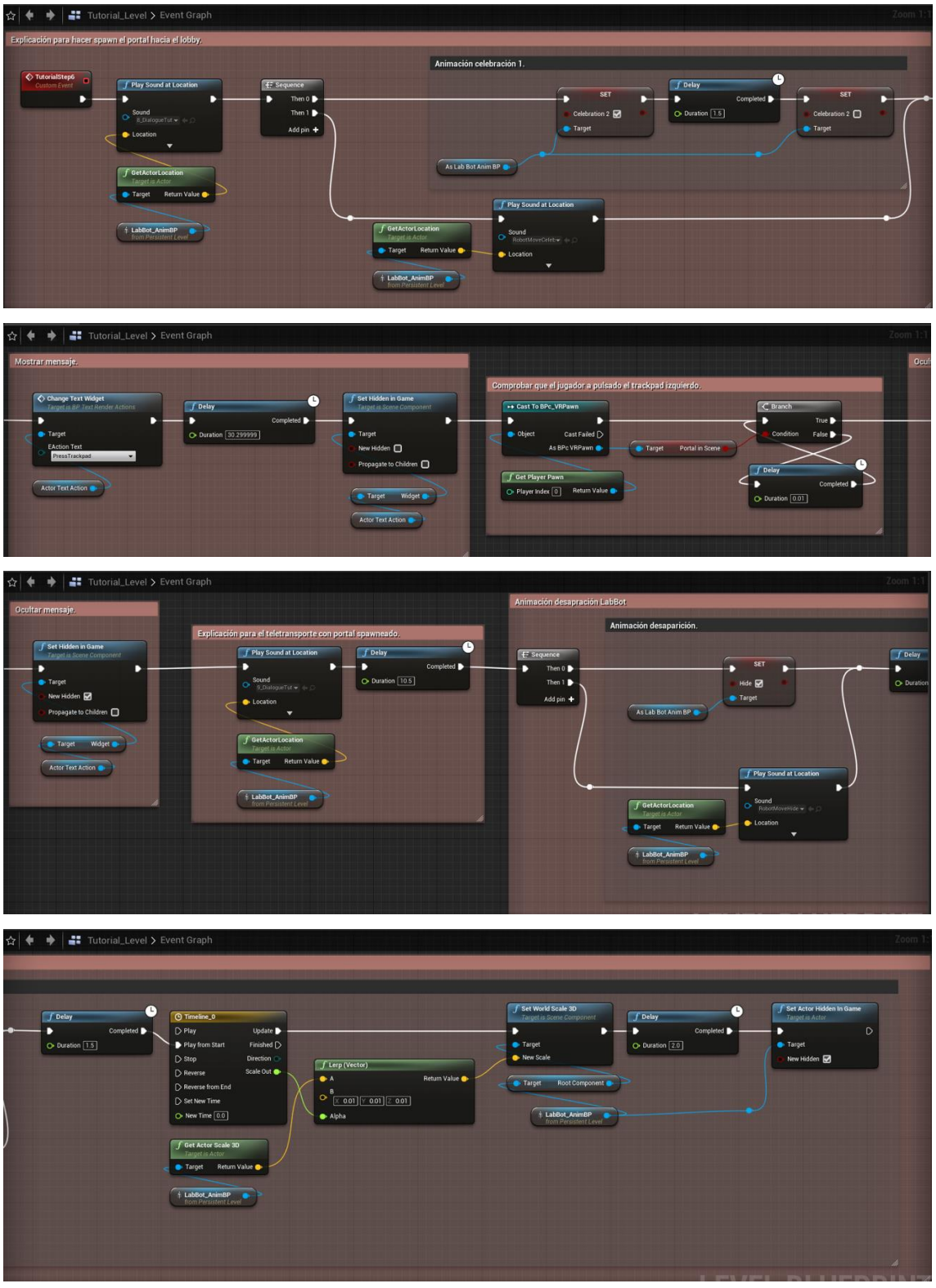
En el evento *PickUp* el objeto cambia su material, se empareja a la mano y se dejan de simular las físicas como era de esperar. Pero en el evento *Drop*, además del desemparejamiento y la activación de las físicas, puede ocurrir que el objeto sea soltado en un lugar correcto –indicado por el *triggerbox* anterior– activando el booleano *GoodPlaceToDrop* o puede que no. Si es el lugar correcto se procede al siguiente tutorial y si no el objeto vuelve a su posición original recuperando el material pulsante.



TutorialStep6 tiene como objetivo mostrar la última de las mecánicas básicas al usuario, hacer aparecer un portal de teletransporte al *lobby* desde cualquier lugar o nivel en la posición en la que se encuentra el usuario. Para ello solo hay que pulsar el *trackpad* izquierdo. Cuando el portal aparece en la escena se convierte a *true* una variable llamada *PortalInScene* dentro del propio *BPc_VRPawn*. En ese momento se despide el *bot* y se da por finalizado el tutorial. El usuario puede seguir explorando la escena hasta que quiera viajar al *lobby* haciendo uso de dicho portal.



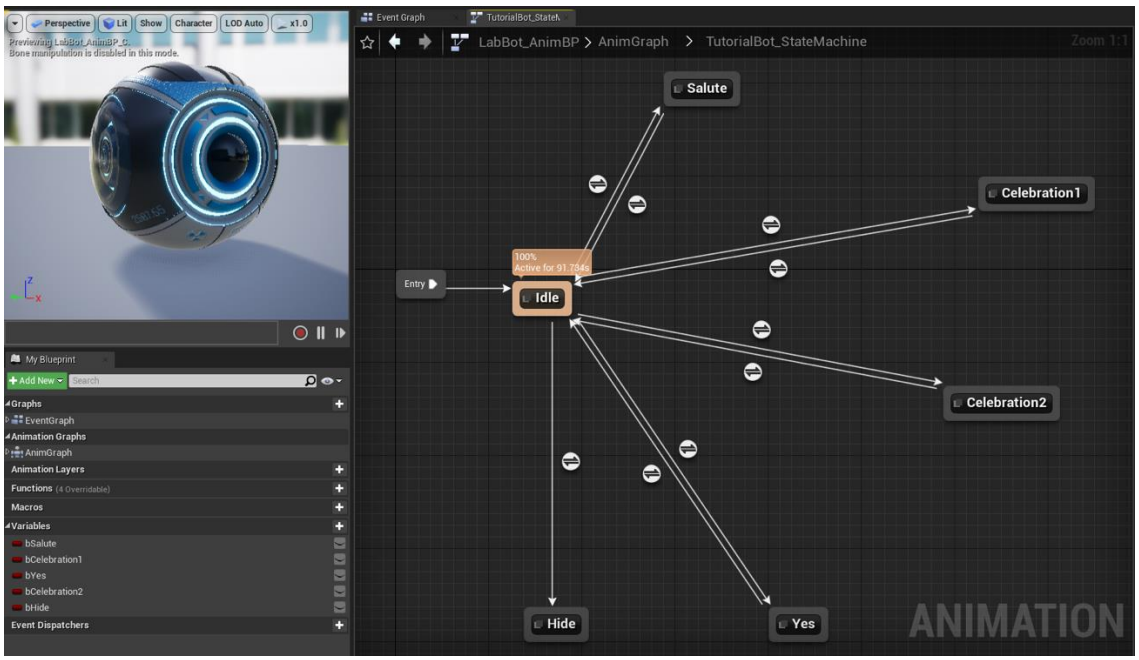
Como se puede ver, al ejecutar el *inputAction SpawnTeleport* se activa la variable *PortalInScene* el tiempo que el portal es visible y se lleva el actor *Portal* – oculto en la escena– hacia la mano del usuario.



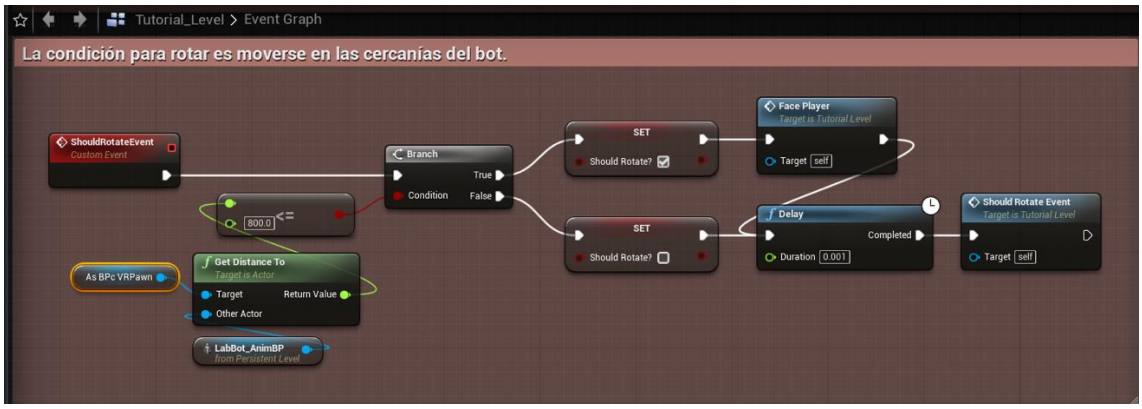
II.5.3 Programación y animaciones del *bot* guía del tutorial: *LabBot*.

LabBot es el nombre del *bot* guía del tutorial. SU misión es la de reproducir las indicaciones que debe seguir el usuario. Para ello se coordinan mensajes de voz reproducidos como audios localizados junto con animaciones y sus sonidos. Este actor es un *asset* llamado

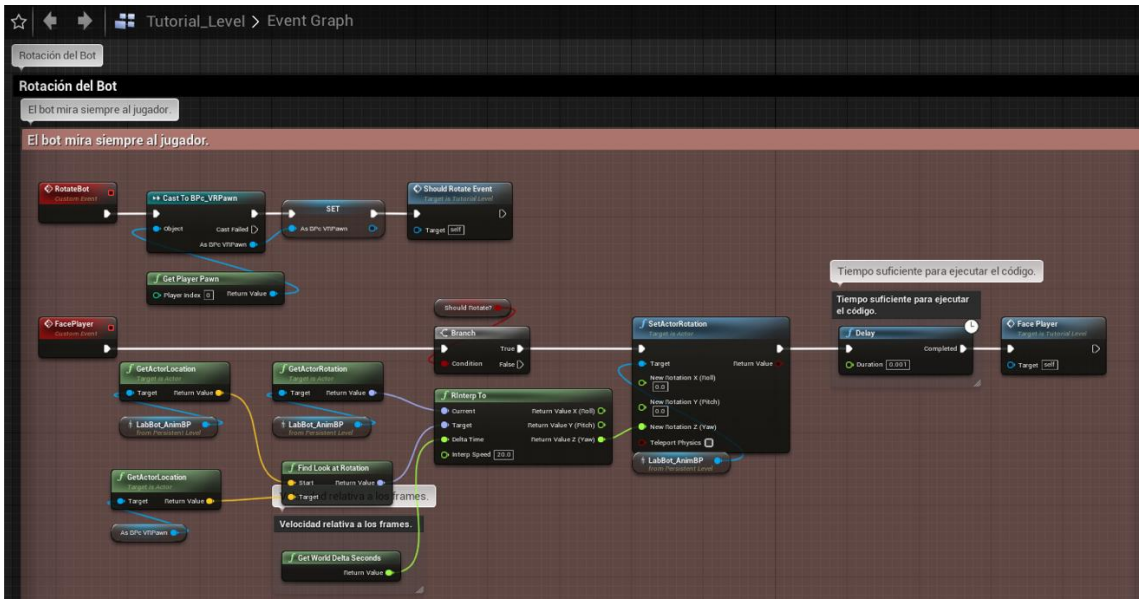
SciFi Drone obtenido desde el *Marketplace* de *Unreal* junto con algunas animaciones. La ventaja de elegir un personaje no humanoide es que se facilita la tarea de coordinar sonidos y animaciones. Su grafo de animaciones está compuesto por seis animaciones diferentes. La activación y desactivación de variables booleanas en el *Level Blueprint* permite la transición entre animaciones.



Otra de las características del *bot* programada en el *Level Blueprint* es que éste siempre mira hacia el usuario. La condición para que cambie su rotación se recoge en el evento *ShouldRotate* y es que el usuario esté lo suficientemente cerca, distancia que establecida en 800 uu.



En los eventos *RotateBot* y *FacePlayer* se desarrolla el resto de la lógica. La función utilizada es *FindLookAtRotation* junto con un nodo *RInterpTo* con un *InterSpeed* de 20 para mayor suavidad en el movimiento.

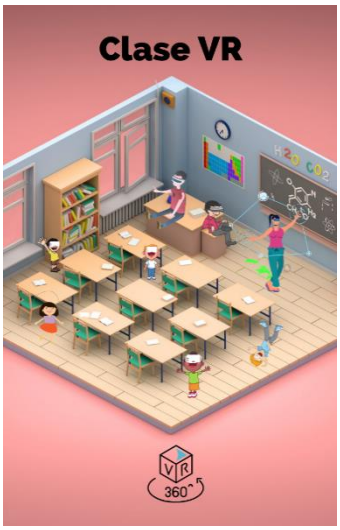


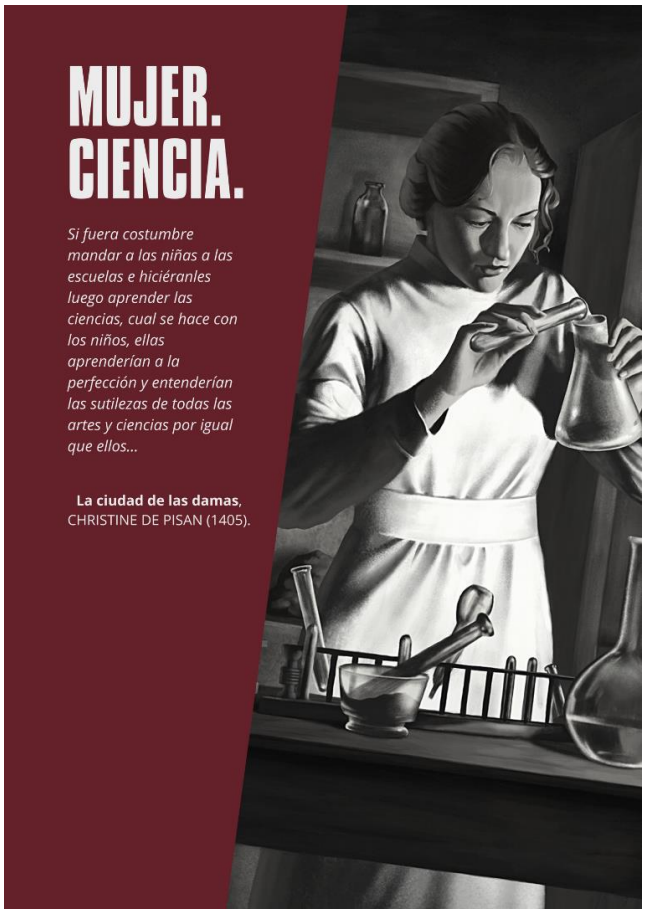
II.6 Diseño y desarrollo del nivel principal: *Lobby*.

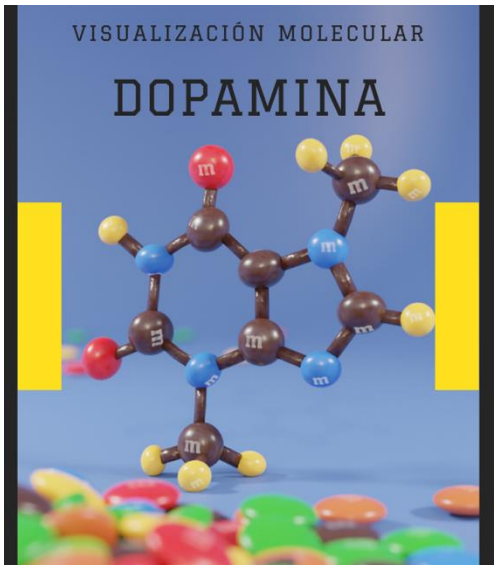
Este nivel es más que un mero tránsito para el usuario. La idea es que éste pueda usarlo además para explorar y jugar. Este nivel se divide en estaciones en las cuales puedes realizar alguna acción como bailar, jugar a baloncesto, viajar de regreso al tutorial, viajar al nivel de laboratorio químico o simplemente salir del juego.

II.6.1 Arte de la escena principal.

La ambientación del juego es la química, todo está relacionado con ella. Ya se comentó al inicio del capítulo aspectos de la escena, pero de nuevo, esta fue modificada con texturas y objetos personalizados para el proyecto. La mayoría de las imágenes que decoran el lugar son diseños propios utilizando elementos gráficos, capturas de pantalla, etc.

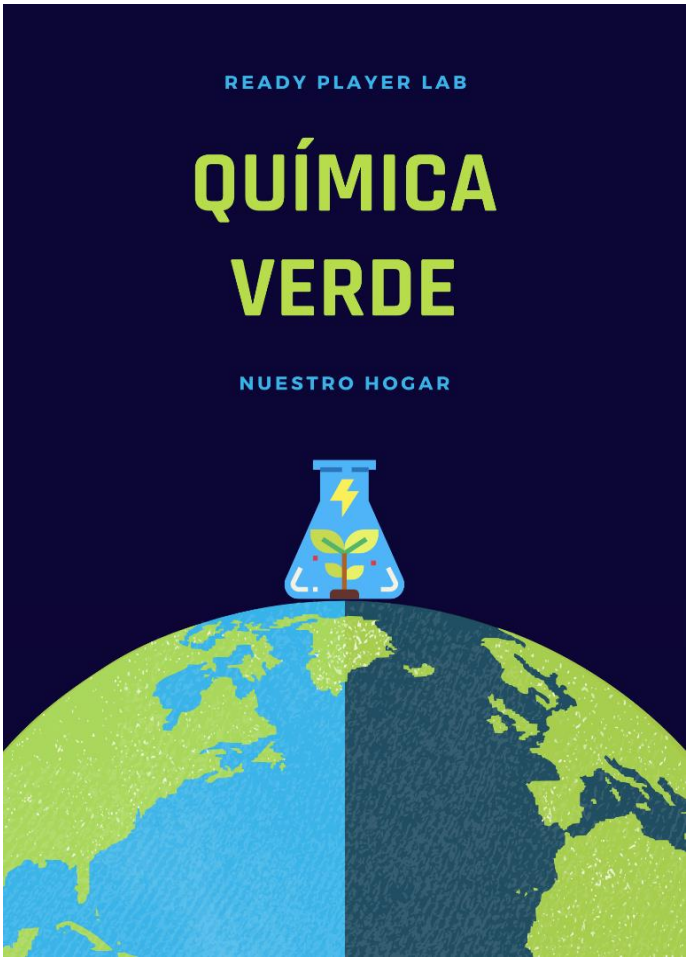



MUJER.
CIENCIA.
Si fuera costumbre mandar a las niñas a las escuelas e hiciéranles luego aprender las ciencias, cual se hace con los niños, ellas aprenderían a la perfección y entenderían las sutilezas de todas las artes y ciencias por igual que ellos...
La ciudad de las damas,
CHRISTINE DE PISAN (1405).

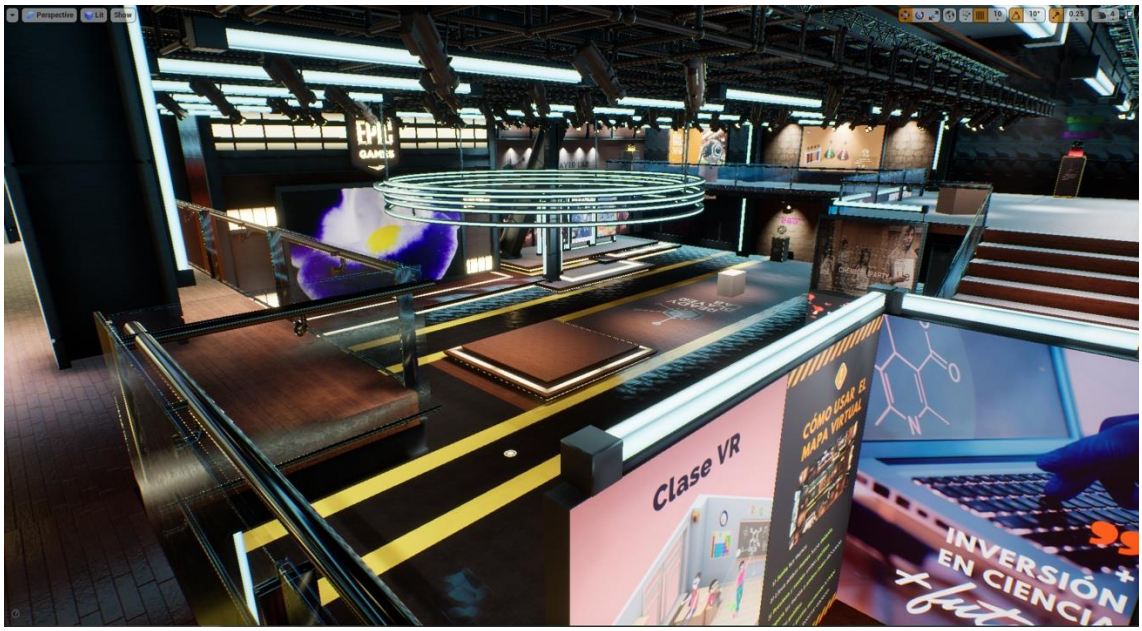

VISUALIZACIÓN MOLECULAR
DOPAMINA


Ready.
Player. Lab.


+
INVERSIÓN
EN CIENCIA
+futuro
I+D


READY PLAYER LAB
QUÍMICA
VERDE
NUESTRO HOGAR


CHEMICAL
DANCE
STATION



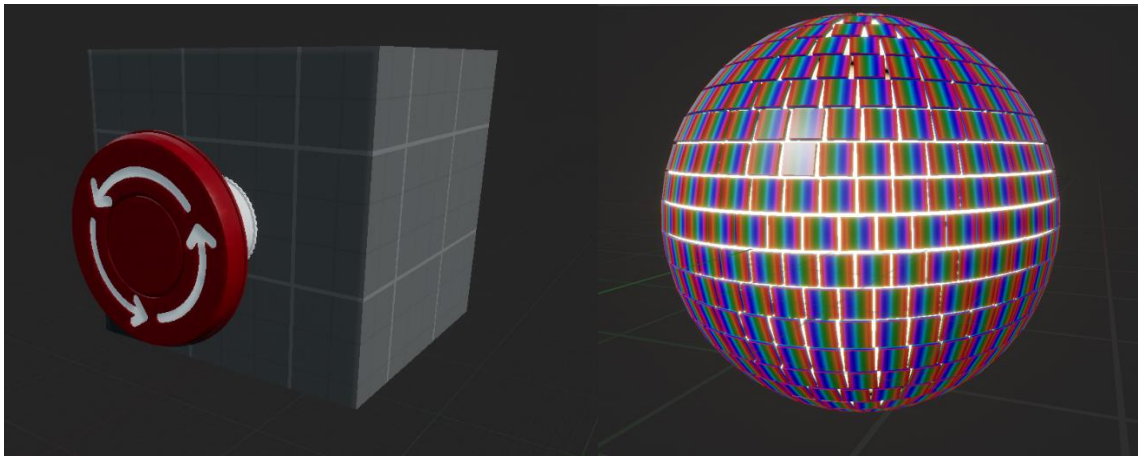
II.6.2 *Mainloop* de la escena principal.

El bucle del juego es simple en esta escena, si el usuario es la primera vez que entra en el nivel, se activa una explicación para usar el mapa de navegación 3D ya desarrollado. Este mapa solo funciona en este nivel y por eso no fue incluido en el tutorial. La explicación se desarrolla en un audio explicativo. Para evitar redundancia, solo se ejecuta la primera vez que se llegue al *lobby*, pero hay unas instrucciones en una zona del escenario para consultar.

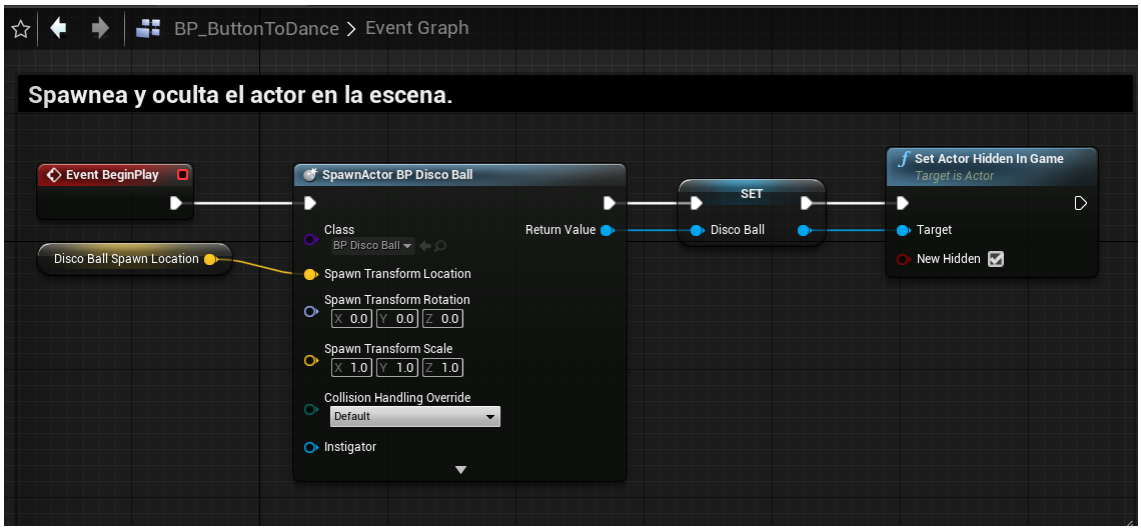
Sin embargo, el nodo *DoOnce* no sirve para ejecutar esta lógica solo una vez, ya que cada vez que se reinicie el nivel se volverá a reproducir. Para solucionar esto se creó una b*lueprint class* de tipo *Game Instance* llamada *VRMapGameInstance*. La programación aquí no se reinicia cada vez que se abre el nivel. Simplemente se creó una variable booleana dentro llamada *bIsFirstTimeLevel?*. Por defecto la variable es *true*, después de ejecutar las instrucciones se vuelve *false*, y ya no se podrá volver repetir el código.

II.6.3 *Chemical Dance Station*.

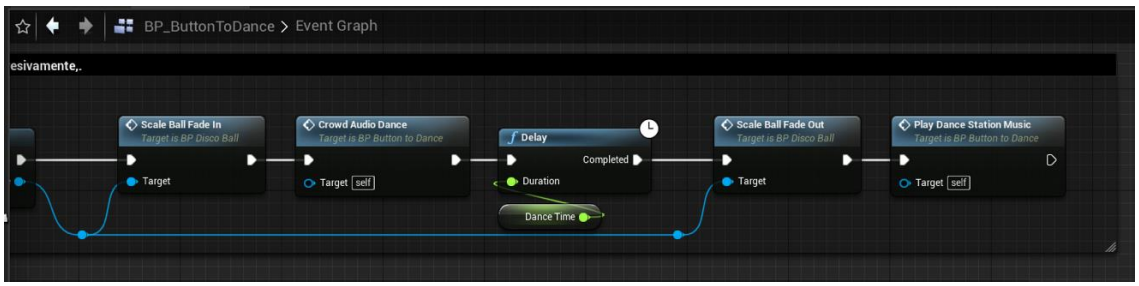
Chemical Dance Station es una estación o lugar de la escena interactuable. Está relacionada con el vídeo que se muestra que compara cómicamente la reactividad química con una fiesta. Es esta estación, además de visualizar el vídeo, el usuario puede interactuar con un botón que activa el baile de un robot junto con el *spawn* de un actor *BP_DiscoBall*. Este botón es un objeto interactuable especial, ya que no se puede coger, se pulsa. La base de la interacción es la misma, pero el evento *PickUp* tiene una programación distinta.

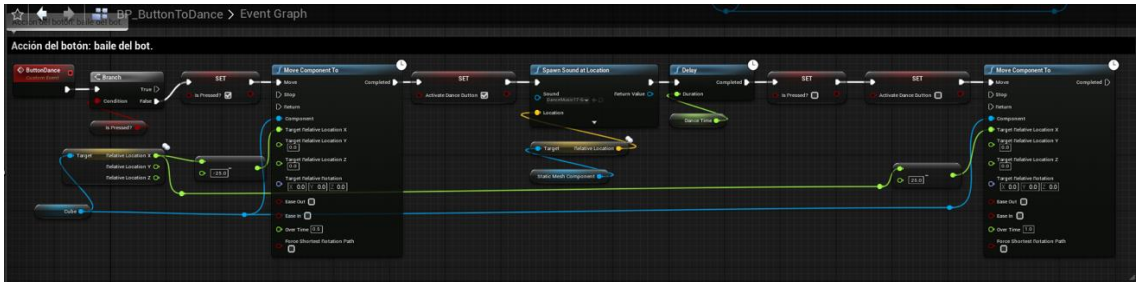


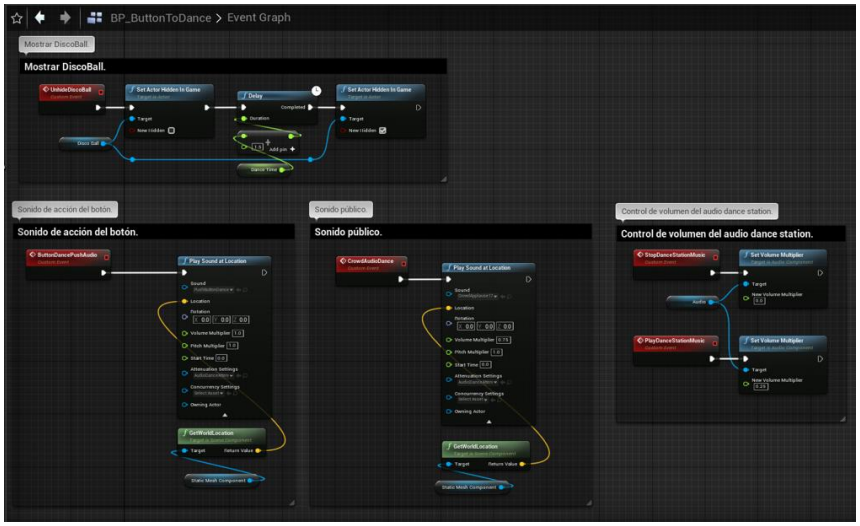
En el *Event Begin Play* se *spawnea* y oculta el actor *BP_DanceBall*. En el *Event PickUp* se ejecutan todas las funciones encargadas de reproducir la música, mover el botón, mostrar la *DanceBall* (girarla y escalarla) y al final volver a desactivar y ocultar todo. El robot que baila es *LobbyBot*, el cual tiene su animación en *LobbyBot_AnimBP* con dos estados, *idle* y *dance*. El evento *PickUp* es quien activa y desactiva la transición entre estado gracias a la variable booleana *bDanceAnim*. La música que suena es *Right Thurr* de *Chingy* como guiño a la película Robots – que será sustituida posteriormente por otra sin derechos de autor–.


BP_ButtonToDance > Event Graph
Spawnea y oculta el actor en la escena.
Event BeginPlay
SpawnActor BP Disco Ball
Class
BP Disco Ball
Return Value
Disco Ball Spawn Location
Spawn Transform Location
Spawn Transform Rotation
X 0.0 Y 0.0 Z 0.0
Spawn Transform Scale
X 1.0 Y 1.0 Z 1.0
Collision Handling Override
Default
Instigator
SET
Disco Ball
Set Actor Hidden In Game
Target is Actor
Target
New Hidden


BP_ButtonToDance > Event Graph
Al pulsar el botón se escucha el efecto push, se mueve al actor, se activa la música de baile, se muestra la DiscoBall escalándose progresivamente, se activa el sonido del publico. Por último se oculta la DiscoBall progresivamente,.
Event Pickup
Button Dance Push Audio
Button Dance
Stop Dance Station Music
Unhide Disco Ball
Get Actor Of Class
Scale Ball Fade In

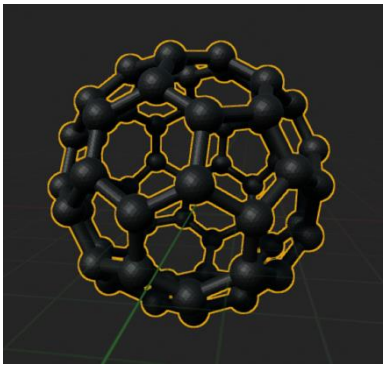

BP_ButtonToDance > Event Graph
esivamente,.
Scale Ball Fade In
Target is BP Disco Ball
Crowd Audio Dance
Target is BP Button to Dance
Delay
Completed
Duration
Dance Time
Scale Ball Fade Out
Play Dance Station Music


BP_ButtonToDance > Event Graph
Acción del botón: baile del bot.


BP_ButtonToDance > Event Graph
Mostrar DiscoBall
Mostrar DiscoBall.
Sonido de acción del boton
Sonido de acción del botón.
Sonido público
Sonido público.
Control de volumen del audio dance station
Control de volumen del audio dance station.

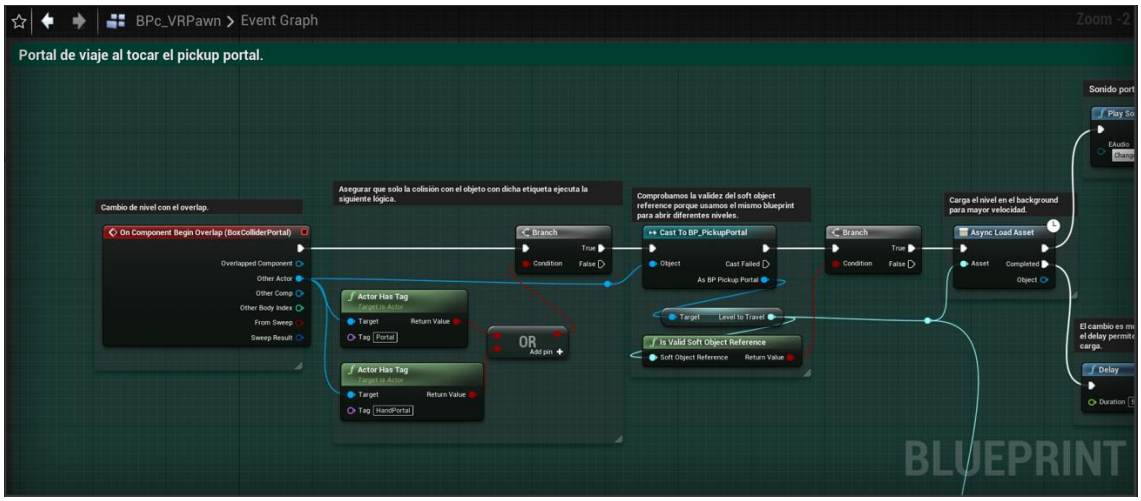
II.6.4 Estación fullereno C_{60}.

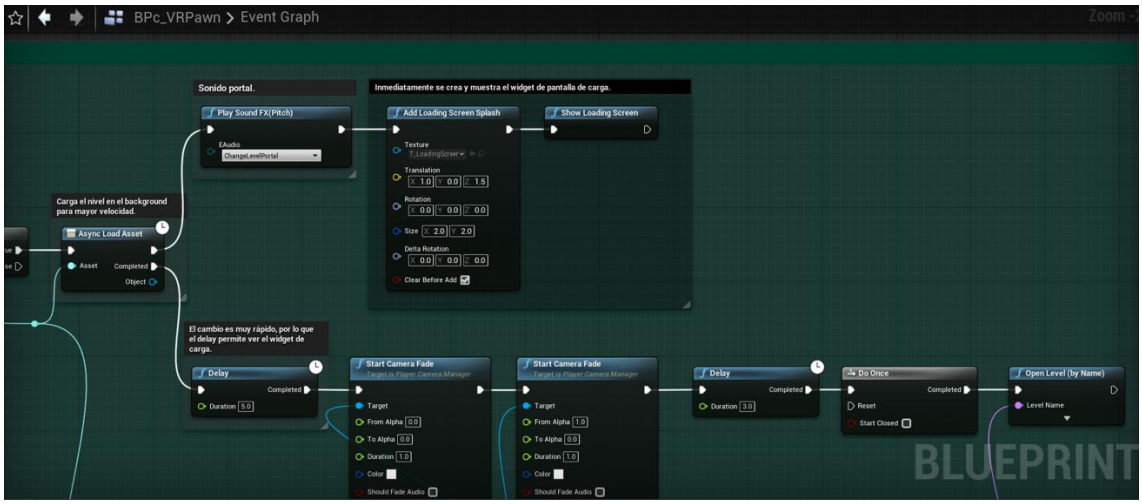
En esta estación el usuario puede interactuar con un objeto redondo y jugar a encestarlo. Este objeto es un fullereno, una molécula compuesta por átomos de carbono –sesenta en este caso–, que adopta una forma geométrica esférica parecida a un balón de fútbol.

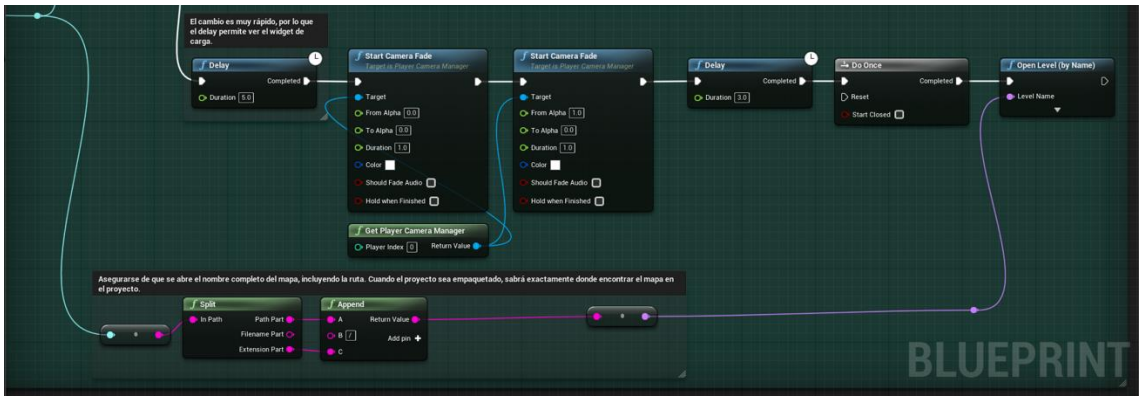


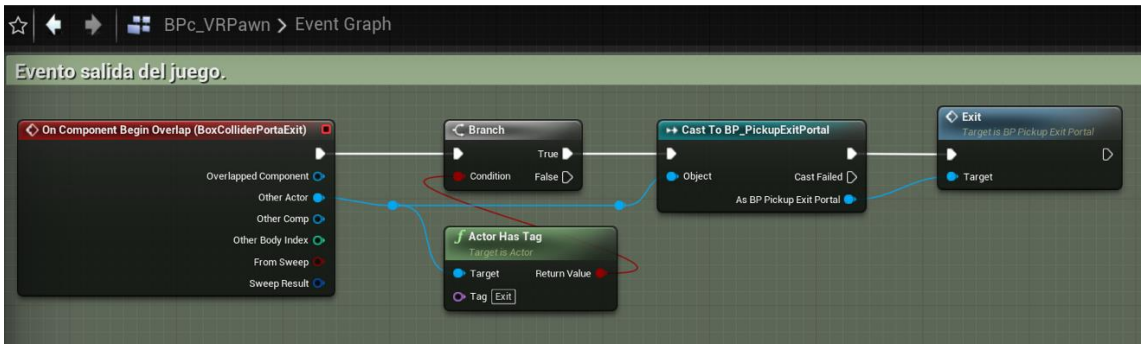
II.6.5 Estaciones de viaje y de salida del juego.

En *ReadyPlayerLab* se utilizan los portales esféricos para viajar entre niveles y/o escenas, o también para salir del juego. Además del portal esférico que el usuario puede llamar en cualquier lugar y momento para regresar al *lobby*, existen en este nivel otros tres portales fijos. Uno de ellos sirve para viajar al nivel del tutorial de nuevo, otro para viajar al nivel de laboratorio químico y otro para salir del juego. Todos estos portales tienen son instancias del actor *BP_PickUpPortal*. Todos son objetos interactuables con la misma estructura base, con métodos *PickUp* y *Drop*. Sin embargo, la lógica de la acción se desarrolla en el *BPc_VRPawn*. La idea es que cuando el usuario sujete un portal, si este colisiona con la cámara del *pawn* se desencadene la acción de viajar. En la cámara del *pawn* hay colocado un *Box Collider* con un *Event Begin Overlap*. En este evento priemero se comprueba que el objeto con el que se ha colisionado tiene la etiqueta correspondiente, luego se comprueba si la instancia del actor *BP_PickUpActor* tiene un *ObjectReference* válido – referencia al nivel al que se quiere viajar–. Si es así, mediante el nodo *Async Load Asset* por un lado se crea un widget de pantalla de carga y al mismo tiempo se abre el nivel correspondiente.


BPc_VRPawn > Event Graph
Portal de viaje al tocar el pickup portal.
Cambio de nivel con el overlap.
Asegurar que solo la colisión con el objeto con dicha etiqueta ejecuta la siguiente lógica.
Comprobamos la validez del soft object reference porque usamos el mismo blueprint para abrir diferentes niveles.
Carga el nivel en el background para mayor velocidad.
On Component Begin Overlap (BoxColliderPortal)
Branch
Cast To BP_PickupPortal
Async Load Asset
Actor Has Tag
OR
Is Valid Soft Object Reference
Delay
BLUEPRINT


BPc_VRPawn > Event Graph
Sonido portal.
Inmediatamente se crea y muestra el widget de pantalla de carga.
Carga el nivel en el background para mayor velocidad.
El cambio es muy rápido, por lo que el delay permite ver el widget de carga.
Play Sound FX(Pitch)
Add Loading Screen Splash
Show Loading Screen
Async Load Asset
Delay
Start Camera Fade
Do Once
Open Level (by Name)
BLUEPRINT


El cambio es muy rápido, por lo que el delay permite ver el widget de carga.
Asegurarse de que se abre el nombre completo del mapa, incluyendo la ruta. Cuando el proyecto sea empaquetado, sabrá exactamente donde encontrar el mapa en el proyecto.
Delay
Start Camera Fade
Do Once
Open Level (by Name)
Get Player Camera Manager
Split
Append
BLUEPRINT


BPc_VRPawn > Event Graph
Evento salida del juego.
On Component Begin Overlap (BoxColliderPortaExit)
Overlapped Component
Other Actor
Other Comp
Other Body Index
From Sweep
Sweep Result
Branch
Cast To BP_PickupExitPortal
Exit
Actor Has Tag
Target is Actor
Target
Return Value
Tag
Exit

Es importante añador un delay durante la pantalla de carga, ya que todo se ejecuta muy rápido. También es importante tener en cuenta que en la pantalla de carga el usuario puede seguir moviendo la vista, para evitar el motion sickness.

II.7 Diseño y desarrollo del laboratorio de química.

En esta escena, el usuario tiene tres misiones o tareas guiadas que constituyen una situación de aprendizaje. La primera de ellas consiste en equiparse con los equipos de protección individual de laboratorio básicos – abreviado *EPIs* o en inglés *PPE–,* que son guantes de nitrilo/látex, bata de laboratorio y gafas de protección. La segunda consiste en visitar las zonas del laboratorio donde están ubicados los elementos de seguridad. Se han incluido seis, que son los más básicos de todo laboratorio – extintor, alarma contra incendios, salida de emergencia, manta ignífuga, botiquín y por último ducha y lavaojos de emergencia. La tercera tarea es seleccionar correctamente material de laboratorio. En esta última misión se incluye un sistema de puntuación que se consulta al final. Para guiar al usuario se utilizan tanto mensajes de audio como visuales. Éstos últimos en forma de texto o imágenes de monitores en la escena. Para seguir el guion también se utiliza un personaje ficticio llamado *Dra Neutrina*, además de *LabBot*.

II.7.1 Arte de la escena de laboratorio.

El escenario base se ha importado desde un proyecto del *Marketplace* de *Unreal* llamado *Realistic Lab Laboratory Equipment*. Sin embargo, muchos de los *assets* utilizados fueron obtenidos de fuentes externas. El arte de la escena intenta ser lo más fiel posible a un laboratorio real. Se puede dividir en tres zonas. En la primera de ellas está situado el despacho del personaje *Dra Neutrina*. La segunda sería un pasillo central donde se ubican los objetos de laboratorio y la tercera donde se sitúa la mesa de trabajo, el fregadero y los reactivos químicos. En la escena hay colocados tres monitores o pantallas de vídeo que se utilizan como recurso

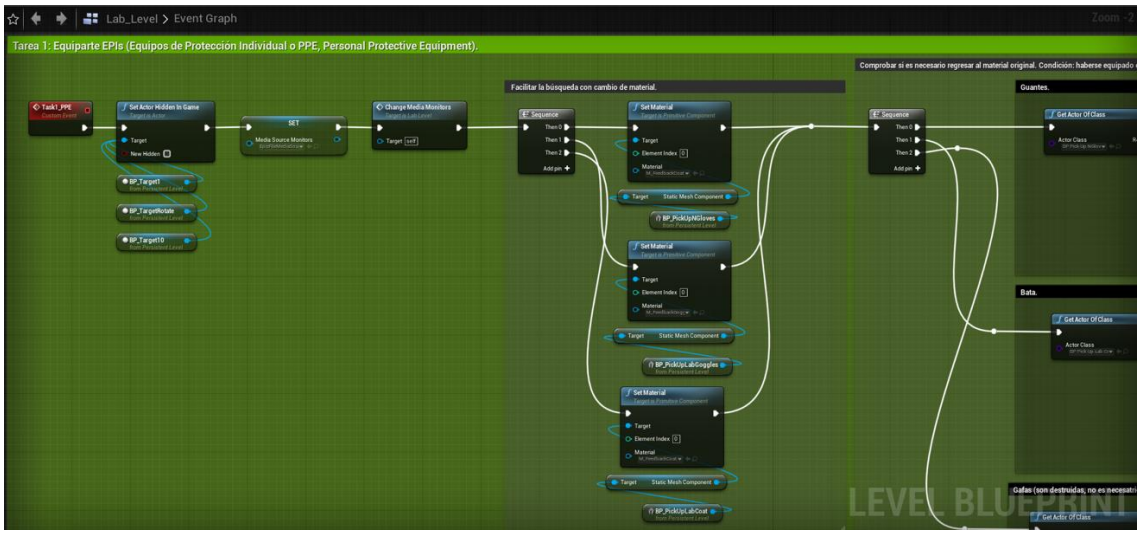
para mostrar instrucciones o vídeos, así como para mostrar la puntuación final. El diseño de la *Dra Neutrina* es original de mi amigo e ilustrador Alexis Plasencia, graduado en Bellas Artes por la ULL.

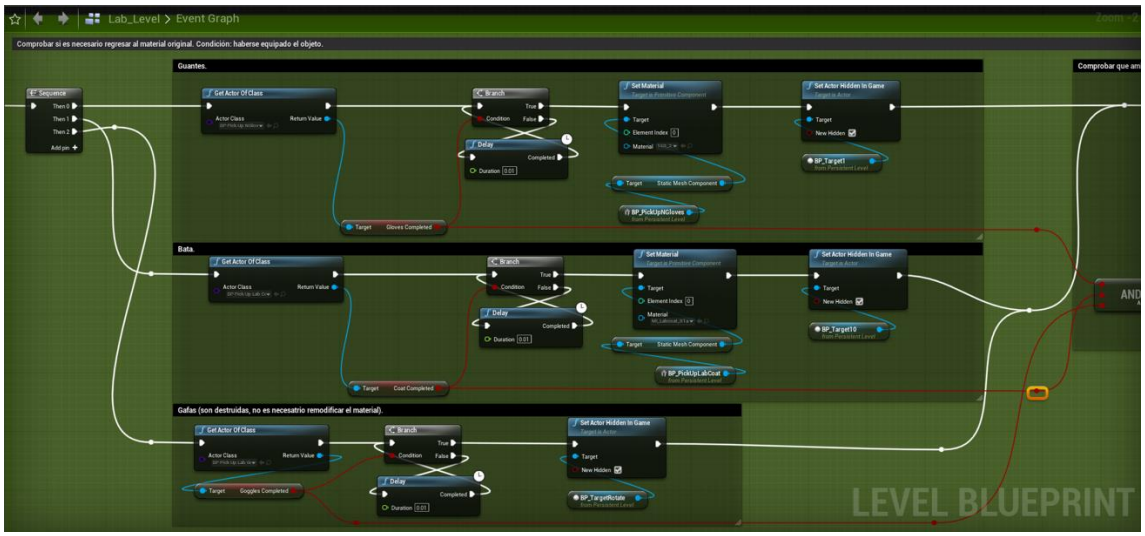
II.7.2 Tarea 1: Equiparse con los *EPIs*.

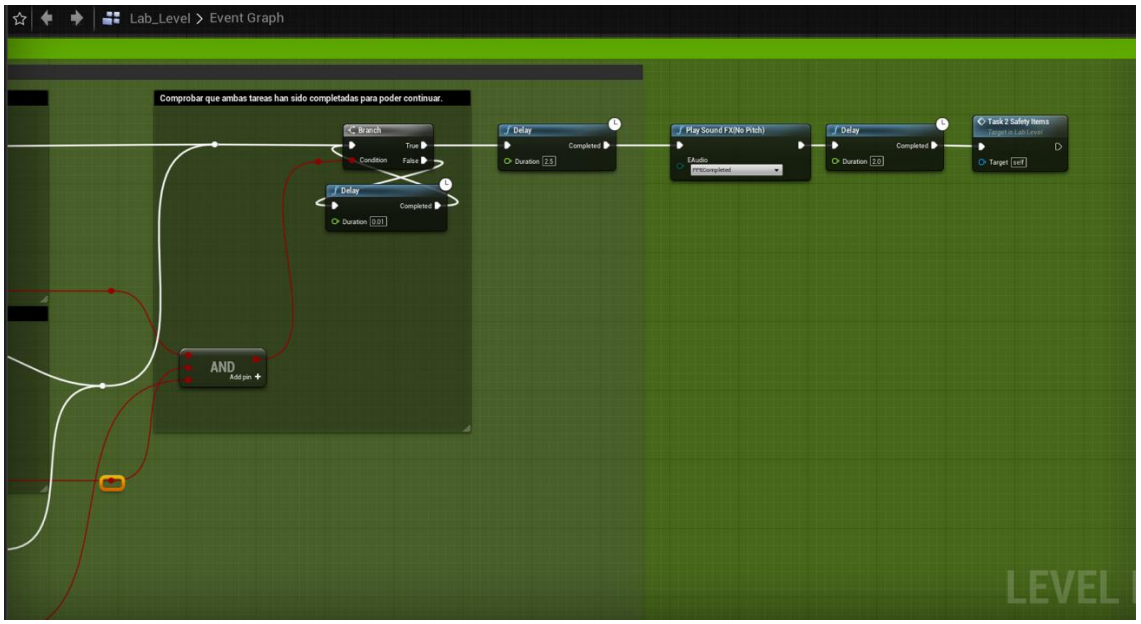
Esta tarea se divide en dos eventos. El primero de ellos se llama *Task0_Information* donde se sitúa al usuario y se le dan instrucciones para completar la tarea. Estas instrucciones se dan en forma de audios, y vídeos en los monitores del nivel haciendo uso de diferentes archivos *media source*.

Luego se pasa al evento *Task1_PPE* muestra los actores que contienen el texto del nombre de los EPIs. Se cambia el *media source* del monitor para informar de la tarea a realizar. Se cambia el material de los objetos a uno pulsante para llamar la atención del usuario y promover la interacción con ellos. Los objetos son actores interactuables del juego por tanto implementan la interfaz *BPi_PickUpActor* y su estructura base es similar a los anteriores descritos. En particular tanto guantes como bata son objetos interactuables, aunque estacionarios. Cuando se interactúa con estos objetos, se activan unas variables que permiten

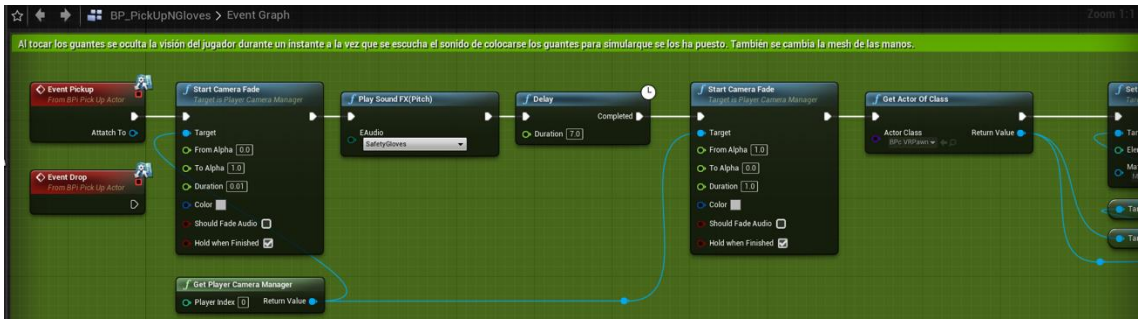
pasar al siguiente evento. Al activarse se dejan de resaltar los objetos, los cuales regresan a su material habitual, además de que se oculta el texto.

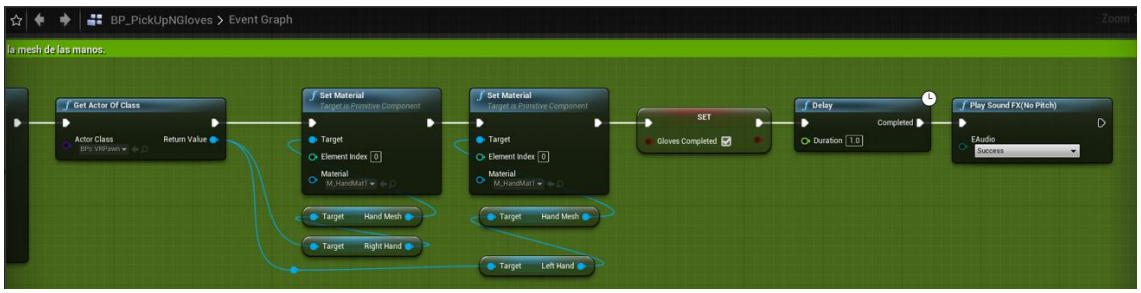




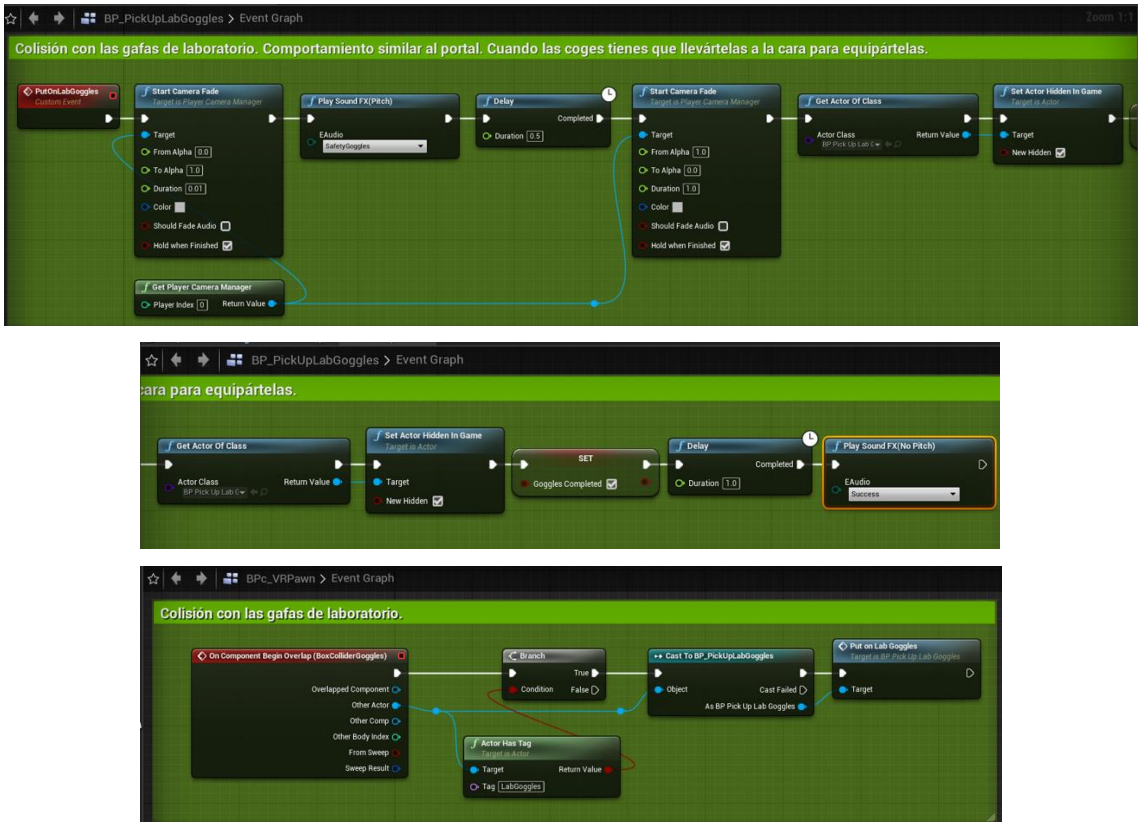


Dentro del actor que representa la bata de laboratorio llamado BP_PickUpLabCoat en el evento PickUp simplemente hay un *fade* de cámara a la vez que se reproduce un sonido de la acción de ponérsela y se la variable *bCoatCompleted* se vuelve true. Sucede lo mismo para los guantes de laboratorio, solo que estos luego cambian la apariencia de las manos. Para ello se duplicó el material base que traía *unreal* y se cambió el color base a azul para simular que el usuario lleva puestos los guantes.



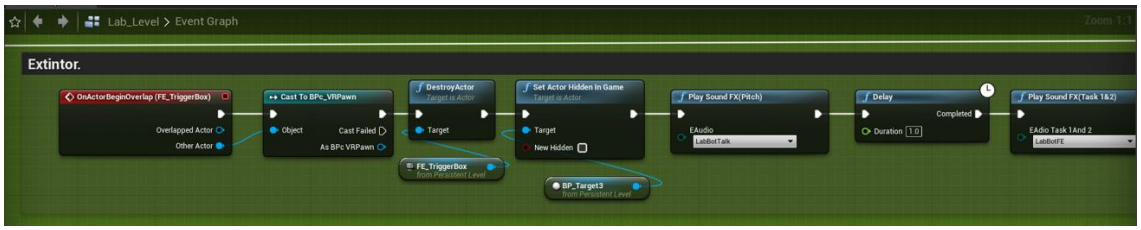


Por último, las gafas sí que tienen movilidad, por tanto, su lógica se parece un poco más a la de los portales, objeto que puede emparejarse a la mano y al soltarlo regresa a su posición original si después de tres segundos no ha sido agarrado de nuevo. Además, el usuario tiene que hacerlas colisionar con la cámara del *BPc_VRPawn* para activarlas, similar al portal. Esto no se explica, pero es bastante intuitivo y supone un factor de inmersión. La lógica, por tanto, se desarrolla en parte en el *pawn*, donde con la colisión se puede llamar al evento *PutOnLabGoggles*.

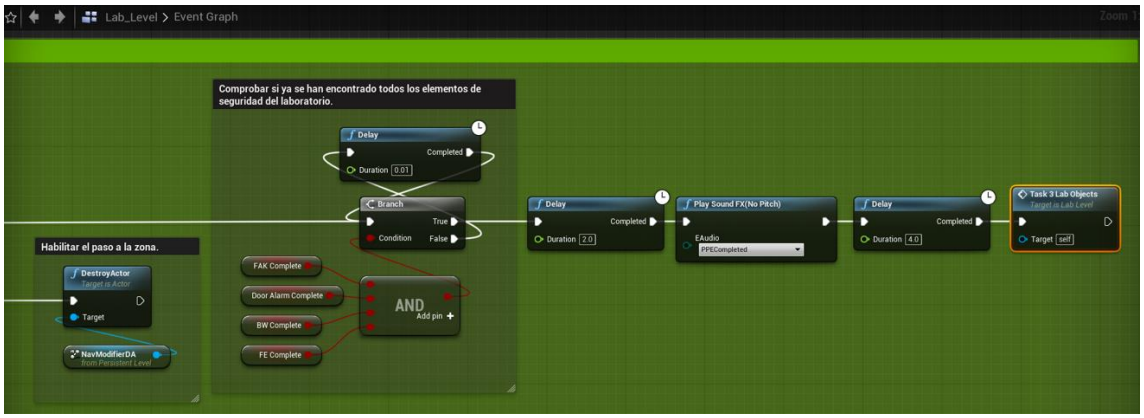


II.7.3 Tarea 2: Encontrar elementos de seguridad.

La segunda tarea consiste en encontrar los elementos de seguridad, los cuales están marcados en la escena con actores especiales. Estas zonas tienen unas *trigger box* que activan los eventos en cada objeto de seguridad. El evento es una explicación básica de lo que es cada cosa y para que sirve. Cuando se han explorado todos los objetos se prosigue con la siguiente misión. Se obvian los nodos iniciales de audios, cambios de *media source* y *delays*, y se muestra un ejemplo para el elemento extintor, puesto que la lógica es equitativa al resto.



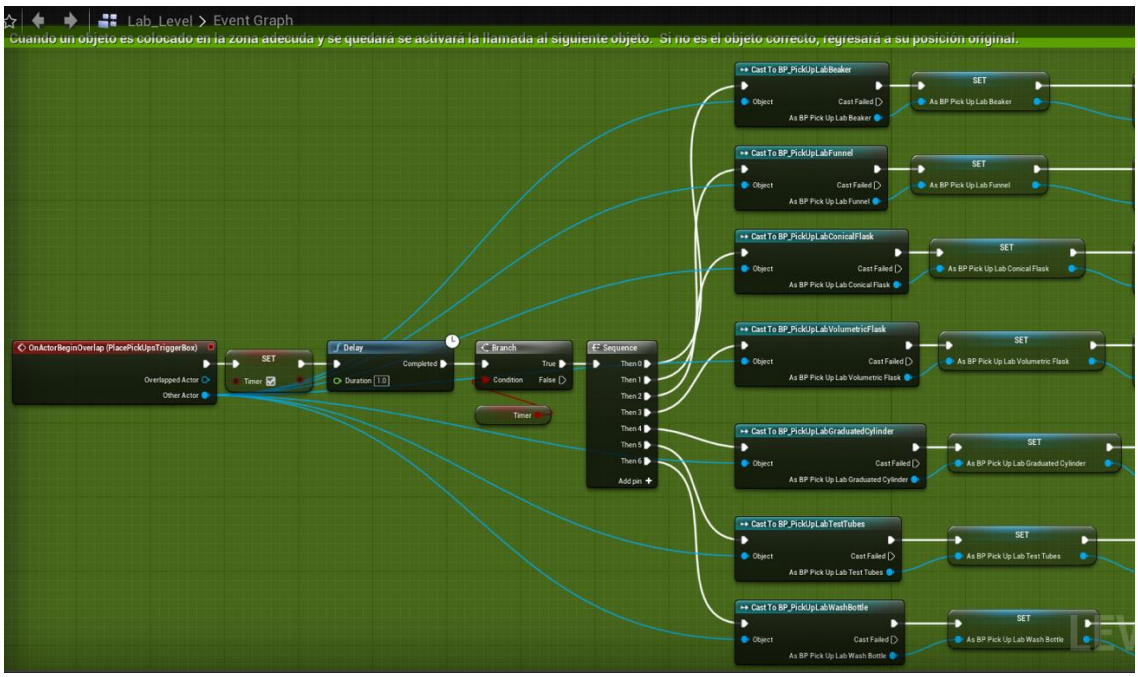
Para evitar que se activen varios objetos a la vez, solo cuando se ha terminado con uno se puede visitar el siguiente. Para ello se juega con la *navmesh* de nuevo.



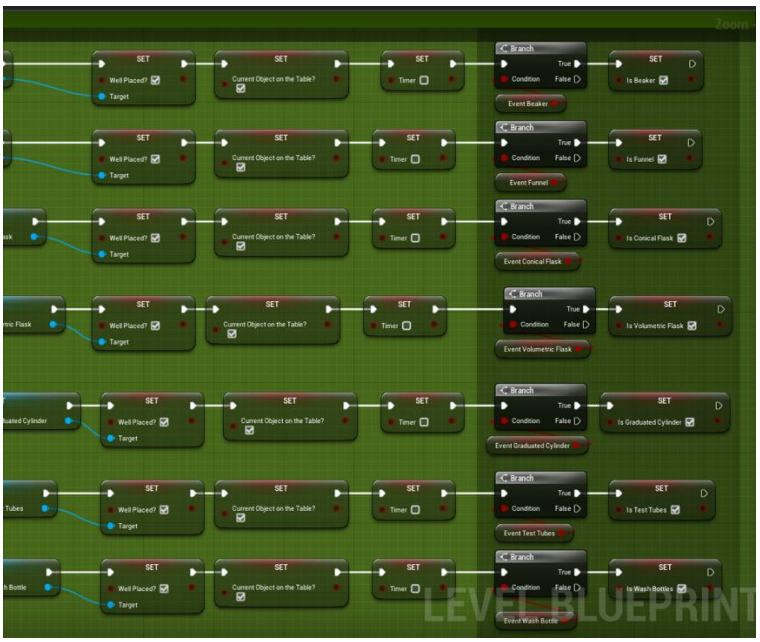
II.7.4 Tarea 3: Seleccionar material de laboratorio.

La última de las tareas es la selección de material de uso típico en el laboratorio. Este material se compone de: un vaso de precipitado, un embudo cónico de vidrio, un Erlenmeyer, una probeta, un matraz aforado, una gradilla con tubos de ensayo y un frasco lavador. El usuario tiene que coger uno a uno los objetos que se le indican y llevarlos a la mesa de trabajo. Estos son objetos interactuables movibles con una programación similar a la del portal, solo que con algunas peculiaridades. El lugar de destino es una mesa que contiene un *trigger box* que detecta si la colisión que recibe en un momento dado es correcta o no. El orden de los objetos se

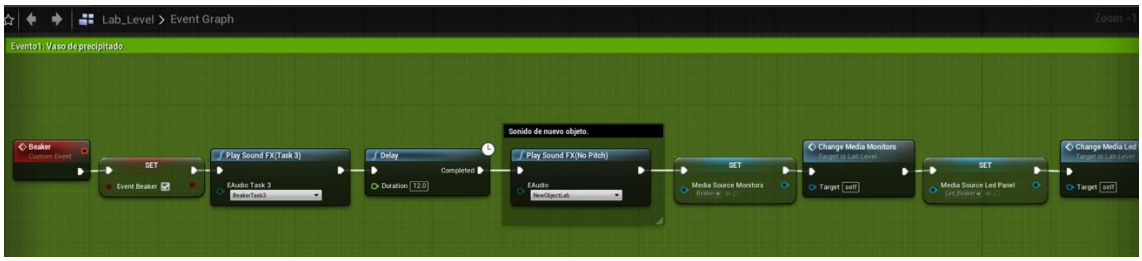
determina mediante eventos. De esta forma como el primer evento es *Beaker*, es el vaso de precipitado el primer objeto que tiene que ser llevado. Si el usuario soltara en la mesa objetivo otro objeto por error, este no sería válido, y el objeto regresaría a su posición original. Si se lleva el objeto correcto el objeto permanece en la mesa objetivo y se continúa con el evento del siguiente objeto.

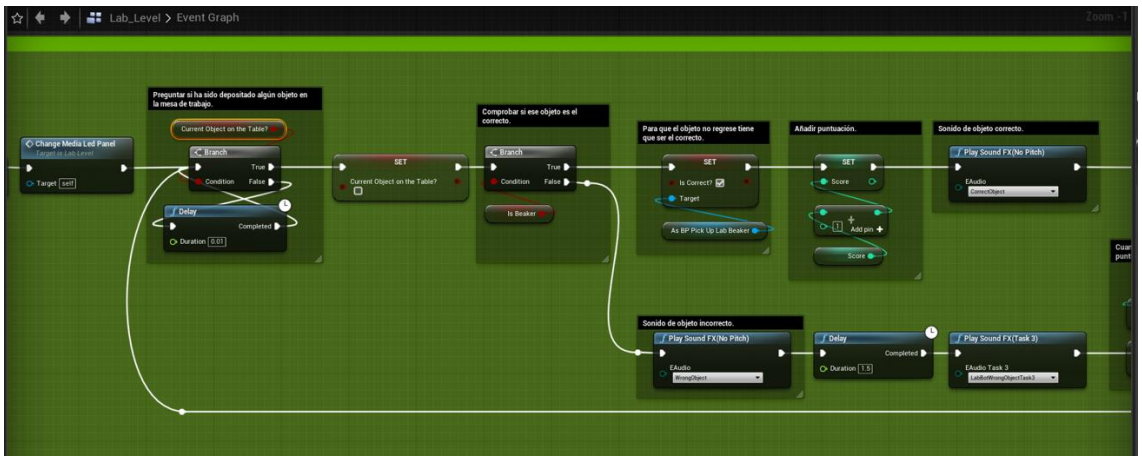


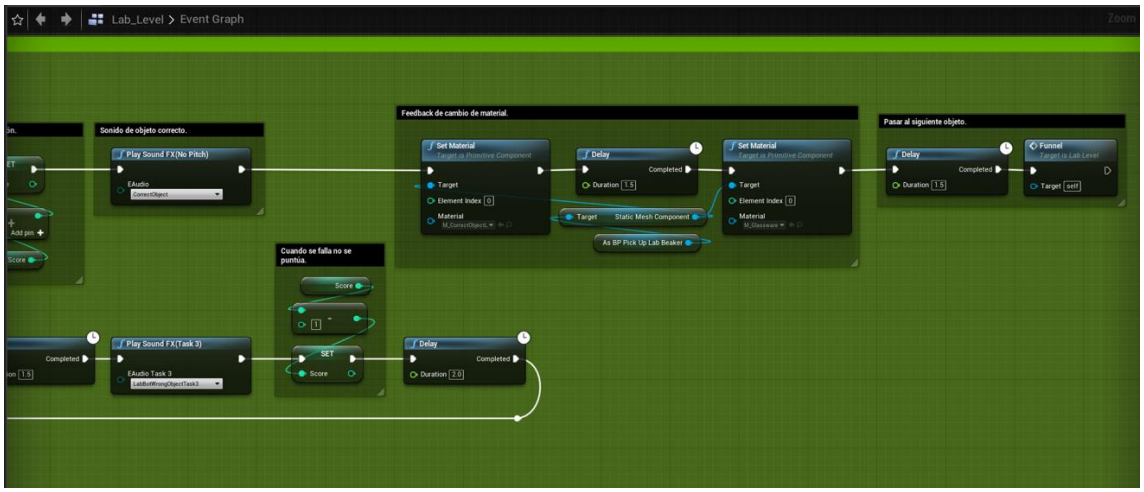
En la imagen se aprecia como el *trigger box* final realiza un *cast* a todos los objetos con un nodo *sequence* puesto, que el usuario puede llevar cualquier objeto en cualquier momento. La variable *timer* es simplemente para que la colisión no se active doblemente. El *cast* se guarda en una variable, ya que será usada posteriormente.



Lo que sucede posteriormente al *overlap* con el *trigger* es lo siguiente. Primero *bWellPlaced* se vuelve *true*, ya que es una variable que indica que el objeto está en *overlap* con la zona correcta, la zona donde se deben depositar objetos. Si el objeto no se suelta en esta zona, este simplemente volverá a su posición original. Esto está programado en el evento drop del *blueprint* de cada objeto. La siguiente variable *bCurrentObjectOnTable* sirve para preguntarse en el evento de cada objeto, si después de dar la instrucción al usuario este ha depositado algo en la mesa. Como se ha depositado algo – puesto que se ha disparado el evento *overlap*–, esta variable se convierte en *true* y el evento pasa a preguntarse si el objeto depositado es el correcto. Para que el objeto sea el correcto, la variable que lleva su mismo nombre tiene que ser *true*, y esta solo se activa en su respectivo evento. Por ejemplo, si estamos en el evento *Beaker* – el primero de ellos, donde se debe dejar el vaso de precipitado– y el usuario lleva el Erlenmeyer, la variable final b*IsBeaker* que da paso al evento del siguiente objeto no se activará. El objeto erróneo volverá a su posición inicial y el evento volverá al bucle de preguntarse si se ha depositado algún objeto en la mesa otra vez. La lógica es similar para todos los demás objetos. Todo esto va acompañado de nodos cuya misión es la de otorgar *feedback* a la acción y guiar al usuario – sonidos de error y acierto, cambio en la *media source* de los monitores con instrucciones y cambios de material de los objetos–.



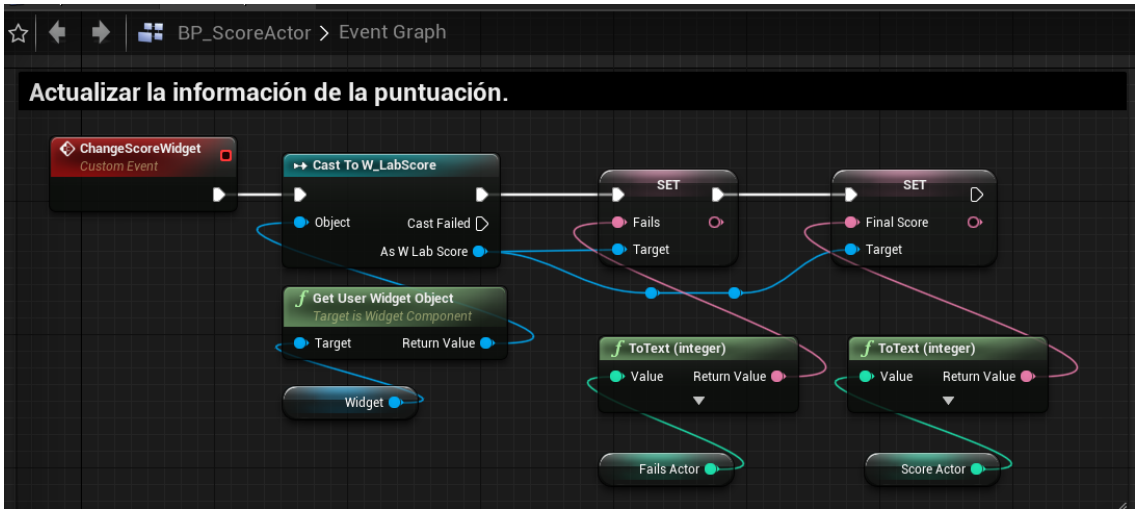




Esta lógica es similar para el resto de los objetos. Así pues, una vez se han seleccionado todos los objetos se llega al evento final del nivel llamado *FinalLabGame*. Se felicita al usuario por haber completado el nivel y se le recuerda que puede volver al *lobby* usando el portal que aparece pulsando el *trackpad* izquierdo.

II.7.5 Sistema de puntuación.

Esta última tarea lleva consigo una puntuación. El sistema de puntuación es simple y aparece en el evento de cada objeto. Simplemente se crea una variable de tipo entero llamada *Score*. Cuando se acierta seleccionando un objeto esta variable incrementa en uno. Cuando se comete un error al depositar un objeto, esta variable disminuye en uno hasta un mínimo de cero. Por lo tanto, las notas finales pueden variar de 0 a 7, el número total de objetos. Esta puntuación se crea en un *widget* llamado *W_LabScore*. Luego éste se añade a un actor llamado *BP_ScoreActor* como componente *Widget*. Este actor es mostrado en una pantalla LED del escenario al finalizar la última tarea. De esta forma el usuario puede consultar su puntuación.

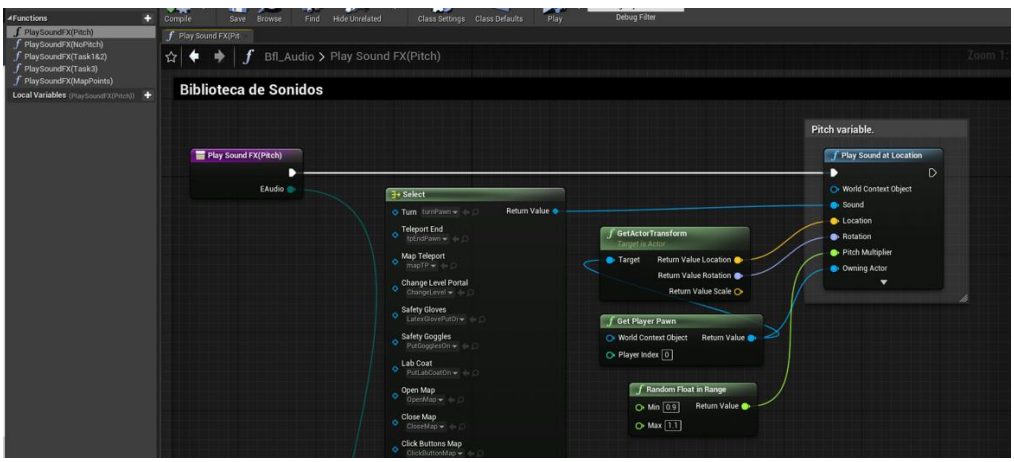


II.8 Otros aspectos de interés.

Antes de concluir este apartado de desarrollo cabe destacar brevemente algunos aspectos de interés.

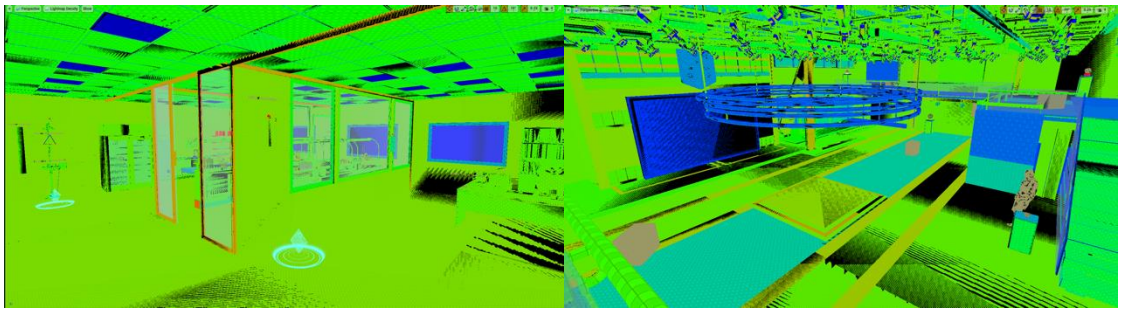
II.8.1 Audio.

En la implementación del audio del juego podemos hablar de diferentes tipos de audio. Algunos audios como los ambientales son incluidos como actores en la escena casi siempre con la opción activada de *Override Attenuation*, que permite focalizar el audio en un lugar en concreto con un radio determinado. Para los sonidos que se escuchan en el mismo *pawn*, sin embargo, se ha utilizado una *Blueprint Function Library* llamada *Bfl_Audio* con diferentes funciones y enumeraciones. Estas funciones son llamadas continuamente en numerosas ocasiones para reproducir diferentes sonidos como efectos especiales o diálogos. Cabe destacar la función *PlaySoundFX(Pitch)* que reproduce sonidos con un *pitch* aleatorio de entre 0.9 y 1.0. El objetivo es que sonidos de efectos que se usan muy a menudo como el del teletransporte o giro no suenen tan repetitivos [10], [17]–[19], [25]–[28].



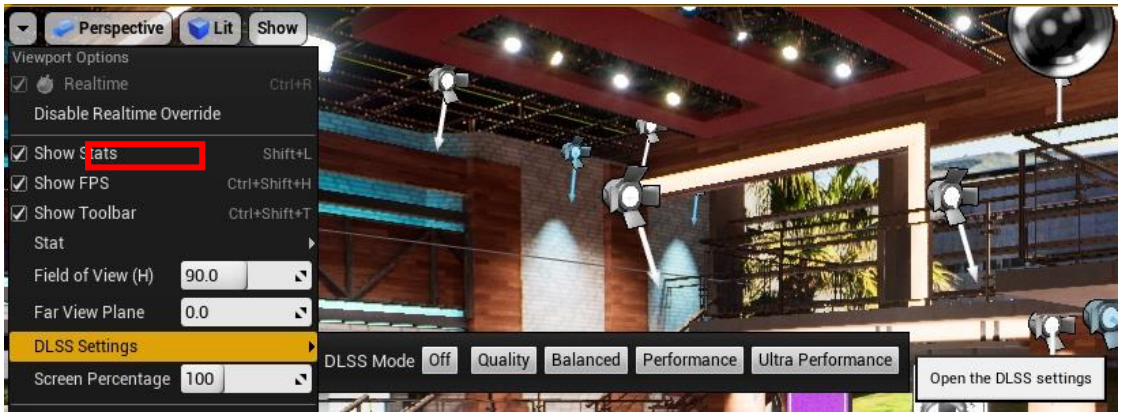
II.8.2 *Lightmaps*.

A pesar de que los escenarios del proyecto ya incluyen la iluminación, las características que se han editado en los mismos, así como la edición e inclusión de multitud de objetos externos hizo necesario revisar los *lightmaps density* para acelerar los tiempos de las *lightmass builds* y optimizar los recursos. Básicamente al seleccionar esta opción de visualización la escena se divide en tres colores: rojo, azul y verde. Cuanto más cerca del rojo está un objeto significa su *lightmap resolution* es muy alto, y cuanto más cerca del azul muy bajo. Lo ideal es que cada objeto tenga un *lightmap resolution* cercano al verde. Para ello se puede seleccionar el objeto y en el panel de detalles seleccionar *Override LightMap Resolution* y ajustar al valor óptimo –relacionado con el tamaño de este–.



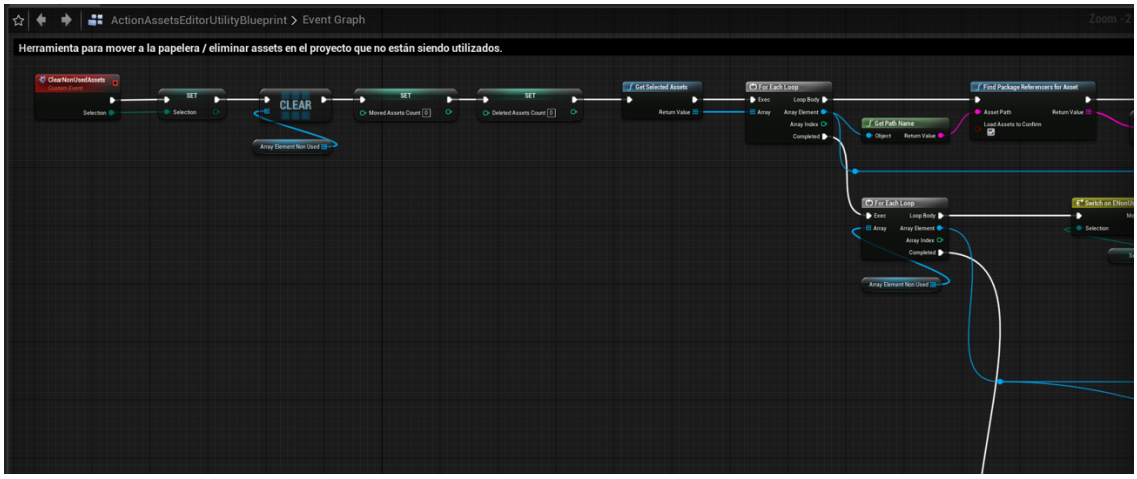
II.8.3 *Nvidia DLSS.*

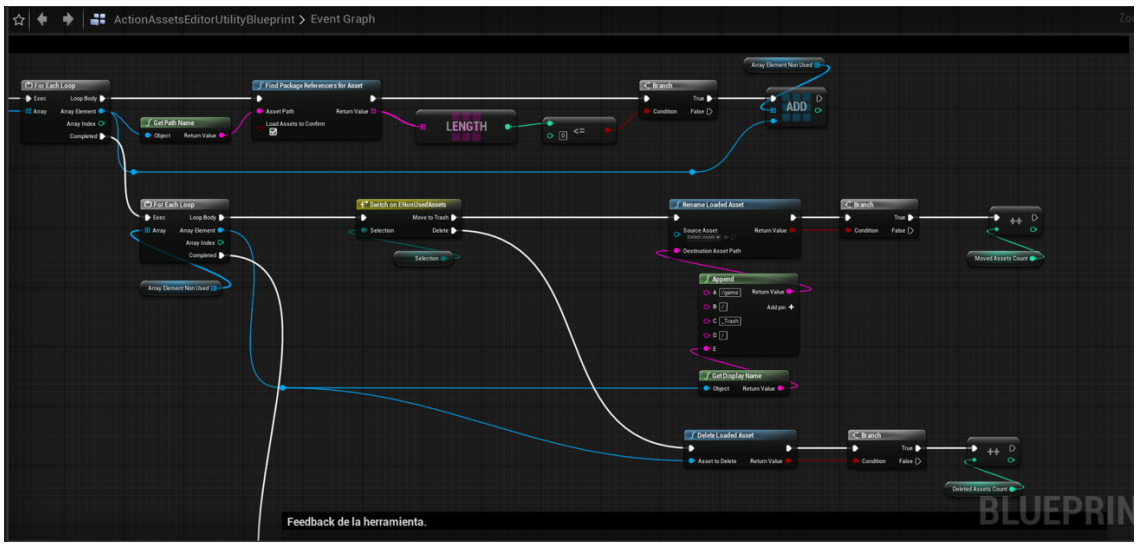
El objetivo del uso de este plugin es obtener la máxima representación visual por un coste de rendimiento bajo. El supersampleado es una técnica que permite mejorar los gráficos que se ven en pantalla con un nivel de detalle mucho más alto. DLSS también es una técnica para mejorar gráficos, pero controlada por una AI. Además, mejorar la calidad de imagen mediante este proceso evita que el motor lo haga mediante otros métodos más costosos. Por ello en este proyecto se ha descargado e instalado el plugin para ue4. Para comprobar los resultados, se pueden probar los diferentes niveles del plugin desde el panel *DLSS Settings* junto con el ajuste del *antialiasing* y mostrar los *fps* en pantalla resultantes [29].

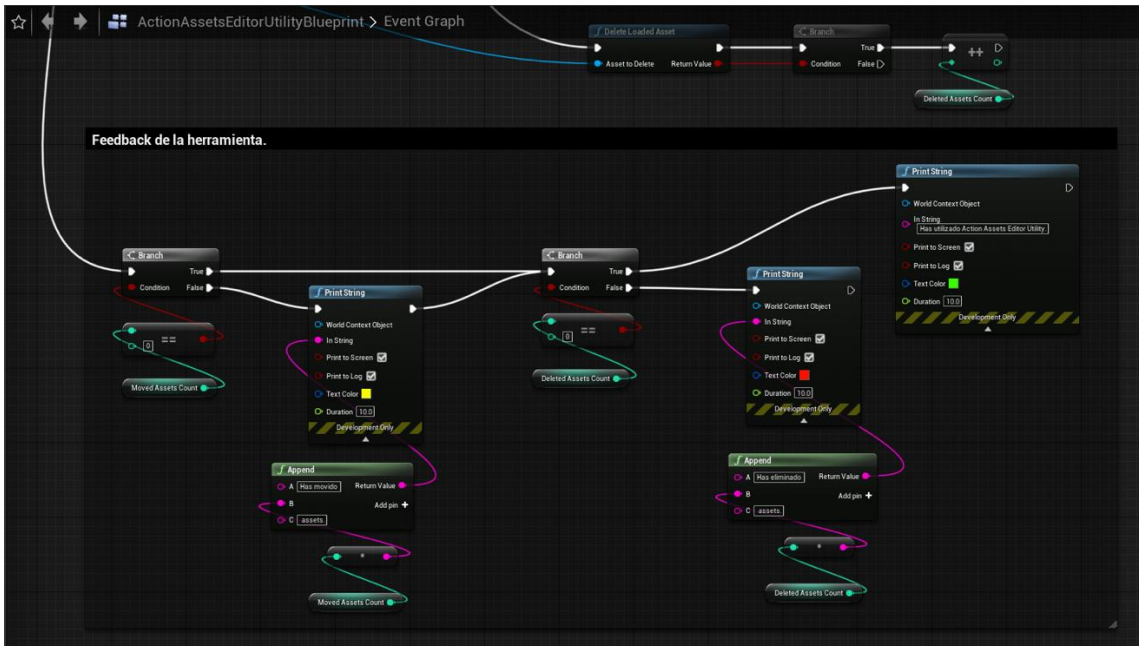


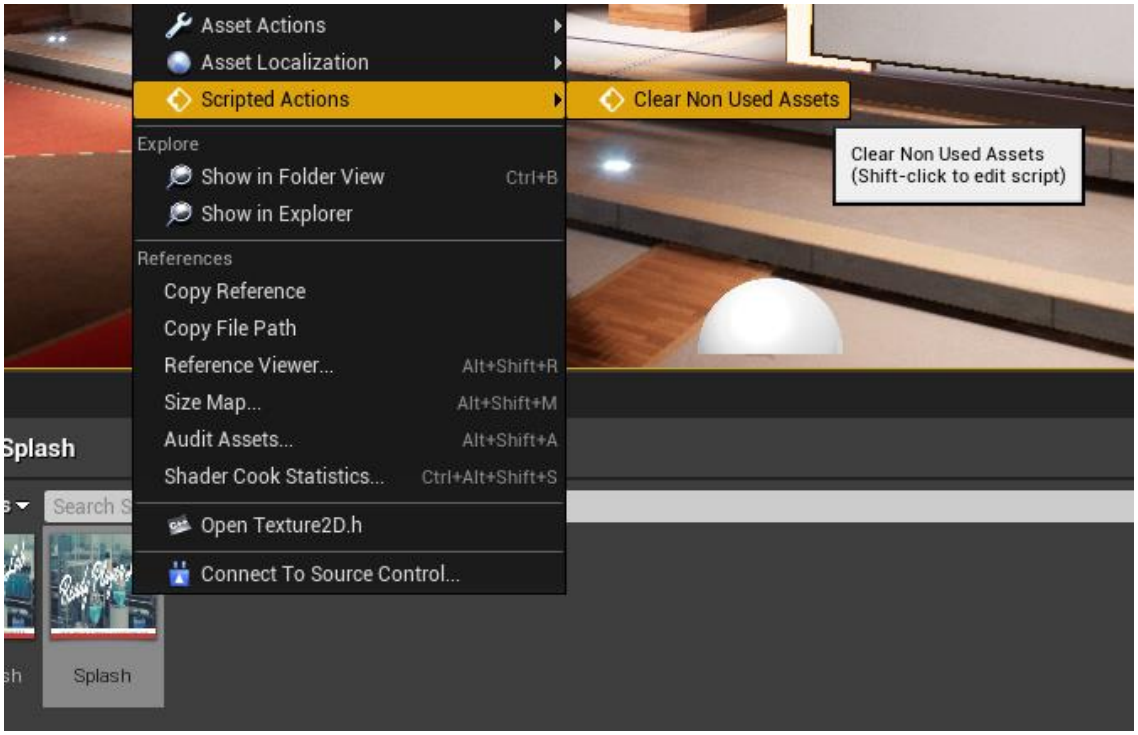
II.8.4 Desarrollo de una herramienta para borrar *assets* residuales.

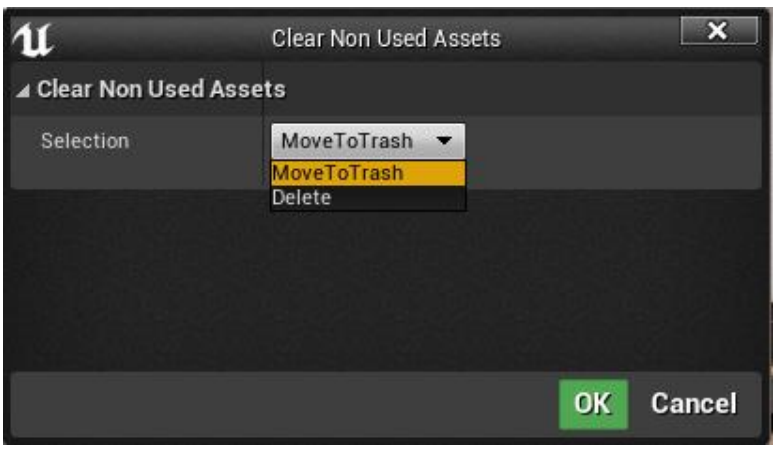
Durante el proyecto han quedado atrás o han sido descartado o migrados por error numerosos *assets*. Dado que el motor no presenta ninguna herramienta de serie para gestiones de este tipo, se ha desarrollado una propia para este proyecto. Para ello hay que instalar el *plugin Editor Scripting Utilities*. Se crea una carpeta llamada *Blutilities* y dentro se crea un nuevo *Editor Utilitie Blueprint* del tipo *AssetActionUtilitie* –ya que está destinado a realizar acciones sobre *assets*– llamado *ActionAssetsEditorUtilityBlueprint*. La idea es que si un *asset* determinado no está siendo utilizado por ningún otro se elimine. Esta información se puede consultar manualmente seleccionando la opción *Reference Viwer*, pero el objetivo es que el motor lo compruebe por sí mismo para un grupo de archivos predeterminado por el desarrollador. Como borrar es algo irreversible se ha programado la acción para elegir entre *MoveToTrash* y *Delete*. La primera acción guarda los *assets* no usados en una carpeta aparte. La segunda, en cambio, los elimina directamente [30].






ActionAssetsEditorUtilityBlueprint > Event Graph
Feedback de la herramienta.


Asset Actions
Asset Localization
Scripted Actions
Explore
Show in Folder View Ctrl+B
Show in Explorer
References
Copy Reference
Copy File Path
Reference Viewer... Alt+Shift+R
Size Map... Alt+Shift+M
Audit Assets... Alt+Shift+A
Shader Cook Statistics... Ctrl+Alt+Shift+S
Open Texture2D.h
Connect To Source Control...
Clear Non Used Assets
Clear Non Used Assets
(Shift-click to edit script)
Splash
Splash


Clear Non Used Assets
Clear Non Used Assets
Selection
MoveToTrash
MoveToTrash
Delete
OK
Cancel

Capítulo III. Resumen y Conclusiones.

En este proyecto llamado READY PLAYER LAB se ha desarrollado un prototipo de videojuego en realidad virtual para PC usando el motor gráfico de *Unreal Engine 4* y el *headset* de *Windows Mixed Reality* de Acer. Se expone que:

1. Se ha creado un proyecto desde cero para la realidad virtual.
2. Se han optimizado el motor para conseguir el mejor rendimiento en realidad virtual.
3. Se ha desarrollado un mapa de navegación virtual 3D.
4. Se han desarrollado una escena como tutorial.
5. Se ha desarrollado un nivel principal con minijuegos.
6. Se ha desarrollado un laboratorio de química orgánica.
7. Se ha introducido un sistema de puntuación.

Para ello se han utilizado diferentes conocimientos y habilidades desarrolladas durante el máster de Desarrollo de Videojuegos de la Universidad de La Laguna. En los siguientes enlaces se puede visualizar contenido como *gameplays* y comentarios:

https://www.youtube.com/watch?v=SAz18tI-wjY&t=522s

https://www.youtube.com/channel/UCLxEjxcbZ86Er5RSn7N9_IQ/featured

Summary and Conclusions

In this project called READY PLAYER LAB, a virtual reality videogame prototype for PC has been developed using the Unreal Engine 4 graphics engine and Acer's Windows Mixed Reality headset.

1. A project has been created from scratch for virtual reality.
2. The engine has been optimized to achieve the best performance in virtual reality.
3. A 3D virtual navigation map has been developed.
4. A tutorial scene has been developed.
5. A main level with minigames has been developed.
6. An organic chemistry laboratory has been developed.
7. A scoring system has been developed.

The knowledge and skills reached during the Master's Degree in VideoGame Development at the University of La Laguna have been used to build this project. In the following links there is content about Ready Player Lab as gameplays and commentary tracks:

https://www.youtube.com/watch?v=SAz18tI-wjY&t=522s

https://www.youtube.com/channel/UCLxEjxcbZ86Er5RSn7N9_IQ/featured

Bibliografía

[1] K. Mack and R. Ruud, “Unreal Engine 4 virtual reality projects : build immersive, real-world VR applications using UE4, C++, and unreal blueprints,” p. 613.

[2] J. Plowman, *Unreal Engine Virtual Reality Quick Start Guide*. 2019.

[3] M. McCaffrey, *Unreal Engine VR Cookbook*, vol. 17, no. 2. 2017.

[4] “HoloLAB Champions | Schell Games.” https://www.schellgames.com/games/hololab-champions (accessed Sep. 07, 2021).

[5] R. M. Broyer, K. Miller, S. Ramachandran, S. Fu, K. Howell, and S. Cutchin, “Using Virtual Reality to Demonstrate Glove Hygiene in Introductory Chemistry Laboratories,” *J. Chem. Educ.*, vol. 98, no. 1, pp. 224–229, Jan. 2020, doi: 10.1021/ACS.JCHEMED.0C00137.

[6] R. Dai, J. A. Laureanti, M. Kopelevich, and P. L. Diaconescu, “Developing a Virtual Reality Approach toward a Better Understanding of Coordination Chemistry and Molecular Orbitals,” *J. Chem. Educ.*, vol. 97, no. 10, pp. 3647–3651, Oct. 2020, doi: 10.1021/ACS.JCHEMED.0C00469.

[7] A. Rychkova, A. Korotkikh, A. Mironov, A. Smolin, N. Maksimenko, and M. Kurushkin, “Orbital Battleship: A Multiplayer Guessing Game in Immersive Virtual Reality,” *J. Chem. Educ.*, vol. 97, no. 11, pp. 4184–4188, Nov. 2020, doi: 10.1021/ACS.JCHEMED.0C00866.

[8] “NC State VR Organic Chemistry Labs.” https://sites.google.com/ncsu.edu/ncstatevrorganicchemistrylabs/home (accessed Sep. 07, 2021).

[9] C. L. Dunnagan, D. A. Dannenberg, M. P. Cuales, A. D. Earnest, R. M. Gurnsey, and M. T. Gallardo-Williams, “Production and Evaluation of a Realistic Immersive Virtual Reality Organic Chemistry Laboratory Experience: Infrared Spectroscopy,” *J. Chem. Educ.*, vol. 97, no. 1, pp. 258–262, Jan. 2019, doi: 10.1021/ACS.JCHEMED.9B00705.

[10] “Micrófonos Blue - Yeti.” https://www.bluemic.com/es-es/products/yeti/ (accessed Sep. 07, 2021).

[11] “EULA - Unreal Engine.” https://www.unrealengine.com/en-US/eula/publishing

(accessed Sep. 07, 2021).

[12] “blender.org - Home of the Blender project - Free and Open 3D Creation Software.” https://www.blender.org/ (accessed Sep. 07, 2021).

[13] “Maya Software | Get Prices & Buy Official Maya 2022 | Autodesk.” https://www.autodesk.com/products/maya/overview (accessed Sep. 07, 2021).

[14] “Software 3ds Max | Obtener precios y comprar el producto oficial 3ds Max 2022.” https://www.autodesk.es/products/3ds-max/overview (accessed Sep. 07, 2021).

[15] “Official Adobe Photoshop | Photo & Design Software.” https://www.adobe.com/products/photoshop.html (accessed Sep. 07, 2021).

[16] “Canva | Una herramienta de diseño gráfico en línea | Prueba Canva Pro.” https://www.canva.com/es_es/q/pro/?v=2&utm_source=google_sem&utm_medium=cpc&utm_campaign=es_es_all_pro_rev_conversion_branded-tier1_em&utm_term=es_es_all_pro_rev_conversion_Branded_Tier1_Canva_EM&gclid=EAIaIQobChMIhpL4jPPs8gIV43xvBB1qEghcEAAYASAAEgJ3VPD_B (accessed Sep. 07, 2021).

[17] “Audacity, editor de audio libre - Audacity.es - Descargar Audacity para PC y Mac.” https://audacity.es/ (accessed Sep. 07, 2021).

[18] “VB-Audio VoiceMeeter.” https://vb-audio.com/Voicemeeter/ (accessed Sep. 07, 2021).

[19] “Free Real Time Voice Changer & Modulator - Voicemod.” https://www.voicemod.net/ (accessed Sep. 07, 2021).

[20] “Open Broadcaster Software | OBS.” https://obsproject.com/es (accessed Sep. 07, 2021).

[21] “Crear o editar vídeo en Windows 10.” https://support.microsoft.com/es-es/windows/crear-o-editar-vídeo-en-windows-10-53b3e8f8-a85f-172f-4efd-2e66afccf43e (accessed Sep. 07, 2021).

[22] “Instalación de las herramientas - Mixed Reality | Microsoft Docs.” https://docs.microsoft.com/es-es/windows/mixed-reality/develop/install-the-tools?tabs=unreal (accessed Sep. 07, 2021).

[23] “Documentación de VR de Windows Mixed Reality - Enthusiast Guide | Microsoft

Docs.” https://docs.microsoft.com/es-es/windows/mixed-reality/enthusiast-guide/ (accessed Sep. 07, 2021).

[24] “Perforce Software | Development Tools For Innovation at Scale.” https://www.perforce.com/ (accessed Sep. 07, 2021).

[25] “3D Map Navigation in VR - Unreal Online Learning Course - Unreal Engine.” https://www.unrealengine.com/en-US/onlinelearning-courses/3d-map-navigation-in-vr (accessed Sep. 07, 2021).

[26] “Freesound - Sounds browse.” https://freesound.org/browse/ (accessed Sep. 07, 2021).

[27] “Free Sound Effects to Download | ZapSplat.” https://www.zapsplat.com/ (accessed Sep. 07, 2021).

[28] “VR Development with Oculus and Unreal Engine - Unreal Engine.” https://www.unrealengine.com/en-US/onlinelearning-courses/vr-development-with-oculus-and-unreal-engine (accessed Sep. 07, 2021).

[29] “Get Started | NVIDIA Developer.” https://developer.nvidia.com/dlss-getting-started (accessed Sep. 07, 2021).

[30] “Scripting the Editor using Blueprints | Unreal Engine Documentation.” https://docs.unrealengine.com/4.26/en-US/ProductionPipelines/ScriptingAndAutomation/Blueprints/ (accessed Sep. 07, 2021).

Bibliografía complementaria

"Unreal Engine 4 Virtual Reality Projects | Packt." https://www.packtpub.com/product/unreal-engine-4-virtual-reality-projects/9781789132878 (accessed Sep. 07, 2021).

E. Pastor Tejera, I. López Bazzochi, P. Esparza Ferrera, J.L. Rodríguez Marrero, and P. Lorenzo Luis, *Experimentación en Química: Principios y Prácticas.* Santa Cruz de Tenerife.

"Alexis Plasencia (@alexjrworks) • Instagram photos and videos." https://www.instagram.com/alexjrworks/?hl=en (accessed Sep. 07, 2021).Comunicación Visual

C. L. Dunnagan and M. T. Gallardo-Williams, "Overcoming Physical Separation During COVID-19 Using Virtual Reality in Organic Chemistry Laboratories," *J. Chem. Educ.*, vol. 97, no. 9, pp. 3060–3063, Sep. 2020, doi: 10.1021/ACS.JCHEMED.0C00548.S.L., 2001.

"Experimentos de química orgánica - Editorial Reverté S.A." https://www.reverte.com/libro/experimentos-de-quimica-organica_81425/ (accessed Sep. 07, 2021).

J. Dixon, S. Markidis, C. Luo, and J. Reynolds, "Three-dimensional, virtual, game-like environments for education and training," *undefined*, 2007.

C. Rytych, L. Conley, H. Wu, and Rizwan-uddin, "Game-like environments for nuclear engineering education using GECK," *2010 1st Int. Nucl. Renew. Energy Conf. INREC'10*, 2010, doi: 10.1109/INREC.2010.5462582.

D. Probst and J.-L. Reymond, "Exploring DrugBank in Virtual Reality Chemical Space," *J. Chem. Inf. Model.*, vol. 58, no. 9, pp. 1731–1735, Sep. 2018, doi: 10.1021/ACS.JCIM.8B00402.

M. O'Connor *et al.*, "Sampling molecular conformations and dynamics in a multiuser virtual reality framework," *Sci. Adv.*, vol. 4, no. 6, Jun. 2018, doi: 10.1126/SCIADV.AAT2731.

Jessica Morrison, "Will chemists tilt their heads for virtual reality?," *C&EN Glob. Enterp.*, vol. 94, no. 14, pp. 22–23, Apr. 2016, doi: 10.1021/CEN-09414-EDUC.

J. B. Ferrell *et al.*, "Chemical Exploration with Virtual Reality in Organic Teaching Laboratories," *J. Chem. Educ.*, vol. 96, no. 9, pp. 1961–1966, Sep. 2019, doi: 10.1021/ACS.JCHEMED.9B00036.

"Immersive stereochemistry: Using virtual reality (VR) to understand chirality with undergraduate students | Morressier." https://www.morressier.com/article/immersive-stereochemistry-using-virtual-reality-vr-understand-chirality-undergraduate-students/5e73d6ce139645f83c22ae6e (accessed Sep. 07, 2021).

M. A. S. Lima *et al.*, "Game-Based Application for Helping Students Review Chemical Nomenclature in a Fun Way," *J. Chem. Educ.*, vol. 96, no. 4, pp. 801–805, Apr. 2019, doi: 10.1021/ACS.JCHEMED.8B00540.

H. A. Gandhi, S. Jakymiw, R. Barrett, H. Mahaseth, and A. D. White, "Real-Time Interactive Simulation and Visualization of Organic Molecules," 2020, doi: 10.1021/acs.jchemed.9b01161.

M. Norrby, C. Grebner, J. Eriksson, and J. Boström, "Molecular Rift: Virtual Reality for Drug Designers," *J. Chem. Inf. Model.*, vol. 55, no. 11, pp. 2475–2484, Nov. 2015, doi: 10.1021/ACS.JCIM.5B00544.

S. Tzima, G. Styliaras, and A. Bassounas, "Augmented reality applications in education: Teachers point of view," *Educ. Sci.*, vol. 9, no. 2, Jun. 2019, doi: 10.3390/EDUCSCI9020099.

M. Poyade, C. Eaglesham, J. Trench, and M. Reid, "A Transferable Psychological Evaluation of Virtual Reality Applied to Safety Training in Chemical Manufacturing," *ACS Chem. Heal. Saf.*, vol. 28, no. 1, pp. 55–65, Jan. 2021, doi: 10.1021/ACS.CHAS.0C00105.

D. A. Sheverev and I. N. Kozlova, "The development of a virtual laboratory based on unreal engine 4," *CEUR Workshop Proc.*, vol. 2212, pp. 98–104, 2018, doi: 10.18287/1613-0073-2018-2212-98-104.


READY
PLAYER
LAB