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

VRope: cuerda retráctil con fricción controlable para simular fuerzas en realidad virtual y su comparación con métodos pseudo-hápticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Alumno: Unai Javier Fernández Ortega

Director: Asier Marzo Perez

Pamplona, 3 de septiembre de 2020

upna
Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

BREVE DESCRIPCIÓN

Nuestro objetivo es generar fuerzas en un entorno de realidad virtual centrándonos especialmente en las fuerzas de tensión. Durante el desarrollo de este proyecto se crearon tres aplicaciones de realidad virtual, un campo de tiro de flechas, una recreación simple de un conjunto de planetas y un botón que se puede presionar. Se utilizan tres modos para dotar de sensación háptica la falta de respuesta física de los controladores VR. El primer modo es el "modo clásico", que consiste en el modo estándar de la mayoría de las aplicaciones de realidad virtual. Este modo no envía ningún estímulo háptico al usuario. El segundo modo es el "modo pseudo-háptico", que envía estímulos al jugador simplemente utilizando las herramientas proporcionadas por el hardware Oculus Quest. Por ejemplo, en la aplicación Arrow Field hacemos que el controlador del usuario vibre cada vez más en función de la tensión del arco. El tercer modo es el "modo háptico". Para este modo hemos creado un dispositivo electrónico que consiste en un carrete controlado por un microcontrolador. Este modo permite sentir la sensación de una fuerza real entre las manos del usuario. La fuerza aplicada debe decidirse en tiempo real dependiendo de la distancia entre las manos enviadas por el oculus quest al microcontrolador. Este proyecto en su conjunto pretende corregir de forma económica y sencilla uno de los principales inconvenientes de la realidad virtual y al mismo tiempo darle al usuario la opción de utilizar su método favorito para hacerlo.

PALABRAS CLAVE

- Unity3D
- ESP8266
- Realidad Virtual
- Háptica
- Controladores

Table of Contents

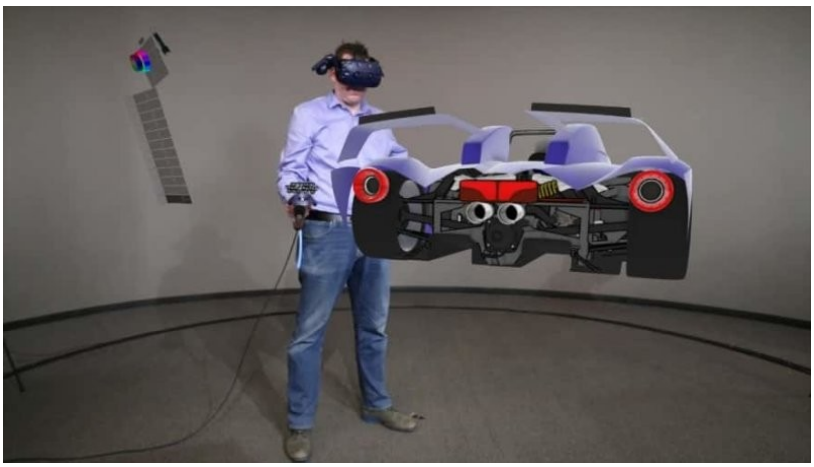
PRÓLOGO: Introducción al proyecto

Parte 1: Agradecimientos

Antes de comenzar, querría dedicar unas palabras a todos aquellos que siempre han estado a mi lado, tanto en este proyecto como en esta vida. A mis amigos, con los que me divierto en los buenos momentos y en los que me apoyo en los momentos más malos. A mis padres, que siempre me han protegido y aconsejado con sabiduría, a mi hermano, con el que paso muchas horas charlando y jugando a videojuegos, a mis compañeros de la universidad, con los que aprendo y me entretengo a partes iguales y con los compañeros y profesores del laboratorio, que me guían y me ayudan a mejorar como estudiante y como persona. A todos ellos les debo este trabajo, sin ellos a mi lado nada de esto hubiera sido posible, y creo que por ellos merecen un sincero agradecimiento y un fuerte abrazo. Si tan solo les otorgara una pequeña parte de todo lo que me aportan, ya tendría motivos de sobra para sentirme una persona realizada y en paz conmigo mismo. No podría imaginar una vida sin ellos.

Parte 2: ¿Qué es la realidad virtual?

La **realidad virtual (de ahora en adelante nos referiremos a este concepto con las siglas VR)** es un concepto que ha entrado en nuestras vidas hace relativamente poco tiempo, si bien sus orígenes y desarrollo se remonta a finales del siglo pasado. Es una tecnología emergente que está haciéndose un hueco en sectores tan diversos como los **videojuegos** (el sector al que más frecuentemente está asociada la **VR**), el **modelado 3D**, el **turismo** o la **industria automovilística**. No en vano, ofrece la posibilidad de experimentar sensaciones y realizar actividades muy difíciles o, directamente imposibles, en un entorno simulado haciendo uso de un **visor** y unos **controladores**.



Además de los videojuegos, otros sectores apuestan por la realidad virtual, un buen ejemplo sería el ***diseño de automóviles***

*Otro campo al cual se aplica la VR es al **turismo**, con el cual puedes descubrir experiencias y lugares nuevos desde la comodidad del hogar*

Ahora que hemos visto algunos de los beneficios de la realidad virtual, podemos entrar a explicar en qué consiste la misma, como se ejecuta y como se aplica y usa.

La realidad virtual consiste en, mediante diversas técnicas informáticas, **simular un entorno tridimensional** visible a través del cual podremos **replicar acciones** que estamos realizando en el mundo real, por ejemplo, acciones como **girar la cabeza** o **cerrar los dedos** de una mano se reproducirían de la misma manera en el entorno virtual, haciendo que el mundo se vea como lo verías con la cabeza girada y **realizando las acciones correspondientes** a cerrar los dedos de una mano. En **entornos informáticos convencionales** el entorno virtual suele representarse en una **pantalla bidimensional** y las acciones realizadas en el mundo ficticio no se corresponden con las que realizas tu como usuario. Por ejemplo, en un videojuego puedes mover a tu avatar hacia el frente presionando la tecla "W", sin embargo, tu no estas realizando per se la acción de desplazarte hacia el frente. Este sistema, comúnmente usado por todos, es sencillo, pero no genera la **sensación de inmersividad** que llega a generar la realidad virtual.

Ahora la pregunta que nos surgiría es, **¿cómo podemos ejecutar estos programas de realidad virtual?** La respuesta puede tener muchas contestaciones posibles, ya que existen diversos sistemas para introducirte en un entorno digital, sin embargo, todos coinciden en la utilización de **dos componentes hardware**:

-**Un visor**: Este elemento nos permite **visualizar el entorno simulado.** Se corresponden a nuestros ojos y nuestros oídos del mundo real. El visor tiene la capacidad de **conocer su propia rotación**, ya que es necesaria para saber si nuestra cabeza ha girado. Suelen tener incorporados unos **auriculares integrados** que hacen los efectos de sonido para dar una sensación mayor de tridimensionalidad. Este visor debe ejecutar los fotogramas a una **mayor velocidad** que en un programa real, para evitar que el usuario experimente lo que en ingles conocemos como "**motion sickness**" (malestar en el usuario causado por la falta de correspondencia de tus acciones, la cual produce generalmente mareos) además de para mostrar un entorno lo más parecido al mundo real posible, sin cortes ni efectos borrosos.

-**Dos controladores**: Cada uno de ellos corresponde a una de las **manos del usuario**, estos dos elementos deben saber en todo momento su **posición y rotación** para poder mostrarlo en el entorno virtual de forma acorde a como rotamos y movemos la mano en el mundo real. Los mandos poseen una serie de botones que permiten **realizar acciones con los dedos de una mano**, como cerrar la mano, señalar, coger objetos virtuales…

*De izquierda a derecha: Controlador de la mano izquierda, visor VR y controlador de la mano derecha (**Valve Index**)*

Las aplicaciones de este sistema son infinitas, desde **videojuegos** de muchos ámbitos hasta **montaje de piezas de un robot** pasando por **clases virtuales** y **modelado 3D**. Estas aplicaciones pueden comprarse y descargarse en el visor o en un ordenador conectado al visor para su posterior ejecución.

Más información

FACTURACIÓN DE LA LA INDUSTRIA DE RV EN 2016
2,7 mil. USD

FACTURACIÓN POR VENTA DE HARDWARE DE RV PREVISTA PARA 2020
17.800 mil. USD

FACTURACIÓN POR VENTA DE SOFTWARE DE RV PREVISTA PARA 2020
19.900 mil. USD

El mercado de realidad virtual y aumentada crecerá un 78,5% en 2020

En cifras 02 DIC 2019

Los casos de uso empresarial representarán casi la mitad de todo el gasto en AR / VR, liderados por casos de uso de formación y mantenimiento industrial, siendo retail y fabricación discreta los sectores que más

El futuro de la realidad virtual parece prometedor, y su impacto en la economía va a hacerse notar cada vez más. (es.statista.com) (www.itreseller.com)

Varios millones de personas usan dispositivos VR con asiduidad, y según las expectativas, esta cifra va a ir aumentando de forma cada vez más pronunciada, y, con ello, los **beneficios económicos y sociales de estas herramientas**.

Parte 3: Introducción al proyecto

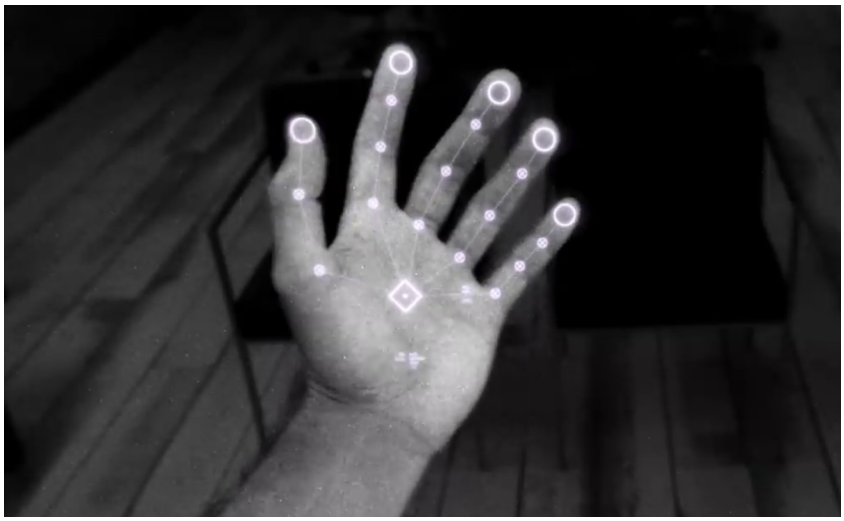
La realidad virtual es un **campo emergente**, lleno de oportunidades e ideas con las que desarrollar experiencias nunca antes vividas en la **interacción hombre-máquina**, sin embargo, aun estamos muy lejos de crear un realidad virtual similar a la realidad cotidiana. Si bien podemos ver a través de los visores y mover nuestros brazos y cuerpo por el entorno, existen ciertos sentidos que se quedan atrás, como son el del **olfato y el tacto**. Mis esfuerzos de cara al proyecto de fin de grado se han centrado en **tratar de incluir otros sentidos o controladores a la realidad virtual**:

- **Control de las piernas**: Una primera idea que me surgió fue darle un mayor peso al control de las piernas, las cuales están completamente relegadas en favor de los brazos. Hay diversas formas de movimiento en los juegos de realidad virtual, entre ellas destacan el **desplazamiento con el joystick**, el **tele-transporte** usando los **botones de los controladores**, o, en algunos juegos, la **detección del movimiento de las piernas** midiendo el movimiento del propio visor, por desgracia estas ideas, aunque fáciles de implementar, no son tan realistas como el control de las piernas. Mi idea fue crear **unas tobilleras capaces de recrear la sensación de caminar**, detectando el levantamiento de las piernas. La idea fue finalmente descartada debido a su complejidad y a la existencia de diversos elementos hardware que realizan esa función, como son los "**Cybershoes**" o el "**KatWalk**", siendo este último un sistema que simula el desplazamiento mediante una cinta estática, permitiendo también agacharse y rotar.

*"**Cybershoes**", un sistema de desplazamiento en realidad virtual*

*El "**KatWalk**" es un sistema casi completo de realidad virtual, generando de esa forma una experiencia inalcanzable con el hardware estándar de VR, por desgracia su precio asciende hasta los 3.000€*

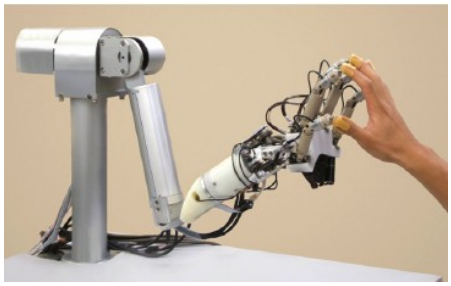
- **Guantes en vez de controladores**: Otra idea que surgió fue la **sustitución de un componente hardware** clásico como es un controlador agarrado con las manos por un **wearable**, mucho más práctico y sencillo de portar. La idea fue descartada debido a su inmensa complejidad, pero, durante el transcurso de este año de proyecto, las gafas utilizadas para la realización del trabajo de fin de grado recibieron una actualización gracias a la cual puedes **navegar por las aplicaciones sin la necesidad de controladores**, usando tan solo tus propias manos



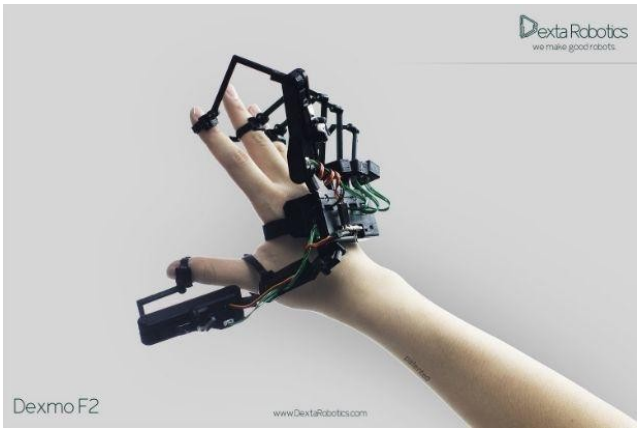
*La función de **Oculus** de "**Hand tracking**", permite usar tus manos como controladores, siendo estas monitorizadas usando las cámaras integradas del propio visor.*

- **Simulación de fuerzas** (**háptica**): Uno de los elementos a mejorar en la realidad virtual es la **simulación del tacto**, aunque también es, tal vez, el sentido más difícil de simular de entre los cinco sentidos. Al jugar un juego en realidad virtual **no sientes ningún tipo de interacción** con la materia del entorno. Si intentas coger un objeto lo atraviesas o, si levantas un objeto aparentemente pesado lo levantarás con la misma facilidad con la que levantas tu propio brazo en el mundo real. Existen diversos elementos, sobretodo a pequeña escala, para **simular**

distintas sensaciones de tacto, utilizando para ello diversos métodos para generar la sensación. Estos son 3 ejemplos de hardware dedicado a **reforzar el sentido del tacto**. Finalmente, tras varias semanas deliberando **decidimos realizar un proyecto centrado en este campo**, pero que ofreciera como elemento diferenciador la **sencillez** además de un **bajo coste de fabricación**.



*Un ejemplo sería "**HIRO II** (**Haptic Interface Robot**)" (1), el cual sigue a tu mano y hace una fuerza en la misma para dar la sensación de que tu mano está tocando ciertos objetos.*



*Otro ejemplo es "**Dexmo F2**", el cual realiza el efecto de tacto y volumen de objetos bloqueando tus propios dedos*

*El "**HaptX**" tal vez sea uno de los equipos más completos que existen en la actualidad, consiste en un par de guantes que, al igual que el "**Dexmo F2**", genera la sensación de volumen en objetos mediante la constricción de los dedos.*

Otros ejemplos serían (2), (3), (4), (5) y (6)

Este sistema estaría enfocado en la **sensación de tacto de las manos**, más específicamente en la **tracción entre 2 elementos unidos uno a cada mano**, como podrían ser un arco, un tirachinas o una banda elástica para hacer ejercicio.

En el ejemplo de la derecha muestro mediante 2 flechas la **fuerza de atracción** entre la mano que sujeta el flexor y la mano que sujeta la cuerda.

Parte 4: Objetivos del proyecto

El objetivo del proyecto es **crear un sistema simple capaz de producir una fuerza de atracción entre las 2 manos del usuario, de forma controlada por una aplicación de realidad virtual de desarrollo propio**, la comunicación entre el casco de realidad virtual y este sistema será inalámbrica. Para hacer la prueba en entornos diversos hemos creado **tres escenarios**. Una aplicación de **tiro con arco** en la que, además, puedes comprobar tu precisión contra una diana comprobando así tu proximidad al centro de la misma, otra aplicación en la que se **generar aleatoriamente una serie de planetas** pares, en la que cada uno de estos estará relacionado con otro con el que exista una fuerza gravitatoria de atracción cuando ambos planetas estén siendo agarrados por el jugador, y, una última aplicación en la que puedes **pulsar un botón** con resistencia a ser pulsado, como un botón en el mundo real.

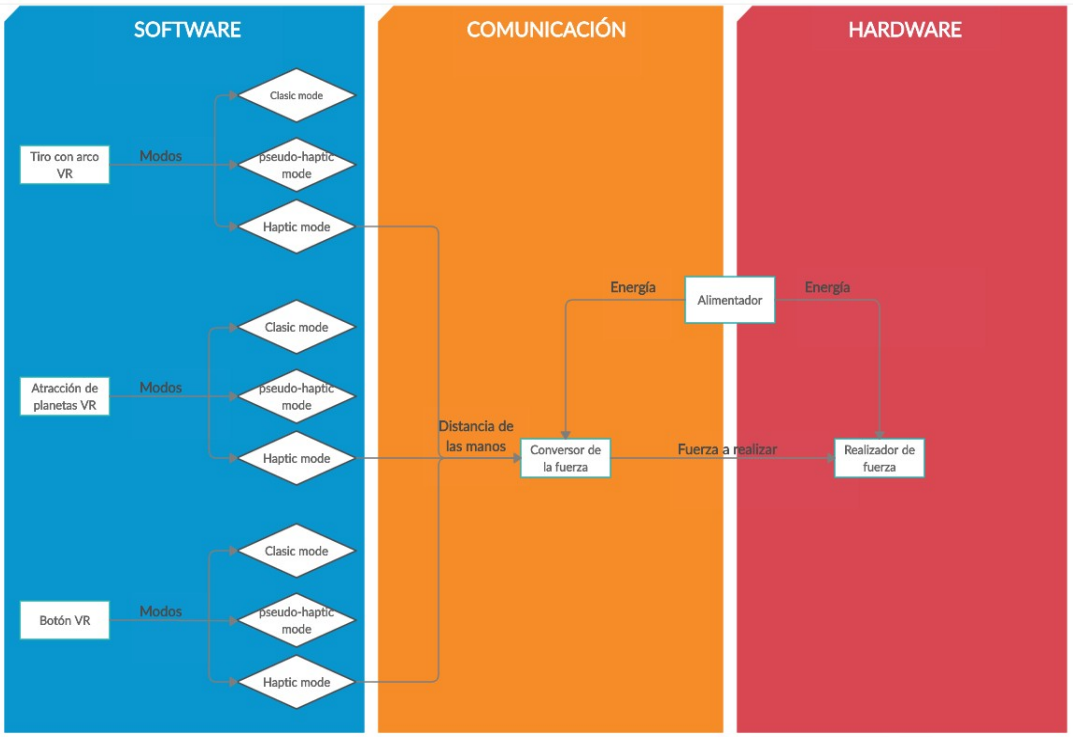
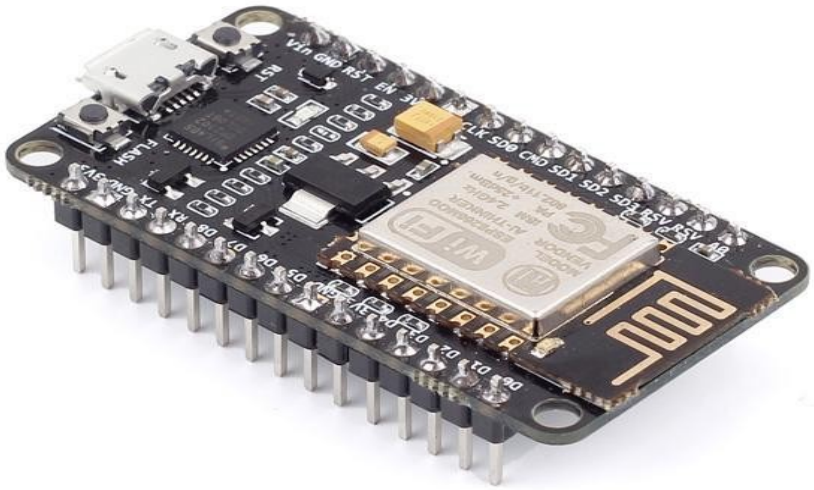


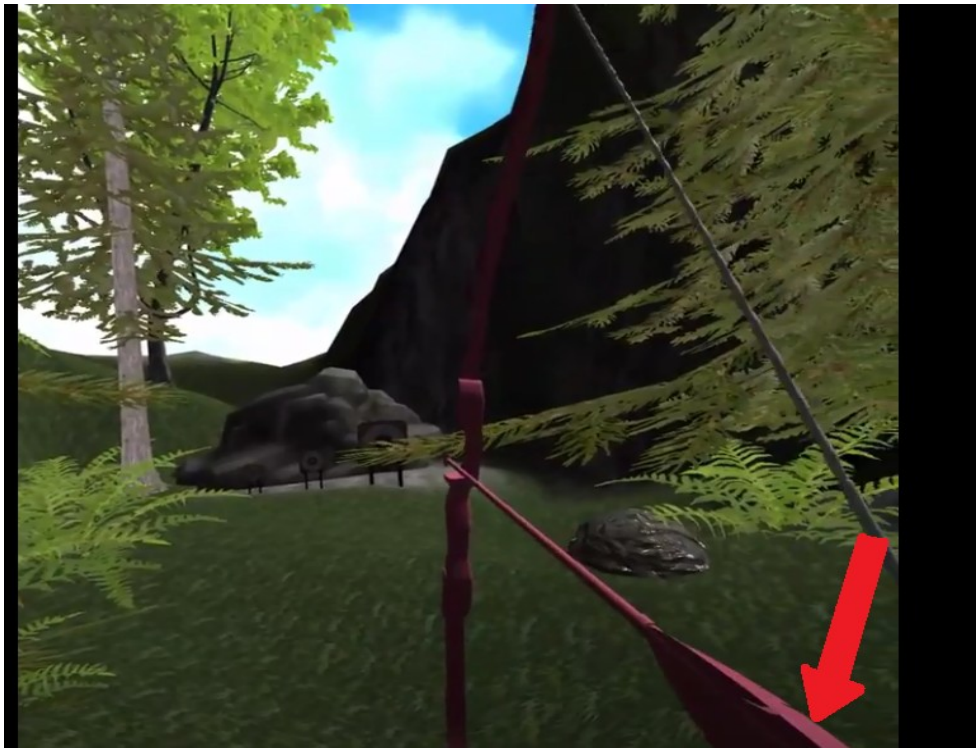
Diagrama del proyecto

Este diagrama representaría un resumen de todo el proyecto, tanto la parte software, realizada principalmente en Unity3D, como la parte hardware realizada utilizando un **circuito electrónico** y una placa **arduino** "**ESP8266**". La funcionalidad de las comunicaciones se halla entre el software puro y el hardware puro, combinando elementos de **programación de comunicaciones** tanto en el **script de Unity3D** como en el s**ketch de la placa arduino**, haciendo uso del hardware de este último también.



*Como piedra angular del hardware y las comunicaciones hemos utilizado un **microcontrolador ESP8266**, ya que, además de las funciones de una placa arduino convencional, ofrece la posibilidad de crear un **Access Point** y establecer comunicaciones con otros dispositivos.*

Primero explicaré un resumen de los objetivos y componentes del software del trabajo. Nuestro objetivo consiste en crear **3 aplicaciones de realidad virtual** para probar en cada uno de ellos a su vez **3 modos distintos de uso**. El **modo clásico** corresponde a la forma más habitual de utilización de la realidad virtual, un modo sencillo pero que no ofrece ningún tipo de respuesta física. Sería el estándar de juegos en VR, donde **no sientes ningún tipo de fuerza** en tus articulaciones. El segundo modo, un poco más complejo, sería la recreación de un **modo "pseudo-háptico"**, que ofrece como ventaja una **mayor respuesta física** al jugador **sin requerir de hardware externo** para realizar estas fuerzas, para ello utilizaríamos **funcionalidades de los controladores** de realidad virtual o **efectos psicológicos** en el jugador, un ejemplo de funcionalidad de los controladores sería la vibración de los mandos, con la cual podremos realizar una fuerza continua de vibración durante el tiempo y con la amplitud deseada, mientras que un efecto psicológico sería hacer vibrar virtualmente la mano del jugador en base a la tensión entre las manos. Estos sistemas se explicarán más en profundidad en la parte de Software de la memoria. Por último, tendremos un tercer modo más complejo y más realista que los dos anteriores, el **modo "háptico"**, en el cual **se generan fuerzas físicas auténticas** mediante un hardware externo.



*Ejemplo: En la aplicación del arco, mostrada en esta foto, aplicaríamos una **vibración sobre la mano** que estira la cuerda, ya sea una vibración real usando las funciones de vibración del controlador o generando esa impresión moviendo de forma aleatoria la posición virtual de esta mano.*

El objetivo dentro del marco de la comunicación era ser capaces de **enviar la tensión generada en cada uno de los proyectos** en tiempo real al hardware para que este actuara en consecuencia, preferiblemente mediante una **conexión inalámbrica**. La solución finalmente pasó por crear un "**Access point**" usando el microcontrolador **Esp8366**, y que este hiciera a su vez de **servidor**, escuchando en todo momento los paquetes enviados por las gafas de realidad virtual. El protocolo utilizado finalmente ha sido **UDP**, debido a que es ligeramente más rápido que TCP, ya que no requiere un ACK por parte del receptor, y, en este proyecto **podemos operar con una tasa de pérdidas grande** gracias a la frecuencia de envío de los paquetes.

Por último haré un resumen de los objetivos del hardware y un esquema del mismo. Se trata del factor más complicado del proyecto, ya que se apoya sobre la base del software, pero además combina conceptos electrónicos mucho más alejados del marco de la informática. En la imagen de abajo muestro un **diagrama de la electrónica del proyecto**, además de una imagen de la **carcasa que une el motor y la cuerda**:

- **Electrónica**: El objetivo del circuito electrónico es **permitir al motor girar dependiendo de la señal enviada al transistor**, cuanto más fuerte sea más veces por segundo permitirá al motor activarse, dando una sensación de mayor fuerza, aún **utilizando una señal digital**.

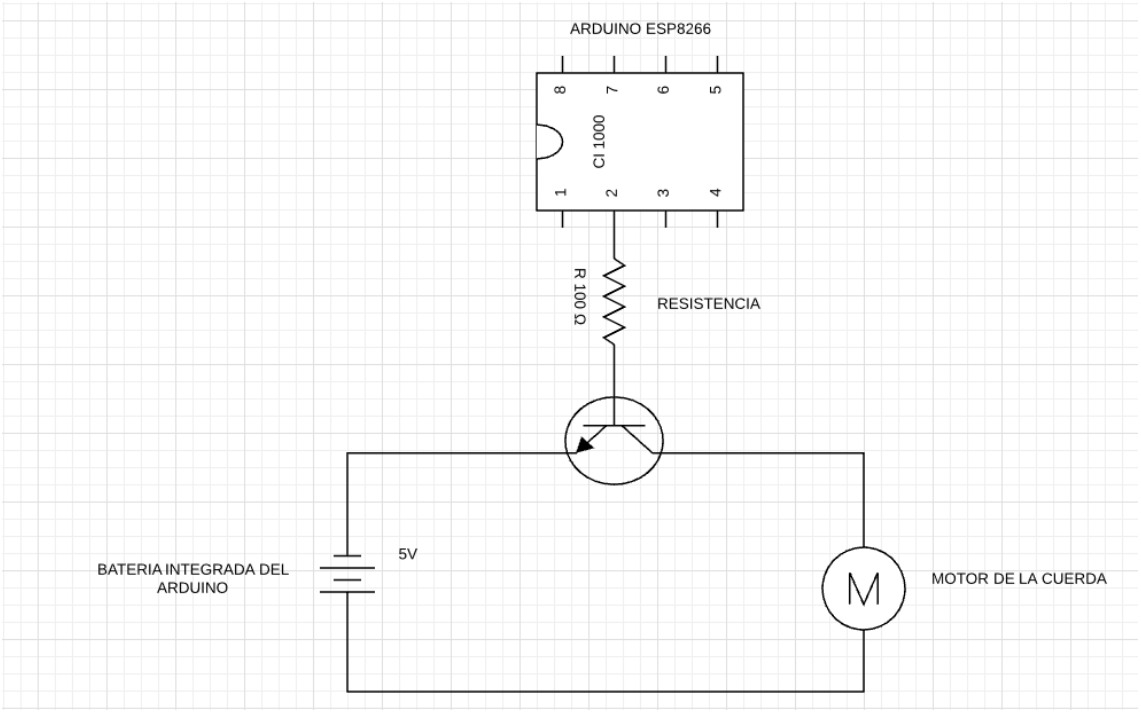
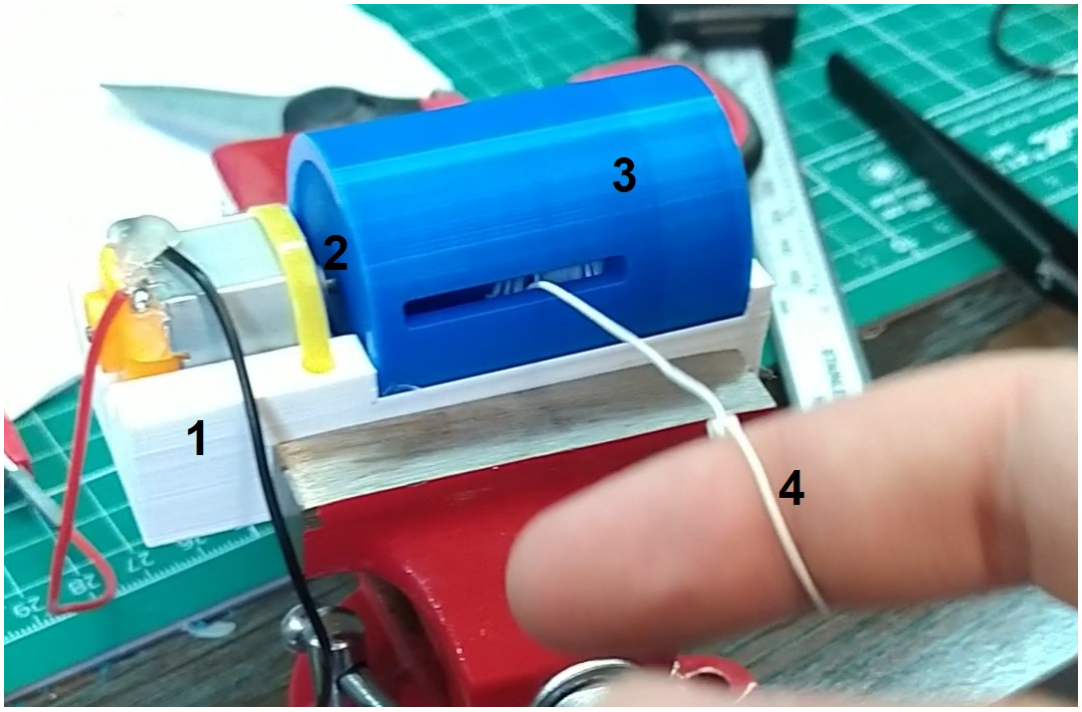


***Diagrama electrónico simplificado del hardware**. Aunque en el proyecto definitivo utilizamos como fuente de alimentación los propios pines de la placa arduino (los cuales generan de forma continua 5V), para una mayor claridad he extraído la batería como si fuera un elemento externo.*

- **Mecánica**: Para generar la sensación de tensión **vamos a utilizar una cuerda que se enrolla sobre un eje con la ayuda del motor** antes nombrado, para ello hemos impreso una serie de piezas con la ayuda de una **impresora 3D**. Estas piezas consisten en: un **eje**, en el cual **encajaría el motor para hacerlo girar**, y sobre el cual se enrollará la cuerda, **un cilindro con una ranura** para permitir la salida de la cuerda al exterior y a su vez **proteger la parte interna del eje**, **una base** para **sujetar todas las piezas** antes nombradas además del motor, y, en la medida de lo posible, **evitar deslizamientos del motor y las piezas** que harían que el mismo perdiera fuerza y diera una sensación desagradable al usuario. Por último hemos añadido **una anilla** al extremo de la cuerda que no se halla unido al eje para **facilitar su manejo** y a la vez evitar, por el propio tamaño de esta, que la cuerda se inserte en la ranura del cilindro y la cuerda se enrolle sobre sí misma.



*En esta imagen se muestra un **prototipo** previo del sistema de recogida de cuerda. **1) Base** de sujeción. **2) Eje** de rotación de la cuerda. **3) Cilindro** de protección y ranura de salida de la cuerda. **4) Anilla** (sería añadida en un futuro).*

En esencia es un sistema similar al del carrete de una caña de pescar en cuanto al giro de la cuerda, pero ofreciendo un **cierto grado de inteligencia** con ayuda de componentes electrónicos y control software.

En resumen, los objetivos del proyecto serían:

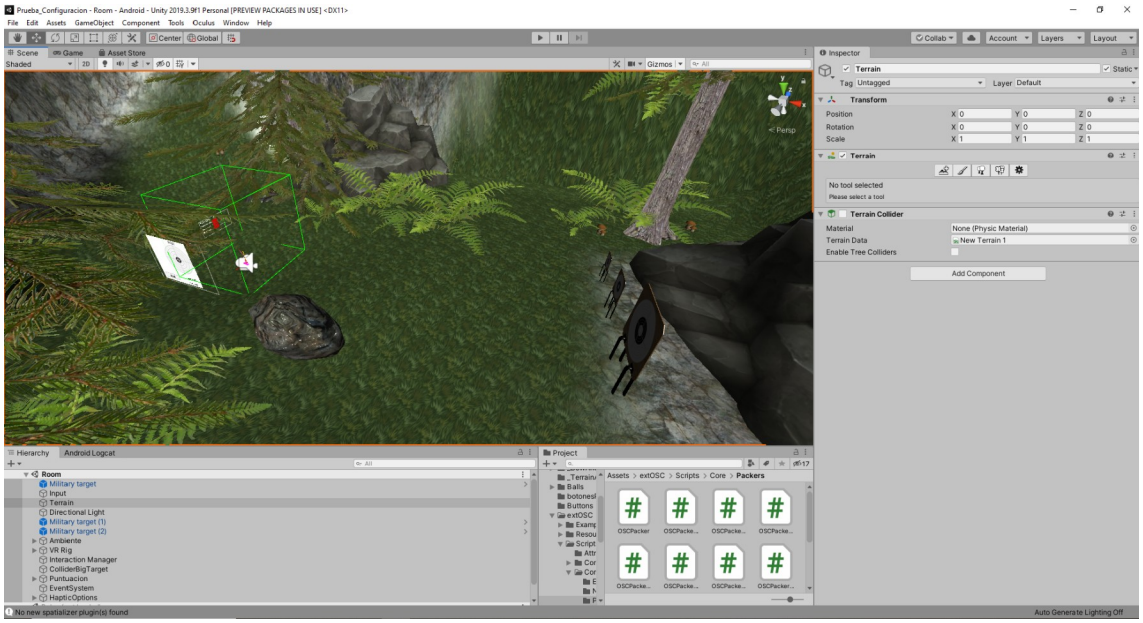
- **Crear** una serie de **aplicaciones en realidad virtual** (Arco, planetas y botón).
- **Dotar** a estas aplicaciones de **3 modos de uso** (Clásico, pseudo-háptico y háptico).
- Ser capaz de **enviar** desde cada uno de estos proyectos el **valor de la tensión** realizada en el entorno virtual.
- Ser capaz, desde el microcontrolador arduino, de **crear una red con la que comunicarse** las gafas de realidad virtual.
- Ser capaz de **recibir** desde el microprocesador **el valor enviado por las gafas de realidad virtual** y modificarlo a conveniencia.
- Desde el microcontrolador, **controlar el motor en tiempo real** para que realice la fuerza acorde a la fuerza realizada en el mundo virtual, y, que de esta forma se recoja la cuerda con la fuerza adecuada para generar tensión entre las 2 manos.

Parte 5: Herramientas utilizadas

Para llevar a cabo el proyecto se ha hecho uso de una serie de **programas** y **componentes**, mostraré un listado de los mismos y su uso en este proyecto en concreto:

- **Programas utilizados**:
    - **Unity 3D**: Se trata de un motor de videojuego desarrollado por **Unity Technologies**. Para el desarrollo de proyectos en esta herramienta se utiliza una serie de **scripts** programados en **C#**, ampliado con librerías propias del motor. Los scripts poseen una estructura compuesta normalmente por **dos funciones** predeterminadas, **Start()**, la cual **se ejecuta tan solo una vez**, cuando el objeto al que pertenece el script entra en escena, y la función **Update()**, la cual **se ejecuta continuamente** mientras el objeto al cual pertenece el script esté activo. Se trata de una herramienta pensada para casi cualquier campo del desarrollo de los videojuegos, desde **animación**, **lógica del juego**, hasta **sonido** y sistema de **físicas** e **iluminación**, pudiendo además **importar modelos tridimensionales o bidimensionales** de otras plataformas como blender (aunque para este proyecto se han usado modelos libres de copyright).

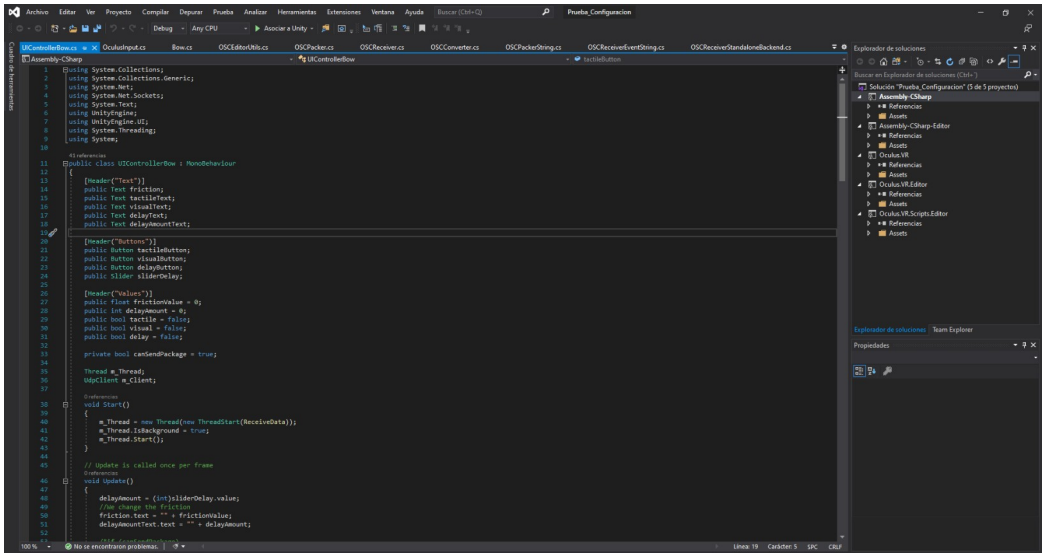
      Se ha tratado de la **herramienta software más utilizada durante el proyecto**, debido en parte a que, aunque se trate de un proyecto que fusiona hardware y software, un **85% del mismo ha consistido en software**. He utilizado en casi su plenitud de funciones esta herramienta para el desarrollo de las aplicaciones, agregando a la misma **sonidos**, **modelos**, un **ambiente** y una **lógica de juego**. Para toda la funcionalidad relacionada con la realidad virtual he utilizado además, una librería conocida como "**Oculus Integration**", la cual está diseñada para la interacción en un entorno VR ejecutado en unas gafas "**Oculus quest**" u "**Oculus Rift**". Más tarde, debido a cambios en la industria, se estandarizaron las diversas librerías de realidad aumentada, virtual y mixta a un único entorno, "**Unity XR Toolkit**".



*En esta imagen se muestra la **interfaz gráfica de Unity**, en este caso mostrando una de las aplicaciones de este proyecto.*

- **Microsoft Visual Studio**: Para programar el código de Unity3D, se te ofrece por defecto la **IDE** "Microsoft Visual Studio". Esta herramienta, desarrollada y mantenida por Microsoft, te permite configurarla de tal forma que se resalten y **autocompleten las funciones propias de Unity3D**, además de ofrecerte un **molde de la estructura de un script de unity** de forma automática cada vez que creas un nuevo script.

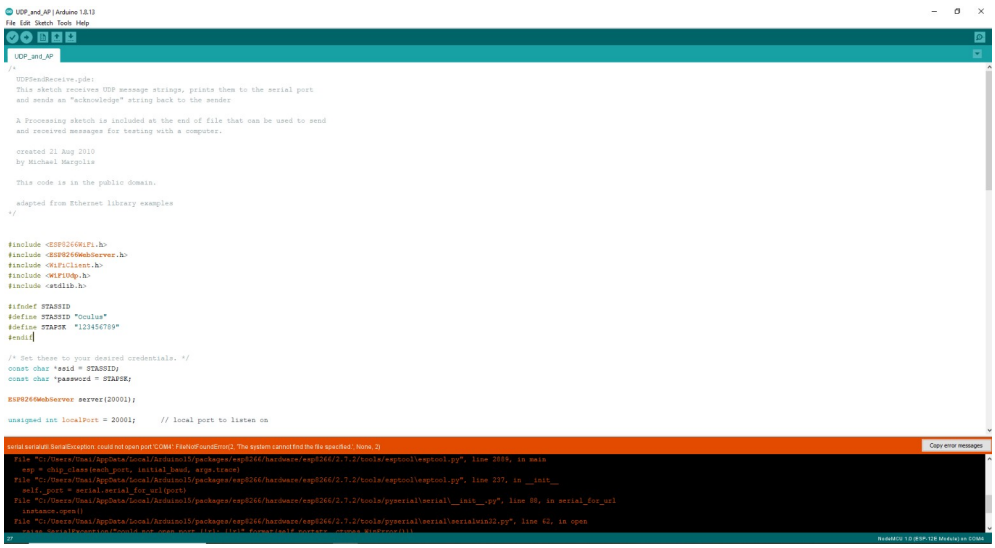
  Se ha tratado del IDE utilizado para el desarrollo del software relacionado con Unity.



*Una de los scripts creados para el proyecto vista en **Microsoft Visual Studio***

- **Arduino**: En la parte de codificación del código para la placa arduino he hecho uso de la **IDE** "Arduino". Un entorno **basado en "Processing"** en el cual puedes crear diversos **sketches** para que estos sean ejecutados en diversos **microcontroladores**. Su estructura es muy similar a la de un script de Unity3D, ya que posee por lo general una función predeterminada "**setup()**", la cual **se ejecuta una única vez** en el momento de activar el microcontrolador, y una función "**loop()**", la cual **se ejecuta en bucle** durante todo el ciclo de vida del código.

  Esta herramienta se ha utilizado para el control del **ESP8266**. Al igual que Unity, posee una serie de funciones propias del programa, además, debes instalar los **drivers** pertinentes para poder actualizar el código del microcontrolador que estés usando en cada momento.



*En esta imagen se muestra la interfaz de "**Arduino**", en este caso enseñando el código para la creación del **access point** del **ESP8266**. La interfaz es mucho **más sencilla que la de Visual Studio**, consistiendo únicamente en una ventana donde se muestra el **código** y otra para visualizar el resultado de la compilación. También se puede visualizar resultados de la consola en **Tools->Serial Monitor***

- **Mendeley**: Se trata de un programa destinado al control y la gestión de diferentes **fuentes bibliográficas** y **papers** de proyectos.

  Al tratarse de un proyecto que toma muchas referencias, se ha tratado de una aplicación de gran ayuda para la gestión de las mismas.

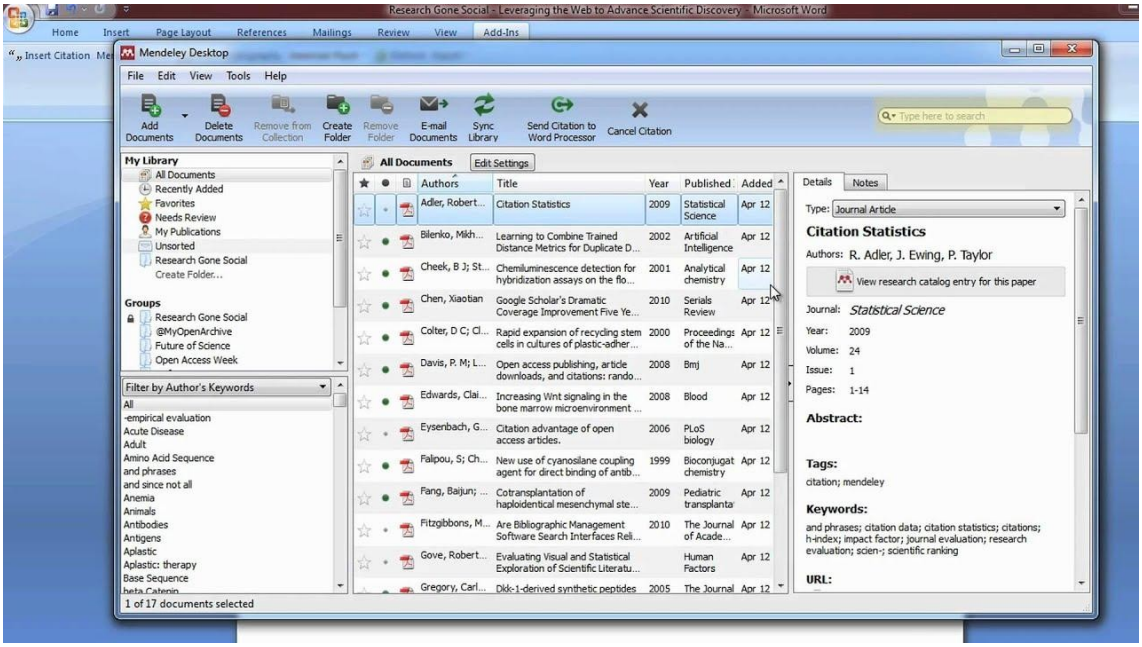


*Imagen de la **interfaz** de Mendeley*

- **Herramientas de hardware**:

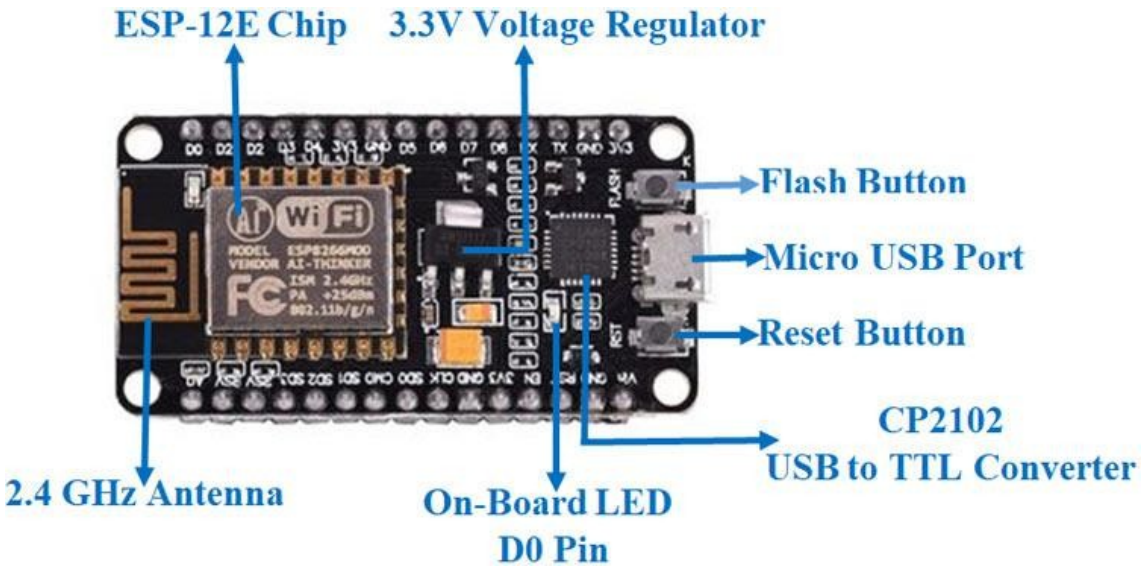
    - **Oculus Quest**: Se trata de un **visor de realidad virtual** creado por **Facebook Inc** lanzado al mercado en **Mayo de 2019**. Posee **dos controladores de realidad virtual**, uno para cada mano. Su principal ventaja con respecto a sus predecesores, como las **Oculus Rift**, es que son **completamente independientes de un ordenador**, lo cual las hace más fáciles de transportar, más cómodas y más baratas en suma. **Posee cuatro cámaras** en los laterales del visor, lo cual permite **procesar el entorno real** y crear un **sistema guardián** en base a él, además de servir para detectar en tiempo real la posición de los controladores. Al ser un producto recientemente distribuido, está dotada de **actualizaciones constantes** que ayudan a ampliar sus funcionalidades. Por ejemplo, recientemente añadieron la **posibilidad de controlar las manos sin necesidad de usar los controladores**, tan solo mediante detección de imagen de las manos. Aunque pueden ser utilizadas sin la necesidad de conectarse a un ordenador, lo cual ofrece una ventaja a la hora de la comodidad, pero también la desventaja de que **todo el software ha de ser ejecutado dentro del procesador** ubicado en el visor, también se puede conectar las gafas mediante cable a un ordenador y utilizar el "**Oculus Link**" para poder usar las "**Oculus Quest**" como si de unas "**Oculus Rift**" se trataran, **solventando así todas las desventajas** que tenía sobre estas últimas. Además de diversas aplicaciones de prueba, las "Oculus Quest" ofrecen una **aplicación móvil** para la realización de pagos, cambios en las opciones o "**mirroring**" de la pantalla del visor a la pantalla del móvil mediante Bluetooth.

      Se trata de este modo del **componente principal de este proyecto** sobre el cual se ejecutan las aplicaciones. Además de la funcionalidad intrínseca al visor, se ha hecho uso de la opción "**háptica**" ofrecida por los controladores, la cual consiste en un control de la vibración de los mandos.

*Imagen de las **Oculus Quest**, las flechas indican las cuatro cámaras laterales que enfocan directamente hacia el **campo visual del usuario.***

- **ESP8266**: Se trata de un **chip** enfocado a la **conectividad** por medio de **WI-FI**. Es habitualmente utilizado debido a sus **buenas prestaciones** y su **bajo coste**, además de la ventaja de poder ofrecer **conectividad inalámbrica**.

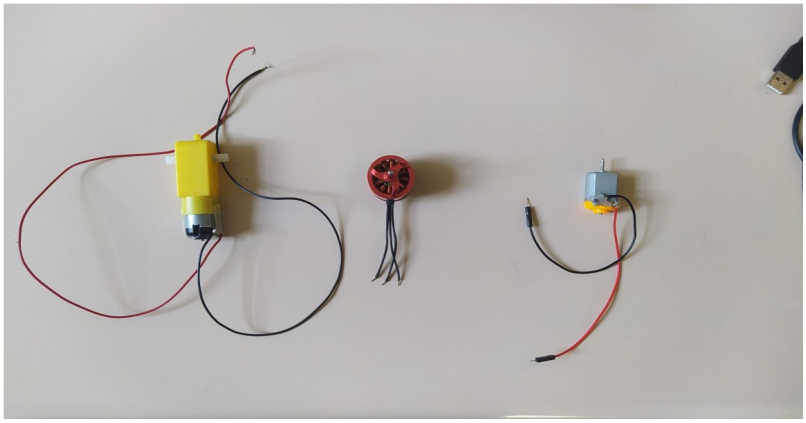
  Su rol en nuestro proyecto es doble, ya que, además de **servir como puente de comunicación** entre el hardware de recogida de cuerda y el visor, también nos sirve para **controlar la fuerza de giro del motor**. Dentro de este chip se ejecuta una función para **crear un access point** de una red privada que une el hardware creado en el proyecto con las gafas y a su vez crea un servidor que pide valores a las Oculus Quest para modificar la fuerza del motor, este programa se llama "**UDP_AP.ino**" y será explicado con más detalle en el capítulo de "Comunicaciones".



*En este esquema del **ESP8266** (**Modelo NodeMCU**) podemos ver los diversos pines de entrada y salida, además, se puede apreciar los componentes utilizados para la comunicación inalámbrica (**ESP -12E** y una **antena de 2.4GHz**). Para la entrada y actualización de código se hace uso de un **puerto Micro USB**.*

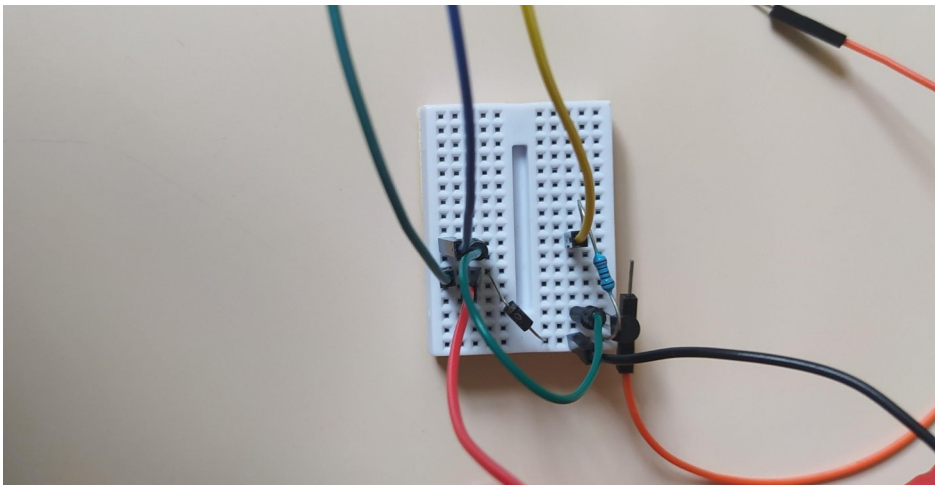
- **Motor**: Para el movimiento de **giro del eje** donde se enrolla la cuerda se ha utilizado un pequeño motor alimentado con **5V** (la energía ha sido suministrada desde la placa ESP8266).

  Se ha barajado el uso de diversos motores de pequeño calibre, tanto con reductora incorporada como sin ella, pero finalmente por comodidad, facilidad de uso y bajo consumo se ha decidido seleccionar un **motor DC**. En el capítulo de Hardware se dará una explicación más detallada de los motores candidatos y las diversas pruebas de consumo, fuerza y resistencia a las que han sido sometidos.



*A la izquierda tenemos un **motor DC con una reductora**, en el centro un **motor "brushless** y a la derecha, un **motor DC simple**, el cual fue finalmente elegido*

- **Componentes electrónicos varios**: En el capítulo de Hardware se hará una mención más detallada sobre los componentes electrónicos que componen el sistema. Estos se han utilizado para **controlar la intensidad a la que está sometida el motor**. Entre estos componentes se incluyen cables, una pequeña "**protoboard**" para unir todos los componentes, **transistores** y **condensadores**.



*En esta imagen se muestra el **circuito electrónico** dibujado en la imagen de la página 12*

- **Componentes mecánicos e impresoras 3D**: Para la realización de las piezas que sujetan el motor y el eje sobre el que rota la cuerda se ha hecho uso de las **impresoras 3D** del laboratorio, se han creado en total tre piezas para este proyecto. Las piezas a su vez fueron diseñadas en "**Catia**" y finalmente para la sujeción de todas las piezas se ha hecho uso de una pareja de **bridas**.

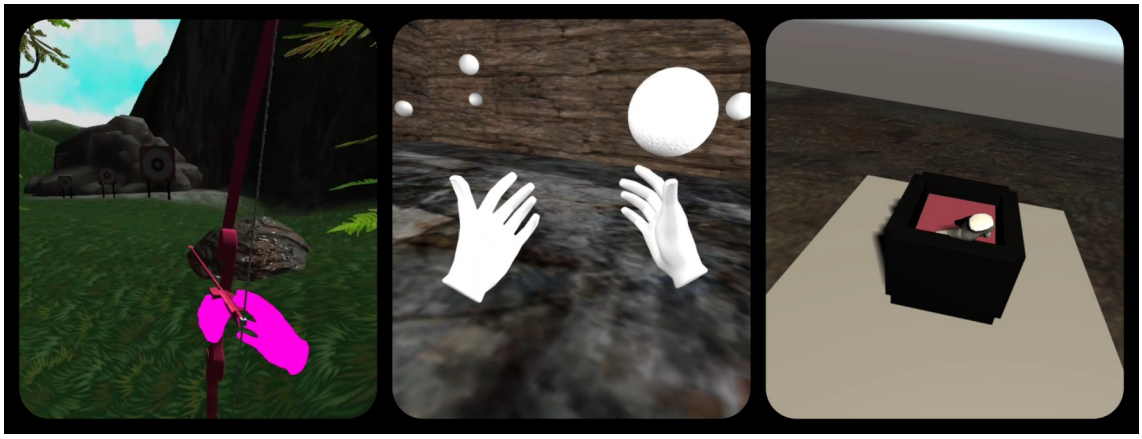
*En el laboratorio de "Profesores Invitados" del departamento de las Encinas hay **8 impresoras 3D**. Con ellas hemos realizado las piezas 1,2 y 3 mostradas en la foto de la página 13.*

CAPÍTULO 1: Software

Parte 1: Introducción

Se ha tratado en su mayoría de un **proyecto enfocado en el software**, ocupando casi **siete de los nueve meses** en los que se ha desarrollado el proyecto. En este capítulo vamos centrarnos exclusivamente al **código desarrollado para las 3 aplicaciones**, y, más específicamente en **dos modos de uso de las mismas**, el **modo clásico**, en el que no se ofrece ningún estímulo háptico y en el modo **pseudo-háptico**, en el que se ofrecen estímulos hápticos sin hacer uso de hardware externo.

Estas tres aplicaciones han sido **desarrolladas en Unity3D** utilizando el lenguaje **C#** para la programación de los **scripts**. En esta parte de la memoria os mostraré el proceso seguido para el **desarrollo de cada una de las aplicaciones exceptuando el código utilizado para la comunicación de las gafas 3D** con el chip ESP8266, el cual será explicado en el siguiente capítulo, dedicado únicamente a las comunicaciones del proyecto.



*De izquierda a derecha. **Tiro con arco** en realidad virtual, **planetas** en realidad virtual y **botón** en realidad virtual.*

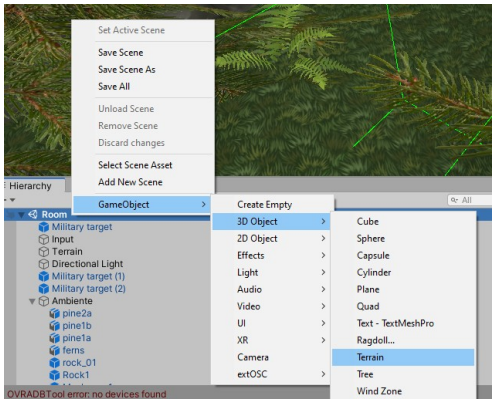
Parte 2: Arco VR

Parte 2.1: Objetivos

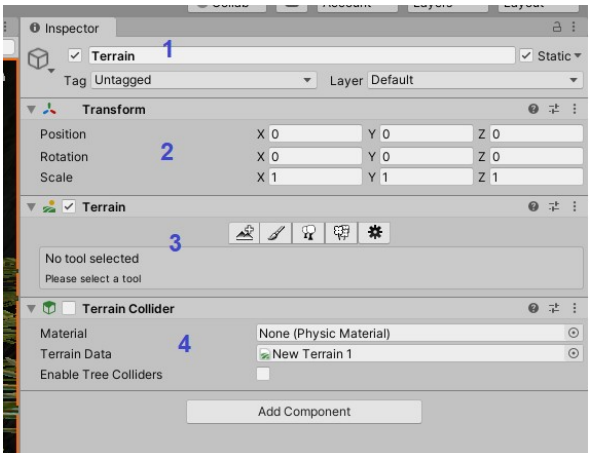
El objetivo de esta aplicación es ofrecer al usuario un **simulador sencillo de tiro con arco**, ofreciendo una **cantidad ilimitada de flechas**, **tres dianas**, una de ellas pudiendo mostrar al usuario la precisión y punto de impacto de las flechas, también posee un **entorno tridimensional para una mayor inmersión** y una **interfaz** para poder **variar entre el modo clásico puro y el modo pseudo-háptico**, pudiendo seleccionar **opciones intermedias** entre ambas.

Parte 2.2: El entorno

El primer elemento creado de la aplicación fue el **ambiente donde estará ubicado el jugador**. Para ello debemos crear un **terreno**, haciendo uso del c**omponente "Terrain"** de Unity3D. Para crear un terreno en Unity3D hacemos clic derecho en un espacio vacío de la jerarquía y seleccionamos "**gameObject -> 3D Object -> Terrain**" como muestra la imagen inferior.

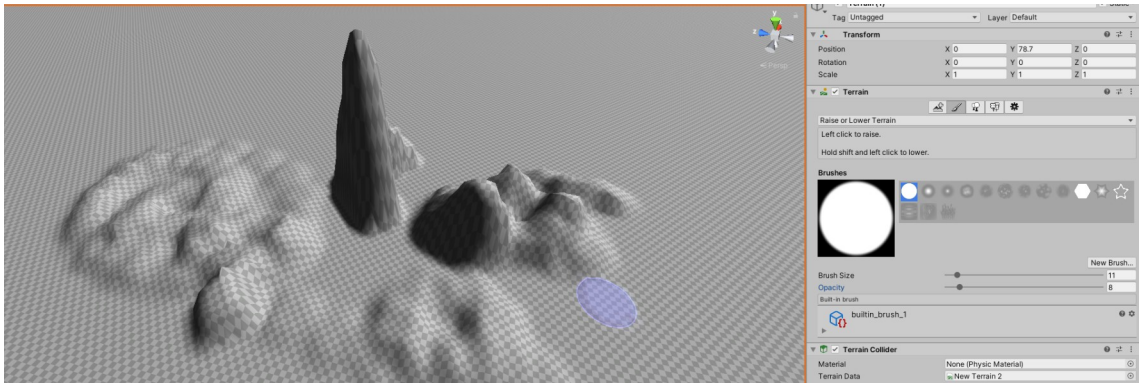


Esta **mecánica es muy recurrente** en el desarrollo de juegos en Unity3D, tanto para **crear primitivas**, **efectos de audio**, **luz**… por lo que será recurrente a lo largo de este capítulo. En la parte del "**Inspector**" del nuevo terreno seleccionado, nos aparecerán **componentes**, los cuales se encargan de **modificar las cualidades de un objeto**.



***1) Nombre** el objeto. **2) Posición, rotación y tamaño** del objeto. **3)** Herramienta de **modificación del terreno. 4) Colisionador** físico del terreno*

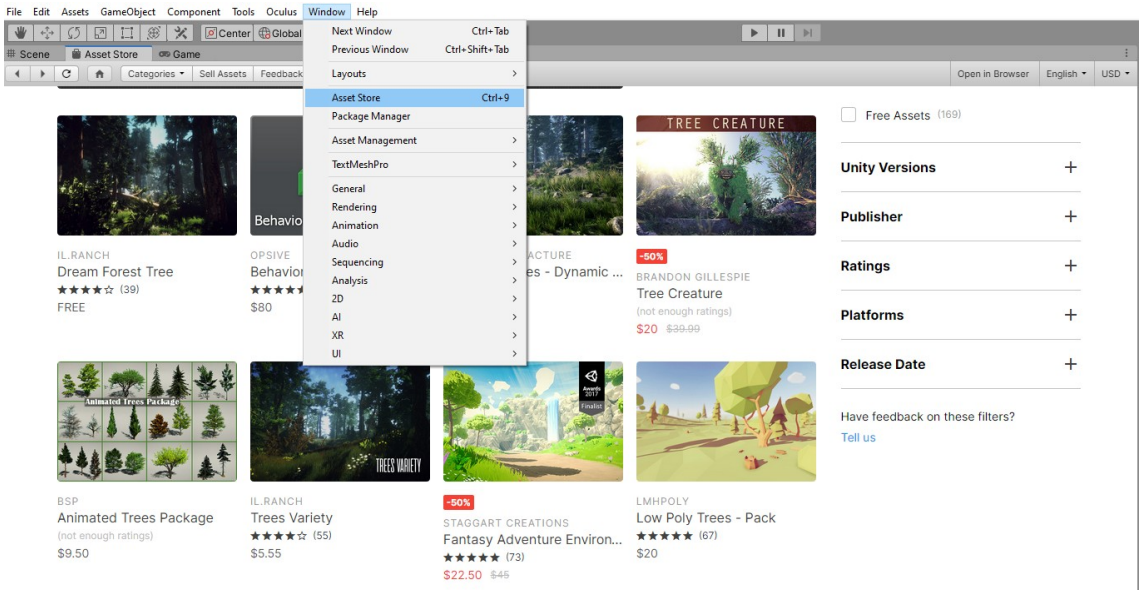
Ahora, para **modificar el terreno** podremos hacerlo **utilizando la tercera pestaña** de la imagen mostrada arriba. Funciona de forma similar a un **pincel virtual**, siendo el objeto terreno un lienzo. Con este pincel puedes **elevar o aplanar terrenos**, **pintar de diversos materiales** o **añadir árboles** y **hierba** entre otros.



*Pequeño ejemplo de **elementos modificados en un terreno** y el pincel circular usado (en color azul)*

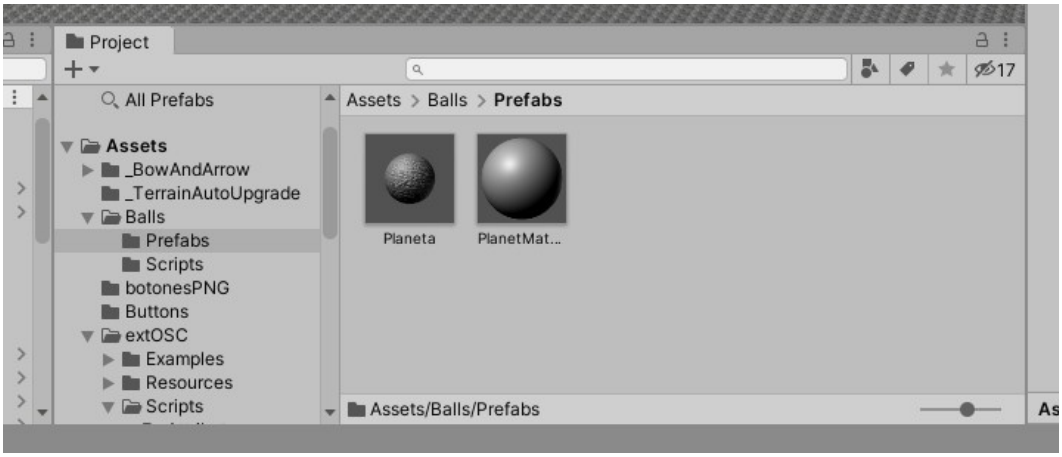
***Terreno** creado para la aplicación del arco en realidad virtual*

Además de modificar la **orografía** del **terreno**, se le pueden **añadir objetos tridimensionales** para dar un mayor realismo. Estos elementos pueden ser **creados** en un programa de modelado 3D u **obtenidos** de la "**Asset Store**" de forma gratuita o de pago.



*En esta imagen se muestra la **interfaz** de la "**Asset Store**" y cómo acceder a ella*

Los objetos pueden ser **importados** a un proyecto, en cuyo caso se almacenarán en la pestaña "**Project**" para ser posteriormente agregados a la escena arrastrandolos.

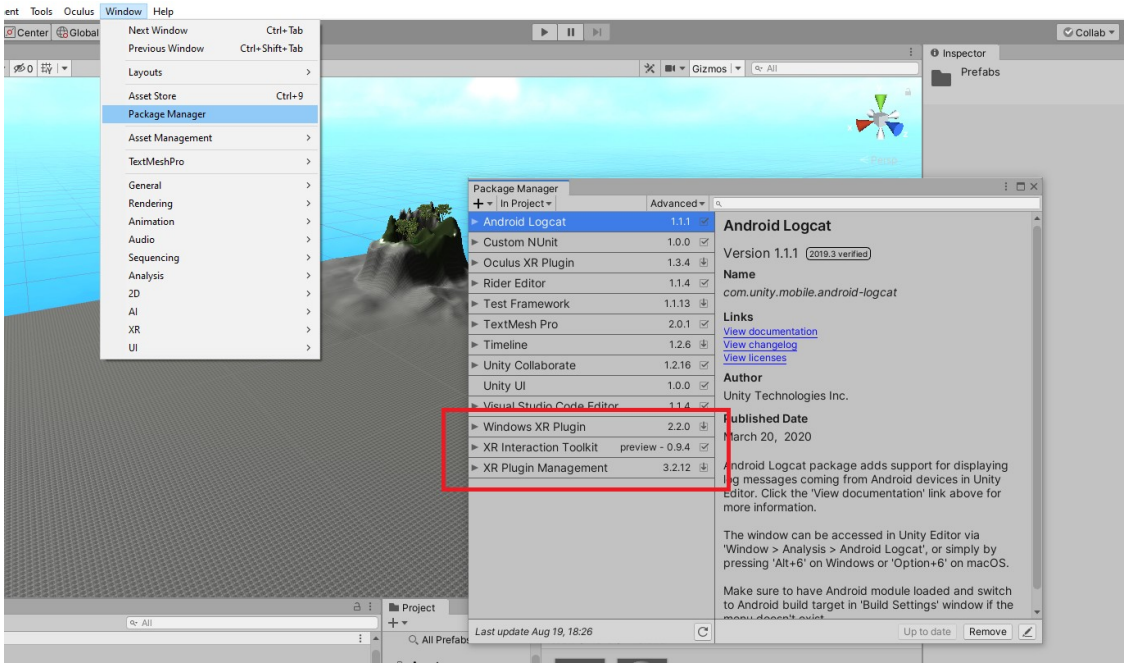


*En la pestaña "**Project**" se almacenan los modelos de objetos, **scripts**, archivos de **audio**, **imágenes**… para poder añadirlos al proyecto.*

Parte 2.3: El jugador

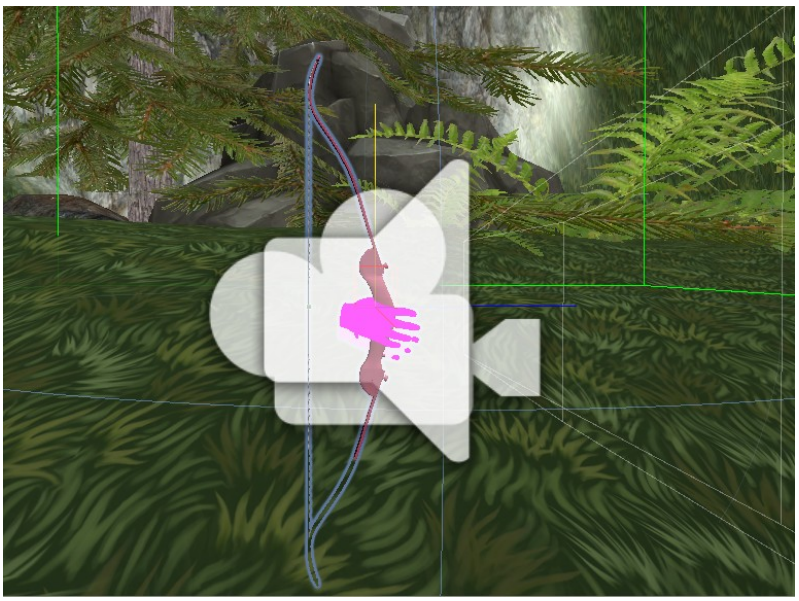
Tras finalizar una primera versión del terreno, creamos al **jugador**. El objeto "jugador" en Unity3D **varía dependiendo de la plataforma** en la que se vaya a ejecutar el programa, por ello, este **procedimiento no servirá para desarrollar juegos en 2D o en 3D para ordenador**.

En primer lugar debemos **descargar e instalar una serie de librerías** mostradas en pantalla, también se muestra como encontrar la pestaña "**Packages manager**" desde donde se gestionan.



*Los tres **paquetes** a descargar*

Ahora mostraré de forma muy resumida en qué consiste un jugador en un entorno de realidad virtual.



*En nuestro caso, un jugador consiste en un **modelos** de **manos**, una **cámara**, un **colisionador** para no caernos del mundo y un **arco**.*

El **objeto 1 contiene a todos los demás**, eso significa que, **si lo movemos, giramos o cambiamos su escala**, proporcionalmente **sus objetos "hijos"** (aquellos que se encuentran por debajo de este) **cambiarán sus propiedades en base a los del objeto "padre"**. En este primer objeto hemos añadido el componente "**XR Rig"**, un script el cual se encuentra en las librerías previamente importadas. Para añadir un componente a un objeto tan solo **debes presionar el botón "Add Component"** en el Inspector y buscar el componente que se desea añadir.

El **objeto 2** está vacío, su único uso es **ordenar** de forma más adecuada sus **objetos "hijo"**.

El **objeto 3** contiene una **cámara**, con la cual el **jugador podrá ver el mundo virtual**, un "**Audio Listener**" para poder escuchar los sonidos del juego y un componente "**Tracked pose Driver**", con el cual se puede saber la **posición y rotación del jugador** equipado con las gafas.

Los **objetos 4** se encargan del **control de las manos del jugador**, dentro de ellas se hallan los componentes "**XR Controller**" (un script de las librerías importadas para poder **mover las manos en el mundo virtual**) y "**XR Direct Controller**" ("un script de las librerías importadas para poder **agarrar objetos virtuales con la mano**").

El **objeto 5** es el **arco**, el cual se explicará en detalle en el siguiente subapartado. Este objeto tiene en todo momento la misma transformada (Combinación de la posición, rotación y escala) que su objeto padre, en este caso la mano izquierda.

Los **objetos 6** se corresponden a los **modelos tridimensionales de las manos**, tomados de entre los modelos por defecto que ofrece Oculus. Su función es meramente estética

*Vistas lateral y frontal del **modelo de la mano izquierda**.*

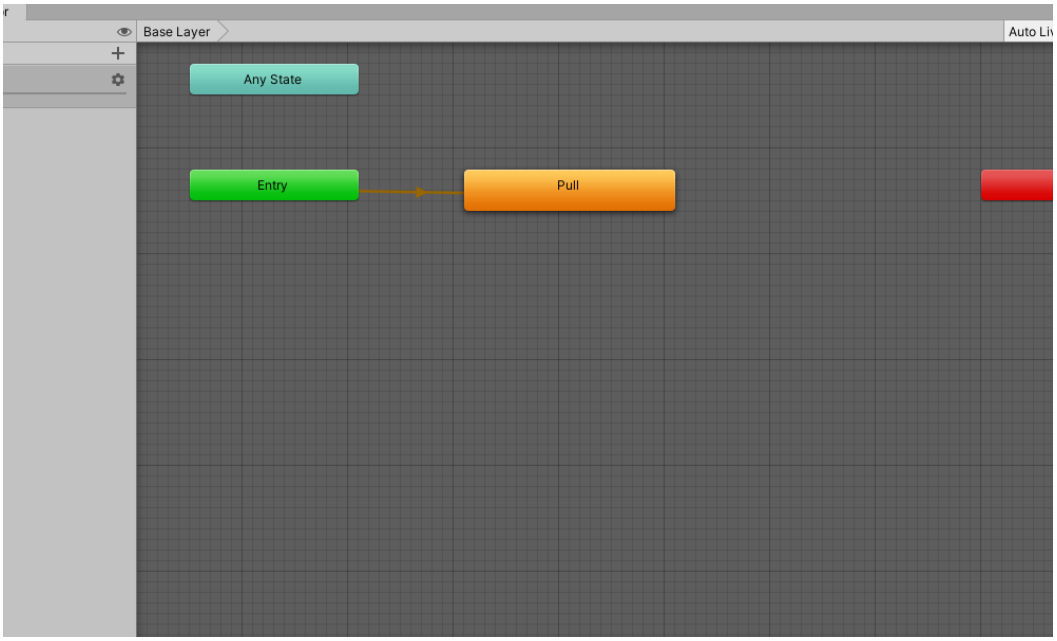
El **objeto 7** tiene como función crear un **puntero con el que interactuar con las interfaces** del juego, será explicado en mayor profundidad en un apartado futuro dedicado a las interfaces gráficas.

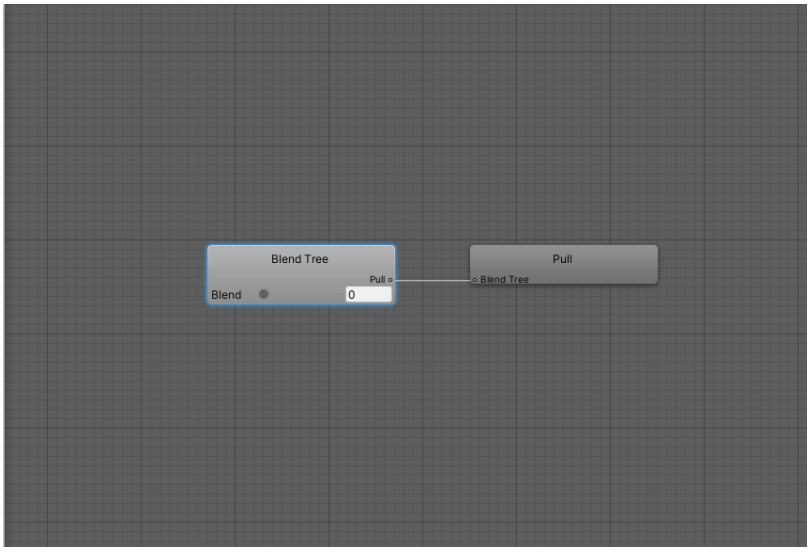
Con todo ello tendríamos un jugador creado en realidad virtual, utilizando para ello los scripts y modelos ofrecidos por las librerías de **XR**.

Parte 2.4: El arco y las flechas

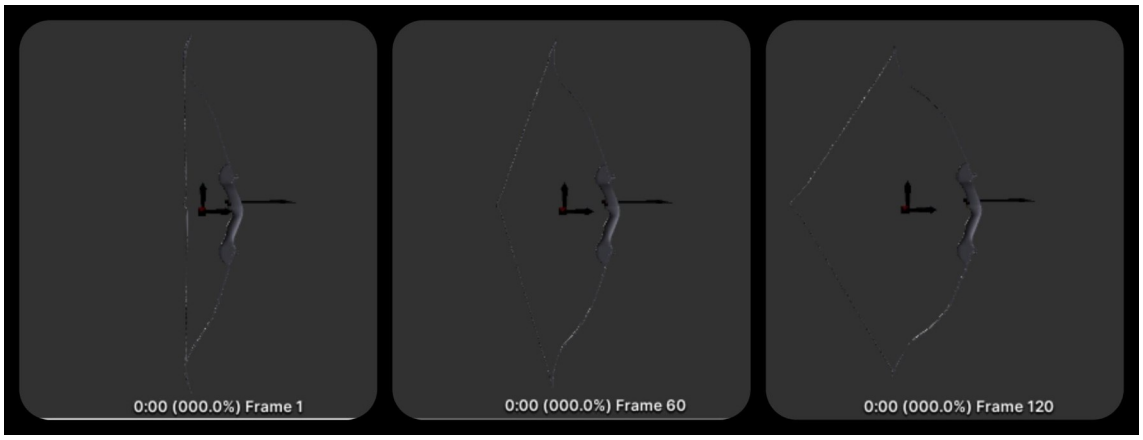
Ahora pasaré a explicar el funcionamiento y proceso de creación del **arco** y las **flechas**. Primero explicaré qué **elementos fueron prefabricados** y cuales han sido **añadidos por mi parte** para adaptarlo a realidad virtual.

En un primer lugar tenemos un **modelo de un arco** y de unas **flechas**, tomado de un **plugin** oficial de "**Steam VR**", apto de forma **gratuita para los usuarios**. El modelo del arco incluye además una **animación condicionada por un float** que puede abarcar **entre 0 y 1**, el cual representa la rigidez del arco.





*En Unity3D el ciclo de las **animaciones** se representa mediante **grafos**, en este caso podemos ver como el nodo inicial pasa a un **nodo** llamado **Pull**, en el cual se mantendrá eternamente tomando el valor "**Blend**" para mostrar la tensión del arco .*



*El arco puede estar en **120 posibles estados de animación** dependiendo de la tensión ejercida. La primera imagen muestra el arco en la **animación de reposo**, el segundo con una **tensión del 50%** y el tercero en su **máxima tensión**.*

*Modelo de la **flecha***

A partir de aquí lo único que falta es **codificar el arco**. Recordemos que el objeto arco es un **hijo de la mano izquierda**, por lo que rotará y se desplazará en base a esta mano, como además hemos hecho que la posición del arco sea la misma que la de la mano no necesitaremos preocuparnos de su movimiento.

Primero debemos conseguir que con la otra mano podamos **pellizcar la cuerda del arco** cuando esta se encuentre cerca, para ello tenemos este script llamado "**OculusInput.cs**".

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using OVR;
using UnityEngine.SceneManagement;
using UnityEngine.XR;
using JetBrains.Annotations;
using System;
using UnityEngine.UI;

1 referencia
public class OculusInput : MonoBehaviour
{
    public Bow m_Bow = null;
    public GameObject m_OppositeController = null;

    public InputDevice m_controller;
    public InputDevice m_controller2;

    public String receivingOSC = "";

    private float formerButton = 0;

    private float primaryButtonValue = 0;
    private bool controllersNotFound = false;

    private bool upScene = true;
    private bool downScene = true;

    private bool previousUpScene = true;
    private bool previousDownScene = true;

    0 referencias
    private void Start()
    {
        List<InputDevice> devices = new List<InputDevice>();

        InputDeviceCharacteristics rightControllerCharacteristics = InputDeviceCharacteristics.Right | InputDeviceCharacteristics.Controller;
        InputDeviceCharacteristics leftControllerCharacteristics = InputDeviceCharacteristics.Left | InputDeviceCharacteristics.Controller;


        InputDevices.GetDevicesWithCharacteristics(rightControllerCharacteristics, devices);

        if(devices.Count == 0)
            //Debug.Log("No se ha encontrado controladores derechos!!!!");

        m_controller = devices[0];

        InputDevices.GetDevicesWithCharacteristics(leftControllerCharacteristics, devices);

        if(devices.Count == 0)
            //Debug.Log("No se ha encontrado controladores izquierdos!!!!");

        m_controller2 = devices[0];

    }

    0 referencias
    private void Update()
    {
```

```
        List<InputDevice> devices = new List<InputDevice>();

        InputDeviceCharacteristics rightControllerCharacteristics = InputDeviceCharacteristics.Right | InputDeviceCharacteristics.Controller;
        InputDeviceCharacteristics leftControllerCharacteristics = InputDeviceCharacteristics.Left | InputDeviceCharacteristics.Controller;

        InputDevices.GetDevicesWithCharacteristics(rightControllerCharacteristics, devices);

        if (devices.Count == 0) {
            //Debug.Log("No se ha encontrado controladores derechos!!!!");
            controllersNotFound = true;
        }
        else {
            m_controller = devices[0];
            controllersNotFound = false;
        }

        InputDevices.GetDevicesWithCharacteristics(leftControllerCharacteristics, devices);

        if (devices.Count == 0){
            //Debug.Log("No se ha encontrado controladores izquierdos!!!!");
            controllersNotFound = true;
        }
        else {
            m_controller2 = devices[0];
            controllersNotFound = false;
        }

        if (!controllersNotFound)
        {
            previousUpScene = upScene;
            previousDownScene = downScene;


            m_controller.TryGetFeatureValue(CommonUsages.primaryButton, out downScene);
            m_controller.TryGetFeatureValue(CommonUsages.secondaryButton, out upScene);

            //Go up
            if (upScene && !previousUpScene)
            {
                SceneManager.LoadScene(1);
            }

            //Go down
            if (downScene && !previousDownScene)
            {
                SceneManager.LoadScene(2);
            }

            formerButton = primaryButtonValue;

            m_controller.TryGetFeatureValue(CommonUsages.trigger, out primaryButtonValue);

            //Debug.Log("Current: " + primaryButtonValue);


            //Debug.Log("Former: " + primaryButtonValue);

            if ((primaryButtonValue > 0.5) && (formerButton <= 0.5))
            {
                Debug.Log("Lets pull the bow");
                m_Bow.Pull(m_OppositeController.transform);
            }

            if ((primaryButtonValue < 0.5) && (formerButton >= 0.5))
            {
                Debug.Log("Lets release the bow");
                m_Bow.Release();
            }

        }
    }
}
```

Primero de todo seleccionamos los **dispositivos VR conectados** en este momento en base a unas **características dadas** (**ser controladores** y ser el **derecho** y el **izquierdo**), en caso de no detectar ninguno **seguirá tratando de seleccionarlos** hasta encontrarlos en la función **Update()**.

```
    List<InputDevice> devices = new List<InputDevice>();

    InputDeviceCharacteristics rightControllerCharacteristics = InputDeviceCharacteristics.Right | InputDeviceCharacteristics.Controller;
    InputDeviceCharacteristics leftControllerCharacteristics = InputDeviceCharacteristics.Left | InputDeviceCharacteristics.Controller;


    InputDevices.GetDevicesWithCharacteristics(rightControllerCharacteristics, devices);

    if(devices.Count == 0)
        //Debug.Log("No se ha encontrado controladores derechos!!!!");

    m_controller = devices[0];

    InputDevices.GetDevicesWithCharacteristics(leftControllerCharacteristics, devices);

    if(devices.Count == 0)
        //Debug.Log("No se ha encontrado controladores izquierdos!!!!");

    m_controller2 = devices[0];

}
```

Después, en la función **Update()** lee los **inputs de los botones del controlador**, siempre y cuando detecte a los controladores. En un primer momento leerá los de los botones "**A**" y "**B**" o "**X**" e "**Y**" (los análogos al A y al B en el controlador izquierdo), y en caso de ser presionado el "**A**" o el "**X**" **dejará la escena del arco y las flechas y moverá al jugador a la siguiente**, en este caso la de los **planetas en realidad virtual**, por otro lado si presiona "**B**" o "**Y**" **avanzará a la tercera escena, la cual en este caso es la del botón en realidad virtual**.

```
previousUpScene = upScene;
previousDownScene = downScene;


m_controller.TryGetFeatureValue(CommonUsages.primaryButton, out downScene);
m_controller.TryGetFeatureValue(CommonUsages.secondaryButton, out upScene);

//Go up
if (upScene && !previousUpScene)
{
    SceneManager.LoadScene(1);
}

//Go down
if (downScene && !previousDownScene)
{
    SceneManager.LoadScene(2);
}
```

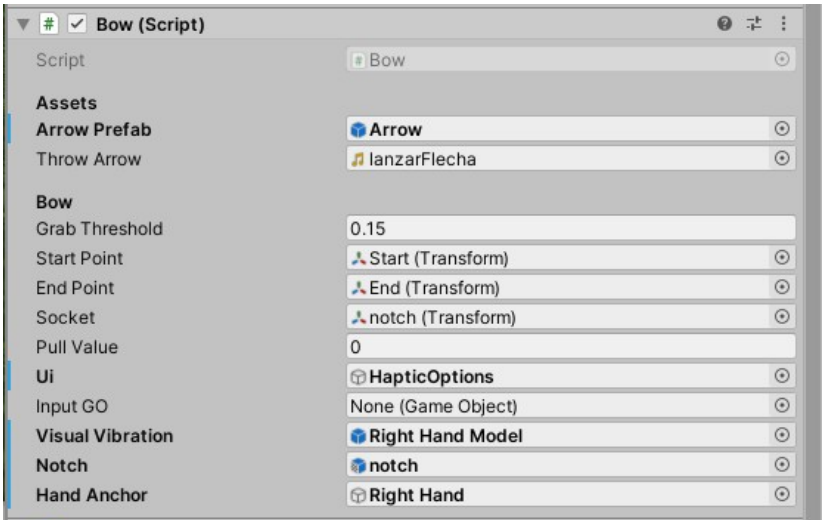
Una vez finalizada la lectura de estos botones, leerá el **input del gatillo trasero del mando**, el cual es un **float entre 0 y 1** que depende de que tan presionado este. Si en la iteración anterior del Update() fue **menor que 0.5** y ahora es **mayor** significa que **hemos pellizcado la cuerda**, si por el contrario en la iteración anterior fue **mayor que 0.5** y ahora es **menor** significa que **la cuerda ha sido soltada**.

```
    formerButton = primaryButtonValue;

    m_controller.TryGetFeatureValue(CommonUsages.trigger, out primaryButtonValue);

    //Debug.Log("Current: " + primaryButtonValue);


    //Debug.Log("Former: " + primaryButtonValue);

    if ((primaryButtonValue > 0.5) && (formerButton <= 0.5))
    {
        Debug.Log("Lets pull the bow");
        m_Bow.Pull(m_OppositeController.transform);
    }

    if ((primaryButtonValue < 0.5) && (formerButton >= 0.5))
    {
        Debug.Log("Lets release the bow");
        m_Bow.Release();
    }
}
```

*En este caso hace uso de **dos funciones** incluidas dentro del objeto público "**m_bow**" la cual es una **referencia al arco del juego**. La función **pull()** toma como valor la transformada del controlador derecho, para así **calcular la tensión del arco**, mientras que la función **release() cambia a 0 esta tensión**.*

Ahora pasaré a mostrar el **código del arco**, el cual está contenido en el Script "**Bow.cs**". Este Script posee las siguientes **variables de entrada**:

- "**Arrow Prefab**" consiste en una **referencia al modelo de la flecha** almacenado en "project".
- "**Throw Arrow**" consiste en un **archivo de audio** del sonido de un arco al ser disparado.
- "**Grab Threshold**" es un valor que indica la **distancia máxima** a la cual puede estar la **mano derecha** del centro de la cuerda **para poder pellizcarla**.
- "**Start Point**" es una referencia al **punto central** de la **cuerda** donde la **tensión sería mínima**.
- "**End Point**" es una referencia al **punto central** de la **cuerda** donde la **tensión sería máxima**.
- "**Socket** " es una referencia al **centro de la cuerda**, del cual se **pellizca la cuerda**.
- "**Pull value**" es el valor de **tensión del arco** normalizado **entre 0 y 1**.
- "**UI**" es una referencia a la **interfaz de usuario** donde se mostrarán las **opciones hápticas**.
- "**Input GO**" es una referencia al objeto que **alberga el Script** "**OculusInput.cs**"
- "**Visual Vibration**" es una referencia a la **mano que haremos vibrar** al usar **pseudo-hápticos**.
- "**Notch**" es una referencia al **objeto en el cual se pondrá una flecha**.
- "**Hand Anchor**" es una referencia a la **mano opuesta** a la que **sujeta el arco**.



Ahora mostraré el propio código explicado.

```
using System.Collections;
using System.Net;
using System.IO;
using System.Threading;
using System.Net.Sockets;
using System.Text;
using UnityEngine;

6 referencias
public class Bow : MonoBehaviour
{
    [Header("Assets")]
    public GameObject m_ArrowPrefab = null;
    public AudioClip throwArrow;

    [Header("Bow")]
    public float m_GrabThreshold = 0.15f;
    public Transform startPoint = null;
    public Transform endPoint = null;
    public Transform socket = null;

    private Transform m_PullingHand = null;
    private Arrow m_PullingArrow = null;
    private Animator m_Animator = null;

    public float m_PullValue = 0.0f;
    private float former_PullValue = 0.0f;

    private AudioSource audioS;
    protected bool isVibrating = false;

    //UI related
    public GameObject ui;

    //The input
    public GameObject inputGO;
    public GameObject visualVibration;

    //Posible parents
    public GameObject notch;
    public GameObject handAnchor;

    0 referencias
    private void Awake()
    {
        audioS = GetComponent<AudioSource>();
        m_Animator = GetComponent<Animator>();
        ui = GameObject.Find("HapticOptions");
        inputGO = GameObject.Find("Input");
    }

    0 referencias
    private void Start()
    {

        StartCoroutine(CreateArrow(0.0f));

```

```
    0 referencias
    private void Update()
    {

        if (!m_PullingHand || !m_PullingArrow)
        {
            m_PullValue = 0;
            ui.GetComponent<UIControllerBow>().frictionValue = m_PullValue;
            return;
        }


        Debug.Log("dasda");

        m_PullValue = CalculatePull(m_PullingHand);
        m_PullValue = Mathf.Clamp(m_PullValue, 0.0f, 1.0f);

        m_PullValue = m_PullValue + visualVibration.GetComponent<VisualVibrationScript>().cuerda;

        //Ahora vamos a añadir un delay
        Debug.Log("CHANGE IN POSITION: " + (former_PullValue - m_PullValue));

        if ((m_PullValue > former_PullValue) && ui.GetComponent<UIControllerBow>().delay)
        {
            m_PullValue = former_PullValue + (m_PullValue - former_PullValue) / (1 + ui.GetComponent<UIControllerBow>().delayAmount*m_PullValue);
        }

        if (m_PullValue > 1.0f)
        {
            m_PullValue = 1.0f;
        }

        former_PullValue = m_PullValue;

        m_Animator.SetFloat("Blend", m_PullValue);

        ui.GetComponent<UIControllerBow>().frictionValue = m_PullValue;

        if(m_PullValue > 0.25f && ui.GetComponent<UIControllerBow>().tactile)
        {
            isVibrating = true;
            //OVRInput.SetControllerVibration(0.1f, m_PullValue, OVRInput.Controller.All);
            inputGO.GetComponent<OculusInput>().m_controller.SendHapticImpulse(0,m_PullValue,0.7f);
            StartCoroutine(setVibration(0.5f));
        }

        if (m_PullValue < 0.25f)
        {
            isVibrating = false;
            //inputGO.GetComponent<OculusInput>().m_controller.StopHaptics();
        }

    }
```

```
1 referencia
private float CalculatePull(Transform pullHand)
{
    Vector3 direction = endPoint.position - startPoint.position;
    float magnitude = direction.magnitude;
    direction.Normalize();

    Vector3 diference = pullHand.position - startPoint.position;

    return Vector3.Dot(diference, direction) / magnitude;
}

2 referencias
private IEnumerator CreateArrow(float waitTime)
{
    //Wait
    yield return new WaitForSeconds(waitTime);

    //Create object
    GameObject arrowObject = Instantiate(m_ArrowPrefab, socket);

    //Orient
    arrowObject.transform.localPosition = new Vector3(0,0,0.425f);
    arrowObject.transform.localEulerAngles = new Vector3(0, 0, -90);

    //Set
    m_PullingArrow = arrowObject.GetComponent<Arrow>();
}

1 referencia
public void Pull(Transform hand)
{
    float distance = Vector3.Distance(hand.position, startPoint.position);
    if(distance > m_GrabThreshold)
    {
        visualVibration.transform.parent = handAnchor.transform;
        visualVibration.transform.position = handAnchor.transform.position;
        visualVibration.transform.localEulerAngles = new Vector3(0, 0, -90);
        visualVibration.transform.localPosition = new Vector3(0, 0, 0);
        return;
    }
    else
    {
        visualVibration.transform.parent = notch.transform;
        visualVibration.transform.position = notch.transform.position;
        visualVibration.transform.localEulerAngles = new Vector3(0, 0, -90);
        visualVibration.transform.localPosition = new Vector3(0,0,0);
    }

    m_PullingHand = hand;
}

1 referencia
public void Release()
{
    visualVibration.transform.parent = handAnchor.transform;
    visualVibration.transform.position = handAnchor.transform.position;
```

```
    visualVibration.transform.localEulerAngles = new Vector3(0, 0, -90);
    visualVibration.transform.localPosition = new Vector3(0, 0, 0);

    if (m_PullValue > 0.25f)
    {
        audioS.PlayOneShot(throwArrow);
        FireArrow();
    }
    m_PullingHand = null;
    m_PullValue = 0.0f;
    m_Animator.SetFloat("Blend", 0.0f);

    if (!m_PullingArrow)
    {
        StartCoroutine(CreateArrow(0.25f));
    }
}

1 referencia
private void FireArrow()
{
    m_PullingArrow.Fire(m_PullValue);
    m_PullingArrow = null;
}

1 referencia
private IEnumerator setVibration(float waitTime)
{
    yield return new WaitForSeconds(waitTime);
    isVibrating = false;
}
}
```

En primer lugar en la función **Awake()**, la cual a efectos prácticos es **muy similar a Start()**, **actualizamos las variables pertinentes** buscando los componentes u objetos que van a albergar. Mientras tanto, en la función **Start()**

creamos un **co-rutina** (Una aproximación a la programación **multihilo** en Unity3D) en la que **añadimos una flecha al arco** con la función **CreateArrow(float)**. Esta función crea una instancia del modelo "**Arrow Prefab**".

```
0 referencias
private void Awake()
{
    audioS = GetComponent<AudioSource>();
    m_Animator = GetComponent<Animator>();
    ui = GameObject.Find("HapticOptions");
    inputGO = GameObject.Find("Input");
}

0 referencias
private void Start()
{

    StartCoroutine(CreateArrow(0.0f));

}
```

```
2 referencias
private IEnumerator CreateArrow(float waitTime)
{
    //Wait
    yield return new WaitForSeconds(waitTime);

    //Create object
    GameObject arrowObject = Instantiate(m_ArrowPrefab, socket);

    //Orient
    arrowObject.transform.localPosition = new Vector3(0,0,0.425f);
    arrowObject.transform.localEulerAngles = new Vector3(0, 0, -90);

    //Set
    m_PullingArrow = arrowObject.GetComponent<Arrow>();
}
```

A partir de ahora operaremos en la función **Update()** empezando por **poner a 0 el valor de tensión del arco si no está siendo pellizcada la cuerda o si no tenemos cuerda**.

```
if (!m_PullingHand || !m_PullingArrow)
{
    m_PullValue = 0;
    ui.GetComponent<UIControllerBow>().frictionValue = m_PullValue;
    return;
}
```

Ahora **calculamos la tensión del arco** con la función **CalculatePull(object)**, usando para ello la **transformada** y hallando así el **módulo del vector** que une la mano con el punto de menor tensión.

```
1 referencia
private float CalculatePull(Transform pullHand)
{
    Vector3 direction = endPoint.position - startPoint.position;
    float magnitude = direction.magnitude;
    direction.Normalize();

    Vector3 diference = pullHand.position - startPoint.position;

    return Vector3.Dot(diference, direction) / magnitude;
}
```

Después realizamos acciones relacionadas con los **pseudo-hápticos** que explicaremos más adelante en una sección dedicada a estos. Después **modificamos** la **animación del arco en base a la tensión**.

```
former_PullValue = m_PullValue;

m_Animator.SetFloat("Blend", m_PullValue);
```

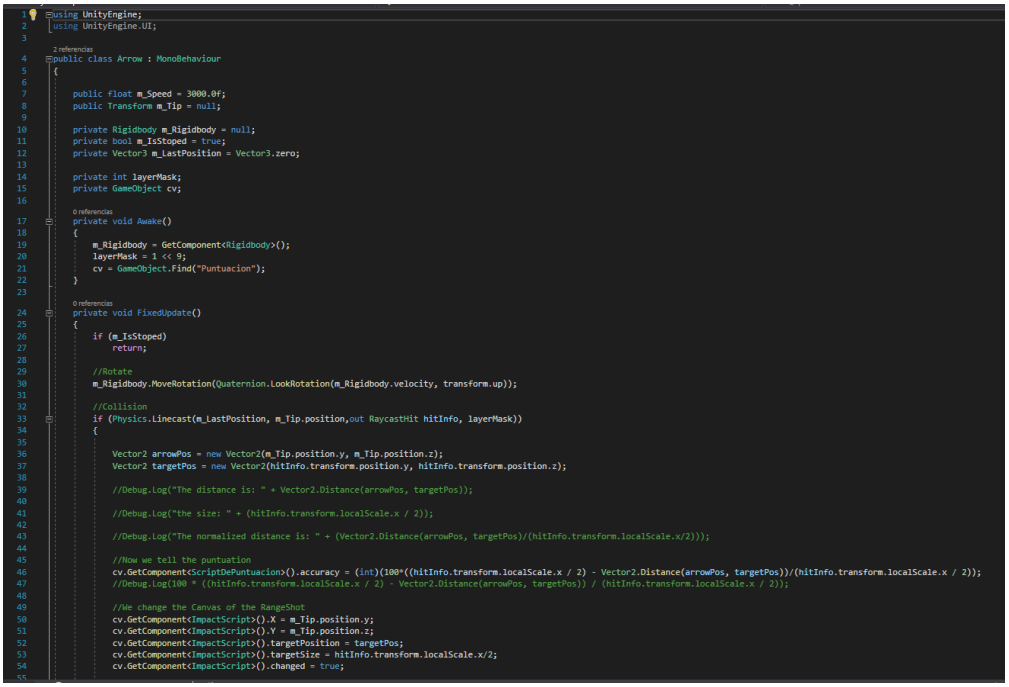
Las funciones **Pull()** y **Release()** a las que se hace referencia en el script "**OculusInput.cs**" se encuentran en este script. La función **Pull() indica si la cuerda está siendo pellizcada** en base al "Grab_Threshold", cambiando en consecuencia la mano tensora. Por otro lado la función **Release() suelta la cuerda**, y dependiendo de **si la tensión era mayor que 0.25** o menor **disparará una flecha** o sencillamente **dejará el arco en reposo** y actualizará su animación. **Si la flecha es disparada** activará el **sonido de disparo del arco** y, al no haber flecha, **ordena crear otra** en un cuarto de segundo a la función **CreateArrow()**.

```
1 referencia
public void Pull(Transform hand)
{
    float distance = Vector3.Distance(hand.position, startPoint.position);
    if(distance > m_GrabThreshold)
    {
        visualVibration.transform.parent = handAnchor.transform;
        visualVibration.transform.position = handAnchor.transform.position;
        visualVibration.transform.localEulerAngles = new Vector3(0, 0, -90);
        visualVibration.transform.localPosition = new Vector3(0, 0, 0);
        return;
    }
    else
    {
        visualVibration.transform.parent = notch.transform;
        visualVibration.transform.position = notch.transform.position;
        visualVibration.transform.localEulerAngles = new Vector3(0, 0, -90);
        visualVibration.transform.localPosition = new Vector3(0,0,0);
    }

    m_PullingHand = hand;
}

1 referencia
public void Release()
{
    visualVibration.transform.parent = handAnchor.transform;
    visualVibration.transform.position = handAnchor.transform.position;
    visualVibration.transform.localEulerAngles = new Vector3(0, 0, -90);
    visualVibration.transform.localPosition = new Vector3(0, 0, 0);

    if (m_PullValue > 0.25f)
    {
        audioS.PlayOneShot(throwArrow);
        FireArrow();
    }
    m_PullingHand = null;
    m_PullValue = 0.0f;
    m_Animator.SetFloat("Blend", 0.0f);

    if (!m_PullingArrow)
    {
        StartCoroutine(CreateArrow(0.25f));
    }
}
```

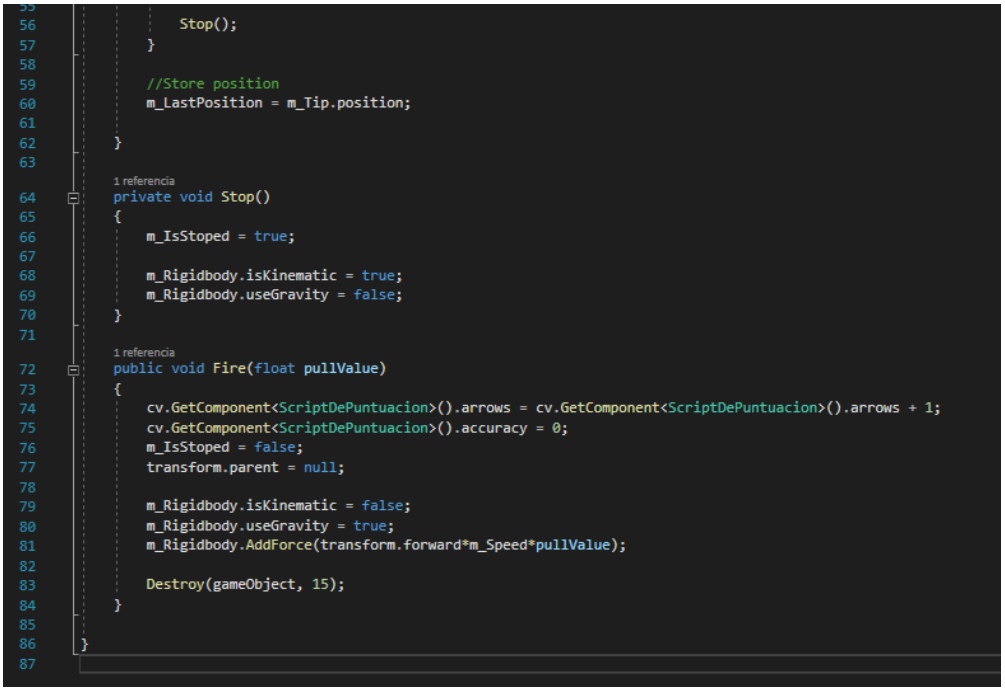
La función **FireArrow()** se encarga de **anular la referencia a la flecha del arco** y de **llamar a la función de disparo** del script de la flecha.

```
1 referencia
private void FireArrow()
{
    m_PullingArrow.Fire(m_PullValue);
    m_PullingArrow = null;
}
```

Por otro lado tenemos un script para la flecha.

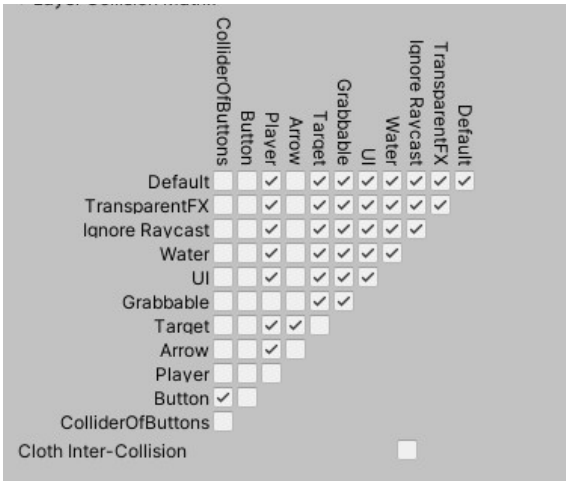


```
using UnityEngine;
using UnityEngine.UI;

2 referencias
public class Arrow : MonoBehaviour
{

    public float m_Speed = 3000.0f;
    public Transform m_Tip = null;

    private Rigidbody m_Rigidbody = null;
    private bool m_IsStoped = true;
    private Vector3 m_LastPosition = Vector3.zero;

    private int layerMask;
    private GameObject cv;

    0 referencias
    private void Awake()
    {
        m_Rigidbody = GetComponent<Rigidbody>();
        layerMask = 1 << 9;
        cv = GameObject.Find("Puntuacion");
    }

    0 referencias
    private void FixedUpdate()
    {
        if (m_IsStoped)
            return;

        //Rotate
        m_Rigidbody.MoveRotation(Quaternion.LookRotation(m_Rigidbody.velocity, transform.up));

        //Collision
        if (Physics.Linecast(m_LastPosition, m_Tip.position,out RaycastHit hitInfo, layerMask))
        {

            Vector2 arrowPos = new Vector2(m_Tip.position.y, m_Tip.position.z);
            Vector2 targetPos = new Vector2(hitInfo.transform.position.y, hitInfo.transform.position.z);

            //Debug.Log("The distance is: " + Vector2.Distance(arrowPos, targetPos));

            //Debug.Log("the size: " + (hitInfo.transform.localScale.x / 2));

            //Debug.Log("The normalized distance is: " + (Vector2.Distance(arrowPos, targetPos)/(hitInfo.transform.localScale.x/2)));

            //Now we tell the puntuation
            cv.GetComponent<ScriptDePuntuacion>().accuracy = (int)(100*((hitInfo.transform.localScale.x / 2) - Vector2.Distance(arrowPos, targetPos))/(hitInfo.transform.localScale.x / 2));
            //Debug.Log(100 * ((hitInfo.transform.localScale.x / 2) - Vector2.Distance(arrowPos, targetPos)) / (hitInfo.transform.localScale.x / 2));

            //We change the Canvas of the RangeShot
            cv.GetComponent<ImpactScript>().X = m_Tip.position.y;
            cv.GetComponent<ImpactScript>().Y = m_Tip.position.z;
            cv.GetComponent<ImpactScript>().targetPosition = targetPos;
            cv.GetComponent<ImpactScript>().targetSize = hitInfo.transform.localScale.x/2;
            cv.GetComponent<ImpactScript>().changed = true;
```



```
            Stop();
        }

        //Store position
        m_LastPosition = m_Tip.position;

    }

    1 referencia
    private void Stop()
    {
        m_IsStoped = true;

        m_Rigidbody.isKinematic = true;
        m_Rigidbody.useGravity = false;
    }

    1 referencia
    public void Fire(float pullValue)
    {
        cv.GetComponent<ScriptDePuntuacion>().arrows = cv.GetComponent<ScriptDePuntuacion>().arrows + 1;
        cv.GetComponent<ScriptDePuntuacion>().accuracy = 0;
        m_IsStoped = false;
        transform.parent = null;

        m_Rigidbody.isKinematic = false;
        m_Rigidbody.useGravity = true;
        m_Rigidbody.AddForce(transform.forward*m_Speed*pullValue);

        Destroy(gameObject, 15);
    }

}
```

En primer lugar hacemos referencia a su "**Rigidbody**", un **componente encargado de controlar las físicas de un objeto**, además, cambiaremos la **máscara** en la cual se halla la flecha para así **controlar con qué clase de objetos puede colisionar**.



*La **máscara de colisiones** es una **matriz triangular** en la cual podemos **indicar que capas pueden colisionar entre sí**, y, por tanto, los objetos pertenecientes a esa capa. Puede modificarla en **Edit -> Project Settings -> Physics***

```
0 referencias
private void Awake()
{
    m_Rigidbody = GetComponent<Rigidbody>();
    layerMask = 1 << 9;
    cv = GameObject.Find("Puntuacion");
}

0 referencias
```

Ahora nos adentraremos en la función **FixedUpdate()**, la cual es muy similar a **Update()** salvo que es independiente de la frecuencia del ordenador usando el "**DeltaTime**". En ella **acabaremos la función** si la **flecha ha sido detenida** y, en **caso contrario cambiaremos la rotación global** para que siga apuntando siempre hacia la punta de la flecha.

```
if (m_IsStoped)
    return;

//Rotate
m_Rigidbody.MoveRotation(Quaternion.LookRotation(m_Rigidbody.velocity, transform.up));
```

Ahora veremos si **enfrente de la flecha hay algún objetivo**, en cuyo caso **detendremos la flecha** y **obtendremos** toda la información de la **precisión**.

```
//Collision
if (Physics.Linecast(m_LastPosition, m_Tip.position,out RaycastHit hitInfo, layerMask))
{

    Vector2 arrowPos = new Vector2(m_Tip.position.y, m_Tip.position.z);
    Vector2 targetPos = new Vector2(hitInfo.transform.position.y, hitInfo.transform.position.z);

    //Debug.Log("The distance is: " + Vector2.Distance(arrowPos, targetPos));

    //Debug.Log("the size: " + (hitInfo.transform.localScale.x / 2));

    //Debug.Log("The normalized distance is: " + (Vector2.Distance(arrowPos, targetPos)/(hitInfo.transform.localScale.x/2)));

    //Now we tell the puntuation
    cv.GetComponent<ScriptDePuntuacion>().accuracy = (int)(100*((hitInfo.transform.localScale.x / 2) - Vector2.Distance(arrowPos, targetPos))/(hitInfo.transform.localScale.x / 2));
    //Debug.Log(100 * ((hitInfo.transform.localScale.x / 2) - Vector2.Distance(arrowPos, targetPos)) / (hitInfo.transform.localScale.x / 2));

    //We change the Canvas of the RangeShot
    cv.GetComponent<ImpactScript>().X = m_Tip.position.y;
    cv.GetComponent<ImpactScript>().Y = m_Tip.position.z;
    cv.GetComponent<ImpactScript>().targetPosition = targetPos;
    cv.GetComponent<ImpactScript>().targetSize = hitInfo.transform.localScale.x/2;
    cv.GetComponent<ImpactScript>().changed = true;

    Stop();
}

//Store position
```

El script contiene también dos funciones, **Stop()** para detener la flecha y **Fire()** para dispararla. La función **Stop()**, además de detener la flecha activa el booleano **isKinematic**, el cual hace que **deje de poseer características físicas**, y desactiva el booleano **useGravity** para **dejar de ser afectada por la gravedad**, la cual puede ser modificada también desde **Edit -> Project Settings -> Physics**. A la hora de disparar la flecha con la función **Fire()** tiene en cuenta no solo la **tensión del arco**, sino también una variable inherente a la flecha llamada "**m_Speed**". En ese caso se realiza la operación inversa, permitiendo que la **flecha sea afectada por la gravedad** y **dejando de ser cinemática**. Al cabo de 1**5 segundos la flecha será destruida** con la función **Destroy(objeto, tiempo)**.

```
    1 referencia
    private void Stop()
    {
        m_IsStoped = true;

        m_Rigidbody.isKinematic = true;
        m_Rigidbody.useGravity = false;
    }

    1 referencia
    public void Fire(float pullValue)
    {
        cv.GetComponent<ScriptDePuntuacion>().arrows = cv.GetComponent<ScriptDePuntuacion>().arrows + 1;
        cv.GetComponent<ScriptDePuntuacion>().accuracy = 0;
        m_IsStoped = false;
        transform.parent = null;

        m_Rigidbody.isKinematic = false;
        m_Rigidbody.useGravity = true;
        m_Rigidbody.AddForce(transform.forward*m_Speed*pullValue);

        Destroy(gameObject, 15);
    }

}
```

Con el **entorno**, el **jugador** y el **arco** y las **flechas** ya tendríamos una **versión primitiva de la aplicación en el "modo clásico"**, pero aún quedarían muchas funcionalidades para mejorar el juego.

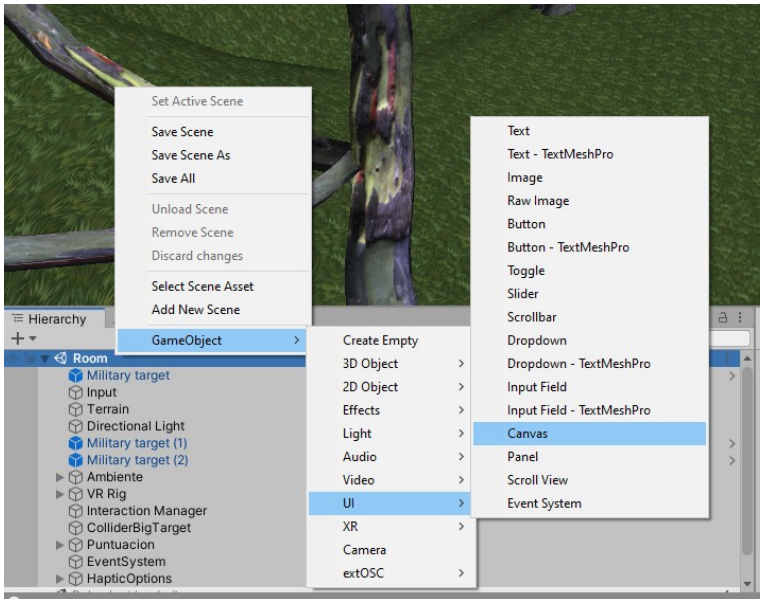
Parte 2.5: Interfaces de usuario

En el **estadío actual** del juego, **un jugador podría tomar el arco y disparar flechas**, pero **no recibiría ningún tipo de información del entorno**, tan solo vería las flechas desaparecer al cabo de unos segundos. Primero debemos pensar **qué información podría interesar al jugador** y, más tarde, **como podría representarse** de una forma sencilla esa información. En un primer momento pensé en **dos variables**, el **número de flechas** disparadas y la **precisión**. El número de flechas se representa usando un **número entero** mientras que la precisión se muestra como un v**alor porcentual donde**

el 0 equivaldría a haber errado la diana y el 100 haber alcanzado el centro, los valores intermedios servirían para mostrar la proximidad al centro, siendo 1 muy cercano al borde exterior y 99 muy cerca del centro.

La explicación de cómo se ha configurado la diana para obtener la precisión se explicará en otro subapartado, en este momento vamos a tomar como que ya lo hemos programado.

Primero vamos a ver el modo antiguo para mostrar una interfaz gráfica en Unity3D. Para ello vamos a crear un "**canvas**", un **lienzo donde colocaremos los elementos** que queramos mostrar en la interfaz.



Una vez tengamos el canvas creado **podremos crear dentro de él diversos elementos clásicos de interfaces gráficas** (etiquetas de texto, sliders, botones, imágenes...) y las distribuiremos como más nos guste, en este caso he puesto tan solo **una imagen** en blanco, que nos servirá de fondo, y **dos etiquetas de texto** que nos indicarán la **precisión** y el **número de flechas**.

Después añadiremos un script al objeto para que actualice los valores y obtendremos el siguiente resultado.

*La etiqueta sigue la **rotación** de nuestra cabeza*

El hecho de que la etiqueta siga nuestra cabeza puede parecer a priori muy bueno, ya que de esa forma el usuario podría visualizar la información en todo momento, sin embargo puede resultar **molesto** ocupar un fragmento tan grande de la pantalla del jugador, y en realidad virtual es un problema aún mayor, ya que esta forma de mostrar interfaces gráficas puede llegar a causar **malestar** y **mareos** al jugador a largo plazo.

Por suerte existe una **solución** para ello que aparece entre las opciones de un "**canvas**" de Unity3D, esa opción consiste en **trasladar una interfaz gráfica anclada a una cámara a un objeto del mundo**, como si se tratará de un lienzo dentro del mundo virtual, para ello solo es necesario "**Render Mode**" del "**Canvas**" a "**World Space**". De este modo podremos trabajar con él como si de un objeto convencional se tratara, usando los gizmos para cambiar su transformada.

Esta interfaz se encarga de ***mostrar los puntos donde se ha acertado el tiro****, además de* ***mostrar el número de flechas disparadas*** *y la* ***precisión del último disparo****. En caso de querer* ***borrar todas las marcas*** *se puede presionar el* ***botón de reset****. .*

*En esta interfaz se muestran las **opciones relacionadas al modo pseudo-háptico** y las **escenas siguiente** junto a los **botones** que presionar para acceder a ellas.*

*Ahora **las interfaces forman parte del mundo virtual** de tal forma que el jugador podría rotar para usarlas.*

Ahora explicaré los scripts utilizados en cada una de las interfaces para controlar los valores que muestran cada uno de ellos.

La interfaz de la diana está contenida dentro del objeto "**Puntuacion**", el cual contiene **tres scripts creados por mi** y **uno por defecto** de la librería de XR. El primer script es "**ImpactScript.cs**", el cual se encarga de **añadir las equis al**

impactar una flecha. Este Script posee una serie de variables públicas que pueden ser modificadas por el script del objeto flecha.

```
//Now we tell the puntuation
cv.GetComponent<ScriptDePuntuacion>().accuracy = (int)(100*((hitInfo.transform.localScale.x / 2) - Vector2.Distance(arrowPos, targetPos))/(hitInfo.transform.localScale.x / 2));
//Debug.Log(100 * ((hitInfo.transform.localScale.x / 2) - Vector2.Distance(arrowPos, targetPos)) / (hitInfo.transform.localScale.x / 2));

//We change the Canvas of the RangeShot
cv.GetComponent<ImpactScript>().X = m_Tip.position.y;
cv.GetComponent<ImpactScript>().Y = m_Tip.position.z;
cv.GetComponent<ImpactScript>().targetPosition = targetPos;
cv.GetComponent<ImpactScript>().targetSize = hitInfo.transform.localScale.x/2;
cv.GetComponent<ImpactScript>().changed = true;

Stop();
```

*En esta parte del código de "**Arrow.cs**" podemos ver **cómo modifica la coordenada X del impacto**, la **coordenada Y**, la **posición del objeto diana**, el **diámetro de la diana impactada** y una **variable para indicar que ha habido un impacto** del script "ImpactScript.cs"*

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

5 referencias
public class ImpactScript : MonoBehaviour
{
    [Header("Objects")]
    public GameObject RangeShot;
    public GameObject equis;

    [Header("Limits")]
    public float lowerLimit;
    public float higherLimit;
    public float rightLimit;
    public float leftLimit;

    [Header("Last Hit")]
    public bool changed;
    public float X;            //Corresponds to Right and left limits
    public float Y;            //Corresponds to Lower and Higher limits
    public Vector2 targetPosition;
    public float targetSize;

    0 referencias
    void Start()
    {
        changed = false;
    }

    // Update is called once per frame
    0 referencias
    void Update()
    {

        if (changed)
        {
            GameObject newX = Instantiate(equis, RangeShot.transform);

            //We calculate the difference between the arrow and the center of the target and normalize it
            float xAux = (X - targetPosition.x) /targetSize;
            float yAux = (targetPosition.y - Y)/targetSize;

            newX.transform.localPosition = new Vector3(yAux*higherLimit, xAux*rightLimit, 0);

            changed = false;
        }


    }
}
```

En este código lo que se hace es **instanciar una nueva equis** y **ponerla en la posición de la interfaz diana** proporcional a la posición real del impacto solo en el caso de haber recibido un impacto en esa iteración.

El segundo script es “**ResetGUI.cs**”, que se encarga de recorrer la lista de puntos en la diana y destruirlos.

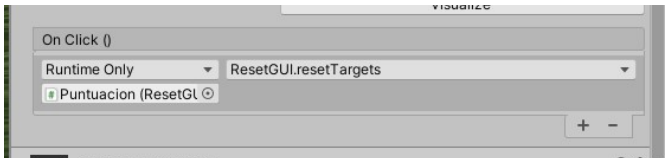
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

0 referencias
public class ResetGUI : MonoBehaviour
{

    public GameObject target;

    0 referencias
    public void resetTargets()
    {

        for (int i = target.transform.childCount - 1; i >= 0; i--)
        {
            GameObject.Destroy(target.transform.GetChild(i).gameObject);
        }
    }
}

```

Además de estos dos scripts existe uno que **se encarga sencillamente de actualizar los valores del número de flechas disparadas y la precisión del último disparo**. Siendo estos 2 valores actualizados por el script “Arrow.cs”.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

4 referencias
public class ScriptDePuntuacion : MonoBehaviour
{
    public int arrows = 0;
    public int accuracy = 0;

    public Text arrowsText;
    public Text accText;

    // Update is called once per frame
    0 referencias
    void Update()
    {
        arrowsText.text = "Arrows: " + arrows;
        accText.text = "Accuracy: " + accuracy + " / 100";
    }
}

```

Además existe dentro del objeto un script ofrecido por la librería de XR llamado “**TrackedDeviceGraphicRaycaster.cs**”, el cual explicaremos en mayor detalle al final de este apartado.

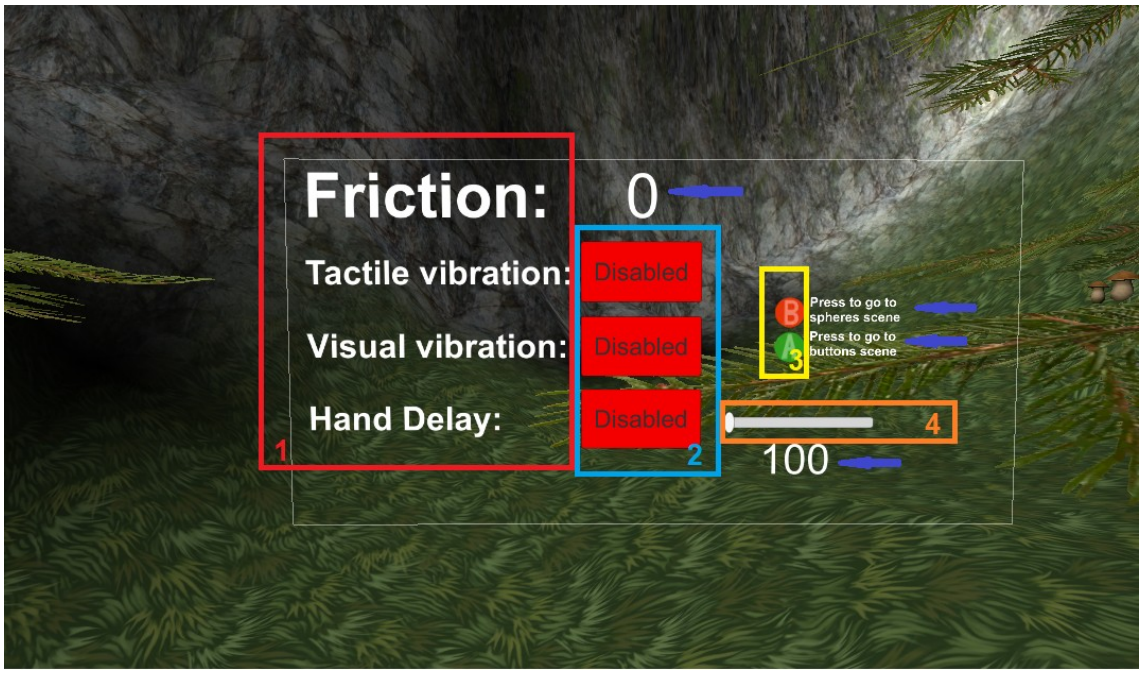
Ahora tenemos esta interfaz completa, pero el botón de **Reset** ahora mismo estaría **inoperativo**, así que le vamos a **indicar que al pulsarlo active la función de reseteo** de la diana que se encuentra dentro del script “ResetGUI.cs”.



*De este modo estamos indicando que la **acción de hacer clic** en el botón **activa la función resetTargets()** contenida en el script "ResetGUI.cs" incluido en el objeto puntuación.*

Ahora el botón realiza una acción al pulsarlo, pero de momento **tampoco funcionará debido a que no nos encontramos en un entorno convencional con un ratón**, sino en uno VR donde las interfaces se activan con las manos. Después de explicar brevemente la interfaz de control de los pseudo-hápticos explicaré como hacerlo.

La **interfaz de control de los pseudo-hápticos** consiste en **tres botones**, **cuatro textos fijos**, **cuatro textos que cambian** gracias a un script, **dos imágenes** y **un slider**.

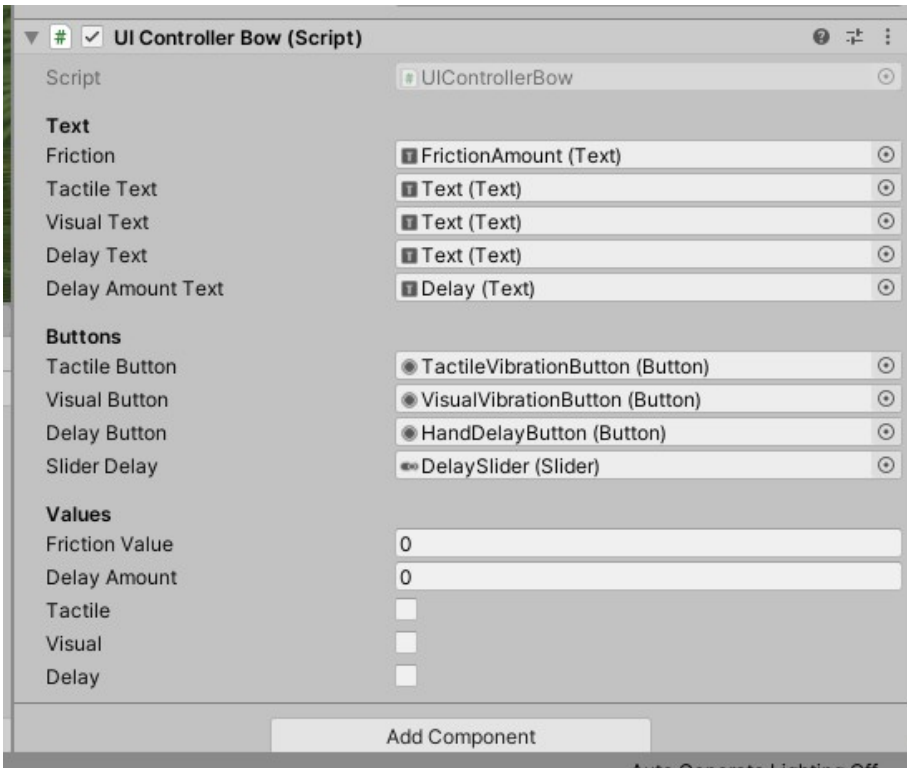


1. Textos fijos**, **2. Botones**, **3. Imágenes**, **4. Slider**, **flechas azules. Textos Variables

Esta **interfaz será reutilizada** en las otras dos aplicaciones debido a su escalabilidad y a la consistencia de los elementos mostrados. Esta interfaz **se halla contenida dentro del objeto HapticOptions**, y posee **un único script** propio llamado "**UIControllerBow.cs**" con las siguientes variables públicas:

- "**Friction**" es el texto correspondiente a la **tensión actual del arco**.
- "**Tactile Text**", "**Visual Text**" y "**Delay Text**" son los textos contenidos dentro de los 3 botones del juego, su función es **cambiar de "Enabled" a "Disabled"** al ser pulsados o viceversa.
- "**Delay Amount Text**" indica el **valor del slider** entre 0 y 100.

- "**Tactile Button**", "**Visual Button**" y "**Delay Button**" son botones que se encargan de **activar las funciones pseudo-hápticas** indicadas en su nombre.
- "**Slider Delay**" contiene el **slider encargado de modificar el "Hand delay"**.
- "**Friction value**" contiene el **valor numérico de la tensión del arco**.
- "**Delay Amount**" contiene el **valor numérico del slider**.
- "**Tactile**", "**Visual**" y "**Delay**" son variables booleanas encargadas de **indicar si la opción pseudo-háptica correspondiente está activada**.



Ahora pasaré a explicar de forma detallada el código.

```
using System.Collections;
using System.Collections.Generic;
using System.Net;
using System.Net.Sockets;
using System.Text;
using UnityEngine;
using UnityEngine.UI;
using System.Threading;
using System;

41 referencias
public class UIControllerBow : MonoBehaviour
{
    [Header("Text")]
    public Text friction;
    public Text tactileText;
    public Text visualText;
    public Text delayText;
    public Text delayAmountText;

    [Header("Buttons")]
    public Button tactileButton;
    public Button visualButton;
    public Button delayButton;
    public Slider sliderDelay;

    [Header("Values")]
    public float frictionValue = 0;
    public int delayAmount = 0;
    public bool tactile = false;
    public bool visual = false;
    public bool delay = false;

    private bool canSendPackage = true;

    Thread m_Thread;
    UdpClient m_Client;

    0 referencias
    void Start()
    {
        m_Thread = new Thread(new ThreadStart(ReceiveData));
        m_Thread.IsBackground = true;
        m_Thread.Start();
    }

    // Update is called once per frame
    0 referencias
    void Update()
    {
        delayAmount = (int)sliderDelay.value;
        //We change the friction
        friction.text = "" + frictionValue;
        delayAmountText.text = "" + delayAmount;

        /*if (canSendPackage)
        {
            StartCoroutine(sendData(0.15f));
        }*/

        sendData();
    }

    1 referencia
    private void sendData()
    {
        //canSendPackage = false;
        //yield return new WaitForSeconds(waitTime);

        IPEndPoint ep = new IPEndPoint(IPAddress.Parse("192.168.4.1"), 20001);
        Socket sock = new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp);

        try
        {
            sock.SendTo(Encoding.ASCII.GetBytes("" + frictionValue), ep);
            Debug.Log("Information sent");
        }
        catch
        {
            Debug.Log("Information not sent");
        }

        canSendPackage = true;
    }

    1 referencia
    void ReceiveData()
    {
        while (true)
        {
            try
            {

                m_Client = new UdpClient(20000);
                m_Client.EnableBroadcast = true;
                while (true)
                {

                    IPEndPoint hostIP = new IPEndPoint(IPAddress.Any, 0);
                    byte[] data = m_Client.Receive(ref hostIP);
                    string returnData = Encoding.ASCII.GetString(data);

                    Debug.Log(returnData);
                }
            }
            catch (Exception e)
            {
                Debug.Log(e);

                OnApplicationQuit();
            }
        }

    }
```

```
109     }
110
        1 referencia
111     private void OnApplicationQuit()
112     {
113         if (m_Thread != null)
114         {
115             m_Thread.Abort();
116         }
117
118         if (m_Client != null)
119         {
120             m_Client.Close();
121         }
122     }
123
        0 referencias
124     public void tactileClick()
125     {
126         if (tactile)
127         {
128             tactile = false;
129             tactileText.text = "Disabled";
130             var colors = tactileButton.colors;
131             colors.normalColor = Color.red;
132             tactileButton.colors = colors;
133         }
134         else
135         {
136             tactile = true;
137             tactileText.text = "Enabled";
138             var colors = tactileButton.colors;
139             colors.normalColor = Color.green;
140             tactileButton.colors = colors;
141         }
142     }
143
        0 referencias
144     public void visualClick()
145     {
146         if (visual)
147         {
148             visual = false;
149             visualText.text = "Disabled";
150             var colors = visualButton.colors;
151             colors.normalColor = Color.red;
152             visualButton.colors = colors;
153         }
154         else
155         {
156             visual = true;
157             visualText.text = "Enabled";
158             var colors = visualButton.colors;
159             colors.normalColor = Color.green;
160             visualButton.colors = colors;
161         }
162     }
163
```

```
163
        0 referencias
164     public void delayClick()
165     {
166         if (delay)
167         {
168             delay = false;
169             delayText.text = "Disabled";
170             var colors = delayButton.colors;
171             colors.normalColor = Color.red;
172             delayButton.colors = colors;
173         }
174         else
175         {
176             delay = true;
177             delayText.text = "Enabled";
178             var colors = delayButton.colors;
179             colors.normalColor = Color.green;
180             delayButton.colors = colors;
181         }
182     }
183
184 }
185
```

En un primer lugar **declaramos las variables que vamos a utilizar**. Se puede ver que en las líneas **14**, **21** y **27** hay unas líneas de código del tipo **[Header("String")]**. Estos comandos no influyen en la lógica del script, pero **ayudan a que los desarrolladores** puedan ver con más claridad las variables públicas en el editor.

```
14      [Header("Text")]
15      public Text friction;
16      public Text tactileText;
17      public Text visualText;
18      public Text delayText;
19      public Text delayAmountText;
20
21      [Header("Buttons")]
22      public Button tactileButton;
23      public Button visualButton;
24      public Button delayButton;
25      public Slider sliderDelay;
26
27      [Header("Values")]
28      public float frictionValue = 0;
29      public int delayAmount = 0;
30      public bool tactile = false;
31      public bool visual = false;
32      public bool delay = false;
33
34      private bool canSendPackage = true;
35
```

En la función **Start()** se hallan funciones relacionadas con la **comunicación con el arduino**, por lo que serán explicadas más tarde junto con las otras funciones de conectividad del script **sendData()**, **receiveData()** y **onAplicationQuit()**. En la función de **Update() actualizamos** continuamente los **valores de la tensión del arco** y el **valor numérico del slider**.

```
// Update is called once per frame
0 referencias
void Update()
{
    delayAmount = (int)sliderDelay.value;
    //We change the friction
    friction.text = "" + frictionValue;
    delayAmountText.text = "" + delayAmount;

    /*if (canSendPackage)
    {
        StartCoroutine(sendData(0.15f));
    }*/

    sendData();
}
```

Además tenemos **tres funciones relacionadas cada una con uno de los botones** de la interfaz. Estas funciones:

-Primero ven si el modo al que hacen referencia **está activado o no**:

-Si **está** activado:

-Cambiamos la variable booleana del modo a **"false"**

-Cambiamos el texto del botón a **"Disabled"**

-Cambiamos el color del botón a **rojo**.

-Si **no está** activado:

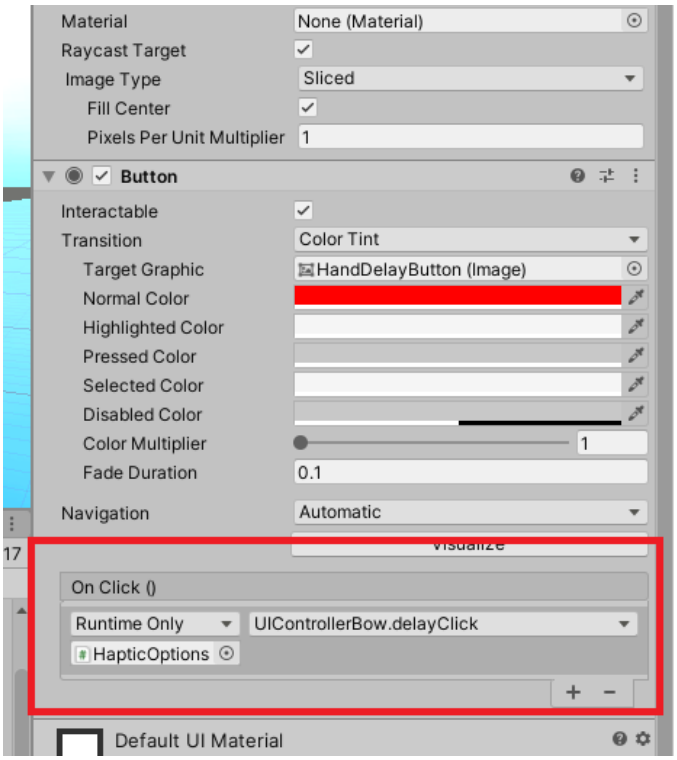
-Cambiamos la variable booleana del modo a **"true"**

-Cambiamos el texto del botón a **"Enabled"**

-Cambiamos el color del botón a **verde**.

```
0 referencias
public void tactileClick()
{
    if (tactile)
    {
        tactile = false;
        tactileText.text = "Disabled";
        var colors = tactileButton.colors;
        colors.normalColor = Color.red;
        tactileButton.colors = colors;
    }
    else
    {
        tactile = true;
        tactileText.text = "Enabled";
        var colors = tactileButton.colors;
        colors.normalColor = Color.green;
        tactileButton.colors = colors;
    }
}

0 referencias
public void visualClick()
{
    if (visual)
    {
        visual = false;
        visualText.text = "Disabled";
        var colors = visualButton.colors;
        colors.normalColor = Color.red;
        visualButton.colors = colors;
    }
    else
    {
        visual = true;
        visualText.text = "Enabled";
        var colors = visualButton.colors;
        colors.normalColor = Color.green;
        visualButton.colors = colors;
    }
}

0 referencias
public void delayClick()
{
    if (delay)
    {
        delay = false;
        delayText.text = "Disabled";
        var colors = delayButton.colors;
        colors.normalColor = Color.red;
        delayButton.colors = colors;
    }
    else
    {
        delay = true;
        delayText.text = "Enabled";
        var colors = delayButton.colors;
        colors.normalColor = Color.green;
        delayButton.colors = colors;
    }
}
```

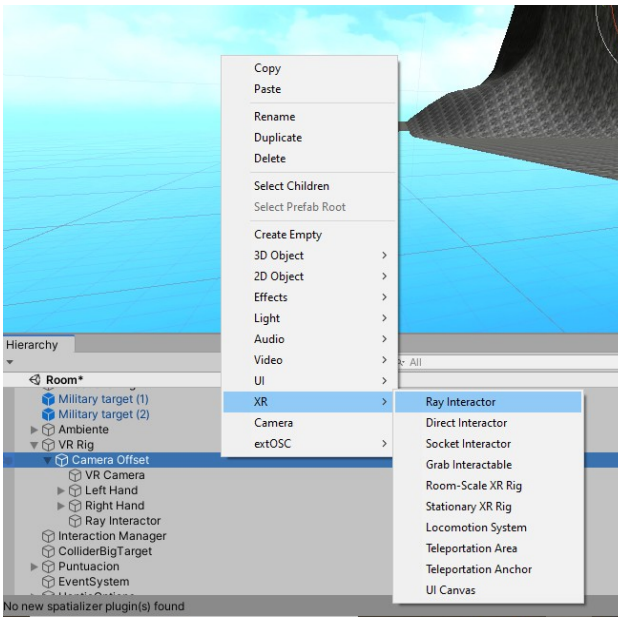
Cada una de estas funciones se relaciona a un botón del mismo modo que el botón de “Reset” de la interfaz de la diana, mediante el evento de los botones **OnClick()**.

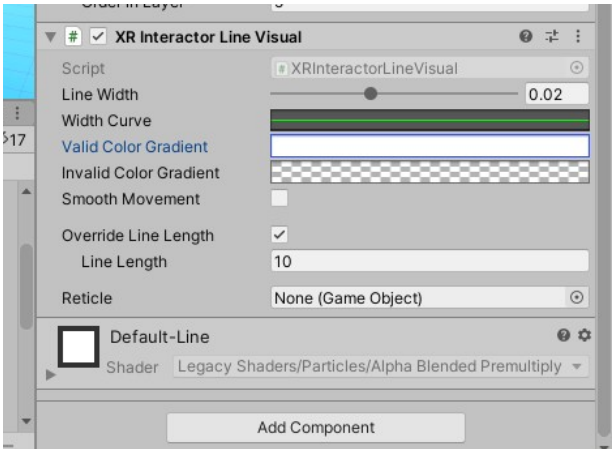


Con todo esto **tendríamos todas las interfaces creadas**, pero **ninguna forma de interactuar con ellas**. Lo único que nos falta es adaptar las interfaces a realidad virtual. Una forma muy cómoda e intuitiva es **usar los propios controladores del dispositivo**, para lo cual las librerías de XR nos proporcionan las herramientas necesarias.

*La **interacción** se realizará **utilizando un rayo** que llegue de la mano seleccionada hasta la interfaz, actuando como el ratón de un ordenador.*

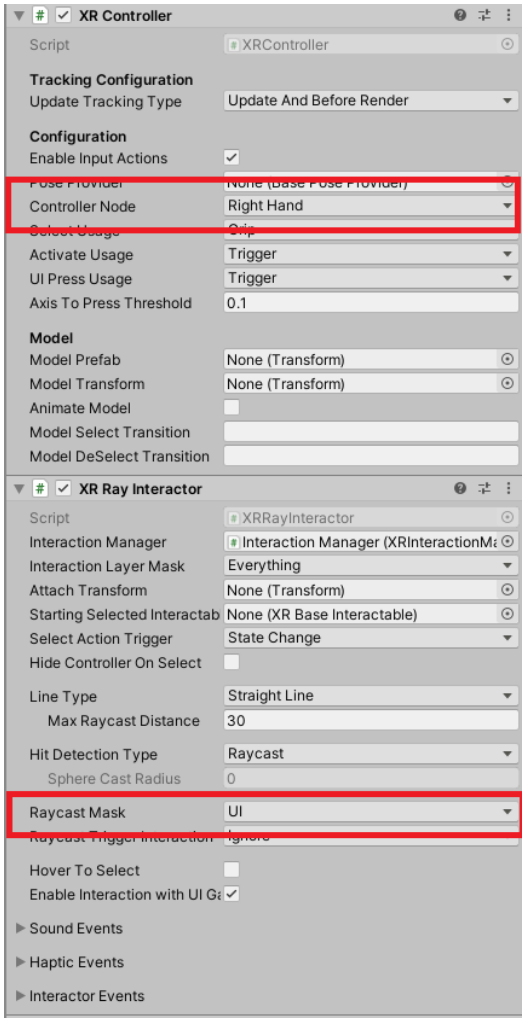
Para ello debemos **crear un objeto Ray Interactor** dentro del objeto “Camera Offset” del jugador. En el objeto “Ray Interactor” podremos **cambiar el color** del rayo utilizando el componente “**XR Interactor Line Visual**”, en mi caso dejaré el color por defecto (blanco).





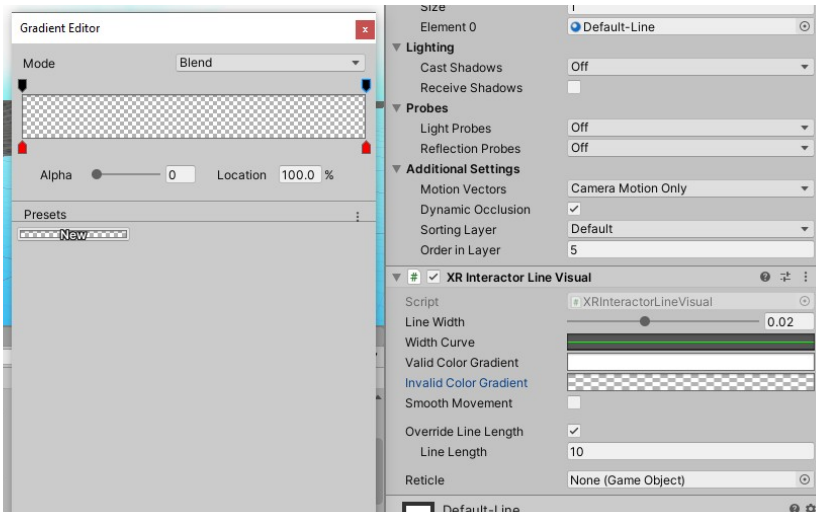
*Podrá elegirse un **degradado de colores** que parta del inicio hasta el final del rayo.*

Ahora debemos **elegir la mano que controla el rayo** y el **tipo de objetos con los que se activa**. Para ello en "**Controller Node**" seleccionaremos el **controlador que controla el rayo** y en "**Raycast Mask**" la **máscara** en la que se hallan las **interfaces** (previamente hay que **cambiar la capa** en la que estas se hallan).



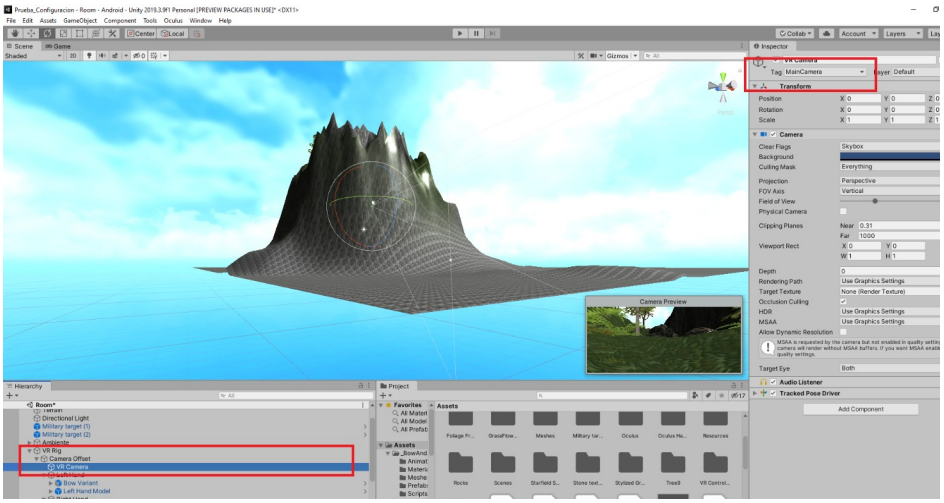
Ahora **tendríamos un rayo que se activa al entrar en contacto con los objetos dentro de la capa "UI"**. Si quisiéramos tener otro rayo para la otra mano tan solo tendríamos que duplicar este objeto y seleccionar como "Controller Node" el controlador de la susodicha mano.

En este momento tenemos un rayo que permanece activo en todo momento, **para hacer que el rayo desaparezca** podemos **cambiar la transparencia** del rayo con la opción "**Invalid Color Gradient**" que se encuentra justo debajo de la opción con la que cambiamos el color del rayo, en el componente "**XR Interactor Line Visual**".



*Si **cambiamos el alfa del inicio y el final del degradado a 0**, el rayo será **transparente** cuando no esté apuntando a objetos dentro de las máscaras seleccionadas.*

Ahora tan solo nos falta **cambiar las interfaces** para hacer que actúen ante estos rayos. Para ello vamos a seleccionarlos y **añadir el componente "Tracked Device Graphic Raycaster"** y después buscaremos el objeto "**Event System**" en la jerarquía y le **añadiremos el componente "XRUI Input Module"**. Este objeto se generó automáticamente al crear el primer "Canvas". Para terminar, debemos **cambiar en la "tag"** de la cámara a **"Main Camera"**, con esto evitamos que el rayo siga apuntando a la interfaz cuando no la estemos apuntando con la mano.

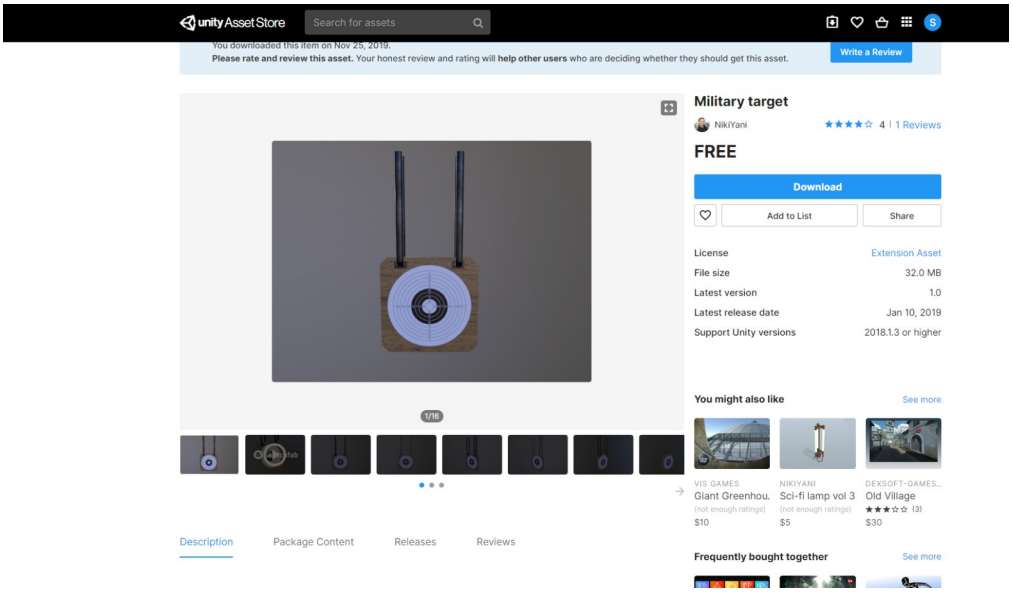


Con todo esto **hemos logrado**:

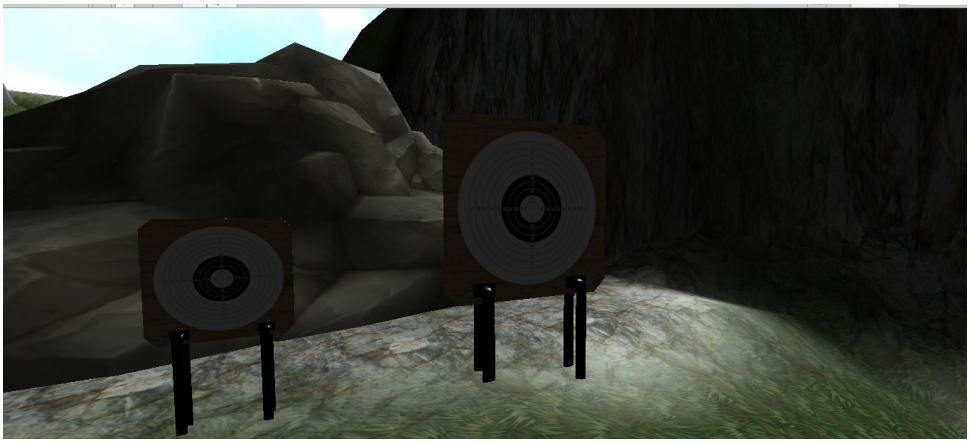
- Crear una **interfaz** que muestre las posiciones de los **disparos en la diana**, el número de flechas disparadas y la precisión numérica de la última flecha.
- Crear una **interfaz** para activar, desactivar y modificar los **pseudo-hápticos**.
- **Manejar las interfaces** en realidad virtual.

Parte 2.6: Diana

En esta subsección mostraremos **cómo crear una diana para disparar las flechas** y que **muestre el punto de disparo**. En primer lugar buscaremos un modelo de diana en la "**Asset Store**" o sino, crearemos nosotros uno. En mi caso buscaré el modelo en la "**Asset Store**".

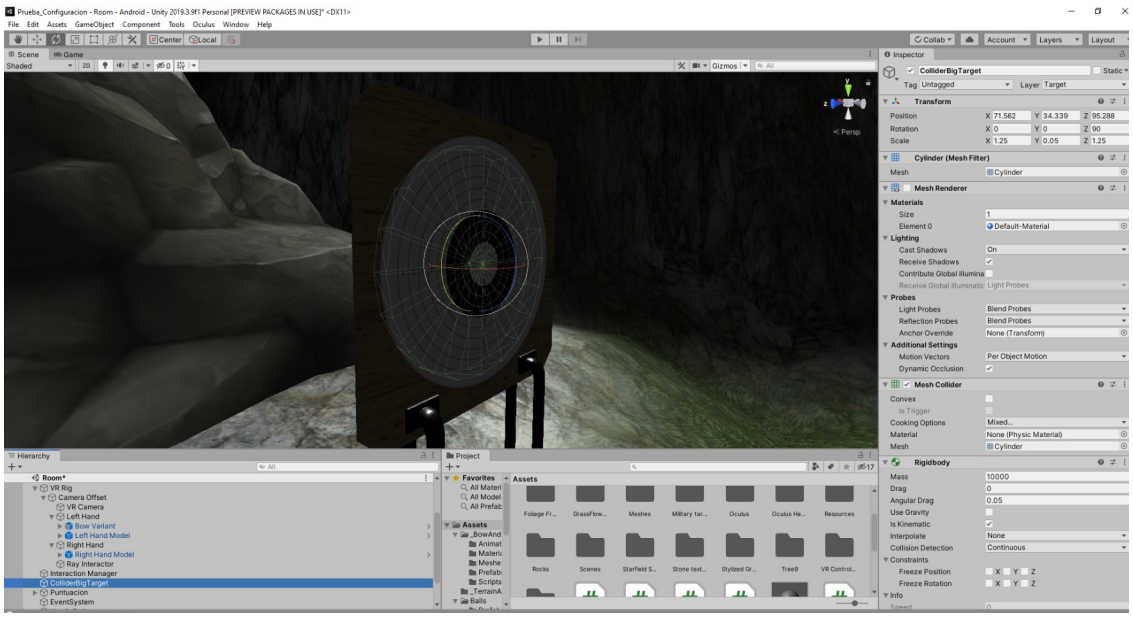


Ahora **posicionamos el modelo** de la diana en la escena moviendolo desde la ventana "project".



En estos momentos la diana **no posee funcionalidad alguna**, ni tan siquiera puede parar las flechas, para ello, lo que haremos es **añadir un "collider"**, el componente de Unity3D encargado de **controlar las colisiones entre objetos**.

Para crear un “collider” ajustado al objetivo de la diana, he decidido **crear un objeto aparte invisible** con forma de cilindro y le he puesto un **“mesh collider”**, el cual toma la forma del objeto en el que está contenido.



Para hacerlo invisible he **desactivado** el componente **“Mesh renderer”**, encargado de renderizar las texturas de un objeto. Además, he creado un **Rigidbody** y he seleccionado la opción “**isKinematic**” para evitar que la gravedad o la fuerza de las flechas lo muevan. Por último he cambiado su “**Layer**” a “**Target**”, el cual está físicamente relacionado en la matriz de colisiones con la capa “**Arrow**”, en la cual se hallan las flechas.

Toda la **lógica de las colisiones** se halla en el script “**Arrow.cs**” de las flechas, el cual mostré en el apartado 4. Ahora explicaré la parte relacionada con las colisiones de este script.

```
        //Collision
        if (Physics.Linecast(m_LastPosition, m_Tip.position,out RaycastHit hitInfo, layerMask))
        {

            Vector2 arrowPos = new Vector2(m_Tip.position.y, m_Tip.position.z);
            Vector2 targetPos = new Vector2(hitInfo.transform.position.y, hitInfo.transform.position.z);

            //Debug.Log("The distance is: " + Vector2.Distance(arrowPos, targetPos));

            //Debug.Log("the size: " + (hitInfo.transform.localScale.x / 2));

            //Debug.Log("The normalized distance is: " + (Vector2.Distance(arrowPos, targetPos)/(hitInfo.transform.localScale.x/2)));

            //Now we tell the puntuation
            cv.GetComponent<ScriptDePuntuacion>().accuracy = (int)(100*((hitInfo.transform.localScale.x / 2) - Vector2.Distance(arrowPos, targetPos))/(hitInfo.transform.localScale.x / 2));
            //Debug.Log(100 * ((hitInfo.transform.localScale.x / 2) - Vector2.Distance(arrowPos, targetPos)) / (hitInfo.transform.localScale.x / 2));

            //We change the Canvas of the RangeShot
            cv.GetComponent<ImpactScript>().X = m_Tip.position.y;
            cv.GetComponent<ImpactScript>().Y = m_Tip.position.z;
            cv.GetComponent<ImpactScript>().targetPosition = targetPos;
            cv.GetComponent<ImpactScript>().targetSize = hitInfo.transform.localScale.x/2;
            cv.GetComponent<ImpactScript>().changed = true;

            Stop();
        }

        //Store position
        m_LastPosition = m_Tip.position;

    }
```

Lo primero que hacemos es crear un bloque condicional en el cual entraremos solo si un rayo detecta un objeto de la capa deseada. Un rayo de tipo "**Linecast**" es una **línea virtual usada para detectar objetos que colisionan con ella**, esta línea se crea entre dos puntos, en nuestro caso "**m_LastPosition**" (el cual guarda la posición de la **punta de la flecha en el ciclo anterior** de la función **Update()** y "**m_Tip.position**" (es la posición actual de la punta de la flecha), de este modo **el rayo se generaría desde la posición de la punta en el ciclo anterior hasta la posición de la punta de la flecha en este ciclo**. Además, a esta línea **se le aplica una máscara** de tipos de objetos que puede detectar, siendo en este caso solo los de tipo "**target**".

Si entramos en este bloque condicional significa que el rayo ha detectado una colisión, por lo que lo primero que haremos es **almacenar la posición de la punta de la flecha** y la **posición de la diana**. La información sobre el objeto impactado por el "**Linecast**" se almacena en la variable "**hitInfo**".

```
Vector2 arrowPos = new Vector2(m_Tip.position.y, m_Tip.position.z);
Vector2 targetPos = new Vector2(hitInfo.transform.position.y, hitInfo.transform.position.z);

//Debug.Log("The distance is: " + Vector2.Distance(arrowPos, targetPos));
```

Después **obtenemos la precisión del disparo** como la distancia de la punta de la flecha al centro de la diana dividido entre el radio de la diana, todo ello normalizado para que quede como un valor entre 0 y 1, posteriormente lo multiplicamos por 100 y convertimos a entero.

```
//Now we tell the puntuation
cv.GetComponent<ScriptDePuntuacion>().accuracy = (int)(100*((hitInfo.transform.localScale.x / 2) - Vector2.Distance(arrowPos, targetPos))/(hitInfo.transform.localScale.x / 2));
//Debug.Log(100 * ((hitInfo.transform.localScale.x / 2) - Vector2.Distance(arrowPos, targetPos)) / (hitInfo.transform.localScale.x / 2));
```

Ahora lo que nos faltaría es **añadir un punto a la diana virtual** para poder guardar los resultados en esta. Para ello vamos pasarle a la función "**ImpactScript.cs**" (ya explicada en la subsección anterior) de la diana virtual la información sobre la coordenada **X** e **Y** de la p**unta de la flecha**, un **booleano positivo** para indicar que ha habido un cambio, el **tamaño del objetivo** y la **posición del objetivo**. Con todo ello el script "**ImpactScript.cs**" añadirá una **cruz roja** a la diana (imagen la cual importamos de una librería libre de derechos de autor).

```
//We change the Canvas of the RangeShot
cv.GetComponent<ImpactScript>().X = m_Tip.position.y;
cv.GetComponent<ImpactScript>().Y = m_Tip.position.z;
cv.GetComponent<ImpactScript>().targetPosition = targetPos;
cv.GetComponent<ImpactScript>().targetSize = hitInfo.transform.localScale.x/2;
cv.GetComponent<ImpactScript>().changed = true;
```

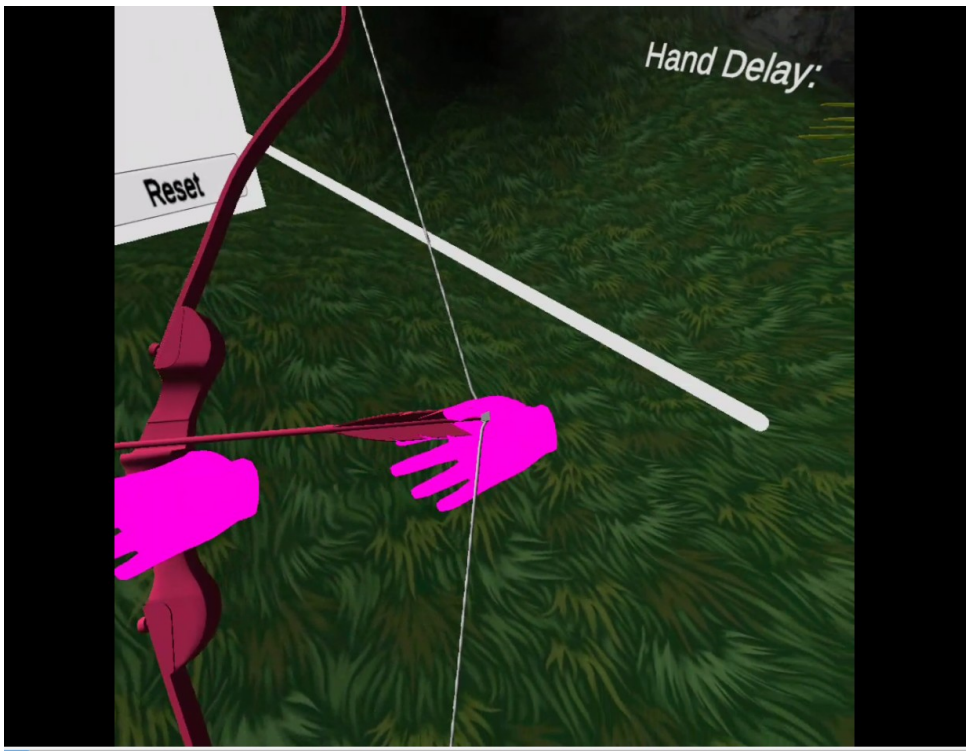
Finalmente detenemos la flecha con la función **Stop()**, la cual cambia el objeto flecha a "**isKinematic**" para que no pueda moverse.

Parte 2.7: Pseudo-hápticos

En secciones anteriores se han visto fragmentos en los códigos que no fueron explicados ya que creí conveniente juntar todos ellos en un apartado propio dedicado a los **pseudo-hápticos**.

Los pseudo-hápticos son técnicas enmarcadas dentro del campo de la **interacción humano-máquina** que tienen como fin **simular** de la manera más realista posible **sensaciones táctiles** en el usuario. Para ello se hace **uso de otros sentidos utilizados en el simulador**, en nuestro caso la vista y el propio tacto.

En este proyecto se han usado **tres formas** distintas de generar sensaciones pseudo-hápticas. La primera consiste en hacer **vibrar el controlador** de la mano actuadora de una forma proporcional a la tensión del arco, la segunda consiste en hacer **vibrar de forma visual el modelo de la mano**, haciendo que la vibración sea más errática y acentuada en base a la tensión del arco y un último modo en el cual la **posición de la mano virtual deja de corresponderse a la de la mano real**, creando un retraso de la posición que se vuelve más pronunciado dependiendo de un valor dado, o, de forma más sencilla y coloquial, sería como si tu mano virtual se hallará en donde tu mano real se encontraba hace unos segundos.



*Esta imagen mostraría este **tercer modo pseudo-háptico**, en el cual se ve como la **mano real** (que se halla en el origen del rayo blanco), se halla **adelantada** con respecto a la posición de la **mano que sujeta el arco virtual**.*

Empezaré explicando cómo se realizó el primer modo. Para ello debemos fijarnos en el script "**Bow.cs**", entre las **líneas 92** y **104**.

```
91
92          if(m_PullValue > 0.25f && ui.GetComponent<UIControllerBow>().tactile)
93          {
94              isVibrating = true;
95              //OVRInput.SetControllerVibration(0.1f, m_PullValue, OVRInput.Controller.All);
96              inputGO.GetComponent<OculusInput>().m_controller.SendHapticImpulse(0,m_PullValue,0.7f);
97              StartCoroutine(setVibration(0.5f));
98          }
99
100         if (m_PullValue < 0.25f)
101         {
102             isVibrating = false;
103             //inputGO.GetComponent<OculusInput>().m_controller.StopHaptics();
104         }
105
106     }
```

Para entrar en el bloque del primer "if" la **potencia de tensado** de tensado debe ser de **al menos un 25%** y la función "**tactile**" tiene que haber sido seleccionada en el botón de la interfaz gráfica de pseudo-hápticos, creada en el apartado 5. En caso de darse las dos condiciones pondremos a "**true**" la variable "**isVibrating**", que nos indica si el controlador está vibrando.

Después usaremos la función de la librería XR "**sendHapticImpulse**" sobre la variable que contiene el controlador derecho en el script "**OculusImput.cs**" del objeto "Input" ya explicada en el apartado 3. Esta función toma tres valores, el **canal**, el cual es común mantener en 0, la **amplitud de la vibración**, o lo que es lo mismo, la fuerza con la que vibrará el controlador (la cual se corresponderá con la tensión del arco) y la **duración de esta vibración** en segundos.

Con esto ya tendríamos esta función pseudo-háptica creada, y cuando presionemos el botón "**Tactile vibration**" nuestro **mando vibrará** cual realizar tensión con el arco. Ahora explicaré el segundo modo, el cual está también contenido en este script.

Para este modo tenemos el script "**VisualVibration.cs**", el cual se halla dentro del objeto que contiene el modelo de la mano derecha, y la función **Pull()**, la cual toma la posición de la mano virtual. Es importante tener en cuenta que tanto este modo **pseudo-háptico** como el siguiente hace uso del objeto que contiene el modelo de la mano, y no del objeto de la mano como tal, este modelo, al ser hijo de la mano real seguirá girando y desplazándose en el espacio global en base a esta.

```
134
        1 referencia
135     public void Pull(Transform hand)
136     {
137         float distance = Vector3.Distance(hand.position, startPoint.position);
138         if(distance > m_GrabThreshold)
139         {
140             visualVibration.transform.parent = handAnchor.transform;
141             visualVibration.transform.position = handAnchor.transform.position;
142             visualVibration.transform.localEulerAngles = new Vector3(0, 0, -90);
143             visualVibration.transform.localPosition = new Vector3(0, 0, 0);
144             return;
145         }
146         else
147         {
148             visualVibration.transform.parent = notch.transform;
149             visualVibration.transform.position = notch.transform.position;
150             visualVibration.transform.localEulerAngles = new Vector3(0, 0, -90);
151             visualVibration.transform.localPosition = new Vector3(0,0,0);
152         }
153
154         m_PullingHand = hand;
155     }
156
```

De esta función se explicó solo la primera línea en el apartado 4, en la cual se **calcula la distancia entre la mano y el punto de donde se toma la cuerda**. Luego en base a ello, si supera el umbral de la distancia necesaria para agarrar el arco se cambiará el padre del modelo de la mano derecha para que sea el objeto de la mano derecha real y más tarde se cambia la posición relativa a un vector de tres ceros para que sus posiciones sean la misma. Si por otro lado la distancia es menor que el umbral el padre será el punto de agarre de la cuerda. Esto se hace porque, en caso de que la distancia para agarrar la cuerda sea pequeña, la mano será adherida al arco, por lo que se perderá la posibilidad de mover la mano con libertad hasta que se suelte el arco. Esto se contempla en la función **Release()**, la cual se activa al soltar la flecha.

```
        1 referencia
157     public void Release()
158     {
159         visualVibration.transform.parent = handAnchor.transform;
160         visualVibration.transform.position = handAnchor.transform.position;
161         visualVibration.transform.localEulerAngles = new Vector3(0, 0, -90);
162         visualVibration.transform.localPosition = new Vector3(0, 0, 0);
163
164         if (m_PullValue > 0.25f)
165         {
166             audioS.PlayOneShot(throwArrow);
167             FireArrow();
168         }
169         m_PullingHand = null;
170         m_PullValue = 0.0f;
171         m_Animator.SetFloat("Blend", 0.0f);
```

*Las cuatro primeras líneas de la función **Release()** son idénticas a las de la función **Pull(Transform)**, ya que al soltar el arco ha de liberarse la mano.*

Ahora pasaría a actuar el script "**VisualVibrationScript.cs**", en el cual realizaremos los cambios de posición relacionados a la vibración.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

1 referencia
public class VisualVibrationScript : MonoBehaviour
{

    public GameObject hapticOptions;
    public GameObject bow;
    public float cuerda = 0.0f;

    private Vector3 originalPosition;



    0 referencias
    private void Start()
    {
        originalPosition = gameObject.transform.localPosition;
        cuerda = 0.0f;
    }

    0 referencias
    void Update()
    {
        //We will make the vibration from the string lenght

        if (hapticOptions.GetComponent<UIControllerBow>().visual)
        {

            if (bow.GetComponent<Bow>().m_PullValue > 0.25f)
            {
                //We get a random X, Y and Z
                float X = Random.Range(-0.01f, 0.01f) * (bow.GetComponent<Bow>().m_PullValue - 0.25f);
                float Y = Random.Range(-0.01f, 0.01f) * (bow.GetComponent<Bow>().m_PullValue - 0.25f);
                float Z = Random.Range(-0.01f, 0.01f) * (bow.GetComponent<Bow>().m_PullValue - 0.25f);

                //We must also applie the vibration to the string
                cuerda = Random.Range(-0.03f, 0.03f) * (bow.GetComponent<Bow>().m_PullValue - 0.25f);

                gameObject.transform.localPosition = new Vector3(X,Y,Z);
            }
            else
            {
                gameObject.transform.localPosition = originalPosition;
            }

        }
        else
        {
            gameObject.transform.localPosition = originalPosition;
            cuerda = 0.0f;
        }


    }
}

```

Este script posee **tres variables públicas** y **una privada**:

-"**hapticOptions**" es una variable pública que contiene una **referencia a la interfaz de las opciones hápticas**.

-"**bow**" es una variable pública que contiene una **referencia al arco**.

-"**cuerda**" es una variable pública de tipo float en la cual **guardaremos el desfase en la posición de la cuerda de la flecha**.

-"**originalPosition**" es una variable privada de tipo Vector3 que nos servirá para almacenar la **posición original de la mano** antes de comenzar a vibrar.

En la función de **Start()** inicializamos la variable “cuerda” a 0 y la variable “originalPosition” con la posición actual del modelo de la mano.

```
0 referencias
private void Start()
{
    originalPosition = gameObject.transform.localPosition;
    cuerda = 0.0f;
}
```

En la función **Update()** tendremos dos condiciones, que el modo vibración visual esté activado y que la fuerza de tensión sea mayor al 25%. Si la primera condición no se da se repite el código de la función **Start()** para poner a 0 el desfase de la cuerda y la posición del modelo mano. Si la segunda condición no se da, se hace lo mismo solo que manteniendo el desfase de la mano.

En el caso de que ambas condiciones se den elegiremos de forma aleatoria el desfase de la posición de la mano en la coordenada **X**, en la coordenada **Y** y en la coordenada **Z**, de tal forma que este valor se verá **magnificado cuanto más tensa esté la cuerda**. Haremos lo mismo con la variable “cuerda”. Después de ello haremos que la posición local (con respecto a su padre) de la mano sea la posición (X,Y,Z). Esto se realizará en cada iteración, por lo que dará una sensación de que la mano y la cuerda está vibrando dependiendo de la tensión del arco, como pasaría en la realidad al pasar mucho tiempo con los brazos en posición de disparo.

```
//We get a random X, Y and Z
float X = Random.Range(-0.01f, 0.01f) * (bow.GetComponent<Bow>().m_PullValue - 0.25f);
float Y = Random.Range(-0.01f, 0.01f) * (bow.GetComponent<Bow>().m_PullValue - 0.25f);
float Z = Random.Range(-0.01f, 0.01f) * (bow.GetComponent<Bow>().m_PullValue - 0.25f);

//We must also applie the vibration to the string
cuerda = Random.Range(-0.03f, 0.03f) * (bow.GetComponent<Bow>().m_PullValue - 0.25f);

gameObject.transform.localPosition = new Vector3(X,Y,Z);
```

Ahora lo único que faltaría es modificar la posición de la cuerda, para lo cual volveremos a observar el script “Bow.cs” en la línea 71. En esta línea se **modifica la tensión de la cuerda ligeramente usando el valor obtenido**, de tal forma que la cuerda también vibrará dependiendo de la tensión.

```
    m_PullValue = Mathf.Clamp(m_PullValue, 0.0f, 1.0f);

    m_PullValue = m_PullValue + visualVibration.GetComponent<VisualVibrationScript>().cuerda;

    //Ahora vamos a añadir un delay
    Debug.Log("CHANGE IN POSITION: " + (former_PullValue - m_PullValue));
```

Con esto ya tendríamos dos efectos pseudo-hápticos, uno en el que se utiliza el tacto simular otra sensación táctil y otra en la que se usa la vista para simular el tacto. En el tercer modo pseudo-háptico también se usa el sentido de la vista para reforzar el tacto, pero de un modo distinto.

Para este último modo pseudo-háptico usaremos el **slider** de la interfaz de pseudo-hápticos, pudiendo elegir la fuerza con la que se aplicará este efecto entre 0 y 100, donde 0 sería sinónimo de no estar usando este efecto. En primer lugar debemos recordar que mientras el usuario esté pellizcando la cuerda su mano virtual tendrá como padre el centro de la cuerda, por lo que **con controlar el valor de la tensión de la cuerda controlaríamos la posición de la mano**.

Para este modo usaremos de nuevo exclusivamente el script "**Bow.cs**"

```
76      if ((m_PullValue > former_PullValue) && ui.GetComponent<UIControllerBow>().delay)
77      {
78          m_PullValue = former_PullValue + (m_PullValue - former_PullValue) / (1 + ui.GetComponent<UIControllerBow>().delayAmount*m_PullValue);
79      }
80
81      if (m_PullValue > 1.0f)
```

Este modo de juego se ejecuta con tan solo una línea de código dentro de un bloque "if". Para entrar en el bloque "if" la **tensión** actual debe ser **mayor que la anterior**, por ello el efecto de **"delay"** se acaba en cuanto la mano real se encuentra en la misma posición que la mano virtual. Esto se aplica tanto si destensas la cuerda manualmente como si sencillamente la mano virtual alcanza la mano real después de pasar un tiempo. Además de esta condición, deberá cumplirse que la función haya sido activada desde el panel de pseudo-hápticos.

Para calcular nuestra tensión final de la cuerda **sumaremos a la tensión del ciclo anterior** la **distancia entre la tensión de la mano real** (la cual en este momento del código se halla en la variable "**m_PullValue**" pero será reemplazado por el valor tras aplicar esta ecuación) y la tensión de la cuerda en el ciclo anterior (la cual fue guardada tras aplicar esta ecuación en el ciclo anterior). A la distancia entre esas dos tensiones l**e dividiremos 1 + la cantidad de fuerza del delay** que hemos seleccionado en el panel de pseudo-hápticos **multiplicado por el valor de tensión de la mano real**. En caso de que la cantidad de fuerza seleccionada fuera 0, estaríamos dividiendo la distancia entre la tensión según la mano virtual y la tensión según la mano real entre 1, por lo que el nuevo resultado sería igual al resultado antiguo, por lo que es como si esta opción no estuviera activada, actuado entonces como el modo "clásico". Por otro lado, si el valor de la fuerza fuera 100, la distancia entre la tensión de la mano virtual y la mano real sería fuertemente penalizada por la división, haciendo que fuera creciendo poco a poco hasta eventualmente alcanzar la tensión de la mano real.

Con esto ya habríamos **terminado todo lo referente a la aplicación del arco**, exceptuando la conectividad, la cual veremos en profundidad en el capítulo 2.

Parte 3: Planetas VR

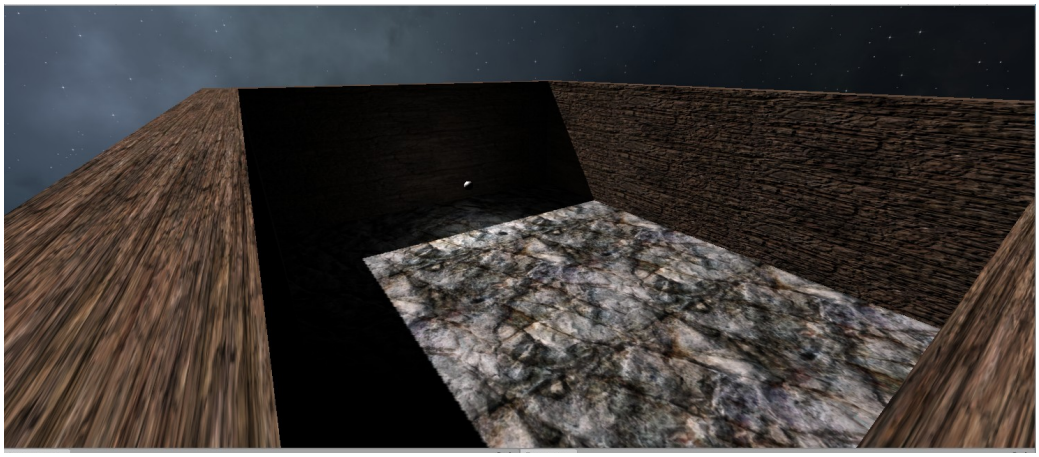
Parte 3.1: Objetivos

El objetivo de esta aplicación es **crear un conjunto de planetas en situación de ingravidez** dentro de una sala cerrada, estos planetas **podrán ser agarrados por el jugador** utilizando los controladores, y, cada uno de estos tendrá asociado otro planeta, sobre el cual se realizará una **fuerza de tensión atrayente** entre las manos al

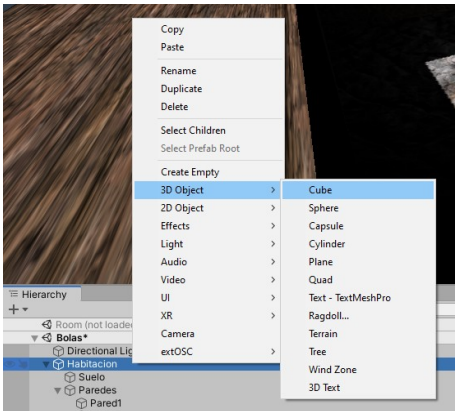
estar estos dos planetas agarrados. En este juego el usuario podrá **desplazarse por toda la sala** para poder interactuar libremente con los objetos.

Parte 3.2: El entorno

Al contrario que con la aplicación anterior, el entorno no estará creado utilizando la función de creación de terrenos de Unity3D, en este caso tan solo **utilizaremos las primitivas** de Unity y una "**Skybox**" para crear un cielo acorde con el ambiente.



Para ello crearemos 5 objetos del tipo **Cube**, y les cambiaremos la **escala**, la **posición** y la **rotación** utilizando los gizmos hasta obtener una sala lo suficientemente amplia con 4 paredes y un suelo.



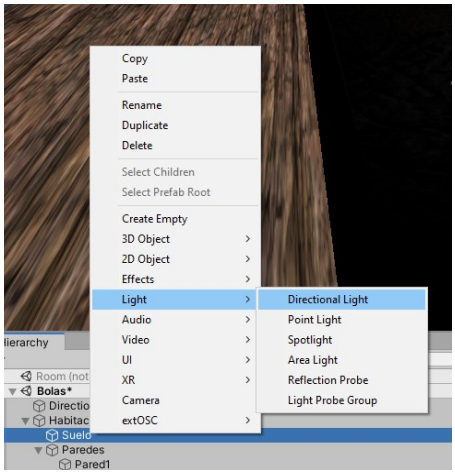
Por defecto las primitivas no poseen un material, tan solo son objetos de color blanco. Si queremos **cambiar el material** del que está compuesto, añadimos un **material** al juego de la "Asset store" o de fuentes externas al proyecto y lo arrastramos a la ventana de componentes del objeto. En mi caso he añadido un **material de piedra marrón a las paredes** y de piedra **grisácea al suelo**.

Tras añadir una **caja de colisiones** a cada uno de los objetos que conforman la sala ya tendríamos el **entorno básico** creado. Para dar una mayor sensación de inmersividad al usuario añadiremos una fuente de **iluminación** y un **cielo**. Primero empezaremos con la iluminación, eligiendo entre los 3 tipos básicos de fuentes de iluminación de Unity3D:

-"**Directional Light**": Se trata de una forma de iluminación **similar a la del sol**, de tal forma que se aplica sobre el terreno con un ángulo concreto. En este caso utilizaremos esta forma de iluminación.

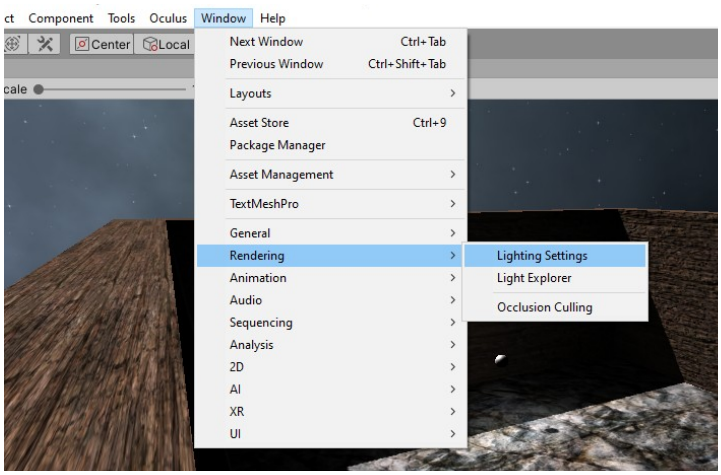
-"**Point Light**": Esta luz sería **similar a la de una bombilla** o un candil en tanto que se genera en el punto central del objeto y se expande de forma esférica a mayor o menor distancia dependiendo de los valores con la que lo configuremos. La usaremos en un futuro para dotar de luz a los planetas.

-"**Spotlight**": Esta luz sería **similar a la producida por una linterna**, tomando una forma cónica. Finalmente no ha sido utilizada en este proyecto.



Existen más tipos de fuentes de luz, pero estas tres son las más usadas generalmente. Una vez añadida la **luz direccional** y seleccionado el ángulo solo restaría seleccionar un cielo para el entorno. En los videojuegos los cielos suelen crearse utilizando una **skybox**, la cual funciona, como su nombre indica, como una **caja con una cierta textura** en la cual está contenido el entorno de juego.

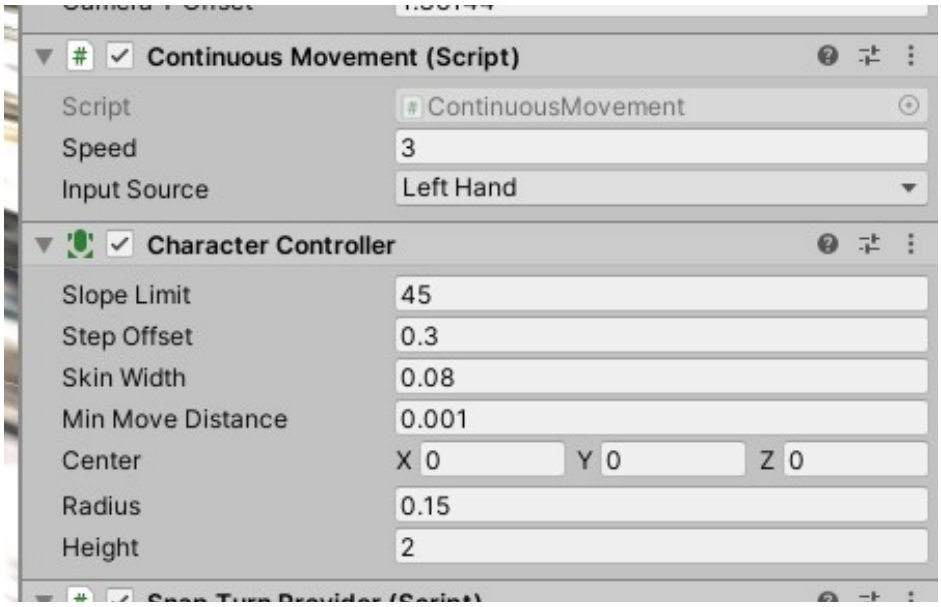
Es muy sencillo de añadir, tan solo debes importar una "**skybox**" y arrastrarla a la pestaña "**skybox material**" dentro de la ventana de "**Lighting Settings**". En mi caso usaré un cielo estrellado importado de la "**Asset Store**".



Parte 3.3: El jugador

Afortunadamente la mayoría de pasos para crear un jugador son idénticos a los vistos en la aplicación del "Arco VR". En este proyecto el jugador será creado **ampliando la funcionalidad del jugador del proyecto anterior** añadiendo además la posibilidad de **desplazarse** por el suelo usando el joystick del controlador y también la posibilidad de usar los controladores para **agarrar objetos** del mundo.

Para poder **caminar con el personaje**, añadiremos 2 componentes al objeto "**VR Rig**", el objeto padre en el que se encuentran todos los elementos relacionados con el jugador. El primer componente sería uno predeterminado de las librerías XR llamado "**Character Controller**", el cual controlará el **ancho del personaje**, la **distancia mínima de desplazamiento**, la **máxima pendiente escalable**… y un script desarrollado por nosotros llamado "**ContinuousMovement.cs**".



El susodicho script posee las siguientes variables públicas:

-"**Speed**": Esta variable de tipo float controla la **velocidad** a la que podrá moverse el jugador.

-"**InputSource**": Esta variable indica el **controlador con el cual vamos a controlar el desplazamiento**, en nuestro caso el controlador de la mano izquierda.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR;
using UnityEngine.XR.Interaction.Toolkit;

0 referencias
public class ContinuousMovement : MonoBehaviour
{
    public float speed = 1;
    public XRNode inputSource;
    private Vector2 inputAxis;
    private CharacterController character;
    private XRRig rig;

    // Start is called before the first frame update
    0 referencias
    void Start()
    {
        character = GetComponent<CharacterController>();
        rig = GetComponent<XRRig>();
    }

    // Update is called once per frame
    0 referencias
    void Update()
    {

        InputDevice device = InputDevices.GetDeviceAtXRNode(inputSource);
        device.TryGetFeatureValue(CommonUsages.primary2DAxis, out inputAxis);

    }

    0 referencias
    private void FixedUpdate()
    {
        Quaternion headYaw = Quaternion.Euler(0,rig.cameraGameObject.transform.eulerAngles.y,0);

        Vector3 direction = headYaw * new Vector3(inputAxis.x,0,inputAxis.y);

        character.Move(direction * Time.fixedDeltaTime * speed);
    }


}
```

El código además posee tres variable privadas:

-"**inputAxis**": Una variable consistente en un vector de dos elementos en la cual se almacena el **estado del joystick**.

-"**character**": Una variable con la que haremos referencia al "**Character Controller**" ya mencionado.

-"**rig**": Una referencia al componente "**XR Rig**" ya explicado en la sección del "Arco VR".

Primero obtendremos las referencias a estos dos componentes en la función **Start()**.

```
0 referencias
void Start()
{
    character = GetComponent<CharacterController>();
    rig = GetComponent<XRRig>();
}
```

En este caso usaremos tanto la función **Update()** como la función **FixedUpdate()**. Si recordamos, esta última se utiliza cuando queremos hacer que ciertas partes del código sean independientes de la velocidad del

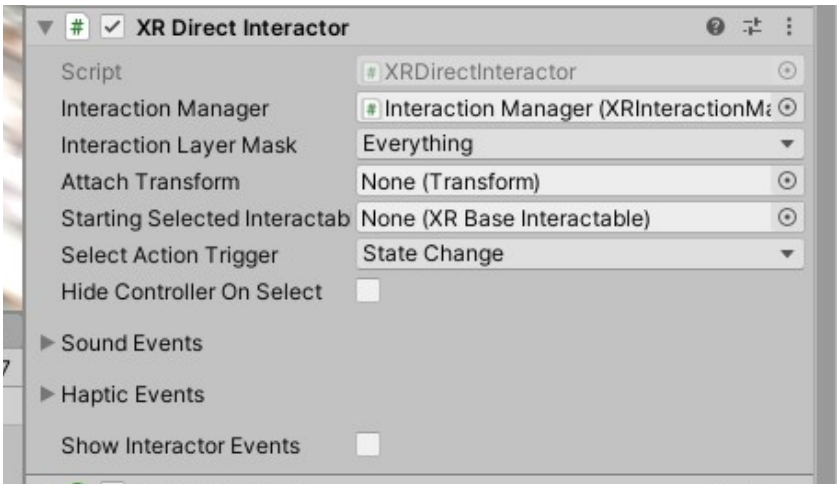
ordenador, **evitando así que se aplique más veces una acción solo por tener una máquina con mayor rendimiento**. En la función **Update()** obtendremos de forma constante el **valor del joystick** y lo almacenaremos en la variable "**inputAxis**".

```
0 referencias
void Update()
{

    InputDevice device = InputDevices.GetDeviceAtXRNode(inputSource);
    device.TryGetFeatureValue(CommonUsages.primary2DAxis, out inputAxis);

}
```

En la función **FixedUpdate()** obtendremos en primer lugar la **rotación de la cámara** para así saber en todo momento cual es la dirección frontal al jugador. Con esta información obtendremos el **vector de desplazamiento del jugador**, **multiplicando a la dirección antes obtenida el vector del estado del joystick convertido a un vector tridimensional**, ya que también tenemos la opción de utilizar la tercera dimensión para poder desplazarnos hacia arriba o abajo, pero no vamos a hacer uso de ella. Por último, vamos a llamar a la función integrada en el componente "**Character Controller**" llamada **Move()** para mover al jugador, multiplicando la nueva dirección obtenida por la velocidad y por "**Time.fixedDeltaTime**" para hacer que la velocidad de movimiento no varíe entre maquinas de distintas prestaciones.

```
0 referencias
private void FixedUpdate()
{
    Quaternion headYaw = Quaternion.Euler(0,rig.cameraGameObject.transform.eulerAngles.y,0);

    Vector3 direction = headYaw * new Vector3(inputAxis.x,0,inputAxis.y);

    character.Move(direction * Time.fixedDeltaTime * speed);
}
```

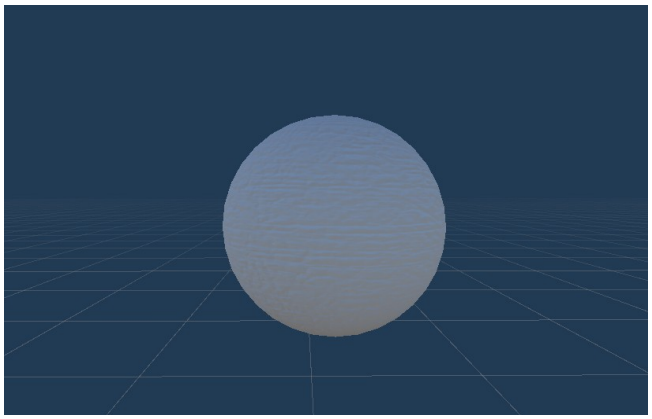
Ahora vamos a permitir al jugador recoger objetos, en este caso los planetas que crearemos en el siguiente subapartado. Para ello añadiremos a ambas manos un script de la librería XR llamado "**XR Direct Interactor**". Con esto no sería suficiente para poder agarrar los planetas, pero esto último será creado en el siguiente subapartado.



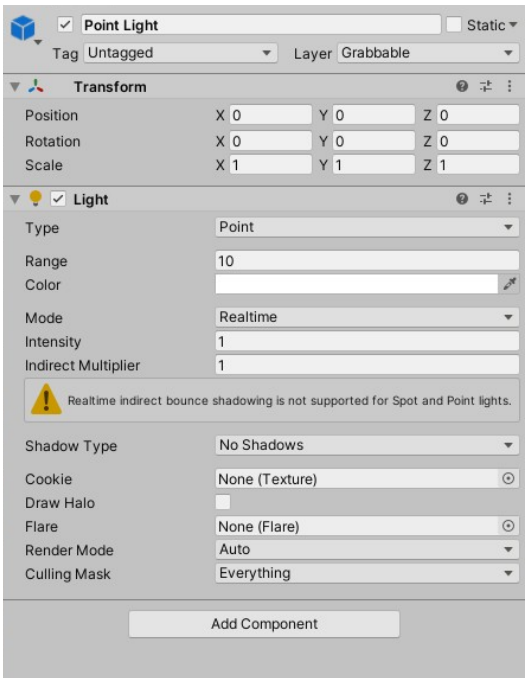
Parte 3.4: Los planetas

Con lo que tenemos ahora ya podíamos desplazarnos con nuestro personaje libremente por la sala creada, pero nos faltarían las interacciones con objetos, en este caso los objetos con los que interactuar serán planetas suspendidos en el aire, que podrán ser agarrados, chocados y lanzados.

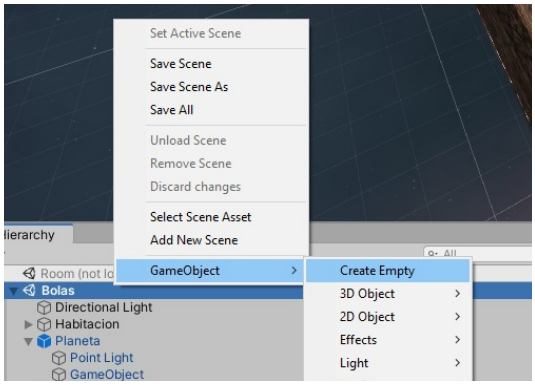
El objeto "**planeta**" consistirá en una **esfera con una textura**, esto lo realizamos del mismo modo que hicimos las paredes en el subapartado del entorno, además, meteremos el objeto en una nueva **capa** llamada "**Grabbable**".



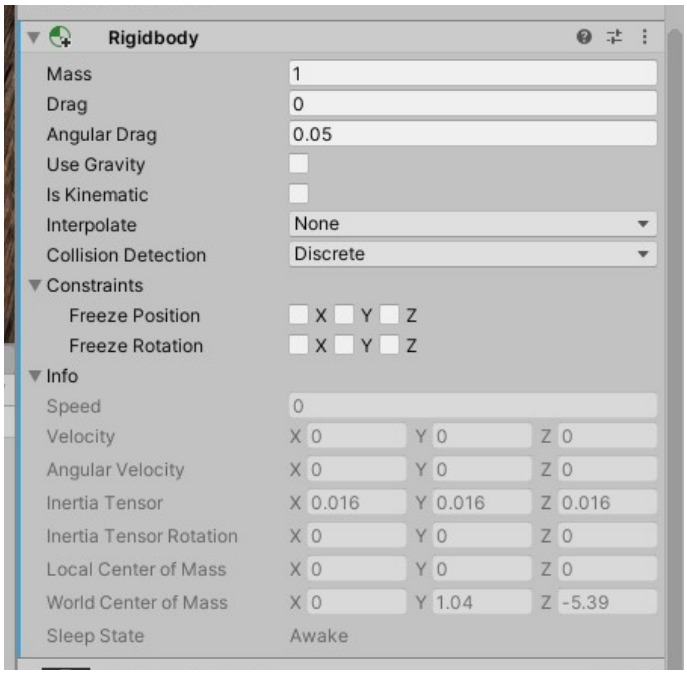
Para evitar que se vean demasiado oscuras **añadiremos una tenue luz blanca al objeto**, en este caso utilizaremos una luz de tipo "**point light**". Con este componente podremos modificar su intensidad y color de la luz.



Ahora todo lo que tenemos del planeta l**o metemos dentro de un objeto vacío** que contendrá tanto la forma como la iluminación de la esfera. Esto lo hacemos para evitar problemas a la hora de desarrollar los pseudo-hápticos.



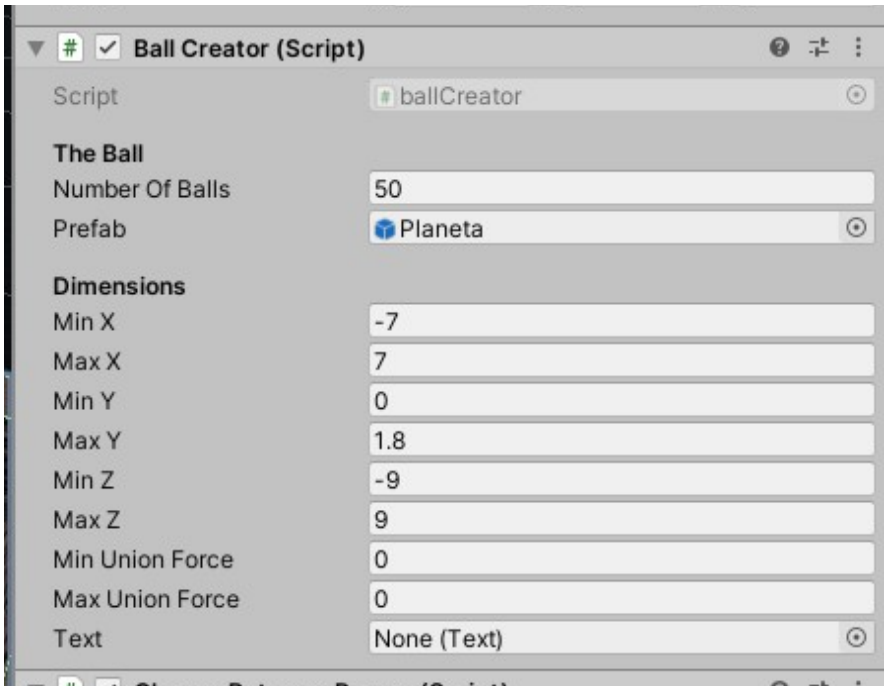
Por último haremos que el planeta pueda colisionar y que no se vea afectado por la gravedad, para ello crearemos un y un "**Rigidbody**" y un "**Sphere Collider**" dentro del modelo del planeta si no los tenía. En el "**Rigidbody**" deseleccionamos la opción "**use gravity**".



Con esto ya tendríamos un planeta creado. Por desgracia solo tenemos **un planeta** generado de forma **determinista** por el desarrollador y que además **no tiene ningún tipo de interacción**.

Para crear más planetas de forma aleatoria vamos a arrastrar el planeta creado a la ventana "**project**", con esto hemos creado un "**prefab**", el cual modificará todos las copias que existan en la escena cuando sea modificado. Es una opción muy útil si queremos crear varios objetos de un mismo tipo.

Para **generar de forma aleatoria planetas** crearemos un script llamado "**BallCreator.cs**", el cual guardaremos dentro de un objeto que contenga el entorno del juego.



Este script posee las siguientes **variables públicas**:

-"**Number Of Balls**": Una variable de tipo entero con la que el desarrollador indica el número de planetas que queremos crear. Este **número** deberá ser **par** para que cada planeta tenga otro asociado.

-"**Prefab**": Una variable en la cual hay una referencia al "**prefab**" **del planeta**.

-"**Min X**", "**Max X**", "**Min Y**", "**Max Y**", "**Min Z**", "**Max Z**": Variables para delimitar el **área en la que generar los planetas**.

-"**Min Union Force**", "**Max Union Force**" y "**Text**": Variables que han sido finalmente rechazadas.

```
using System.Collections;
using System.Collections.Generic;
using System.Globalization;
using UnityEngine;
using UnityEngine.UI;

0 referencias
public class ballCreator : MonoBehaviour
{
    [Header("The Ball")]
    public int numberOfBalls;
    public GameObject prefab;

    [Header("Dimensions")]
    public float minX;
    public float maxX;
    public float minY;
    public float maxY;
    public float minZ;
    public float maxZ;

    public float minUnionForce;
    public float maxUnionForce;

    planetScript referenceScript;

    public Text text;

    // Start is called before the first frame update
    0 referencias
    void Start()
    {
        //Get a seed
        //Generate the amount of balls specified
        for (int i = 0; i < numberOfBalls; i++)
        {
            //We generate the x, y, z
            float x = Random.Range(minX, maxX);
            float y = Random.Range(minY, maxY);
            float z = Random.Range(minZ, maxZ);

            //We generate the force
            //

            //We create the ball
            GameObject planet = Instantiate(prefab, new Vector3(x, y, z), Quaternion.identity);
            planet.name = "planet" + i.ToString();
```

```
            referenceScript = planet.GetComponent<planetScript>();
            referenceScript.myNumber = i;

            //Le asignamos la variable de Texto de la interfaz
            referenceScript.text = text;

            if (i % 2 == 0)
            {
                referenceScript.associatedNumber = i+1;
            }
            else
            {
                referenceScript.associatedNumber = i-1;
            }

            planet.GetComponent<Renderer>().material.color = Color.white;

        }

    }

    // Update is called once per frame
    0 referencias
    void Update()
    {

    }
}
```

En este script usamos únicamente la función **Start()** en la que hay un bucle que se ejecuta una vez por cada planeta que queramos crear. En primer lugar **elegimos** de forma **aleatoria tres números** dentro del área donde queremos **instanciar** los **planetas**.

```
//We generate the x, y, z
float x = Random.Range(minX, maxX);
float y = Random.Range(minY, maxY);
float z = Random.Range(minZ, maxZ);
```

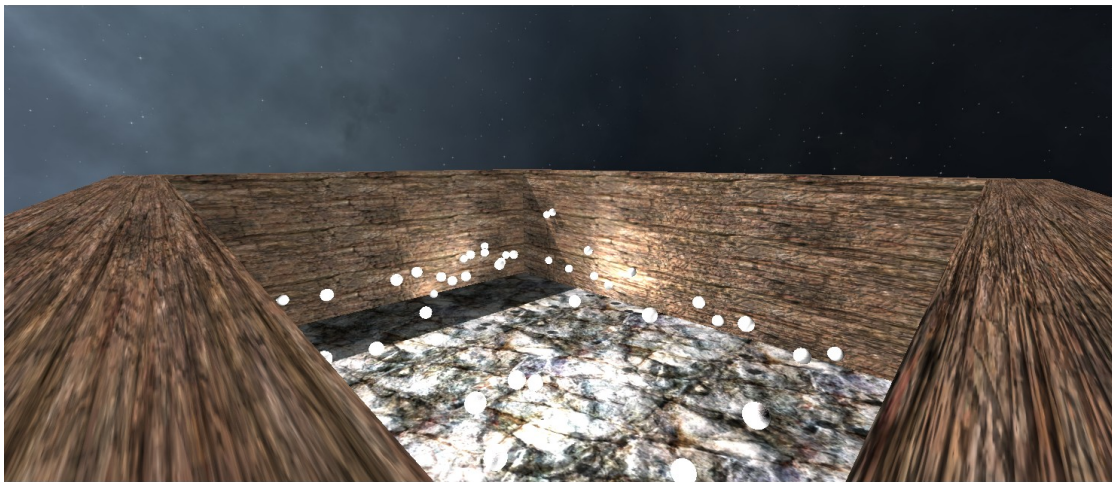
Después **instanciamos** un planeta en esa posición y le daremos un nombre del tipo **"planeta"+"numero del planeta"**. Además, modificaremos este número en el script "**PlanetScript.cs**", el cual posee cada uno de los planetas y que veremos más adelante.

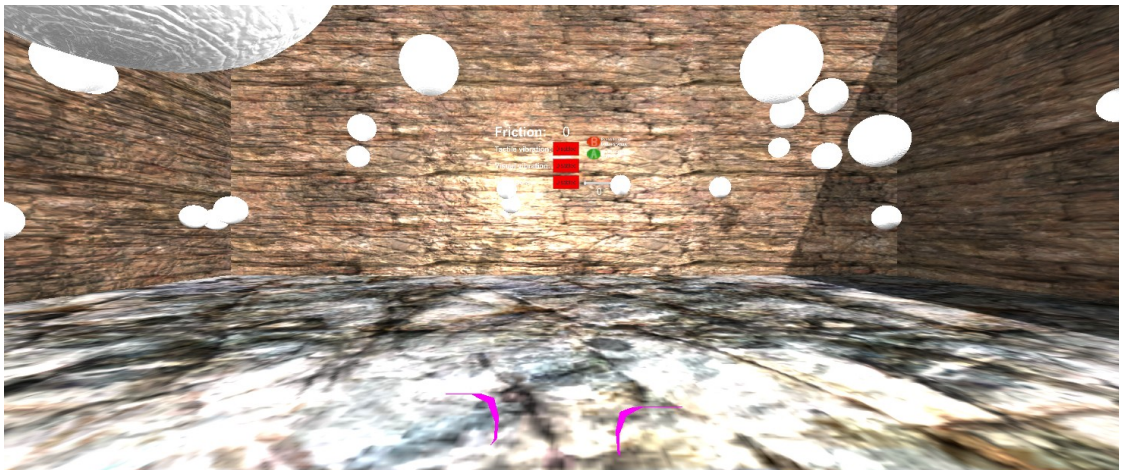
```
//We create the ball
GameObject planet = Instantiate(prefab, new Vector3(x, y, z), Quaternion.identity);
planet.name = "planet" + i.ToString();

referenceScript = planet.GetComponent<planetScript>();
referenceScript.myNumber = i;
```

Luego creamos las referencias del planeta de tal forma que en caso de que su **número** sea **par** se **asociará** con el **siguiente número**, y, en caso **contrario** con el **anterior**. Al comenzar los números en 0 y terminar en un número impar, ningún planeta permanecerá sin un planeta asociado. Por último, cambiamos el color del material a blanco.

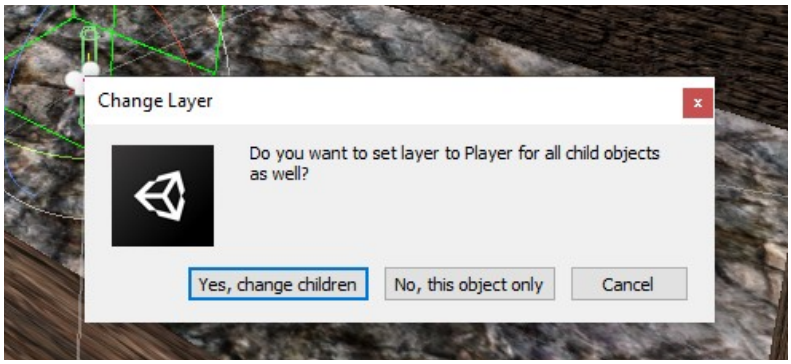
```
if (i % 2 == 0)
{
    referenceScript.associatedNumber = i+1;
}
else
{
    referenceScript.associatedNumber = i-1;
}

planet.GetComponent<Renderer>().material.color = Color.white;
```

Con esto generaríamos una nube de planetas aleatoria cada vez que carguemos esta escena.

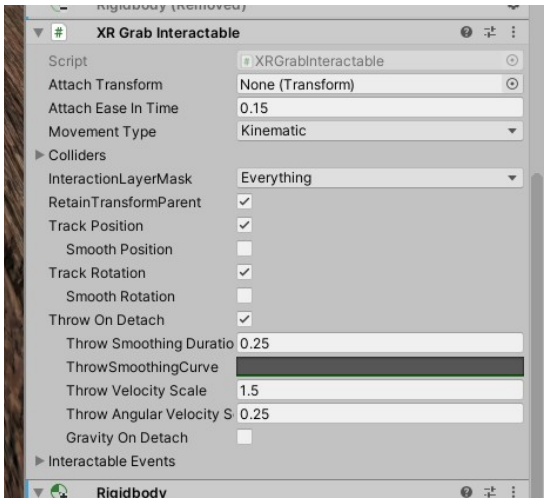


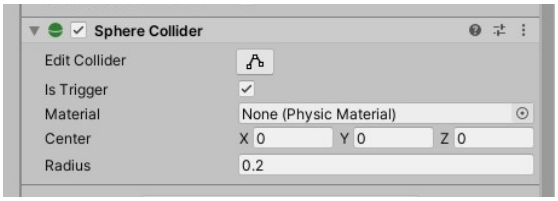


Ahora los **planetas pueden interactuar entre sí**, pero el **jugador no podría interactuar** con ellos salvo colisionando con el cuerpo virtual, efecto que no es conveniente ni agradable para el jugador. Esto lo podemos solucionar añadiendo nuestro objeto raíz del jugador en una capa, en mi caso la capa "**Player**" y cambiando la matriz de colisiones para que la capa "**Grabbable**" y la capa "**Player**" no puedan colisionar. Es importante que cuando te pregunte si quieres cambiar la capa solo para este objeto o para sus hijos también selecciones "**No, this object only**".



Ahora tan solo nos falta que el jugador pueda **agarrar los planetas**, para ello vamos a añadir al objeto **prefab** de los planetas un script "**XR Grab Interactable**" y a cada una de las manos un "**sphere collider**" con la opción "**isTrigger**" seleccionada.

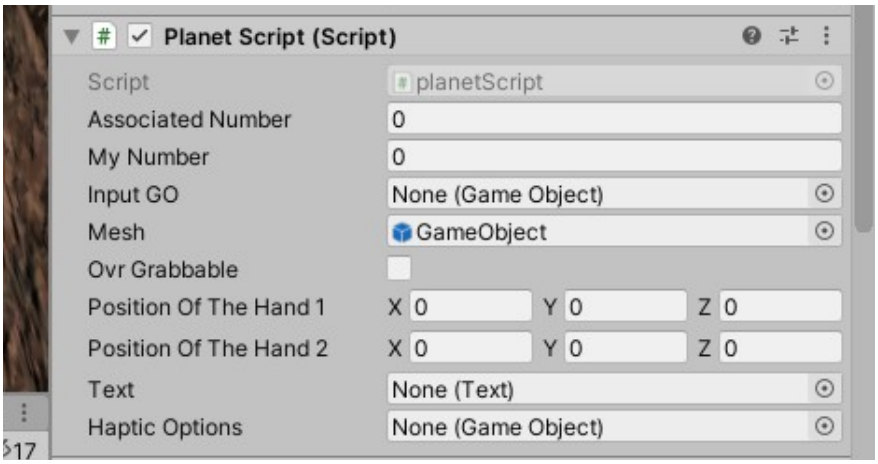




Con esto ya podríamos agarrar los planetas y moverlos a nuestro antojo, pero no ofrecería ningún tipo de respuesta al usuario, ni clásica ni pseudo-háptica. Para ello vamos a añadir al **prefab** de planeta un script propio al que llamaremos "**planetScript.cs**". En este script tenemos las siguientes variables públicas:

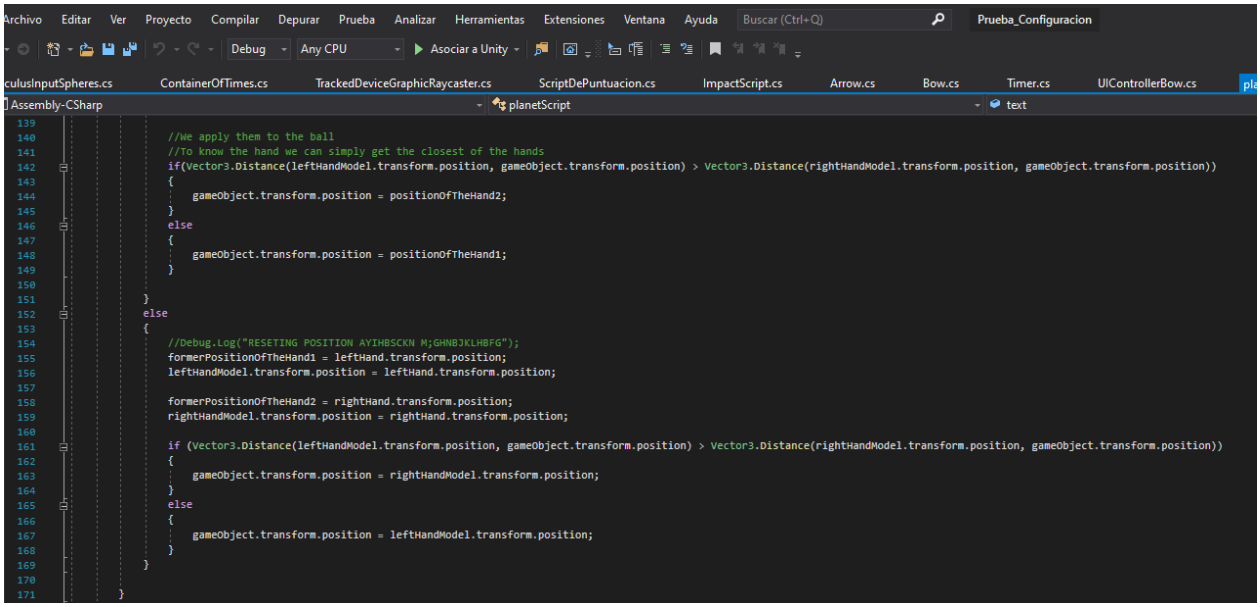
- "**Associated Number**": Es una variable entera que contiene el **id del planeta asociado** a este planeta.
- "**My Number**": Es una variable entera que contiene el **id del planeta**.
- "**Input GO**": Es una variable que contiene la **referencia al objeto** que contiene el análogo en esta aplicación al script "**OculusInput.cs**", aunque con ligeros matices.
- "**Mesh**": Es una variable que contiene la **referencia a la textura del planeta**.

- "**OvrGrabbable**": Es una variable **booleana** que nos indica si el planeta **está siendo agarrado o no**.
- "**Position Of The Hand 1**": Es una variable de tipo Vector3 que almacena la **posición de la mano derecha**.
- "**Position Of The Hand 2**": Es una variable de tipo Vector3 que almacena la **posición de la mano izquierda**.
- "**Text**": Es una variable que almacena la referencia al **texto** donde se mostrará la **tensión entre los 2 planetas agarrados**. (Finalmente esta variable no se utiliza).
- "**Haptic Options**": Es una variable que almacena la referencia al **panel de opciones pseudo-hápticas**.

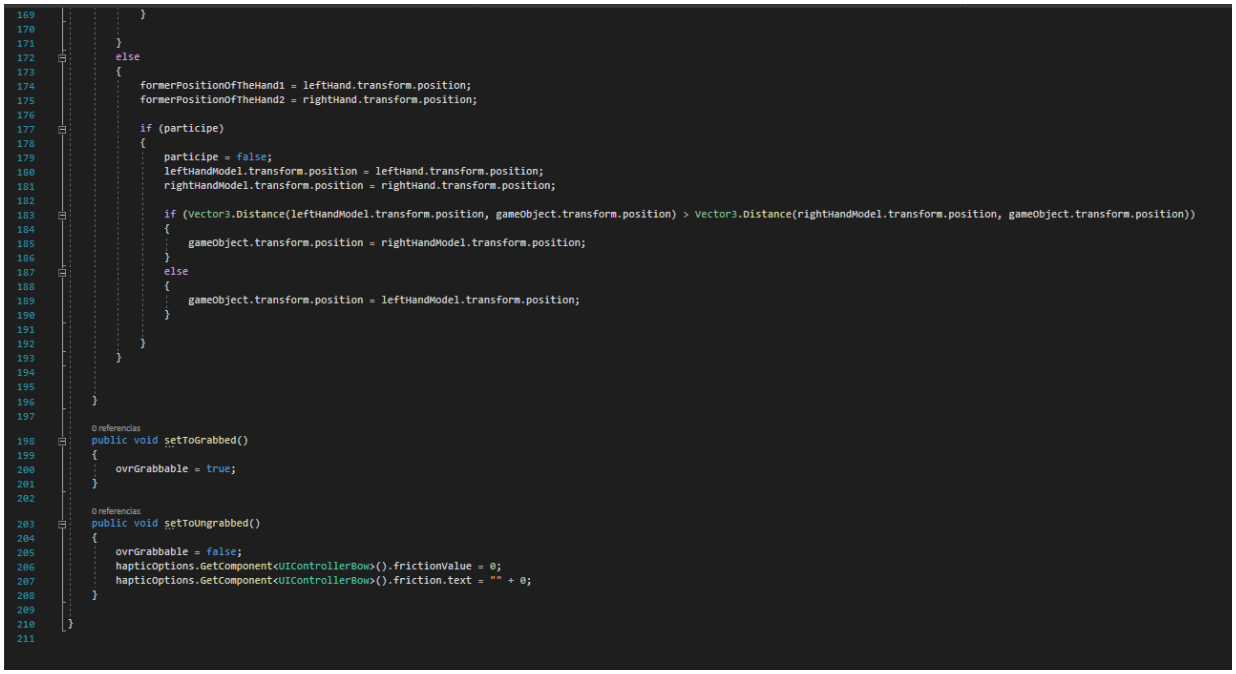



```
0 referencias
void Update()
{

    //Conseguimos su asociado
    GameObject planet = GameObject.Find("planet" + associatedNumber.ToString());

    if (ovrGrabbable)
    {
        //Le cambiamos el color
        planet.GetComponent<planetScript>().mesh.GetComponent<Renderer>().material.color = Color.red;
    }
    else
    {
        //Le quitamos el color
        planet.GetComponent<planetScript>().mesh.GetComponent<Renderer>().material.color = Color.white;
    }

    //Si ambos estan agarrado podremos visualizar la distancia entre ambos

    if (ovrGrabbable && planet.GetComponent<planetScript>().ovrGrabbable)
    {
        participe = true;

        formerDistance = x;

        dist = Vector3.Distance(gameObject.transform.position, planet.transform.position);

        x = dist / 2;

        x = (float)Math.Round(x, 2);

        hapticOptions.GetComponent<UIControllerBow>().frictionValue = x;

        hapticOptions.GetComponent<UIControllerBow>().friction.text = "" + x;

        Debug.Log("According to planet: " + hapticOptions.GetComponent<UIControllerBow>().frictionValue);

        //We apply the tactile vibration
        if (hapticOptions.GetComponent<UIControllerBow>().tactile)
        {
            inputGO.GetComponent<OculusInputSpheres>().m_controller.SendHapticImpulse(0, x, 0.7f);
            inputGO.GetComponent<OculusInputSpheres>().m_controller2.SendHapticImpulse(0, x, 0.7f);
        }

        //We apply the visual vibration
        if (hapticOptions.GetComponent<UIControllerBow>().visual)
        {
            //We get a random X, Y and Z
            float X = UnityEngine.Random.Range(-0.03f, 0.03f) * (hapticOptions.GetComponent<UIControllerBow>().frictionValue);
            float Y = UnityEngine.Random.Range(-0.03f, 0.03f) * (hapticOptions.GetComponent<UIControllerBow>().frictionValue);
            float Z = UnityEngine.Random.Range(-0.03f, 0.03f) * (hapticOptions.GetComponent<UIControllerBow>().frictionValue);

            mesh.transform.localPosition = new Vector3(X, Y, Z);
        }
```

```
        //We apply the delay
        if (hapticOptions.GetComponent<UIControllerBow>().delay)
        {
            //First we get the position of the hands
            thePositionShouldBe1 = leftHand.transform.position;
            thePositionShouldBe2 = rightHand.transform.position;

            //Now we get the vector between them
            Vector3 leftDistanceVector = thePositionShouldBe1 - formerPositionOfTheHand1;
            Vector3 rightDistanceVector = thePositionShouldBe2 - formerPositionOfTheHand2;

            //The distance of the vector
            float differenceDistance = Vector3.Distance(thePositionShouldBe1, formerPositionOfTheHand1);

            //We get the new point
            positionOfTheHand1 = formerPositionOfTheHand1 + new Vector3(leftDistanceVector.x/(1+(hapticOptions.GetComponent<UIControllerBow>().delayAmount)*5), leftDistanceVector.y/ (1 + (hapticOptions.GetComponent<UIControllerBow>().delayAmount) * 5), leftDistanceVector.z/ (1 + (hapticOptions.GetComponent<UIControllerBow>().delayAmount) * 5));
            positionOfTheHand2 = formerPositionOfTheHand2 + new Vector3(rightDistanceVector.x / (1 + (hapticOptions.GetComponent<UIControllerBow>().delayAmount) * 5), rightDistanceVector.y / (1 + (hapticOptions.GetComponent<UIControllerBow>().delayAmount) * 5), rightDistanceVector.z / (1 + (hapticOptions.GetComponent<UIControllerBow>().delayAmount) * 5));

            //We apply the changes
            formerPositionOfTheHand1 = positionOfTheHand1;
            leftHandModel.transform.position = positionOfTheHand1;

            formerPositionOfTheHand2 = positionOfTheHand2;
            rightHandModel.transform.position = positionOfTheHand2;
```



```
Archivo  Editar  Ver  Proyecto  Compilar  Depurar  Prueba  Analizar  Herramientas  Extensiones  Ventana  Ayuda  Buscar (Ctrl+Q)  Prueba_Configuracion
Debug  Any CPU  Asociar a Unity
culusInputSpheres.cs  ContainerOfTimes.cs  TrackedDeviceGraphicRaycaster.cs  ScriptDePuntuacion.cs  ImpactScript.cs  Arrow.cs  Bow.cs  Timer.cs  UIControllerBow.cs
Assembly-CSharp  planetScript  text

                //We apply them to the ball
                //To know the hand we can simply get the closest of the hands
                if(Vector3.Distance(leftHandModel.transform.position, gameObject.transform.position) > Vector3.Distance(rightHandModel.transform.position, gameObject.transform.position))
                {
                    gameObject.transform.position = positionOfTheHand2;
                }
                else
                {
                    gameObject.transform.position = positionOfTheHand1;
                }

            }
            else
            {
                //Debug.Log("RESETING POSITION AYIHBSCKN M;GHNBJKLHBFG");
                formerPositionOfTheHand1 = leftHand.transform.position;
                leftHandModel.transform.position = leftHand.transform.position;

                formerPositionOfTheHand2 = rightHand.transform.position;
                rightHandModel.transform.position = rightHand.transform.position;

                if (Vector3.Distance(leftHandModel.transform.position, gameObject.transform.position) > Vector3.Distance(rightHandModel.transform.position, gameObject.transform.position))
                {
                    gameObject.transform.position = rightHandModel.transform.position;
                }
                else
                {
                    gameObject.transform.position = leftHandModel.transform.position;
                }
            }

        }
```



```
            }

        }
        else
        {
            formerPositionOfTheHand1 = leftHand.transform.position;
            formerPositionOfTheHand2 = rightHand.transform.position;

            if (participe)
            {
                participe = false;
                leftHandModel.transform.position = leftHand.transform.position;
                rightHandModel.transform.position = rightHand.transform.position;

                if (Vector3.Distance(leftHandModel.transform.position, gameObject.transform.position) > Vector3.Distance(rightHandModel.transform.position, gameObject.transform.position))
                {
                    gameObject.transform.position = rightHandModel.transform.position;
                }
                else
                {
                    gameObject.transform.position = leftHandModel.transform.position;
                }

            }
        }


    }

    0 referencias
    public void setToGrabbed()
    {
        ovrGrabbable = true;
    }

    0 referencias
    public void setToUngrabbed()
    {
        ovrGrabbable = false;
        hapticOptions.GetComponent<UIControllerBow>().frictionValue = 0;
        hapticOptions.GetComponent<UIControllerBow>().friction.text = "" + 0;
    }
}
```

Ahora procederé a explicar este código. En primer lugar cambiamos a **false la variable "ovrGrabbable"** y obtenemos las referencias para las variables "**hapticOptions**", "**inputGO**", para las **manos virtuales** y para los **modelos de las manos** en la función **Start()**.

```
// Start is called before the first frame update
0 referencias
void Start()
{
    ovrGrabbable = false;
    hapticOptions = GameObject.Find("HapticOptions");
    inputGO = GameObject.Find("Input");

    leftHand = GameObject.Find("VR Rig").transform.FindChild("Camera Offset").FindChild("Left Hand").gameObject;
    rightHand = GameObject.Find("VR Rig").transform.FindChild("Camera Offset").FindChild("Right Hand").gameObject;

    leftHandModel = leftHand.transform.FindChild("Left Hand Model").gameObject;
    rightHandModel = rightHand.transform.FindChild("Right Hand Model").gameObject;
}
```

Ahora, en la función **Update()** vamos a **cambiar el color** del planeta asociado dependiendo de si este planeta ha sido agarrado, será rojo en caso de estar agarrado y blanco en el caso contrario.

```
//Conseguimos su asociado
GameObject planet = GameObject.Find("planet" + associatedNumber.ToString());

if (ovrGrabbable)
{
    //Le cambiamos el color
    planet.GetComponent<planetScript>().mesh.GetComponent<Renderer>().material.color = Color.red;
}
else
{
    //Le quitamos el color
    planet.GetComponent<planetScript>().mesh.GetComponent<Renderer>().material.color = Color.white;
}
```

Para cambiar la variable que indica si el planeta está agarrado o no poseemos dos funciones "**setToGrabbed()**" y "**setToUngrabbed()**". Las cuales veremos cómo se modifican tras analizar este script.

```
0 referencias
public void setToGrabbed()
{
    ovrGrabbable = true;
}

0 referencias
public void setToUngrabbed()
{
    ovrGrabbable = false;
    hapticOptions.GetComponent<UIControllerBow>().frictionValue = 0;
    hapticOptions.GetComponent<UIControllerBow>().friction.text = "" + 0;
}
```

En caso de que tanto este planeta como su asociado estén siendo agarrados procederemos a **calcular la distancia entre ambos** y **modificar el texto** del panel de pseudo-hápticos en consecuencia.

```
if (ovrGrabbable && planet.GetComponent<planetScript>().ovrGrabbable)
{
    participe = true;

    formerDistance = x;

    dist = Vector3.Distance(gameObject.transform.position, planet.transform.position);

    x = dist / 2;

    x = (float)Math.Round(x, 2);

    hapticOptions.GetComponent<UIControllerBow>().frictionValue = x;

    hapticOptions.GetComponent<UIControllerBow>().friction.text = "" + x;

    Debug.Log("According to planet: " + hapticOptions.GetComponent<UIControllerBow>().frictionValue);
```

Después se realizan las funciones pseudo-hápticas, las cuales se explicarán en un apartado aparte al igual que hicimos con la aplicación anterior. Por último, en caso de que los **dos planetas no hayan sido agarrados retornaremos las manos a sus posiciones habituales**.

.

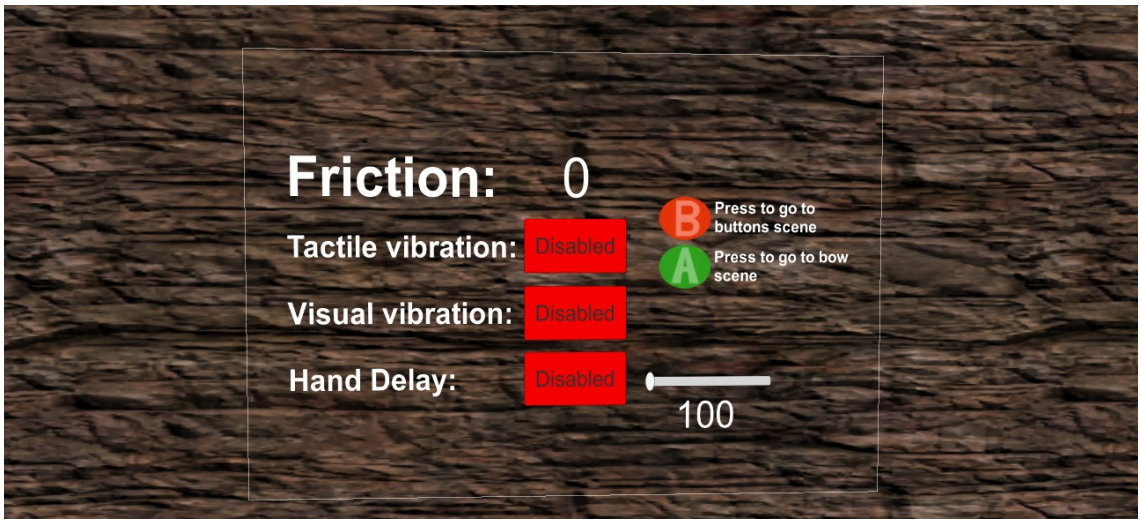
```
}
else
{
    formerPositionOfTheHand1 = leftHand.transform.position;
    formerPositionOfTheHand2 = rightHand.transform.position;

    if (participe)
    {
        participe = false;
        leftHandModel.transform.position = leftHand.transform.position;
        rightHandModel.transform.position = rightHand.transform.position;

        if (Vector3.Distance(leftHandModel.transform.position, gameObject.transform.position) > Vector3.Distance(rightHandModel.transform.position, gameObject.transform.position))
        {
            gameObject.transform.position = rightHandModel.transform.position;
        }
        else
        {
            gameObject.transform.position = leftHandModel.transform.position;
        }

    }
}
```

Por último, voy a explicar cómo se **activan la función** "**setToGrabbed()**" y "**setToUngrabbed()**". Para ello vamos al "**prefab**" del planeta y hacemos clic en "**Interactable Events**", con lo cual se abrirá una pestaña con distintos eventos. En el evento "**On Select Enter**" añadiremos la función "**setToGrabbed()**", de tal forma que se ejecutará al **agarrar el planeta** y en el evento "**On Select Exit**" añadimos "**setToUngrabbed()**" para que al soltar el planeta sea ejecutado.

El script "**OculusInputSpheres.cs**" es idéntico al utilizado para la aplicación del arco tras haberle quitado todo el código relacionado al propio arco.

Parte 3.5: Interfaces de usuario

En esta aplicación se utiliza únicamente el panel de pseudo-hápticos ya analizado en la aplicación del arco en realidad virtual. La única diferencia es que los **textos que indican cuales son la escena siguiente y anterior** han sido modificados para ajustarse a la nueva realidad.



Parte 3.6: Pseudo-hápticos

Al igual que en la aplicación del "Arco VR" tenemos tres formas de generar pseudo-hápticos. En este caso el código de los pseudo-hápticos se ejecuta en el script "**planetScript.cs**".

En el caso de la vibración, tanto **táctil** como **visual** el código es idéntico al utilizado en la aplicación anterior, salvo que ahora la vibración se aplica sobre las **dos manos**.

```
//We apply the tactile vibration
if (hapticOptions.GetComponent<UIControllerBow>().tactile)
{
    inputGO.GetComponent<OculusInputSpheres>().m_controller.SendHapticImpulse(0, x, 0.7f);
    inputGO.GetComponent<OculusInputSpheres>().m_controller2.SendHapticImpulse(0, x, 0.7f);
}

//We apply the visual vibration
if (hapticOptions.GetComponent<UIControllerBow>().visual)
{
    //We get a random X, Y and Z
    float X = UnityEngine.Random.Range(-0.03f, 0.03f) * (hapticOptions.GetComponent<UIControllerBow>().frictionValue);
    float Y = UnityEngine.Random.Range(-0.03f, 0.03f) * (hapticOptions.GetComponent<UIControllerBow>().frictionValue);
    float Z = UnityEngine.Random.Range(-0.03f, 0.03f) * (hapticOptions.GetComponent<UIControllerBow>().frictionValue);

    mesh.transform.localPosition = new Vector3(X, Y, Z);
}
```

Sin embargo, en el caso del "**hand delay**" el código cambia más. En primer lugar vamos a obtener las posiciones de las **manos virtuales** y **reales**. Luego, obtenemos la **distancia** entre donde la posición de la mano en el ciclo anterior y donde la mano real se halla.

```
//We apply the delay
if (hapticOptions.GetComponent<UIControllerBow>().delay)
{
    //First we get the position of the hands
    thePositionShouldBe1 = leftHand.transform.position;
    thePositionShouldBe2 = rightHand.transform.position;

    //Now we get the vector between them
    Vector3 leftDistanceVector = thePositionShouldBe1 - formerPositionOfTheHand1;
    Vector3 rightDistanceVector = thePositionShouldBe2 - formerPositionOfTheHand2;

    //The distance of the vector
    float differenceDistance = Vector3.Distance(thePositionShouldBe1, formerPositionOfTheHand1);
```

La **nueva posición** de la mano virtual (a la cual el planeta está unido) se calculará como la **posición antigua más** un v**ector tridimensional en el que cada dimensión es la distancia entre la mano virtual del ciclo anterior y la mano real** para esta dimensión **dividido entre 1 más el "delay"** escogido en el panel de pseudo-hápticos **multiplicado por 5** (para hacerlo avanzar más lentamente).

```
//We get the new point
positionOfTheHand1 = formerPositionOfTheHand1 + new Vector3(leftDistanceVector.x/(1+(hapticOptions.GetComponent<UIControllerBow>().delayAmount)*5), leftDis
positionOfTheHand2 = formerPositionOfTheHand2 + new Vector3(rightDistanceVector.x / (1 + (hapticOptions.GetComponent<UIControllerBow>().delayAmount) * 5),
```

```
), leftDistanceVector.y/ (1 + (hapticOptions.GetComponent<UIControllerBow>().delayAmount) * 5), leftDistanceVector.z/ (1 + (hapticOptions.GetComponent<UIControllerBow>().delayAmount) * 5));
nt) * 5), rightDistanceVector.y / (1 + (hapticOptions.GetComponent<UIControllerBow>().delayAmount) * 5), rightDistanceVector.z / (1 + (hapticOptions.GetComponent<UIControllerBow>().delayAmount) * 5));
```

Con este nuevo valor **modificamos** la **posición** de la mano virtual y actualizamos la posición en el ciclo anterior como la actual.

```
//We apply the changes
formerPositionOfTheHand1 = positionOfTheHand1;
leftHandModel.transform.position = positionOfTheHand1;

formerPositionOfTheHand2 = positionOfTheHand2;
rightHandModel.transform.position = positionOfTheHand2;

//We apply them to the ball
```

Por último **reseteamos** las **posiciones** de las manos y hacemos que la posición de la mano en el ciclo anterior sea igual a la del actual en caso de que la función de "**delay**" no esté activa.

```
}
else
{
    //Debug.Log("RESETING POSITION AYIHBSCKN M;GHNBJKLHBFG");
    formerPositionOfTheHand1 = leftHand.transform.position;
    leftHandModel.transform.position = leftHand.transform.position;

    formerPositionOfTheHand2 = rightHand.transform.position;
    rightHandModel.transform.position = rightHand.transform.position;
```

Con todo esto ya tendríamos las mismas funcionalidades pseudo-hápticas que en la aplicación anterior. Ahora vamos a ver como se ha creado la aplicación del botón en realidad virtual cambiando que la tensión que teníamos en el arco ahora es la distancia entre los dos planetas agarrados.

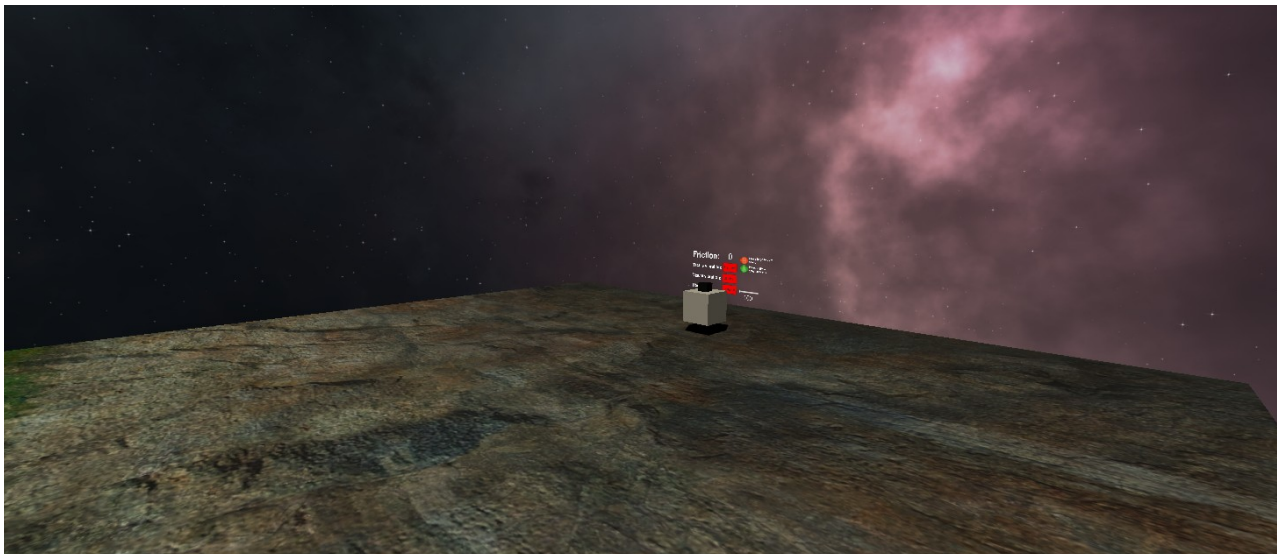
Parte 4: Botón VR

Parte 3.1: Objetivos

Esta es la más sencilla de entre las tres aplicaciones. Consiste únicamente en **una plataforma** en la que hay un **pulsador** sobre una mesa. En este caso en vez de la **tensión** del arco o la **distancia** entre los planetas, la variable principal es la **presión sobre el botón**, la cual estará normalizada **entre 0 y 1**.

Parte 3.2: El entorno

Esta aplicación posee el entorno más sencillo, consistente tan solo en **una base** creada mediante el reescalado de un cubo al cual se le ha añadido una textura rocosa. El **skybox** se ha creado de la misma manera que en la aplicación anterior.



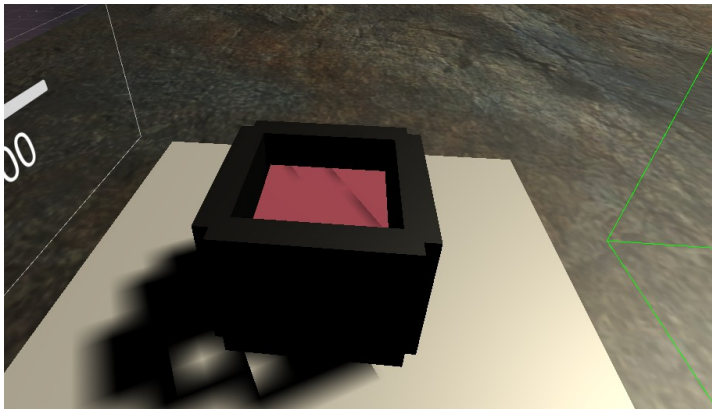
Parte 3.3: El jugador

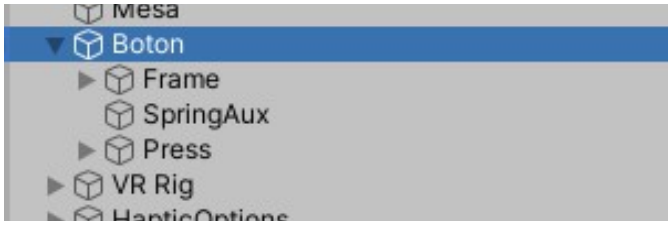
Al tener la misma funcionalidad que en la aplicación anterior, el objeto jugador es el mismo que el creado en la aplicación de los planetas pero **quitándole** la **posibilidad** de **agarrar objetos**, ya que en esta aplicación no hay objetos que agarrar.

Parte 3.4: El botón

El botón está ubicado sobre una **mesa creada** con un **cubo escalado**. Sobre esta mesa crearemos las partes del pulsador:

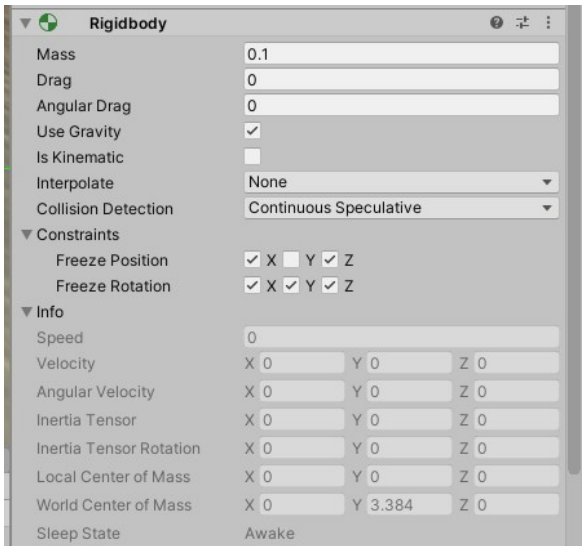
- El **marco del botón**, el cual consiste en 4 cubos que hacen de paredes del botón. (Frame)
- El **pulsador**, el cual consiste también en un cubo de color rojo insertado dentro del marco del botón. (Press)
- Por último tendremos también un **objeto invisible** que nos ayudará a **devolver el pulsador** a su posición original cuando dejemos de presionarlo.



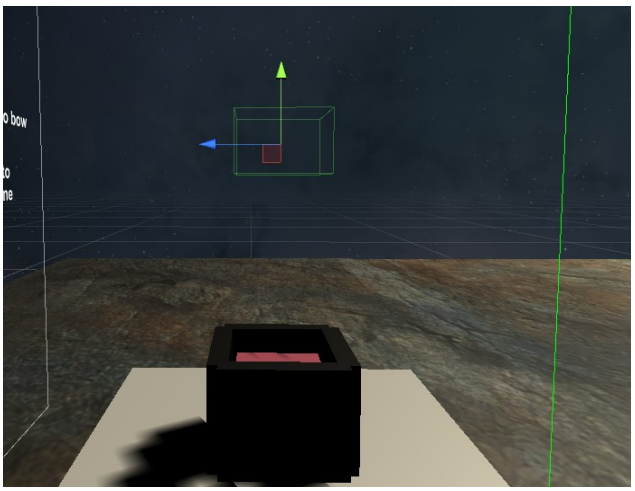


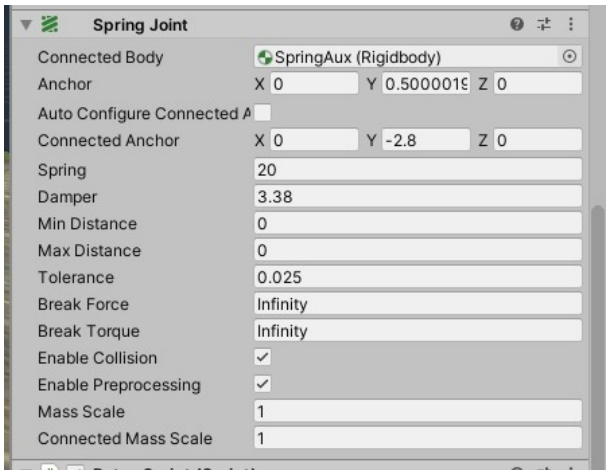
Crear las paredes del botón es muy sencillo, al igual que el pulsador, una vez tengamos ambos podremos empezar a desarrollar la lógica del botón.

Queremos que el pulsador se vea afectado por la física, por lo que le añadiremos un **Rigidbody**. Al mismo tiempo, tan solo queremos que pueda bajar y subir, por lo que editaremos la variable "**Constraint**" del **Rigidbody**, de tal forma que no pueda rotar y tan solo pueda desplazarse en el eje Y.

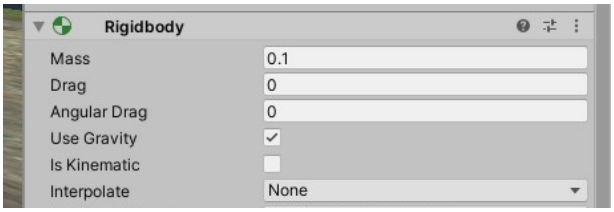


Ahora, para hacer que el pulsador vuelva a su posición original vamos a utilizar un tipo de unión, en nuestro caso una unión de tipo "muelle" (Spring joint), la cual como su nombre indica simula el efecto de un muelle que está unido a un objeto a cada lado. La primera unión estará en el propio pulsador mientras que en el otro extremo habrá un objeto invisible, intocable y que no esta afectado por la gravedad u otras fuerzas.

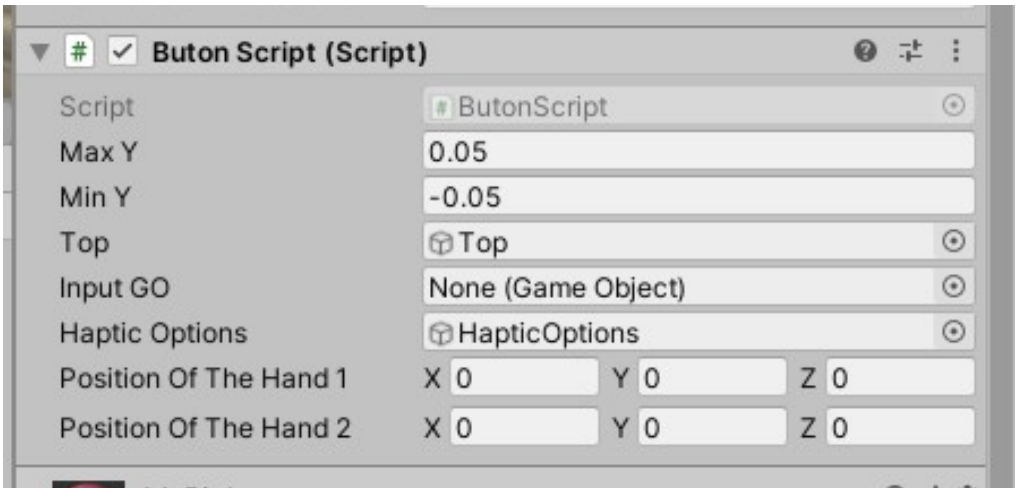




El objeto invisible (**SpringAux**) posee un **Rigidbody**, el cual arrastraremos hasta la variable "**Connected Body**" del "**Spring Joint**". Usaremos la configuración mostrada en la foto, ya que en diversas pruebas he visto que hace bien su cometido. También es importante tener en cuenta que este efecto no solo depende de la **fuerza** del **muelle**, sino también de la **masa del pulsador** al que está unido. Podemos cambiar la masa de un objeto en el componente **Rigidbody**, en mi caso a **0'1**.



Con esto ya tendríamos un **botón** que **reacciona** ante **colisiones**, pero nuestra **mano no puede realizar colisiones** ahora mismo. Para solucionar esto podríamos añadir una caja de colisiones a nuestro controlador. Por desgracia esta idea no es del todo buena, ya que las colisiones cuando uno de los dos objetos no es movido por una fuerza del juego, sino por una externa como es el movimiento de nuestro brazo **son impredecibles** funcionando solo en ciertas ocasiones. Para solucionar este problema tenemos un script dentro del pulsador llamado "**ButtonScript.cs**".



En primer lugar presentaré sus variables públicas:

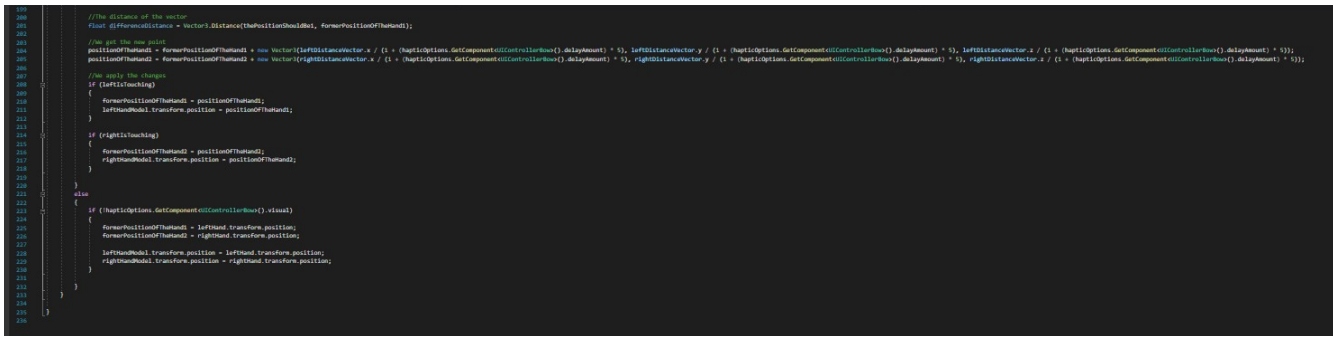
- "**Max Y**" y "**Min Y**": Variables que indican la **posición más alta** y **más baja** a la que se puede encontrar el pulsador..
- "**Top**": Variable que contiene una referencia a un objeto invisible que indica la **parte superior del pulsador**, con ella obtendremos la posición de la cara superior del mismo.
- "**Input GO**": Variable que contiene una referencia al objeto donde se halla el análogo a "**OculusInput.cs**" en la aplicación del arco o "**OculusInputSpheres.cs**" en la de los planetas. En esta aplicación se llama "**OculusInputButton.cs**" y es idéntica a "**OculusInputSpheres.cs**" salvo por pequeñas diferencias no significativas.
- "**Haptic Options**": Variable que contiene una referencia al **panel de control** de los **pseudo-hápticos**.
- "**Position Of The Hand 1**" y "**Position Of The Hand 2**": Variables que contienen las **posiciones de las manos izquierda y derecha**.

```
using System.Collections;
using System.Collections.Generic;
using System.Collections.Specialized;
using UnityEngine;
using UnityEngine.UI;

0 referencias
public class ButonScript : MonoBehaviour
{

    public float MaxY;
    public float MinY;

    public GameObject top;

    public GameObject inputGO;

    public GameObject hapticOptions;

    private GameObject leftHand;
    private GameObject rightHand;

    private GameObject leftHandModel;
    private GameObject rightHandModel;

    public Vector3 positionOfTheHand1 = new Vector3(0, 0, 0);
    private Vector3 formerPositionOfTheHand1 = new Vector3(0, 0, 0);
    private Vector3 thePositionShouldBe1 = new Vector3(0, 0, 0);

    public Vector3 positionOfTheHand2 = new Vector3(0, 0, 0);
    private Vector3 formerPositionOfTheHand2 = new Vector3(0, 0, 0);
    private Vector3 thePositionShouldBe2 = new Vector3(0, 0, 0);

    private int toucher = 0;

    private bool rightIsTouching = false;
    private bool leftIsTouching = false;

    private float theY = 0;

    // Start is called before the first frame update
    0 referencias
    void Start()
    {
        leftHand = GameObject.Find("VR Rig").transform.FindChild("Camera Offset").FindChild("Left Hand").gameObject;
        rightHand = GameObject.Find("VR Rig").transform.FindChild("Camera Offset").FindChild("Right Hand").gameObject;

        leftHandModel = leftHand.transform.FindChild("Left Hand Model").gameObject;
        rightHandModel = rightHand.transform.FindChild("Right Hand Model").gameObject;

        inputGO = GameObject.Find("Input");
    }
```

```
    // Update is called once per frame
    0 referencias
    void Update()
    {

        //Vamos a detectar que el dedo esta cerca o no
        //The condition is to be close in the Y
        if (Mathf.Abs(leftHandModel.transform.FindChild("ColliderOfButtonsLeft").position.y - top.transform.position.y) < 0.03)
        {
            //Now we have to see if the distance is close to the point
            if (Vector3.Distance(leftHandModel.transform.FindChild("ColliderOfButtonsLeft").position, top.transform.position) < 0.25)
            {
                theY = leftHandModel.transform.FindChild("ColliderOfButtonsLeft").position.y;
                leftIsTouching = true;
                rightIsTouching = false;
                gameObject.GetComponent<Rigidbody>().isKinematic = true;
            }
            else
            {
                leftIsTouching = false;
                gameObject.GetComponent<Rigidbody>().isKinematic = false;
            }

        }else if (Mathf.Abs(rightHandModel.transform.FindChild("ColliderOfButtonsRight").position.y - top.transform.position.y) < 0.03)
        {
            leftIsTouching = false;
            //Now we have to see if the distance is close to the point
            if (Vector3.Distance(rightHandModel.transform.FindChild("ColliderOfButtonsRight").position, top.transform.position) < 0.25)
            {
                theY = rightHandModel.transform.FindChild("ColliderOfButtonsRight").position.y;
                rightIsTouching = true;
                gameObject.GetComponent<Rigidbody>().isKinematic = true;
            }
            else
            {
                rightIsTouching = false;
                gameObject.GetComponent<Rigidbody>().isKinematic = false;
            }
        }
        else
        {
            rightIsTouching = false;
            leftIsTouching = false;
            gameObject.GetComponent<Rigidbody>().isKinematic = false;
        }


        if ((gameObject.transform.localPosition.y > MaxY) || ((theY > (3.559)) && leftIsTouching))
        {
            gameObject.transform.localPosition = new Vector3(0, MaxY, 0);
        }
```

```
        if ((gameObject.transform.localPosition.y > MaxY) || ((theY > (3.559)) && leftIsTouching))
        {
            gameObject.transform.localPosition = new Vector3(0, MaxY, 0);
        }
        else if((gameObject.transform.localPosition.y < MinY) || ((theY < (3.459)) && leftIsTouching))
        {
            gameObject.transform.localPosition = new Vector3(0, MinY, 0);
        }
        else
        {
            if (leftIsTouching)
            {
                top.transform.position = new Vector3(top.transform.position.x, theY, top.transform.position.z);
                gameObject.transform.position = new Vector3(gameObject.transform.position.x, theY - 0.075f, gameObject.transform.position.z);
            }

        }

        if ((gameObject.transform.localPosition.y > MaxY) || ((theY > (3.559)) && rightIsTouching))
        {
            gameObject.transform.localPosition = new Vector3(0, MaxY, 0);
        }
        else if ((gameObject.transform.localPosition.y < MinY) || ((theY < (3.459)) && rightIsTouching))
        {
            gameObject.transform.localPosition = new Vector3(0, MinY, 0);
        }
        else
        {
            if (rightIsTouching)
            {
                top.transform.position = new Vector3(top.transform.position.x, theY, top.transform.position.z);
                gameObject.transform.position = new Vector3(gameObject.transform.position.x, theY - 0.075f, gameObject.transform.position.z);
            }

        }


        //Debug.Log(gameObject.transform.localPosition.y);
        // gameObject.transform.localPosition.y;
        hapticOptions.GetComponent<UIControllerBow>().frictionValue = (MaxY - gameObject.transform.localPosition.y) / (MaxY - MinY);

        //Now we make the tactile vibration
        if (hapticOptions.GetComponent<UIControllerBow>().tactile && hapticOptions.GetComponent<UIControllerBow>().frictionValue > 0.25)
        {

            if (rightIsTouching)
            {
                inputGO.GetComponent<OculusInputButton>().m_controller.SendHapticImpulse(0, hapticOptions.GetComponent<UIControllerBow>().frictionValue, 0.7f);
            }

```

```
            if (leftIsTouching)
            {
                inputGO.GetComponent<OculusInputButton>().m_controller2.SendHapticImpulse(0, hapticOptions.GetComponent<UIControllerBow>().frictionValue, 0.7f);
            }

        }

        //Now we make the visual vibration
        if (hapticOptions.GetComponent<UIControllerBow>().visual)
        {
            if (rightIsTouching)
            {
                float X = UnityEngine.Random.Range(-0.005f, 0.005f) * (hapticOptions.GetComponent<UIControllerBow>().frictionValue);
                float Y = UnityEngine.Random.Range(-0.005f, 0.005f) * (hapticOptions.GetComponent<UIControllerBow>().frictionValue);
                float Z = UnityEngine.Random.Range(-0.005f, 0.005f) * (hapticOptions.GetComponent<UIControllerBow>().frictionValue);

                rightHandModel.transform.localPosition = new Vector3(X, Y, Z);
            }

            if (leftIsTouching)
            {
                float X = UnityEngine.Random.Range(-0.005f, 0.005f) * (hapticOptions.GetComponent<UIControllerBow>().frictionValue);
                float Y = UnityEngine.Random.Range(-0.005f, 0.005f) * (hapticOptions.GetComponent<UIControllerBow>().frictionValue);
                float Z = UnityEngine.Random.Range(-0.005f, 0.005f) * (hapticOptions.GetComponent<UIControllerBow>().frictionValue);

                leftHandModel.transform.localPosition = new Vector3(X, Y, Z);
            }
        }

        //We apply the delay
        if (hapticOptions.GetComponent<UIControllerBow>().delay)
        {

            if (!rightIsTouching && !leftIsTouching)
            {

                formerPositionOfTheHand1 = leftHand.transform.position;
                formerPositionOfTheHand2 = rightHand.transform.position;

                leftHandModel.transform.position = leftHand.transform.position;
                rightHandModel.transform.position = rightHand.transform.position;

            }

            //First we get the position of the hands
            thePositionShouldBe1 = leftHand.transform.position;
            thePositionShouldBe2 = rightHand.transform.position;

            //Now we get the vector between them
            Vector3 leftDistanceVector = thePositionShouldBe1 - formerPositionOfTheHand1;
            Vector3 rightDistanceVector = thePositionShouldBe2 - formerPositionOfTheHand2;

```



En primer lugar obtenemos las **referencias** de las **manos virtuales**, los modelos de las manos y el "**input**" en la función **Start()**.

```
// Start is called before the first frame update
0 referencias
void Start()
{
    leftHand = GameObject.Find("VR Rig").transform.FindChild("Camera Offset").FindChild("Left Hand").gameObject;
    rightHand = GameObject.Find("VR Rig").transform.FindChild("Camera Offset").FindChild("Right Hand").gameObject;

    leftHandModel = leftHand.transform.FindChild("Left Hand Model").gameObject;
    rightHandModel = rightHand.transform.FindChild("Right Hand Model").gameObject;

    inputGO = GameObject.Find("Input");
}
```

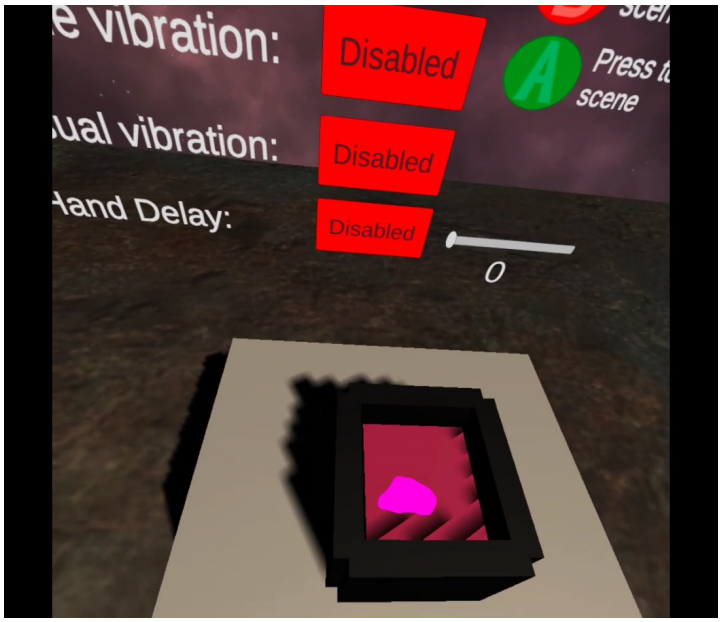
Después, vamos a calcular la **distancia** de las **manos** al **botón** en el bloque **Update()**. De este modo, si la mano se encuentra a una distancia **menor a un umbral** tanto a lo alto como en las tres dimensiones indicaremos que una de las manos está **tocando** el **pulsador**. En caso de que ninguna de las dos manos estén tocando el pulsador lo indicaremos.

```
// Update is called once per frame
0 referencias
void Update()
{

    //Vamos a detectar que el dedo esta cerca o no
    //The condition is to be close in the Y
    if (Mathf.Abs(leftHandModel.transform.FindChild("ColliderOfButtonsLeft").position.y - top.transform.position.y) < 0.03)
    {
        //Now we have to see if the distance is close to the point
        if (Vector3.Distance(leftHandModel.transform.FindChild("ColliderOfButtonsLeft").position, top.transform.position) < 0.25)
        {
            theY = leftHandModel.transform.FindChild("ColliderOfButtonsLeft").position.y;
            leftIsTouching = true;
            rightIsTouching = false;
            gameObject.GetComponent<Rigidbody>().isKinematic = true;
        }
        else
        {
            leftIsTouching = false;
            gameObject.GetComponent<Rigidbody>().isKinematic = false;
        }

    }else if (Mathf.Abs(rightHandModel.transform.FindChild("ColliderOfButtonsRight").position.y - top.transform.position.y) < 0.03)
    {
        leftIsTouching = false;
        //Now we have to see if the distance is close to the point
        if (Vector3.Distance(rightHandModel.transform.FindChild("ColliderOfButtonsRight").position, top.transform.position) < 0.25)
        {
            theY = rightHandModel.transform.FindChild("ColliderOfButtonsRight").position.y;
            rightIsTouching = true;
            gameObject.GetComponent<Rigidbody>().isKinematic = true;
        }
        else
        {
            rightIsTouching = false;
            gameObject.GetComponent<Rigidbody>().isKinematic = false;
        }
    }
    else
    {
        rightIsTouching = false;
        leftIsTouching = false;
        gameObject.GetComponent<Rigidbody>().isKinematic = false;
    }
```

Ahora vamos a actuar en consecuencia. En el caso de que el **pulsador** esté por **encima** o por **debajo** de sus márgenes vamos a **forzar su posición** al margen correspondiente aunque esté la mano tocándolo. En caso de que no se haya salido de sus posiciones máxima o mínima la **posición del pulsador** será la **misma que la de la mano**.

```
if ((gameObject.transform.localPosition.y > MaxY) || ((theY > (3.559)) && leftIsTouching))
{
    gameObject.transform.localPosition = new Vector3(0, MaxY, 0);
}
else if((gameObject.transform.localPosition.y < MinY) || ((theY < (3.459)) && leftIsTouching))
{
    gameObject.transform.localPosition = new Vector3(0, MinY, 0);
}
else
{
    if (leftIsTouching)
    {
        top.transform.position = new Vector3(top.transform.position.x, theY, top.transform.position.z);
        gameObject.transform.position = new Vector3(gameObject.transform.position.x, theY - 0.075f, gameObject.transform.position.z);
    }

}

if ((gameObject.transform.localPosition.y > MaxY) || ((theY > (3.559)) && rightIsTouching))
{
    gameObject.transform.localPosition = new Vector3(0, MaxY, 0);
}
else if ((gameObject.transform.localPosition.y < MinY) || ((theY < (3.459)) && rightIsTouching))
{
    gameObject.transform.localPosition = new Vector3(0, MinY, 0);
}
else
{
    if (rightIsTouching)
    {
        top.transform.position = new Vector3(top.transform.position.x, theY, top.transform.position.z);
        gameObject.transform.position = new Vector3(gameObject.transform.position.x, theY - 0.075f, gameObject.transform.position.z);
    }

}
```

Por último, calcularemos la presión del botón como la resta entre el punto máximo posible en la coordenada **Y** (“**MaxY**”) menos su posición, todo ello **normalizado entre 0 y 1** dividiéndolo entre “**MaxY**” **-** “**MinY**”.



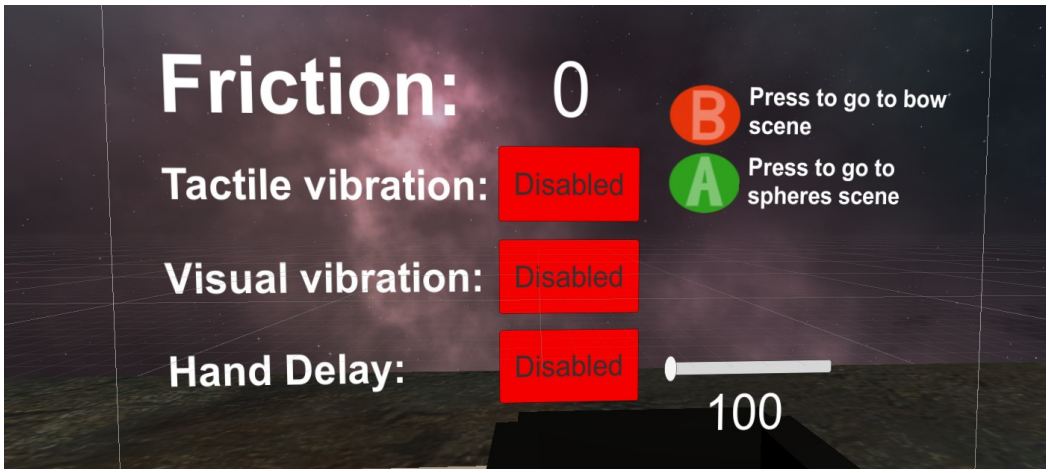
Botón ligeramente presionado.

Botón sin presionar

Con esto ya tendríamos un botón con el que interactuar, ahora vamos a ver de qué interfaces dispone.

Parte 4.5: Interfaces de usuario

Las interfaces utilizadas son las mismas que en la aplicación de los planetas, tan solo cambia el **texto de la anterior y siguiente escena** a la que pasar.



Parte 4.6: Pseudo-hápticos

Al igual que en las otras dos aplicaciones, se usarán las **tres funciones pseudo-hápticas** ya mencionadas, tan solo cambia el hecho de que en vez de utilizar la **tensión** del arco o la **distancia** entre planetas vamos a utilizar la **presión** del pulsador.

Para la **vibración táctil** y la **vibración visual** es igual que con la aplicación de los planetas, solo que en este caso debemos de distinguir qué mano está tocando el pulsador.

```
//Now we make the tactile vibration
if (hapticOptions.GetComponent<UIControllerBow>().tactile && hapticOptions.GetComponent<UIControllerBow>().frictionValue > 0.25)
{

    if (rightIsTouching)
    {
        inputGO.GetComponent<OculusInputButton>().m_controller.SendHapticImpulse(0, hapticOptions.GetComponent<UIControllerBow>().frictionValue, 0.7f);
    }

    if (leftIsTouching)
    {
        inputGO.GetComponent<OculusInputButton>().m_controller2.SendHapticImpulse(0, hapticOptions.GetComponent<UIControllerBow>().frictionValue, 0.7f);
    }

}

//Now we make the visual vibration
if (hapticOptions.GetComponent<UIControllerBow>().visual)
{
    if (rightIsTouching)
    {
        float X = UnityEngine.Random.Range(-0.005f, 0.005f) * (hapticOptions.GetComponent<UIControllerBow>().frictionValue);
        float Y = UnityEngine.Random.Range(-0.005f, 0.005f) * (hapticOptions.GetComponent<UIControllerBow>().frictionValue);
        float Z = UnityEngine.Random.Range(-0.005f, 0.005f) * (hapticOptions.GetComponent<UIControllerBow>().frictionValue);

        rightHandModel.transform.localPosition = new Vector3(X, Y, Z);
    }

    if (leftIsTouching)
    {
        float X = UnityEngine.Random.Range(-0.005f, 0.005f) * (hapticOptions.GetComponent<UIControllerBow>().frictionValue);
        float Y = UnityEngine.Random.Range(-0.005f, 0.005f) * (hapticOptions.GetComponent<UIControllerBow>().frictionValue);
        float Z = UnityEngine.Random.Range(-0.005f, 0.005f) * (hapticOptions.GetComponent<UIControllerBow>().frictionValue);

        leftHandModel.transform.localPosition = new Vector3(X, Y, Z);
    }
}
```

El código para obtener el "**delay**" de las manos también es idéntico al de la aplicación anterior, pero de nuevo, debemos de tener en cuenta **cuál de las manos es la que está haciendo contacto con el pulsador**.

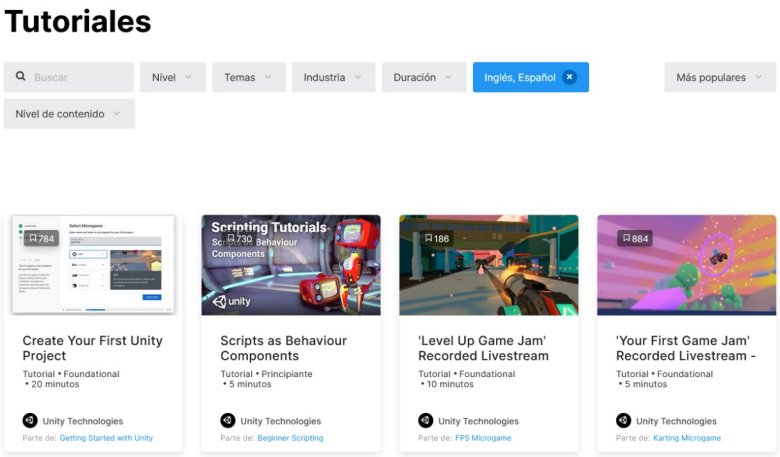
Con todo esto ya tendríamos las tres aplicaciones creadas con todas las funciones clásicas y pseudo-hápticas. Ahora haré un pequeño resumen de una serie de **consejos** que creo que podrían ser **útiles** para aquellos que deseen desarrollar aplicaciones enfocadas a la realidad virtual en Unity3D.

Parte 5: Desarrollo de aplicaciones VR en Unity3D

Parte 5.1: Desarrollo clásico en Unity3D

El desarrollo de aplicaciones de realidad virtual en Unity3D **no dista** demasiado del desarrollo de **videojuegos convencionales** en esta plataforma. Es importante familiarizarse con el **lenguaje C#** y las diversas funciones que ofrece este motor de videojuegos, pudiendo incluso complementar tus proyectos con objetos obtenidos en aplicaciones de **modelado 3D** como **Blender** o programas de edición de audio como podría ser **Audacity**.

Si deseas aprender a usar Unity3D existen **multitud de tutoriales** en internet tanto escritos como en forma de video, además de haber una **sección** en la página web de Unity3D enfocada específicamente al aprendizaje (7). En esta web existen infinidad de **tutoriales** paso a paso para gente de **todos los niveles**. Además, para los que prefieran aprender mediante videos hay, además de multitud de canales individuales que enseñan **Unity3D**, un canal oficial en **Youtube** de "Unity3D" en el que suben tutoriales semanales de corta duración, enfocados a temas concretos (8). Otros canales de Youtube muy interesantes para aprender Unity son "**Brackeys**" (9), "**Academia de Videojuegos**" (10), "**Game Grind**" (11) o "**Quill18Creates**" (12).



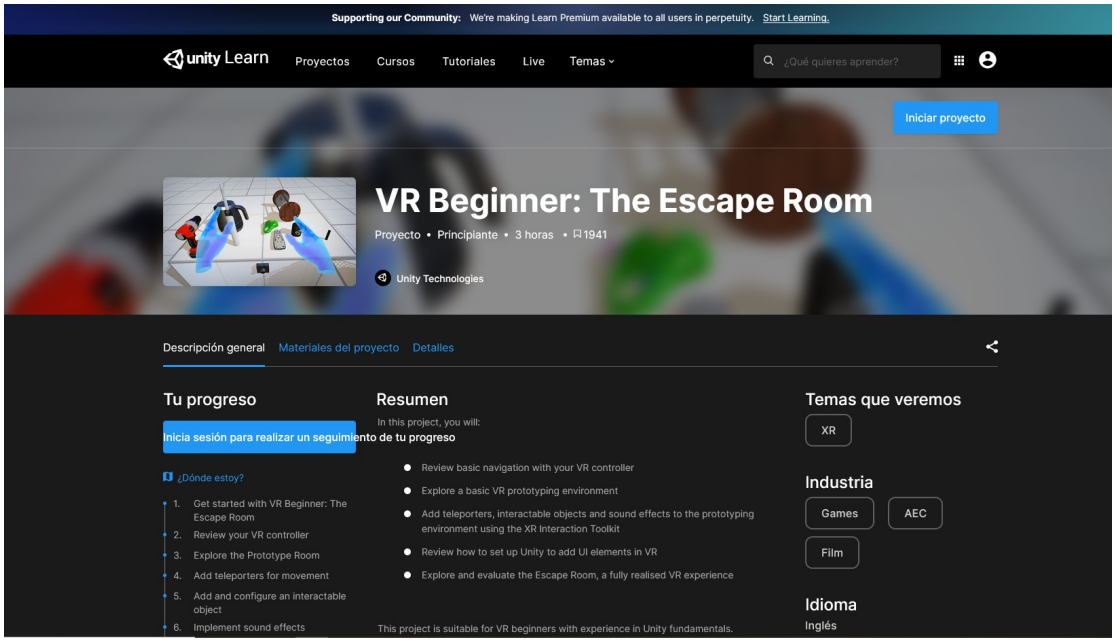
En caso de no saber cómo funciona exactamente una función en Unity existen diversos **foros**, incluido uno en la página de Unity, además de la propia **documentación**, la cual está muy bien detallada (13).

Por último, es importante **experimentar**, crear distintas escenas en Unity tratando de **explorar todas las funciones** del programa y sus **teclas de acceso rápido**, las cuales a la larga ayudan a ahorrar mucho tiempo.

Parte 5.2: Desarrollo VR en Unity3D

Si sabes desarrollar **aplicaciones convencionales** en Unity3D sabes desarrollar **aplicaciones VR** en Unity3D, o al menos, estas muy cerca. Recientemente se estandarizaron las librerías de desarrollo de realidad virtual, por lo que su popularidad, y, por tanto, la **documentación** y **tutoriales** al respecto irán en **aumento**.

Dentro de la página de **tutoriales** de Unity3D existe uno específicamente **pensado para desarrolladores** de aplicaciones en **realidad virtual** llamado "**VR Beginner: The Escape Room**". Es un tutorial realmente sencillo y explicativo, donde se enseña como **preparar Unity3D** para el desarrollo de aplicaciones VR, cómo utilizar los **controladores**, **desplazarte**…



En mi caso empecé con este proyecto cuando **aún no existían muchos** de estos **tutoriales**, por lo que aprendí principalmente mediante tutoriales en Youtube. Algunos muy recomendables para aprender especificamente realidad virtual en Unity son "**Valem**" (14) o "**Dilmer Valecillos**" (15).

Existe también una sección dentro de la **documentación** de **Unity3D** enfocada al desarrollo de aplicaciones en **realidad virtual**, **aumentada** y **mixta** (englobada en las siglas **XR**). (16) .

Parte 5.3: Motion sickness

Dedicaré un último apartado en este capítulo al tema de la "**motion sickness**". Se trata de un concepto nuevo con el que lidiar para los desarrolladores de videojuegos y/o aplicaciones que quieren dar el salto a la **realidad virtual**. En aplicaciones clásicas, existen una serie de patrones para **evitar** que las **interfaces** del usuario

sean **molestas** o poco estéticas para un usuario, y lo mismo existe en realidad virtual, solo que sus efectos afectan de forma más pronunciada al usuario.

La "**motion sickness**" es un malestar que todos hemos sufrido en algún momento de nuestras vidas, ya sea viajando en un vehículo o montándonos en una atracción de feria. Este efecto se produce porque **tu cerebro se halla confundido ante el hecho de que estas desplazándote sin ser tú el artífice de ese movimiento**. En aplicaciones de realidad virtual existen una serie de pautas que ayudan a **evitar este malestar** en la medida de lo posible.

Algunos consejos son:

- Crear **rampas** en lugar de **escaleras**.
- **Aumentar** los **fotogramas** por segundo de la aplicación a **90 por segundo** aproximadamente. Unity3D permite realizar esta función en "**edit -> project settings**".
- **Evitar** las **rotaciones bruscas** del jugador que no sean producidas por la propia cabeza del usuario, para ello se puede **rotar al personaje de forma discreta** (de 30 en 30 grados por ejemplo).
- Utilizar si es posible **formas de movimiento** como el **teletransporte**, o creando aplicaciones que **no requieran al usuario desplazarse**.
- Por último, es importante saber que el usuario normalmente **se irá adaptando paulatinamente** al dispositivo y acabará por no sentir o **sentir en menor medida este efecto**.

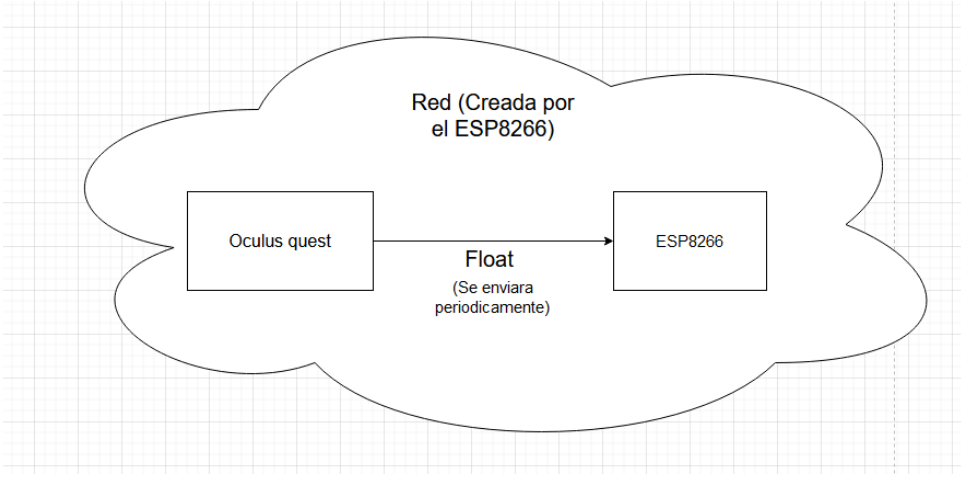
CAPÍTULO 2: Conectividad

Parte 1: Introducción

Ahora que ya hemos logrado crear una serie de aplicaciones con funciones hápticas y pseudo-hápticas, vamos a intentar crear una **funcionalidad háptica** para estas aplicaciones. Para ello deberemos establecer una **comunicación** entre el **software** y el **hardware** que vamos a utilizar.

Como **intermediario** entre el **software** y el **hardware** que realiza la función háptica vamos a utilizar un **componente hardware** conocido como **ESP8266**, un pequeño **microcontrolador** que permite establecer comunicaciones e incluso crear tu propio "**access point**", como vamos a hacer en esta aplicación.

El objetivo es sencillo de alcanzar, tan solo debemos ser capaces de enviar un mensaje conteniendo un número de tipo "**float**" de las **Oculus Quest** a este microprocesador. Este **float** contendrá la **tensión** del **arco** en el caso de la aplicación del arco, la **distancia** entre los dos **planetas** agarrados en el caso de la aplicación de los planetas o la **presión** sobre el **pulsador** en el caso de la aplicación del botón.



Parte 2: Conexión Oculus - ESP8266

En primer lugar vamos a ver cómo haremos para realizar la **comunicación** entre las **gafas de realidad virtual** y el **microcontrolador**. Para ello usaremos de nuevo los scripts de Unity3D, en este caso podemos aprovechar para incluir el código de la comunicación en uno ya creado que tenemos, como es "**UIControllerBow.cs**".

Si lo pensamos detenidamente lo que **queremos conseguir** es:

-**Conectarnos al dispositivo** al que hemos de enviar los datos, el cual es el mismo que nos provee la red.

-Cada cierto tiempo **enviar** la **variable** deseada.

-**Recibir** un **ACK** indicando que nuestro mensaje ha sido recibido (**opcional**, solo se usó para las primeras pruebas).

El **protocolo** utilizado finalmente ha sido **UDP**, ya que es una aplicación **tolerante a pérdidas**, además de tratarse de un protocolo ligeramente más simple que **TCP**. De igual manera la aplicación hubiera funcionado perfectamente utilizando otros protocolos.

El código para enviar los mensajes es el siguiente.

```
46      // Update is called once per frame
        0 referencias
47      void Update()
48      {
49          delayAmount = (int)sliderDelay.value;
50          //We change the friction
51          friction.text = "" + frictionValue;
52          delayAmountText.text = "" + delayAmount;
53
54          /*if (canSendPackage)
55          {
56              StartCoroutine(sendData(0.15f));
57          }*/
58
59          sendData();
60      }
61
        1 referencia
62      private void sendData()
63      {
64          //canSendPackage = false;
65          //yield return new WaitForSeconds(waitTime);
66
67          IPEndPoint ep = new IPEndPoint(IPAddress.Parse("192.168.4.1"), 20001);
68          Socket sock = new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp);
69
70          try
71          {
72              sock.SendTo(Encoding.ASCII.GetBytes("" + frictionValue), ep);
73              Debug.Log("Information sent");
74          }
75          catch
76          {
77              Debug.Log("Information not sent");
78          }
79
80          canSendPackage = true;
81      }
```

```
1 referencia
void ReceiveData()
{
    while (true)
    {
        try
        {

            m_Client = new UdpClient(20000);
            m_Client.EnableBroadcast = true;
            while (true)
            {

                IPEndPoint hostIP = new IPEndPoint(IPAddress.Any, 0);
                byte[] data = m_Client.Receive(ref hostIP);
                string returnData = Encoding.ASCII.GetString(data);

                Debug.Log(returnData);
            }
        }
        catch (Exception e)
        {
            Debug.Log(e);

            OnApplicationQuit();
        }
    }

}

1 referencia
private void OnApplicationQuit()
{
    if (m_Thread != null)
    {
        m_Thread.Abort();
    }

    if (m_Client != null)
    {
        m_Client.Close();
    }
}
```

En primer lugar vamos a crear una función para **crear** y **enviar paquetes** llamada "**SendData()**", la cual incluiremos en la función **Update()** del script para que se ejecute constantemente.

```
// Update is called once per frame
0 referencias
void Update()
{
    delayAmount = (int)sliderDelay.value;
    //We change the friction
    friction.text = "" + frictionValue;
    delayAmountText.text = "" + delayAmount;

    /*if (canSendPackage)
    {
        StartCoroutine(sendData(0.15f));
    }*/

    sendData();
}

1 referencia
private void sendData()
{
    //canSendPackage = false;
    //yield return new WaitForSeconds(waitTime
```

En esta función lo primero que haremos es **crear** la **variable** de los datos del "**endPoint**" y **abrir** un **socket** para el tráfico de **datagramas**. Por suerte la IP del **endPoint** coincide con la del **access point**, la cual siempre es "**192.128.4.1**", además el puerto de escucha del **ESP8266** será el **20001**. Para abrir el **socket** indicaremos el protocolo a utilizar, en nuestro caso **UDP**.

```
IPEndPoint ep = new IPEndPoint(IPAddress.Parse("192.168.4.1"), 20001);
Socket sock = new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp);
```

Después, en un bloque de tipo **try-catch** intentaremos enviar un paquete conteniendo únicamente el valor del **float** que queremos enviar, el cual será codificado a "**ASCII**". En caso de que se **envíe correctamente** imprimimos por la consola un **mensaje satisfactorio**, en **caso contrario**, imprimiremos por la consola un **mensaje** indicando que ha **fallado**.

```
try
{
    sock.SendTo(Encoding.ASCII.GetBytes("" + frictionValue), ep);
    Debug.Log("Information sent");
}
catch
{
    Debug.Log("Information not sent");
}
```

Para estar a la escucha de una respuesta de nuestro destinatario, ejecutaremos en un **hilo aparte** la función "**ReceiveData()**". Para ello abriremos un hilo en la función **Start()**.

```
0 referencias
void Start()
{
    m_Thread = new Thread(new ThreadStart(ReceiveData));
    m_Thread.IsBackground = true;
    m_Thread.Start();
}
```

La función "**ReceiveData()**" consiste en un **bucle infinito** en el cual estaremos constantemente a la **escucha de paquetes** que entren por un puerto específico.

```
while (true)
{
    try
    {

        m_Client = new UdpClient(20000);
        m_Client.EnableBroadcast = true;
        while (true)
        {

            IPEndPoint hostIP = new IPEndPoint(IPAddress.Any, 0);
            byte[] data = m_Client.Receive(ref hostIP);
            string returnData = Encoding.ASCII.GetString(data);

            Debug.Log(returnData);
        }
    }
    catch (Exception e)
    {
        Debug.Log(e);

        OnApplicationQuit();
    }
}
```

Lo que hará este bucle es crear un **cliente UDP** en el **puerto 20000**, el cual se hallará constantemente a la **escucha** de mensajes procedentes de **cualquier IP**, cuando se obtenga un paquete, este será **reconvertido a string** e **impreso** en la **consola**.

```
m_Client = new UdpClient(20000);
m_Client.EnableBroadcast = true;
while (true)
{

    IPEndPoint hostIP = new IPEndPoint(IPAddress.Any, 0);
    byte[] data = m_Client.Receive(ref hostIP);
    string returnData = Encoding.ASCII.GetString(data);

    Debug.Log(returnData);
}
```

En caso de que **falle la ejecución** de este código, **imprimimos por consola el error** generado y **destruimos el hilo** que habíamos creado para esta función.

```
}
catch (Exception e)
{
    Debug.Log(e);

    OnApplicationQuit();
}
```

"**OnApplicationQuit()**" es una función predefinida en **Unity3D**, la cual se ejecuta antes de que la aplicación vaya a cerrarse, en ella vamos a destruir los hilos generados.

```
1 referencia
private void OnApplicationQuit()
{
    if (m_Thread != null)
    {
        m_Thread.Abort();
    }

    if (m_Client != null)
    {
        m_Client.Close();
    }
}
```

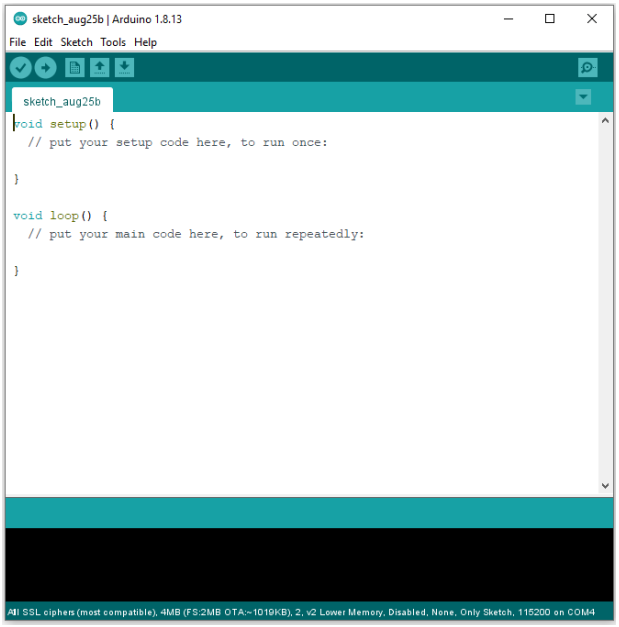
Con esto, nuestra aplicación ya podría **enviar la variable indicada a cualquier dispositivo cambiando la IP y el puerto**. Para experimentar si el código funcionaba correctamente, he descargado una **aplicación** para el **móvil** llamada "**UDP Sender/Receiver**", con la cual se pueden **enviar** y **recibir paquetes**, pudiendo elegir si estamos a la escucha de una respuesta o no, y cuanto tiempo la espera.

Parte 3: Introducción a arduino

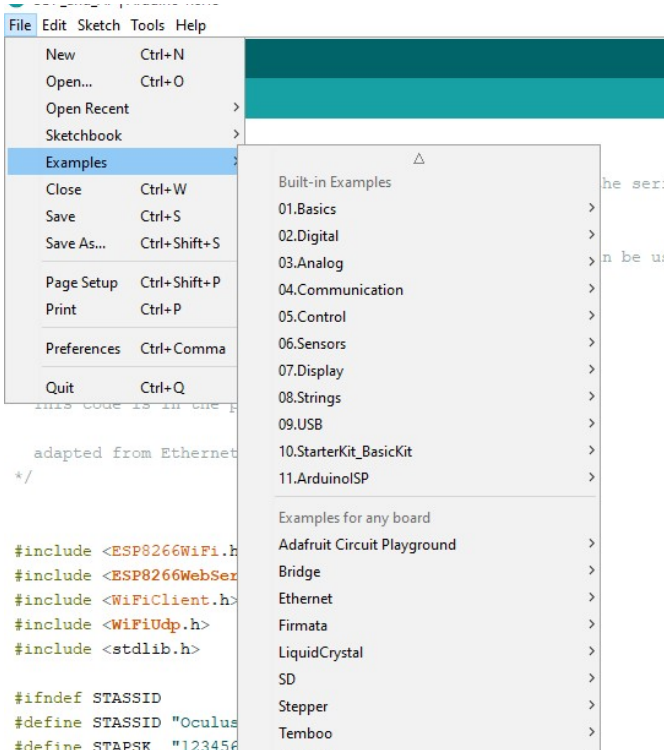
Antes de pasar a ver la comunicación "**Esp8266 - Oculus**" vamos a hacer un pequeño inciso para explorar esta nueva herramienta, con la cual configuraremos no solo la **conectividad**, sino también el **hardware** de este proyecto.

Se trata de una herramienta que **ejecuta hojas de código** conocidas como **Sketches**, las cuales están escritas en un **lenguaje inspirado en C++,** pero con funciones propias y otras diferencias.

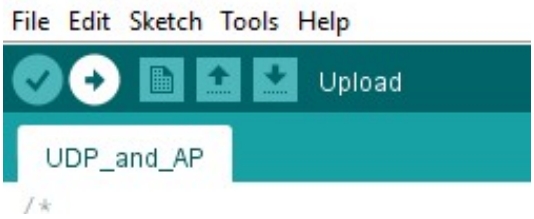
Para **crear un sketch** en arduino, tan solo hay que pulsar "**File -> New**". Esto nos generará automáticamente un **documento** con las funciones básicas de **Arduino** "**Setup()**" y "**Loop()**", que funcionan de igual manera que las funciones "**Start()**" y "**Update()**" de **Unity3D**.



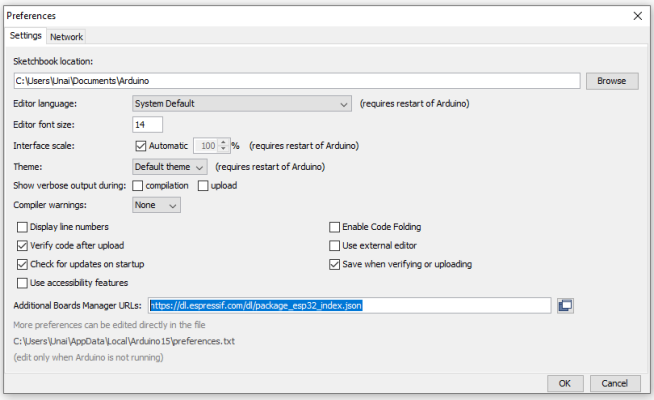
Además de crear un sketch nuevo, podemos **cargar y modificar sketches de ejemplo** que vienen con la propia IDE, estos ejemplo se encuentran en "**File -> Examples**", e incluyen ejemplos de casi todas las funciones de **Arduino**, desde encender un **led** hasta establecer conexiones **Wi-Fi**. Además de ejemplos propios de las librerías que descarguemos.

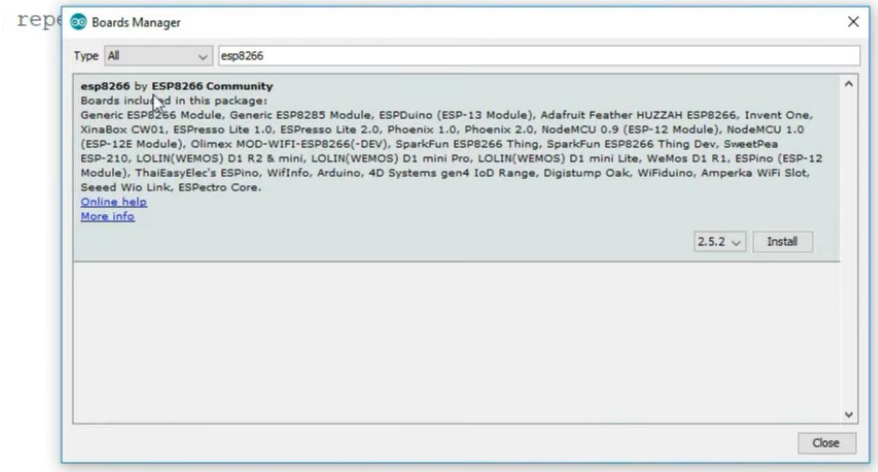


Para compilar e insertar el código en la placa haremos clic en la flecha que dice "**Upload**".



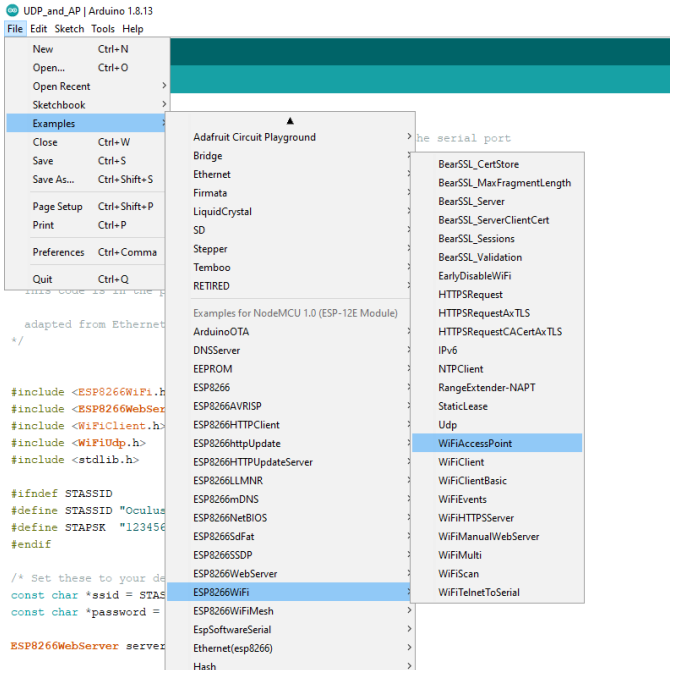
Sin embargo, si queremos que el ordenador detecte nuestra placa deberemos instalar la **librería ESP8266**, y, si fuera necesario un **driver**, para lo cual se pueden buscar diversos **tutoriales** en **internet**. Para instalar la librería **ESP8266** haremos clic en "**File -> Preferences**" y en la barra de "**Aditional Boards Manager URLs**" escribiremos "https://dl.espressif.com/dl/package_esp32_index.json". Por último, accederemos a "**tools -> Manage Libraries**" y buscaremos "**ESP8266**", haremos clic en **Install** y esperamos a que se instale.





Parte 4: Conexión ESP8266 - Oculus

Ahora que ya tenemos las nociones básicas de **Arduino**, vamos a crear el **sketch** en el cual se crea el **access point** y se escuchan los paquetes. Para ello mezclaremos dos ejemplos provistos por la librería **ESP8266**, "**UDP**" y "**WiFiAccessPoint**".



El código resultante es el siguiente.

```
#include <ESP8266WiFi.h>
#include <ESP8266WebServer.h>
#include <WiFiClient.h>
#include <WiFiUdp.h>
#include <stdlib.h>

#ifndef STASSID
#define STASSID "Oculus"
#define STAPSK  "123456789"
#endif

/* Set these to your desired credentials. */
const char *ssid = STASSID;
const char *password = STAPSK;

ESP8266WebServer server(20001);

unsigned int localPort = 20001;      // local port to listen on

// buffers for receiving and sending data
char packetBuffer[UDP_TX_PACKET_MAX_SIZE + 1]; //buffer to hold incoming packet,
char  ReplyBuffer[] = "acknowledged\r\n";       // a string to send back

WiFiUDP Udp;

void setup() {
  Serial.begin(115200);

  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(4, OUTPUT);

  Serial.print("Configuring access point...");
  /* You can remove the password parameter if you want the AP to be open. */
  WiFi.softAP(ssid, password);
  WiFi.mode(WIFI_AP);

  Udp.begin(20001);
```

```
  IPAddress myIP = WiFi.softAPIP();
  Serial.print("AP IP address: ");
  Serial.println(myIP);
  server.begin();
  Serial.println("Server started");
}

void loop() {

  // if there's data available, read a packet
  int packetSize = Udp.parsePacket();
  if (packetSize) {
    //Serial.printf("Received packet of size %d from %s:%d\n    (to %s:%d, free heap = %d B)\n",
    //              packetSize,
    //              Udp.remoteIP().toString().c_str(), Udp.remotePort(),
    //              Udp.destinationIP().toString().c_str(), Udp.localPort(),
    //              ESP.getFreeHeap());

    // read the packet into packetBufffer
    int n = Udp.read(packetBuffer, UDP_TX_PACKET_MAX_SIZE);
    packetBuffer[n] = 0;
    //Serial.println("Contents:");

    //Here we activate the led
    float light = atof(packetBuffer)*255;
    Serial.println((int)light);
    analogWrite(LED_BUILTIN, light*4+3);
    analogWrite(4, light*4+3);
    delay(100);
    ////////////////////////
    /////Quitar sonido

    // send a reply, to the IP address and port that sent us the packet we received
    Udp.beginPacket(Udp.remoteIP(), Udp.remotePort());
    Udp.write(ReplyBuffer);
    Udp.endPacket();
  }

}
```

En primer lugar definimos el **nombre de la red** que vamos a crear y su contraseña. En este caso el **id** será "**Oculus**" y la **contraseña** "**123456789**".

```
#ifndef STASSID
#define STASSID "Oculus"
#define STAPSK  "123456789"
#endif

/* Set these to your desired credentials. */
const char *ssid = STASSID;
const char *password = STAPSK;
```

Luego definiremos dos **buffers de texto**, uno para **englobar las respuestas** y otro para **almacenar los envíos**. Además de definir un **puerto** para **escuchar** los **envíos**, si recordamos el código de **Unity3D** para enviar paquetes, decidimos que el puerto del **ESP8266** sería el **20001**.

```
ESP8266WebServer server(20001);

unsigned int localPort = 20001;      // local port to listen on

// buffers for receiving and sending data
char packetBuffer[UDP_TX_PACKET_MAX_SIZE + 1]; //buffer to hold incoming packet,
char  ReplyBuffer[] = "acknowledged\r\n";       // a string to send back

WiFiUDP Udp;
```

Ahora, en la función **Setup()** vamos a declarar el **led** que utilizaremos para saber la magnitud del **float recibido**, cuanto más brille menor es el número. También crearemos el **access point** con los datos que tenemos e iniciaremos el servidor.

```
WiFiUDP Udp;

void setup() {
  Serial.begin(115200);

  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(4, OUTPUT);

  Serial.print("Configuring access point...");
  /* You can remove the password parameter if you want the AP to be open. */
  WiFi.softAP(ssid, password);
  WiFi.mode(WIFI_AP);

  Udp.begin(20001);

  IPAddress myIP = WiFi.softAPIP();
  Serial.print("AP IP address: ");
  Serial.println(myIP);
  server.begin();
  Serial.println("Server started");
}
```

Con esto ya tendríamos una **red creada**, a la que podríamos **conectarnos con las Oculus Quest** o cualquier otro dispositivo, además, la placa estará a la escucha de paquetes.

Para **mantener** el **servidor constantemente** a la **escucha**, vamos a crear una bloque "**if**" en el que entraremos solo si se ha detectado un paquete. En este bloque lo primero que haremos es leer el paquete y convertirlo a **float**.

```
void loop() {

  // if there's data available, read a packet
  int packetSize = Udp.parsePacket();
  if (packetSize) {
    //Serial.printf("Received packet of size %d from %s:%d\n    (to %s:%d, free heap = %d B)\n",
    //              packetSize,
    //              Udp.remoteIP().toString().c_str(), Udp.remotePort(),
    //              Udp.destinationIP().toString().c_str(), Udp.localPort(),
    //              ESP.getFreeHeap());

    // read the packet into packetBufffer
    int n = Udp.read(packetBuffer, UDP_TX_PACKET_MAX_SIZE);
    packetBuffer[n] = 0;
    //Serial.println("Contents:");

    //Here we activate the led
    float light = atof(packetBuffer)*255;
```

La función para iluminar los **leds** toman valores entre **0** y **1023**, donde **1023** significa completamente apagado y **0** completamente **iluminado**. Por otro lado, la patilla encargada de comunicarse con el motor en la parte del **hardware** toma valores entre **0** y **255**, por lo que multiplicaremos el "**float**" por **255** primero, y después lo multiplicaremos por **4** solo en la función de escritura en el led.

```
//Here we activate the led
float light = atof(packetBuffer)*255;
Serial.println((int)light);
analogWrite(LED_BUILTIN, light*4+3);
analogWrite(4, light*4+3);
delay(100);
//////////////////////////
/////Quitar sonido
```

Después de esto, esperamos **100 milisegundos** y enviamos una respuesta, conteniendo únicamente la palabra "**acknowledged**".

```
  // send a reply, to the IP address and port that sent us the packet we received
  Udp.beginPacket(Udp.remoteIP(), Udp.remotePort());
  Udp.write(ReplyBuffer);
  Udp.endPacket();
}
```

Con esto ya tendríamos la parte de la **conectividad solucionada**, y esto se puede demostrar enviando paquetes desde las **Oculus Quest** hasta el **microcontrolador**. Lo único que faltaría para terminar el proyecto es crear el **hardware**.

CAPÍTULO 3: Hardware

Parte 1: Introducción

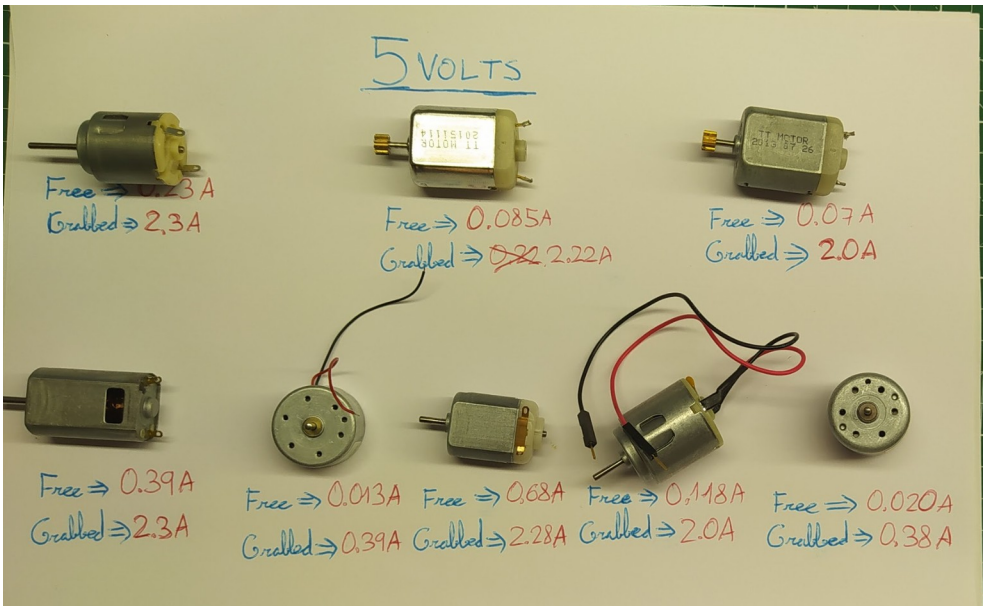
Hasta ahora tenemos un sistema capaz de ejecutar tres aplicaciones distintas en modo modo clásico y en modo pseudo-háptico y comunicar una variable de forma constante a un microcontrolador. El paso final es utilizar esa variable para **controlar un motor** que realice una **fuerza de tensión**.

Parte 2: El motor

Para controlar el hardware vamos a utilizar el **microcontrolador ESP8266**, el cual una vez hayamos introducido el código puede ser utilizado sin necesidad de estar conectado al ordenador. A partir de ahora conectaremos el **ESP8266** a una **batería externa**.

Primero vamos a hacer una **selección de materiales**, empezando por el motor. Para ello barajamos distintos tipos de **motores de corriente continua**. Finalmente por cuestiones de eficiencia energética y comodidad hemos utilizado un **motor DC** del laboratorio que consume a lo sumo **0.6A** con **5V**.

Se probaron **ocho motores** del laboratorio, utilizando en todos ellos un voltaje de **5 voltios** y midiendo el **amperaje** tanto cuando el motor **gira con libertad** como cuando se **constriñe el eje**. Sabiendo que el **transistor** utilizado para controlar el paso de la corriente al motor **no soporta intensidades mayores a 0.6A**, todos aquellos que superaban esta intensidad al estar agarrados fueron directamente **descartados**. De entre todos los restantes se eligió el que **mayor fuerza** tenía, teniendo en cuenta que el mismo **era invertible** (que no reseteará el microcontrolador al ser girado contra su voluntad) ya que eso implicaría añadir diodos al circuito.

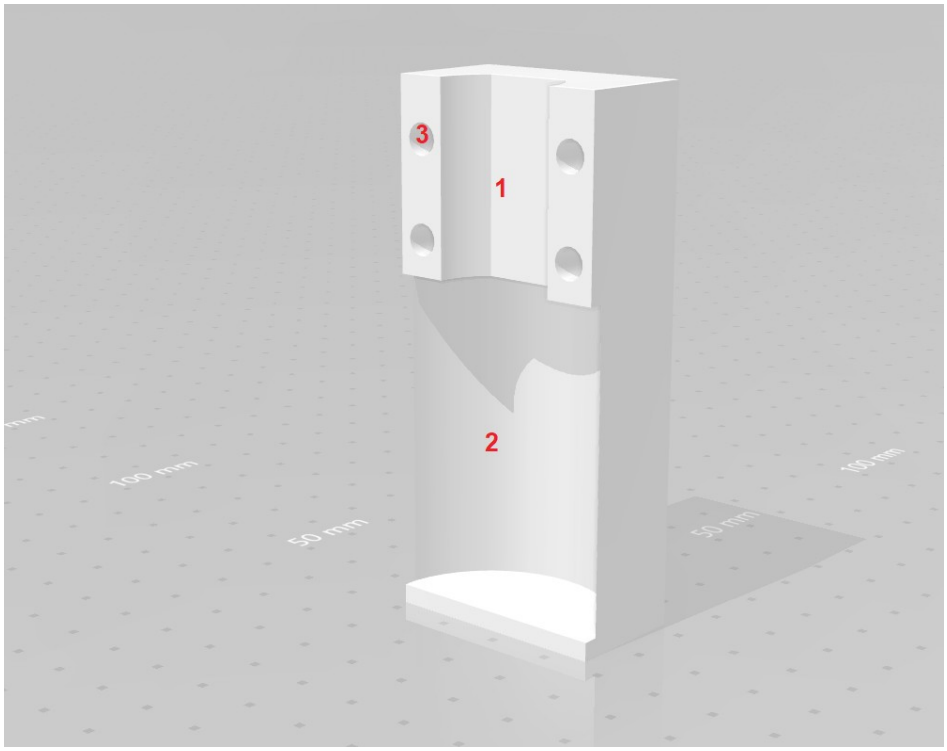


***Motores candidatos** para el proyecto, finalmente fue seleccionado el **tercero** de la **segunda fila**.*

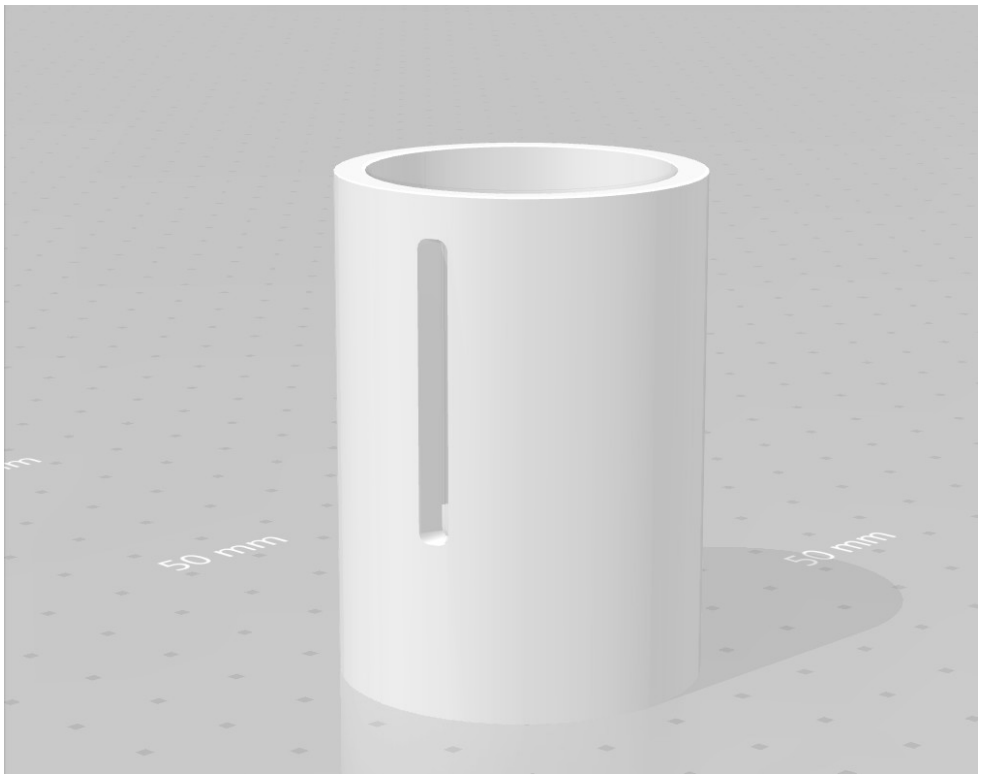
Parte 3: La carcasa

Ahora que tenemos las dimensiones del motor podemos crear una **carcasa** que contenga el **carrete del proyecto**. Para crear estas piezas se ha utilizado el programa de diseño 3D "**Catia**". Llegando a crear **tres piezas**. Las cuales posteriormente han sido ensambladas usando **pegamento** y **bridas**:

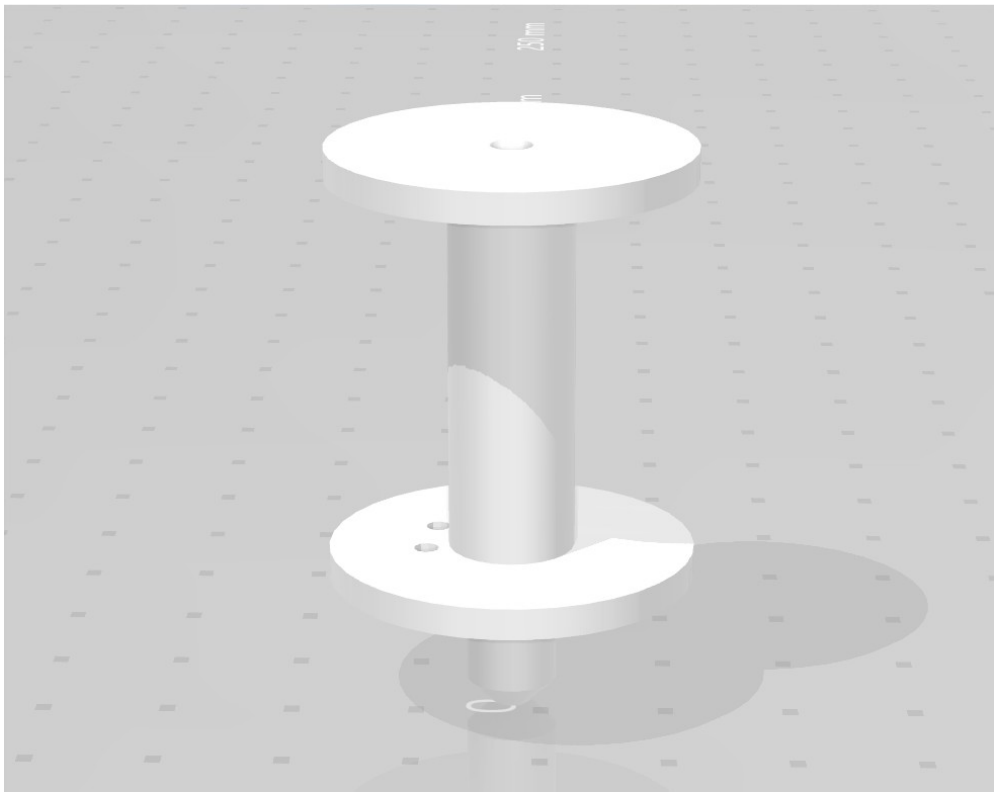
- Un **soporte**, el cual posee **dos orificios** a medida, uno para el **cilindro del carrete** y otro para el **motor**, habiendo en la parte del motor **cuatro agujeros** para sujetarlo con **bridas**.
- Un **cilindro** con una **ranura** por la cual podrá salir la **cuerda**. Es importante que la ranura sea más estrecha que la **anilla** y más ancha que el hilo a utilizar.
- Un **eje** sobre el cual rotará la cuerda, el cual tiene un pequeño **agujero** por el cual se introducirá el eje del motor.



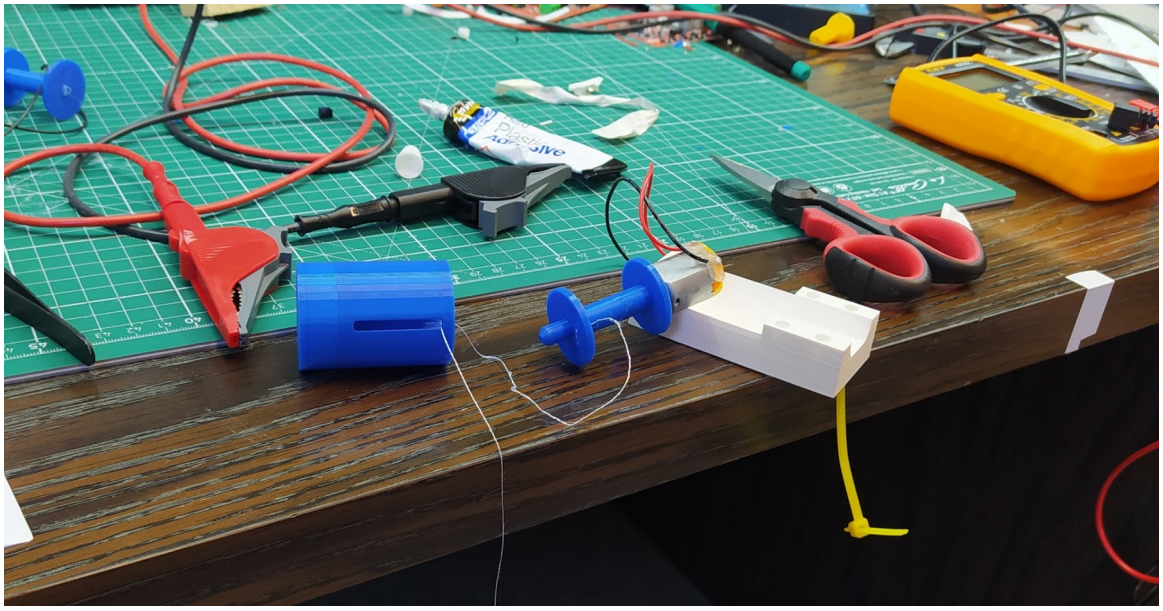
Base del carrete. ***1) Surco*** *para insertar el* ***motor****.* ***2) Surco*** *para insertar el* ***cilindro*** *del carrete.* ***3) Agujeros*** *para colocar las* ***bridas****.*



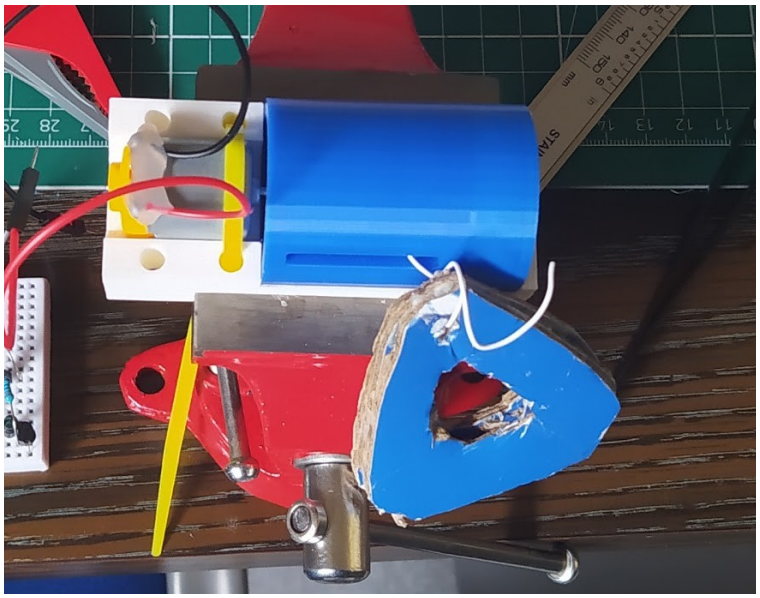
Cilindro *que alberga el* ***eje*** *sobre el que gira la cuerda.*



***Eje** sobre el cual **gira** la **cuerda**. Tiene **dos agujeros** en la aleta inferior para poder **atar** la **cuerda**.*



***Piezas** durante el ensamblado.*



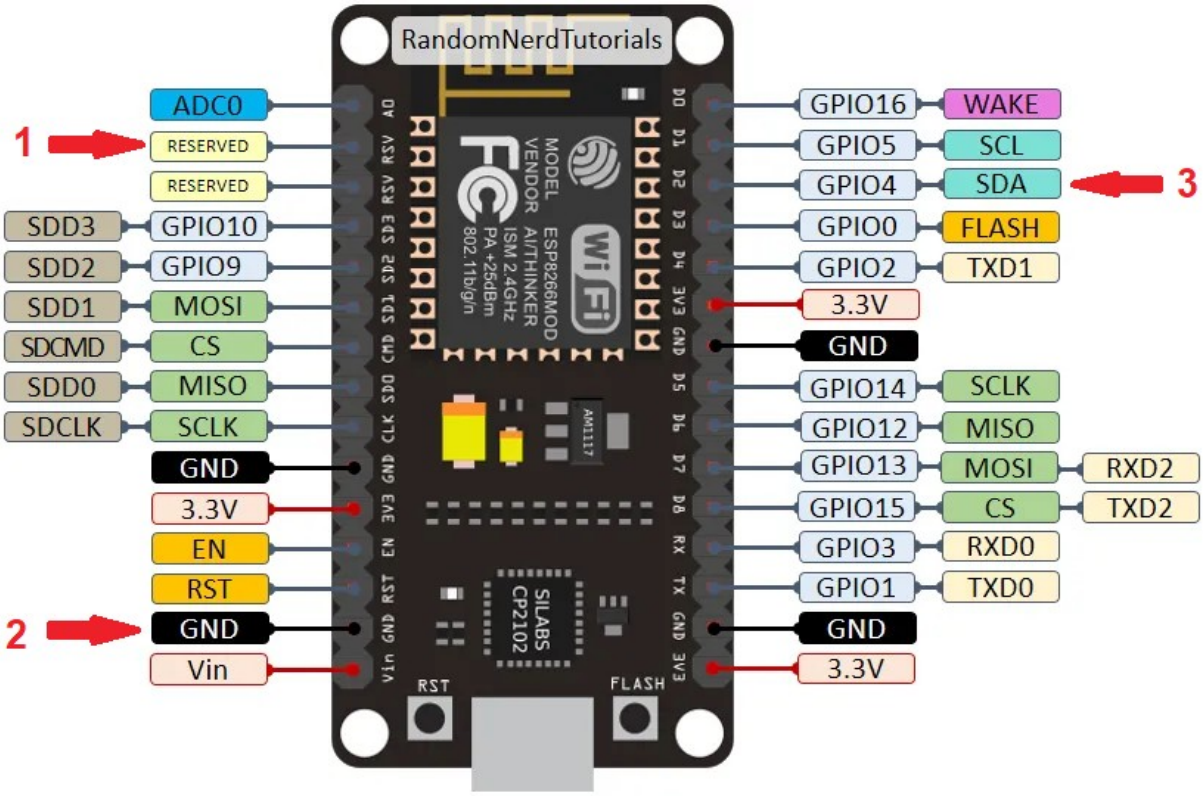
Piezas tras el ensamblado y adición de un **hilo** y **anilla** temporales.

Parte 4: El panel electrónico

Una vez añadidos un **cable** a cada **borne** del **motor**, podemos pasar a crear el circuito que unirá el **motor** con el **ESP8266**. Para ello usaremos una pequeña placa **protoboard**, en la cual montaremos el **sistema electrónico**.

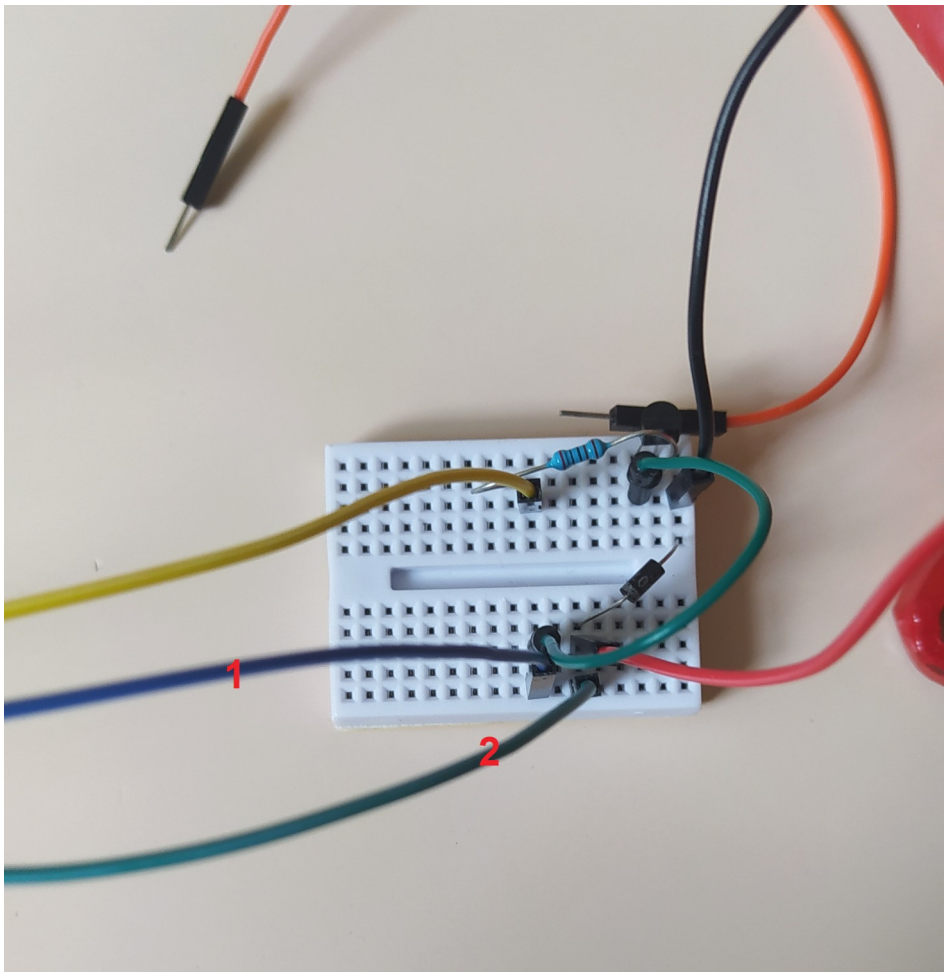
Este sistema tiene por objetivo suministrar **5 voltios** de **energía** al **motor** de forma condicional, dependiendo del valor que le otorgue las aplicaciones.

En primer lugar vamos a conectar el **ESP8266** con la **protoboard**. Para ello vamos a usar únicamente **tres de sus pines**, un pin de **5 voltios**, uno de **tierra** y uno que otorgue una **señal digital**.



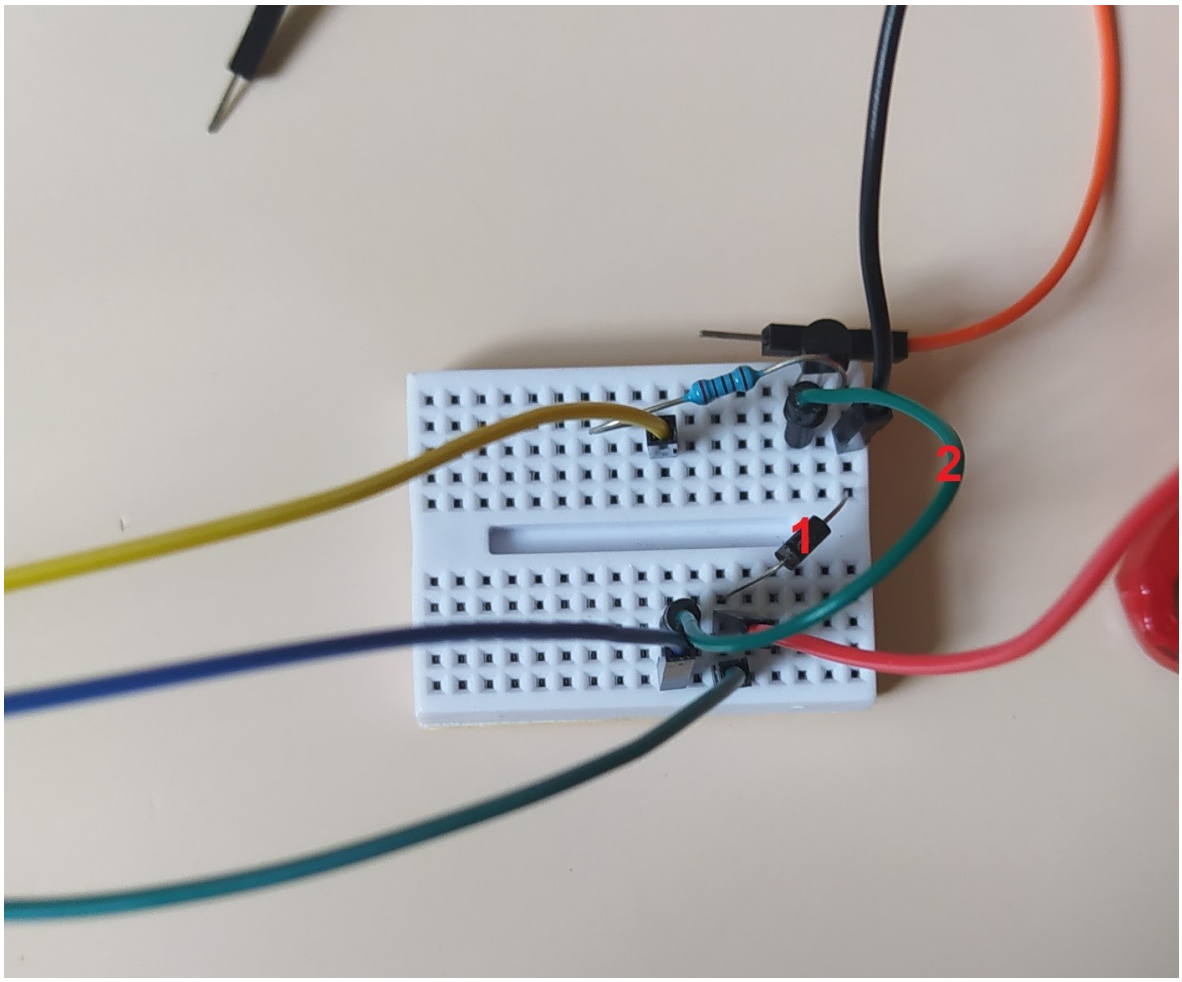
***ESP8266**. **1)** Pin que **suministra** de forma constante **5 voltios**. **2)** Pin de **tierra**. **3)** Pin con el cual vamos a controlar la **entrada** de **corriente** al motor.*

Colocaremos un cable en cada uno de estos pines, y de ahí llevaremos los **2** encargados de la **corriente** a la **protoboard**.



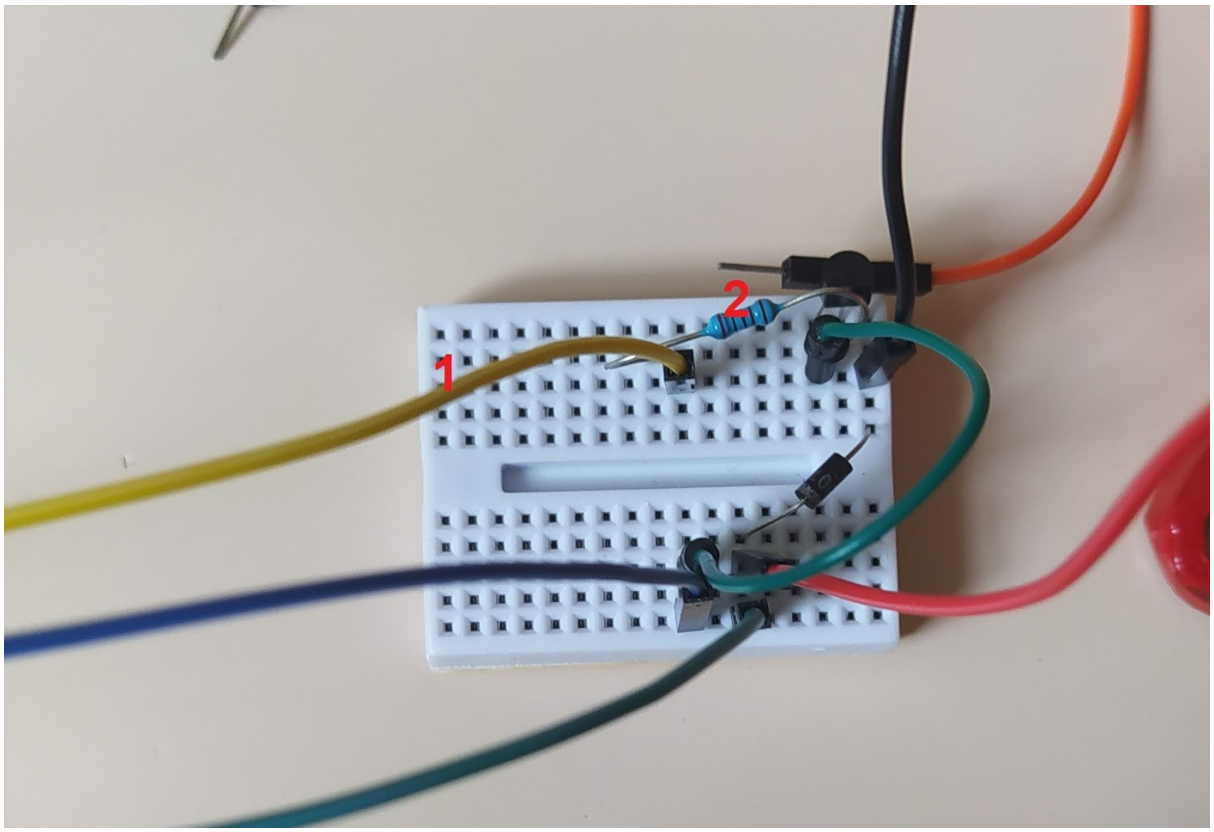
***Protoboard**. **1)** Conexión al pin de **5 voltios**. **2)** Conexión al pin de **tierra**.*

Además necesitaremos un **transistor** y una **resistencia** para no quemar el circuito. El **transistor** funciona como un **botón** que permite el **paso de la corriente** por sus **dos patillas laterales** cuando la **patilla central** es **estimulada** por una corriente. Al utilizar un pin digital solo podemos **permitir pasar 0 o 5 voltios**, para generar una **sensación de continuidad** mandaremos una señal a la patilla central con una cierta **frecuencia**, entre **0** y **255 veces por segundo**, creando de este modo la sensación de que le estamos suministrando corrientes distintas. Cada una de las **patillas laterales** se conectará a uno de los pines que suministran **corriente al motor**, poniendo además un **condensador** entre la **patilla** del **transistor** y el **pin** de **0 voltios**.



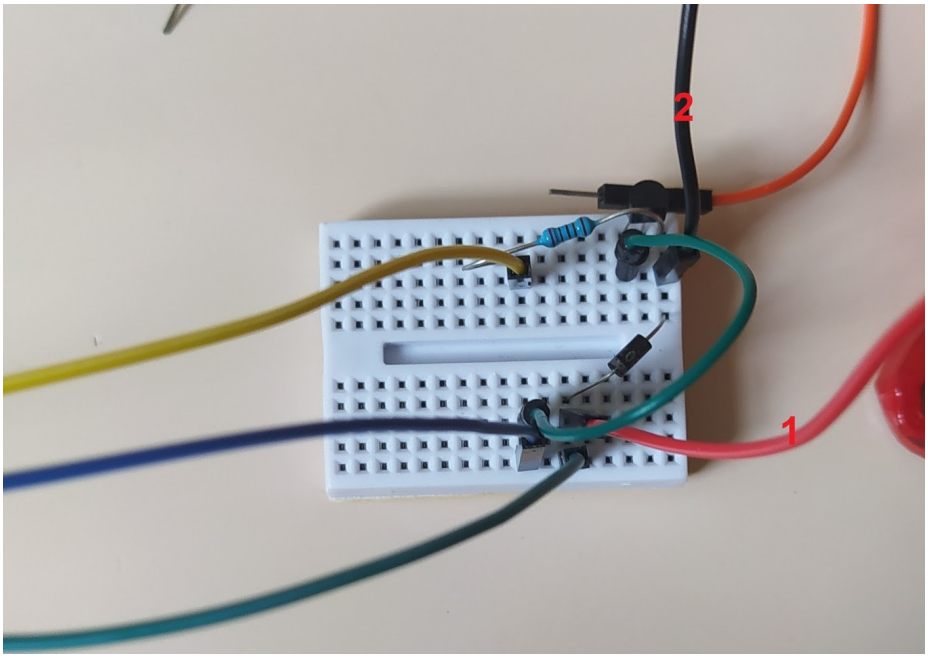
***Protoboard**. **1)** Conexión al pin de **0 voltios** mediante un **condensador**. **2)** Conexión al pin de **5 voltios**.*

Ahora vamos a **conectar** el **pin** encargado de suministrar la **señal** de control al **transistor**, entre medias incluiremos una **resistencia**. Con esto ya tendríamos un circuito en el cual la **frecuencia del paso** de corriente está controlada por el **microcontrolador**, quien a su vez **recibe** la orden de las **gafas** de **realidad virtual**, las cuales están siendo **controladas** por su **usuario**.



Protoboard. 1) *Conexión al pin de* ***control****.* ***2)*** *Conexión a la* ***patilla central*** *del* ***transistor*** *con una* ***resistencia*** *entre medias .*

Por último conectaremos los cables del motor al circuito.



Protoboard. 1) *Conexión a uno de los* ***bornes****.* ***2)*** *Conexión al otro* ***borne****.*

Con esto ya tendríamos todo el **hardware integrado**, el usuario ya puede sentir las **fuerzas** que realiza en las aplicaciones gracias a este **sencillo sistema**. Con el sistema ya creado tan solo queda **probar el sistema** y sacar **conclusiones** y **posibles mejoras** del **prototipo**.

CAPÍTULO 4: Conclusiones

Parte 1: Introducción

Una vez terminado todo el proyecto dedicaremos el tiempo sobrante a **analizar el proyecto** y **sacar conclusiones**. Se han realizado dos tipos de pruebas, unas pruebas objetivas centradas en el rendimiento del hardware y unas pruebas en usuarios, **enfocadas** más a que tan **satisfechos** se encontraban con el producto.

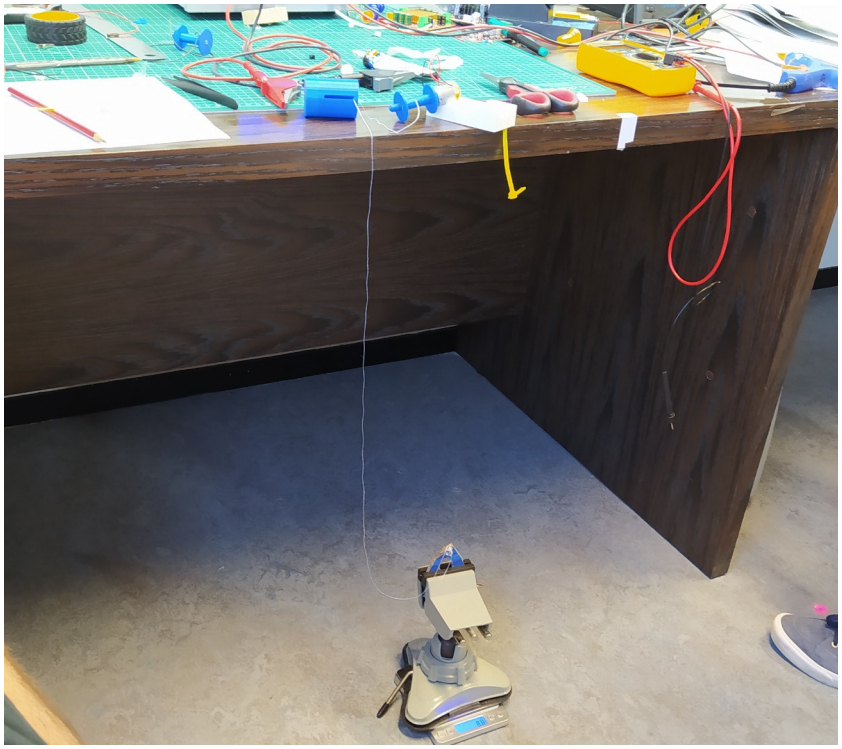
Parte 2: Pruebas objetivas

Las pruebas objetivas se han centrado principalmente en los **motores**, dedicándonos a buscar aquellos que **mayor fuerza** tenían. Para ello, como ya se dijo en el apartado del hardware, probamos una serie de motores para medir su intensidad al aplicar **5 voltios** cuando se halla **libre** y cuando se **bloquea** el **eje**. Los resultados son los siguientes.

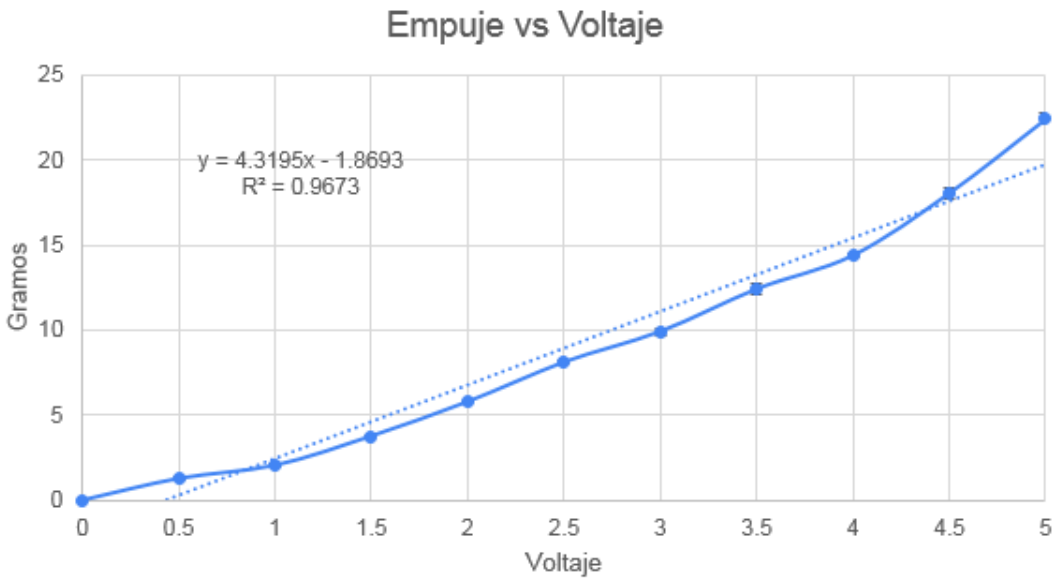
Motor:	Motor 1	Motor 2	Motor 3	Motor 4	Motor 5	Motor 6	Motor 7	Motor 8
Intensidad (libre)	0.23 A	0.085 A	0.07 A	0.39 A	0.013 A	0.68 A	0.118 A	0.020 A
Intensidad (bloqueado)	2.3 A	2.22 A	2.0 A	2.3 A	0.39 A	2.28 A	2.0 A	0.38 A

Una vez tomadas todas las mediciones se descartan aquellos con una intensidad de **más de 0.6A** en cualquier momento. El **motor 8** es **descartado** también debido a que no genera **apenas fuerza** y sería **demasiado complicado** crearle una **carcasa** que se ajuste a el.

Ahora realizaremos pruebas de fuerza en el **motor 5,** pero las haremos con el sistema completamente montado. Estas pruebas consistirán en añadir un peso a la anilla del carrete, el cual está apoyado sobre una báscula de precisión. El dato a anotar es la fuerza que realiza ante diversos **voltajes entre 0.5** y **5**.



*Como peso se utilizó un **tornillo de mesa** , con el cual podíamos agarrar la **anilla** de forma segura.*



***Resultados** mostrados en una gráfica.*

En la gráfica observamos como la **masa** levantada **aumenta** conforme aumentamos el voltaje, sin llegar a superar en ningún momento los **22 gramos**.

Parte 3: Conclusiones y posibles mejoras de las pruebas objetivas

Las conclusiones principales extraídas de las pruebas objetivas son:

- Los **motores** son **endebles**, pero si que muestran una **progresión** dependiendo del **voltaje** aplicado.

- Muchos de los **motores** descartados poseen una **fuerza excesiva** para lo requerido en este proyecto.

- La **carcasa cumple** con la función para la que fue diseñado.

Para solucionar los problemas se plantean las siguientes soluciones:

- **Añadir** un **motor** con **mayores prestaciones**, aunque esto implique cambiar la carcasa y el circuito electrónico.

- **Añadir** una **reductor** al **motor** para que realice más fuerza, aunque esto implique cambiar la carcasa.

- **Aumentar** el **voltaje** para obtener más fuerza, aunque esto implique un mayor calentamiento del motor, llegando a poder derretir el pegamento que sujeta el eje y la silicona que sujeta los cables a los bornes.

- Añadir **varios motores**, aunque esto implique reajustar la electrónica del proyecto.

Parte 4: Pruebas subjetivas

El sistema y especialmente las **funciones clásicas** y **pseudo-hápticas** han sido **probadas** en diferentes personas. Al tratarse de una **aplicación** principalmente **lúdica** he preferido darles **completa libertad** en el experimento, dejando que prueben el sistema mientras me preguntan dudas y me ofrecen sugerencias.

Algunas sugerencias recurrentes sobre el software son:

- Utilizar una **forma discreta** de **mostrar** la **precisión** del disparo, por ejemplo: fallido, normal, bueno, muy bueno. (Arco VR).

- **Cambiar** el **botón** de **agarre** de los planetas a uno más intuitivo, como por ejemplo el gatillo. (Planetas VR).

- Permitir **girar** con el otro **joystick** al jugador. (Planetas VR).

- Añadir un botón para **reiniciar** los **planetas** (Planetas VR).

- Añadir un **entorno más llamativo** (Botón VR).

- Añadir una acción relacionada al **pulsado del botón** (Botón VR).

Por desgracia no pude realizar pruebas en tanta gente con el elemento hardware debido a:

- **Dificultades** para **transportar** el hardware.

- **Dificultades** para **organizar** a la gente.

- **Dificultades** derivadas de la **pandemia** actual.

Sin embargo logré experimentar con unas **pocas personas** utilizando la carcasa creada y la aplicación móvil "**UDP Sender/Receiver**". Mi labor era **variar** la **fuerza** del **motor** enviando paquetes desde la aplicación y ellos me indicarían si notan algún tipo de diferencias. No se utilizó **ningún protocolo común**.

Parte 5: Conclusiones y posibles mejoras de las pruebas subjetivas

Con respecto al **software**, las opiniones de los usuarios han sido muy **favorables**, pero se ve un mayor interés en las aplicaciones del arco y los planetas. Con respecto a los modos **pseudo-hápticos**, los usuarios informan que sí que consideran más **inmersiva** la aplicación al activar estos modos.

En cuanto al **hardware**, si bien los usuarios dicen **notar diferencias** entre los distintos voltajes del motor, echan de menos una **mayor fuerza** tensora y un sistema de recogida de cuerda **más rápido**. Un usuario ha comentado que **mejoraría** mucho el **modo háptico** si el **motor** pudiera **girar** en **ambos sentidos** para que además de recoger cuerda pudiera soltarla.

Las **conclusiones** resumidas serían:

- Las **aplicaciones** son satisfactoriamente **entretenidas**, pero la aplicación del **botón** es demasiado **simple**.

- Los **modos pseudo-hápticos** sí que generan nuevas sensaciones en los usuarios que resultan en entornos más **inmersivos**.

- El **modo háptico** funciona **correctamente**, pero el **motor no genera la suficiente fuerza** como para generar una sensación creíble.

Las posibles **soluciones** son:

- **Mejorar** la **escena** del **botón** añadiendo efectos a los botones o incluso varios botones en la escena.

- Se debería buscar un **motor** que generará **más fuerza** y **velocidad** de **giro**.

- Se podría hacer más notable modo **háptico** si en vez de subir la fuerza con respecto a la tensión de forma lineal lo hiciera de forma **exponencial**.

Bibliografía

(1) Endo, T., Kawasaki, H., Mouri, T., Ishigure, Y., Shimomura, H., Matsumura, M., & Koketsu, K. (2010). Five-fingered haptic interface robot: HIRO III. *IEEE Transactions on Haptics*, *4*(1), 14-27.

(2) Paljic, A., & Coquillart, S. (2004, June). A passive stringed haptic system for immersive environments. In *Eurohaptics*.

(3) Ma, S., Akahane, K., & Sato, M. (2015). String-based force display for mobile haptics. In *SIGGRAPH Asia 2015 Haptic Media And Contents Design* (pp. 1-2).

(4) Achibet, M., Girard, A., Talvas, A., Marchal, M., & Lécuyer, A. (2015, March). Elastic-Arm: Human-scale passive haptic feedback for augmenting interaction and perception in virtual environments. In *2015 IEEE Virtual Reality (VR)* (pp. 63-68). IEEE.

(5) Hinchet, R., Vechev, V., Shea, H., & Hilliges, O. (2018, October). Dextres: Wearable haptic feedback for grasping in vr via a thin form-factor electrostatic brake. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology* (pp. 901-912).

(6) Wang, C. H., Hsieh, C. Y., Yu, N. H., Bianchi, A., & Chan, L. (2019, March). HapticSphere: Physical support to enable precision touch interaction in mobile mixed-reality. In *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)* (pp. 331-339). IEEE.

(7) https://learn.unity.com/tutorials

(8) https://www.youtube.com/user/Unity3D

(9) https://www.youtube.com/user/Brackeys

(10) (https://www.youtube.com/user/juande

(11) https://www.youtube.com/user/AwfulMedia

(12) https://www.youtube.com/user/quill18creates

(13) https://docs.unity3d.com/Manual/index.html

(14) https://www.youtube.com/channel/UCPJlesN59MzHPPCp0Lg8sLw

(15) https://www.youtube.com/channel/UCHM37DnT_QGJT5Zyl4EmqcA

(16) https://docs.unity3d.com/Manual/XR.html