PROYECTO - REALIDAD VIRTUAL

METÁFORAS DE MOVIMIENTO EN GRAVEDAD 0 CON RETROALIMENTACIÓN HÁPTICA

Andrés González Antohi

Rodrigo Hervada Llahosa

Marco Sánchez Nishimura

Contenido

Introducción

Idea inicial

Cuando en la presentación de los proyectos se nos explicó el material disponible para realizarlos, nos llamó mucho la atención el chaleco háptico y empezaron a surgir ideas de cómo usarlo.

Posteriormente, surgieron entre los integrantes tres vertientes para el proyecto, que se centraban en el uso del chaleco háptico, una alternativa a las metáforas de movimiento ya existentes y por último el uso de portales realistas.

El resultado final es este proyecto. Es un simulador/videojuego de realidad virtual en el espacio, en gravedad 0 en el que se llevará equipado un *jetpack* y se tendrá que pasar por 3 niveles distintos. El primero de ellos es una serie de pasadizos donde se aprenderá a usar el *jetpack* mientras se tendrá que esquivar obstáculos y lidiar con unas bolas que serán disparadas. Posteriormente, a través de un portal, se llegará al exterior de la nave espacial, en el que se tendrá que resolver un par de sencillos puzles moviéndose alrededor de la nave. Al terminar, un nuevo portal permitirá llegar a la ultima sección en la que se obtendrá una pistola de portales con la que resolver tres puzles más.

Chaleco háptico

Problemas

El chaleco háptico resulta un reto, no por la complejidad de su uso, que es prácticamente nula, sino por la poca información que hay respecto a su uso, sus constantes bugs y archivos que se corrompen sin razón aparente.

No solo hay poca información, sino que, en medio del desarrollo, se actualizaron las distintas aplicaciones (*bHaptics Player, bHaptics Developer y bHaptics Designer*), lo cual corrompió algunas vibraciones, provocando así que se tuviera que empezar todo desde cero, volviendo obsoletos algunos de los pocos tutoriales funcionales y cambiando la forma de trabajar (principalmente en *bHaptics Designer*) sin ningún tipo de documentación que facilitase el aprendizaje, teniendo que aprender mediante la prueba y el error.

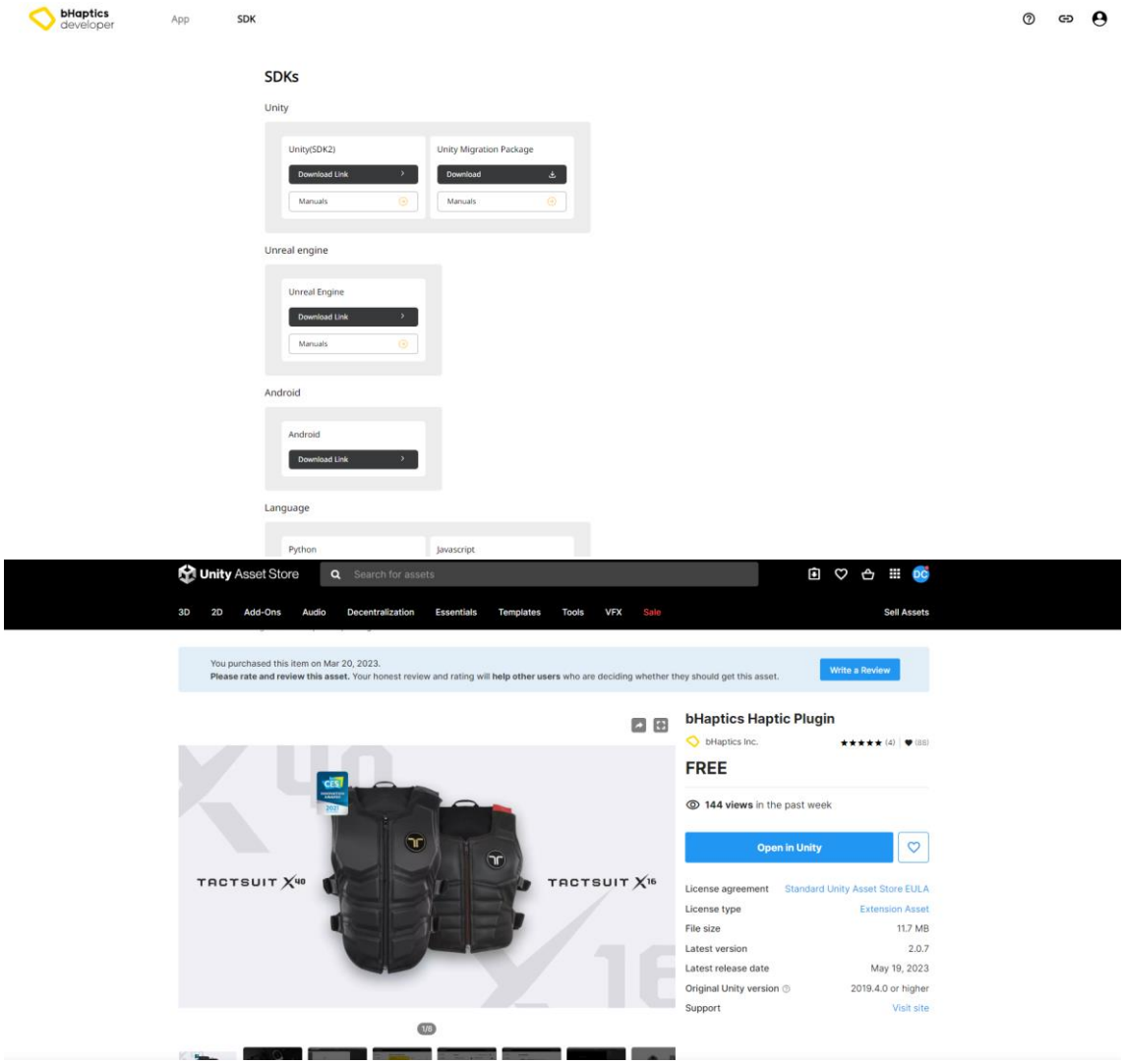
El otro problema principal consiste en la creación de vibraciones con sentido. *bHaptics Designer* contiene *feedback* visual para poder ver las vibraciones a la vez que se sienten, esto, de forma inconsciente, ayuda a sentir mejor los motores y tener una mejor experiencia, la cual se pierde si no se ha tenido esta visión de las vibraciones, lo que provoca que muchas vibraciones distintas se sientan igual. Siguiendo con la propia sensación de las vibraciones, a pesar de todos los distintos niveles de intensidad, es casi imposible diferenciar que ha habido un cambio en esta, diferenciar una vibración al x1.5 del x3 se puede volver una tarea complicada.

El diseño del chaleco tampoco es de mucha ayuda, a partir de cierta intensidad la cremallera empieza a rebotar contra el chaleco, provocando ruido, rompiendo bastante la inmersión. Además, la falta de motores en los costados, implica que es imposible crear una vibración coherente que pase de, por ejemplo, de espalda a frente, o una simple colisión en el costado.

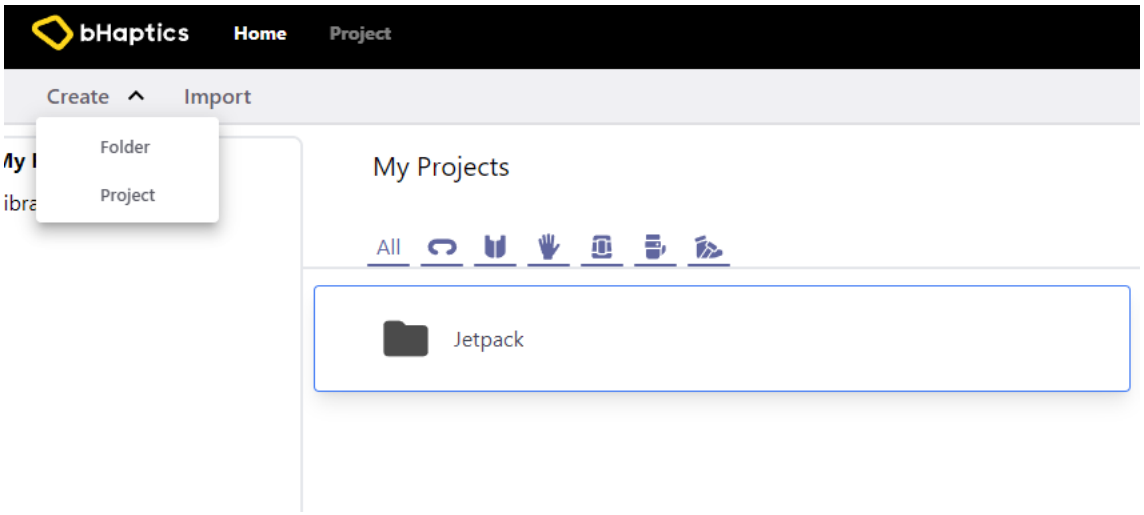
Tutorial y proceso de uso

En primer lugar, es necesario instalar *bHaptics Player* en el ordenador y crear una cuenta. Dentro de la aplicación, es necesario conectar el chaleco por *Bluetooth*, donde se podrá realizar un "*Feedback Test*" para probar los distintos motores y la intensidad de la vibración.

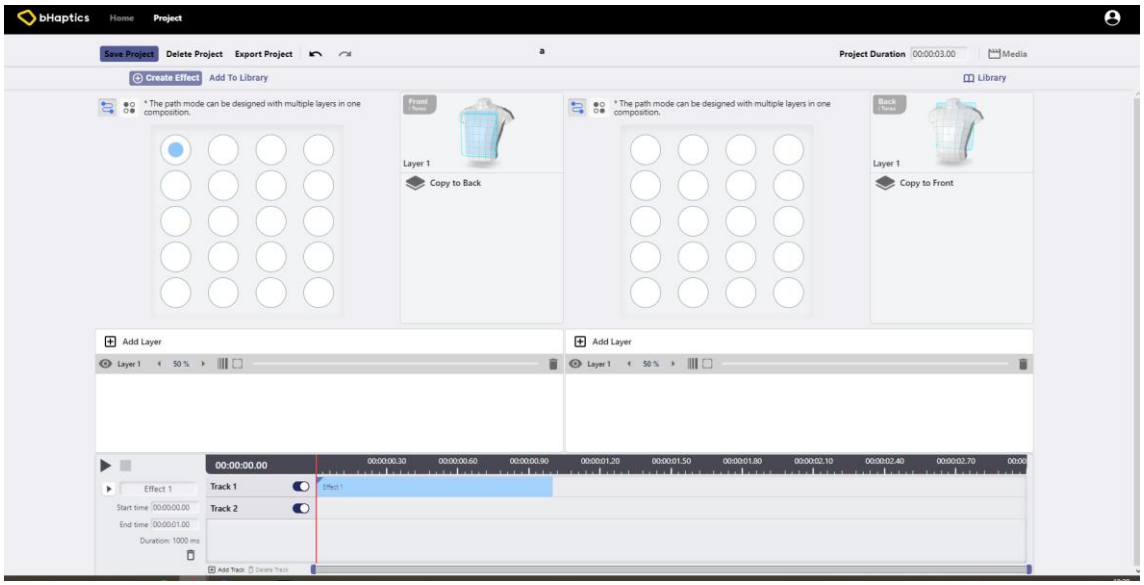
Yendo a *bHaptics Developer Portal*, en el apartado "*SDK*", selecciona la opción de descargar, que llevará a la *Asset Store* de *Unity* para poder importarlo al proyecto más adelante.



A continuación, en la pestaña de "*App*", crea un proyecto, dentro podrás importar eventos. Los eventos se crean en *bHaptics Desginer*. Para ello, dale a "*Create*", "*Project*", y selecciona el dispositivo para el que se quiera crear el evento.



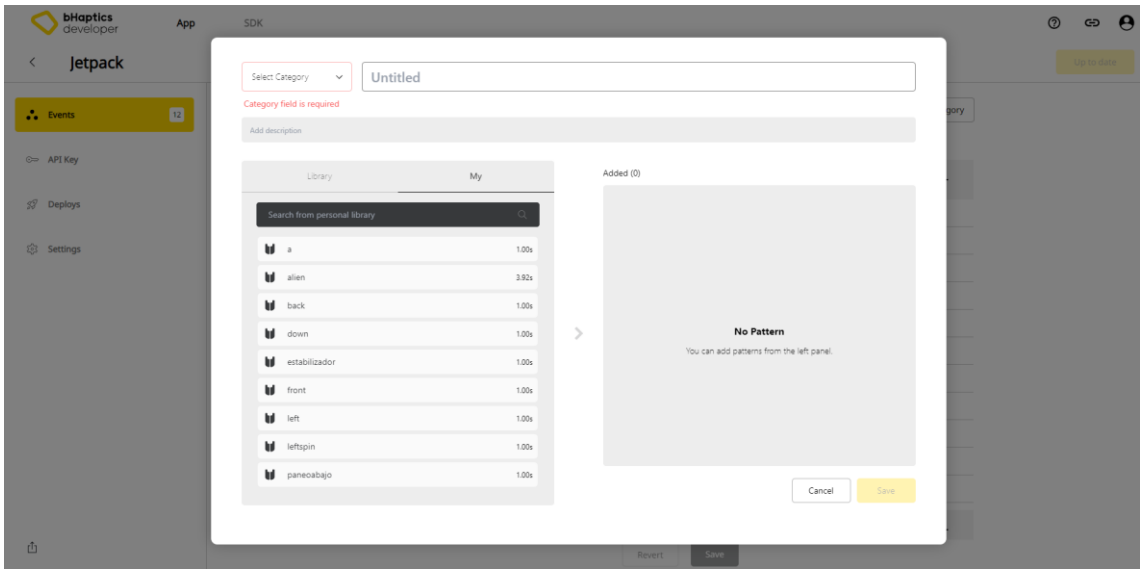
Una vez dentro del proyecto, es necesario darle a crear efecto para poder crear las pistas de vibración, que pueden ser por puntos o por recorrido. Cada parte (frente y espalda) tienen distintas capas en las que se pueden crear distintas vibraciones. Es necesario, una vez hecho el proyecto (que puede tener varios efectos distintos), guardarlo no solo con la opción "*Save Project*" sino también en la librería, para poder usar esos efectos más adelante en otros proyectos, poder probarlos desde la aplicación *bHaptics Player* y, lo más importante, poder importarlos dentro de *bHaptics Developer*.



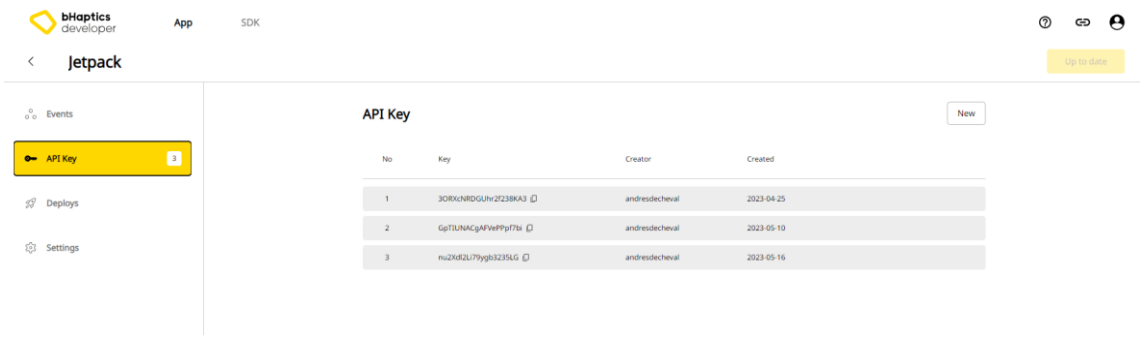
Volviendo a *bHaptics Developer*, en el espacio de eventos, hay que crear una categoría, en la que se podrán importar distintas vibraciones desde la librería personal. Hay que tener en cuenta que no es posible importar todas las vibraciones a la vez, ya que esto creará un evento en el que ocurran todas las vibraciones simultáneamente.

IMPORTANTE: es necesario crear proyectos por separado de cada una de las vibraciones de la librería, así como guardar esas vibraciones en la librería a pesar de que, teóricamente, al importar, se importan las vibraciones de la librería. No obstante, es posible que, de forma aleatoria, se puedan importar proyectos que

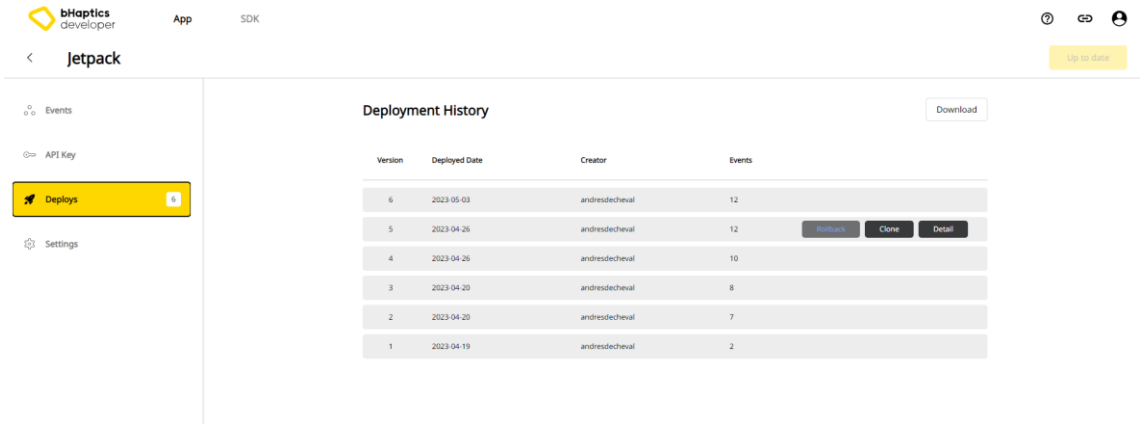
no estén en la librería y vibraciones de la librería que no sean proyectos, así como que se corrompan si no están ambas dentro del proyecto. No sabemos cómo funciona realmente.



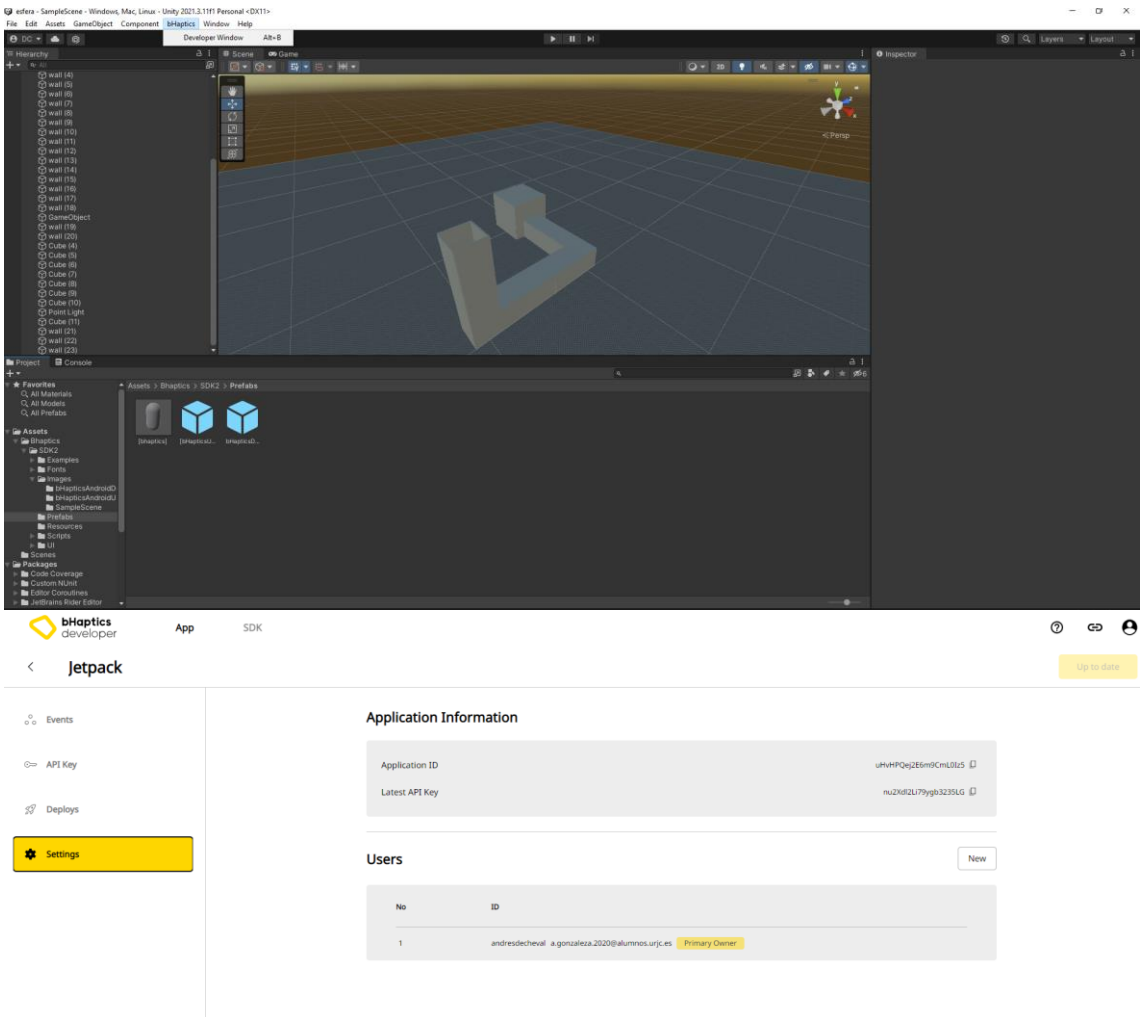
Una vez se tengan los eventos con sus patrones, se puede crear una "*API Key*", necesario para poder usar la aplicación creada dentro de *Unity*. En determinadas ocasiones, solo será necesaria una llave por dispositivos, pero también es probable que se necesiten llaves únicas por cada proyecto.

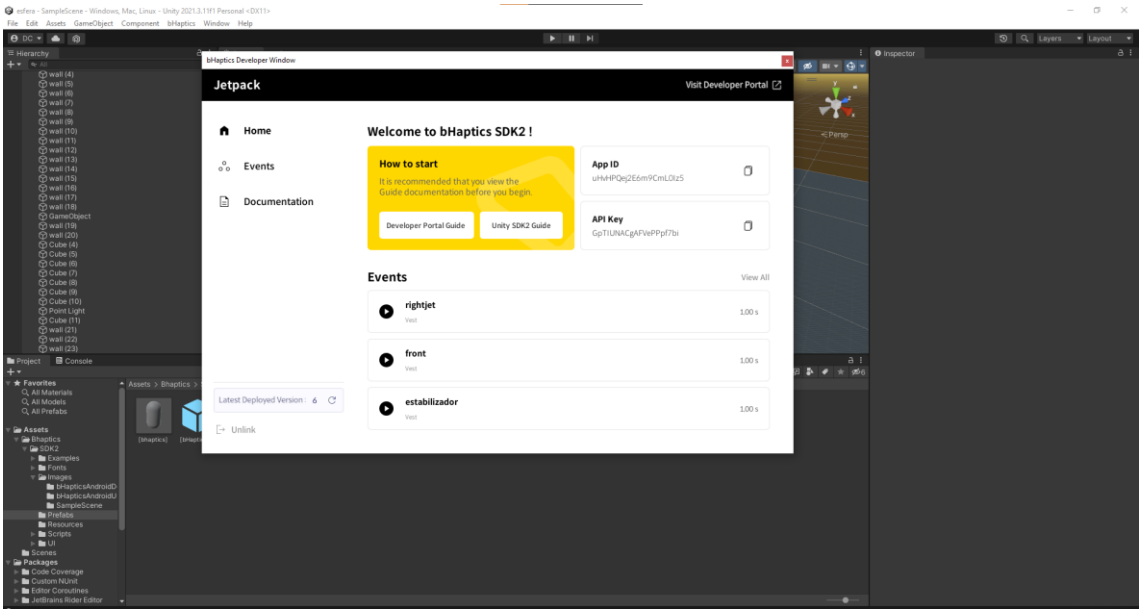


A continuación, es necesario realizar un "*Deploy*", cada vez que se realice uno se crea una versión distinta. Se tiene la opción de volver a una versión anterior, realizar esta acción puede corromper el proyecto por completo o solo algunas vibraciones. Se ha de crear un nuevo "*Deploy*" cada vez que se realice algún cambio.



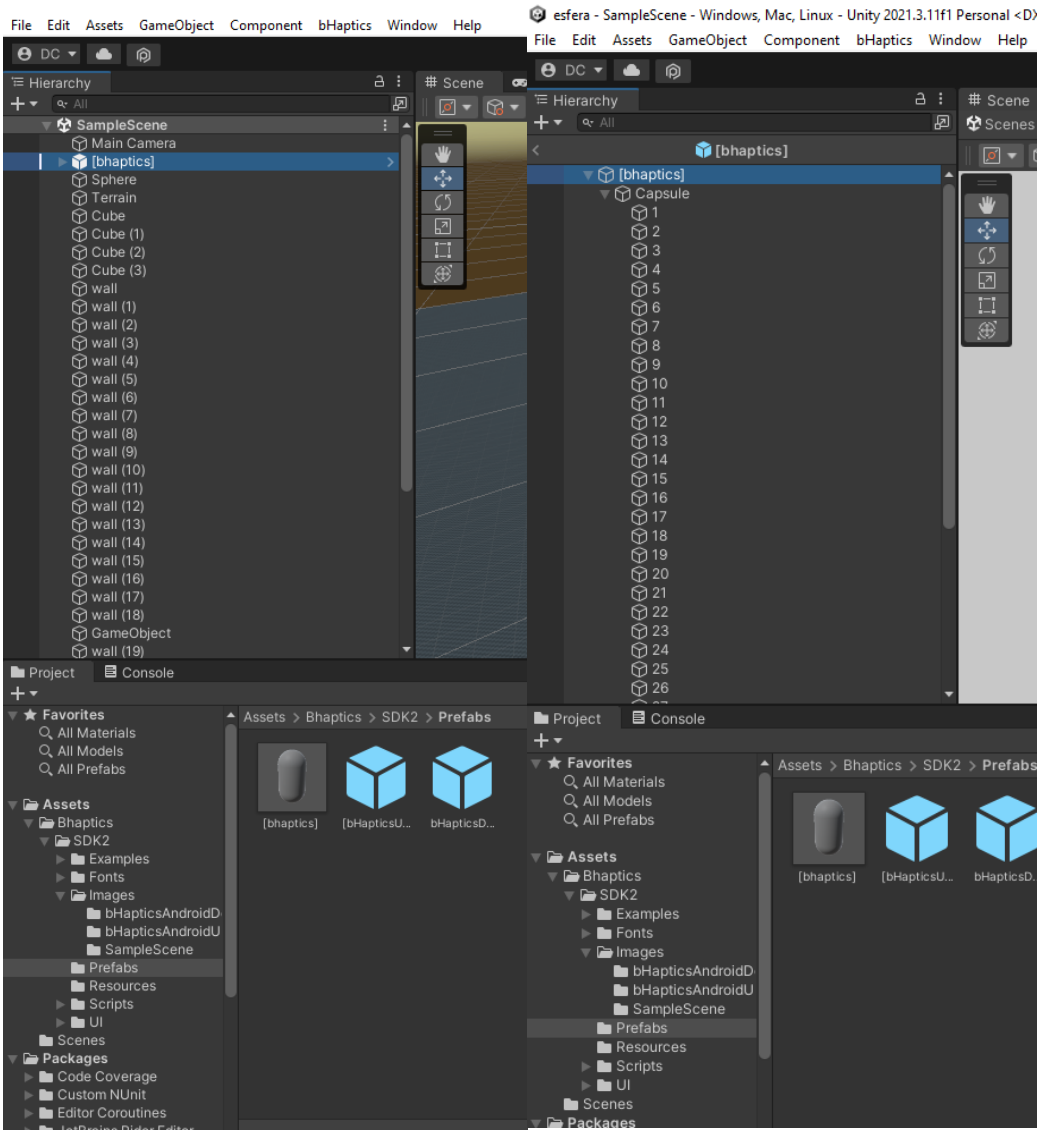
Dentro de *Unity*, tras haber importado el paquete, es necesario ir a la pestaña de *bHaptics* y abrir el "*Developer Window*", una vez dentro hay que meter el ID del proyecto y una llave. Estos se encuentran en la opción de ajustes de *bHaptics Developer*.





En esa ventana se pueden probar las distintas vibraciones, es recomendable probarlas todas para asegurarse de que se hayan importado bien, en caso de que existan errores y teóricamente se encuentre en el último *deploy*, se puede solucionar cerrando Unity, eliminando la llave desde *bHaptics Developer* y creando una nueva. Si los errores se mantienen y todo está correctamente actualizado, se recomienda simplemente eliminar la llave y el proyecto de Unity y reimportar todo en un nuevo proyecto.

Tras haberse asegurado de que las vibraciones funcionan correctamente, se tiene que arrastrar el *prefab* "*[bHaptics]*" en la escena en la que se quiera usar. Los códigos tienen que estar en objetos dentro del *prefab*, si no, no funcionará correctamente o directamente no funcionará.



Existen dos tipos de vibraciones principales, las ya hechas mediante el *bHaptics Desginer* o controlar las vibraciones de forma independiente de cada motor. En el caso de este proyecto, se decidió controlar las vibraciones del *jetpack* mediante eventos y las colisiones de forma independiente. A pesar de que los tutoriales estén desactualizados, los códigos del tutorial siguen siendo funcionales. El primer código muestra cómo se usan y modifican los eventos ya hechos, el segundo como controlar cada motor de forma independiente. En el caso del proyecto, creamos 40 *colliders*, uno para cada motor.

IMPORTANTE: para que funcione solo es necesario escribir *using Bhaptics.SDK2*, no es necesario instalar ningún paquete *NuGet* como dicen algunos tutoriales en *GitHub*.

- You can recall events you created with bHaptics Developer portal and plug into your scripts using the functions below. The main functions used are in `BhapticsLibrary.cs`.
  - Play

```
// Assuming the event name is "shootpistol"

BhapticsLibrary.Play(BhapticsEvent.SHOOTPISTOL);
or
BhapticsLibrary.Play("shootpistol");

// If you want to customize the event, call PlayParam()
// For example, when you attcked
BhapticsLibrary.PlayParam(BhapticsEvent.SHOOTPISTOL,
    intensity: 1f,    // The value multiplied by the original value
    duration: 1f,     // The value multiplied by the original value
    angleX: 20f,      // The value that rotates around global Vector3.up(0~360f)
    offsetY: 0.3f);  // The value to move up and down(-0.5~0.5)
```

  - PlayMotors

C# Copy

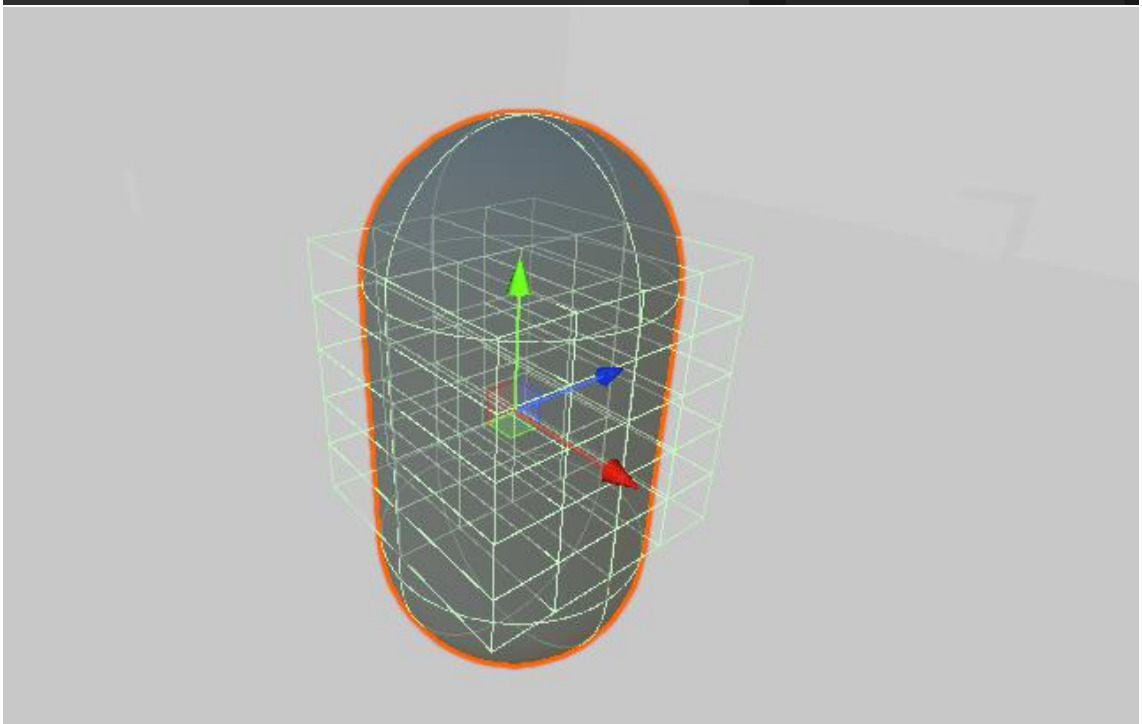
```
// If you don't want to create an event, call PlayMotors()

BhapticsLibrary.PlayMotors(
    position      : (int)Bhaptics.SDK2.PositionType.Head,
    motors        : new int[6] { 60, 80, ... },
    durationMillis: 1000
);

// Description of parameters
position      : The device type that you use
motors        : Assign the length of the array by the number of motors for device.
                And the int value means motor's intensity(0~100)
durationMillis: Duration of haptic(milliseconds)
                Recommended to use higher than 100ms
```

OffsetY only works for vest

Kim Sanghun Nov 16

A Position value by device

0: Vest
1: ForearmL
2: ForearmR
3: Head
4: HandL
5: HandR
6: FootL
7: FootR
8: GloveL
9: GloveR

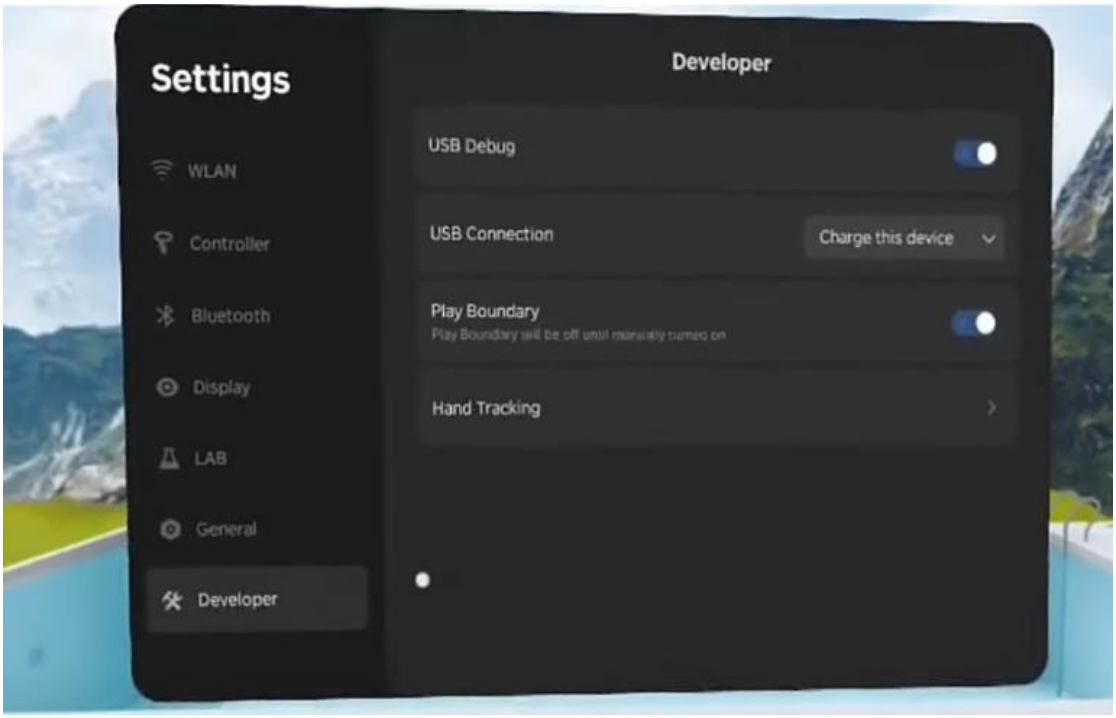


Configuración para usar PICO 4

Al igual que para el uso de las *Oculus*, se tuvo que realizar una serie de pasos previos para integrar el dispositivo con Unity.

Activar modo de desarrollador desde las PICO 4

Desde los ajustes de las gafas y haciendo click repetidas veces en el apartado "*Software Version*", se desbloquea un nuevo submenú etiquetado como "*Developer*". Desde ahí hay que activar la opción "*USB Debug*".

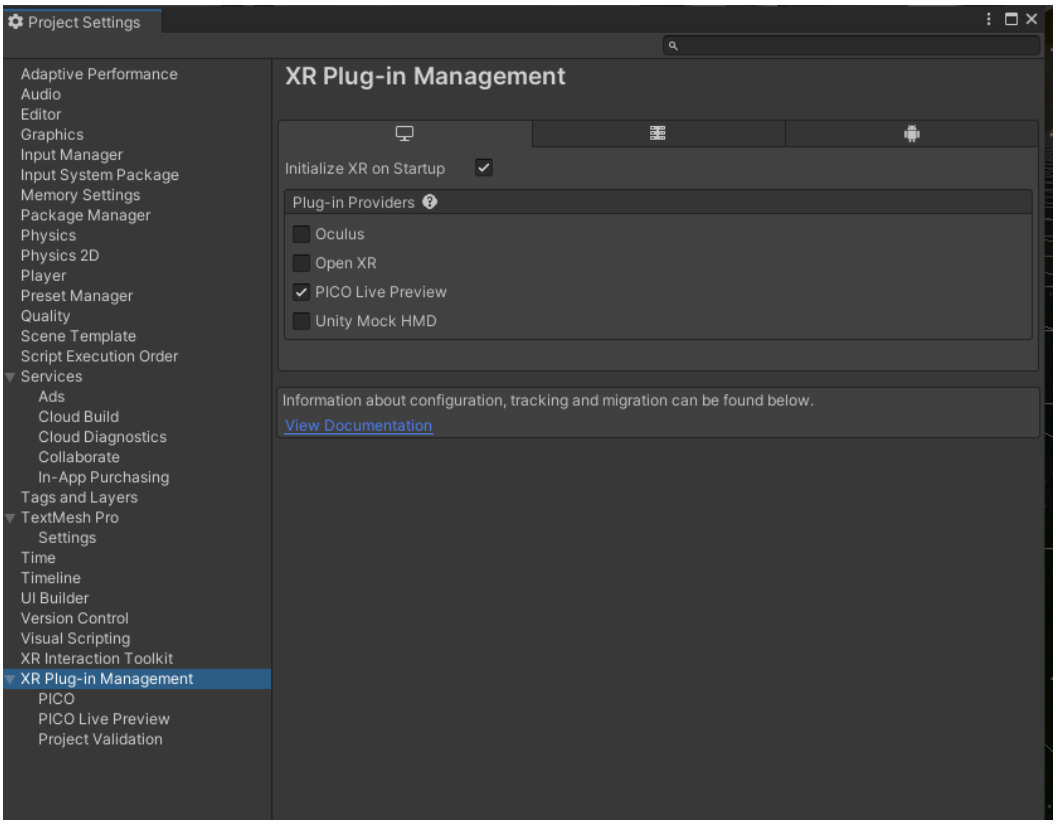


Importar "*PICO Unity Integration SDK*" y "*PICO Unity Live Preview Plugin*"

Tras haber creado una escena de *Unity*, se deben descargar dos *plugins*, uno para la integración con *Unity* y otra para previsualizar la escena de la aplicación.

Tras descargar desde la página web oficial ambos *plugins*, se deben importar desde el "*Package Manager*" de *Unity*, haciendo click en importar desde disco y seleccionando los archivos "*.json*" descargados.

Una vez importados, desde "*Project Settings*" se marca la casilla siguiente:



Con esto, ya se podrá visualizar la escena de Unity en el dispositivo (siguiendo también los pasos del enunciado de la práctica de HMD).

IMPORTANTE: por temas de rendimiento es importante añadir la siguiente sentencia en cualquier script en su apartado *Start()*:

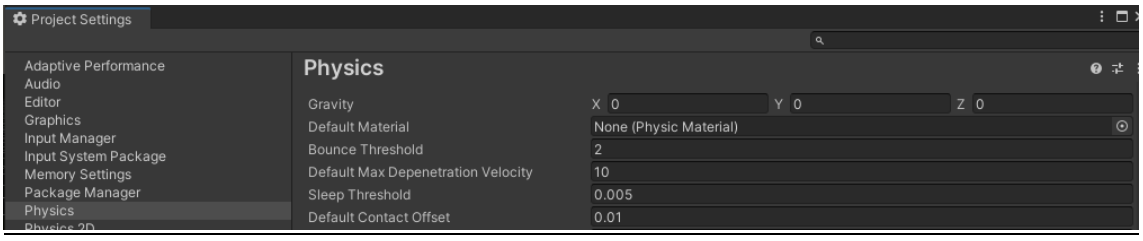
```
UnityEngine.Application.targetFrameRate = 72;
```

Realidad virtual en gravedad cero

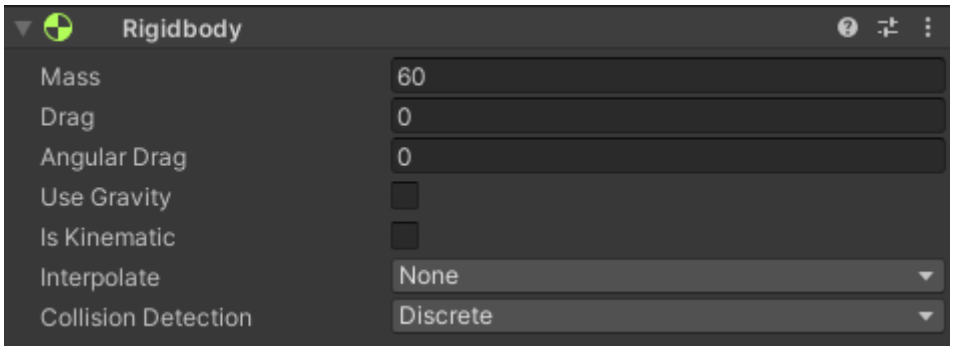
Teniendo como punto de partida una escena simple con un *XR Origin*, con los controles predeterminados y las manos animadas, se comenzó a idear el sistema de locomoción para simular un *jetpack* en el espacio.

Configurando físicas en *Unity*

Para empezar, se haría uso del sistema de físicas interno ofrecido por *Unity*, para lo cual, como primer paso se puso a cero la gravedad desde los ajustes de físicas de *Unity*.

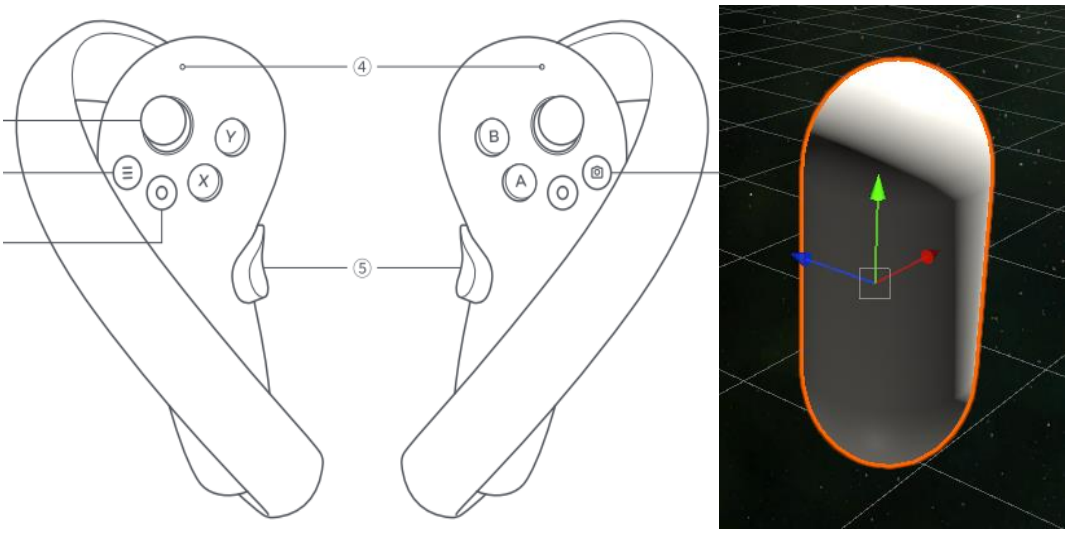


A continuación, se añadió un componente *Rigidbody* a nuestro *XR Origin*, para que tuviera el comportamiento de un sólido rígido, modificando sus parámetros de la siguiente forma:



Controles y su implementación

Ya que se deseaba un comportamiento completamente basado en físicas se prescindió del uso del sistema locomotor que ofrecía el *XR Interaction Toolkit* por lo que tocaría programar de cero todo el sistema de locomoción.



Antes que nada, tocaba plantear el esquema de control de los mandos, para lo que se optó por un sistema lo más parecido a un control de *FPS* (*first person shooter*), ya que son los más extendidos en uso. El *joystick* izquierdo sería el responsable del movimiento en el plano horizontal, permitiendo el avance hacia el frente, atrás, izquierda y derecha, mientras que el *joystick* derecho se encargaría de la rotación del cuerpo, alrededor de los ejes "*up*" (verde) y "*left*" (rojo), no habiendo rotación en el eje "*front*" (azul). Estos sería los controles más básicos necesarios.

Para la implementación de dichos controles, se recurriría al sistema *Action Based* para el control de *Input*.

Movimiento en plano horizontal

Tras documentarse acerca de los *Rigidbody*, se descubrió y probó el método *AddRelativeForce()*, que permite añadir fuerzas sobre nuestro cuerpo respecto a sus coordenadas locales. Una vez probados los distintos parámetros, se

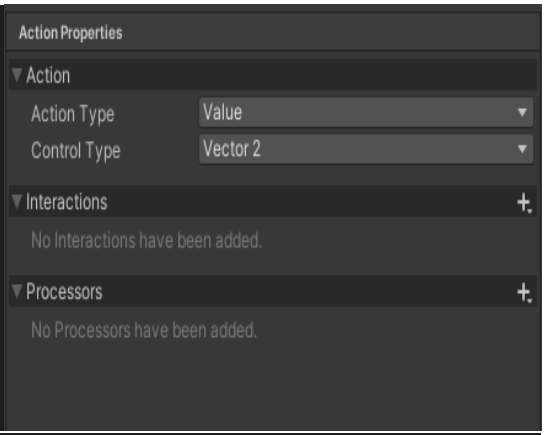
concluyó en utilizar el modo "*Acceleration*" que añade una aceleración continúa ignorando la masa del cuerpo.

```
myRigidbody.AddRelativeForce(xValue * Speed * Time.fixedDeltaTime, 0,
yValue * Speed * Time.fixedDeltaTime, ForceMode.Acceleration);
```

Esta sería la línea de código resultante, definiendo previamente el valor de "*Speed*". Los valores "*xValue*" e "*yValue*" se obtienen a través del *Input*.

Utilizaremos la acción de mover del control izquierdo, que lee los valores "x" e "y" del *joystick* izquierdo.

```
1 referencia
void Move(InputAction.CallbackContext context)
{
    xValue = context.ReadValue<Vector2>().x;
    yValue = context.ReadValue<Vector2>().y;


    pressedDown = true;

}

1 referencia
void MoveCancel(InputAction.CallbackContext context)
{
    pressedDown = false;
}
```




```
void Start()
{
    UnityEngine.Application.targetFrameRate = 72;
    myRigidbody = GetComponent<Rigidbody>();
    actionMove = myActionsAsset.FindAction("XRI LeftHand Locomotion/Move");
    actionMove.performed += Move;
    actionMove.canceled += MoveCancel;
}
```

Aquí surge una primera complicación, ya que ahora mismo, por así decirlo, se detectan únicamente las variaciones en el *Input*, por lo que esta solución no es válida para que, si se mantiene pulsado hacia una dirección, se añadieran fuerzas constantemente. Tras investigar acerca del sistema *Action Based*, se encontró un método que permite detectar cuando se deja de presionar un *Input*, por lo que usando un booleano se puede controlar las fuerzas a añadir en nuestro cuerpo en el bucle *FixedUpdate()*.

```
private void FixedUpdate()
{
    if (pressedDown)
    {
        myRigidbody.AddRelativeForce(xValue * Speed * Time.fixedDeltaTime, 0, yValue * Speed * Time.fixedDeltaTime, ForceMode.Acceleration);
    }
}
```

Con esta modificación nuestro movimiento quedaría implementado.

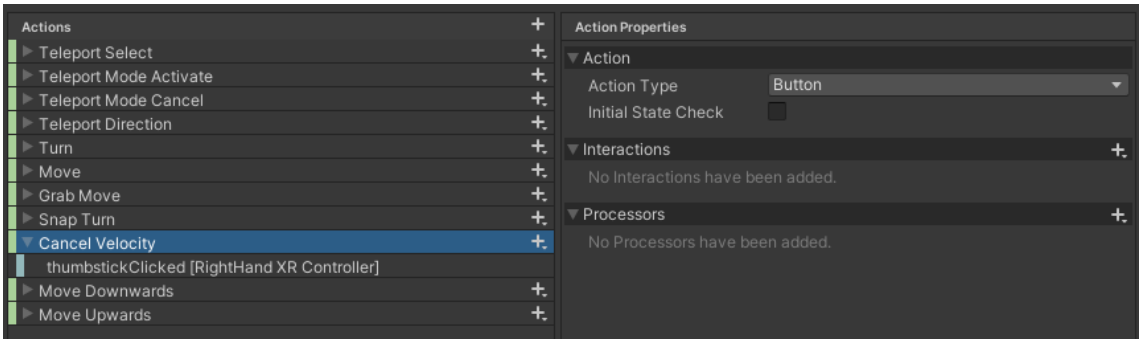
Rotación

Los pasos para la rotación son los mismos realizados para el movimiento, solo que, en vez de añadir una fuerza lineal a nuestro cuerpo, añadiríamos una fuerza de torque local mediante el método *AddRelativeTorque()* sobre los ejes "*up*" y "*left*".

```
if (pressedDown)
{
    myRigidbody.AddRelativeTorque(Vector3.up * xValue * Speed * Time.fixedDeltaTime, ForceMode.Acceleration);
    myRigidbody.AddRelativeTorque(Vector3.left * yValue * Speed * Time.fixedDeltaTime, ForceMode.Acceleration);
```

Otros controles

Tras implementar la rotación se descubrió la necesidad de anular toda fuerza que tuviera nuestro cuerpo en cualquier instante, por lo que se añadió un "botón del pánico" ("*Thumbstick Clicked*" derecho) que hiciera esto mismo y frenara cualquier fuerza. Para ello, se añadió una nueva acción, para luego vincularla por código.

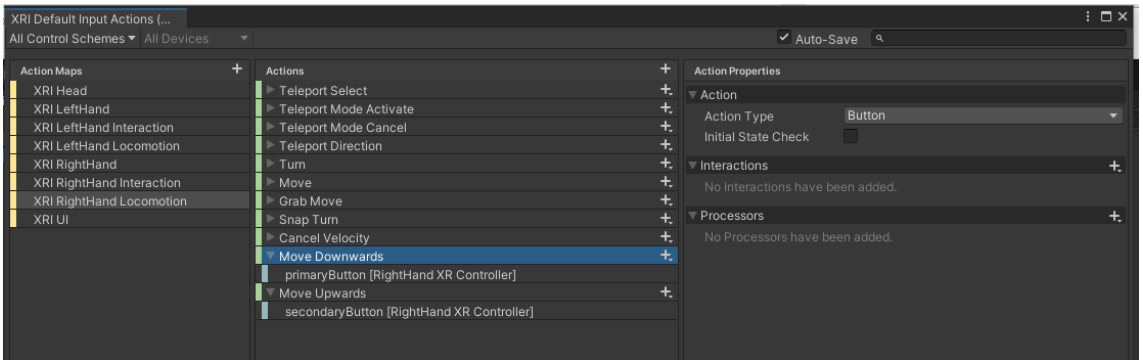



```
actionNullifyVelocity = myActionsAsset.FindAction("XRI RightHand Locomotion/Cancel Velocity");
actionNullifyVelocity.performed += NullifyVelocity;
```

```
1 referencia
void NullifyVelocity(InputAction.CallbackContext context)
{
    myRigidbody.velocity = Vector3.zero;
    myRigidbody.angularVelocity = Vector3.zero;
}
```

Por último, se quiso añadir un movimiento vertical para poder ascender y descender, ampliando y facilitando el movimiento en gravedad cero.

Los pasos fueron los mismos que anteriormente:



```
myRigidbody = GetComponent<Rigidbody>();
actionUp = myActionsAsset.FindAction("XRI RightHand Locomotion/Move Upwards");
actionUp.performed += MoveUp;
actionUp.canceled += UpCancel;
actionDown = myActionsAsset.FindAction("XRI RightHand Locomotion/Move Downwards");
actionDown.performed += MoveDown;
actionDown.canceled += DownCancel;
```

```
Mensaje de Unity | 0 referencias
void FixedUpdate()
{
    if (upPressed)
    {
        myRigidbody.AddRelativeForce(0, Speed * Time.fixedDeltaTime, 0, ForceMode.Acceleration);
    }
    if (downPressed)
    {
        myRigidbody.AddRelativeForce(0, -Speed * Time.fixedDeltaTime, 0, ForceMode.Acceleration);
    }
}
```

Problemas derivados y soluciones

Un problema inicial mientras se probaban los controles, fue el centro de masas de los *Rigidbody*, ya que, de no ser asignado manualmente, se calcula automáticamente utilizando todos los *colliders* asignados al cuerpo, por lo que las rotaciones se realizaban de manera extraña, girando entorno a un punto situado en el frente del cuerpo. Para arreglar esto fue tan simple como reajustar y asignar nuevos *colliders* a las manos y al cuerpo.

Otro problema fue que, al anular las fuerzas del cuerpo, se realizaba de manera brusca causando mareos. Esto se solucionó modificando el código para que se realizara de manera progresiva, implementándolo en el *FixedUpdate()*:

```
if (panicButton)
{
    if(myRigidbody.velocity != Vector3.zero)
    {
        myRigidbody.velocity -= myRigidbody.velocity * Time.fixedDeltaTime;
        myRigidbody.angularVelocity -= myRigidbody.angularVelocity * Time.fixedDeltaTime;
    }
}
```

Tunneling

Ya que el movimiento en gravedad cero puede ser desorientador, para evitar el *Virtual Reality Sickness*, hemos incluido el *Tunneling* (*vignette*), para así tener un punto fijo de referencia y evitar que los movimientos bruscos o los que no estamos acostumbrados a realizar en la vida real nos mareen. Además, simula el hecho de llevar una escafandra espacial.

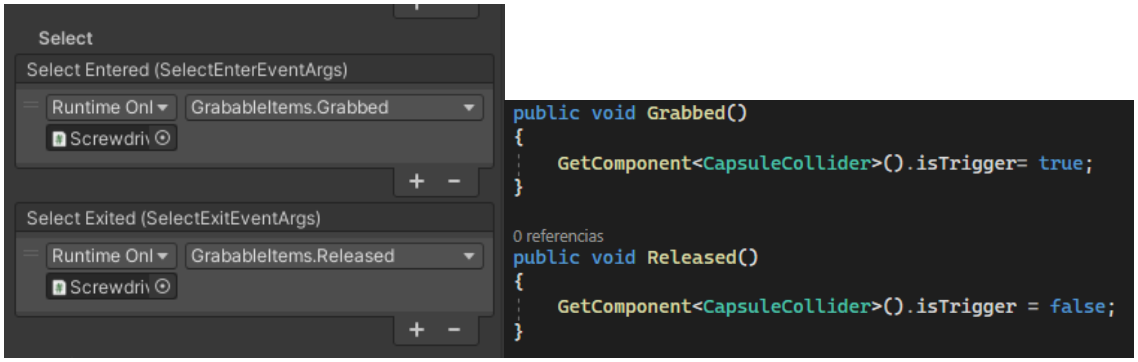
Interactuando con el entorno

Introducción

Las interacciones principales con el entorno son las colisiones físicas con los objetos y el escenario, así como la interacción per se, con estos primeros para la resolución de puzles.

La primera parte fue sencilla, únicamente había que añadir *colliders* físicos (no marcados como "*IsTrigger*") y *rigidbodies* tanto al cuerpo y las manos como al resto de la escena.

La segunda parte constó de pensar maneras de utilizar los *XR Interactable* ofrecidos por el *Toolkit*. La primera prueba se realizó con un *XR Grab Interactable* y las manos en modo *XR Direct Interactor*. Aquí surgió un problema, ya que debido a poner *colliders* físicos a las manos, cuando estas agarraban algún objeto, se producía un comportamiento indeseado por el cual comenzabas a añadir fuerzas descontroladas al cuerpo. La solución fue tan simple como marcar los *colliders* de los objetos como "*IsTrigger*", asignando al estado de "*Select*" los métodos creados:

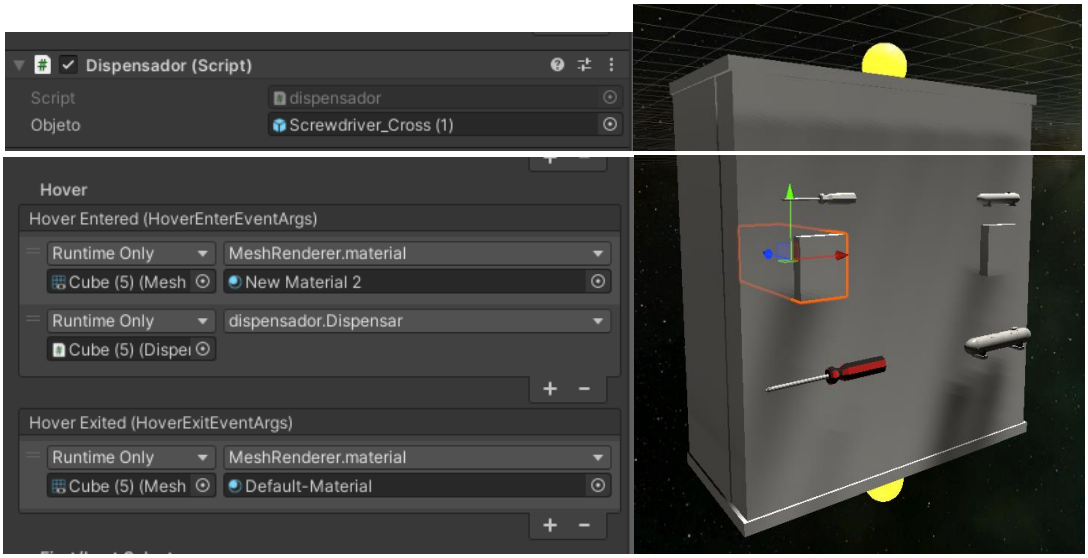


Con esta solución, solo quedaría adaptar el agarre de los objetos añadiéndoles un *Attach Transform* modificado convenientemente.

Puzle del fusible

El primer puzle implementado constaba de tres pasos: conseguir las herramientas adecuadas, localizar y llegar a la caja de fusibles, desatornillar la tapa y colocar el fusible.

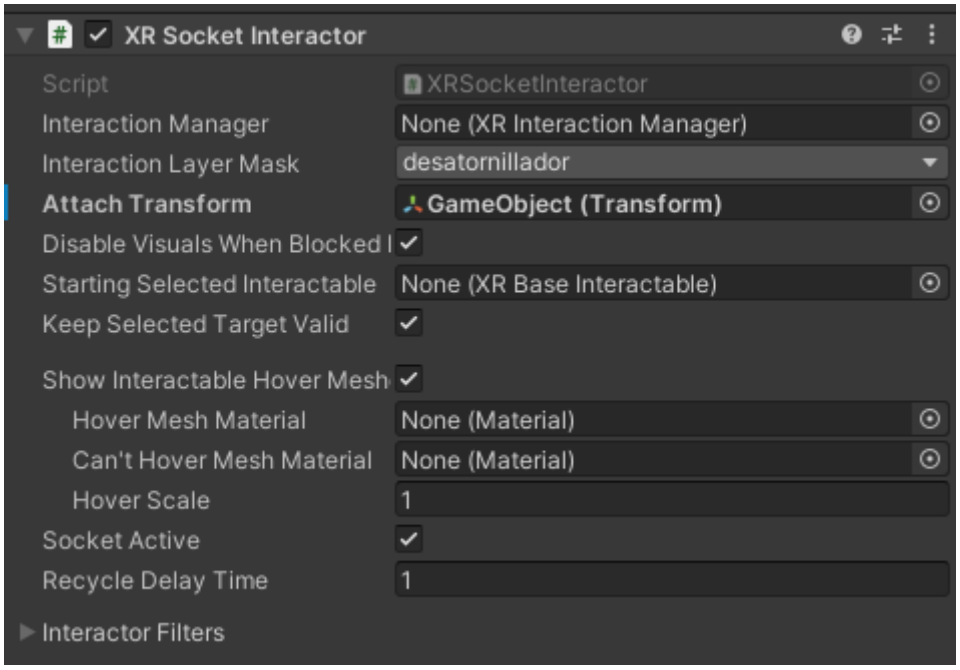
Para el primer paso, se creó un dispensador que genere destornilladores y fusibles de manera ilimitada para que, en caso de perder dichos objetos en la infinidad y desolación del espacio, pudieras recuperarlos.



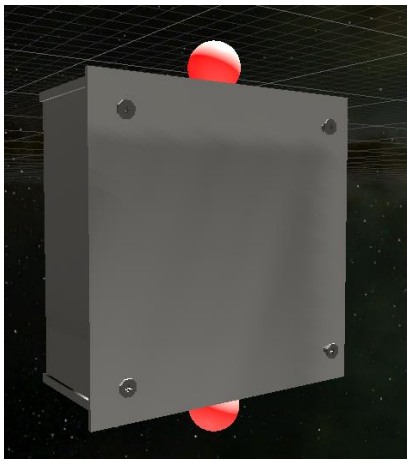
```
0 referencias
public void Dispensar()
{
    objetos = GameObject.FindGameObjectsWithTag(tagObjeto);
    foreach (GameObject objeto in objetos)
    {
        if (objeto.activeSelf)
        {
            Destroy(objeto);
        }
    }
    GameObject gameo = Instantiate (objeto);
    gameo.SetActive(true);
    gameo.transform.position  = objeto.transform.position;
    gameo.transform.localScale= objeto.transform.localScale;
    gameo.transform.rotation= objeto.transform.rotation;
}
```

Para aprovechar los recursos del *Toolkit*, se decidió implementar botones táctiles, que se activaran cuando acercaras la mano a ellos, asignando al estado *hover*, un cambio de material (para el *feedback* visual) y el código para generar el objeto asignado.

A continuación, para transportar de manera cómoda los objetos, se simuló un cinturón situado en la parte superior del cuerpo que sirviera para sujetarlos. Para ello se crearon dos *GameObjects* vacíos, con el componente "*XR Socket Interactor*". Se pensaba permitir que los objetos se pudieran colocar en cualquiera de los dos huecos creados, pero por motivos de simplicidad, se asignó a cada uno un hueco concreto, modificando el campo "*Interaction Layer Mask*" a la correspondiente.



Esto se realizó de esta manera debido a la modificación anterior de los "*Attach Transform*" de cada objeto, ya que, para su correcta colocación en el cinturón, había que modificar individualmente este parámetro para que esta fuera correcta.



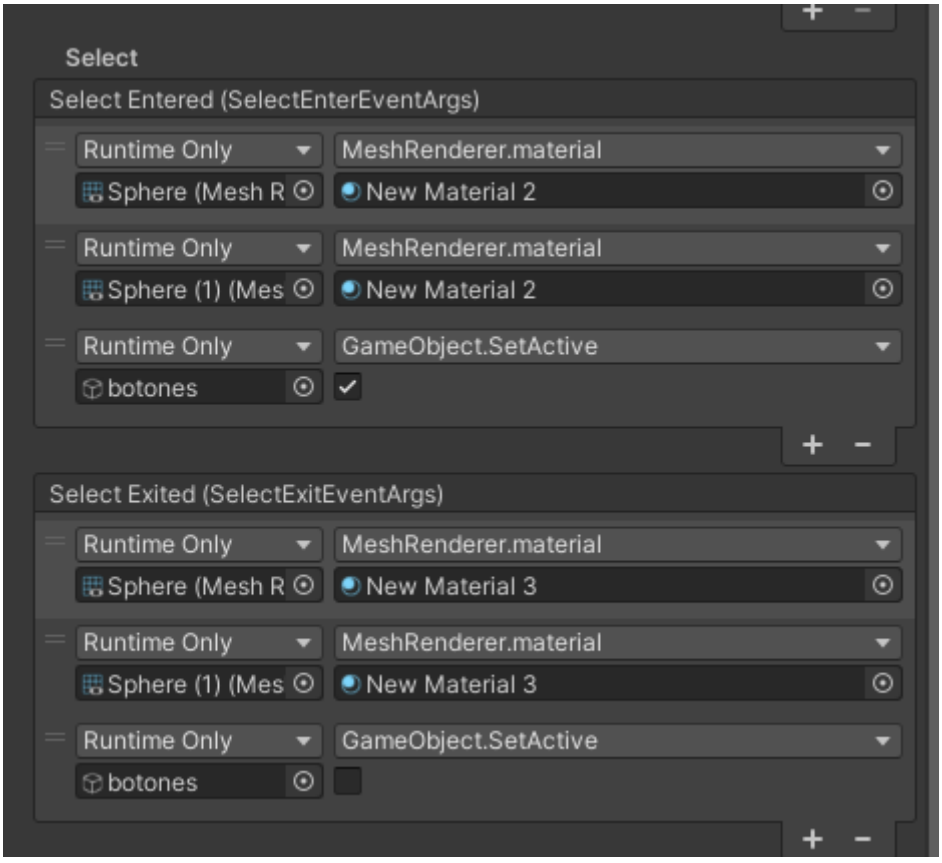
Tras esto, el objetivo es localizar la caja de fusibles, que estaría marcada con dos luces rojas para su fácil localización.

Una vez en frente de ella, se debe agarrar el destornillador y acercarlo a los tornillos, para que, una vez realizado con todos, la tapa se suelte y deje al jugador colocar el fusible en su interior.

El código de los tornillos y la tapa no hacen uso del *Toolkit*, por lo que no se va a explicar.

Para colocar el fusible se crearon dos cubos para hacer de soporte, añadiendo a uno el componente "*XR Socket Interactor*", reajustando su "*Attach Transform*" y cambiando su "*Interaction Layer Mask*" para que únicamente acepte al fusible.

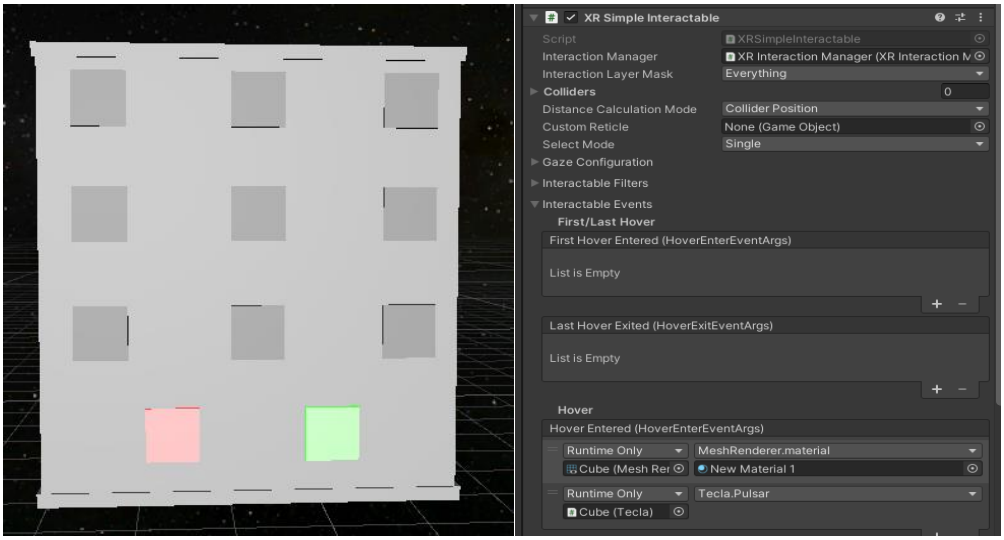
Cuando se coloca el fusible en su interior se activan los siguientes métodos, que cambian el color de las luces a verde y activan el siguiente puzle.



Puzle del teclado

Este puzle consta de un teclado de 11 teclas, 9 para insertar valores, otro para confirmar y el último para borrar. La prueba consiste en pulsar las teclas en el orden correcto y pulsar el botón de confirmación. El propósito de este puzle era el de aprovechar la mecánica de los botones táctiles y a su vez requerir que el jugador los alcance con las manos.

La implementación es la misma que la de los botones del dispensador, pero asignando por código la funcionalidad necesaria en este caso. Los Scripts no se explicarán aquí, pero quedan adjuntos en los archivos entregados.



Portales

Motivación y objetivo

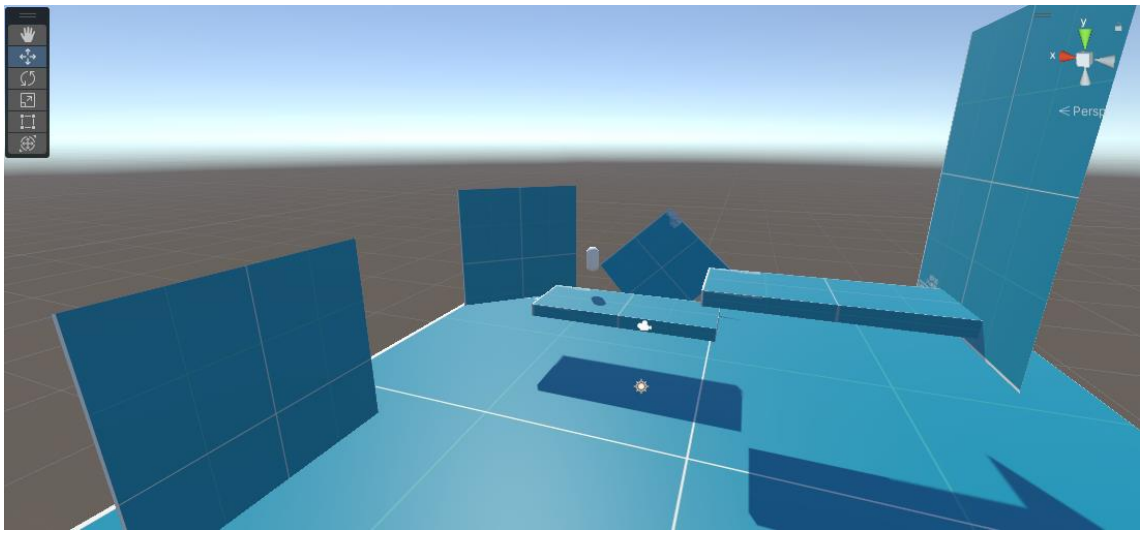
Los portales se incluyeron en el simulador como una metáfora de movimiento por varios motivos. Entre ellos está el hecho de que se quería añadir un nivel adicional de complejidad al juego, ya que los portales proporcionan una mecánica extra que sirve como herramienta para completar algunos de los niveles del juego. Además, son muy útiles para agregar dinamismo al movimiento en ingravidez.

Sin embargo, la razón más importante por la cual se incluyeron fue porque podrían ofrecer una experiencia visual impactante. Al igual que en el videojuego "*Portal*", romper con el espacio euclidiano y hacerlo en realidad virtual podría resultar muy llamativo y sorprendente para los jugadores.

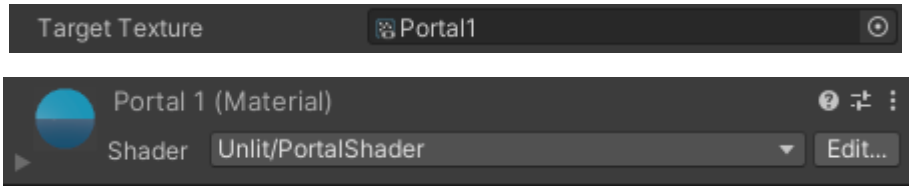
Implementación

Lo primero que se hizo fue investigar utilizando diversos tutoriales para aprender a simular los portales del videojuego *Portal* en *Unity* y qué herramientas proporcionaba Unity para lograrlo. Un ejemplo de recurso útil encontrado fue el siguiente vídeo.

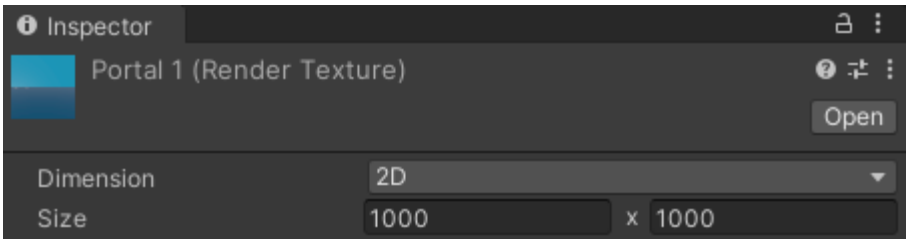
Se procedió a crear un escenario de pruebas con un jugador con controles *FPS* que incluía diferentes planos y cajas para poder probar los portales a medida que se iban programando.



Se crearon dos planos fijos que representarían los portales. A estos planos se les asignó un material utilizando una textura de tipo "*Render Texture*", que mostraba lo que era renderizado por una cámara asociada al otro portal. Se utilizó un *shader* de tipo "*Unlit Shader*" ligeramente modificado para limitar el renderizado a solo lo que debería ser visible a través del hueco del portal, en lugar de capturar toda la imagen de la cámara.

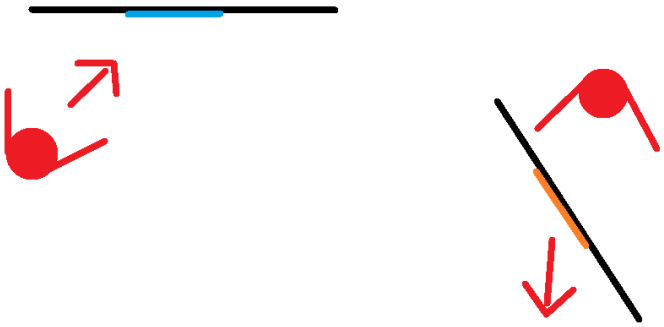


Se identificó un problema de baja calidad en el renderizado de los portales, el cual se solucionó aumentando el tamaño de las texturas utilizadas.



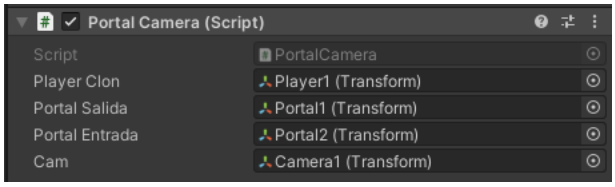
Se implementó el control del movimiento y rotación de las cámaras que renderizaban los portales para que coincidieran con los del jugador, pero con las transformaciones necesarias. Esta etapa resultó ser la más compleja debido a las transformaciones necesarias en términos de rotación y posición entre objetos y sus relaciones padre-hijo.

En este apartado nos hemos valido de conocimientos previos en transformaciones geométricas y cambios de base, además de la lógica y el uso de diagramas.

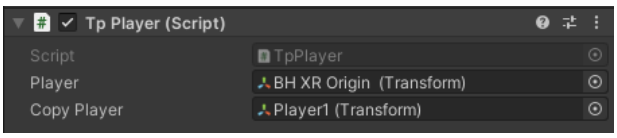


En este ejemplo, para obtener la posición y rotación de la cámara que renderiza la textura de un portal, se ha tenido en cuenta la posición y rotación relativas del jugador respecto a los portales.

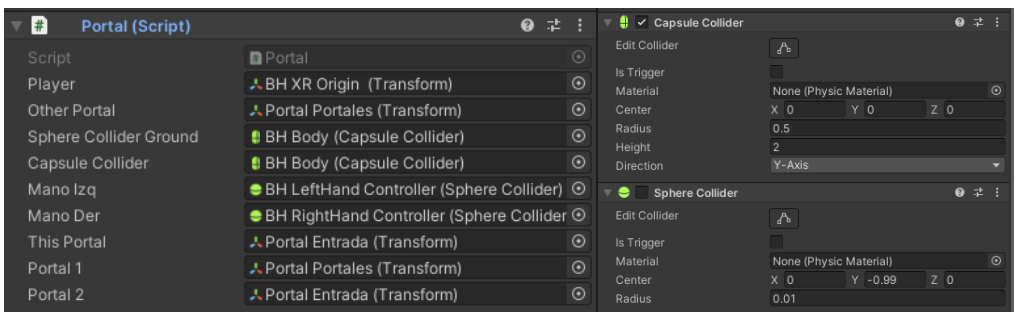
```
private void Update()
{
    cam.localPosition = new Vector3(-playerClon.localPosition.x, -playerClon.localPosition.y, playerClon.localPosition.z);
    cam.localRotation = Quaternion.AngleAxis(180, Vector3.forward) * playerClon.localRotation;
```



Para facilitar esta tarea, se creó un objeto vacío para cada portal que copiaba la rotación y posición del jugador.

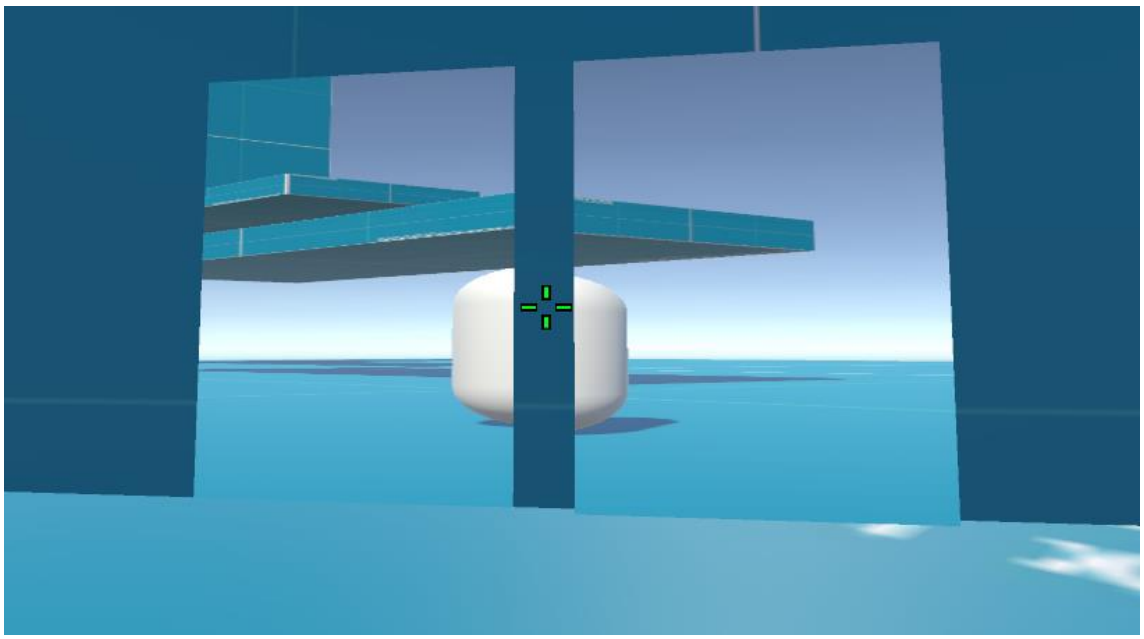


Se trabajó en la implementación del teletransporte entre los portales, asegurando que se mantuviera la velocidad, el ángulo de entrada y la posición del jugador. Surgió un problema en esta etapa, donde se consideró poco realista que el jugador fuera teletransportado al tocar el portal con cualquier parte de su cuerpo. Para mejorar la ilusión de pasar a través del portal como si fuera una puerta, se agregó un segundo colisionador más pequeño en el centro del jugador. Se configuró el colisionador principal para que cambie su estado a "*IsTrigger*", permitiendo al jugador acercarse más al portal hasta que se active el segundo colisionador y se realice el teletransporte.



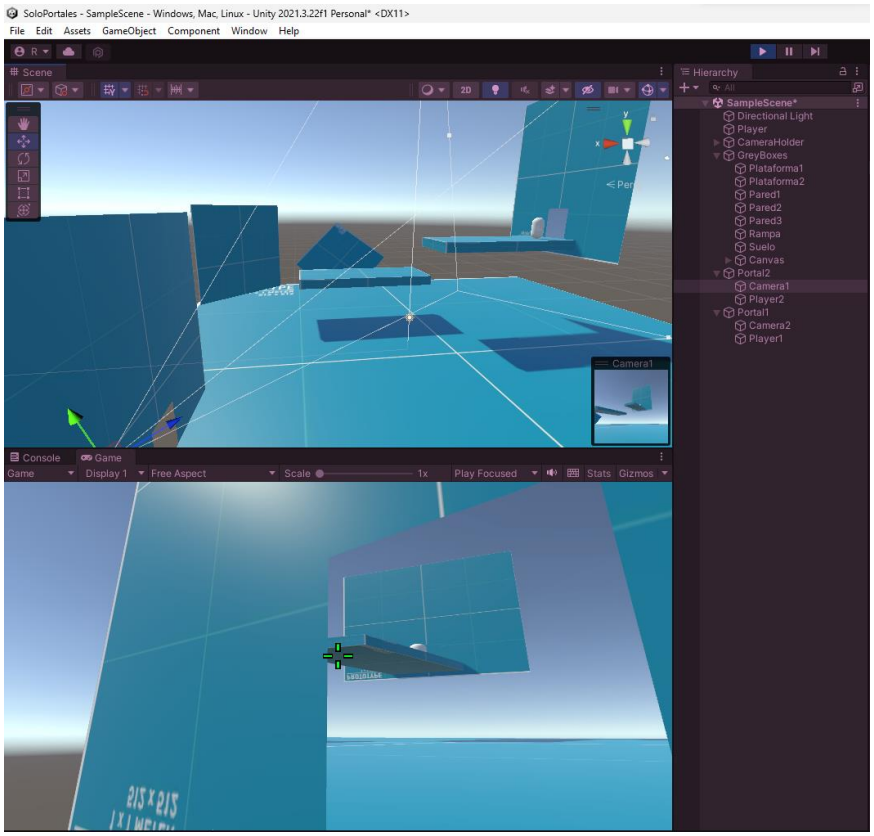
```
private void OnTriggerEnter(Collider other)
{
    if (other == capsuleCollider)
    {
        sphereColliderGround.enabled = true;
        capsuleCollider.isTrigger = true;

        manoDer.isTrigger = true;
        manoIzq.isTrigger = true;
    }

    if (other == sphereColliderGround)
    {
        Vector3 direction = otherPortal.up / 1;
        Vector3 distance = new Vector3(player.position.x - thisPortal.position.x, player.position.y - thisPortal.position.y, player.position.z - thisPortal.position.z);
        Vector3 newPosition = new Vector3(otherPortal.position.x + distance.x, otherPortal.position.y + distance.y, otherPortal.position.z + distance.z);

        player.position = newPosition;
        player.position += direction;

        float up = Vector3.SignedAngle(thisPortal.up, otherPortal.up, Vector3.Cross(portal1.up, portal2.up));
        float forward = Vector3.SignedAngle(thisPortal.forward, otherPortal.forward, Vector3.Cross(portal1.forward, portal2.forward));
        float right = Vector3.SignedAngle(thisPortal.right, otherPortal.right, Vector3.Cross(portal1.right, portal2.right));

        player.gameObject.GetComponent<Rigidbody>().velocity = Quaternion.Euler(0, 180-up, 0) * player.gameObject.GetComponent<Rigidbody>().velocity;
        player.Rotate(0,180-up,0, Space.World );
    }
}

Mensaje de Unity | 0 referencias
private void OnTriggerExit(Collider other)
{
    capsuleCollider.isTrigger = false;
    sphereColliderGround.enabled = false;
}
```

Para permitir que los portales se coloquen en cualquier posición deseada y no estén fijos, se agregó una retícula en la pantalla del jugador mediante la cual se lanzó un *raycast*. Cuando el *raycast* colisiona con una pared o el suelo, se coloca el portal correspondiente en esa ubicación. En este apartado, también hubo que recolocar la posición y rotación de los portales.



```
void RayCastPortal(int portalID)
{
    // Crear un rayo desde la posición de la cámara en la dirección de su vista
    Ray ray = new Ray(Camera.main.transform.position, Camera.main.transform.forward);

    // Almacenar la información de colisión del rayo
    RaycastHit hit;

    // Lanzar el rayo y comprobar si ha colisionado con algún objeto
    if (Physics.Raycast(ray, out hit))
    {
        // Imprimir el nombre del objeto colisionado en la consola
        Debug.Log("Objeto colisionado: " + hit.collider.gameObject.name);
        GameObject collisionObject = hit.collider.gameObject;
        Vector3 collisionPoint = hit.point;

        // Colocamos el portal 1 o 2 según el ID recibido
        if (portalID == 1)
        {
            portal1.rotation = collisionObject.transform.rotation;
            portal1.position = collisionPoint;
            portal1.position += (collisionObject.transform.up / 100f);
            // Reiniciamos posición y rotación de los portales
            portalCamera1.resetCam();

        }
        else if (portalID == 2)
        {
            portal2.position = collisionPoint;
            portal2.rotation = collisionObject.transform.rotation;
            portal2.position += (collisionObject.transform.up / 100f);
            // Reiniciamos posición y rotación de los portales
            portalCamera2.resetCam();
        }
    }
}
```

El resultado final es el siguiente:



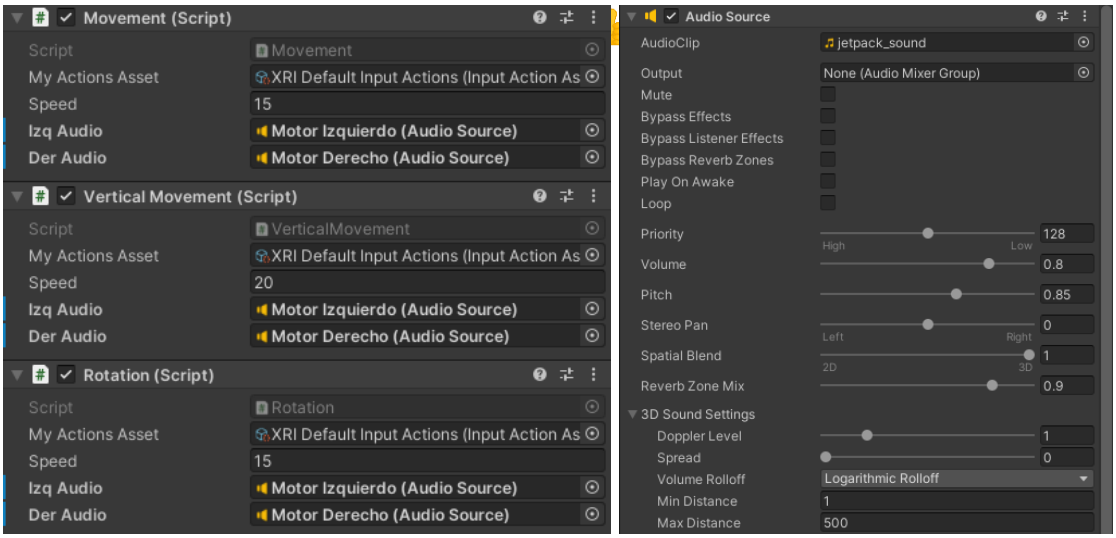
Otros elementos

Apartado gráfico

Los modelos 3D y las texturas utilizadas fueron cogidos de la Unity Asset Store, y están importados en el proyecto.

Audio

Con la intención de hacer el proyecto lo más realista posible, y al no haber sonido en el espacio, se decidió incluir solo un sonido de respiración dentro de la escafandra, sonidos de los motores del *jetpack* y de disparos de las bolas para alertarte de su presencia.



Se activó la opción de "*Spatial Blending*" para que el sonido fuera afectado por los cálculos de espacialización 3D de sonido.

Reasignar portales para adaptarlos a VR

Para adaptar los portales a la realidad virtual solo hubo que hacer pequeños cambios en cuanto a la cámara que se usaba como principal y al *transform* del jugador, que ahora sería el "*Body*" que se encontraba dentro del XR Origin.

El mayor cambio fue en la pistola de portales, se agregó un componente a esta, que fuera *XR Grab Interactable*, asignando a su evento *Activate* la funcionalidad de colocar un portal.

Bibliografía

https://youtu.be/NGeZCAi2bJs

https://www.youtube.com/watch?v=f473C43s8nE

https://github.com/daniel-ilett/portals-urp

https://www.youtube.com/watch?v=PkGjYig8avo

https://docs.unity3d.com/Manual/index.html

https://bhaptics.notion.site/Plug-in-deployed-events-to-Unity-33cc33dcfa44426899a3f21c62adf66d

https://assetstore.unity.com/packages/tools/integration/bhaptics-haptic-plugin-76647

https://bhaptics.notion.site/How-to-migrate-from-SDK1-old-to-SDK2-new-007c00b65129404287d9175b71fa029c

https://www.bhaptics.com/support/developers

https://www.youtube.com/watch?v=Pyq9GHdchzc

https://studio.bhaptics.com/project/workspace

https://designer.bhaptics.com/projects

https://bhaptics.notion.site/Create-haptic-events-using-bHaptics-Developer-Portal-b056c5a56e514afeb0ed436873dd87c6

https://bhaptics.notion.site/Plug-in-deployed-events-to-Unity-33cc33dcfa44426899a3f21c62adf66d

https://docs.unity3d.com/es/2019.4/Manual/class-Rigidbody.html

https://github.com/bhaptics/haptic-guide

https://github.com/bhaptics/haptic-library/wiki/Getting-Started(Unity)

https://github.com/NuGet/docs.microsoft.com-nuget.es-es/blob/live/docs/consume-packages/install-use-packages-powershell.md

https://developer-global.pico-interactive.com/sdk?deviceId=1&platformId=1&itemId=12

https://developer-global.pico-interactive.com/document/unity/preview-tool/

https://p16-platform-static-va.ibyteimg.com/tos-maliva-i-jo6vmmv194-us/ae723dcf5efe77849bb06a1cdd4ee77b.pdf

https://docs.unity3d.com/ScriptReference/ForceMode.html

https://docs.unity3d.com/ScriptReference/Rigidbody-centerOfMass.html