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

Evaluación de interfaces de entrada para un display tridimensional holográfico en tareas de trazado y dibujo

Grado en Ingeniería Informática

Trabajo Fin de Grado

Iñaki Zabalegui Aizpun

Oscar Ardaiz Villanueva

Pamplona, 3 de septiembre de 2020

Índice del proyecto

1-Presentación e introducción al proyecto 3
1.1 Objetivos del proyecto y contenido de la memoria 4
2-Explicación de los distintos métodos 5
2.1 Método de tableta moviendo el plano mediante teclado 5
2.2 Método de tableta y moviendo el plano con la Leap Motion 7
2.3 Método de Leap Motion y bola en el espacio 8
3-Estudio del arte 9
4-Herramientas utilizadas 11
4.1 A nivel de hardware 11
4.1.1 Looking Glass Holographic Display 11
4.1.2 Leap Motion 16
4.1.3 Tableta Wacon 25
4.2 A nivel de software 27
4.2.1 Unity 27
5-Pruebas preliminares 29
6-Implementación del proyecto 37
6.1Elementos en común 37
6.2 Método de tableta moviendo el plano mediante teclado 40
6.3 Método de tableta y moviendo el plano con la Leap Motion 47
6.4 Método de Leap Motion y bola en el espacio 52
7-Evaluación y datos obtenidos de los usuarios 57
8-Discusión de resultados 65
9-Bibliografía 68

1-Presentación e introducción al proyecto

Desde pequeño me ha gustado pintar y dibujar, ya sea cosas básicas que se pintan siendo de niños como árboles, una casa con su tejado, chimenea, ventanas y puerta, unas nubes… Además de eso, de pequeño me hicieron saber que tenía muy buen gusto para combinar los colores, por lo que me parecía algo necesario de introducir en el proyecto, ya que los resultados que se pueden obtener son más coloridos y divertidos.

El hecho de dibujar me sigue gustando, pero lo que más me llama la atención hoy en día es, además de los videojuegos y lo realistas e increíbles que se desarrollan en esta generación (y no quiero casi ni pensar lo que traerá las nuevas generaciones), son los monitores que permiten observar la realidad holográficamente. Tomando unos ejemplos más populares aunque no del todo realista, son los ejemplos de Los Vengadores cuando emplean un monitor holográfico para distintos usos (ver el estado de la armadura, planear misiones, etc.) o todos los ejemplos holográficos que podemos ver en cualquiera de las sagas de Star Wars, así como las películas y series futuristas.

Con esto quiero llegar a que, en cuanto tuve la oportunidad de poder trabajar con el Looking Glass, fue una decisión que no dudé aceptar. En cuanto descubrí cómo y con qué se puede trabajar con este display, las opciones eran muy diversas, pero la que más prevalecía era la de poder pintar en un espacio 3D.

Además, vivimos en un mundo en el que prevalece la realidad virtual, concretamente las gafas VR, con las que cada vez se realizan más trabajos de fin de grado y/o máster, por lo que me parecía poco original realizar un proyecto con las gafas.

Para empezar a desarrollar los diferentes métodos, se pensaron qué herramientas podían usarse para poder realizar las tareas, y pensar en las distintas combinaciones de éstas.

Ha sido todo un desafío el implementar este proyecto al completo, ya que antes de comenzar con ello no apenas había tenido contacto con estos elementos de hardware y tampoco había tenido el placer de poder trabajar con la increíble herramienta de desarrollo software como es Unity. Puedo decir que, sin ningún lugar a dudas, durante este proceso de desarrollo he estado proceso he estado en pleno proceso de aprendizaje, bien para entender el funcionamiento de las distintas herramientas utilizadas y para después poder implementar el proyecto final.

1.1 Objetivos del proyecto y contenido de la memoria

Los objetivos principales son del proyecto son comprender el funcionamiento de los distintos interfaces de entrada, como son el Looking Glass Holographic display, la Leap Motion, la tableta Wacom y el teclado del ordenador (aunque destacaremos todos los anteriores, ya que éste es habitual), y el desarrollo de distintos métodos de trazado y dibujo en un espacio tridimensional, en el que estos método usaremos distintas combinaciones de estas interfaces.

Esta memoria contiene, primero, la explicación de los métodos que se han desarrollado, seguido del estudio del arte actual. Una vez explicado estos apartados, me centraré en las distintas herramientas que he utilizado para desarrollar el proyecto, dividiendo estas herramientas por tipo hardware y software. Después, contaré las distintas pruebas que realicé con el Looking Glass, ya que es el protagonista de las interfaces de entrada sobre la que he basado el proyecto. Seguido de esto, explicaré la implementación de los métodos, aunque explico también de manera general los elementos en común que tienen todos los métodos. A continuación, mostraré y explicaré las pruebas realizadas a distintos usuarios/as y discutiré los resultados obtenidos en otro apartado. Y por último, tenemos la bibliografía el cual he usado tanto para realizar este documento como para desarrollar el proyecto al completo.

2-Explicación de los distintos métodos

2.1 Método de tableta moviendo el plano mediante teclado

Este primer método consiste en dibujar sobre un plano que se encuentra dentro del display con una posición inicial. Este plano contiene una fina franja amarilla en la parte superior, otra de color azul a la derecha y de color rojo a la izquierda para poder distinguir y orientar los lados del plano una vez lo movamos y rotemos. Como explicaremos más adelante, la zona activa de la tableta corresponde con el área del plano de dibujo.

El/la usuario/a puede mover, como el nombre de este sistema indica, mediante el teclado del ordenador. Concretamente, pulsando la tecla W el plano se desplazará hacia el fondo, es decir, teniendo en cuenta las coordenadas, se moverá en el plano Z con valor positivo. En cambio, si pulsamos la tecla S el plano vendrá hacia nosotros (teniendo en cuenta la posición inicial), en otras palabras, se moverá en el plano Z con valor negativo.

Además de poder mover el plano, el/la usuario/a tiene la posibilidad de poder rotar el poder rotar el plano con las flechas del teclado dependiendo de su valor: pulsando la flecha hacia arriba el plano se inclinará hacia adelante (rotación con valor positivo sobre el plano X), con la flecha hacia abajo el plano se inclinará hacia nosotros teniendo en cuenta la posición inicial (rotación con valor negativo sobre el plano X), y con las flechas izquierda y derecha el plano rotará hacia la izquierda y derecha, obviamente (rotación con valor positivo y negativo sobre el plano Y, respectivamente).

Imagen 2.1.1. *Usuario moviendo el plano de dibujo con 'W' hacia el fondo.*

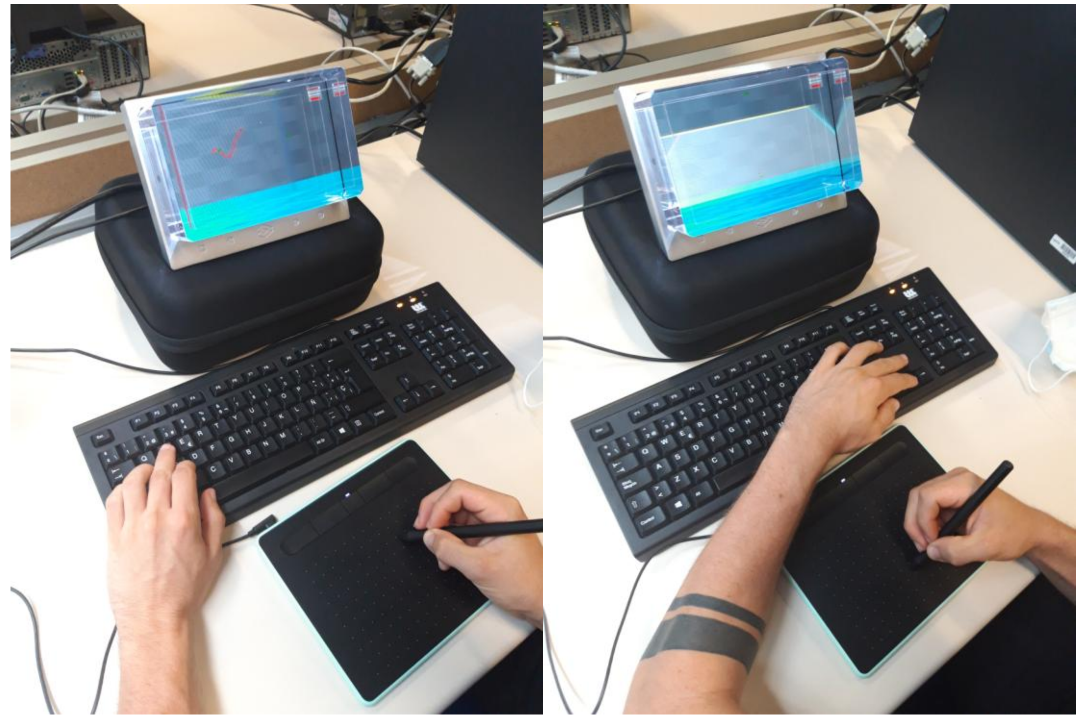


Imagen 2.1.2. *Usuario dibujando sobre plano de dibujo y rotando éste con las flechas.*

Un comentario que me gustaría decir acerca de este método antes de pasar al siguiente es que, al rotar el plano donde dibujamos, hay que tener en cuenta que si queremos mover el plano con las teclas W y S, éste no se moverá respecto nuestra visión frontal al display, sino que se moverá como si estuviéramos colocados en frente del plano, es decir, al rotar el plano, nosotros "también" rotamos.

Para facilitar el dibujado del trazo, en el plano se mostrará un puntero el cual aparecerá cuando acerquemos el lápiz a la tableta. Para poder pintar, simplemente realizamos la simulación de un papel y bolígrafo, lápiz o cualquier utensilio de dibujo: mantenemos en contacto el lápiz con la tableta donde queramos dibujar, de tal forma que, cuando al mantener el contacto se creará el trazo dependiendo de la posición de la esfera. En otras palabras, es como si la bola dejara un "rastro" por donde pasa.

Además de cambiar la posición del plano, este método tiene la opción de poder cambiar el color del trazo mediante un selector de colores RGB (Red, Green, Blue) en la esquina superior derecha.

Para acabar con este método, decir que si la persona que en un momento esté usando este método ha rotado y movido el plano varias veces, esta persona tiene la opción de volver a situar el plano en la posición inicial con su rotación inicial pulsando la tecla R.

2.2 Método de tableta y moviendo el plano con la Leap Motion

Este método tiene la misma idea que el explicado anteriormente, es decir, rotar y mover un plano que tiene posición inicial, el cual dibujamos mediante la tableta y poder cambiar el color del trazo, pero con la diferencia de que en este caso, para mover y rotar el plano vamos a utilizar la mano, de tal manera que el plano seguirá de manera paralela a la palma de la mano. En otras palabras, podemos imaginar que tenemos la tableta pegada a la palma de la mano, y dependiendo de cómo movemos y/o rotamos la mano, el plano seguirá el mismo movimiento.

Imagen 2.2.1. *Usuario con el plano de dibujo paralelo a la mano e inclinando éste con la mano, preparado para pulsar espacio para parar el plano.*

Se puede pensar que es muy complicado mover el plano con la mano y dibujar a la vez con la otra mano. Para poder dar mayor precisión a los dibujos que se realicen, el sistema cuenta con la función de parar el plano pulsando la tecla espacio, de tal manera que movemos y rotamos el plano en un punto y queremos empezar a dibujar. Estando en ese punto, pulsamos el espacio y el plano quedará completamente quieto, pudiendo realizar tranquilamente los trazos deseados. Para volver a mover el plano, simplemente volvemos a pulsar la tecla espacio y podemos seguir manipulando el plano.

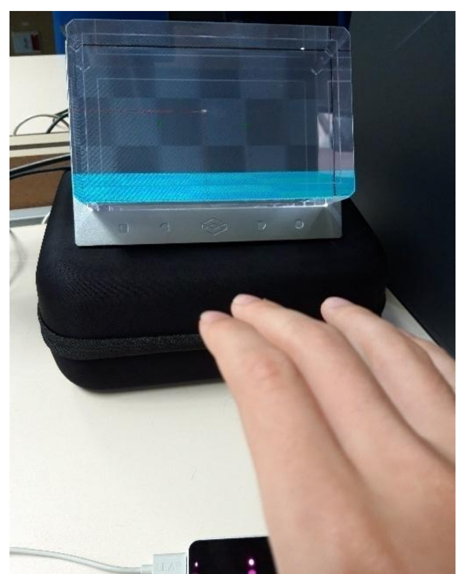
Este sistema también cuenta con la opción de volver el plano a la posición inicial pulsando la tecla R, pero para volver a mover el plano tenemos que pulsar la tecla espacio.

Una peculiaridad de este método es que, si empezamos a pintar y movemos el plano con la mano al mismo tiempo, sin levantar el lápiz de la tableta, dibujaremos en 3D, como hacemos en el siguiente método.

2.3 Método de Leap Motion y bola en el espacio

Este último método es más diferente que de los explicados hasta ahora. En este caso, tenemos una esfera que seguirá nuestra mano derecha, concretamente, el dedo índice. Esta esfera no tiene ningún tipo de rotación, solamente seguirá la mano que podemos mover en el espacio que contiene el display (y el campo el cual la *Leap Motion* tiene para la lectura de la mano, obviamente). Seguirá todas las direcciones posibles: arriba, abajo, izquierda, derecha, en diagonal…

Para empezar a pintar, lo único que se debe hacer es pulsar la tecla espacio, pero no sólo pulsar una vez, sino que, mientras mantengamos la tecla pulsada, se dibujará en el espacio un trazo que sigue a la bola. Con la posibilidad de poder mover en todas las direcciones, lo que se obtiene es el poder pintar en 3D, además de que el *Looking Glass* nos permite observar la profundidad y la altura que tienen los trazos.



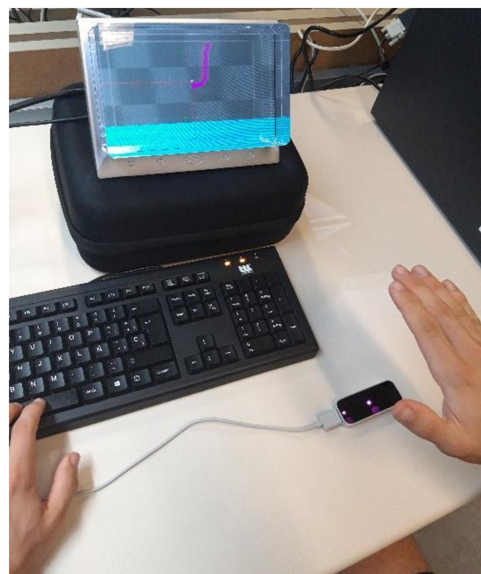


Imagen 2.3.1. *Usuario moviendo la esfera con la mano y dibujando con la tecla espacio.*

Para terminar con la explicación de este método de dibujo, decir que la bola contiene dos ejes, uno vertical y otro horizontal, de tal manera que ayudamos a los/las distintos/as usuarios/as a tener una mejor referencia en el espacio. Además, estos ejes reflejan una sombra con el elemento *Directional Light*, por lo que si no llegan a apreciar bien el punto en el que se encuentra el puntero, pueden guiarse mediante su sombra.

3-Estudio del arte

En estos momentos estamos en un mundo en el que los juegos de realidad virtual, o el mundo de realidad virtual en general, se encuentra en auge. Pero no sólo este mundo está en pleno crecimiento, sino que poco a poco está tomando cada vez con más fuerza los distintos monitores holográficos, aunque no con la proyección de la VR.

Y es que estos dos mundos comparten la misma funcionalidad, la de mostrar un espacio y/o todos los diferentes objetos en un espacio tridimensional, de tal manera que tengamos la sensación de estar en él. Además de esto, lo divertido es el hecho poder crear animaciones, realizar interacciones con los objetos existentes en el espacio o crear unos nuevos, ya sea mediante pellizcos o usando otras herramientas además de los monitores.

La principal diferencia de estos mundos es clara, y es el número de usuarios/as que pueden disfrutar de la herramienta, ya sean unas gafas de Realidad Virtual o un display holográfico. Éste primero solo puede usarse por una persona en cada uso y sólo ella pueda disfrutar de la experiencia en ese momento, mientras que un monitor holográfico permite la percepción de la Realidad Virtual creada por varias personas simultáneamente, pudiendo personalizar la experiencia para que la interacción sea realizada con todas esas personas, dependiendo de la funcionalidad con la que se realice el proyecto (dibujar en 3D, mover, rotar, pintar, crear objetos, etc.).

Enfocando más la idea de dibujar en un espacio tridimensional, estos son algunos de los ejemplos en los que me he basado a la hora de crear mi proyecto:

a) **T(ether): Spatially-Aware Handhelds, Gestures and Proprioception for Multi-user 3D Modeling and Animation**[1]

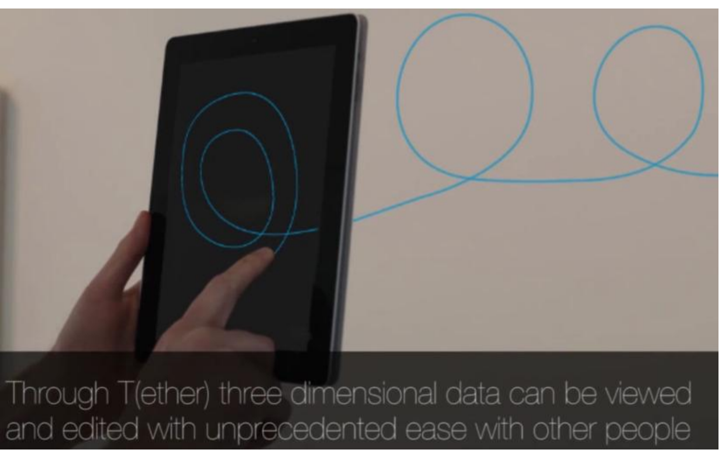


Imagen 3.1. *Captura de T(ether) en funcionamiento.*

b) Experimental Evaluation of Sketching on Surfaces in VR[2]

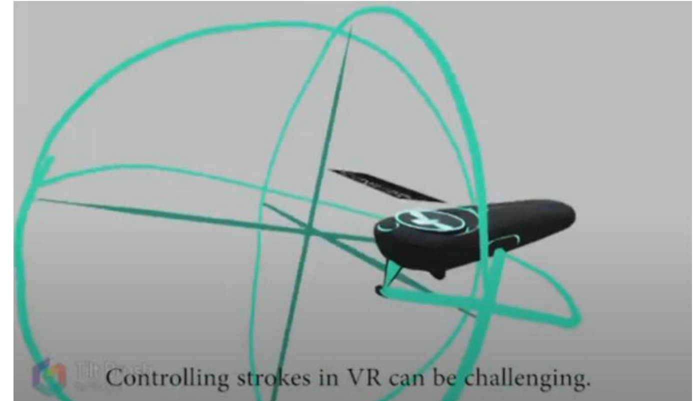


Imagen 3.2. *Captura de Sketch de VR en funcionamiento.*

c) SymbiosisSketch: Combining 2D and 3D Sketching for Designing Detailed 3D objects in Situ[3]

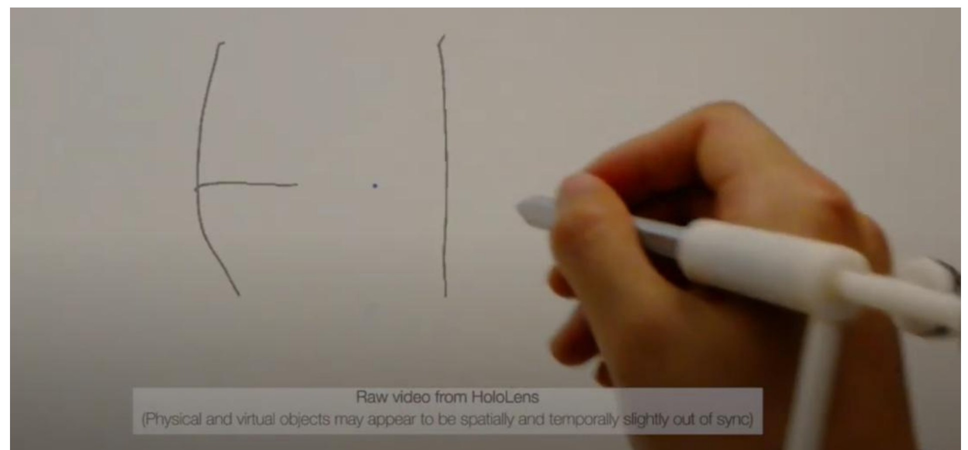


Imagen 3.3. *Captura de SymbiosisSketch dibujando 2D en 3D.*

d) PROPOSED APPLICATION: PAINTING

Imagen 3.4. *Captura de aplicación dibujando 2D en 3D con diferentes colores.*

4-Herramientas utilizadas

4.1 A nivel de hardware

4.1.1 Looking Glass Holographic Display

En los últimos cuatro años, un equipo de holográficos, ingenieros ópticos, mecánicos y eléctricos, desarrolladores de Unity y artistas gráficos en 3D han estado trabajando para crear el Looking Glass: una pantalla holográfica que utiliza una combinación de campo de luz y tecnologías volumétricas.

Ahora, por primera vez, grupos de personas pueden ver e interactuar con un mundo tridimensional virtual en su escritorio, sin la ayuda de auriculares VR o AR. Para ello, han creado una serie de herramientas para permitir a los creadores 3D dar vida fácilmente a su trabajo en el Looking Glass.

Para empezar a trabajar y comprender esta herramienta, disponemos de decenas de aplicaciones holográficas utilizando la **Biblioteca de aplicaciones** (*Library*). Estos incluyen videoclips volumétricos, cortometrajes y juegos holográficos, escaneo 3D,mascotas virtuales (como veremos más adelante), un previsualizador de modelos 3D y mucho más. Además, los creadores también podrán compartir sus propias aplicaciones a través de *Library* con los propietarios de Looking Glass en todo el mundo.

Además de tener su propio software desarrollado, el *Looking Glass Hologram Display* cuenta con distintas SDK y compatibilidades para poder trabajar con otros programas conocidos como son: Unity, Unreal Engine, Three.js, AutoCAD, Maya, Blender, Voxatron, Skecthfab, Poly, Cinema 4D, Paint 3D, 3D Print App (también propia del display y desarrollada por Looking Glass Factory), etc. [4]

Cabe destacar que Looking Glass dispone de una amplia gama de complementos periféricos como son Leap Motion Controllers, Nintendo Switch Joycon Controllers, Intel Real Sense, Microsoft Kinect, Xbox Game Controller, Arduino y más para ofrecer la posibilidad de desarrollar distintos tipos de aplicaciones y juegos dependiendo de la finalidad del mismo [5].

Decir también que para poder usar este dispositivo, el ordenador con el que trabajemos debe tener instalado el driver *Holoplay Service*, actualmente con la versión 1.0.0.

Estas son algunas de las grandes empresas que utilizan este tipo de producto: Intel, Disney, Microsoft, Schrödinger, Huge y Verizon.

Formatos:

Actualmente se encuentran disponibles 3 versiones de Looking Glass Holographic Display: Kit de desarrollo de 8.9”, Kit de desarrollo de 15.6”/ 15.6” Pro y Pantalla inmersiva 8K.

Imagen 4.1.1.1. *Looking Glass de 8.9” y 15.6”.*

Kit de desarrollo de 8.9”

Un sistema de visualización holográfica personal de 8.9” creado para el escritorio de cualquier creador que trabaje en 3D. Los kits para desarrolladores Looking Glass son compatibles con Unity, Unreal Engine y una gran cantidad de otras plataformas de desarrollo 3D. Esta es la versión con la que hemos desarrollado el proyecto [4]. En la siguiente imagen podemos observar la escala, dimensiones y peso de este formato:

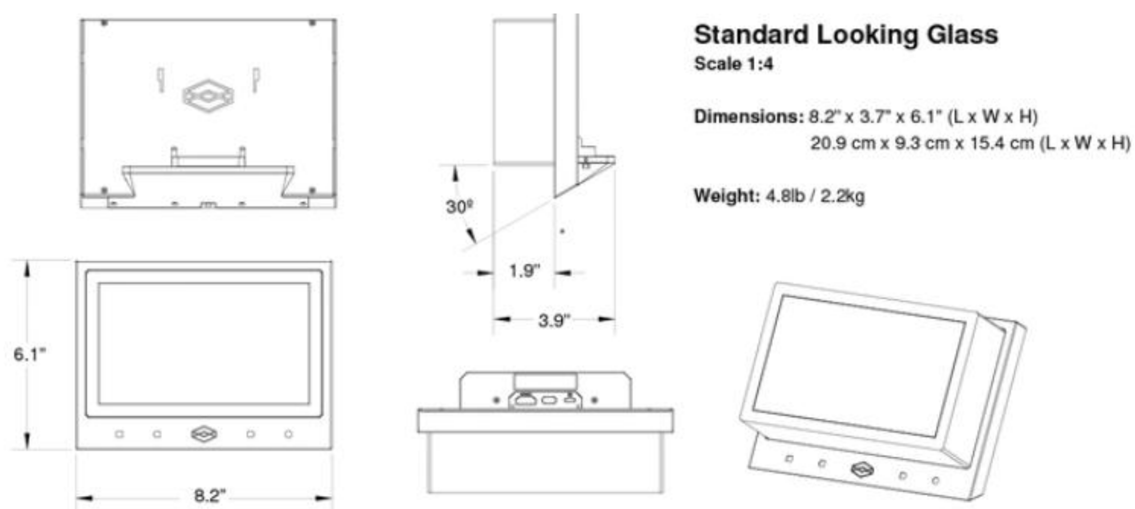


Imagen 4.1.1.2. *Escala, dimensiones y peso de la versión de 8.9”.*

Kit de desarrollo de 15.6”/15.6” Pro

	Kit de desarrollo de 15.6”	**Kit de desarrollo de 15.6” Pro**
Imagen	"Engine" by Sketchfab user spoutnix	"Engine" by Sketchfab user spoutnix
Visión general	Para clientes comerciales que buscan una pantalla holográfica de escritorio premium que se pueda conectar su propio PC o a un PC separado.	Para clientes comerciales y empresariales que desean una solución holográfica totalmente integrada. Con sensibilidad táctil, una computadora incorporada y una pantalla táctil plegable de 7”.
Especificaciones del producto	-14.5”x6.9”x9.6” (L x W x H) -15.6” en diagonal (pantalla Lightfield) -18.6lbs / 8,44 kg -Resolución 3840px x 2160px (4k), 60 FPS	-14.5”x6.9”x9.6” (L x W x H) -15.6” en diagonal (pantalla Lightfield) -25lbs / 11,34 kg -Resolución 3840px x 2160px (4k), 60 FPS
Interfaz táctil	No	Si

Pantalla inmersiva 8k

A 4x la escala de cualquier otra cosa en el mercado, con 33,2 millones de píxeles incomparables con una gama de colores de más de mil millones de cuentas 60Hz, esta pantalla es la pantalla más grande y de mayor resolución del mundo [4].

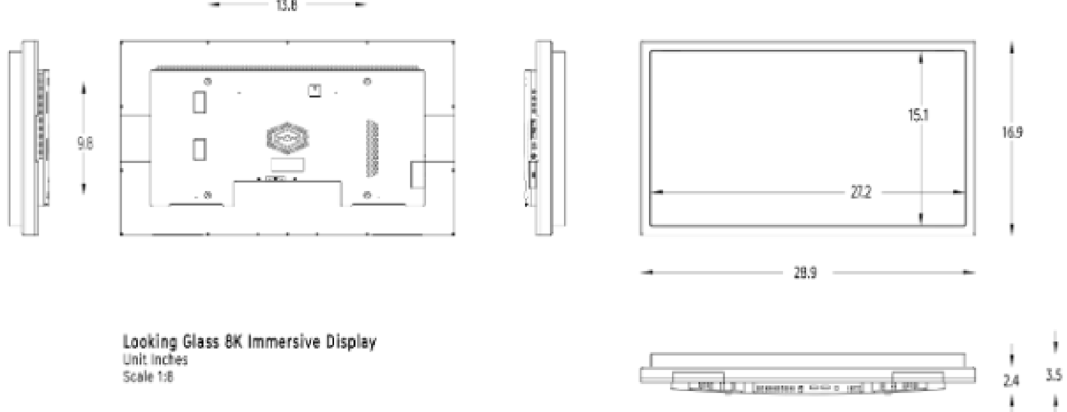


Imagen 4.1.1.3. *Escala y dimensiones de la Pantalla inmersiva de 8K.*

Cómo funciona:

Looking Glass es una combinación de pendiente de patente de campo de luz y tecnologías de visualización volumétrica dentro de un nuevo tipo de pantalla que da vida a al contenido virtual en 3D a todo color hasta 60 fps.

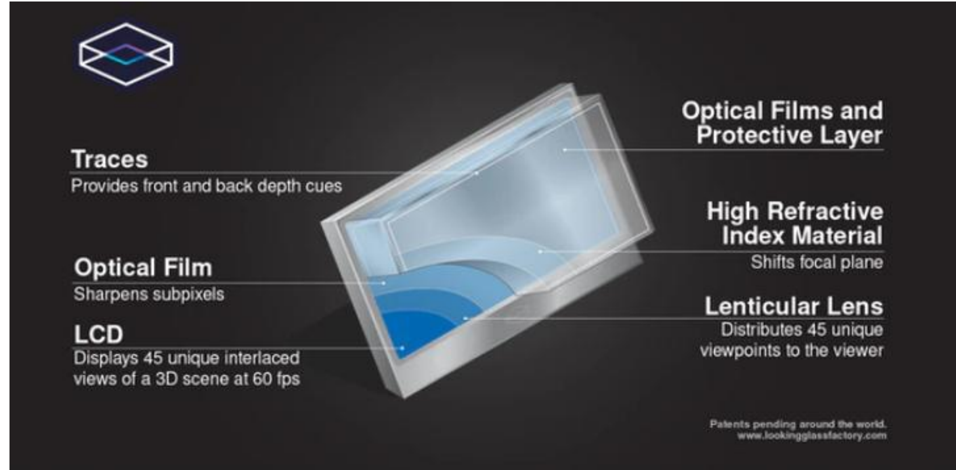


Imagen 4.1.1.4. *Componentes de la estructura del Holographic Display.*

El LookingGlass genera 45 vistas distintas de una escena tridimensional para que varias personas de pie alrededor de un display holográfico puedan ver diferentes partes de la escena en el mundo real.

A medida que se mueve en el alrededor del espejo, sus ojos están expuestos a diferentes conjuntos de información en 3D, creando una experiencia 3D en vivo para múltiples personas al mismo tiempo. Mientras que los VR producen dos vistas estereoscópicas de una escena tridimensional, esa escena solo puede ser experimentada por una persona a la vez. El Looking Glass, en comparación, genera 45 vistas de un mundo, lo que permite que el contenido que vive en un Looking Glass sea visto e interactuado por varias personas a la vez [5].

Qué se necesita para trabajar [5]

La versión con la que vamos a trabajar viene con los siguientes elementos:

- Display holográfico de 8.9"
- Cable de alimentación USB-C
- Cable HDMI

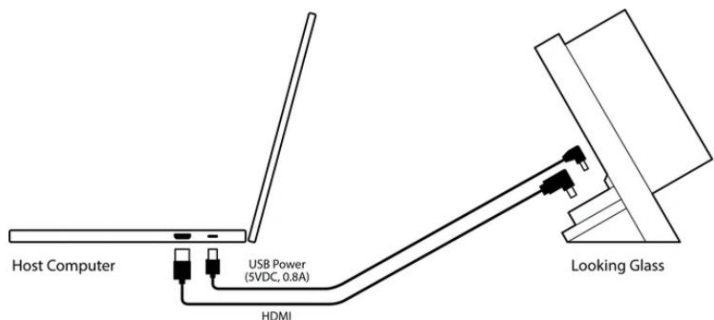


Imagen 4.1.1.5. *Conexión de PC y Looking Glass.*

Looking Glass admite varias configuraciones de hardware, incluido el MacBook Pro "Core i5" 2.7. Sin embargo, para una mejor experiencia visual, se sugieren las siguientes especificaciones:

- **Sistema operativo:** Windows 10 de 64 bits
- **CPU:** equivalente a Intel Core i5 o superior
- **Tarjeta gráfica *:** equivalente a Nvidia GTX 1060 o superior
- **RAM:** 4 GB o superior
- **Almacenamiento interno:** 128 GB o superior

(*El LookingGlass sólo funciona con tarjetas gráficas que pueden producir 2560x1600 o más.)

Unity Holoplay SDK:

Estos son algunos de los distintos tipos archivos (scripts, prefabs, escenas de ejemplo, etc.) que contiene el SDK del Looking Glass al importar a Unity [6]:

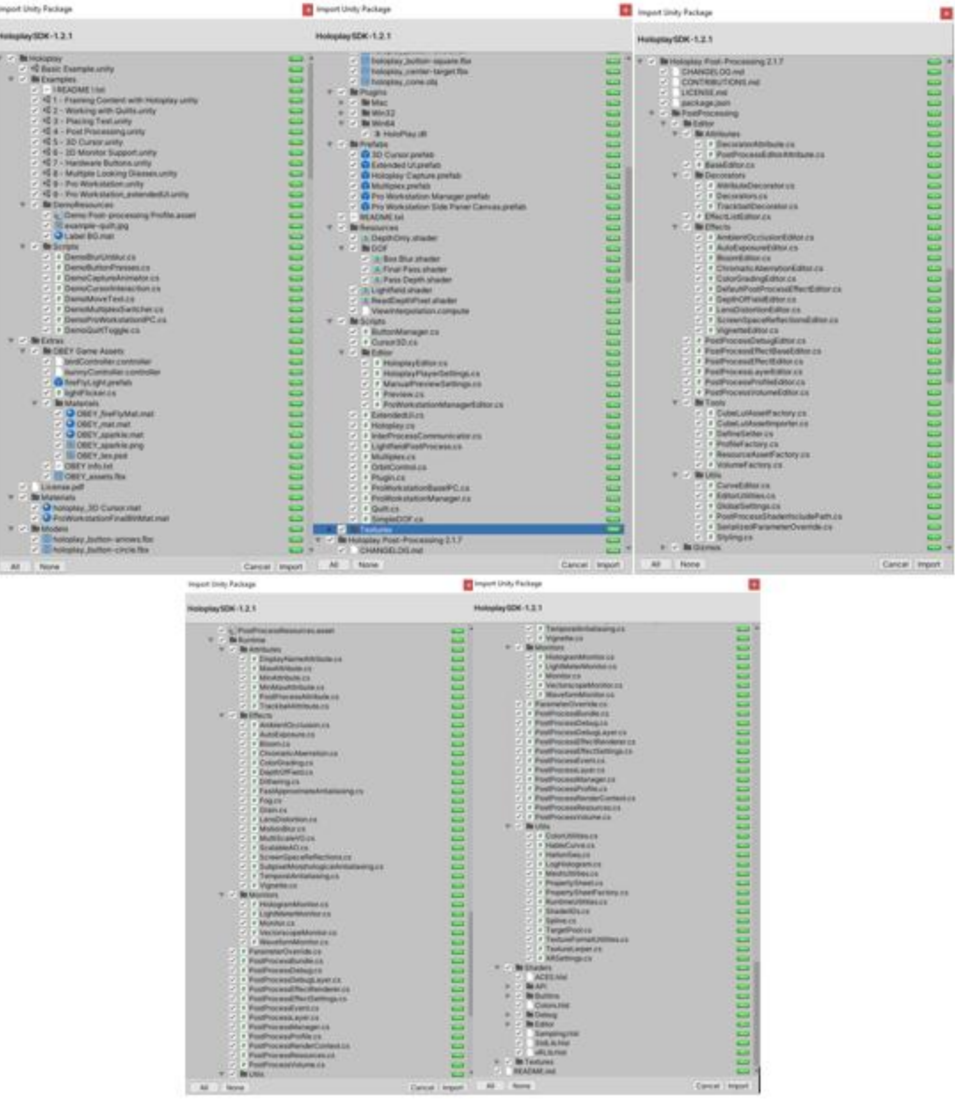


Imagen 4.1.1.6. *Archivos que contiene el HoloplaySDK.*

4.1.2 Leap Motion

La compañía LeapMotion fue fundada en el año 2010 por Michael Buckwald y David Holz. El primer dispositivo comercial fue lanzado por primera vez en Julio del 2013 bajo el nombre “The Leap”. Sus inicios y primeros prototipos se remontan al año 2008, cuando su cofundador David Holz estaba estudiando un doctorado en matemáticas. En la siguiente imagen podemos observar la evolución de los prototipos durante los años:

Imagen 4.1.2.1. *Evolución prototipos Leap Motion.*

El prototipo que usaremos en el proyecto es el actual, que corresponde al diseño final para el consumidor del diciembre de 2012:

Imagen 4.1.2.2. *Prototipo actual Leap Motion.*

Se va a distinguir y analizar a continuación el hardware y el software del dispositivo Leap Motion, para una mejor estructuración y entendimiento del funcionamiento del mismo. Para ello, se explicarán las características físicas y su SDK (Software Development Kit). Además de explicar cómo está constituido este SDK, se explicarán las facilidades que incorpora para diseñar aplicaciones en tiempo real, y la información que puede obtener de dedos y manos en cada *frame* recibido.

Hardware

Se trata de un pequeño dispositivo con forma rectangular. Tiene unas dimensiones de 75x22x11 mm (largo, ancho y alto) y un peso aproximado de 50gr. Su precio aproximado es de 90€. El dispositivo tiene su alimentación por USB 2.0 o 3.0 a través del cual manda la información al driver del PC al que se está conectado. Dependiendo de si se usa el USB 2.0 o 3.0 y del rendimiento del PC, se tendrá diferentes velocidades en la transmisión de la información. Pero puede permitir un *framerate* de hasta 300 fps (frames per second) con una precisión de 0.01mm [10].

En la siguiente figura se puede observar la zona de cobertura del dispositivo. Como se puede ver, esta zona es una semiesfera de 61 cm de radio:

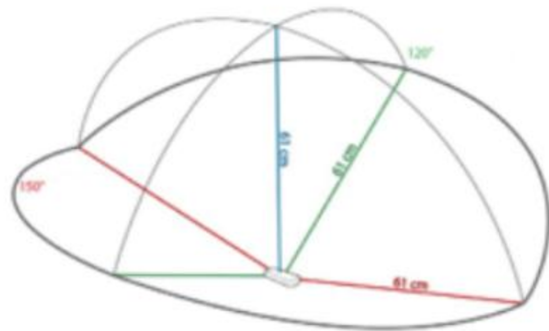


Imagen 4.1.2.3. *Zona de cobertura de Leap Motion.*

Esta zona depende del ángulo de visión de las lentes de las cámaras y de la intensidad máxima que puede entregar la conexión USB a los LEDs. A su vez, el ángulo de visión depende de la distancia focal y del tamaño del sensor de la siguiente forma, donde *d* es la diagonal del sensor y *f* la distancia focal:

$$\alpha = 2 \cdot \arctan\left(\frac{d}{2 \cdot f}\right)$$

Imagen 4.1.2.4. *Fórmula para el cálculo del ángulo de visión.*

Tanto el ángulo de visión horizontal de Leap Motion como el vertical son de 150,92°. Estos ángulos delimitan la zona de interacción.

En la API del se define una zona de trabajo llamada “Interaction Box” por un volumen de 110.55 mm de altura x 110.55 mm de anchura x 69.43 mm de profundidad, que varía sus dimensiones dependiendo de donde se encuentre el objeto a rastrear. Esta es la zona en la que se marca el centro del sistema de coordenadas cartesiano de Leap Motion. Desde el driver del dispositivo se puede configurar la altura a la que se encontrará el centro de esta zona de interacción. Esta altura puede estar entre 7 y 25 cm desde el dispositivo.

Este Interaction Box será el espacio de trabajo del dispositivo. Hace un seguimiento de las manos, dedos y objetos "*pointables*" (similares a un dedo, aunque haciendo distintas pruebas en el laboratorio, parece ser que han eliminado el reconocimiento de objetos como lápices) situados dentro de este espacio con una precisión muy alta. Aquí se encuentra la gran diferencia con respecto a la Kinect, dispositivo el cual hace seguimiento de todo el cuerpo en dimensiones de una habitación.

Internamente el dispositivo cuenta con 2 cámaras, 3 LEDS de infrarrojos, un microcontrolador y un controlador USB:

- Cada **cámara** cuenta con un sensor monocromático CMOS sensible a la luz infrarroja, capaz de trabajar a una velocidad de hasta 300 fps.
- Los **LEDs** iluminan la Interaction Box por inundación. Esta iluminación va variando para asegurar una igualdad en resolución de imagen. Están separados por barreras de plástico asegurando así una iluminación uniforme.
- El **microcontrolador**es un circuito integrado que funciona como BIOS. Controla en dispositivo y capta y envía la información de los sensores al driver o controlador del PC.
- El **controlador USB** es de alta velocidad y permite que el PC reconozca al dispositivo.

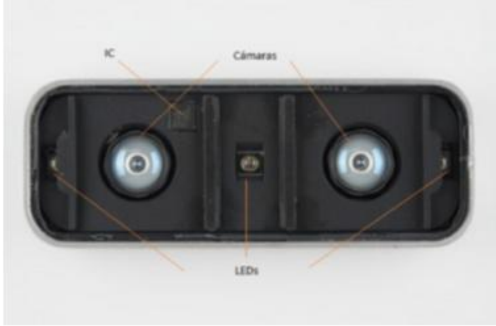


Imagen 4.1.2.5. *Componentes del dispositivo Leap Motion.*

En la parte superior, cuenta con un plástico negro que funciona como filtro óptico dejando transmitir únicamente la luz infrarroja.

El uso de la CPU que hace el dispositivo es relativamente bajo para la alta precisión que se obtiene. Llega a ser de aproximadamente 12% tras cierto tiempo empleando *tracking* de manos y dedos con un procesador i5. En la página oficial podemos encontrar los requisitos mínimos para el uso del dispositivo:

- Windows 7 + Mac OS X 10.7 o superiores
- Procesador AMD Phenom II o Intel Core i3/i5/i7
- 2 GB RAM
- Puerto USB 2.0/3.0
- Conexión a internet

Software y SDK

Leap Motion SDK es soportado por los sistemas operativos Windows, Macintosh y Linux. Cada vez son más las aplicaciones que aparecen en el mercado cuyo control viene dado por el dispositivo Leap Motion. La empresa cuenta con un App Store el cual se conecta con el "Leap Motion App Home" que se añade como aplicación de escritorio en el ordenador al instalar el controlador del dispositivo. Aquí se pueden descargar aplicaciones gratuitas o de pago que son subidas por los desarrolladores. Son muchas las posibilidades que ofrece, por ejemplo: control del escritorio Windows, jugar a videojuegos, instrumentos de aire, etc. Además, están ofreciendo diversos plugins que permiten por ejemplo que sea posible navegar por Chrome usando el Leap Motion, o moverse por Google Earth. La interacción con imágenes médicas anatómicas en 3D, integración en smartphones, integración con gafas de realidad virtual...son algunos de los objetivos con vista al futuro de los desarrolladores que ya se pueden contemplar. Por tanto, como se puede deducir, el dispositivo cuenta con un SDK muy flexible y con el que se pueden desarrollar aplicaciones muy diversas.

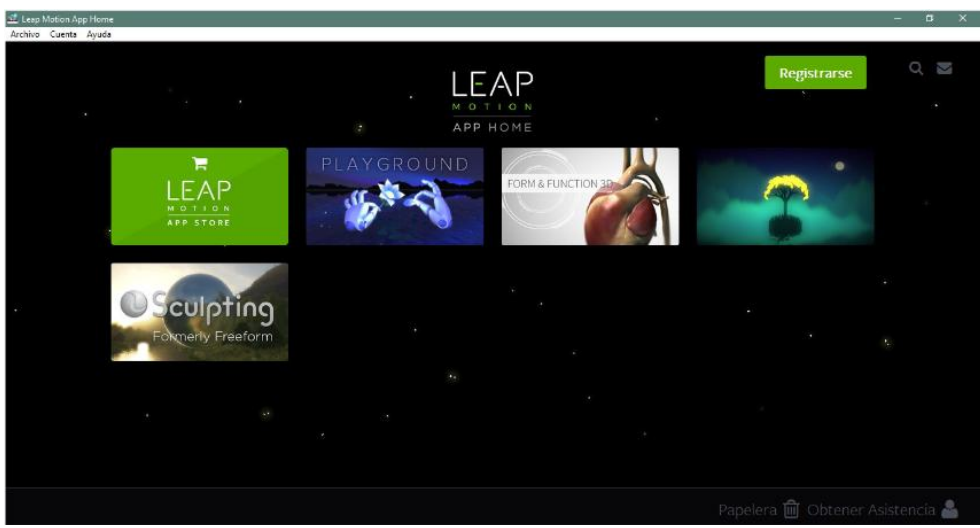


Imagen 4.1.2.6. *Página principal de la Leap Motion App Home.*

Para la puesta en marcha del dispositivo, es necesario descargar el "LeapDeveloperKit" para el correspondiente sistema operativo (Windows en el caso de este TFG) directamente de la página oficial de Leap Motion. En el paquete descargado se incluye un instalador del controlador y una carpeta que contiene el SDK. Al instalar el controlador, será instalado también el "Leap Motion App Home" citado anteriormente. El controlador cuenta principalmente con dos aplicaciones:

- **Panel de control de Leap Motion:** permite cambiar prácticamente cualquier opción del dispositivo. Por ejemplo: área de interacción del dispositivo, mecanismo de ahorro de energía, recalibrar el dispositivo, resolución o informar acerca de problemas, configuración de rastreo, actualizaciones, etc. Además, cuenta con un apartado "Acerca de" donde se puede encontrar información relativa a la versión del SDK y enlaces a las páginas web oficiales.

- **Visualizador:** muestra un espaciográfico3Dcon un sistema de coordenadas cartesianas(x, y, z). El eje “X” pertenece a la parte horizontal, el “X” va verticalmente desde el sensor y el “Z” perpendicular a la pantalla. En este gráfico 3D se monitorizan las manos, dedos y elementos “*pointables*” (aunque, por lo dicho anteriormente, elementos como lápices ya no se reconocen) siendo representadas y desplazándose tal y como el usuario las mueve. Además, ofrece la posibilidad de visualizar información relativa al muestreo, coordenadas, dibujar la Interaction Box, cambiar el tipo de representación, etc.

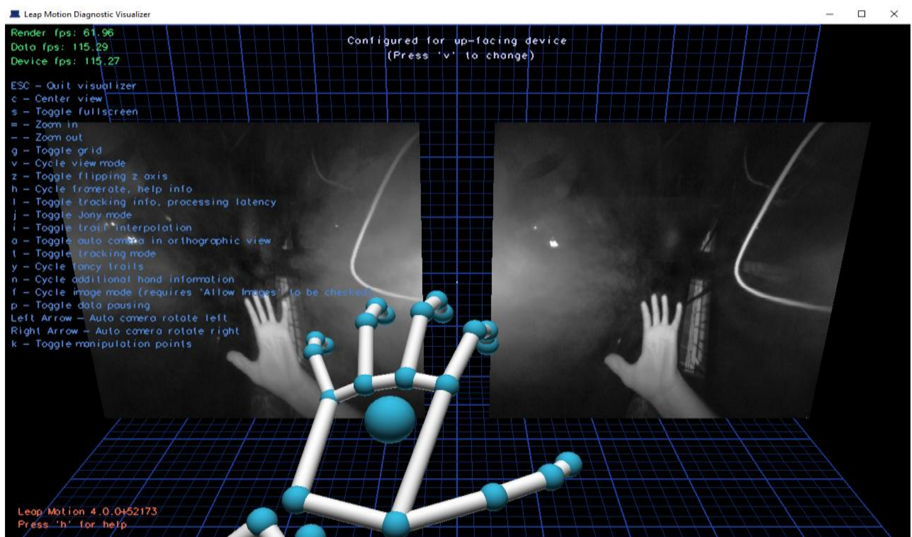


Imagen 4.1.2.7. *Visualizador de diagnósticos de Leap Motion.*

Para crear la aplicación que se quiere desarrollar en este trabajo, es fundamental conocer el SDK de Leap Motion. Sus clases, lenguajes, modo de trabajo, etc. Este SDK permite una gran variedad de lenguajes de programación entre los que están C++, C#, Unity (estos dos últimos son en los que hemos empleado en este proyecto), Objetive-C, Java, Java Script, Python y Unreal Engine. Como bien se ha dicho anteriormente, el dispositivo crea un espacio 3D y tiene la información correspondiente a manos, dedos y objetos “*pointables*”. Para posicionarlos usa un sistema de coordenadas cartesianas (x, y, z) centrando el origen en el medio del dispositivo.

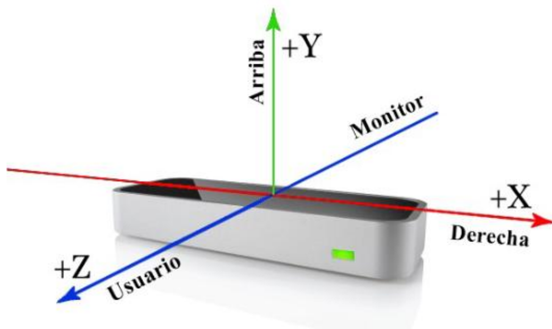


Imagen 4.1.2.8. *Sistema de coordenadas Leap Motion.*

A continuación, se van a describir los objetos y clases esenciales de la API (Application Programming Interface)del dispositivo que van a ser fundamentales para poder programar los distintos métodos desarrollados. Además, se detallarán sus propiedades más características de las cuales algunas serán usadas en esta aplicación. Es importante conocer qué información es capaz de capturar el dispositivo para conocer las limitaciones y posibilidades que se tendrán a la hora de diseñar el sistema que se quiere implementar [7][8][9].

FRAMES

Es el contenedor principal de la información que captura Leap Motion. Cada *frame* contiene en un instante determinado la siguiente información:

- ***Hand:*** información de las manos, brazo y herramientas "*pointables*" (como dedos, por ejemplo).
- ***Fingers:*** información de los dedos.
- ***Bones:*** información de los huesos de los dedos.
- Posición de las articulaciones.
- Imágenes de las cámaras.

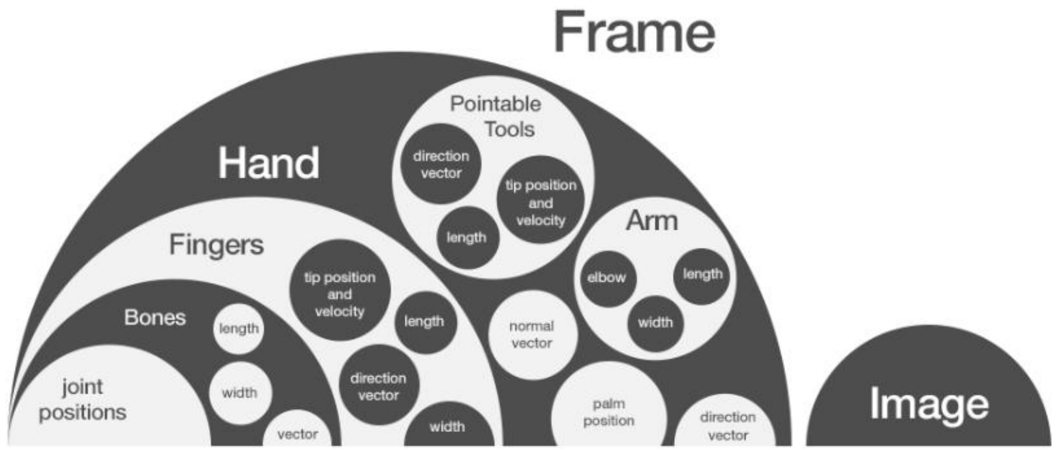


Imagen 4.1.2.9. *Información que contiene cada "frame".*

Hay que resaltar que cada objeto se asocia con otro objeto superior, siendo el más superior el *Frame.* Esta jerarquía es uno de los aspectos más importantes a tener en cuenta. Así, para poder acceder por ejemplo a *Fingers* o *Pointable Tools*, se debe hacer a través del objeto *Hand* dentro de un *Frame* determinado.

Además de esta información más relativa a manos y dedos, cada *Frame* también contiene información de identificación de *Frame, timestamp, etc.* El software analiza la información del *Frame* actual y el anterior y sintetiza el eje de rotación, ángulo de rotación, matriz de rotación, factor de escala y translación para la monitorización de las manos. Por tanto, se basa en el cambio continuo de dicha información.

HANDS

Hand es la principal entidad de información (después de *frame*) ya que como se ha visto, a través de ella se accederá a los objetos inferiores. El controlador tiene un modelo de la mano humana y asocia la información de sus sensores con este modelo. Esto permite hacer un seguimiento de la posición de los dedos hasta cuando el dedo no es completamente visible. Puede distinguir entre mano derecha e izquierda. Para cada *hand*, se tiene los siguientes atributos:

- ***Palm Position:*** posición del centro de la palma en milímetros, formado por un vector. Atributo usado en los métodos de Tableta moviendo el plano de dibujo con Leap Motion y Leap Motion y bola en el espacio.
- ***Palm Velocity:*** velocidad de la palma en milímetros/segundo.
- ***Palm Normal:*** vector perpendicular a la palma.
- ***Direction:*** dirección de la mano formada por un vector.
- ***grabStrength, pinchStrength:*** información de la esfera que describe la postura de la mano.
- ***Motion Factors:*** factores del movimiento como escala, rotación y translación.

Se puede acceder a la orientación de la mano usando la dirección de la mano y los vectores normales. Su postura viene dada por una información de una esfera que "encaja" en la mano:

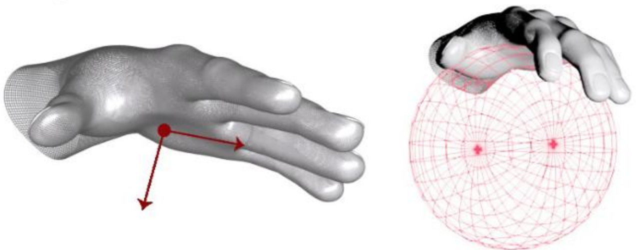


Imagen 4.1.2.10. *Algunos atributos del objeto "Hand".*

FINGERS (pointables)

Tanto *Finger* como *Tool* se encuentran dentro de la clase *Pointable*, ya que ambos son objetos que pueden considerase "*pointables*". Esta aplicación se centra solo en *Finger*, ya que, por lo dicho con antelación, ya no se detectan objetos que no sean parte del cuerpo humano. A través de *Hand* se accede a los *Fingers* asociados con esa determinada mano, la cual está disponible en un *Frame* concreto. Al acceder a los dedos de una mano, se devuelve un vector que contiene los 5 dedos de la mano. Será posible acceder al dedo en concreto que sea requerido bajo la función Finger::type(), o si tenemos una instancia del objeto *Hand,* usando los valores de la siguiente enumeración:

- TYPE_THUMB = = 0 (Dedo pulgar)
- TYPE_INDEX = = 1 (Dedo índice)
- TYPE_MIDDLE = = 2 (Dedo corazón)
- TYPE_RING = = 3 (Dedo anular)
- TYPE_PINKY = = 4 (Dedo pequeño)

Los atributos que se tienen del objeto *Finger* son los siguientes:

- ***Tip position:*** vector que determina la posición del extremo del dedo en milímetros.
- ***Tip velocity:*** velocidad del extremo del dedo en milímetros/segundo.
- ***Stabilized tip position:*** vector de la posición estabilizada del extremo del dedo usando la velocidad y la posición en el *frame* anterior.
- ***Direction:*** vector que determina la dirección del dedo.
- ***Length:*** longitud del dedo.
- ***Width:*** anchura del dedo.

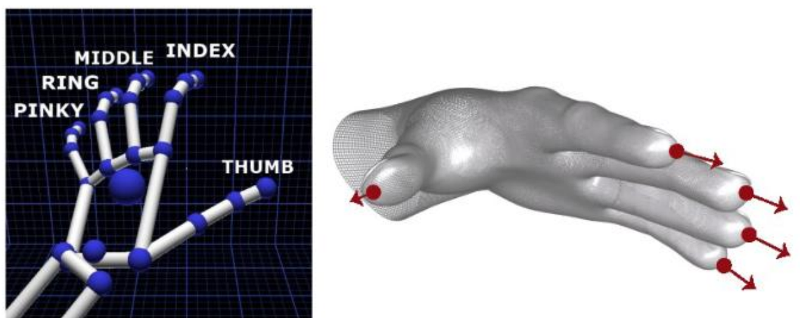


Imagen 4.1.2.11. *Dedos nombrados con sus respectivas direcciones.*

BONES

De forma similar a como se obtiene la información de los dedos en el apartado anterior, se accede a la información de los huesos de la mano. Así, para acceder a *Bone*, habrá que hacerlo a través de *Frame/Hand/Finger*. Y análogamente a los dedos, se devuelve un vector que contiene los cuatro huesos para cada dedo. En este caso, será posible acceder al hueso en concreto que sea requerido usando la función Bone::type() siendo su enumeración, ordenados de la base al extremo del dedo, la siguiente:

- TYPE_METACARPAL = = 0 (Hueso que conecta con la muñeca).
- TYPE_PROXIMAL = = 1 (Hueso de la base del dedo).
- TYPE_INTERMEDIATE = = 2 (Hueso entre el extremo y la base del dedo).
- TYPE_DISTAL = = 3 (Hueso en el extremo del dedo).

NOTA*: *El dedo pulgar no tiene hueso metacarpiano (type == 0), así que contiene un valor de longitud igual a cero.*

Los atributos que posee el objeto *Bone* son los siguientes:

- ***Direction:*** dirección del hueso.
- ***Length:*** longitud del hueso.
- ***Width:*** anchura del hueso.
- ***Center:*** punto medio del hueso.

Resumiendo todos los apartados anteriores, el resultado final que obtenemos cuando la Leap Motion detecta la mano es la siguiente:

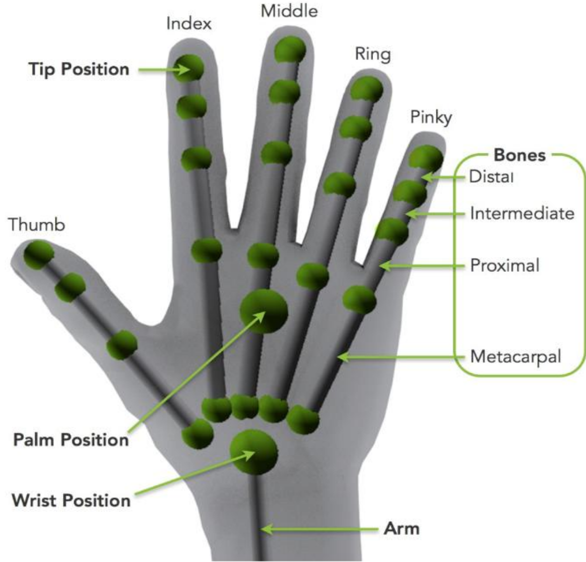


Imagen 4.1.2.12. *Resultado final de la mano con todos los atributos descritos.*

4.1.3 Tableta Wacon

La tableta que he empleado a la hora de trabajar en este proyecto ha sido la tableta Wacom Intuos S, concretamente la versión con Bluetooth integrado. Para comprender el funcionamiento de esta tableta, al igual que en el apartado anterior de la Leap Motion, voy a explicar tanto las especificaciones referentes al hardware y el software que requiere el dispositivo para poder trabajar con ello.

Hardware

Estas son las especificaciones técnicas del producto [12][13]:

- **Modelo:** Wacom CTL-4100WL
- **Dimensiones:** 200 x 160 x 8,8 mm (ancho x alto x grosor)
- **Área activa:** 152 x 95 mm (ancho x alto)
- **Peso:** 250 gramos
- **ExpressKeys:** 4 teclas personalizables
- **Portalápices:** 1
- **Lápiz:** Wacom Pen 4K (LP-1100K)
- **Resolución:** 2540 LPI (líneas por pulgada)
- **Tasa de informe (velocidad de lectura):** 133 PPS
- **Sensibilidad:** 4096 niveles de presión
- **Peso del lápiz:** 11,2 gramos
- **Altura de lectura del lápiz:** 7 mm
- **Interfaz:** USB/Bluetooth
- **Colores:** negro y pistacho
- **Batería:** integrada
- **Autonomía:** 15 horas aproximadamente
- **Tiempo de carga:** 3-4 horas
- **Sistemas operativos:** compatible con Windows 7 o posterior/ macos X 10.11 o posterior/ Android 6.0 o posterior

Imagen 4.1.3.1. *Tableta Wacom Intuos S Bluetooth.*

Driver

Para poder trabajar con la tableta, lo primero que debemos hacer es descargar el driver. Para ello, nos dirigimos a la página principal de Wacom, buscamos el modelo correspondiente y descargamos el driver dependiendo del sistema operativo (Windows 10 en nuestro caso).

Una vez descargado el driver, podemos personalizar tanto la configuración de la tableta, ya sea los *ExpressKeys* y/o la pantalla activa de ésta, y el lápiz dependiendo del programa con el que estemos trabajando:

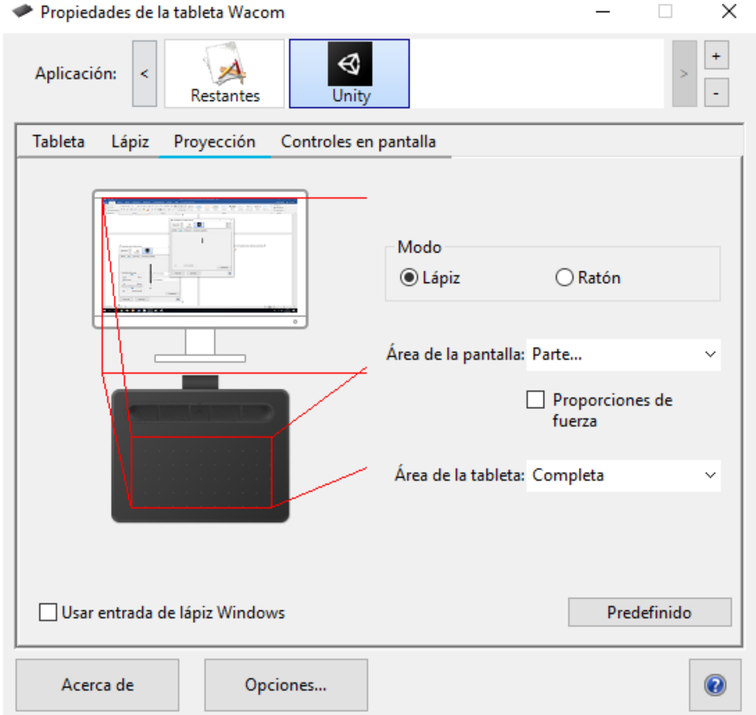


Imagen 4.1.3.2. *Configuración de la proyección de la tableta.*

Me gustaría comentar que en la imagen 4.1.3.14 la proyección no coincide el lado derecho de la tableta con la pantalla del ordenador. Eso se debe a que el display holográfico está conecta y podemos acceder a él moviendo el ratón hacia la derecha de la pantalla del ordenador.

4.2 A nivel de software

4.2.1 Unity

Como he mencionado anteriormente, el programa que he empleado para desarrollar el proyecto es Unity, concretamente la versión 2019.4.1f1, ya que era la última versión disponible cuando comencé a realizar el proyecto. Para poder cargar los distintos métodos y un proyecto Unity inicial que empleé para descubrir el funcionamiento y los ejemplos del SDK del display, he usado Unity Hub, programa gratuito que dispone Unity. La versión empleada para este último programa es la 2.3.2:

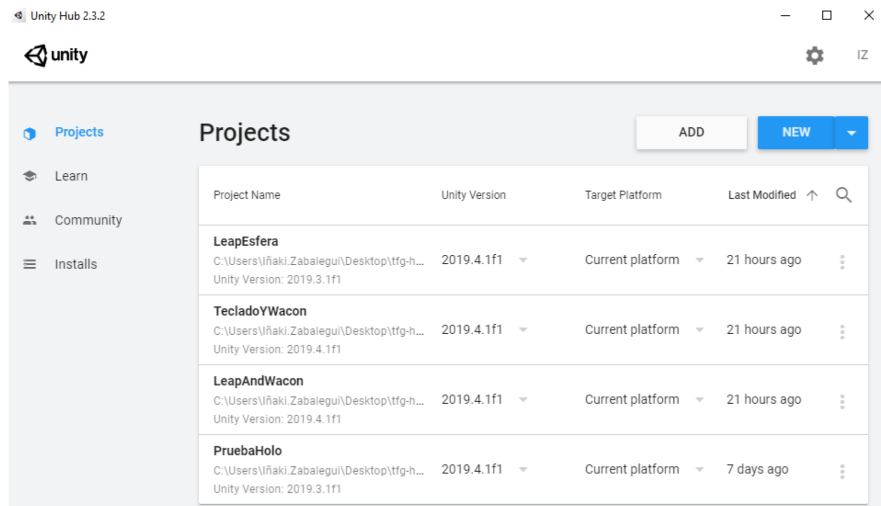


Imagen 4.2.1.1. *Captura de Unity Hub 2.3.2.*

Para poder desarrollar el proyecto con el *Looking Glass*, una vez creado el proyecto en Unity, lo primero que debemos hacer es importar el SDK que la empresa desarrolladora ha desarrollado para poder trabajar con el *display* en Unity (una vez creado el proyecto, obvio). Para los métodos que usamos la Leap Motion, obviamente importamos también su SDK [6][14].

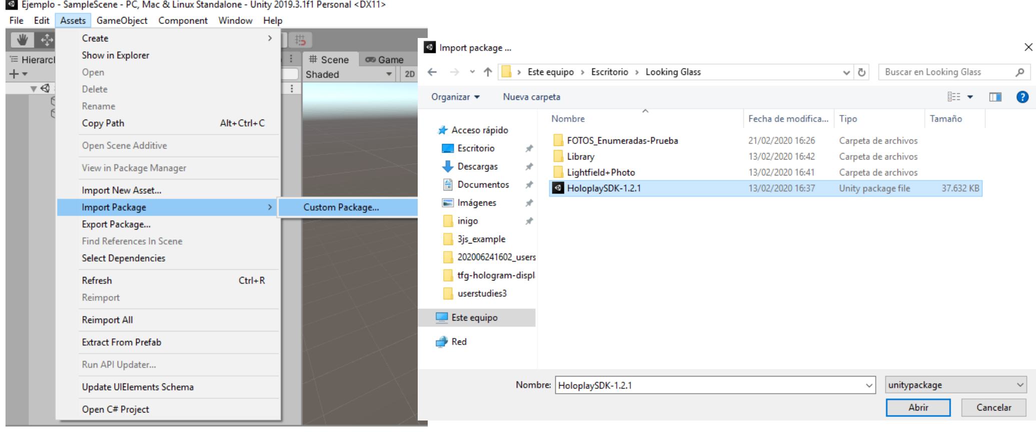


Imagen 4.2.1.2. *Proceso de selección del HoloplaySDK.*

Para acabar con la explicación de Unity, voy a explicar brevemente algunos de los elementos de este programa: **jerarquía** (tiene los elementos de componen la escena), **configuración** de un objeto (como el nombre indica, permite configurar un objeto dependiendo de sus prestaciones: posición, rotación, escala, material, scripts, etc.), **escena**(contiene el espacio 3D con los elementos de la jerarquía), y por último la **ventana del proyecto**, el cual he seguido con la misma estructura y organización en todos los métodos**.** Decir que Unity también tiene un **Asset Store** para poder obtener modelos gratuitos, pero no hemos usado para este proyecto.

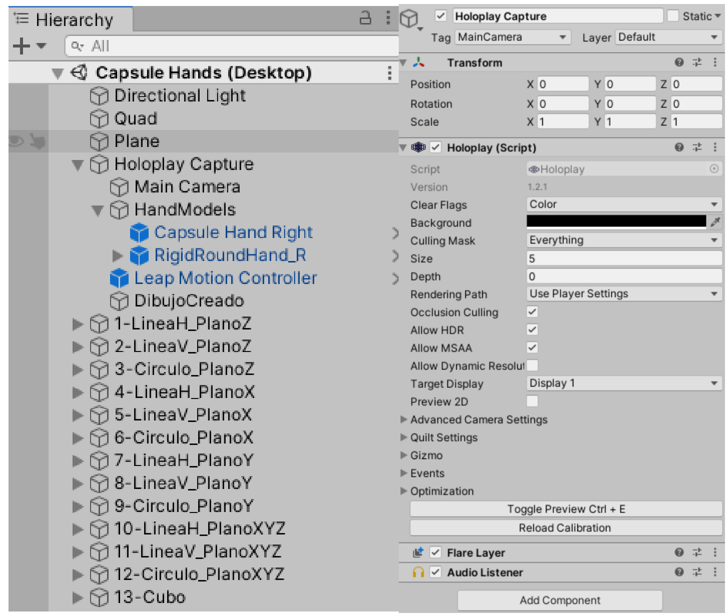


Imagen 4.2.1.3. *Jerarquía de la escena y configuración de HoloplayCapture.*

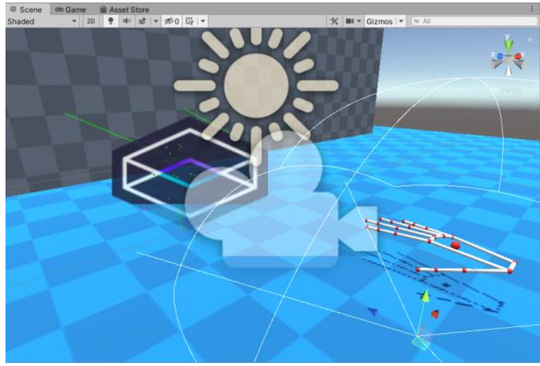


Imagen 4.2.1.4. *Escena del proyecto.*

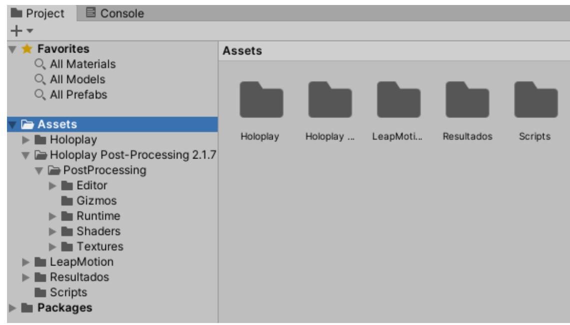


Imagen 4.2.1.5. *Ventana del proyecto y sus directorios.*

5-Pruebas preliminares

Como he explicado al comienzo del apartado referente al Looking Glass Holographic Display (4.1.1), la empresa desarrolladora, Looking Glass Factory, ha desarrollado distintas aplicaciones software que permite trabajar y explotar al máximo las funcionalidades que ofrece el display. La primera con la que me familiaricé fue la ya mencionada **Library**:

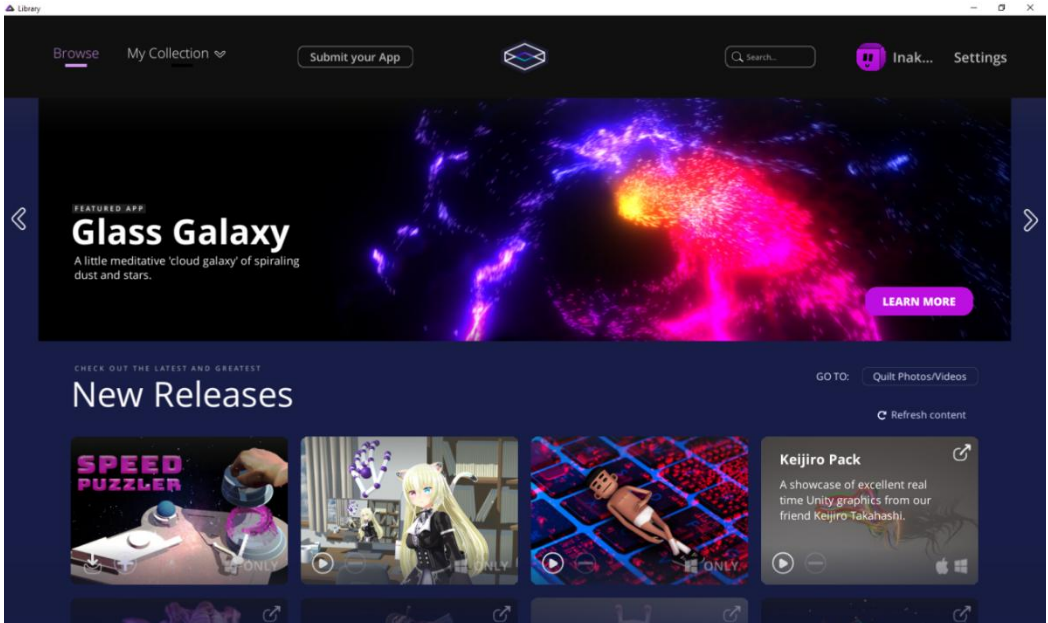


Imagen 5.1. *Pantalla de inicio de Library.*

Un ejemplo que me gustaría destacar es el de la rana, el cual nos muestra una rana y tiene integrado también la Leap Motion. Si no ponemos la mano, la rana repite los gestos periódicamente, y en caso de poner la mano, lo que aparece en la escena es una antorcha, la cual la rana sigue con sus ojos. Es increíble lo bien hecha que está la rana, ya que al ejecutar el ejemplo en la calidad de gráficos en ultra da la total sensación real de que se encuentra una rana encerrada en el display.

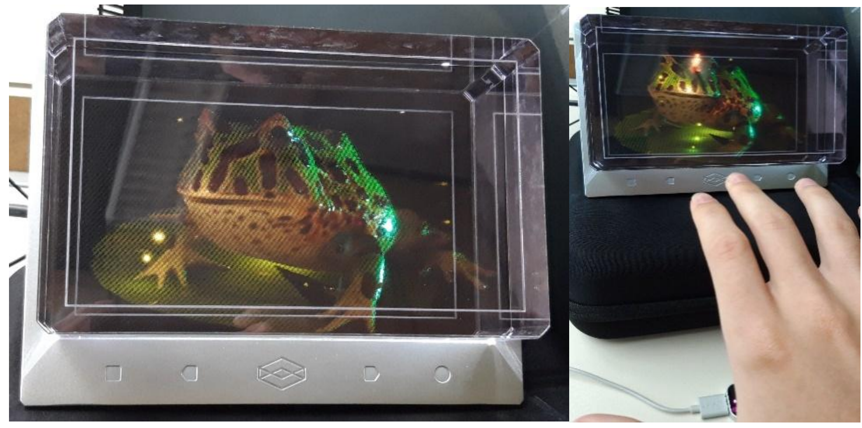


Imagen 5.2. *Ejemplo de la rana de Library sin y con la Leap Motion.*

Pero el ejemplo que más me ha ayudado ha sido el del texto en movimiento. Este ejemplo consiste en un texto que está en constante movimiento hacia adelante y hacia atrás, o bien podemos manejar dicho texto con las flechas de arriba y abajo. Gracias a este ejemplo he podido medir el límite del campo de visión que alberga el display, además de ver que los objetos más céntricos respectos a ese campo son los que mejor se aprecian, mientras que los que están muy cerca o muy lejos su nivel de visualización es peor:

Imagen 5.3. *Ejemplo del texto en movimiento del HoloplaySDK.*

Otra herramienta que he usado para entender mejor las distintas perspectivas que tiene el display es **Lightfield Photo**, la cual viene con unos modelos por defecto como una navaja suiza, un cubo rubik encima de un libro o un frutero con distintas frutas.

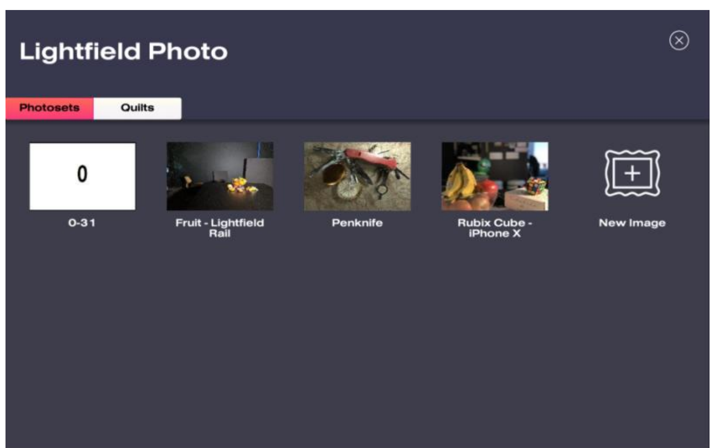


Imagen 5.4. *Pantalla de inicio de Lightfield Photo.*

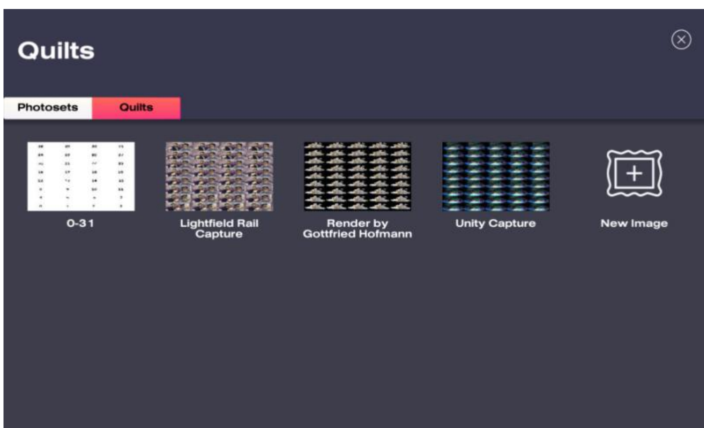


Imagen 5.5. *Apartado "Quilts" de Lightfield Photo.*

Este apartado de “*Quilts*” podemos entender mejor que, para mostrar un modelo 3D en el display, lo que realmente se utilizan son una composición de imágenes que forman una, en la que, como veremos en las siguientes imágenes, están ordenadas por filas y columnas. Para ver qué imagen se visualiza en el dispositivo según la perspectiva del usuario, lo primero que hice fue crear una imagen compuesta por números de 0 a 31 ordenado por filas y columnas:

28	29	30	31
24	25	26	27
20	21	22	23
16	17	18	19
12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

Imagen 5.6. *Imagen formada por números de 0 a 31.*

Una vez creada la imagen, la importamos al programa de *Lightfield Photo* para después poder visualizarla:

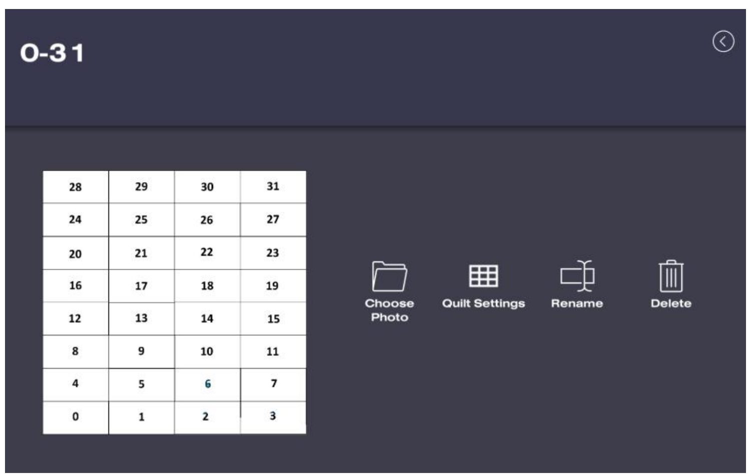


Imagen 5.7. *Ejemplo de números de“Quilts” en Lightfield Photo.*

Una vez llegados a este punto, ajustamos la imagen a sus características (32 vistas, 4 columnas y 8 filas):

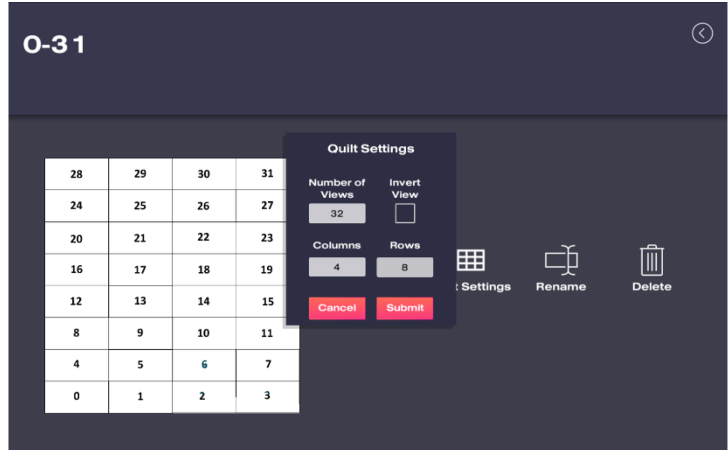


Imagen 5.8. *Ajuste del ejemplo de números en Lightfield Photo.*

Después de realizar los ajustes, pulsamos el botón de "submit" y veremos que dependiendo de la posición del usuario respecto al *Looking Glass*, el número que se observa es distinto. En las siguientes imágenes se aprecia mejor el resultado de las diferentes perspectivas:

Imagen 5.9. *Imagen 0 a 31 desde 3 puntos de vista diferentes.*

Tras haber probado esta última aplicación, lo que hice fue adentrarme en el mundo de **Three.js**, por lo que necesitaba instalar un servidor Apache para poder ejecutar el código Javascript. El programa que decidí instalar para ello fue **Xampp**, programa que pude conocer en una de las asignaturas de la carrera y me resultó de gran utilidad y fácil de usar.

En este punto, me dediqué a leer la documentación Javascript referente al Holoplay, concretamente en el apartado Holoplay.js de la documentación de la página oficial del *Looking Glass.* En esta página web podemos encontrar tanto la documentación necesaria para poder trabajar con este lenguaje y el *display*, además de unos tutoriales en los que se describe los pasos a seguir para obtener dicho resultado. Para comprobar que hemos seguido bien los pasos, encontramos la opción de ejecutar el resultado de cada fase haciendo click en el apartado "*Live example*" [11]:

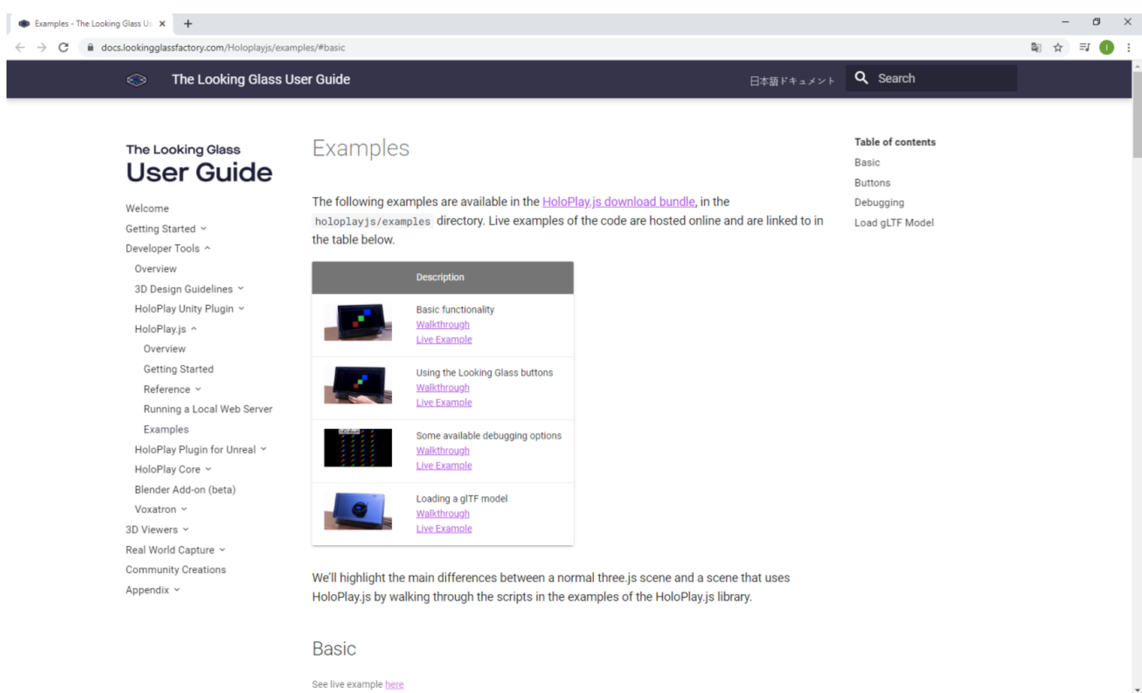


Imagen 5.10. *Página web con documentación y ejemplos Javascript para el display.*

Me gustaría destacar que el código está preparado de tal manera que el navegador en el que ejecutemos el código detecta si la pestaña se encuentra en el display o no. Además, como se puede observar en las siguientes imágenes, una vez que la pestaña de ejecución se encuentra en el display, podemos hacer click encima de la página para mostrarlo en pantalla completa:

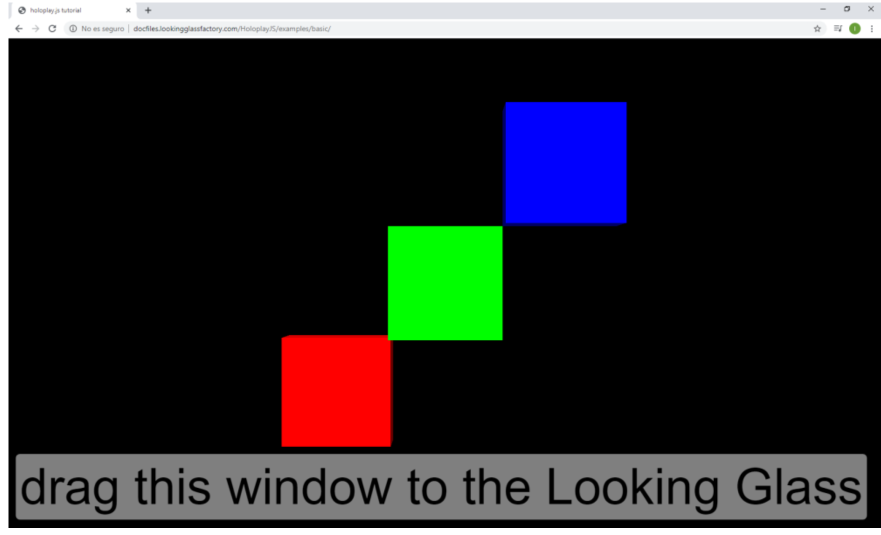


Imagen 5.11. *Ejecución del ejemplo básico en una pestaña del navegador.*

Imagen 5.12. *Ejecución del ejemplo básico en el Looking Glass.*

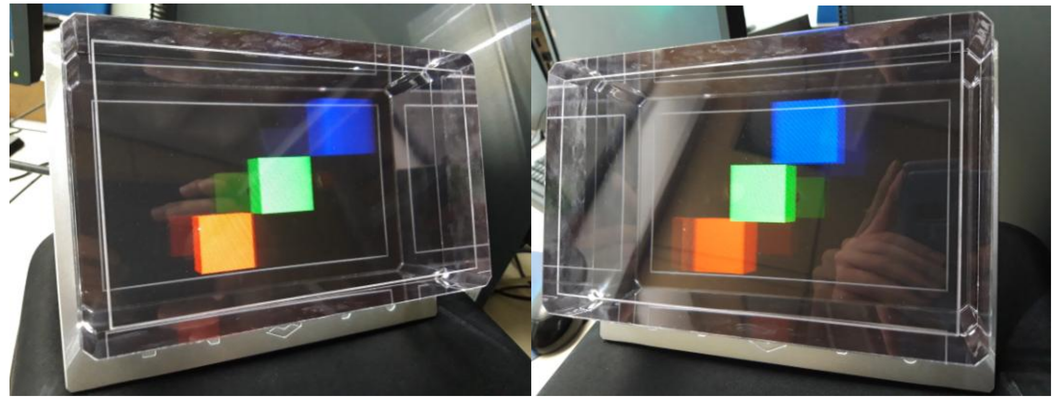


Imagen 5.13. *Dos puntos de vista diferentes del ejemplo.*

Como se puede apreciar en las imágenes 5.7 y 5.8, el código detecta cuando la pestaña del navegador que contiene la ejecución se encuentra en el display o no. Una vez en el display, pinchamos encima y se ejecuta en pantalla completa, como un ejemplo de Unity.

Después de haber probado el ejemplo que viene dado en la página oficial, lo que hice fue probar un ejemplo sencillo en el que se muestra un cubo que está constantemente rotando, junto con los ejes cartesianos. Dicho código se encuentra en el disco local C, concretamente en directorio '*C:\xampp\htdocs\JS\3js_example*'. Todos los ejemplos Javascript que queramos probar deben estar en el directorio '*htdocs'*, sino no se ejecutará el código referente al script.

```
<html>
    <head>
        <title>Ejemplo threejs sencillo</title>
        <style>
            body { margin: 0; }
            canvas { display: block; }
        </style>
    </head>
    <body>
        <script src="js/three.js"></script>
        <script type="module">
            // reference your own local copy of holoplay.js
            import * as HoloPlay from './holoplayjs/dist/holoplay.module.js';

            var scene = new THREE.Scene();

            var axesHelper = new THREE.AxesHelper( 500 );
            scene.add( axesHelper );

            var light = new THREE.DirectionalLight( 0xffffff );
            light.position.set( 0, 0, 1 );
            scene.add(light);

            var geometry = new THREE.BoxBufferGeometry(0.05, 0.05, 0.05);
            var material = new THREE.MeshLambertMaterial({color: new THREE.Color().setHSL(1/3, 1, 0.5)});
            var cube = new THREE.Mesh( geometry, material );
            cube.position.setScalar(1).multiplyScalar(0.05);
            scene.add( cube );

            // the holoplay camera should be used like a THREE.PerspectiveCamera
            const camera = new HoloPlay.Camera();

            // the holoplay renderer should act as your THREE.WebGLRenderer
            const renderer = new HoloPlay.Renderer();

            // add the renderer's canvas to your web page (it will size to fill the page)
            document.body.appendChild(renderer.domElement);

            var animate = function () {
                requestAnimationFrame(animate);
                cube.rotation.x += 0.01;
                cube.rotation.y += 0.01;
                renderer.render( scene, camera );
            };
            animate();
        </script>
        <canvas style="position: absolute; left: 0px; top: 0px; "></canvas>
    </body>
</html>
```

Imagen 5.14. *Código del ejemplo básico con el cubo en constante rotación.*

Una vez programado el código y almacenado en el directorio mencionado (‘*htdocs*’), si introducimos ‘*localhost*’ en la barra de direcciones, se cargarán los archivos que se encuentren el directorio ‘*htdocs*’, por lo que, para poder ejecutar el ejemplo desarrollado introducimos en la barra de direcciones la siguiente dirección: http://localhost/JS/3js_example/ . Decir que, el ejemplo (llamado *index.html*) al ser el único del directorio, no es necesario introducir también su nombre y extensión del archivo, pero en caso de introducirlo funcionaría igualmente.

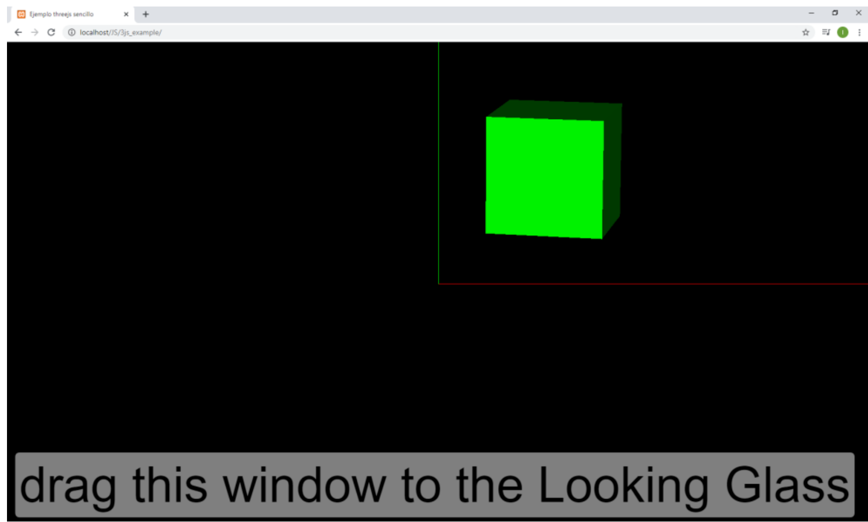


Imagen 5.15. *Ejecución del ejemplo en el navegador.*

Imagen 5.16. *Ejecución del ejemplo en el Looking Glass.*

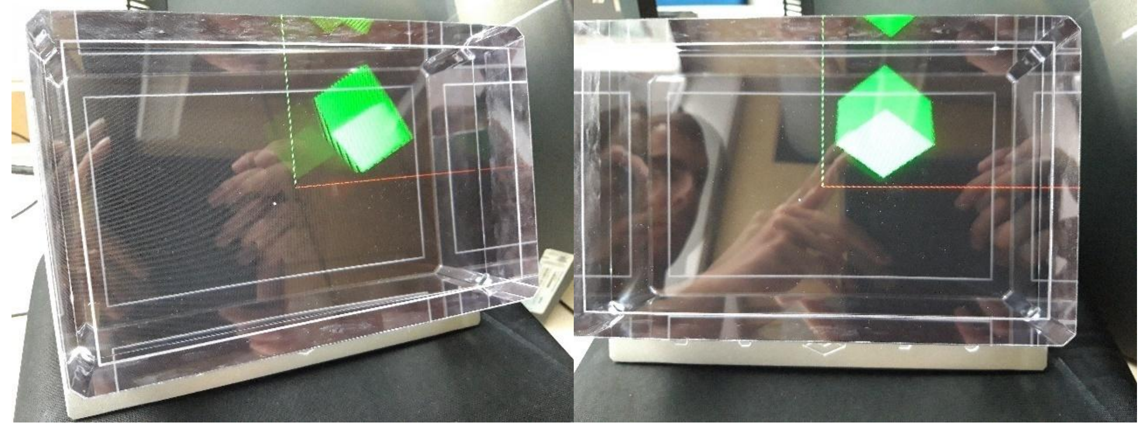


Imagen 5.17. *Dos puntos de vista diferentes del ejemplo.*

6-Implementación del proyecto

6.1Elementos en común

En este apartado dedicaré a explicar aquellos elementos que toman parte de todos los métodos implementados. Antes de entrar en la explicación de dichos elementos, esto es lo que se puede apreciar al abrir cualquier proyecto de Unity en el que hemos implementando los diferentes métodos. Obviamente, la organización de carpetas no es la misma en todas, aunque sí que sigue el mismo criterio en general: dos carpetas que contienen lo necesario del HoloplaySDK, una carpeta para los materiales creados, otra para los scripts que definen el funcionamiento de cada método (sin tener en cuenta los usados en la *Leap Motion* y en el *Looking Glass*) y, por último, una carpeta en la que almacenaremos los resultados de los trazos que los/as usuarios/as realicen guardados en prefabs.

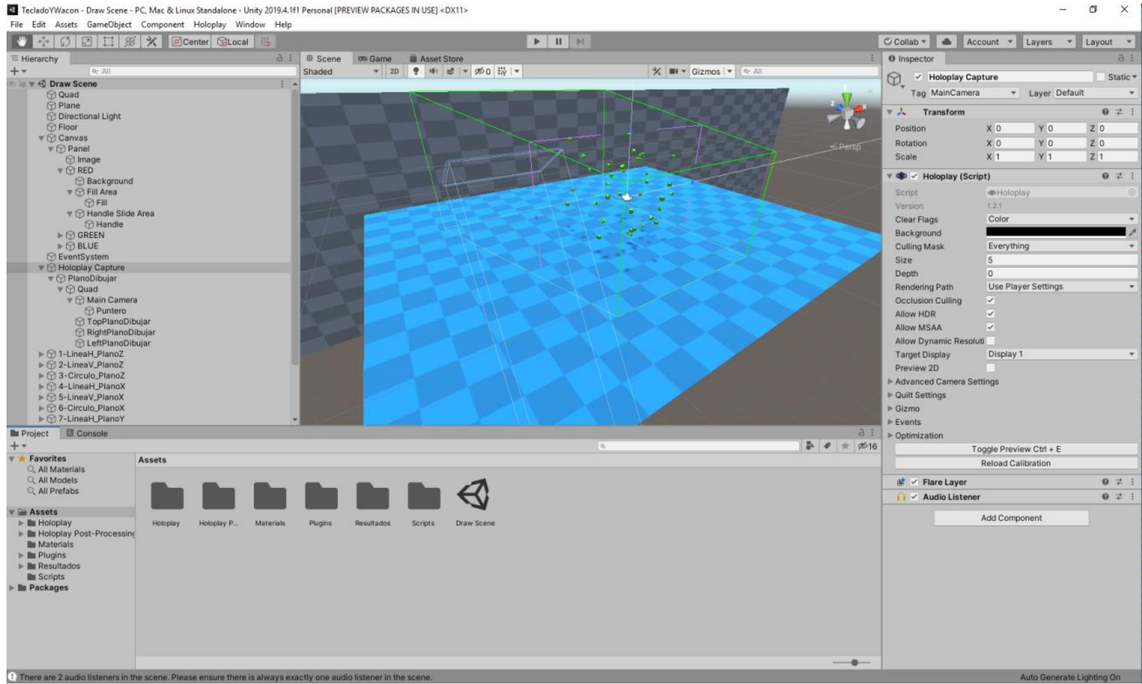


Imagen 6.1.1. *Vista general del proyecto "TecladoYWacom" en Unity.*

Vamos ahora con los elementos repetidos. Los dos primeros elementos que toman parte en todos los métodos son el suelo (*Plane*) y la pared (*Quad*) de la escena, en la que podemos ver en las siguientes imágenes cómo están establecidas con sus respectivos materiales. Decir que el tamaño de estos elementos puede ser demasiado grande, pero esta hecho a conciencia ya que de esta manerá el trabajo se realizaba con mayor comodidad.

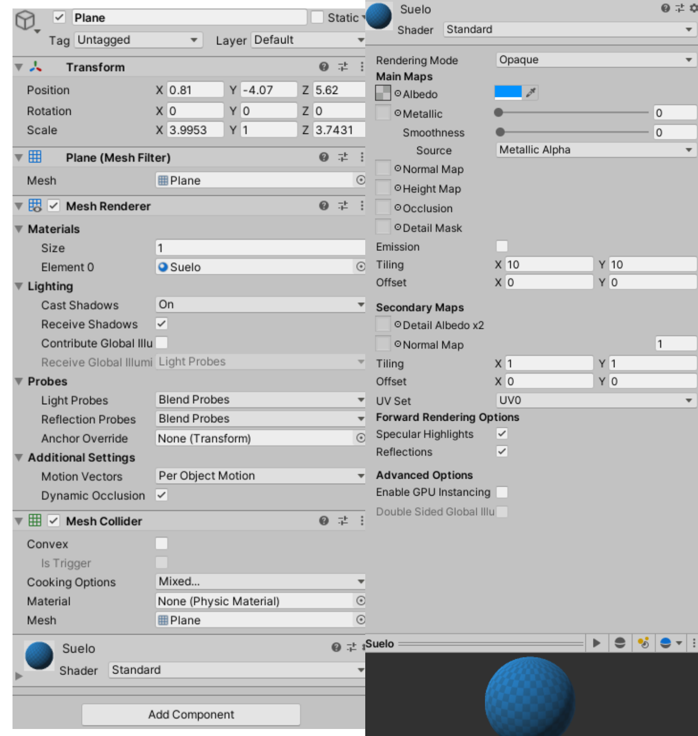


Imagen 6.1.2. *Configuración y material del suelo (Plane).*

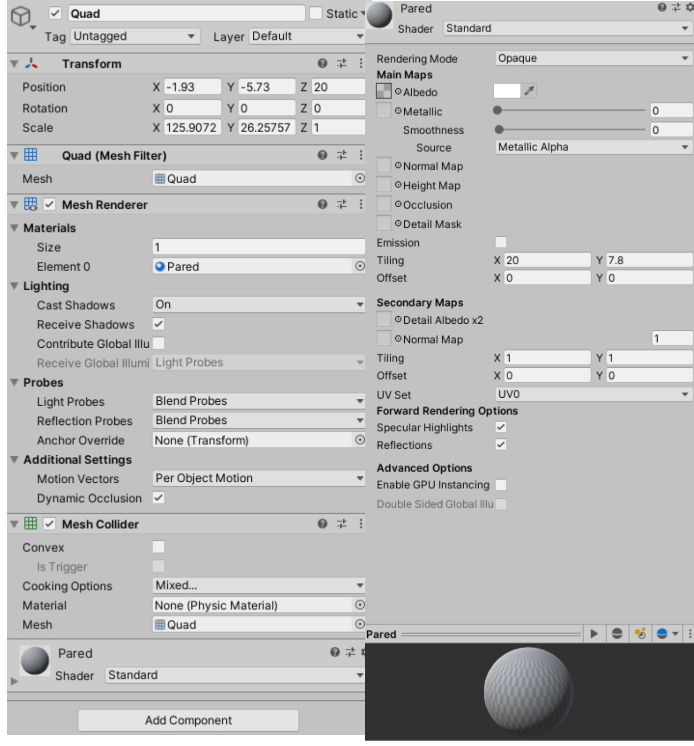


Imagen 6.1.3. *Configuración y material de la pared (Quad).*

El siguiente es *Directional Light*, unos de los elementos que viene predeterminadamente al crear una escena Unity. La luz está colocada encima de los pequeños cubos que usaremos en las pruebas, de tal manera que la sombra de estos elementos se refleje en el suelo y los usuarios/as puedan usar dichas sombras para guiarse a la hora de realizar las pruebas.

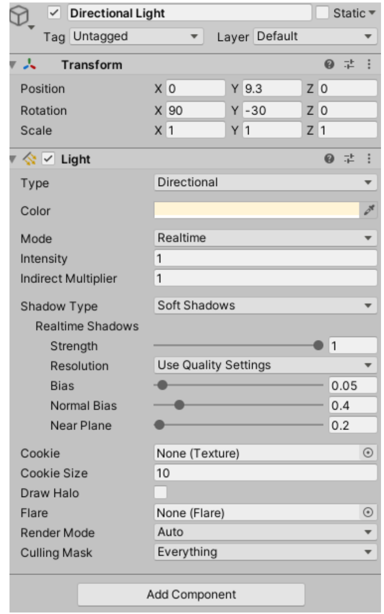


Imagen 6.1.4. *Configuración de la Directional Light.*

Para acabar este apartado, no podía faltar el *Holoplay Capture*. El interior del área que ocupa en la escena es el campo de visión del display, es decir, todos elementos que estén dentro de dicha área serán los elementos que podremos ver en el Looking Glass. Para poder realizar las pruebas de usuarios, lo que he hecho ha sido introducir los distintos vértices que forman los diferentes elementos de la prueba dentro de dicha área.

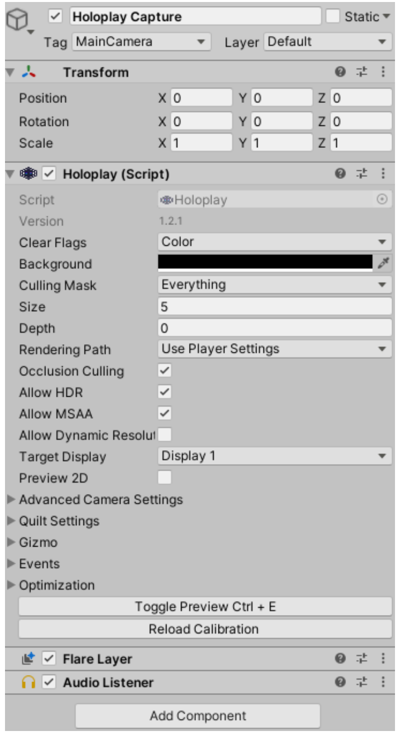


Imagen 6.1.5. *Configuración del Holoplay Capture.*

6.2 Método de tableta moviendo el plano mediante teclado

Comenzamos con la jerarquía de los elementos de la escena de este método:

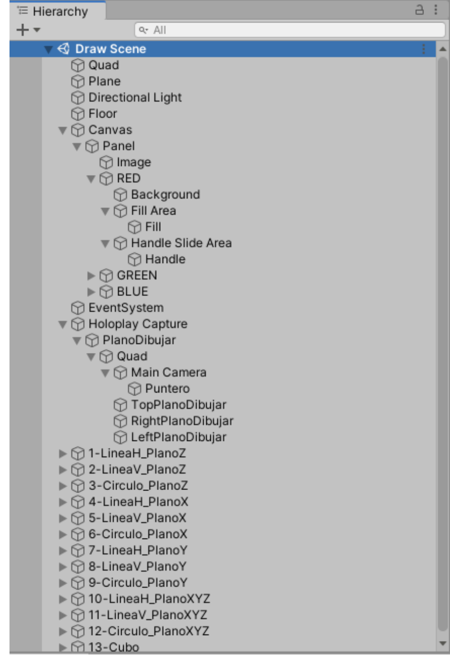


Imagen 6.2.1. *Jerarquía de los elementos de la escena.*

El primer elemento que quiero destacar (de los que no haya explicado hasta ahora) es *Floor*, que es un objeto 'vacío' cuya función será la de ser el "padre" de los trazos que el/la usuario/a vaya creando, por lo que no merece la pena entrar al detalle en su configuración.

Para dar la oportunidad de que los usuarios/as puedan cambiar los colores de los trazos, lo que he hecho ha sido crear un *Canvas* la cual contiene un selector de colores:

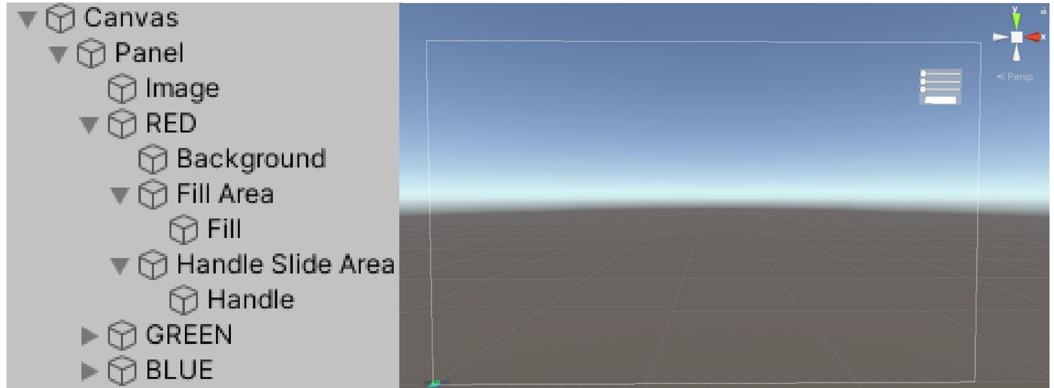


Imagen 6.2.2. *Jerarquía y la interfaz del Canvas del selector de colores.*

Como se puede ver en la jerarquía, el *Canvas* está formado por un *Panel*, el cual está formado por los 3 selectores de colores RGB (Red, Green, Blue), y una *Image*. Este último elemento lo que hace es mostrar el color resultante de la combinación de los selectores de colores, dependiendo obviamente de sus valores.

Cada selector está formado por 2 partes: la barra del selector y en círculo. Cada elemento está formado de la misma manera: El 'padre' conforma el total de la barra, mientras que el 'hijo' será el valor que irá variando; en el caso de la barra, se rellenará del color que corresponda (rojo, verde o azul) hasta donde se encuentre el círculo.

Vamos ahora con un elemento fundamental de la escena: *EventSystem*. La función de este objeto es el de enviar eventos a objetos de la escena que están basados en input (teclado, ratón, etc.), por lo que nos es imprescindible para poder dibujar, como veremos más adelante.

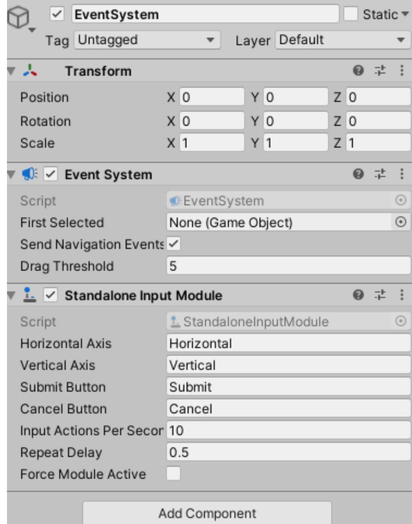


Imagen 6.2.3. *Configuración de EventSystem.*

Vamos a centrar ahora en los diferentes componentes que conforman el plano de dibujo:

Imagen 6.2.4. *Jerarquía del Holoplay Capture.*

Primero me gustaría comentar la razón de por qué he introducido el plano donde el usuario va a dibujar dentro del *Holoplay Capture*. La razón es que, jerárquicamente hablando, como el/la usuario/a va a dibujar siempre dentro del área del display (esa es la idea al menos), me parecía lo más correcto que el plano de dibujo fuera parte del *Capture*.

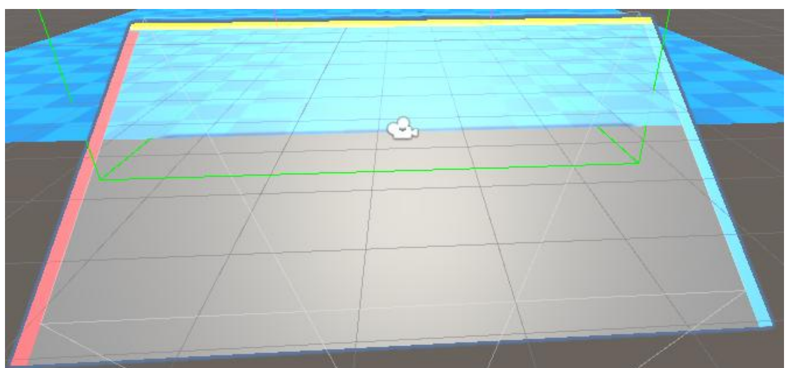


Imagen 6.2.5. *Plano donde los/las usuarios/as pintarán.*

Si nos fijamos bien en la jerarquía del plano de dibujo, tenemos un elemento llamado '*PlanoDibujar*' y otro '*Quad*'. Esto se debe a que, inicialmente, el plano estaba formado por el '*Quad*', pero al comenzar a programar el script de mover el plano, mediante el código no podía mover dicho plano. Debido a esto, lo que decidí fue crear un objeto vacío ('*PlanoDibujar*') el cual es el que movemos mediante el script, de tal manera que la componente '*Quad*' es hijo del dicho objeto. Con este método lo que consigo es "mover" el objeto vacío, pero se moverá a su vez todos los hijos.

Siguiendo con la composición, el plano está compuesto por los tres elementos que he usado de guía para el usuario y la componente *Main Camera*, el otro elemento que, junto con el *Directional Light*, vienen de manera predeterminada en una escena de Unity. Los tres elementos que he mencionado son: el 'top' (parte superior) del plano en color amarillo, el lado izquierdo en rojo y el lado derecho en azul. Para distinguir izquierda y derecha he seguido un criterio universal, el cual observé en Internet. Con esto quiero decir que he elegido estos colores a conciencia y no de manera aleatoria. Me gustaría comentar también que el material de dichos elementos es el mismo del cual está compuesto el *Quad*, pero cambiando a sus respectivos colores.

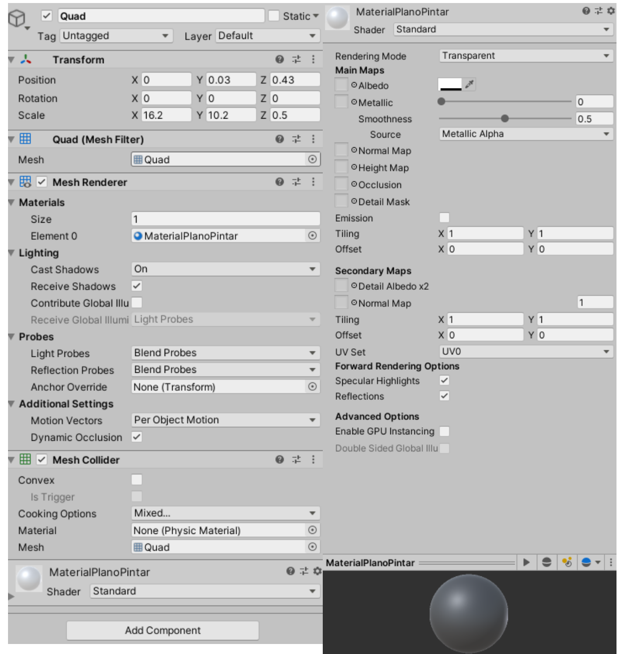


Imagen 6.2.6. *Configuración del Quad y de su material.*

Como se puede ver en la imagen 6.1.7, la cámara está colocada de manera que queda paralela al *Quad*, y como veremos más adelante, usaremos el campo de visión de esta cámara para que ocupe el área que conforma el plano de dibujo, así obtendremos la posición del ratón para poder pintar. La posición del ratón que se obtiene es respecto a la pantalla, lo que viene perfecto para poder usar la tableta porque recordemos que el área activa es proporcional a la pantalla. En otras palabras, el plano de dibujo tiene la proyección de la pantalla del ordenador, que a su vez nos sirve para poder usar la tableta con su área activa. De esta manera lo que obtenemos es la integración casi literal de la tableta en el display porque el tamaño que tiene el plano de dibujo es el mismo que el del área de la tableta que usaremos para pintar.

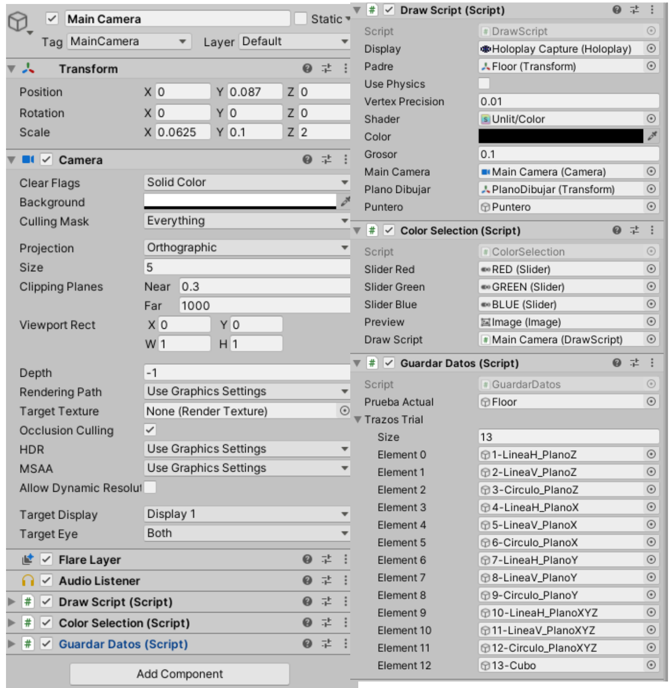


Imagen 6.2.7. *Configuración de Main Camera y sus scripts.*

Para acabar con los elementos de la jerarquía (sin tener en cuenta los elementos que hemos usado para las pruebas de usuarios), tenemos el *Puntero*, el cual es una esfera que tiene como función principal seguir el cursor. Lo que se consigue con esta esfera es permitir al/a la usuario/a poder visualizar donde va a generar el trazo. No tiene material por defecto ya que al comenzar la ejecución se le asigna el color rojo, un color que se puede contemplar y diferenciar fácilmente del resto de materiales que conforman la escena, además de que el/la usuario/a solamente se debe concentrar en realizar las pruebas y no en cambiar el color.

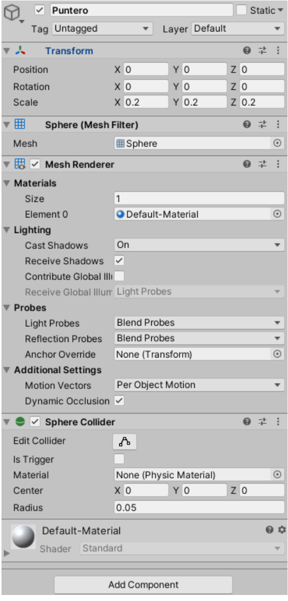


Imagen 6.2.8. *Configuración del Puntero.*

Vamos ahora con los scripts el cual realizan el funcionamiento de este método de dibujo. Comenzamos con el más sencillo que es el selector de colores. Explicando brevemente su función, el valor inicial del color seleccionado para dibujar es el rojo, y al cambiar el color mediante el selector de colores del *Canvas*, calcula el valor RGB y lo asigna al color:

```
1  using UnityEngine;
2  using UnityEngine.UI;
3
   Script de Unity | 0 referencias
4  public class ColorSelection : MonoBehaviour
5  {
6      Color color;
7      public Slider sliderRed;
8      public Slider sliderGreen;
9      public Slider sliderBlue;
10     public Image preview;
11     public DrawScript drawScript;
12
       Mensaje de Unity | 0 referencias
13     private void Start()
14     {
15         sliderRed.value = 255f;
16     }
17
       Mensaje de Unity | 0 referencias
18     void Update ()
19     {
20         color = new Color(sliderRed.value, sliderGreen.value, sliderBlue.value, 1);
21         if (preview)
22             preview.color = color;
23         if (drawScript)
24             drawScript.color = color;
25     }
26 }
```

Imagen 6.2.9. *Captura del código de ColorSelection.cs.*

Del script desarrollado para dibujar (*DrawScript.cs*) poco hay que describir ya que en el mismo código viene dado con las explicaciones necesarias de cada apartado:

```
1   using LookingGlass;
2   using System.Collections;
3   using System.Collections.Generic;
4   using UnityEngine;
5   using UnityEngine.EventSystems;
6
    Script de Unity | 1 referencia
7   public class DrawScript : MonoBehaviour
8   {
9       public Holoplay display;  //Atributo que referencia al Looking Glass
10
11      public Transform padre;  //Atributo que usamos para el padre de los trazos
12      public bool usePhysics = true;  //Atributo booleano para pode
13      public float vertexPrecision = 0.01f;  //Diferencia de distancia entre puntos
14      public Shader shader;   //Shader del material del color que asignamos al trazo
15      public Color color;   //Color que usamos para asignar al trazo
16      public float grosor = 0.1f;  //Grosor del trazo de dibujo
17
18      public Camera mainCamera;  //Atributo del Main Camera
19      public Transform planoDibujar;  //Atributo del objeto vacío que usamos para mover con teclas
20      public GameObject puntero;   //Atributo referente al puntero donde el/la usuario pinta
21      Vector3 posPuntero;  //Vector para obtener/determinar la posición del puntero
22
23      Vector3 calibrarAltura;  //Vector auxiliar que usamos para ajustar la posición del puntero
24
25      //Método Start que se ejecuta al comienzo de la ejecución de las pruebas
        Mensaje de Unity | 0 referencias
26      private void Start()
27      {
28          Vector3 posInicial = new Vector3(0, 0, 0);
29          planoDibujar.position = posInicial;
30          planoDibujar.rotation = new Quaternion(0, 0, 0, 1);
31      }
32
33      // Método Update que se ejecuta cada Frame
        Mensaje de Unity | 0 referencias
34      void Update()
35      {
36          //Pulsar R -> Resetear la posición del plano de dibujo
37          if (Input.GetKeyDown(KeyCode.R))
38          {
39              Vector3 posInicial = new Vector3(0, 0, 0);
40              display.transform.position = posInicial;
41              display.transform.rotation = new Quaternion(0, 0, 0, 1);
42              planoDibujar.position = posInicial;
43              planoDibujar.rotation = new Quaternion(0, 0, 0, 1);
44          }
45
46          //Click izquierdo --> pinchar con el lápiz en la tableta
47          if (Input.GetMouseButtonDown(0))
48          {
49              if(!EventSystem.current.IsPointerOverGameObject())
50                  StartCoroutine(draw());
51          }
52          //Pulsar W -> mover la posición del plano de dibujo en el eje Z positivamente
53          if (Input.GetKeyDown(KeyCode.W))
54          {
55              planoDibujar.Translate(new Vector3(0, 0, 1.25f));
56          }
57          //Pulsar S -> mover la posición del plano de dibujo en el eje Z negativamente
58          if (Input.GetKeyDown(KeyCode.S))
59          {
60              planoDibujar.Translate(new Vector3(0, 0, -1.25f));
61          }
62          //Pulsar Flecha Izq. -> Rotar a la izquierda el plano de dibujo
63          if (Input.GetKeyDown(KeyCode.LeftArrow))
64          {
65              planoDibujar.Rotate(new Vector3(0, 5.0f, 0));
66          }
67          //Pulsar Flecha Arriba -> Rotar a arriba el plano de dibujo
68          if (Input.GetKeyDown(KeyCode.UpArrow))
69          {
70              planoDibujar.Rotate(new Vector3(5.0f, 0, 0));
71          }
72          //Pulsar Flecha Dcha. -> Rotar a la derecha el plano de dibujo
73          if (Input.GetKeyDown(KeyCode.RightArrow))
74          {
75              planoDibujar.Rotate(new Vector3(0, -5.0f, 0));
76          }
77          //Pulsar Flecha Abajo -> Rotar a abajo el plano de dibujo
78          if (Input.GetKeyDown(KeyCode.DownArrow))
79          {
80              planoDibujar.Rotate(new Vector3(-5.0f, 0, 0));
81          }
82
83          //Asignación de color y calibración del puntero
84          puntero.GetComponent<Renderer>().material.color = color;
85          posPuntero = Camera.main.ScreenToWorldPoint(Input.mousePosition) - posicionRaton();
86          puntero.transform.position = posPuntero;
```

```
89      //Atributos que usaremos en la corutina
90      Vector3 newVertex;
91      Vector3 lastVertex;
92      //Corutina que simula el dibujado de los trazos
        1 referencia
93      IEnumerator draw()
94      {
95          //Creando y configurando el line render que conforma el dibujo
96          LineRenderer r = new GameObject().AddComponent<LineRenderer>();
97          r.transform.SetParent(padre);
98          r.startWidth = grosor;
99          r.endWidth = grosor;
100         r.material = new Material(shader);
101         r.material.color = color;
102
103         List<Vector3> posiciones = new List<Vector3>();
104
105         //Llamada al método para calcular la posición del ratón en el display
106         calibrarAltura = posicionRaton();
107
108         while (Input.GetMouseButton(0)) // Añadiendo los puntos del mouse al Line Render
109         {
110             //Obtención de la posición del puntero con su ajuste
111             newVertex = Camera.main.ScreenToWorldPoint(Input.mousePosition) - calibrarAltura;
112             //Comprobación de distancia entre vértices
113             if (Vector3.Distance(lastVertex, newVertex) >= vertexPrecision)
114             {
115                 posiciones.Add(newVertex);
116                 r.positionCount = posiciones.Count;
117                 r.SetPositions(posiciones.ToArray());
118                 lastVertex = newVertex;
119             }
120             yield return new WaitForEndOfFrame();
121         }
122         r.useWorldSpace = false;
123     }
126     /* Método que calibra la posición del ratón dependiendode la profundidad del plano
127      * de dibujo en el plano Z */
        2 referencias
128     public Vector3 posicionRaton()
129     {
130         int valor = (int)planoDibujar.transform.position.z;
131
132         if (valor <= 4 || valor >= -4)
133         {
134             return calibrarAltura = new Vector3(0, (float)1.0, 0);
135         }
136         else if(valor <= 9 || valor >= 5)
137         {
138             return calibrarAltura = new Vector3(0, (float)1.05, 0);
139         }
140         else if (valor <= 10)
141         {
142             return calibrarAltura = new Vector3(0, (float)1.15, 0);
143         }
144         else
145         {
146             return calibrarAltura = new Vector3(0, (float)1.1, 0);
147         }
148     }
149 }
```

Imagen 6.2.10. *Capturas de DrawScript.cs.*

6.3 Método de tableta y moviendo el plano con la Leap Motion

Al igual que en el anterior método, comenzaremos fijándonos en la jerarquía que compone esta escena:

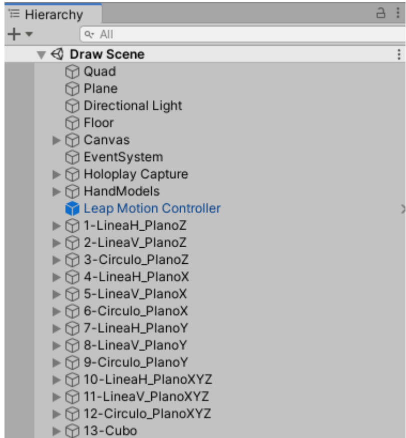


Imagen 6.3.1. *Jerarquía de los elementos de la escena.*

Como podemos observar en la imagen anterior (6.2.1), esta escena tiene en común muchos elementos de los que contiene la escena del anterior método, pero no los he colocado en el apartado 6.1 ya que en el último método no están implementados, como veremos más adelante. Estos elementos que hemos explicado en el apartado 6.2 y aparecen aquí tienen la misma configuración, por lo que no entramos de nuevo a explicar sus configuraciones y scripts. Aunque sí que hay una única diferencia, y es que hemos creado un script para controlar cuando el/la usuario/a pulsa espacio para parar el plano. Ese script es *ControlPlanoDibujar.cs*, que lo contiene el elemento *Quad*.

A diferencia del anterior método, el movimiento del plano está definido en función de la mano del/de la usuario/a, por lo que requerimos usar los elementos que componen las manos y la Leap Motion:

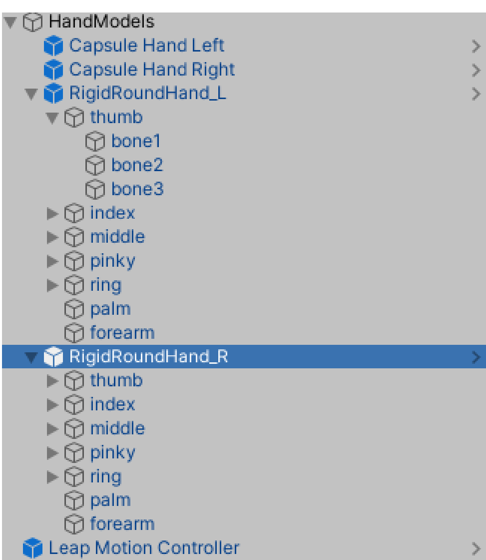


Imagen 6.3.2. *Leap Motion Controller y jerarquía de HandModels.*

HandModels es un elemento que ya viene definido en el SDK de la Leap Motion, pero lo que he hecho ha sido cambiar su posición en el espacio, concretamente en el plano Z, para que las manos no se visualicen dentro del display, ya que pensé que con todos los elementos que tiene este método, mostrar las manos en el display sería más confuso para la persona que esté dibujando.

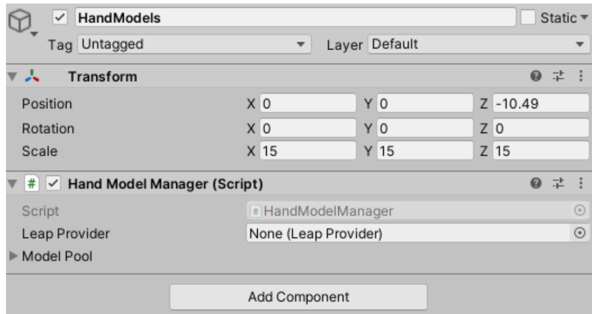


Imagen 6.3.3. *Configuración de HandModels.*

Entrando más a detalle de las componentes de *HandModels*, los primeros elementos que encontramos son *CapsuleHandLeft/Right*. Estos elementos son el "molde" de la mano que se muestra en Unity cuando la Leap Motion obtiene la forma y posición de la mano. En este ejemplo, ese 'model' es el mismo que el que representa el visualizador de diagnósticos. No mostramos su configuración ya que es irrelevante y es el que viene dado por el SDK, por lo que no hemos alterado nada de ello. La forma de la mano que vemos en la siguiente imagen puede parecer extraña. Eso es debido a la escala que hemos configurado al *LeapMotionController* que veremos más adelante:

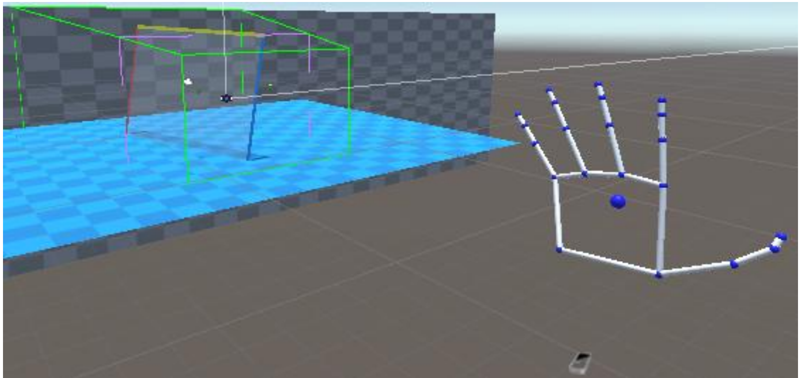


Imagen 6.3.4. *CapsuleHandLeft paralelo al plano de dibujar.*

En cuanto a los elementos *RigidRoundHand_L/R*, su función es el de mostrar el esqueleto de la mano, con su palma y dedos, y el antebrazo (aunque este último no lo usamos para nada). Cada uno de los dedos contiene los 3 huesos que conforma cada dedo, el cual podemos observar la separación entre éstos mediante las esferas azules en la imagen 6.3.4. Aunque es de gran utilidad poder contar con estos atributos de la mano, lo que realmente nos importa en la implementación de este método es la palma, ya que el plano de dibujar va a estar paralelo a ella. Decir que la rotación está ajustada porque inicialmente las manos estaban en otra dirección, y quería que estuvieran en la posición "real" en la que se encuentra la persona que esté en ese momento.

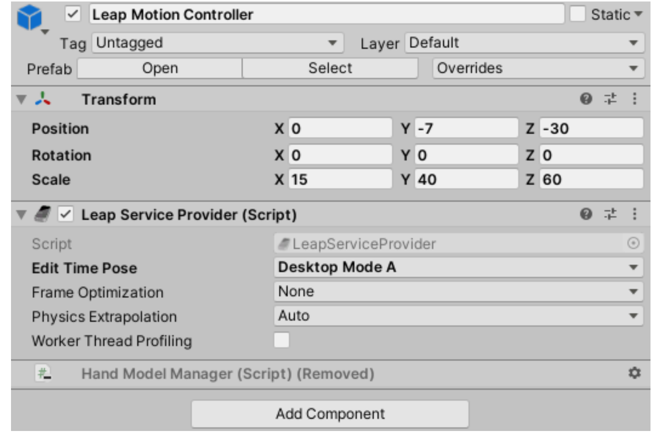


Imagen 6.3.5. *Configuración deLeapMotionController.*

Si nos fijamos en la imagen 6.3.4, la *LeapMotionController* se encuentra alejada del display, y si queremos que el/la usuario/a pueda mover el plano en toda el área que compone el display, necesita mayor escala en el plano Z que en el resto. En cuanto a la diferencia de escala del plano X e Y, lo he realizado después de distintas pruebas y ajustes y está me ha parecido la que mejor se ajusta para la finalidad del método.

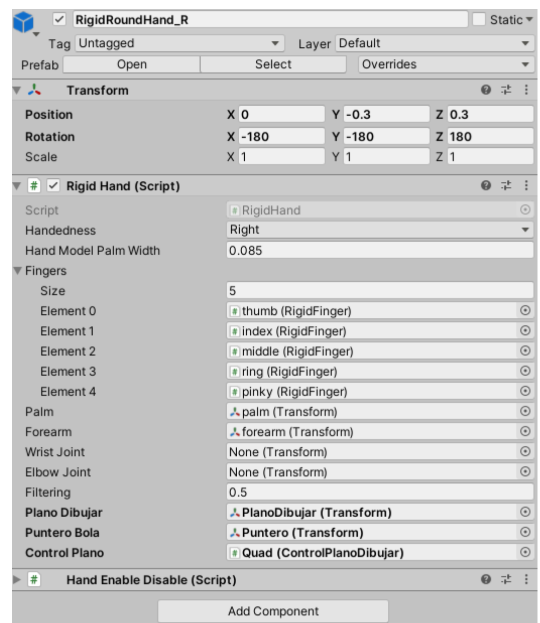


Imagen 6.3.6. *Configuración deRigidRoundHand_L/R.*

Antes de mostrar el script de *RigidHand.cs*, el cual contiene el funcionamiento para el movimiento del plano de dibujo respecto a la mano de la Leap Motion, quería decir que los scripts de *DrawScript.cs* y *ColorSelection.cs* son exactamente igual que en el método anterior, pero en el caso del primero, sin tener la implementación del movimiento del plano mediante el teclado obviamente. Como veremos ahora, el script de *RigidHand.cs*, script que viene dado en el SDK de la Leap Motion, accede a la variable booleana pública *"pararPlano"* que contiene el script *ControlPlanoDibujar.cs* para comprobar si el plano de dibujo debe seguir el movimiento de la mano.

```csharp
/******************************************************************************
 * Copyright (C) Ultraleap, Inc. 2011-2020.                                   *
 * Ultraleap proprietary and confidential.                                    *
 *                                                                            *
 * Use subject to the terms of the Leap Motion SDK Agreement available at     *
 * https://developer.leapmotion.com/sdk_agreement, or another agreement       *
 * between Ultraleap and you, your company or other organization.             *
 ******************************************************************************/

using UnityEngine;
using System.Collections;
using Leap;

namespace Leap.Unity {
  /** A physics model for our rigid hand made out of various Unity Collider. */
  public class RigidHand : SkeletalHand {
    public override ModelType HandModelType {
      get {
        return ModelType.Physics;
      }
    }
    public float filtering = 0.5f;

    public Transform planoDibujar;
    HandModel hand_model;
    Hand leap_hand;
    public Transform punteroBola;

    public ControlPlanoDibujar controlPlano;

    public override bool SupportsEditorPersistence() {
      return true;
    }

    public override void InitHand() {
      base.InitHand();
            hand_model = GetComponent<HandModel>();
            leap_hand = hand_model.GetLeapHand();
            planoDibujar.position = new Vector3(0, 0, 0);
            controlPlano = GameObject.Find("PlanoDibujarQuad").GetComponent<ControlPlanoDibujar>();
        }
```

```csharp
    public override void UpdateHand() {

      for (int f = 0; f < fingers.Length; ++f) {
        if (fingers[f] != null) {
          fingers[f].UpdateFinger();
        }
      }

      if (palm != null) {

        if (!controlPlano.pararPlano)
        {
            Rigidbody palmBody = palm.GetComponent<Rigidbody>();
            if (palmBody)
            {
                palmBody.MovePosition(GetPalmCenter());
                palmBody.MoveRotation(GetPalmRotation());
                ////////////////
                // ROTACIÓN *******
                Quaternion rotation = new Quaternion(palmBody.rotation.x, -palmBody.rotation.y, palmBody.rotation.z, -palmBody.rotation.w);
                planoDibujar.transform.rotation = rotation;
                Vector3 rot = planoDibujar.rotation.eulerAngles;
                rot = new Vector3(rot.x + 60f, rot.y + 180, rot.z);
                planoDibujar.rotation = Quaternion.Euler(rot);
                if (planoDibujar.rotation.x > 75)
                {
                    punteroBola.transform.Translate(new Vector3(0, 0, -1.03f));
                }
                // POSICIÓN *******
                planoDibujar.transform.position = new Vector3(palmBody.position.x, palmBody.position.y, palmBody.position.z + 35f);

            }
```

```
75              else
76              {
77                  palm.position = GetPalmCenter();
78                  palm.rotation = GetPalmRotation();
79                  ////////////////
80                  // ROTACIÓN *******
81                  Quaternion rotation = new Quaternion(palm.rotation.x, -palm.rotation.y, palm.rotation.z, -palm.rotation.w);
82                  planoDibujar.transform.rotation = rotation;
83                  Vector3 rot = planoDibujar.rotation.eulerAngles;
84                  rot = new Vector3(rot.x + 60f, rot.y + 180, rot.z);
85                  planoDibujar.rotation = Quaternion.Euler(rot);
86                  if (planoDibujar.rotation.x > 75)
87                  {
88                      punteroBola.transform.Translate(new Vector3(0, 0, -1.03f));
89                  }
90                  // POSICIÓN *******
91                  planoDibujar.transform.position = new Vector3(palm.position.x, palm.position.y, palm.position.z + 35f);
92              }
93          }
94      }
95
96        if (forearm != null) {
97          // Set arm dimensions.
98          CapsuleCollider capsule = forearm.GetComponent<CapsuleCollider>();
99          if (capsule != null) {
100           // Initialization
101           capsule.direction = 2;
102           forearm.localScale = new Vector3(1f / transform.lossyScale.x, 1f / transform.lossyScale.y, 1f / transform.lossyScale.z);
103
104           // Update
105           capsule.radius = GetArmWidth() / 2f;
106           capsule.height = GetArmLength() + GetArmWidth();
107         }
108
109         Rigidbody forearmBody = forearm.GetComponent<Rigidbody>();
110         if (forearmBody) {
111           forearmBody.MovePosition(GetArmCenter());
112           forearmBody.MoveRotation(GetArmRotation());
113         } else {
114           forearm.position = GetArmCenter();
115           forearm.rotation = GetArmRotation();
116         }
117       }
118     }
119   }
120 }
```

Imagen 6.3.7. *Capturas de RigidHands.cs.*

```
1   using UnityEngine;
2
    Script de Unity | 2 referencias
3   public class ControlPlanoDibujar : MonoBehaviour
4   {
5       public bool pararPlano;
6
7       //Método Start que se ejecuta al comienzo de la ejecución de las pruebas
        Mensaje de Unity | 0 referencias
8       void Start()
9       {
10          pararPlano = false;
11      }
12
13      // Método Update que se ejecuta cada Frame
        Mensaje de Unity | 0 referencias
14      void Update()
15      {
16          //Pulsar ESPACIO -> Parar/Volver a permitir movimiento del plano
17          if (Input.GetKeyDown(KeyCode.Space))
18          {
19              if (pararPlano)
20              {
21                  pararPlano = false;
22              }
23              else
24              {
25                  pararPlano = true;
26              }
27          }
28      }
29
        0 referencias
30      public bool getPararPlano() { return this.pararPlano; }
31  }
```

Imagen 6.3.8. *Capturas de ControlPlanoDibujar.cs.*

6.4 Método de Leap Motion y bola en el espacio

Para este método no vamos a cambiar el formato de la memoria, por lo que comenzamos con la jerarquía de la escena:

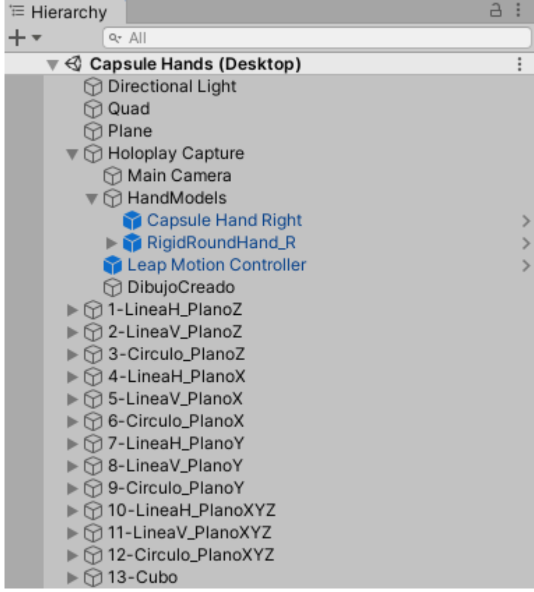


Imagen 6.4.1. *Jerarquía de los elementos de la escena.*

Para este ejemplo que no contiene un plano/tableta sobre el que dibujar, el elemento *MainCamera* va a mantenerse estática durante las pruebas, la cual tiene la siguiente configuración:

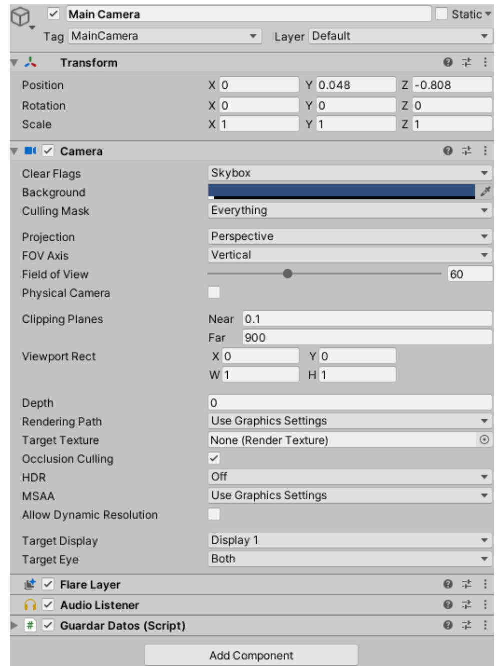


Imagen 6.4.2. *Configuración de MainCamera.*

Como en este método esta principalmente basado en el movimiento de la esfera con la mano derecha, la escena contiene, al igual que el método anterior, el *LeapMotionController* y *HandModels*. Me gustaría destacar que, al igual que antes, el controlador de la Leap Motion está colocado en la escena de manera que no se vea la mano de la persona que esté realizando la prueba y así evitar mayores confusiones, aunque la escala es mucho menor que en el anterior caso porque las dimensiones de todos los elementos en general son menores de lo visto hasta ahora.

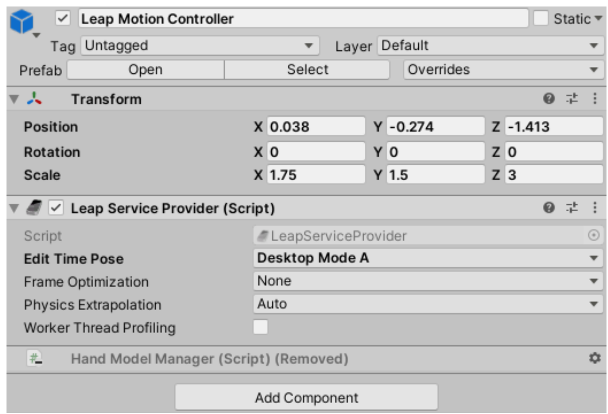


Imagen 6.4.3. *Configuración deLeapMotionController.*

En cuanto a *HandModels*, en este sistema se usa únicamente la mano derecha, por lo que hemos suprimido las componentes de la mano izquierda. Además, comentar que he colocado la esfera como hijo del dedo índice, ya que me parece lo más correcto jerárquicamente hablando y porque, como veremos después en el código, usamos el dedo índice como referencia para determinar la posición de la bola.

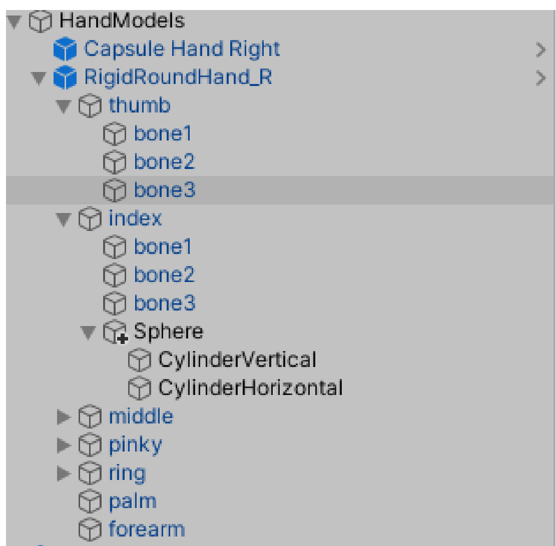


Imagen 6.4.4. *Jerarquía de HandModels.*

Antes de entrar en la configuración de la esfera, me gustaría decir que el haber elegido el color rojo me parecía el más adecuado para poder distinguir más fácilmente los ejes del suelo y la pared, además de tener mayor contraste.

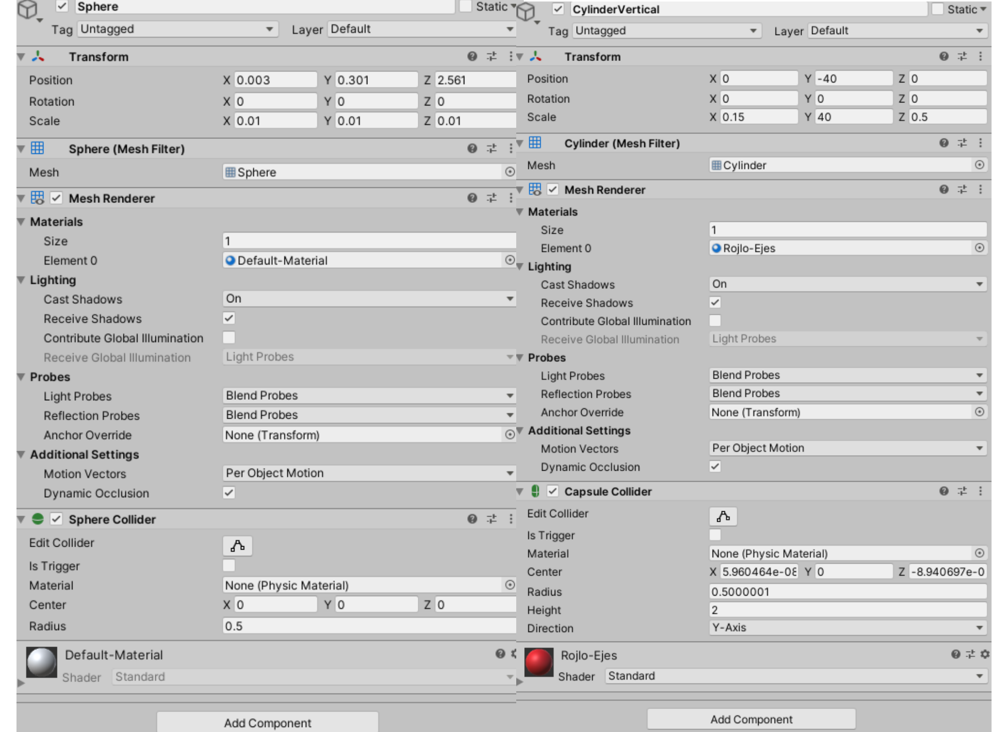


Imagen 6.4.5. *Configuración de la esfera y cilindro vertical.*

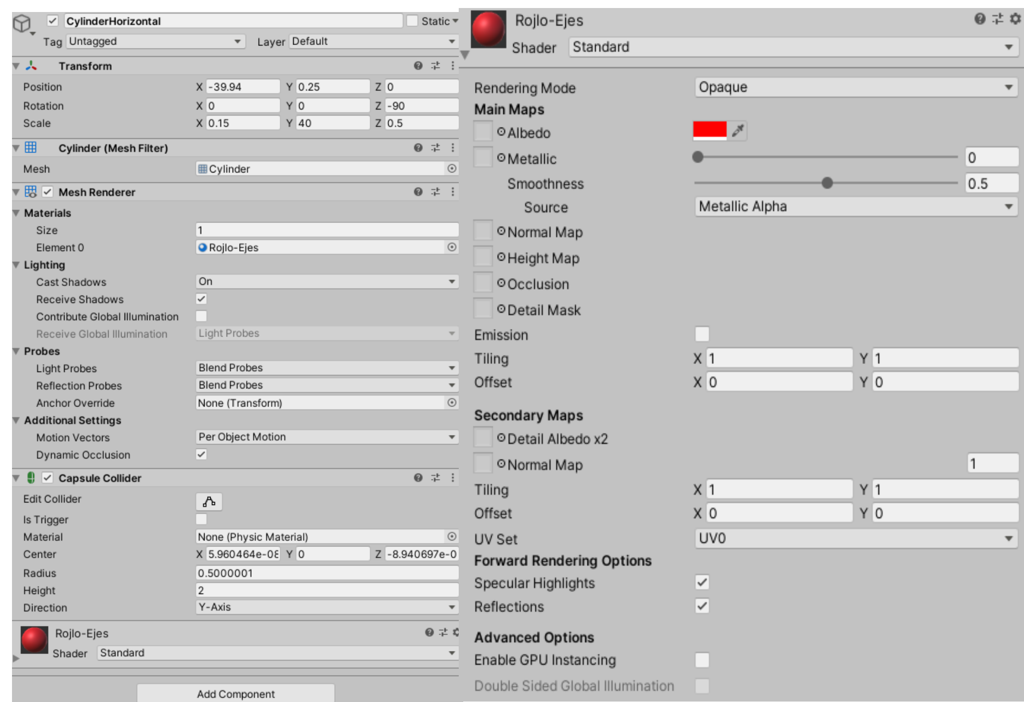


Imagen 6.4.6. *Configuración del cilindro horizontal y material del mismo.*

Vamos ahora con la configuración de la mano. Al igual que en el método anterior, la rotación está ajustada porque inicialmente las manos estaban en otra dirección, y quería que estuvieran en la posición “real” en la que se encuentra la persona que esté en ese momento. Decir que en este caso el script de *RigidHand.cs* es el original y no lo he modificado, ya que en funcionamiento del método está implementado en el script de *GetLeapFingers.cs*.

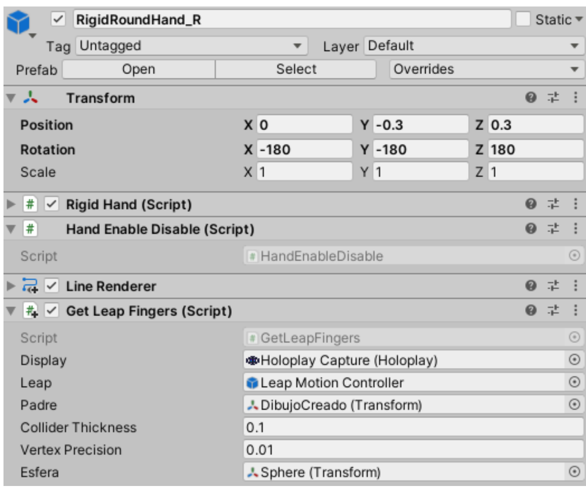


Imagen 6.4.7. *Configuración deRigidRoundHand_R.*

Antes de entrar en las capturas del script, en este método el “padre” de los trazos que el/la usuario/a va a generar es el objeto vacío *DibujoCreado*, que en los ejemplos anteriores era *Floor*.

```
1  using Leap;
2  using Leap.Unity;
3  using LookingGlass;
4  using System.Collections;
5  using System.Collections.Generic;
6  using UnityEngine;
7
   Script de Unity | 0 referencias
8  public class GetLeapFingers : MonoBehaviour
9  {
10     public Holoplay display;
11     public GameObject leap;
12
13     HandModel hand_model;
14     Hand leap_hand;
15     FingerModel finger;
16     private LineRenderer line;
17
18     public Transform padre;
19     public float colliderThickness = 0.1f;
20     public float vertexPrecision = 0.01f;
21
22     Vector3 posEsfera, finLaser;
23     float profX, profY, profZ;
24
25     public Transform esfera;
26
27     // Método que se ejecuta al comienzo de la ejecución de las pruebas
       Mensaje de Unity | 0 referencias
28     void Start()
29     {
30         //Definición de la profundidad a la que se va a dibujar respecto a la mano
31         profX = 0;
32         profY = 0;
33         profZ = 1.2f;
34         //Obtención de la componente que actua de "laser" puntero
35         line = gameObject.GetComponent<LineRenderer>();
36         //Obtención de la componente HandModel
37         hand_model = GetComponent<HandModel>();
38         //Obtención de la LeapHand del HandModel
39         leap_hand = hand_model.GetLeapHand();
40         //Obtención el dedo índice de la HandModel
41         finger = hand_model.fingers[1];
42         if (leap_hand == null) Debug.LogError("No leap_hand founded");
43         //Definición de la posición final del laser == posición de la ESFERA
44         finLaser = new Vector3(finger.GetTipPosition().x, finger.GetTipPosition().y, profZ);
45     }
```

```
47      // Método que se ejecuta cada Frame
        Mensaje de Unity | 0 referencias
48      void Update()
49      {
50          //Definición de posición inicial del "laser" desde el dedo ÍNDICE
51          line.SetPosition(0, finger.GetRay().origin);
52          //Cálculo posición final del "laser"
53          finLaser = new Vector3(finger.GetTipPosition().x + profX, finger.GetTipPosition().y + profY, finger.GetTipPosition().z + profZ);
54          line.SetPosition(1, finLaser);
55          //No queremos que se vea el laser -> sin anchura ni altura
56          line.SetWidth(0, 0);
57
58          //Definición posición final de la ESFERA
59          posEsfera = finLaser - esfera.transform.position;
60          esfera.Translate(posEsfera);
61
62          //Pulsar espacio -> corutina para pintar
63          if (Input.GetKeyDown("space"))
64          {
65              StopCoroutine(draw(finger, profX, profY, profZ));
66              StartCoroutine(draw(finger, profX, profY, profZ));
67          }
68      }
69
70      //Vectores que usamos para calcular distintos puntos en la corutina
71      Vector3 newVertex, lastVertex;
72      //Corutina para dibujar
        2 referencias
73      IEnumerator draw(FingerModel finger, float pX, float pY, float pZ)
74      {
75          //Configuración de la línea de trazado (Line Renderer)
76          LineRenderer r = new GameObject().AddComponent<LineRenderer>();
77          r.transform.SetParent(padre);
78          r.startWidth = 0.01f;
79          r.endWidth = 0.01f;
80          r.useWorldSpace = false;
81
82          List<Vector3> posiciones = new List<Vector3>();
```

```
84          //Dibuja mientras se mantenga pulsada la tecla de ESPACIO
85          while (Input.GetKey("space"))
86          {
87              //Obtención del punto nuevo
88              newVertex = new Vector3(finger.GetTipPosition().x + pX, finger.GetTipPosition().y + pY, finger.GetTipPosition().z + pZ);
89              //Cálculo de la diferencia entre punto nuevo y antiguo
90              if (Vector3.Distance(lastVertex, newVertex) >= vertexPrecision) //Checking distance between vertx
91              {
92                  posiciones.Add(newVertex);
93                  r.positionCount = posiciones.Count;
94                  r.SetPositions(posiciones.ToArray());
95                  lastVertex = newVertex;
96              }
97              yield return new WaitForEndOfFrame();
98          }
99      }
100 }
```

Imagen 6.4.8. *Capturas de GetLeapFingers.cs.*

7-Evaluación y datos obtenidos de los usuarios

Para la realización de las pruebas he decidido que el número de participantes sea proporcional al número de combinaciones posibles de estos tres métodos, por lo que tenemos: *3!* = ***6*** combinaciones únicas. 6 usuarios/as me parecían pocos para realizar las pruebas, y 18 no es mal número pero no disponía de gente ni de tiempo para ello, por lo que son 12 las personas que han realizado las pruebas. Además, se han tenido en cuenta los distintos perfiles en cuanto a su trabajo habitual: estudiante de ingeniería informática, estudiante de ingeniería y máster de telecomunicación, profesor de universidad, diseñador/a gráfico/a, enfermera…

Vamos ahora con las pruebas. La prueba que cada persona realizaba con cada método eran los siguientes (se repiten estas pruebas en todos los métodos): una línea horizontal, otra vertical y realizar un círculo en el plano X, lo mismo para los planos Z e Y, realizar una línea horizontal, otra vertical y un círculo que contenga las 3 dimensiones y por último un cubo. Contando cada una de estas pruebas, tenemos que cada persona realiza 13 pruebas en cada método.

Cada una de ellas consiste en unir los vértices de las líneas, y en el caso de los círculos, el/la usuario/a tiene 4 puntos el cual debe unir trazando el círculo. En el caso del cubo, al igual que en las líneas, debe unir los 8 vértices que componen el cubo mediante rectas. Estos son algunos de los resultados visuales obtenidos:

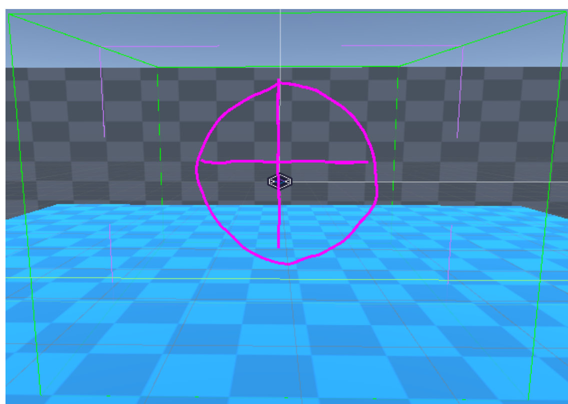


Imagen 7.1. *Línea horizontal, vertical y círculo del plano Z.*

Imagen 7.2. *Línea horizontal, vertical y círculo del plano X.*

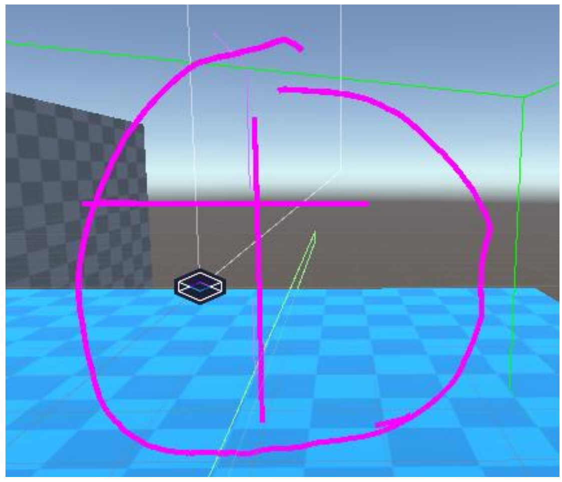


Imagen 7.3. *Línea horizontal, vertical y círculo del plano Y.*

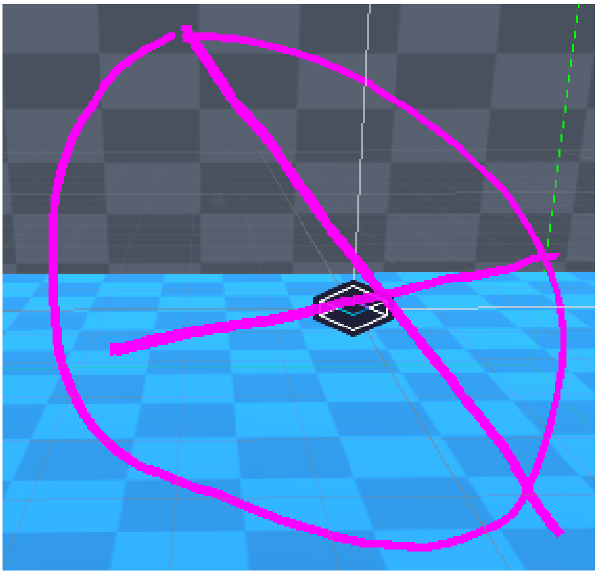


Imagen 7.4. *Línea horizontal, vertical y círculo en las 3 dimensiones.*

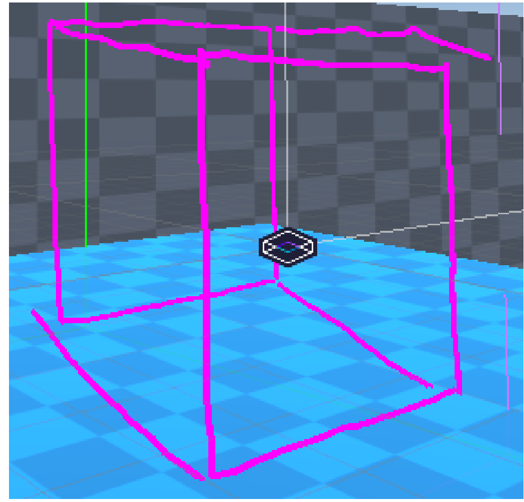


Imagen 7.5. *Última prueba, el cubo.*

Para guardar los resultados que los/as usuarios/as que realizan en cada prueba he desarrollado un script que empleo de igual manera en los 3 métodos (*GuardarDatos.cs*), que además de guardar los datos y trazos que dibujan los/as usuarios/as, también es el encargado de la ejecución de las pruebas, haciendo que se oculten o aparezcan los objetos necesarios de cada prueba.

Me gustaría comentar un pequeño detalle, y es que al cambiar entre pruebas, lo que se hace es, primero guardar el trazo dibujado e manera local, borrar esos trazos de la ejecución (eliminar los hijos), ocultar los objetos de la prueba actual y finalmente mostrar el de la siguiente prueba.

Para poder distinguir las distintas pruebas que los/as usuarios/as realizan con los diferentes métodos desarrollados, los *prefabs* que almacenamos se guardan con el siguiente formato de nombre:

"nombreUsuario/a"_"nombrePrueba"_"duraciónPruebaEnMilis".prefab

```
1   using UnityEngine;
2   using UnityEditor;
3
    Script de Unity | 0 referencias
4   public class GuardarDatos : MonoBehaviour
5   {
6       private float startTime;  //Variable que usamos para obtener el tiempo de inicio en milisegundos
7       private float tiempoActual;  //Variable que almacena el tiempo en milisegundo al pulsar f12
8       private int duracion;  //Variable que usaremos para obtener la duración en milisegundos de cada prueba
9       private int numPrueba;  //Variable que usaremos para saber el número de prueba en la que nos encontramos (del 1 al 11)
10      public GameObject pruebaActual;  //Variable en la que obtendremos el trazo actual de cada prueba ->'DibujoCreado' en la jerarquía
11      public GameObject[] trazosTrial;  //Array de las pruebas a realizar
12      private string prefabName, prefabPath, userName, nombrePrueba;
13
14      // Start is called before the first frame update
        Mensaje de Unity | 0 referencias
15      void Start()
16      {
17          startTime = (float)(Time.timeSinceLevelLoad * 1000f);
18          numPrueba = 0;
19          duracion = 0;
20          userName = "PRUEBA_";                //Nombre o ID del usuario que realice las pruebas
21          prefabPath = "Assets/Resultados/" + userName;  //directorio donde almacenaremos los resultados
22          nombrePrueba = "1-LineaH-PlanoZ";   //Nombre de la primera prueba que el usuario realizará
23
24          //ocultamos las pruebas que se realizarás mas adelante
25          for(int i = 1; i < trazosTrial.Length; i++)
26              trazosTrial[i].SetActive(false);
27      }
28
29      // Update is called once per frame
        Mensaje de Unity | 0 referencias
30      void Update()
31      {
32          //Pulsar F12 -> Guardar objeto y tiempo de la demo
33          if (Input.GetKeyDown(KeyCode.F12))
34          {
35              tiempoActual = (float)(Time.timeSinceLevelLoad * 1000f);
36              duracion = (int) (tiempoActual - startTime);
37              startTime = tiempoActual;
38              prefabName = prefabPath + nombrePrueba + "_" + duracion.ToString() + ".prefab";      //nombre del prefab que almacenaremos
39
```

```
40          switch (numPrueba)
41          {
42              //Prueba 1 --> línea HORIZONTAL plano Z
43              case 0:
44                  PrefabUtility.SaveAsPrefabAsset(pruebaActual, prefabName);
45                  destruirHijosPruebaActual();
46                  trazosTrial[numPrueba].SetActive(false);
47                  numPrueba++;
48                  trazosTrial[numPrueba].SetActive(true);
49                  nombrePrueba = "2-LineaV-PlanoZ";
50                  break;
51              //Prueba 2 --> línea VERTICAL plano Z
52              case 1:
53                  PrefabUtility.SaveAsPrefabAsset(pruebaActual, prefabName);
54                  destruirHijosPruebaActual();
55                  trazosTrial[numPrueba].SetActive(false);
56                  numPrueba++;
57                  trazosTrial[numPrueba].SetActive(true);
58                  nombrePrueba = "3-Circulo-PlanoZ";
59                  break;
60              //Prueba 3 --> círculo plano Z
61              case 2:
62                  PrefabUtility.SaveAsPrefabAsset(pruebaActual, prefabName);
63                  destruirHijosPruebaActual();
64                  trazosTrial[numPrueba].SetActive(false);
65                  numPrueba++;
66                  trazosTrial[numPrueba].SetActive(true);
67                  nombrePrueba = "4-LineaH-PlanoX";
68                  break;
69              //Prueba 4 --> línea HORIZONTAL plano X
70              case 3:
71                  PrefabUtility.SaveAsPrefabAsset(pruebaActual, prefabName);
72                  destruirHijosPruebaActual();
73                  trazosTrial[numPrueba].SetActive(false);
74                  numPrueba++;
75                  trazosTrial[numPrueba].SetActive(true);
76                  nombrePrueba = "5-LineaV-PlanoX";
77                  break;
78              //Prueba 5 --> línea VERTICAL plano X
79              case 4:
80                  PrefabUtility.SaveAsPrefabAsset(pruebaActual, prefabName);
81                  destruirHijosPruebaActual();
82                  trazosTrial[numPrueba].SetActive(false);
83                  numPrueba++;
84                  trazosTrial[numPrueba].SetActive(true);
85                  nombrePrueba = "6-Circulo-PlanoX";
86                  break;
87              //Prueba 6 --> círculo plano X
88              case 5:
89                  PrefabUtility.SaveAsPrefabAsset(pruebaActual, prefabName);
90                  destruirHijosPruebaActual();
91                  trazosTrial[numPrueba].SetActive(false);
92                  numPrueba++;
93                  trazosTrial[numPrueba].SetActive(true);
94                  nombrePrueba = "7-LineaH-PlanoY";
95                  break;
96              //Prueba 7 --> línea HORIZONTAL plano Y
97              case 6:
98                  PrefabUtility.SaveAsPrefabAsset(pruebaActual, prefabName);
99                  destruirHijosPruebaActual();
100                 trazosTrial[numPrueba].SetActive(false);
101                 numPrueba++;
102                 trazosTrial[numPrueba].SetActive(true);
103                 nombrePrueba = "8-LineaV-PlanoY";
104                 break;
105             //Prueba 8 --> línea VERTICAL plano Y
106             case 7:
107                 PrefabUtility.SaveAsPrefabAsset(pruebaActual, prefabName);
108                 destruirHijosPruebaActual();
109                 trazosTrial[numPrueba].SetActive(false);
110                 numPrueba++;
111                 trazosTrial[numPrueba].SetActive(true);
112                 nombrePrueba = "9-Circulo-PlanoY";
113                 break;
114             //Prueba 9 --> círculo plano Y
115             case 8:
116                 PrefabUtility.SaveAsPrefabAsset(pruebaActual, prefabName);
117                 destruirHijosPruebaActual();
118                 trazosTrial[numPrueba].SetActive(false);
119                 numPrueba++;
120                 trazosTrial[numPrueba].SetActive(true);
121                 nombrePrueba = "10-LineaH-PlanoXYZ";
122                 break;
123             //Prueba 10 --> línea VERTICAL en las 3 dimensiones
124             case 9:
125                 PrefabUtility.SaveAsPrefabAsset(pruebaActual, prefabName);
126                 destruirHijosPruebaActual();
127                 trazosTrial[numPrueba].SetActive(false);
128                 numPrueba++;
129                 trazosTrial[numPrueba].SetActive(true);
130                 nombrePrueba = "11-LineaV-PlanoXYZ";
131                 break;
```

```
                //Prueba 11 --> línea HORIZONTAL en las 3 dimensiones
                case 10:
                    PrefabUtility.SaveAsPrefabAsset(pruebaActual, prefabName);
                    destruirHijosPruebaActual();
                    trazosTrial[numPrueba].SetActive(false);
                    numPrueba++;
                    trazosTrial[numPrueba].SetActive(true);
                    nombrePrueba = "12-Circulo-PlanoXYZ";
                    break;
                //Prueba 12 --> círculo inclinado en las 3 dimensiones
                case 11:
                    PrefabUtility.SaveAsPrefabAsset(pruebaActual, prefabName);
                    destruirHijosPruebaActual();
                    trazosTrial[numPrueba].SetActive(false);
                    numPrueba++;
                    trazosTrial[numPrueba].SetActive(true);
                    nombrePrueba = "13-Cubo";
                    break;
                //Prueba 13 --> Cubo
                case 12:
                    PrefabUtility.SaveAsPrefabAsset(pruebaActual, prefabName);
                    destruirHijosPruebaActual();
                    trazosTrial[numPrueba].SetActive(false);
                    break;

            }
        }
    }

    //Método que elimina los hijos del GameObject pruebaActual
    13 referencias
    void destruirHijosPruebaActual()
    {
        //PrefabUtility.UnloadPrefabContents(pruebaActual);
        foreach (Transform child in pruebaActual.transform)
        {
            Destroy(child.gameObject);
        }
    }
}
```

Imagen 7.6. *Capturas de DrawScript.cs.*

Una vez realizadas las pruebas por cada usuario/a, he obtenido un promedio de los tiempos de realización y he convertido los promedios de milisegundos a segundos:

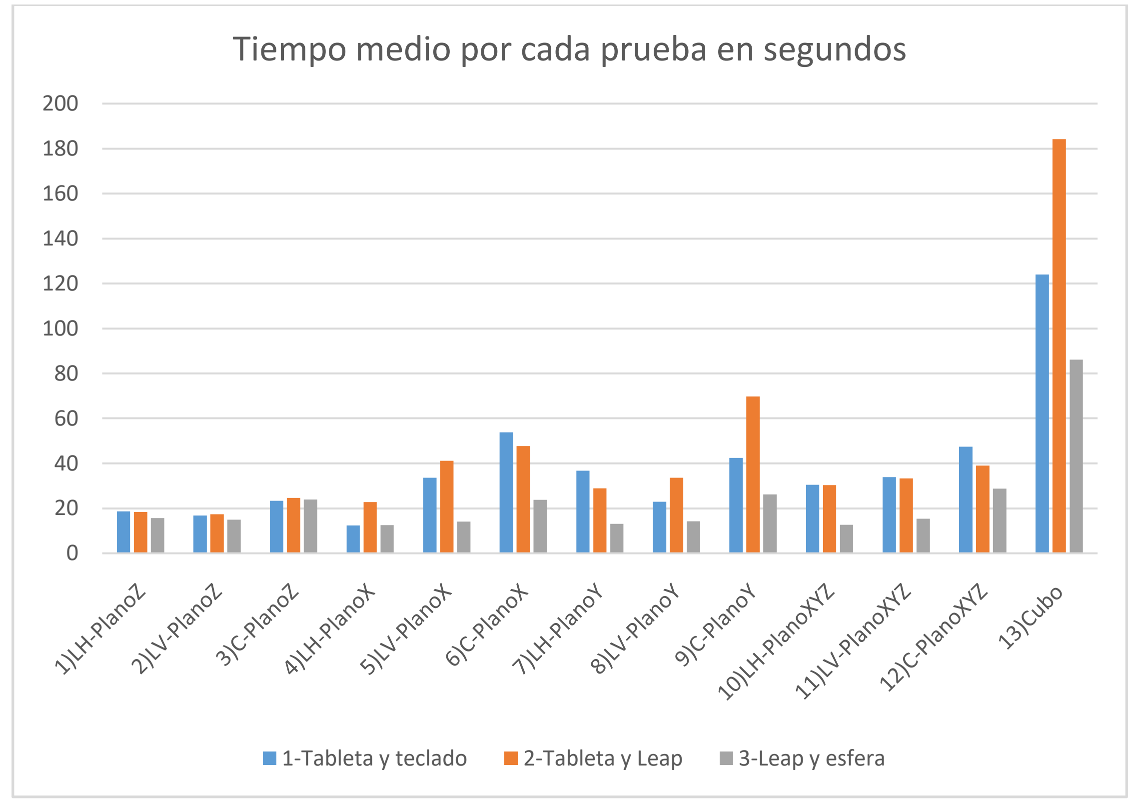


Imagen 7.7. *Gráfica de los tiempos medios de cada prueba por cada método.*

Además de eso, he puntuado bajo mi criterio la precisión de los resultados obtenidos por cada usuario/a en cada prueba, y al igual que antes, he hecho promedio de cada prueba con cada técnica. Para puntuar los trazos he seguido un baremo de manera que he puntuado con valores entre 1 y 5:

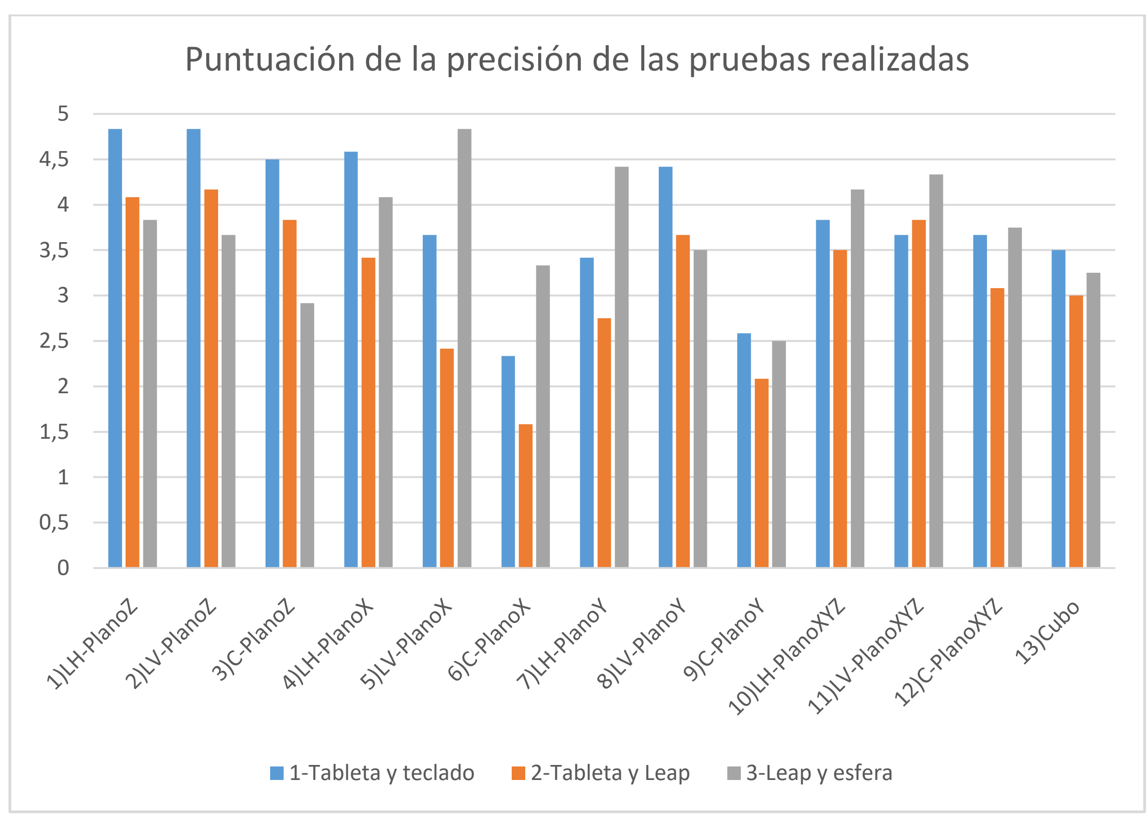
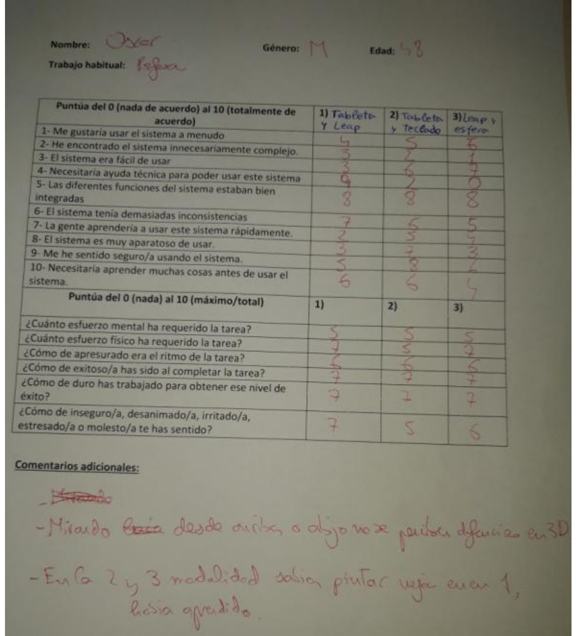


Imagen 7.8. *Gráfica de la valoración media de cada prueba por cada método.*

Una vez realizadas las pruebas a cada usuario/a, les pedí que rellenaran un cuestionario con un total de 16 preguntas, en el cual 10 de esas preguntas son relacionadas con el sistema SUS (las 10 primeras preguntas) y el resto con el sistema NASA-TLX (las 6 últimas). El sistema SUS [16] es la escala de usabilidad del sistema, que se debe evaluar posterior a la prueba de usabilidad. En cuanto a NASA-TLX, es otro tipo de cuestionario posterior a la tarea que es útil para medir la carga de trabajo de la tarea realizada [15].

Además de este tipo de preguntas, después de la tabla que contiene dichas preguntas, le he permitido a las personas que han realizado las prueba a hacer distintos comentarios adicionales. Estos comentarios pueden ser tanto positivos como negativos, ya que ambos son de gran ayuda: los comentarios positivos me ayudan a saber qué es lo que más ha gustado y/o lo que mejor funciona, mientras que los negativos me ayudan a comprender algunos fallos a mejorar o aspectos puntuales en los que al desarrollar yo el proyecto no había tenido en cuenta. El punto de vista del/de la usuario/a es muy útil para este último, como explicaré más en profundidad en las conclusiones.



Nombre: 	Género: 	Edad:
Trabajo habitual:

Puntúa del 0 (nada de acuerdo) al 10 (totalmente de acuerdo)	1) Tableta y Leap	2) Tableta y Teclado	3) Leap y esfera
1- Me gustaría usar el sistema a menudo			
2- He encontrado el sistema innecesariamente complejo.			
3- El sistema era fácil de usar			
4- Necesitaría ayuda técnica para poder usar este sistema			
5- Las diferentes funciones del sistema estaban bien integradas			
6- El sistema tenía demasiadas inconsistencias			
7- La gente aprendería a usar este sistema rápidamente.			
8- El sistema es muy aparatoso de usar.			
9- Me he sentido seguro/a usando el sistema.			
10- Necesitaría aprender muchas cosas antes de usar el sistema.			
Puntúa del 0 (nada) al 10 (máximo/total)	**1)**	**2)**	**3)**
¿Cuánto esfuerzo mental ha requerido la tarea?			
¿Cuánto esfuerzo físico ha requerido la tarea?			
¿Cómo de apresurado era el ritmo de la tarea?			
¿Cómo de exitoso/a has sido al completar la tarea?			
¿Cómo de duro has trabajado para obtener ese nivel de éxito?			
¿Cómo de inseguro/a, desanimado/a, irritado/a, estresado/a o molesto/a te has sentido?			

Comentarios adicionales:

Imagen 7.9. *Ejemplo del cuestionario NASA-TLX y SUS que los/as usuarios/as realizaron.*

Después de haber realizado las pruebas y haber obtenido los resultados de todos los cuestionarios, estos son los promedios en función del tipo de cuestionario:

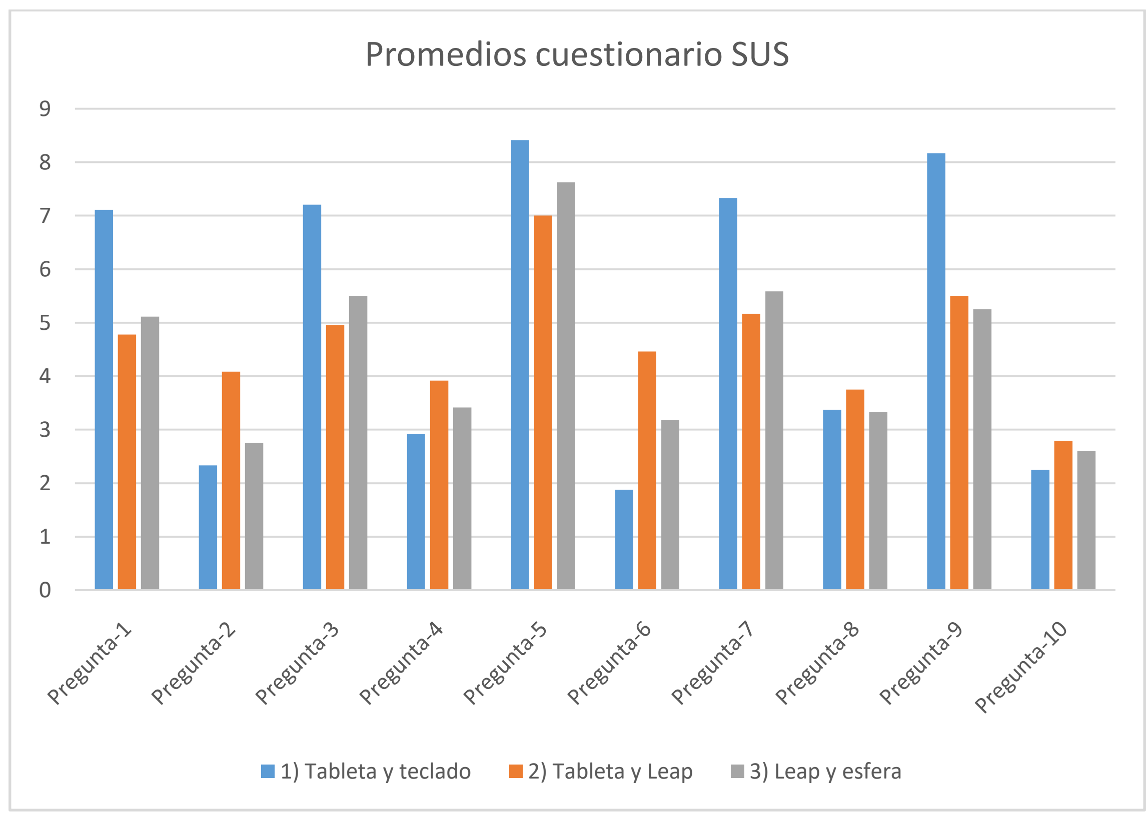


Imagen 7.10. *Gráfica del promedio de preguntas de tipo SUS.*

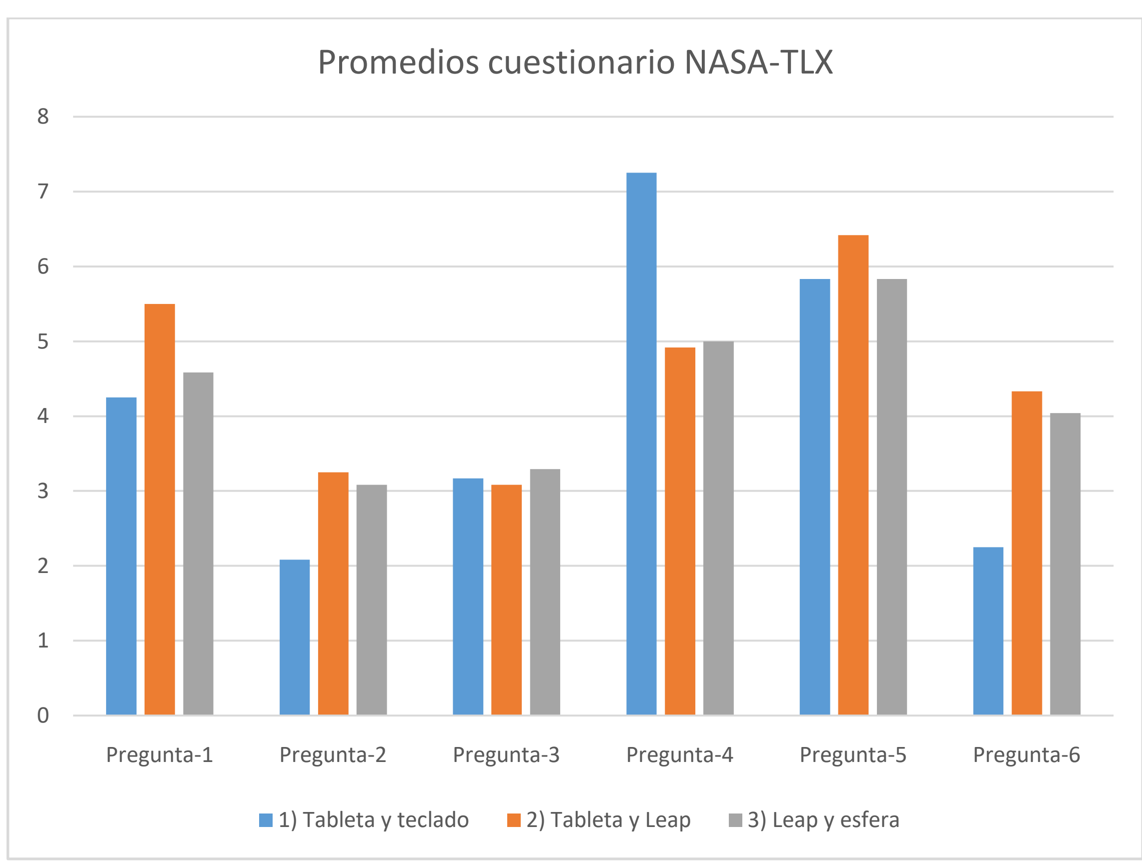


Imagen 7.11. *Gráfica del promedio de preguntas de tipo NASA-TLX.*

8-Discusión de resultados

Para este apartado, vamos a diferenciar dos tipos de conclusiones: la primera parte irá centrada a los resultados obtenidos en las pruebas, tanto en el tiempo de realización como en la precisión obtenida. La segunda parte será un conjunto de conclusiones obtenidas de los cuestionarios mencionados en el apartado anterior, así como los comentarios adicionales el cual son muy parecidas a mis sensaciones acerca del proyecto.

Observando los resultados obtenidos, los tiempos obtenidos son esperados, ya que, en cuanto a las pruebas, están ordenadas de manera que las primeras tienen menor dificultad, y la dificultad aumenta a medida del paso de las pruebas, hasta llegar al cubo, el cual casi no es comparable el esfuerzo que requería esta tarea respecto al resto de pruebas.

Pero fijándonos en la diferencia de tiempos por método de dibujo, el segundo es el método que más problemas ha podido generar a los/as usuarios/as, por lo que es lógico que, generalmente, el promedio ha sido mayor que el resto de métodos. El que menor tiempo medio ha obtenido en todas las tareas ha sido el método de la Leap Motion y la esfera (el último). Esto se debe a que es el método más fácil a la hora de realizar las pruebas, aunque esto no significa que sea el que mejor resultados obtiene.

En cuanto a la precisión obtenida de las pruebas, en general, el primer sistema es el que mejores resultados ha obtenido, y es lógico ya que es el más cómodo (que no fácil, a diferencia de antes) de realizar en mi opinión y el de varios/as usuarios/as: fácil de mover con las teclas y más fácil dibujar los trazos en la tableta. Aunque cabe destacar que el método de la Leap Motion y la esfera, ha destacado en varias de las pruebas, como son línea vertical en el plano X (prueba 5), círculo plano X (prueba 6), línea horizontal plano Y (prueba 7) y las 3 pruebas en las que entran las 3 dimensiones. Creo que esto se debe a que las primeras pruebas ayudaban a las personas a comprender mejor el espacio del Looking Glass, así como el ayudarse de los ejes de la esfera y las sombras que reflejan los vértices a unir.

Vamos ahora con la segunda parte, y comenzamos el cuestionario de tipo NASA-TLX. El método que más inseguridades y más esfuerzo ha requerido por parte de las personas ha sido el segundo método, seguido por el tercero. Esto se debe, primero por la complejidad del método (sobre todo para el segundo método), y segundo, a inconsistencias hardware que surgían al realizar las evaluaciones, principalmente por parte de la Leap Motion.

En el otro lado, tenemos las preguntas del cuestionario SUS. Los/as usuarios/as creen que los mejores resultados han sido obtenidos mediante el primer método, el cual ha sido el que mayor puntuación ha obtenido en todos los apartados, excepto en la pregunta de si necesitarán aprender muchas cosas antes de usar el sistema. Por lo que podemos pensar que este método ha sido el que más fácil de usar le ha parecido a la

gente y con la que más cómodos/as se han sentido, cosa que, de manera general, se demuestra en los resultados. En cuanto a los otros métodos, tienen puntuaciones muy similares, aunque prevalece el aspecto de que el segundo método es el que más inconsistencias tiene y un poco más aparatoso de usar.

Para finalizar con el apartado, estos son algunos de los comentarios adicionales de las personas que han realizado las pruebas. Como veremos, hay todo tipo de opiniones:

Positivos:

- ✓ *"Métodos originales y nada complicados."*
- ✓ *"Muy ingenioso y muy bien hecho."*
- ✓ *"El último método es el que me ha parecido más cómodo y preciso."*
- ✓ *"Resulta una idea original que puede tener aplicaciones muy interesantes, como la utilización en diseño gráfico u otras temáticas artísticas."*
- ✓ *"Utilizar el teclado para mover el plano y la tableta para dibujar me parece el método más intuitivo."*
- ✓ *"En general, muy buen trabajo."*
- ✓ *"Me ha parecido muy rápido, seguro e intuitivo realizar las pruebas mediante el teclado."*
- ✓ *"Muy útil poder acceder a cualquier punto en el espacio para realizar cualquier actividad. Sistemas muy útiles."*
- ✓ *"El Looking Glass es un poco difícil de visualizar ciertos detalles pero es un avance increíble poder ver 3D en directo."*
- ✓ *"Me han resultado unos sistemas muy interesantes. El primer método es el que me ha resultado más sencillo y fácil de realizar. En alguna ocasión, a la hora de dibujar existía alguna imprecisión, pero en general ha sido una buena experiencia."*
- ✓ *"El segundo método es muy cómodo, tienes un punto de apoyo y poder mover el plano con la mano facilita las tareas."*
- ✓ *"Los colores empleados son muy claros y te dejan seguir la trayectoria para pintar."*
- ✓ *"La integración entre Looking Glass y pintar en 3D es perfecta. Me gusta mover la cabeza para observar el trazado desde varias perspectivas."*

Negativos o aspectos a mejorar:

- ➢ *"Si se mejorase el hardware empleado el margen de fallo humano sería menor y los resultados obtenidos en las puntuaciones serían considerables."*
- ➢ *"Sugerencias: mostrar si la mano está siendo detectada o no en el segundo método, y en el primero que el movimiento del plano pueda ser continuo al mantener la tecla necesaria pulsada."*
- ➢ *"**Para primer y segundo método**: quitar el cursor, empezar el plano en el punto de inicio para cada prueba, línea a tierra como en el tercer sistema, poner letras*

en los colores del plano de dibujo para diferenciar mejor. ***Para métodos con Leap Motion****: mostrar cuando detecta la mano y un candado o algo que indique cuando está bloqueado el plano (método 2).* ***Para todos****: aplicar función de borrar último o borrar todo lo hecho.”*

- *“La Leap Motion ‘tiembla’ cuando dibujo.”*
- *“La Leap es demasiado imprecisa.”*
- *“Me costaba ver la profundidad en la pantalla.”*
- *“El último método me parece algo complejo al no tener un punto de apoyo. Si estás un rato, el brazo empieza a ‘cansarse’, estando un poco débil.”*
- *“En la prueba del plano con interacción de la Leap y la tableta ha habido problemas para posicionarlo, seguramente debido a la propia Leap o su conexión con Unity.”*
- *“El esfuerzo requerido o el trabajo duro que se han indicado son altos, no solo por lo que supone este método (el segundo) sino por la propia dificultad que me supone trabajar en 3D.”*
- *“Me ha parecido muy complicado ejecutar correctamente sólo con la Leap.”*
- *“La rotación del plano con la mano es algo extraña.”*
- *“Mirando desde arriba o abajo no se percibe diferencia en 3D.” **

**Este último comentario se debe al funcionamiento del display como explico a lo largo de la memoria, y es que el display trabajo dependiendo de la perspectiva horizontal, es decir, si nos movemos de un lado a otro, veremos las diferencias del objeto en el espacio 3D. Pero si nos movemos de arriba abajo y viceversa no apreciamos cambios.

9-Bibliografía

[1] Dávid Lakatos, Matthew Blackshaw, Alex Olwal, Zachary Barryte, Ken Perlin, Hiroshi Ishii, "T(ether): Spatially-Aware Handhelds, Gestures and Proprioception for Multi-user 3D Modeling and Animation", 2012 [Vídeo].

Disponible en: http://tangible.media.mit.edu/project/tether/

[2]Rahul Arora, Rubaiat Habib Kazi, Fraser Anderson, Tovi Grossman, Karan Singh, George Fitzmaurice,"Experimental Evaluation of Sketching on Surfaces in VR", 2017 [Vídeo].

Disponible en:https://www.youtube.com/watch?v=AmiijdKlZYI

[3] Rahul Arora, Rubaiat Habib, Tovi Grossman, George Fitzmaurice, and Karan Singh, "SymbiosisSketch: Combining 2D and 3D Sketching for Designing Detailed 3D objects in Situ", 2018 [Vídeo]

Disponible en:https://www.youtube.com/watch?v=CaHnh_q2EIw

[4] Looking Glass Factory, "Looking Glass Factory", 2020 [En línea]

Disponible en: https://lookingglassfactory.com/

[5] Kickstarter, Looking Glass, "The Looking Glass: A Holographic Display for 3D Creators", 2018 [En línea]

Disponible en: https://www.kickstarter.com/projects/lookingglass/the-looking-glass-a-holographic-display-for-3d-cre

[6] Looking Glass Factory, "Holoplay Unity Plugin", 2020 [En línea]

Disponible en: https://docs.lookingglassfactory.com/Unity/

[7] Looking Glass Factory, "Developing for LEAP on Looking Glass", 2020 [En línea]

Disponible en: https://docs.lookingglassfactory.com/Unity/Leap/

[8] Leap Motion, "Leap Motion´s Unity SDK", 2020 [En línea]

Disponible en: https://leapmotion.github.io/UnityModules/index.html

[9] Leap Motion, "Leap Motion API Reference", 2020 [En línea]

Disponible en: https://developer-archive.leapmotion.com/documentation/v2/csharp/api/Leap_Classes.html

[10] ShowLeap, "Leap Motion (I): características técnicas", 2015 [En línea]

Disponible en: https://www.showleap.com/2015/04/20/leap-motion-caracteristicas-tecnicas/

[11] Looking Glass Factory, "Holoplay.js", 2020 [En línea]

Disponible en: https://docs.lookingglassfactory.com/Holoplayjs/

[12] Wacom, "Wacom Intuos", 2020 [En línea]

Disponible en: https://www.wacom.com/es-es/products/pen-tablets/wacom-intuos

[13] QuéTabletaGráfica.com, "Wacom Intuos S", 2020 [En línea]

Disponible en: https://quetabletagrafica.com/wacom-intuos-s/

[14] Unity, "Unity Documentation – Scripting API", 2020 [En línea]

Disponible en: https://docs.unity3d.com/ScriptReference/

[15] NASA, "NASA TLX: Task Load Index" , 2020 [En línea]

Disponible en: https://humansystems.arc.nasa.gov/groups/TLX/index.php

[16] Page Laubheimer,"Beyond the NPS: Measuring Perceived Usability with the SUS, NASA-TLX, and the Single Ease Question After Tasks and Usability Tests", 2018 [En línea]

Disponible en: https://www.nngroup.com/articles/measuring-perceived-usability/