UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Estudio y experimentación de dispositivos de interacción avanzados sobre Unity 3D: PSMove

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Ferran Devesa Marco

Tutor: Manuel Agustí Melchor

Curso 2018-2019

Resum

Els dispositius d'interacció estan en constant evolució, des de la creació dels primers ordinadors s'han desenvolupat múltiples maneres d'interactuar amb ells, començat pel ratolí i el teclat fins a arribar als sistemes de reconeixement de veu. Això s'aplica també al món dels videojocs, en què sempre s'ha buscat la manera de fer-los més immersius, ja fos simulant armes i vehicles a les antigues recreatives o buscant maneres de perfeccionar el control del personatge amb la incorporació dels *joysticks*. Tot aquest ímpetu per millorar la tecnologia i fer els videojocs més realistes desemboca en els actuals dispositius de realitat virtual. Aquests s'utilitzen a dia d'hui en sectors molt variats tant lúdics com formals, inclouen la medicina i l'educació.

En aquest projecte, s'estudiarà aquesta evolució i es mostrarà que alguns dispositius, com el *PSMove*, que es podrien considerar obsolets segueixen sent punters. Aquest, junt amb una càmera, detecta els moviments de l'usuari i permet replicar-los, el que obri un gran ventall de possibilitats d'ús.

S'analitzarà la viabilitat del *PSMove* com a dispositiu d'interacció avançada per al desenvolupament de videojocs. Per a això, s'estudiarà la seua connexió en dispositius no natius i es crearan una sèrie de prototips que experimentaran amb les possibilitats d'aquest. Incloent, un joc en el qual s'empunyarà una espasa làser, un duel de naus espacials i un joc per fer exercici.

Paraules clau: Unity, Realitat Virtual, PSMove, Joc Seriós, Videojoc , Interacció Avançada

Resumen

Los dispositivos de interacción están en constante evolución, desde la creación de los primeros ordenadores se han desarrollado múltiples maneras de interactuar con ellos, comenzado por el ratón y el teclado hasta llegar a los sistemas de reconocimiento de voz. Esto se aplica también al mundo de los videojuegos, en el que siempre se ha buscado la manera de hacerlos más inmersivos, ya fuese simulando armas y vehículos en las antiguas recreativas o buscando maneras de perfeccionar el control del personaje con la incorporación de los *joysticks*. Todo este ímpetu por mejorar la tecnología y hacer los videojuegos más realistas desemboca en los actuales dispositivos de realidad virtual. Estos se utilizan a día de hoy en sectores muy variados tanto lúdicos como formales, incluyen la medicina y la educación.

En este proyecto, se estudiará dicha evolución y se mostrará que algunos dispositivos, como el *PSMove*, que se podrían considerar obsoletos siguen siendo punteros. Este, junto con una cámara, detecta los movimientos del usuario y permite replicarlos, que abre un gran abanico de posibilidades de uso.

Se analizará la viabilidad del *PSMove* como dispositivo de interacción avanzado para el desarrollo de videojuegos. Para ello, se estudiará su conexión en dispositivos no nativos y se crearán una serie de prototipos que experimentarán con las posibilidades de este. Incluyendo, un juego en el que se empuñará una espada láser, un duelo de naves espaciales y un juego para hacer ejercicio.

Palabras clave: Unity, Realidad Virtual, PSMove, Juego Serio, Videojuego, Interacción Avanzada

Abstract

Human Interface Devices or HIDs are continuously evolving, since the creation of the first computers many ways of interacting with them have been developed. From the the use of mouse and keyboard to voice recognition system used in modern devices. This has also affected the videogame world, where there has also been a constant seek to make games more immersive. Simulating weapons and vehicles in old arcades or improving the control of in-game characters with the introduction of the joysticks among others. All this momentum towards developing this area of technology and making games more vivid has lead to nowadays virtual reality devices. These are used today in many different sectors, including entertainment, medicine and education.

In this project, this evolution will be studied and will show that some devices, such as the *PSMove*, which could seem obsolete are indeed still viable. Together with a camera, this controller is able to detect the player's movements and replicate them, which opens a wide spectrum of usage possibilites.

The feasibility of the *PSMove* as a HID in the area of game development will be analysed. To achieve this, the connection with non-native devices will be studied and a series of prototypes will be developed, each using a different approach to the possibilities this controller offers. Including, a game where a lightsaber will be wielded, a spaceship duel and a game to exercise.

Key words: Unity, Virtual Reality, PSMove, Serious Game, Videogame, Advanced Interaction

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

Desde la aparición de los primeros videojuegos cómo *Bertie the Brain*, *Tennis for Two* y *Spacewar!* la evolución del sector ha sido incesante. Siempre se ha buscado la manera de implementar nuevas mecánicas y de llevar al límite la tecnología. A día de hoy, existen muchas opciones para jugar a videojuegos, pero muchos creen que el futuro está en la realidad virtual. Esta se define como la «representación de escenas o imágenes de objetos producida por un sistema informático, que da la sensación de su existencia real» [1] ya sea en un entorno similar o diferente al nuestro.

Siempre se han buscado alternativas al *joystick*, los juegos de disparos en las antiguas salas recreativas o los primeros dispositivos de realidad virtual como la *Virtual Boy* de Nintendo. Todos ellos muestran que desde el principio se han estado buscando maneras de hacer los videojuegos más inmersivos.

Actualmente, los videojuegos que utilizan cascos de realidad virtual o *HMD* (*Head-Mounted Display*) están presentes en el sector, sin embargo no acaban de destacar respecto a los juegos tradicionales. Esto se debe principalmente al precio. El precio de estos dispositivos es más alto que el de una consola. Además, en el caso de *PlayStation VR* se debe tener la consola para para poder utilizarlo y en el caso de *Oculus Rift* o *HTC VIVE* se necesita un ordenador potente para que funcione correctamente. Es decir, adentrarse en el mundo de la realidad virtual no es barato.

No obstante, hay juegos que son realmente conocidos y han triunfado en este sector del mundo de los videojuegos. Propuestas como *Beat Saber* o *SuperHOT* han hecho que cada vez más personas estén dispuestas a comprarse un dispositivo de realidad virtual.

Por otro lado, esta tecnología se utiliza en aumento en el desarrollo de *Serious Games*. Estos juegos no tienen como fin la diversión, sino que se utilizan en aspectos más formales como la medicina y la educación. Son programas, que utilizando unas mecánicas similares a las de los videojuegos, fomentan el crecimiento y progreso en apartados muy concretos como el proceso de rehabilitación.

La idea de este proyecto es descubrir si utilizando alternativas mucho más accesibles como es el caso de *PSMove* junto a una cámara web, se puede disfrutar también de esta tecnología. Explorando específicamente el sector de desarrollo de videojuegos. Para ello, se crearan varios prototipos que exploren las diferentes características que ofrece este mando.

El motor de videojuegos que se utilizará es *Unity*, ya que aparte de ser gratuito es un entorno muy potente en el que se han desarrollado juegos premiados tanto en el sector de juegos tradicionales como en el sector de la realidad virtual. Entre ellos destaca *Beat*

[1] Diccionario de la Real Academia Española

Saber que, como se ha mencionado anteriormente, es un incentivo para la compra de esta tecnología.

1.1 Objetivos

El objetivo principal de este proyecto es obtener una conclusión respecto a la validez de utilizar el dispositivo *PSMove* como tecnología de desarrollo de videojuegos contrastando con las alternativas actuales cómo el *Oculus Rift* o el *HTC VIVE*. Para alcanzar esta respuesta se han definido unos objetivos menores:

- Estudiar la evolución de la realidad virtual y los dispositivos de interacción en el ámbito de los videojuegos.
- Comparar los dispositivos que se utilizan a día de hoy en el contexto de la interacción para realidad virtual.
- Analizar los diferentes modos en los que se puede conectar el *PSMove* a un ordenador.
- Estudiar la conexión en los sistemas operativos: *MacOS* y *Windows*.
- Experimentar con la interacción que ofrece el *PSMove* a través de las bibliotecas existentes.
- Estudiar la usabilidad del *PSMove* dentro del entorno de desarrollo *Unity*.
- Desarrollar varios prototipos utilizando el abanico de posibilidades que ofrece *PSMove*.
- Explorar la conexión del *PSMove* con el servicio de realidad virtual para *smartphone Trinus VR*.

Cuando se logren estos objetivos se podrá obtener un veredicto que demuestre si realmente el *PSMove* está a la altura de los demás dispositivos en cuanto a capacidad de interacción y desarrollo.

1.2 Estructura de la memoria

La memoria se estructurará en 6 capítulos, empezando por este como presentación de las bases y objetivos. En el segundo capítulo, se presentará la evolución de la tecnología de realidad virtual y cómo ha influido en el desarrollo de videojuegos. Además, se hará una comparativa de los aparatos más utilizados a día de hoy y dónde se sitúa el *PSMove*.

Seguidamente, se estudiará de manera más exhaustiva el *PSMove* en el tercer capítulo. Se expondrán las alternativas de las que se dispone actualmente para utilizar el mando en dispositivos no nativos y se evaluará la conexión en los sistemas operativos *MacOS* y *Windows*. Por último, se describirá el proceso de implementación con *Unity*.

En el cuarto capítulo, se mostrarán los diferentes prototipos creados y su proceso de desarrollo. Se explicará cómo en cada prototipo se ha experimentado con una funcionalidad distinta para cubrir todas las posibilidades que ofrece el *PSMove*. También, se desglosaran las facilidades y/o problemas que se han encontrado.

Para concluir el trabajo, en el quinto capítulo, se pondrán a prueba los prototipos y se evaluarán las opiniones de los usuarios. Asimismo, se enumerarán los posibles trabajos futuros.

CAPÍTULO 2
Estado del arte

En este capítulo, se va a hablar sobre los dispositivos de interacción avanzados que han ido apareciendo en el mundo de los computadores y concretamente en el de los videojuegos. Los dispositivos de interacción humana o *Human Interface Device (HID)* son «dispositivos informáticos que interactúan directamente con los humanos, tanto al recibir información de los mismo como proporcionándosela» [2]. Estos han estado en constante evolución desde que se crearon los primeros ordenadores. En un principio, se trabajaba únicamente con el teclado, más tarde se complementó junto con el ratón y hoy en día se puede utilizar un ordenador únicamente por comandos de voz. Los dispositivos de interacción avanzada, potencian alternativas más creativas o concretas para trabajar con los ordenadores. Aparatos que replican con más precisión la funcionalidad para la que han sido desarrollados, por ejemplo los simuladores de coche que imitan a la perfección el asiento del conductor.

En el sector de los videojuegos, se han utilizado los dispositivos de interacción desde que la tecnología lo ha permitido. En el auge de las recreativas, se desarrollaron muchos videojuegos que iban más allá del uso de la palanca de mando. Estos cubrían una gran cantidad de diversos géneros, desde juegos de disparos hasta juegos rítmicos. Durante este periodo, se experimentó con diversos acercamientos a la inmersión del jugador dentro del juego. Lo más común era replicar objetos de la vida real que los jugadores podían utilizar para interactuar con el juego, por ejemplo las imitaciones de armas, vehículos e instrumentos musicales (Figura 2.1).

Más tarde, con la aparición de la realidad virtual y la realidad aumentada, se han desarrollado otras maneras de interacción que le ofrecen al usuario una experiencia única. A diferencia de la realidad virtual, como se ha explicado en la introducción, la realidad aumentada se define como «la experiencia interactiva que basandose en el mundo real, lo realza añadiendo información generada por ordenador» [3]. Juegos como el *Pokémon GO* [2] han revolucionado esta industria ya que actualmente un elevado número de personas disponen de un *smartphone* y es lo único que hace falta para poder disfrutar de esta tecnología. La realidad virtual también se está utilizando en ámbitos muy diversos, desde el desarrollo de videojuegos hasta el entrenamiento de pilotos [3]. Por lo tanto, no es descabellado pensar que el futuro de los dispositivos de interacción avanzados estará enfocado en el uso de esta tecnología.

[1] Imágenes extraídas de https://medium.com/@gwobcke/outrun-a-1986-arcade-game-by-am2-sega-a2485a709cf7 y https://arcadeclassics.com.au/arcade_games/gun_games/house-of-the-dead-2/ respectivamente.

[2] https://www.pokemongo.com/es-es/

[3] http://bsmotion.com/noticias/2017/8/25/entrenamiento-de-pilotos-ante-situacin-crtica

Figura 2.1: Ejemplos de recreativas interactivas. Izquierda, *Outrun* juego de carreras desarrollado en 1986. Derecha, *House of the dead* 2 juego de disparos desarrollado en 1998. [1]

No obstante, es importante investigar como se ha llegado hasta este punto. Para ello, en la siguiente sección, se va a estudiar como han evolucionado los dispositivos de interacción junto a los videojuegos.

2.1 Evolución de la tecnología

En este apartado, se volverá la vista atrás para ver como han cambiado los dispositivos de interacción en los videojuegos. Se empezará viendo como eran los mandos que se utilizaban para jugar al *Pong* y se finalizará explorando la tecnología más novedosa que se usa hoy en día. No se van a cubrir todos los dispositivos desarrollados ya que la lista se volvería muy extensa, sin embargo se van a mencionar los que se han considerado más importantes en la evolución de la industria.

Pong [4] fue desarrollado en 1972 y es a día de hoy uno de los juegos mundialmente más conocidos. Para controlar las barras laterales se utilizaban dos diales y cada uno controlaba una de ellas (Figura 2.2), se podían rotar a izquierda y derecha para especificar el movimiento de la barra. Estos se pueden considerar unos de los primeros dispositivos de interacción en el mundo de los videojuegos y sirvieron para asentar las bases de sus sucesores.

Aunque la primera patente del *joystick* o palanca de mando se crease en 1926 como mando de control remoto de un aeroplano [5], la consola que estableció las bases para su uso en el mundo de los videojuegos fue la *Atari 2600* [6]. Su mando consistía de un *joystick* con movimiento en ocho direcciones y un único botón (Figura 2.3). Aún así, fue un estándar a seguir durante muchos años hasta dar paso a las crucetas o *D-Pad*.

El establecido diseño de las crucetas [7] fue desarrollado por *Nintendo* en 1982, concretamente por Gunpei Yokoi el creador de la consola portátil *Game Boy* [8]. La cruceta se creó como alternativa al *joystick* en la primeras consolas portátiles, específicamente en la *Game and Watch* (Figura 2.4). Esta solución funcionó tan bien, que se sigue utilizando a día de hoy en los mandos más modernos.

[4] Imagen extraída de https://en.wikipedia.org/wiki/Paddle_(game_controller)
[5] Imagen extraída de https://hipertextual.com/2011/05/historia-tecnologica-atari-2600-y-adventure

Figura 2.2: Diales utilizados en el videojuego *Pong* [4].

Figura 2.3: Consola *Atari 2600* y su *joystick* [5].

Figura 2.4: Consola *Game and Watch* [6].

[6]Imagen extraída de https://www.vinted.es/enfants/jeux-electroniques/150119727-nintendo-game-and-watch-1982-green-house

Más tarde, con el lanzamiento de la *Nintendo Entertaiment System* o NES en 1983 [9] se introdujeron los botones *start* y *select* que también siguen utilizándose actualmente (Figura 2.5). Seguidamente, en 1990 se lanzó al mercado su sucesora, la *Super Nintendo* o *SNES* [10]. En esta, se introdujeron dos notables cambios, los gatillos o *triggers* y las dos filas de botones, que también se acabarían incorporando en los mandos futuros. Además, en el diseño de este mando se empezó a destacar la importancia de la ergonomía ya que se redondearon los bordes para hacerlo más cómodo de sujetar (Figura 2.5).

Figura 2.5: Izquierda, consola *Nintendo Entertainment Sysetm*. Derecha, consola *Super Nintendo* [7].

La *virtual Boy*, lanzada en 1995, fue una de las primeras consolas en experimentar con la realidad virtual [11]. Utilizaba unas gafas con un estereoscopio incorporado para simular un efecto en tres dimensiones. Aunque la consola fuese tal fracaso que no se llegase a comercializar en Europa, fue un primer acercamiento a utilizar la realidad virtual en las consolas de sobremesa. No obstante, incorporó varias mejoras en su mando que sí se imitaron en consolas futuras. Se añadieron dos crucetas, una a cada lado del mando y se perfeccionó todavía más la ergonomía añadiendo dos agarres para reposar mejor las manos (Figura 2.6).

Posteriormente, en 1996, salió al mercado la *Nintendo 64* [12] que adaptó el clásico *joystick* creando el primer *stick* analógico (Figura 2.7), este surgió de la necesidad de moverse por entornos en tres dimensiones que ya eran capaces de implementarse gracias a la recién incorporada tecnología de procesadores de 64-bits.

El siguiente gran cambio ocurrió con la *PlayStation* [15]. Aunque saliese al mercado en 1994, en ese entonces el mando que utilizaba la consola, el *DualShock* (Figura 2.8), era similar al de la *Super Nintendo*, pero con una ergonomía mucho más lograda. No fue hasta 1997, que gracias a la necesidad de crear un mando para jugar a los juegos en tres dimensiones, se creó el reinventado *DualShock*. Este incluía dos *sticks* analógicos y además incorporaba la vibración para darle más inmersión al jugador.

Aunque las consolas de sobremesa se habían establecido en el mercado, las recreativas seguían desarrollando nuevas manera de entretener a los jugadores. En 1998 se lanzó al mercado el primer *Dance Dance Revolution* [14], este es un juego de baile en el que

[7] Imágenes extraídas de https://theglobalesportsacademy.com/historia-del-videojuego-como-arte-y-practica-depor
nes/ y https://www.amazon.com.mx/Nintendo-CLVSSNSG-Super-Classic-Edition/dp/B0721GGGS9 respectivamente.

[8] Imagen extraída de https://es.wikipedia.org/wiki/Virtual_Boy

[9] Imagen extraída de https://es.wikipedia.org/wiki/Nintendo_64

[10] Imágenes extraídas de https://articulo.mercadolibre.com.ar/MLA-751854066-joystick-de-play-1-nuevo-_
JM y https://www.amazon.com/Sony-Playstation-DualShock-Controller-Gray-Pc/dp/B00002DHER respectivamente.

Figura 2.6: Consola *Virtual Boy* con mando [8].

Figura 2.7: Consola *Nintendo 64* con mando [9].

Figura 2.8: Mandos *Dualshock*. Izquierda, primera versión. Derecha, segunda versión.[10].

se utilizaban los pies para jugar. Disponía de cuatro flechas apuntando cada una hacia uno de los puntos cardinales, como si fuese una cruceta gigante (Figura 2.9). El juego

consistía en pisar sobre las flechas siguiendo la secuencia correspondiente que aparecía en la pantalla. Este juego mostró a los usuarios que aún habían manera de innovar en esta industria.

Figura 2.9: Recreativa *Dance Dance Revolution* [11].

En 1998, salió la *Dreamcast* [16] que implementó una ranura en su mando en la que se podían meter distintas modificaciones (Figura 2.10). Entre ellas, una segunda pantalla para mostrar alguna información del juego o un sistema de vibración. Esta idea, como se verá más adelante, se perfeccionó y se implementó en la *Wii U*.

Figura 2.10: Consola *Dreamcast* con mando [12].

Un avance muy importante en cuanto a dispositivos de interacción dentro del mundo de los videojuegos fue la *Eye Toy* (Figura 2.11). Desarrollada para *PlayStation 2* y sacada al mercado en 2003 [17], permitía al usuario interactuar con los juegos utilizando movimientos corporales y colores. Fue un primer contacto con esta tecnología, que más adelante se

[11] Imagen extraída de https://www.vintagearcade.net/shop/arcade-games/dance-dance-revolution-arcade-game/

[12] Imagen extraída de https://www.lainformacion.com/arte-cultura-y-espectaculos/dreamcast-la-consola-que-acabo-con-el-imperio-de-sega_ShNi5NrN5hgmEhWS6qMbQ1/

mejoró con el *Kinect* (Figura 2.11) [18]. Este se desarrolló en 2010 para *Xbox 360* y añadió más funcionalidades como la capacidad de reconocer objetos y comandos de voz.

Figura 2.11: Arriba, cámaras *EyeToy*. Abajo, cámara *Kinect*.[13].

En 2004, se lanzó al mercado la *Nintendo DS* [19] que revolucionó el mercado de las consolas portátiles. Esta incorporó dos pantallas como las antiguas *Game and Watch*, sin embargo la pantalla inferior era táctil (Figura 2.12). Esto añadió mucha profundidad y libertad a la hora de interactuar con los juegos ya que se permitía dibujar formas o escribir simulando el papel. Más adelante, en 2011, se desarrollo su sucesora, la *Nintendo 3DS* [20] (Figura 2.12), que permitía al usuario visualizar los juegos en tres dimensiones sin necesidad de usar gafas especializadas. Esto se consiguió gracias a la técnica de la autoestereoscopia que permite simular una sensación de profundidad en objetos planos mediante ilusiones ópticas.

Mas adelante, al ser bastante populares los juego rítmicos y musicales en las recreativas, se buscaron maneras de adaptarlos en las consolas de hogar. En 2004, salió al mercado para la *PlayStation 2* el primer *SingStar* [21]. Este utilizaba un micrófono como dispositivo de interacción (Figura 2.13) y calculaba la puntuación comparando la entonación y melodía del jugador con las de la canción que estaba sonando. Un año más tarde, en 2005, se desarrolló la primera versión de *Guitar Hero* para la misma consola [22]. Utilizaba como mando una réplica de una guitarra eléctrica con cinco botones para representar los trastes, uno para simular el punteo de las cuerdas y otro para simular la palanca de

[13]Imágenes extraídas de https://en.wikipedia.org/wiki/EyeToy y https://www.vidaextra.com/hardware/kinect-esta-oficialmente-muerto

[14]Imágenes extraídas de https://www.fantasymundo.com/presentacion-de-nintendo-ds/ y https://es.wikipedia.org/wiki/Nintendo_3DS

Figura 2.12: Izquierda, consolas *Nintendo DS*. Derecha, consola *Nintendo 3DS*.[14].

vibrato (Figura 2.13). Estos juegos fueron muy populares y llegaron a sacar versiones en las que se podían utilizar la guitarra y el micrófono en una misma canción. Además, se desarrollaron nuevos instrumentos como la batería.

Figura 2.13: Consolas *Nintendo DS* y *Nintendo 3DS*.[15].

Con el lanzamiento de la *Nintendo wii* [23] en 2006, se volvió a dar un paso hacia delante en cuanto a la interacción del usuario con la consola. Con la aparición del *Wii Remote* o *WiiMote* (Figura 2.14), el mando característico de esta consola, se le ofrecía al usuario la oportunidad de utilizar este mando como una espada, bate de béisbol o pistola. Utilizando rayos infrarrojos, se detectaba la posición y movimiento del mando y se trasladaba al elemento en pantalla. Años más tarde, en 2011 salió al mercado la sucesora de la *Wii*, la *Nintendo Wii U* [24] que modificó los característicos mandos, por un nuevo mando que incluía una pantalla táctil (Figura 2.14). Este permitía interactuar con los

[15]Imágenes extraídas de https://www.emere.es/Sony-Microfonos-Singstar-PS2-/-PS3-Nuevos y https://www.amazon.co.uk/Guitar-Hero-Stand-Alone-PS2/dp/B000GPB9CM

juegos funcionando como una segunda pantalla o se podía utilizar directamente como pantalla principal.

Figura 2.14: Mandos de la *Nintendo Wii* y *Nintendo Wii U*.[16].

Para competir con el éxito de la *Wii*, en 2010, *Sony* lanzó al mercado el *PSMove* (Figura 2.15) [25] que al igual que el *WiiMote*, permitía replicar los gestos del jugador en pantalla. No obstante, al contrario que este, el *PSMove* no se detectaba por infrarrojos, sino que venía con una cámara, la *PSEye*, que lo detectaba utilizando el color de la bola del mando. Más tarde, con la aparición de los dispositivos de realidad virtual que utilizaban casco, *Sony* le volvió a dar uso a estos mandos y los utilizó para el *PlayStation VR* (Figura 2.15) [26]. Este salió en 2016 y su objetivo era competir con *Oculus Rift* [27] y *HTC Vive* [29] (Figura 2.15) que ya habían sacado sus propios cascos de realidad virtual. Desarrollados también en 2016, siendo el orden de salida *Oculus Rift*, *HTC Vive* y *Playstation VR* siguen siendo a día de hoy los dispositivos que lideran el mundo de la realidad virtual.

Por último, en 2018 *Microsoft* publicó su *Xbox Adaptive Controller* (Figura 2.16), que [30] «dispone de botones programables de gran tamaño y se conecta a conmutadores, botones, soportes y *joysticks* externos para hacer que los juegos sean más accesibles», de esta manera se permite a los jugadores con discapacidades disfrutar también de los videojuegos.

Como se ha podido ver en este repaso a la evolución de los dispositivos de interacción enfocados a videojuegos, la tecnología ha cambiado mucho y seguramente lo siga haciendo en un futuro. Se ha podido comprobar que actualmente se dispone de un gran número de alternativas, que aún siendo antiguas, se pueden seguir utilizando para desarrollar videojuegos. En la siguiente sección, 2.2, se van a comparar algunos de estos dispositivos más a fondo para evaluar y defender la elección de utilizar el *PSMove*.

[16]Imágenes extraídas de https://www.walmart.com/ip/White-Wireless-Remote-Wiimote-Nunchuck-Controller-Combo-Set-483019465 y https://articulo.mercadolibre.com.mx/MLM-641359820-control-wii-u-gamepad-para-nintendo-original-n-JM

[17]Imágenes extraídas de https://www.profesionalreview.com/2017/10/02/sony-lanza-nuevo-sistema-psvr/, https://www.amazon.com.mx/Sistema-realidad-virtual-Oculus-Touch/dp/B073X8N1YW y https://www.amazon.in/HTC-VIVE-Virtual-Reality-System/dp/B06ZY6LJ2F respectivamente.

[18]Imagen extraída de https://www.xbox.com/es-ES/xbox-one/accessories/controllers/xbox-adaptive-controller

2.2 Comparativa entre las alternativas actuales

Como se ha visto en la sección anterior, hoy en día hay muchas opciones disponibles que se pueden utilizar para el desarrollo de videojuegos. En este apartado, se analizarán las que se han considerado competidores más directos del *PSMove*. Estas son: el *WiiMote*, el *Oculus Rift*, el *HTC Vive* y el *PlayStation VR*.

En primer lugar, el *WiiMote* [23] ofrece prácticamente las mismas funcionalidades que el *PSMove* con la diferencia de que este se controla utilizando sensores infrarrojos y el otro se detecta mediante una cámara. Sin embargo, la barra infrarroja no se puede conectar directamente al ordenador y esto dificulta su conexión.

En segundo lugar, el *Oculus Rift* [28] combina la tecnología del estereoscopio con el infrarrojo. Dentro del casco, se incluyen además unos altavoces para dar una mayor sensación de inmersión. Los movimientos se detectan utilizando un sistema infrarrojo, llamado *Constellation*, este se debe colocar en varios puntos para generar un área de juego. Para el movimiento del jugador, se utilizan dos mandos, uno para cada mano, que al igual que el casco se detectan mediante infrarrojos. Además, incluye un sistema de reconocimiento de gestos de los dedos. No obstante, la mayor desventaja es que el casco debe esta conectado por cable al ordenador y puede limitar el movimiento del jugador.

En tercer lugar, el *HTC Vive* [29] es muy similar al *Oculus Rift* respecto al tema de detección del casco y los mandos ya que también utiliza un sistema infrarrojo. Sin embargo, añade una cámara delantera al casco que permiten ver el exterior en caso de que se necesite. Los mandos no incluyen un sistema de reconocimiento de gestos, pero en su defecto utilizan un *trackpad*. Al igual que el *Oculus Rift*, este casco va conectado por cable al ordenador.

Por último, el *PlayStation VR* [26] utiliza también la tecnología del estereoscopio para generar esa profundidad. Pero al contrario que sus competidores utiliza una única cámara que detecta el casco utilizando los *LEDs* que están distribuidos en él. Los mandos que usa son los *PSMove*, que se detectan de la misma manera. Como el *Oculus Rift* y el *HTC Vive*, este se conecta a la *PlayStation 4* mediante un cable.

Como se puede observar, aún siendo un dispositivo que se lanzó al mercado en 2010, el *PSMove* se sigue utilizando junto al *PlayStation VR*. Esto se debe a que a día de hoy, sigue ofreciendo un sistema de detección muy completo y una ergonomía bastante lograda. Además, ofrece una alternativa mucho más accesible respecto a lo dispositivos que utilizan *Head-Mounted Display*.

Figura 2.15: Arriba, mandos *PSMove* junto a los demás componentes del *PlayStation VR*. En medio, casco de realidad virtual *Oculus Rift*. Abajo, cascos de realidad virtual *HTC Vive* [17].

Figura 2.16: Mando *Xbox Adaptative Controller* [18].

CAPÍTULO 3
Experimentación y conexión del PSMove

Tras desglosar los diferentes dispositivos de los que se dispone actualmente para experimentar y desarrollar con esta tecnología, el siguiente objetivo es indagar en las funcionalidades que ofrece el *PSMove*. Se han encontrado varias alternativas para utilizar este mando fuera de sus dispositivos nativos, la *PlayStation 3* y la *PlayStation 4*. En este apartado, se hará una comparativa y se argumentará la selección final. En un principio, la idea era desarrollar este proyecto en un *Macbook pro* con *macOS* porque así se tenía la ventaja de la portabilidad. Por esta razón, se buscaron, en mayor medida, soluciones que funcionasen en este dispositivo. Además, se quiere utilizar la cámara del portátil para estudiar la viabilidad de usar una alternativa a la cámara oficial, la *PSEye*. Más adelante, en la sección 3.2, se verá que se ha acabado desarrollando en *Windows 10*.

En primer lugar, como se conocía de antemano que se iba a utilizar *Unity* para desarrollar los diferentes prototipos, lo más lógico era buscar en la *Unity Asset Store* [1]. *Unity* explica en su manual [31] que un «*asset* es una representación de cualquier ítem que pueda ser utilizado en el juego o proyecto». Luego, la *Unity Asset Store* es una plataforma donde se pueden encontrar *assets* tanto gratuitos como comerciales creados por *Unity Technologies* o por miembros de la comunidad.

Utilizando esta plataforma se encontró únicamente una herramienta relacionada con el *PSMove*, el *PlayStation Move Wrapper for Move.me*. Esta permitía obtener los valores del mando mediante el software *Move.me*. *Move.me* se desarrolló en 2011 como herramienta de desarrollo en ordenador que utilizaba la *PlayStation 3* junto con el *PSMove* y la *PSEye* para enviar los valores del mando (como la posición y orientación) al ordenador mediante un servidor dedicado. *PlayStation Move Wrapper for Move.me* obtenía estos valores para utilizarlos directamente en *Unity*. Aún siendo gratuita, para poder utilizarla se necesita una *PlayStation 3* con la aplicación *Move.me* instalada, que significaba una inversión extra de 99, 99 $ [2]. Como se requerían muchos extras, esta herramienta quedó descartada.

Eliminadas las posibilidades de encontrar algo por la *Unity Asset Store* se buscaron otras alternativas no oficiales. Las 3 librerías más populares eran: *PSMoveApi*, *PSMove-Unity5* y *PSMoveService*. Investigando más a fondo se encontró que *PSMove-Unity5* y *PSMoveService* partían de *PSMoveApi* como base. Por lo tanto, la primera librería que se utilizó fue *PSMoveApi*.

PSMoveApi se define en su *GitHub* [32] como «una librería de código abierto para Linux, MacOS y Windows para acceder al *PSMove* via Bluetooth o USB directamente

[1] https://assetstore.unity.com/

[2] Precio obtenido de https://store.playstation.com/en-us/product/UP9002-NPUO00014_00-MOVEMESERVER0000

desde el ordenador sin necesidad de una *PlayStation 3*. [...] Utilizando la *PSEye*, *iSight* o cualquier otra cámara». Esto apoya la idea de buscar una alternativa más económica a la realidad virtual porque permite utilizar la cámara web que se prefiera, desde una cámara que venga por defecto en un ordenador portátil hasta una cámara más antigua a la que no se le daba uso. La *PSMoveApi* no está pensada directamente para utilizarse con *Unity*, se encarga, como he mencionado en el párrafo anterior, de obtener los valores del mando, permitiendo al usuario procesar esta información como se necesite. La librería está escrita en C y los programas de ejemplo que trae se ejecutan en el terminal. Sin embargo, con el tiempo se han añadido más lenguajes incluyendo *Python* y *Java*. En la sección 3.1, se explicará con más detalle el proceso de instalación y compilación. Además, se ejecutarán algunos de los ejemplos que trae para mostrar como utilizarla. Aunque desde un principio no estuviese pensada para trabajar con entornos de desarrollo de videojuegos, sí que se han añadido funcionalidades para permitirlo. Esto se expondrá más a fondo en la sección 3.2.

PSMove-Unity5 es una librería que se utiliza para «iterar con aplicaciones/juegos de realidad virtual cuando se tiene acceso limitado o directamente ningún acceso a dispositivos más avanzados» como bien se menciona en su *GitHub* [33]. A pesar de que esta aplicación parecía justamente lo que se necesitaba para experimentar con el *PSMove* venía con dos problemas.

En primer lugar, la aplicación está pensada para ser utilizada únicamente en Windows, como he citado en la introducción de este capítulo, la idea original era desarrollar utilizando *MacOS*. Igualmente, ya que esta librería parecía ser perfecta para lo que se buscaba, se investigó si alguien había conseguido instalarla en este sistema operativo. Aunque sí existe la manera, fallaba a la hora de utilizar las librerías compiladas del *PSMoveApi* que *PSMove-Unity5* necesita para funcionar. Sin embargo, en *MacOS* no se detectaban correctamente ya que este sistema operativo, *Unity* no detecta correctamente las bibliotecas de enlace dinámico (*.dll*) ni las bibliotecas de enlaces dinámico específicas de *MacOS* (*.dylib*), que son las que utiliza *PSMove-Unity5*. Este ha sido un problema constante a la hora de la implementación con *Unity*, pero se explicará mejor en la sección 3.2.

En segundo lugar, al contrario que *PSMoveApi*, esta librería solo funciona con la *PSEye*. Esto no es un problema directamente si se tiene esta cámara, no obstante en este proyecto, como he mencionado al principio de esta sección, se quería experimentar con la viabilidad de utilizar otras opciones de cámara. Dadas estas dos complicaciones, se decidió descartar esta librería.

Por último, se investigó *PSMoveService*. A diferencia de *PSMoveApi* y *PSMove-Unity5*, esta no es una librería sino un servicio. En su *GitHub* [34] se explica como «un servicio que maneja múltiples mandos *PSMove* y cámaras *PSEye*. Los clientes se conectan al servicio y este se encarga de obtener la información del mando». Los creadores de esta herramienta son los mismos que los de *PSMove-Unity5*, sin embargo *PSMoveService* está enfocada sobre todo hacia utilizar los mandos en aplicaciones o juegos ya existentes no hacía herramienta de desarrollo. Además, tiene el mismo problema que ocurre en *PSMove-Unity5*: solo funciona con la cámara oficial. Encontrados estos contratiempos no se utilizó esta opción tampoco.

En retrospectiva, la única alternativa que cubre todas las necesidades que se buscan era *PSMoveApi*, entre ellas la capacidad de poder utilizarse en entornos de desarrollo de videojuego, y por esta razón ha sido la que se ha utilizado finalmente. Asimismo, esta librería se puede utilizar en todos los sistemas operativos, como se mostrará en la siguiente sección.

3.1 Instalación en los distintos sistemas operativos

En esta sección se expondrá el proceso que se ha seguido para instalar la *PSMoveApi* en los diferentes sistemas operativos, empezando por *MacOS*. Se dividirá en tres partes: descarga y compilación de la librería, conexión del *PSMove* y configuración de la cámara web.

3.1.1. MacOS

Se han seguido dos guías para la instalación de *PSMoveApi*, la guía oficial [35] y otra obtenida del grupo de *PSMoveApi* de Google [36], no obstante se han tenido que hacer algunas pequeñas adaptaciones para que funcionase todo correctamente. Como se ha mencionado, se empezará hablando de dónde se descarga esta librería. Pero antes, se tiene que instalar *Homebrew*. *Homebrew* [37] es un gestor de paquetes para *MacOS* el cual facilitará la tarea de instalar los programas extra necesarios. Estos son: *CMake, Libtool* y *Automake*. *CMake* y *Automake* se complementan y se utilizan para la compilar, evaluar y empaquetar software. Por otro lado, *Libtool* simplifica el trabajar con librerías compartidas.

Una vez instalados los programas necesarios, se tiene que descargar *PSMoveApi*. Esta librería se compone de código abierto y todo sus recursos se pueden encontrar en su *GitHub* [32]. Se ha hecho uso de esto y se ha clonado el repositorio. No obstante, la librería tiene algunas dependencias que han de instalarse para que funcione todo correctamente. Para lograrlo hemos utilizado el siguiente comando:

```
1  $ git clone git://github.com/thp/psmoveapi.git
2  $ cd psmoveapi
3  $ git submodule init
4  $ git submodule updates
```

Seguidamente, la librería incluye un *script* para automatizar el proceso.

```
1  $ bash -e -x scripts/macos/build-macos
```

Durante la primera ejecución saltó un error. El problema se debía a que no se tenían los *command line tools* de *MacOS* instalados. Para solucionar esto se ejecutó el siguiente comando:

```
1  $ xcode-select --install
```

Con esto la librería ya está instalada. Para comprobar que todo se ha instalado de manera satisfactoria, se ha conectado el mando por USB al ordenador y se ha ejecutado el programa de ejemplo. Hecho esto, se ha iluminado la luz del mando y ha vibrado durante unos segundos.

Uno de los problemas de la *PSMoveApi* es que las funcionalidades que se pueden procesar cuando se tiene el mando conectado por USB son muy limitas. Por lo tanto, ahora se tiene que configurar el mando por *bluetooth* para ser capaz de utilizar el mando a su máximo potencial.

Lo primero que se tiene que hacer es enlazar el *PSMove* con el ordenador en el que se va a utilizar. Con el *bluetooth* del ordenador enchufado se utilizando el comando:

```
$ ./psmove pair
```

se devuelve una dirección MAC que hay que utilizar para enlazarlo de manera manual. Para autorizar esta dirección MAC hay que utilizar la siguiente instrucción:

```
$ sudo defaults write /Library/Preferences/com.apple.Bluetooth HIDDevices -
    array-add "<aa-bb-cc-dd-ee-ff>"
```

donde <aa-bb-cc-dd-ee-ff> es la dirección que se ha devuelto previamente. Para probar que ha funcionado, se ejecuta el programa de ejemplo de nuevo y en efecto, el mando se ha enlazado correctamente. Los botones se detectan y la intensidad de la esfera varía dependiendo de la profundidad con la que se apriete el gatillo del mando.

Por último, falta conectar la cámara. Como se está trabajando con un *MacBook pro* se va a utilizar la cámara que viene incorporada, la *iSight*. Para ello, lo primero que se tiene que hacer es modificar el *script* de instalación, *build-macos*, ya que por defecto te instala los *drivers* para usar la *PSEye*. La línea de código que hay que cambiar es:

```
$ cmake -DPSMOVE_USE_PS3EYE_DRIVER=OFF
```

Con el *driver* de la cámara original instalado, cuando se utiliza esta librería siempre se intenta detectar la *PSEye* y si no se encuentra, el programa falla. Es por esto, que no se debe instalar si se quiere utilizar otra cámara. Aunque existen métodos para seleccionar la cámara que se quiere utilizar en el momento de ejecución, como solo se va a utilizar la *iSight* se ha valorado directamente no instalar el driver.

Es importante mencionar que la *iSight* no tiene ninguna manera para poder controlar la exposición manualmente. La cámara funciona de la siguiente manera, cuando se enciende calcula una buena exposición y luego se va modificando automáticamente para ajustarla. El problema, es que de esta manera, se pierde control por parte de la aplicación a la hora de detectar y seguir el movimiento del mando ya que la imagen que captura la cámara no es constante. Por está razón, *PSMoveApi* hace un bloqueo de exposición, es decir, se detecta una buena exposición al iniciarse y se bloquea para que no varíe con el tiempo. Para que la librería lo detecte automáticamente hay que seguir el siguiente procedimiento al ejecutar una aplicación que la utilice:

1. La esfera del *PSMove* se enciendo con un color muy brillante.
2. Se tapa la cámara con la esfera.
3. Se utiliza la exposición que detecta la cámara en ese momento.
4. Se mueve el mando hacia atrás y se inicia la calibración normal.

De esta forma, la *iSight* queda bloqueada y la detección del mando mejora notablemente. Una vez completadas las tres sección de la instalación, podemos comprobar que no hay fallos ejecutando desde la carpeta *build*, que se encuentra en el directorio de instalación de la *PSMoveApi*, los siguientes programas:

```
$ ./test_tracker
$ ./test_opengl
```

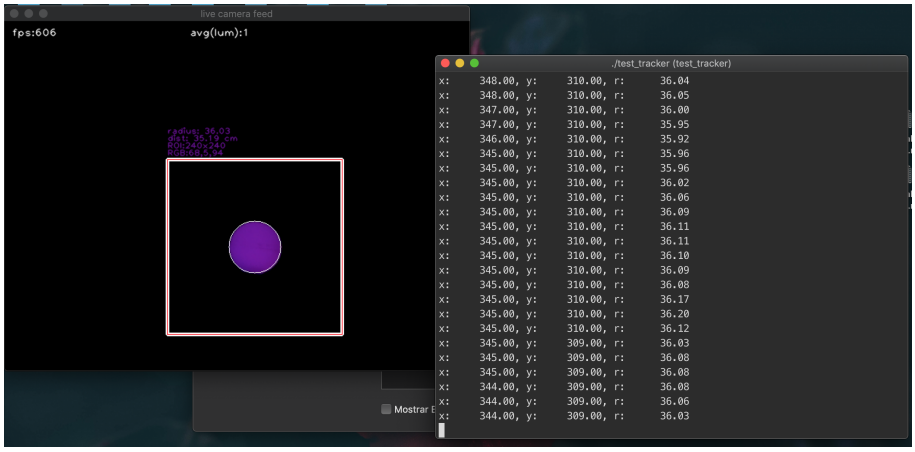

Figura 3.1: Captura de pantalla programa *test_tracker*.

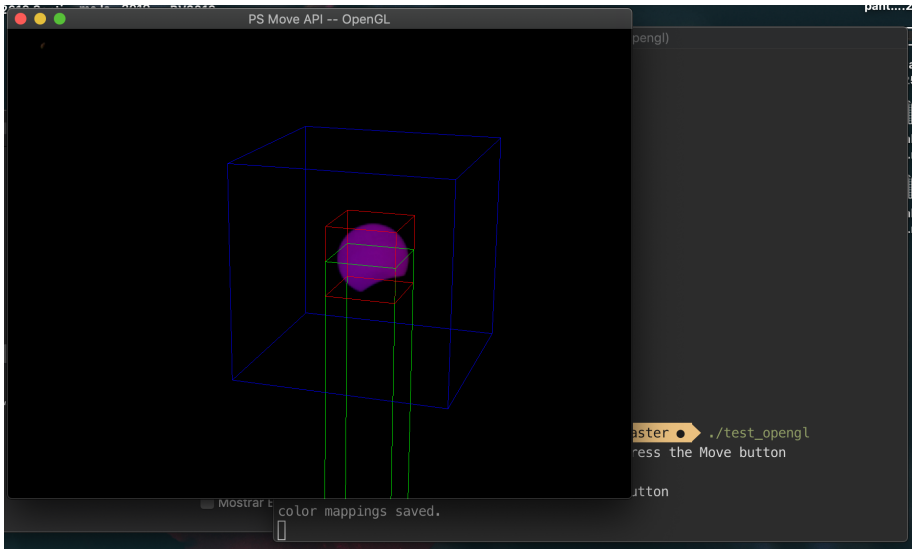

Figura 3.2: Captura de pantalla programa *test_opengl*.

El primero, *test_tracker*, muestra una esfera que replica el movimiento del *PSMove* y se hace más grande o pequeña dependiendo de cómo de cerca esté de la cámara. El segundo, *test_opengl* genera un mando utilizando dos cubos que además de simular el movimiento del mando, también replica la orientación de este. Como se puede ver en las figuras 3.1 y 3.2, la instalación de la *PSMoveApi* ha sido satisfactoria. Además, se ha conectado un segundo mando siguiendo las instrucciones descritas anteriormente. No ha aparecido ningún contratiempo y para comprobar que funciona correctamente, se ha vuelto ha ejecutar el programa *test_tracker*. Indagando en el código de este programa se ha comprobado que genera el mismo número de figuras que de mandos conectados. Se puede apreciar en la figura 3.3 que los resultados son los esperados.

Con esto, se da por concluido el proceso de instalación en *MacOS*, ahora se verá como se desenvuelve en *Windows 10*.

3.1.2. Windows 10

Para instalar *PSMoveApi* en *Windows* se ha seguido la misma guía que para la instalación en *MacOS* [35]. Como prerrequisitos se necesitaban: *Visual Studio* 2017, *Git* y *CMake*. Una vez descargado, la librería ofrece dos opciones para la compilación. En primer lugar,

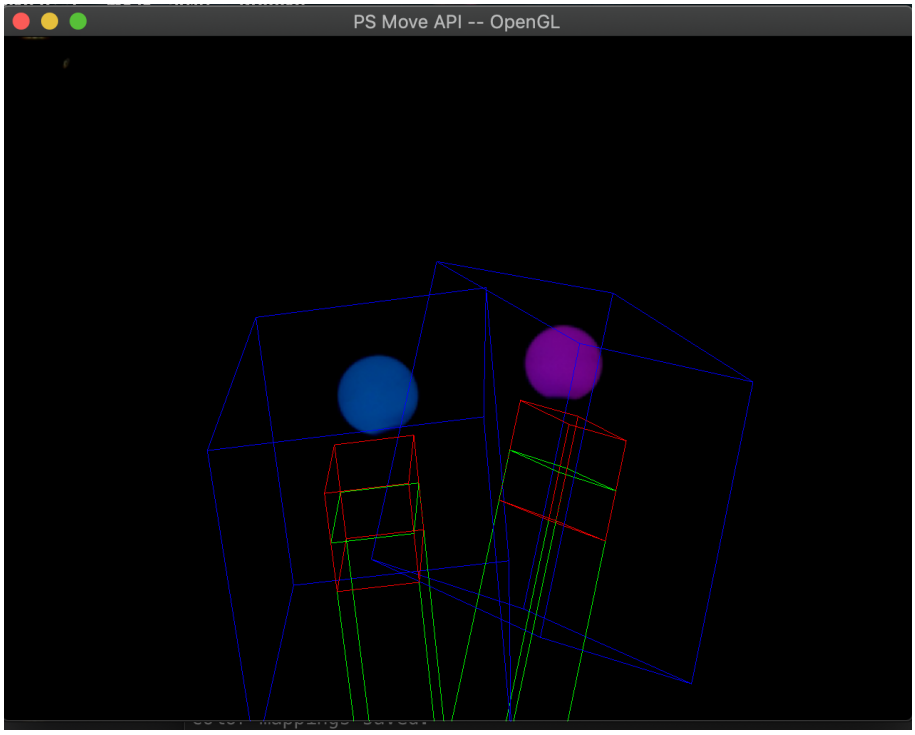

Figura 3.3: Captura de pantalla programa *test_opengl* utilizando dos mandos.

la librería trae un *script* de instalación mediante *Visual Studio*. Este se instala mediante el símbolo de sistema para desarrolladores de *Visual Studio* [38]. En segundo lugar, se puede instalar manual o automáticamente utilizando *MinGW*. *MinGW* es «es una implementación de los compiladores GCC para la plataforma Win32, que permite migrar la capacidad de este compilador en entornos Windows.» [39]. Se instalará utilizando *Visual Studio* ya que te genera además un proyecto que permite modificar y recompilar código.

Para ejecutar los comandos necesarios se necesita utilizar el símbolo del sistema para desarrolladores propio de *Visual Studio*. Este se encuentra buscando *commando prompt* en el propio buscador de *Windows*. Ahora, para ejecutar el *script* de instalación se ejecuta el siguiente comando:

```
1    $ call scripts/visualc/build_msvc.bat 2017 x64
```

Se usará el argumento "x64" ya que se esta trabajando con un procesador de 64-bits. No obstante, al ejecutar este comando han saltado unos errores. En primer lugar, no se detectaba el *SDK* de *Windows* 8,1, al parecer el instalador de *Visual Studio* no lo incluye por defecto. Modificando los parámetros del instalador para añadir este componente solventaba el problema. Sin embargo, esto provocaba otro error que se solucionaba instalando la dependencia *Windows Universal CRT SDK* que tampoco se incluye automáticamente. Al hacer esto, la librería ya estaba lista para usarse.

El siguiente paso es la conexión de los mandos. Para la conexión por USB únicamente hay que conectar el *PSMove* al ordenador. Ejecutando el método de ejemplo se ha podido comprobar que al igual que en *MacOS* el mando ha empezado a vibrar y a iluminarse. La conexión por *bluetooth* es un poco más compleja.

Al principio se utilizaban aplicaciones de terceros como *MotionInJoy*, no obstante la fiabilidad de estas no es muy buena. Por lo tanto, se encontraron dos herramientas que facilitan el proceso: *PSMovePair* y *PSMove-Pair-Win*. Estas son complementarias y se deben utilizar con un orden establecido. Se empieza con *PSMovePair* que te muestra unos pasos

a seguir. Conectar el mando por USB y cuando el ejecutable lo detecte desconectarlo para seguidamente utilizar el botón *PSButton* del mando para que se enlace con el ordenador. La luz piloto del mando parpadeara hasta que se conecte y quede fija. El ejecutable indica que se vuelva a pulsar el botón en caso de que acabe de parpadear y no haya quedado fija la luz. Este proceso puede llevar un tiempo y varias pulsaciones del botón.

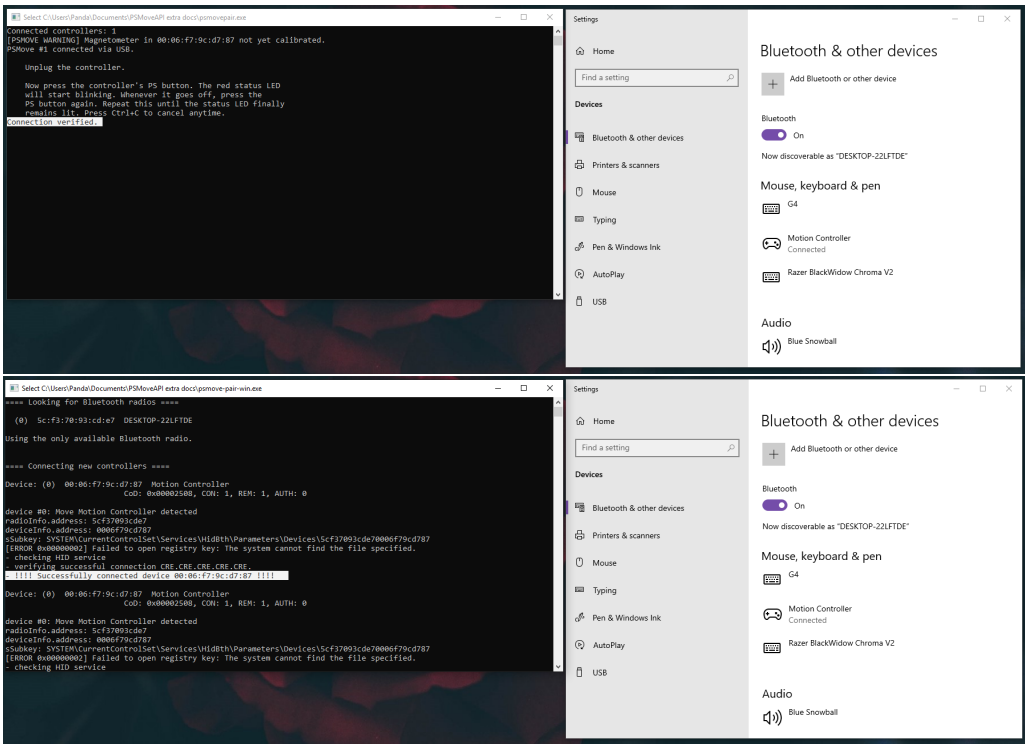

Figura 3.4: Captura de pantalla conexión del *PSMove* en *Windows 10*. Arriba ejecutable *psmove-pair* y abajo *psmove-pair-win*.

Se hizo un pequeño experimento en *Windows* 7 y transcurrieron unos 15 minutos hasta que se detectó la conexión correctamente. Se puede apreciar en la figura 3.5, como el mando se iba conectando y desconectado hasta que acabó por enlazarse. En *Windows* 10 el proceso es más rápido y no se conecta y desconecta.

Cuando *PSMovePair* ha cumplido su función. Se utiliza *PSMove-Pair-Win* para verificar que el mando se ha enlazado satisfactoriamente. Hecho esto, el mando quedará registrado en el ordenador y no hará falta repetir este proceso cada vez que se quiera utilizar. Únicamente será necesario activar el *bluetooth* y pulsar una vez en el botón *PSButton*. Seguidamente se ha seguido el proceso para conectar un segundo mando.

Para comprobar que realmente funciona todo como debe, se ha ejecutado el programa *multiple*, que se encuentra dentro de la carpeta *build-x64* en el directorio de instalación de la *PSMoveApi*. Este se encarga de iluminar todos los mandos disponibles para comprobar que se han enlazado correctamente y tras ejecutarlo ambos mandos han iluminado su bola. Ahora, hay que conectar la cámara.

En contraste con *MacOS* y la *iSight*, en *Windows* se ha utilizado la cámara *EyeToy*. Esta es una cámara que salió al mercado en 2003 para la *PlayStation 2*. Se puede apreciar que es una cámara que ya se puede considerar desfasada. Sin embargo, tiene todas las condiciones necesarias para que funcione impecablemente.

Un problema es que al ser antigua, instalar los *drivers* en los sistemas más modernos es un poco engorroso. Una vez instalado [3], se puede volver a cambiar deshabilitar de

[3]Siguiendo las instrucciones del siguiente vídeo: https://www.youtube.com/watch?v=01-CRhZnv90

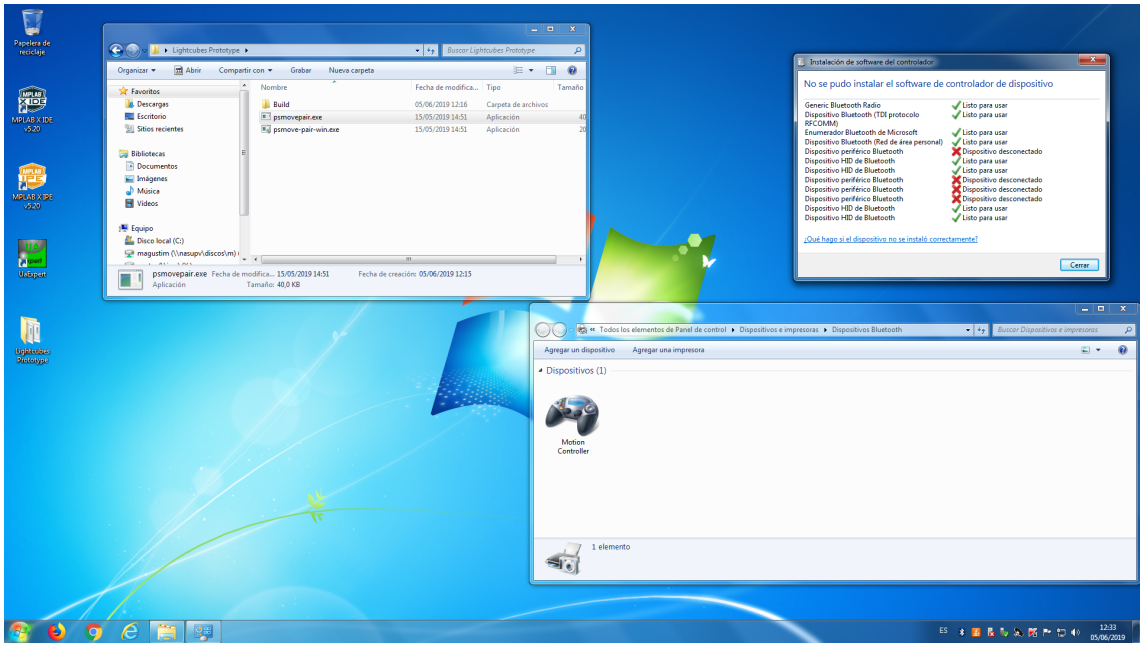

Figura 3.5: Captura de pantalla conexión del *PSMove* en *Windows 7*

nuevo la opción de *TESTSIGNIN* de la siguiente manera:

```
1    $ bcdedit.exe -set TESTSIGNING OFF
```

Como bien se explica en la documentación de *Windows* [40] «esta opción de configuración sirve para determinar si versiones de *Windows Vista* y posterior cargarán algún tipo de código de *kernel* en modo de prueba. Esta opción no está por defecto, por lo tanto cualquier *driver* que requiera utilizar el *kernel* en modo de prueba no se podrá instalar.» En consecuencia, una vez se haya habilitado para instalar el *driver* necesario ya se puede deshabilitar.

Para probar que todo ha funcionado correctamente he utilizado el programa *CLEye-Test* que venía adjunto en los archivos que incluía el vídeo mencionado en el párrafo anterior. Este programa permite visualizar lo que está capturando la cámara y se ha podido comprobar que funcionaba adecuadamente. Ahora, hay que modificar el *script* de compilación de la *PSMoveApi* para que no se utilice la *PSEye* por defecto, como hicimos en *MacOS* y volver a compilar la librería. Para ello se ha modificado la siguiente línea de código:

```
1    $ cmake .. -G "%MSVC_CMAKE_GENERATOR%" -DPSMOVE_USE_PS3EYE_DRIVER=0
```

Después, se han comprobado, al igual que en *MacOS*, el funcionamiento de las aplicaciones *test_tracker* y *test_opengl*. Al principio, la calibración ha durado demasiado tiempo y se ha probado a oscurecer la habitación. Al hacer esto, se ha detectado el mando, pero igualmente ha tardado mucho más que con la *iSight* en *MacOS* y no se seguía de manera fluida. Se ha deducido que el problema era la exposición de la cámara ya que si había un mínimo destello de luz la cámara dejaba de detectar el mando. Esto se debe a que el *PSMove* se rastrea utilizando la luz de la bola y si hay otra fuente de luz se puede confundir con el propio mando.

Para modificar la exposición se puede utilizar el programa *CLEye-Test*. Este permite modificar varias opciones para personalizar la manera de capturar la imagen. Al princi-

pio, se ha modificado el valor de la exposición e iluminación para que fuese cero como se puede ver en la figura 3.6.

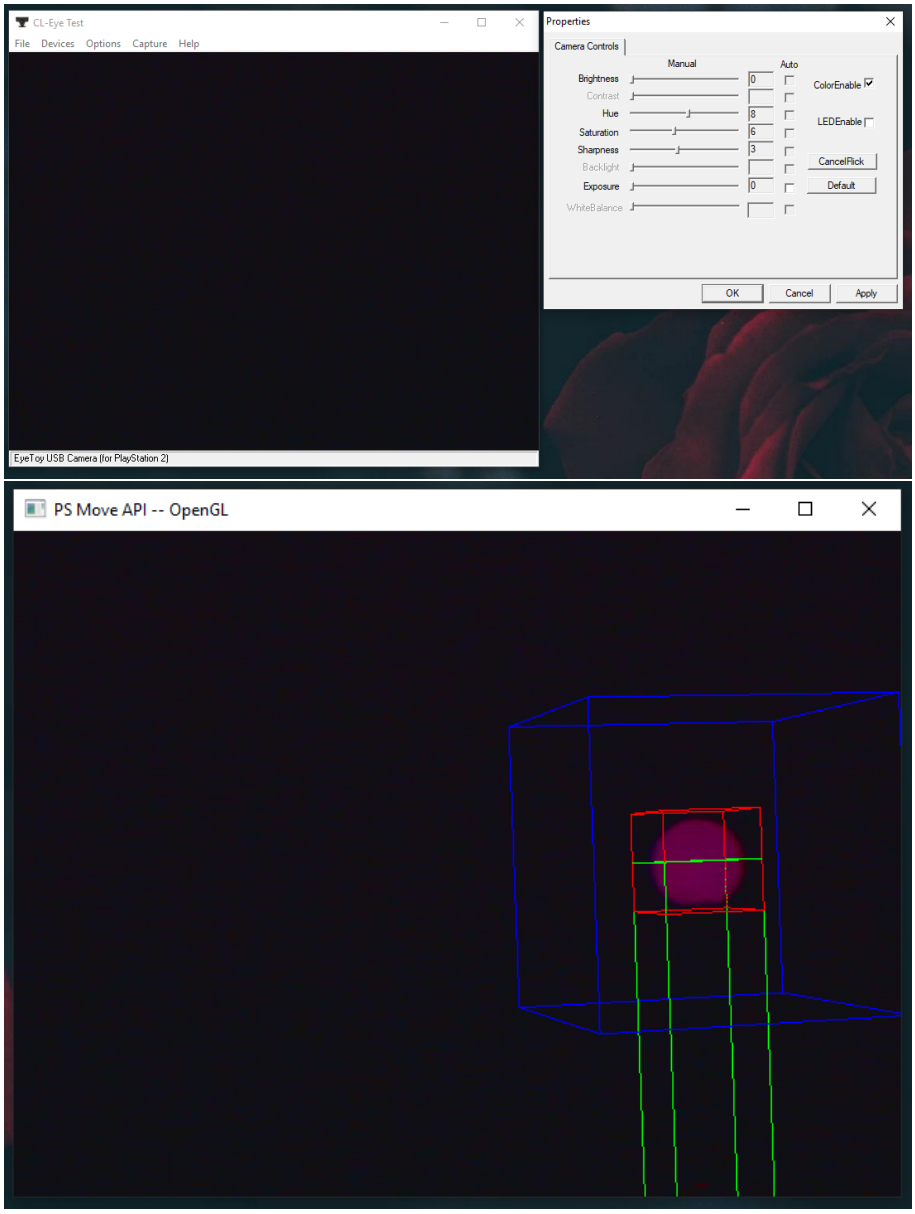

Figura 3.6: Arriba, captura de pantalla con nuevos ajustes *CLEye-Test*. Abajo, prueba de los nuevos ajustes en *test_opengl*.

Esto ha mejorado notablemente la detección, sin embargo al probar con dos mandos no ha funcionado porque ambos se detectaban en la esfera de un único mando y la otra se ignoraba completamente. Para solventar esto, se han modificado los valores hasta dar con los adecuados. Como se puede ver en la figura 3.7.

Con esto se da por terminada la configuración e instalación en *Windows 10*.

3.2 Conexión con Unity

En esta sección se explicará cómo se ha conectado la librería elegida, *PSMoveApi*, en el entorno de desarrollo de videojuegos *Unity*. Se explicará el proceso en *MacOS* y *Windows*

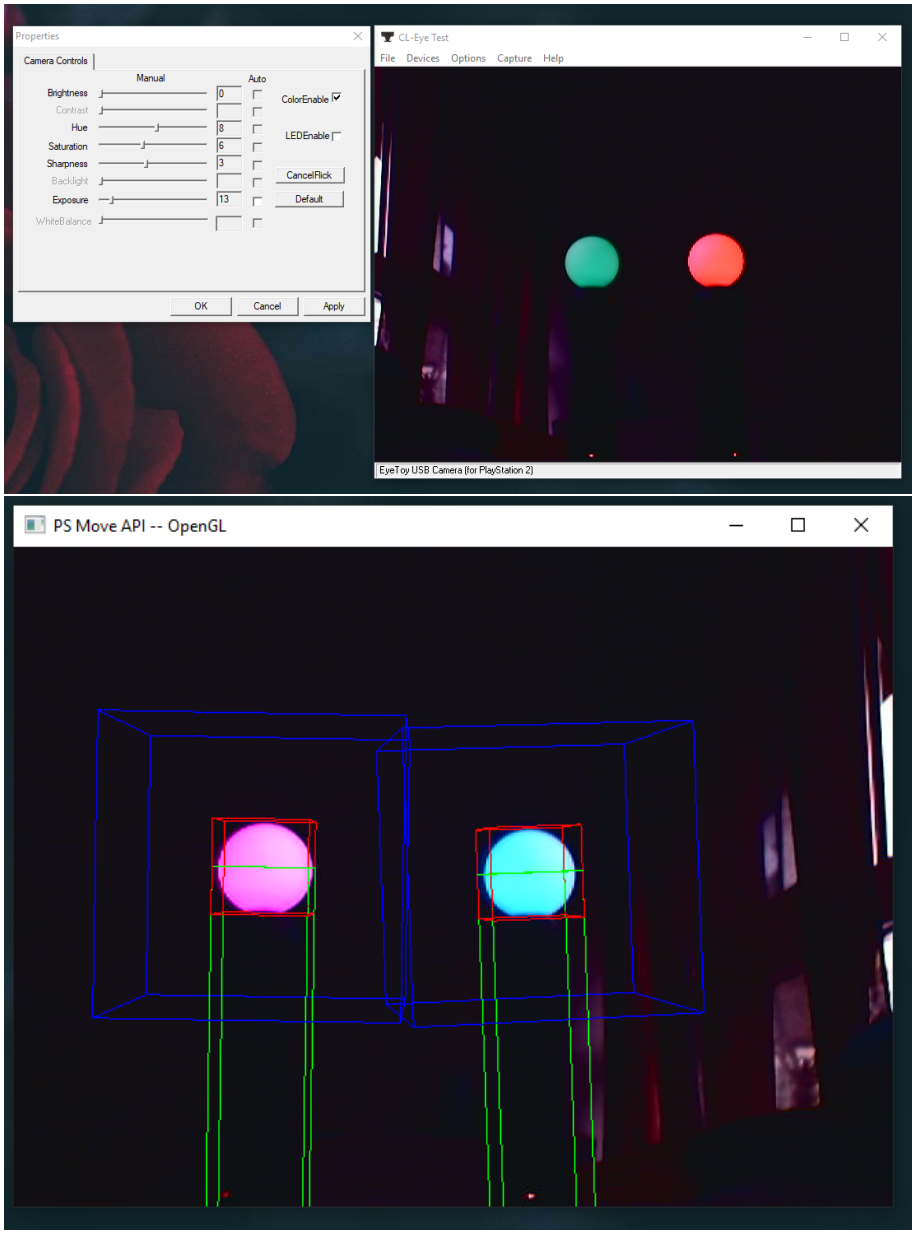

Figura 3.7: Arriba, captura de pantalla con nuevos ajustes *CLEye-Test* y demostración con los mandos iluminados. Abajo, prueba con *test_opengl*.

10, empezando con *MacOS* y los problemas que se han tenido con este sistema operativo y como se ha terminado utilizando *Windows 10* para desarrollar los prototipos.

3.2.1. MacOS

En la página web principal de la *PSMoveApi*, se muestran varias aplicaciones donde se ha utilizado en el entorno de *Unity*. Entre ellas se encuentran: *UniMove*, *UniMoveX* y *PSMove-Unity5*. En primer lugar, se ha investigado *UniMove*. *UniMove* es «un *plug-in* de código abierto para *Unity* que permite utilizar los mandos *PSMove* en un proyecto *Unity*» como bien se expone en su página web [42]. Además, ofrece un proyecto de ejemplo para probar su funcionamiento. Así pues, se ha descargado para ver si su funcionamiento era adecuado y en consecuencia, utilizarlo como base para los prototipos que se van a desarrollar.

Se puede descargar el proyecto del *GitHub* que se encuentra en la web, sin embargo en la descripción se indica que este *plug-in* solo es capaz de leer los valores internos del mando (acelerómetro, giroscopio, etc), no detecta la posición en la que está. Aunque no utilice la cámara, se va a probar de todas manera para ver si la *PSMoveApi* se conecta con *Unity* satisfactoriamente.

Una vez descargados los archivos, se pueden abrir con *Unity*. Para ello hay que crear un nuevo proyecto vacío e importar el *UnityPackage* que se encuentra dentro estos. Al ejecutar el juego se puede ver un modelo 3D del *PSMove* que responde a la orientación que tiene el mando de verdad y a la velocidad con la que se mueve, pero no se mueve del sitio ya que solo obtiene los valores internos. También, se ha probado a conectar dos mandos y ha funcionado a la perfección, han aparecido dos modelos 3D y cada se movía respectivamente al mando que representaba. Esto se puede ver en la figura 3.8. Además, se podía cambiar el color de la esfera utilizando equis, cuadrado, triángulo y círculo y activar la vibración el mando con el gatillo.

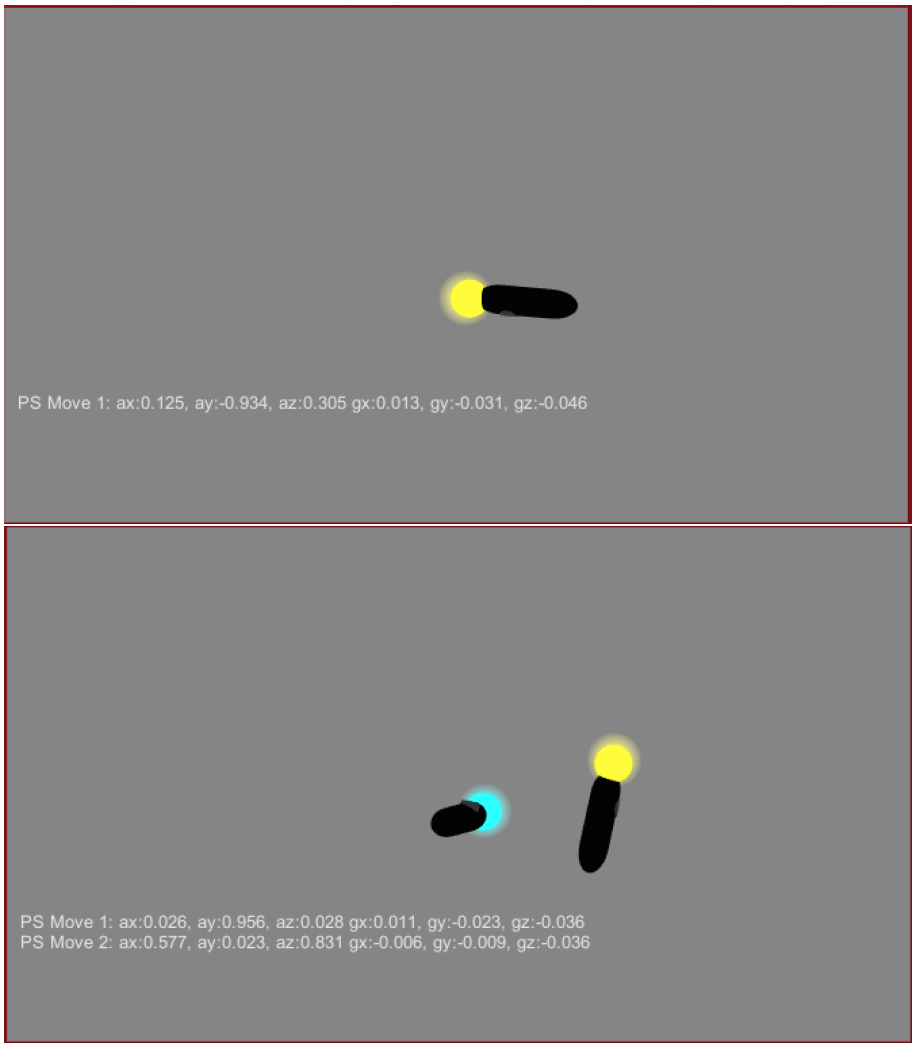

Figura 3.8: Arriba, captura de pantalla de *UniMove* con un mando. Abajo, utilizando dos mandos.

El funcionamiento de *UniMove* se compone de tres partes. Primero, se incluyen la biblioteca de enlace dinámico (*DLL*) *libpsmoveapi.dll* y la estructura de archivos para *MacOS*, *libpsmoveapi.bundle*. Estos que contienen los métodos de la *PSMoveApi* compilados para *Windows* y *MacOS* respectivamente. Segundo, el *script UniMoveController.cs* que se encarga de acceder al *DLL* y transformar los métodos al lenguaje C#, que es el que utiliza *Unity*. Además, se encarga de procesar el movimiento del mando y de hacer los cálculos

necesarios para que las transformaciones se vean de manera apropiada. Por último, el script *UniMoveTest.cs* que obtiene el número de mandos conectados y le asigna a cada uno el *UniMoveController* para que se procesen individualmente las acciones.

Así, se ha obtenido un proyecto funcional que utilizando la *PSMoveApi* como pilar permite trasladar esos movimientos a *Unity*, no obstante es muy limitado ya que no hace uso de la cámara. El objetivo es utilizar una cámara para que se puedan simular movimientos más precisos, aunque más adelante, en la sección 4.2, se mostrará como se ha desarrollado un prototipo usando únicamente los movimientos internos del mando. Es importante mencionar, que si los mandos se desconectan *Unity* se cierra automáticamente informando de un error.

Como bien se ha mencionado al principio del capítulo 3, la idea de utilizar *PSMove-Unity5* está descartada porque está solo operativa en *Windows* y utilizando la *PSEye*. Por lo tanto, la siguiente alternativa es *UniMoveX*, esta se define en su página web [43] como «una extensión de *UniMove* creada para manejar el seguimiento completo de la posición y la orientación». En esta web, se encuentra una modificación del *script UniMoveController.cs* que se puede descargar, por lo tanto se remplazará el archivo del mismo nombre que usa el proyecto *UniMove* con este y así probar su funcionalidad.

Antes de modificar el *script UniMoveTest.cs* para implementar estos nuevos métodos. Aparece el error: *DLLNotFoundException: libpsmoveapi_tracker* esto ocurre porque está intentando utilizar una *DLL* que no existe. El nuevo *script* necesita los métodos para acceder a la cámara y detectar la posición del mando y los esta intentando obtener de esta librería que no encuentra. El problema nace en las librerías que vienen con el proyecto de *UniMove* ya que no contienen estos métodos. Se ha buscado dónde encontrar esta librería en concreto y se ha encontrado en la carpeta de instalación de la *PSMoveApi*, concretamente dentro de la carpeta *build*. No obstante, la que se incluye en *MacOS* es una biblioteca de enlace dinámico específica de *MacOS*, una *dylib* y estas no las puede leer *Unity*, ya que solo es capaz de leer *DLLs* y *Bundles*. Seguidamente, se han investigado las posibles soluciones para utilizar estas *dylibs*. En una de ellas [44] se menciona que se puede modificar la extensión de *dylib* a *bundle* ya que en las versiones más actuales de *MacOS* estas dos son intercambiables. Haciendo esto, *Unity* sí detecta estas librerías y el error desaparece.

El siguiente paso, es modificar el script *UniMoveTest.cs* para que implemente los métodos necesarios del nuevo *UniMoveController.cs*. Una vez hecho esto, al ejecutar el proyecto este se quedaba bloqueado, pero la esfera del *PSMove* se iluminaba. Únicamente, se podía forzar la salida de *Unity* para salir de este error. Después, se ha caído en la cuenta de que el problema se debía a la calibración del mando, el mando se iluminaba para hacer el proceso de bloquear la exposición de la *iSight*, el mismo que se explicó el la sección 3.1.1. Hecho esto, el programa se ejecuta.

Se han modificado los valores para que el movimiento del mando virtual fuese igual al del mando real, sin embargo el nuevo *UniMoveController.cs* utiliza el método *psmove_tracker_get_position()* que devuelve la posición de los ejes x e y del mando y el tamaño de la esfera según la distancia con la cámara. Para que se obtenga también el valor del eje z, la *PSMoveApi* tiene un método llamado *psmove_fusion_get_position()* en el que sí se devuelven los valores de los tres ejes. Al sustituir el método, la detección del mando ha dejado de funcionar correctamente, ahora el mando salta de un lado para otro y no coincide con el *PSMove* real.

Después de un tiempo sin encontrar la solución a este problema, se notó que la calibración que ocurría en *Unity* era diferente a la que ocurría en los ejecutables de ejemplo de la *PSMoveApi*. Se ha investigado el tema y el error desemboca en las librerías, como he mencionado anteriormente, se ha modificado la extensión del *dylib* para convertirlo en *bundle*, aunque esto solvente el problema directo de no detectar las librerías, el compor-

tamiento no es el adecuado. Por lo tanto, hay que generar un nuevo *bundle* que incluya esta librería.

Para generar un *bundle* en *MacOS* se utiliza el programa *XCode*. «*Xcode* es un entorno de desarrollo integrado (*IDE*) para *macOS* que contiene un conjunto de herramientas creadas por *Apple* destinadas al desarrollo de software para *macOS*, *iOS*, *watchOS* y *tvOS*» [45]. Para generar un *bundle* hay que crear un nuevo proyecto tipo *bundle*, añadir las librerías que se quieran incluir (los *dylibs*) y un *info.plist*, «un archivo de texto estructurado que contiene información de configuración esencial para un ejecutable tipo *bundle*» [46], que incluya todos los permisos necesarios y compilarlo para generar el *bundle*.

Cuando se ha generado el *bundle* lo añadimos al proyecto *Unity*, de primeras ya parece que no se haya generado correctamente porque *Unity* lo esta mostrando como un directorio de archivos y no un único archivo como se muestra el que venía con *UniMove*. Igualmente, se ha comprobado si funcionaba modificando el *script UniMoveController.cs* para que los métodos de la *PSMoveApi* se obtengan de este nuevo *bundle*. Como se esperaba, ha saltado el error de que no se detecta esta librería. También, se ha probado a generarlo utilizando directamente los distintos programas en C que se incluyen en la *PSMoveApi* y que forman los *dylibs* que trae esta. Pero de igual manera, *Unity* lo interpretaba incorrectamente.

Dados los problemas que se estaban dando a la hora de generar el *bundle* apropiadamente y visto que cuando se genera al *PSMoveApi* en *Windows* se generan los *DLLs* directamente. Se ha decido probar en este sistema operativo para ver si la cámara con los tres ejes se detectan ya que *Unity* es capaz de leer correctamente los *DLLs* sin necesidad de modificarlos.

Como conclusión, en *MacOS* la conexión con *Unity* es satisfactoria hasta cierto punto. Un proyecto que únicamente utilice las valores internos del *PSMove* es viable, pero en un proyecto que se utilice la cámara se dan más problemas. Como se ha mencionado, detectar el mando y sus movimientos en los ejes x e y, es funcional. Sin embargo, se parte de utilizar una librería que no funciona correctamente y por lo tanto para utilizar el eje z surgen más problemas todavía.

3.2.2. Windows 10

Al igual que en *MacOS*, se va a empezar probando el proyecto de *UniMove*. No han habido problemas a la hora de utilizarlo, se han seguido los mismos pasos que se hicieron en *MacOS* y ha funcionado de manera impecable. Ahora, hay que comprobar que la cámara se detecta como debe. Para ello se ha utilizado un proyecto llamado *UnityMove* [4], que utilizaba también los métodos para acceder a la cámara, como *UniMoveX*, pero que al contrario que este, ofrecía un proyecto de *Unity* ya montado. Para utilizarlo, se ha creado un proyecto vacío en *Unity* y se ha importado.

Primeramente, ha aparecido el error de que los *DLLs* no se encuentran. Así pues, se han sustituido estos por lo que se instalaban junto a la *PSMoveApi*. A pesar de ello, el error seguía apareciendo. Se ha abierto el directorio donde se encuentran los ejecutables y librerías en *Visual Studio* para volver a compilarlo manualmente. Se ha probado primero ha modificar uno de los ejemplos en C añadiendo un mensaje por consola para ver si la compilación manual era funcional. Lo ha sido, pero se tiene que compilar individualmente ya que si se utiliza el botón general, muestra un error ya que hay muchos tipos de archivos en esta carpeta. Seguidamente, he repetido el proceso en las librerías y no se ha dado ningún error. Se han importado estas nuevas librerías en *Unity*, pero el error seguía apareciendo.

[4] https://www.youtube.com/watch?v=FrJGh6AQmf4

Después, se ha descubierto [5] que se tienen que compilar en modo *release* y no *debug*, que es el que se estaba utilizando. Para hacer esto, se ha accedido al administrador de configuración y se ha cambiado el modo de compilación de las librerías como se muestra en la figura 3.9. Importando estas nuevas librerías en *Unity* el error se ha solucionado. Ahora falta probar si los métodos de acceso a la cámara funcionan adecuadamente.

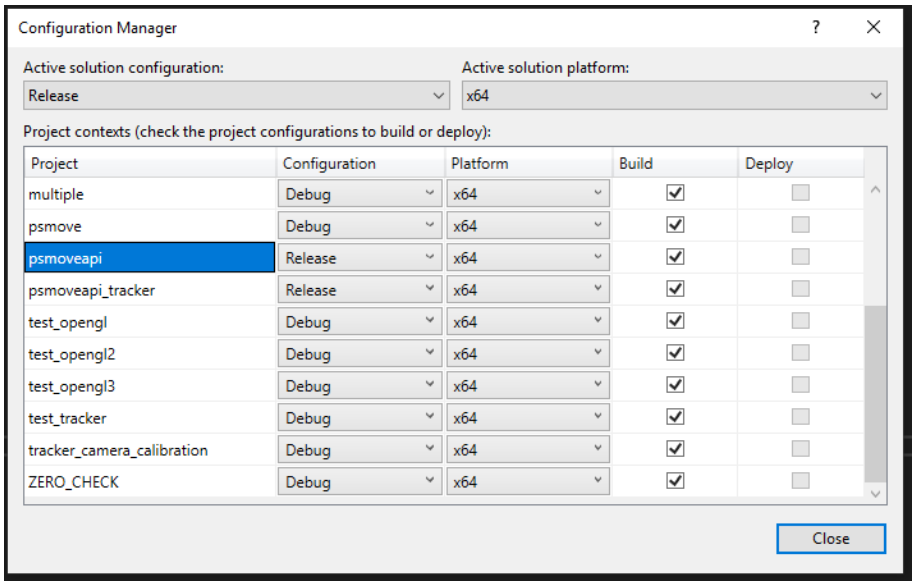

Figura 3.9: Modificaciones en el administrado de configuración para que las librerías se compilen a modo *release*.

Como en este proyecto ya viene un ejemplo preparado, se ha procedido a ejecutarlo directamente. El problema es que al no haber una cámara conectada, *Unity* ha dejado de funcionar y se ha cerrado automáticamente. Se han conectado la cámara y los mandos y tras volver a ejecutar no ha dado problema. El mando de se mueve por la pantalla, pero no coincide con los movimientos que se están haciendo en la vida real. Por ejemplo, al subir el *PSMove* real, el mando virtual baja. Para arreglar esto hay que modificar las transformaciones que se están haciendo de los valores que se obtienen del *PSMove*. Se van a aprovechar estos cambios para desarrollar un pequeño prototipo y así tener un primer contacto con los métodos de la librería.

El experimento que se ha preparado, consiste en una esfera que se mueve utilizando el *PSMove* y tres cubos repartidos en el escenario. Cuando la esfera entra en contacto con uno de los cubos, este cambia de color y el mando vibra. Tras probar con varias valores, las modificaciones finales que se han hecho para que el movimiento se replicara correctamente son las siguientes:

```
1     float px = 0, py = 0, pz = 0;
2     psmove_fusion_get_position(fusion, handle, ref px, ref py, ref pz);
3
4     position.x = px * 5;
5     position.y = -py * 5;
6     position.z = -pz * 5;
```

Como se puede apreciar, la 'z' e 'y' se han invertido y todos se han multiplicado por 5 para que el movimiento coincidiera mejor ya que con el valor por defecto el mando se desplaza muy poco por la pantalla. En la figura 3.10, se puede apreciar el funcionamiento de este primer prototipo.

[5] https://answers.unity.com/questions/993154/failed-to-load-dll-error.html

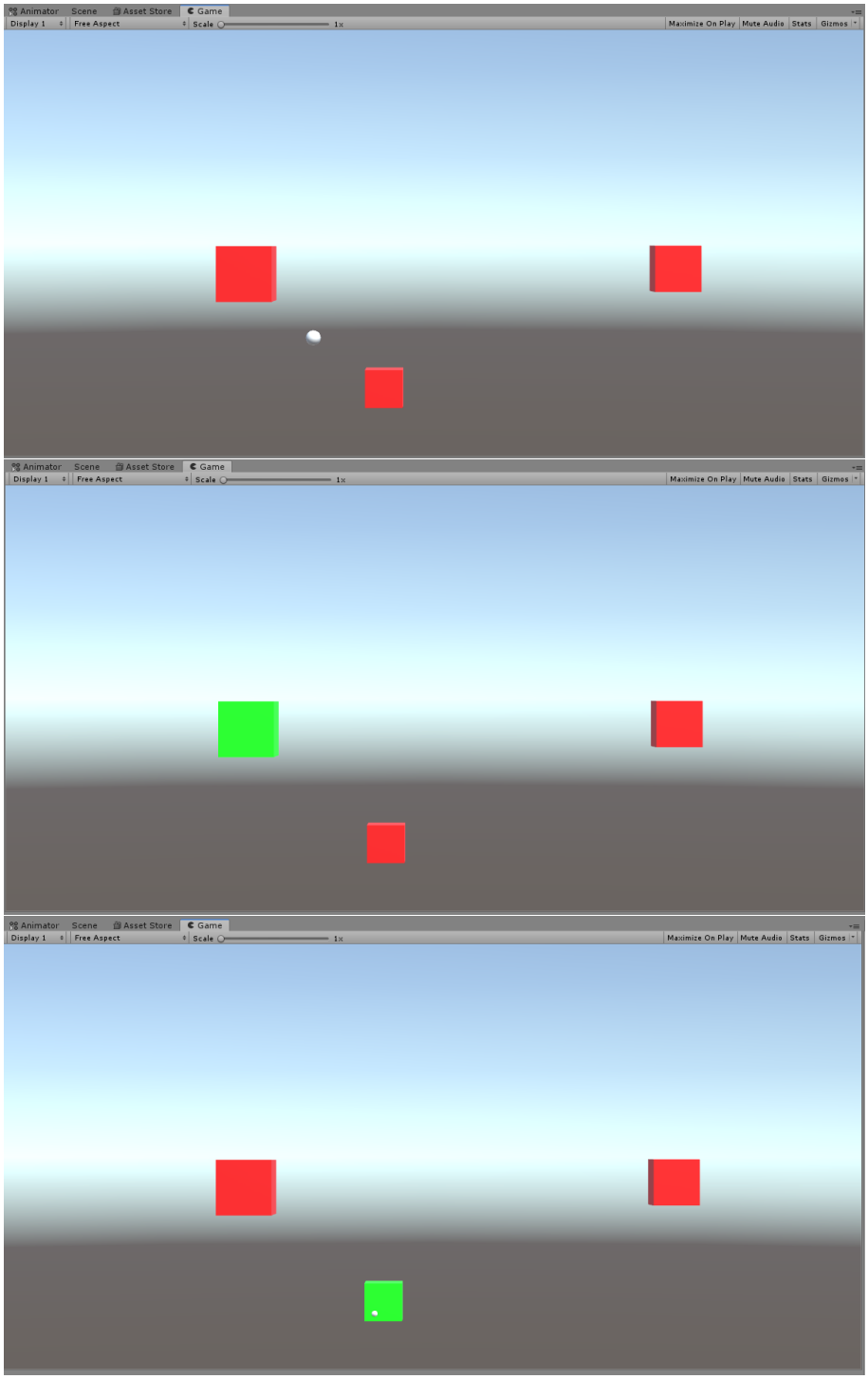

Figura 3.10: Capturas de pantalla del prototipo de la esfera y los cubos. La foto superior, esfera sin tocar ningún cubo. En medio, la esfera esta dentro y por tanto tocando el cubo de la izquierda. Abajo, la esfera tocando el cubo inferior.

Ahora que ya funciona la *PSMoveApi* junto a *Unity*, se va a proceder a desarrollar los prototipos más serios. Aunque en un principio, la intención era utilizar *MacOS* para trabajar en este proyecto, visto que en *Windows* la conexión con *Unity* ha dado menos problemas, se va a utilizar este sistema operativo a partir de ahora.

CAPÍTULO 4
Desarrollo de prototipos

En este capítulo se explorarán los diferentes prototipos que se han desarrollado para experimentar con las múltiples posibilidades que ofrece el *PSMove*. Se empezará con *Lightcubes* que utiliza la cámara y el mando para simular un sable láser. Después, se hablará de *Space Duel* que utiliza los valores internos del mando para imitar el movimiento de un *joystick* y así pilotar una nave espacial. En *Lightcubes Trinus VR* se utilizará el servicio de *Trinus VR* para trasladar la pantalla del ordenador al móvil y lograr una experiencia más próxima a la que ofrecen los dispositivos de realidad virtual actuales, utilizando *Lightcubes* como base. Por último, en *YogaMove* se verá una aproximación más enfocada hacia los *Serious Games*, que son como he mencionado en el capítulo 1, juegos en los que el objetivo principal no es la diversión, sino ayudar en aspectos más formales como la rehabilitación dentro del ámbito de la medicina.

Es importante mencionar las especificaciones del ordenador en el que se van a desarrollar ya que este afectará directamente al rendimiento de los diferentes prototipos. Estas son:

- *Windows 10* 64−*bits*
- Procesador *Intel Core i*7 − 6700*K*
- 32GB de *RAM*.
- Tarjeta gráfica *NVIDIA GeForce GTX* 1080

4.1 Primer Prototipo - Lightcubes

La idea de este prototipo es crear algo similar al juego *Beat Saber* [47], figura 4.1. Este utiliza los mandos para simular 2 sables virtuales que se utilizan para destruir unos cubos al ritmo de la música. Para empezar a prototipar se hará una única fila de cubos y un sable láser que sea capaz de romperlos.

El primer paso, será crear la espada láser. Para ello se utilizarán las figuras básicas que ofrece *Unity* para generar el mango y el filo. Además, se ha creado una animación para activar y desactivar el filo para imitar el funcionamiento de las espadas láser de *Star Wars*. Para darle un efecto más realista, se ha añadido un efecto de de postprocesado para que el filo tengo luz propia, usando la opción *glow* que ofrece *Unity*. Combinando todo esto ha quedado como se muestra en la figura 4.2.

[1]Imagen extraída de https://store.steampowered.com/app/620980/Beat_Saber/

Figura 4.1: Captura de pantalla del juego *Beat Saber*. [1]

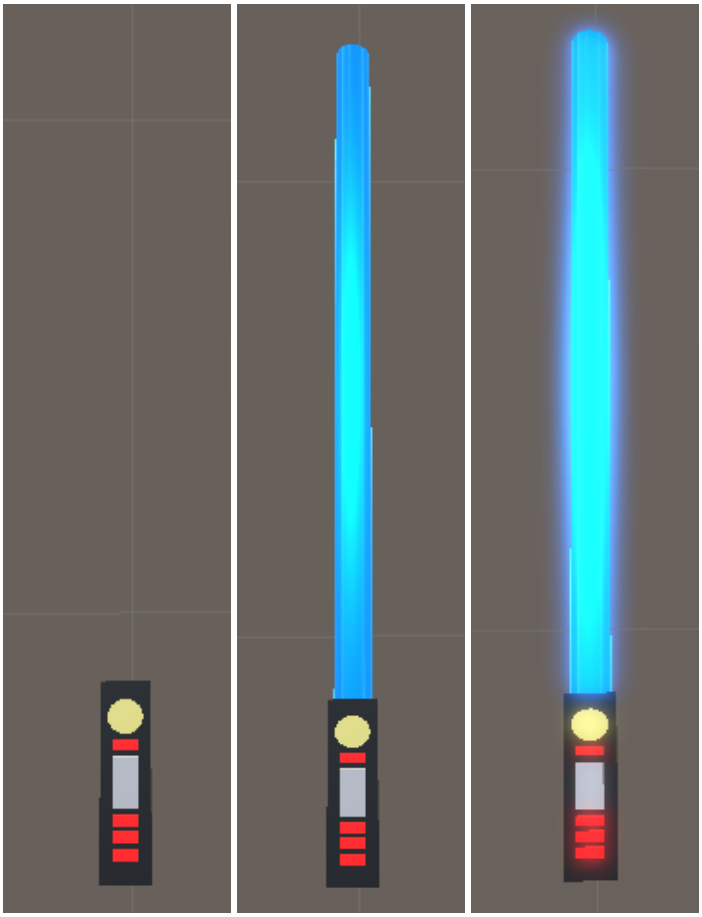

Figura 4.2: Capturas de pantalla de primer diseño del sable láser. A la izquierda, utilizado el efecto de postprocesado.

Partiendo del ejemplo de la esfera y los cubos (Figura 3.10), se ha implementado el movimiento del mando. Sin embargo, se ha tenido que modificar la orientación ya que al ser una esfera no se notaba el error. Basándose en el ejemplo que traía *UniMove*, cuando se reiniciaba la orientación, el mando se posicionaba como se muestra en la figura 4.3. Modificando entonces el modelo del sable láser para que fuese igual a este, se ha conseguido que la orientación se imite correctamente.

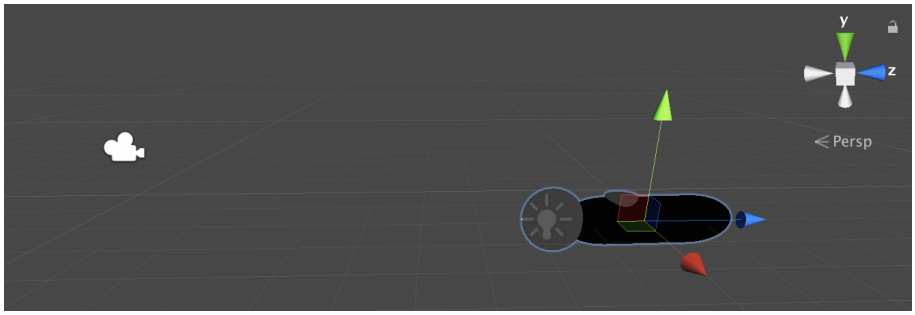

Figura 4.3: Reinicio de orientación en el proyecto *UniMove*.

Por último, hay que hacer que la espada genere un rastro al moverse, al igual que ocurre en las películas de *Star Wars*. En primer lugar, *Unity* ofrece un componente llamado *trail renderer* que se encarga de generar rastros en los objetos en los que se añade. Se hizo una prueba utilizando el mismo material que tiene el filo de la espada en este componente, pero dejaba mucho que desear. Las opciones eran muy limitadas y el rastro no seguía con fluidez los movimientos del sable (Figura 4.4).

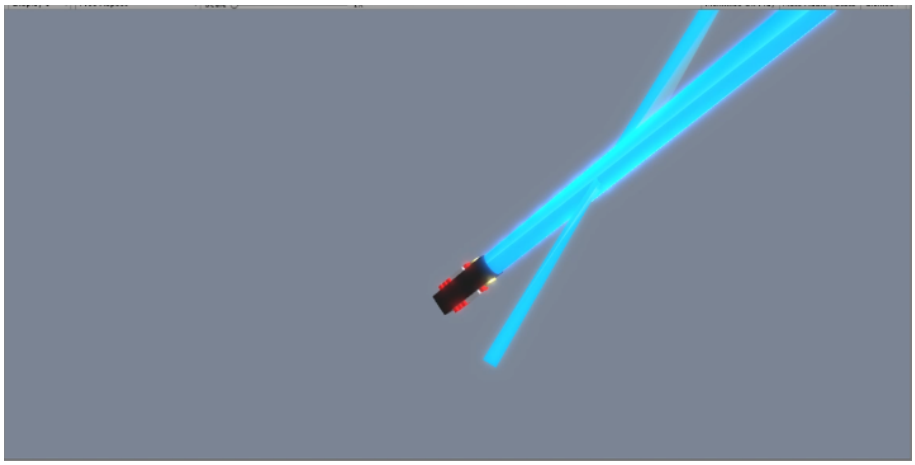

Figura 4.4: Captura de pantalla utilizando el componente *trail renderer*.

La siguiente alternativa que se probó, fue utilizar otro componente de *Unity*. El *particle system* es un sistema para generar y configurar emisores y emisiones de partículas. La idea parte de un un tutorial [2] en el que se explica cómo usar este componente para generar rastros en una espada. Aunque los efectos que enseña no son exactamente los que se están buscando, se explica muy bien cómo utilizar todas las funcionalidades que ofrece el *particle system*.

Se modificó el componente para que las partículas se generaran siguiendo la forma del filo y además, que se generaran muchas para dar una sensación de solidez. La única opción de emisión que se parecía al rastro que se buscaba era la de aleatorio, pero el resultado (Figura 4.5) no se aproxima al efecto que se quiere conseguir. Asimismo, al hacer movimientos rápidos, el rastro quedaba todavía más disperso.

Por último, se encontró un tutorial [3] que se basaba en un tutorial oficial de *Unity* en el que se explicaba como generar los rastros de las carreras de motos de la película *Tron Legacy* (Figura 4.6) y lo adaptaba a un sable láser. Esta es la opción finalmente escogida por ofrecer la imitación más realista hacia los sables que se utilizan en el universo cinematográfico de *Star Wars*.

[2] https://www.youtube.com/watch?v=c8hijUge7IY
[3] https://www.youtube.com/watch?v=goB0obtRAeg

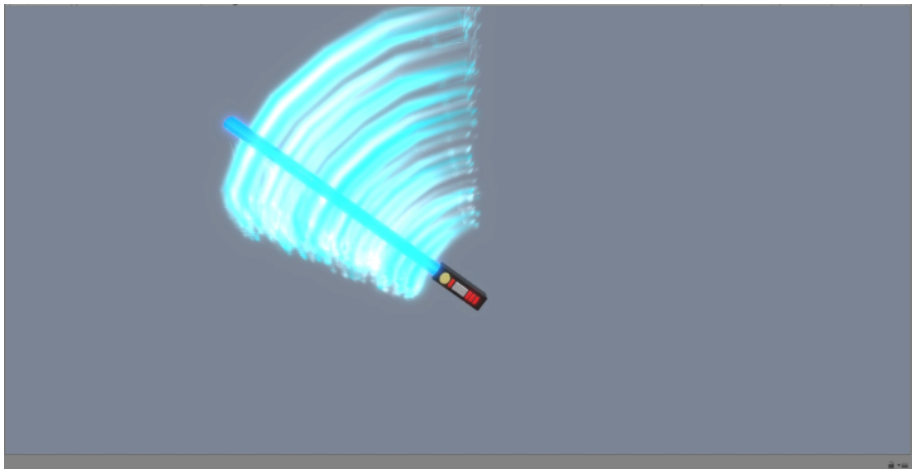

Figura 4.5: Captura de pantalla utilizando el componente *particle system*.

Figura 4.6: Carrera de motos de luces en la película: *Tron Legacy*. [4]

Una vez incluidos los archivos necesarios, se han utilizado para modificar el filo del sable láser y ahora sí que ha quedado como se buscaba. Si se compara con el efecto que utilizan en las películas este ha quedado bastante logrado como se puede ver en la figura 4.7.

Durante este proceso, se han probado varios colores de filo (azul, rojo, morado y verde) y se ha pensado en una nueva mecánica para hacer el juego más interesante. Que los cubos se generen aleatoriamente con los cuatro colores y que solo se pueda romper un cubo si coincide con el color de la espada. Se ha configurado el mando para que cambie el color del filo dependiendo del botón que pulse: equis lo cambia a azul, cuadrado a morado, triángulo a verde y círculo a rojo. Se ha decidido utilizar esta distribución ya que el color del botón también coincide con el color del filo y se facilita el aprendizaje del jugador.

Antes de crear el generador, se va a voltear la vista del juego. Ahora mismo, si se acerca el *PSMove* a la cámara, dentro del juego el sable se acerca a la pantalla. Se quiere que este movimiento esté invertido para así imitar al completo las acciones del jugador, es decir cuando el *PSMove* se acerque a la cámara, el sable simule el movimiento y se aleje

[4]Imagen extraída de https://wall.alphacoders.com/big.php?i=643235
[5]Imagen extraída de https://www.slashfilm.com/star-wars-bits-lightsaber-fight/

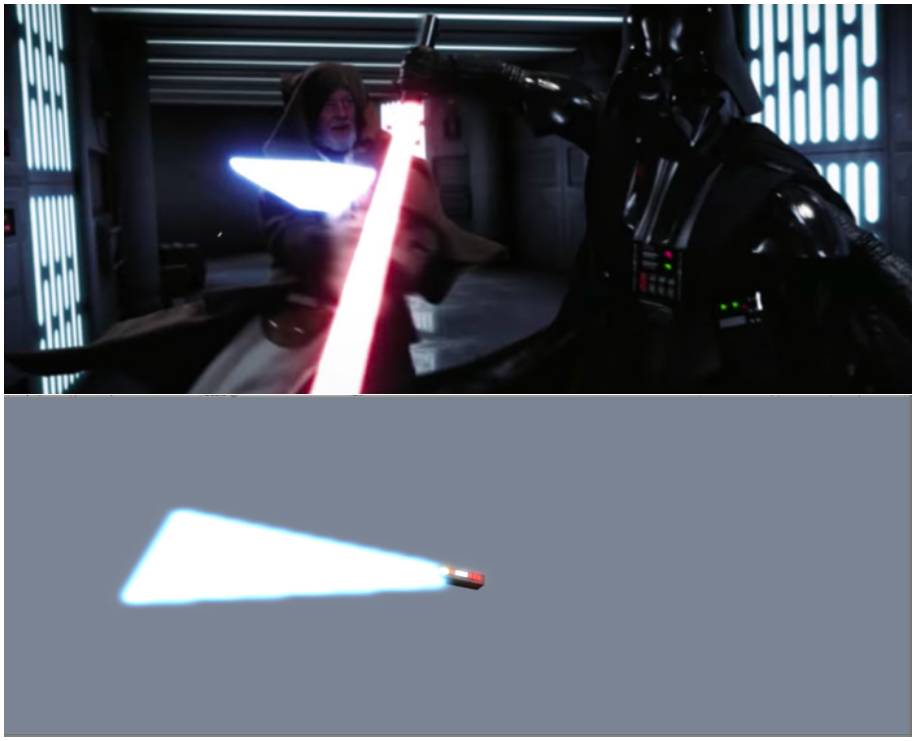

Figura 4.7: Arriba, ejemplo del rastro que dejan los sables láser en las películas de *Star Wars* [5]. Abajo, imitación de este en el proyecto *Unity*.

de la pantalla. Para hacer esto, lo primero es modificar la posición de la cámara del juego para invertir la vista. Sin embargo, los valores de posición y rotación ahora han dejado de coincidir, por lo tanto hay que volver a cambiarlos dentro del *script UniMoveController.cs*.Respecto al primer cambio, ahora se ha invertido además el eje 'x' de la posición y se han invertido los ejes de la orientación 'z' e 'y', como se puede ver a continuación:

```
1        psmove_fusion_get_position(fusion, handle, ref px, ref py, ref pz);
2
3        position.x = -px * 5;
4        position.y = -py * 5;
5        position.z = -pz * 5;
6
7        float rw = 0, rx = 0, ry = 0, rz = 0;
8        psmove_get_orientation(handle, ref rw, ref rx, ref ry, ref rz);
9
10       orientation.w = rw;
11       orientation.x = rx;
12       orientation.y = -ry;
13       orientation.z = -rz;
```

La siguiente modificación que se quería hacer era que al cambiar de color de filo, también se cambiase la esfera del mando como ocurría en el proyecto de *UniMove*. Al utilizar el método *SetLed()*, al que se le puede pasar el color al que se quiere cambiar la esfera, solo se cambiaba brevemente durante un único fotograma y luego volvía al color inicial. Esto se debe a que al calibrar el mando, la *PSMoveApi* le asigna un color específico a cada mando para que se pueda detectar correctamente. Si se quisiera utilizar otro, existen dos opciones. Primero, se tendría que volver a calibrar cada vez que se cambia de color. Esto causaría mucha ralentización porque la calibración no es instantánea. Segundo, modificar el código fuente de la librería para que en vez de detectar colores, detectara formas, en este caso objetos circulares. Pero esto viene con muchos inconvenientes, ya que no podría

haber ningún otro objeto cilíndrico en la imagen que captura la cámara y esto incluye otro *PSMove*. Dados estos problemas, se ha decido descarta la idea de modificar también el color de la esfera del *PSMove*.

Una vez terminada la funcionalidad del sable, hay que implementar las mecánicas del juego. Como he mencionado anteriormente, un cubo solo se podrá romper si los colores de este y el sable coinciden. Si no se rompe, el jugador perderá una vida, si se rompe, el jugador aumentará su puntación. Se empezará creando un generador de cubos aleatorios que se acerquen hacia el jugador en línea recta. Para detectar que colores tiene cada objeto en el momento de colisión, se puede acceder a la propiedad *material* se cada uno y hacer la comprobación. Con esto, los cubos ya se rompen cuando se cumplen las condiciones, pero no hay vida ni puntuación. Por lo tanto, lo siguiente es añadirle una interfaz de usuario al juego que muestre la puntuación y vida actual.

Para que se transmita la información necesaria entre los distintos objetos del juego, se va a utilizar el *script Messenger.cs*. Este se puede obtener de la *wiki* oficial de *Unity* [6] y sirve para generar un sistema de eventos para C#. Estos eventos se van a crear en una clase aparte llamada *GameEvent* y todos los scripts podrán acceder a ella. Se empezará utilizando *'MINUS_LIFE'* y *'ADD_SCORE'* para restar una vida y aumentar la puntuación respectivamente. Con esto ya tenemos una primera versión del juego funcional (Figura 4.8). Además, se han añadido algunas mejoras. Se ha incorporado música al juego, tanto música de fondo como de efectos especiales, entre ellos activar y desactivar la espada. También, se han acompañado las acciones del juego con la vibración del mando para darle más retroalimentación e inmersión al usuario.

Figura 4.8: Captura de pantalla de *Lightcubes* con el generador de cubos y la interfaz de usuario ya impletados.

Después de jugar a este prototipo, se ha encontrado un fallo. La activación y desactivación del sable no es fluida y a veces no funciona como debería. Ahora mismo, se estaba utilizando *Invoke()* que permite ejecutar el método que le pasas como primer argumento tras los segundos que le pases como segundo argumento, para activar los componentes del sable y la vibración. El problema de esto es que no siempre se ejecuta al mismo tiempo y no es la alternativa más eficiente. Por ello, se ha utilizado los *AnimationEvents* que ofrece *Unity*. Estos te permiten [48] «llamar a los métodos del *script* asignado al objeto en el punto de la línea del tiempo de la animación espeficiado». De esta manera, se puede aumentar la presición ya que se esta llamando al método necesario en el momento justo.

Aunque esto no ha arreglado el problema de la fluidez, ha ayudado a encontrar el error. El problema se debía a que se había configurado el filo para que su componente

[6] http://wiki.unity3d.com/index.php/CSharpMessenger_Extended

se inhabilitara completamente cuando se desactivaba. Esto desembocaba en que antes de que la animación terminará, el componente se inhabilitaba y por lo tanto la animación se cortaba a mitad. Este comportamiento se había añadido porque con el filo desactivado se podían romper los cubos igualmente. Se ha cambiado este funcionamiento para que una vez la animación llegase a su fin en lugar de deshabilitar todo el componente se deshabilitaba únicamente el *collider* que se encarga de detectar las colisiones con los cubos.

Los cubos ahora mismo no ofrecen mucho feedback ya que cuando se destruyen simplemente desaparecen, no hay un componente visual que muestre realmente que se ha roto. Para arreglar esto se van a generar una explosión de cubos una vez este sea destruido. La idea es que se sustituya el cubo por clones más pequeños y a estos aplicarles una fuerza en dirección contraria para crear ese efecto de explosión. Asimismo, se ha implementado una vibración de pantalla que también ocurre al destruir uno de los cubos, para aumentar la respuesta visual todavía más. Hecho esto, el juego queda como se muestra en la figura 4.9.

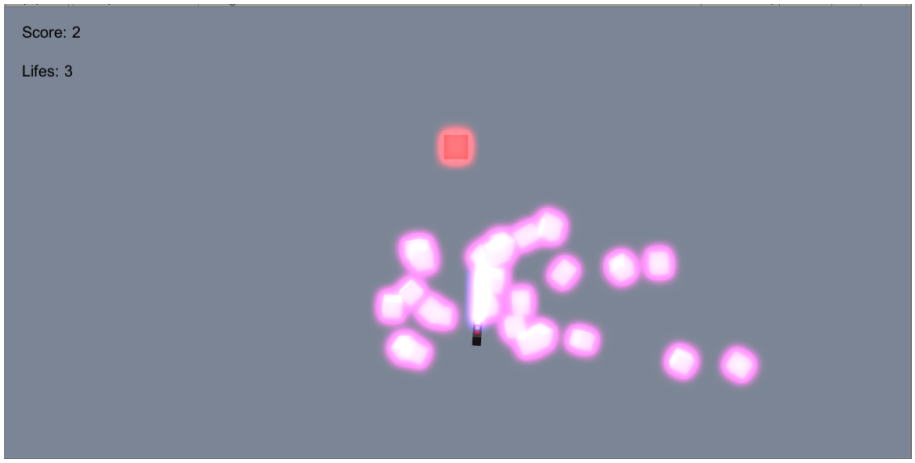

Figura 4.9: Captura de pantalla de la explosión al destruir un cubo.

Cuando se ejecuta el juego, el mando siempre tiene la orientación desfasada y para que esta vuelva a funcionar como debe existe un método para reiniciar la orientación. Hasta ahora, este método se había asignado al botón *select* pero esto no se daba a conocer al jugador. Así pues, se ha creado un mensaje de inicio que explica como reiniciar la orientación y hasta que el usuario no quiera no se inicializa el juego (Figura 4.10). De esta manera, se asegura que el juego siempre empiece con un movimiento del mando correcto y se explica como se reinicia durante la partida por si la orientación deja de coincidir.

Hecho todo esto, ya se tiene un juego funcional, pero ahora se va a probar a utilizar 2 mandos para hacer el juego más dinámico. En primer lugar, se va a crear otra escena para no trabajar sobre el modo actual. En *Unity* una escena se define como [49] «los contenedores de entornos y menús del juego. Cada escena se puede considerar como un nivel único y en estas se colocarán los obstáculos, decoraciones, etc.» En esta nueva escena, se han duplicado los objetos que se utilizaban en la de un solo jugador y para poder utilizar dos mandos, se ha creado un *script* que le añade a cada uno el *script UniMoveController.cs*. De este modo, los movimientos de cada mando se calculan independientemente. Así, los mandos funcionan y se mueven por la pantalla, pero no se mueven de manera fluida.

Se ha investigado que es lo que causa esta ralentización del juego utilizando el *profiler* de *Unity*. Este muestra una gráfica con los aspectos del juego que más tiempo consumen. En este caso, se ha podido comprobar que la parte que más consume es el *script UniMoveController.cs*. Es lógico ya que se encarga de detectar y calcular la posición de los mandos

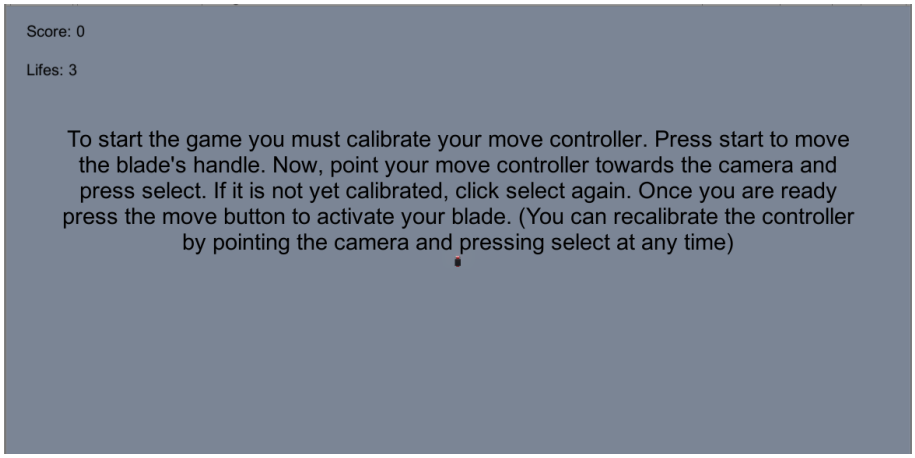

Figura 4.10: Captura de pantalla del mensaje de inicio para reiniciar la orientación. Del inglés, «para empezar el juego debes calibrar el *PSMove*. Presiona *start* para mover el mando. Ahora, apunta hacia la cámara y presiona el botón *select*. Si aún no se ha calibrado correctamente, vuelve a pulsarlo. Cuando estés preparado presiona el botón *move* para activar tu sable. (Puedes recalibrar el mando apuntando a la cámara y presionando *select* en cualquier momento).»

en cada fotograma. Se ha comprobado si este problema ocurre también en los ejecutables de ejemplo de la *PSMoveApi*, aunque también ocurre si se mueven los mandos bruscamente, no es tan notable como en el proyecto de *Unity*. Se han probado varias mejoras para ver si la ralentización se podía reducir. Se han eliminado los cálculos que no se utilizaban como el acelerómetro o el magnetómetro, la diferencia ha sido inapreciable. Por esto, se ha probado a reducir el número de veces que ocurre la captura de de posición del mando, ya que la orientación no se ve afectado por ello porque se envía la información mediante bluetooth. Esta captura ocurre cuando se ejecutan los siguientes métodos:

```
1         psmove_tracker_update_image(tracker);
2         psmove_tracker_update(tracker, handle);
```

En primer lugar, se ha reducido para que solo se ejecutasen una de cada cinco veces. Esto ha mejorado el problema del rendimiento, pero en su lugar ha perjudicado la fluidez con la que se mueven los sables ya que ahora se teletransportan por la pantalla en vez de moverse progresivamente. Seguidamente, se ha probado a aumentar el número de veces a una de cada dos. Con este cambio, se ha reducido el rendimiento en comparación con la prueba anterior y el número de teletransportes queda más disimulado. Comparando este último cambio con el comportamiento original, se ha aumentado el rendimiento lo suficiente para que valga la pena sacrificar ese punto de fluidez. En el modo para dos jugadores, se dejará de esta manera, en el modo un jugador no es necesario ya que ha ido fluido desde el principio. Es probable que utilizando una máquina más potente se alcanzará un mejor rendimiento y por lo tanto no harán falta añadir esta modificaciones. En la figura 4.11, se pueden apreciar como los distintos cambios han afectado al rendimiento del programa.

Para acabar de completar este prototipo se han añadido elementos necesarios. Primeramente, se ha modificado la una interfaz de usuario para que aparezcan la puntuación y vidas de ambos jugadores y un mensaje final para mostrar quien ha ganado o si ha habido empate. También, se comprueba al principio el número de mandos que hay conectados y si no detecta dos mandos muestra un mensaje de error y no deja entrar al juego. En segundo lugar, se ha añadido la limitación de que un jugador solo puede destruir los cubos que le corresponden ya que en un principio, cualquier jugador podía destruir cualquier cubo que aparecía en pantalla. Por último, se ha creado una nueva escena para que el

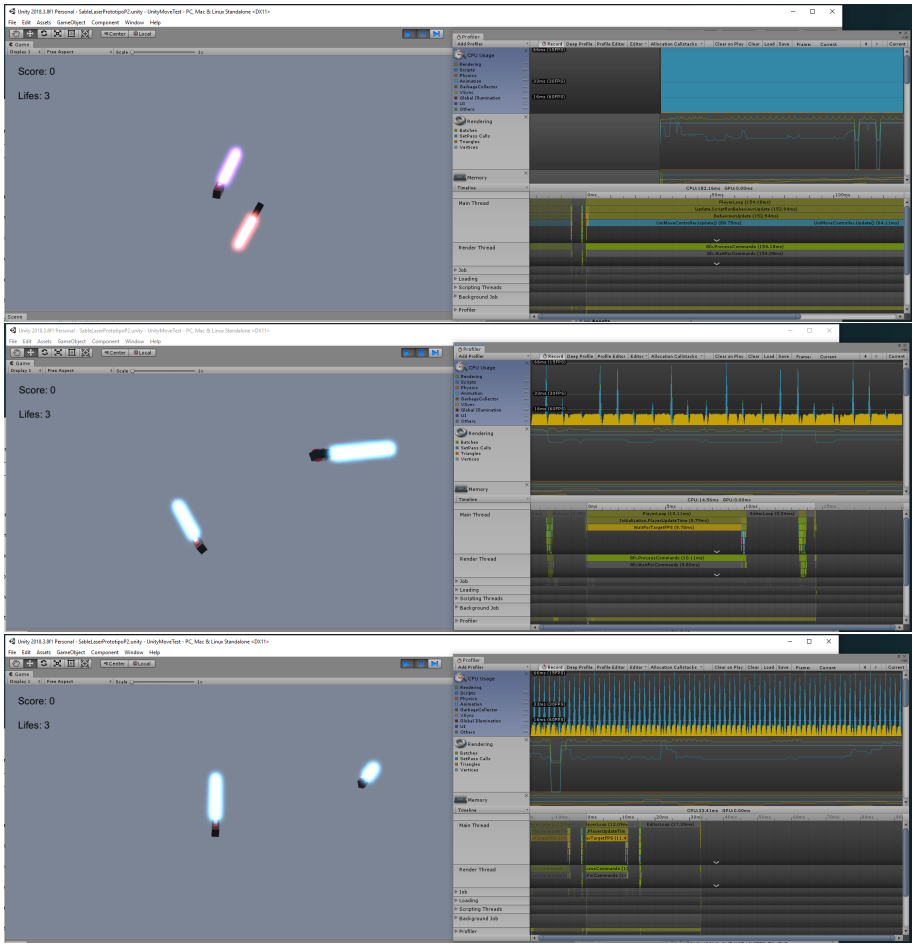

Figura 4.11: Capturas de pantalla del *profiler* con las distintas pruebas para aumentar el rendimiento. La parte azul del gráfico muestra los recursos que consume el *script UniMoveController.cs*. Arriba, el rendimiento original sin ninguna modificación. En medio, la primera modificación para que los métodos se ejecutaran una de cada cinco veces. Abajo, la segunda modificación para que se ejecuten una de cada dos veces.

usuario pueda elegir a que modo de juego puede entrar, uno o dos jugadores. A la hora de crear esta escena, se ha bloqueado en el movimiento del eje 'z' para simular un movimiento en dos dimensiones simulando el comportamiento de un cursor. Estos cambios se muestran en la figura 4.12.

Para finalizar el desarrollo de *Lightcubes* se ha modificado el modo de un jugador para que los cubos se generarán en cuatro posiciones distintas y así añadir un poco de dinamismo. Además, se ha cambiado el fondo para que tuviese más vida, añadiendo cubos repartidos por la escena que rotan de manera aleatoria y se ha oscurecido el color que se utilizaba para que estos resalten (Figura 4.13). Por último, se ha compilado el proyecto para generar un ejecutable. Para probar si era necesario tener la *PSMoveApi* instalado para poder usarlo, se ha ejecutado en otro ordenador. Primero, se han instalado los mandos utilizando los programas *PSMovePair* y *PSMove-Pair-Win* en esta nueva máquina. Una vez hecho esto, se ha abierto el ejecutable generado por *Unity* y ha funcionado sin ningún problema. Así pues, se puede concluir que únicamente hace falta la *PSMoveApi* para desarrollar los juegos, no para jugarlos.

En este primer prototipo se han podido comprobar las posibilidades y limitaciones que ofrece el *PSMove* junto con la *PSMoveApi* en un proyecto que utilice la cámara. Por un lado, se ha podido comprobar que cuando se utilizan dos mandos el consumo de recursos es muy alto y el juego comienza a perder calidad. Sin embargo, esto no limita

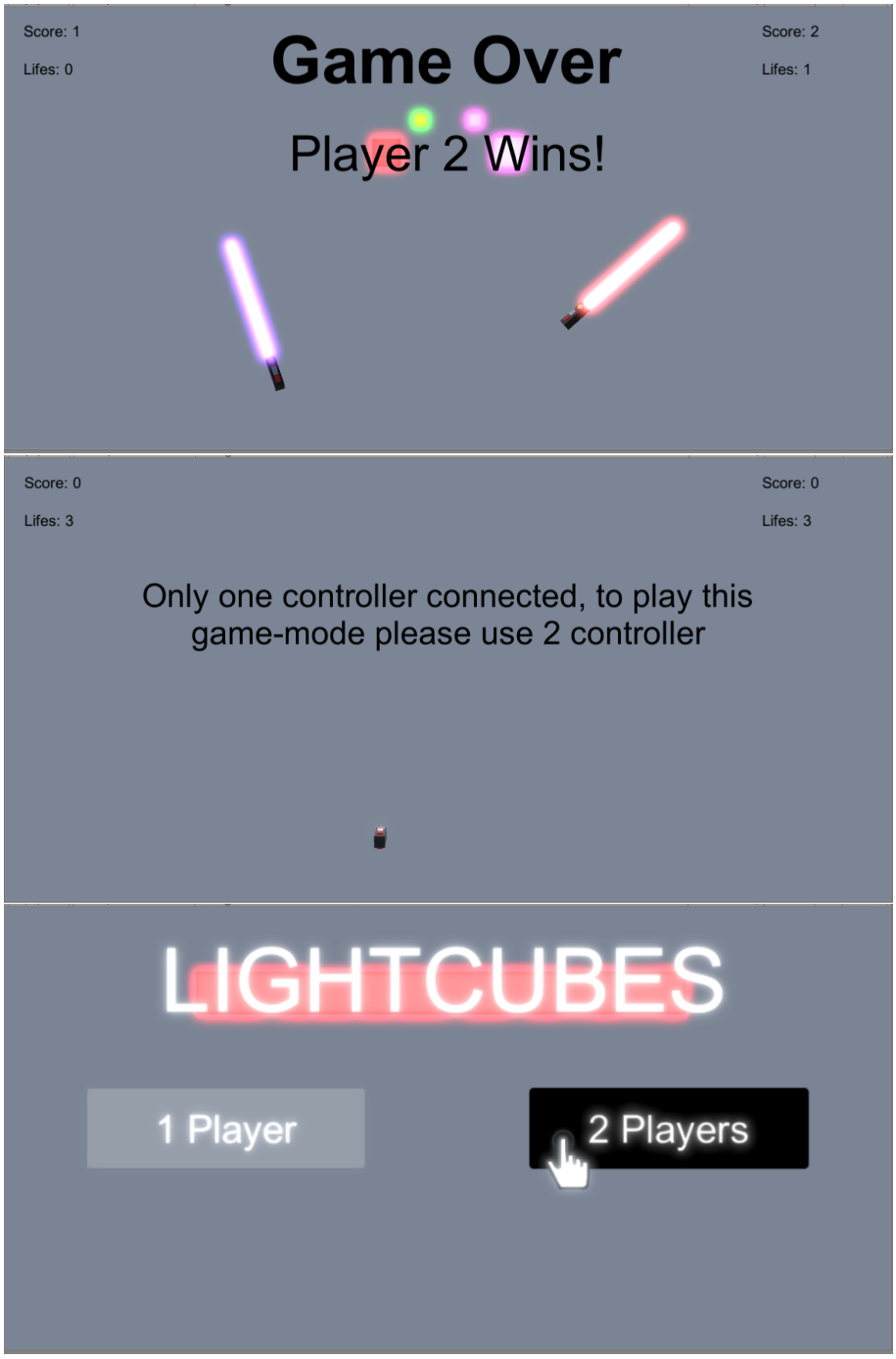

Figura 4.12: Capturas de pantalla de las modificaciones para dos jugadores. Arriba, nuevo mensaje al acabar la partida. En medio, comprobación de número necesario de mandos conectados. Abajo, nueva escena con menú principal.

las posibilidades que ofrece el *PSMove* ya que en aplicaciones donde no se requieran movimientos tan fluidos, se puede utilizar perfectamente si se añaden las modificaciones de capturar la imagen cada cierto número de fotogramas. Por otro lado, utilizando un único mando, no surgen problemas de rendimiento y por lo tanto se pueden desarrollar juegos que sean más dinámicos. En el siguiente prototipo, sección 4.2, se considerará una alternativa que no utilizará la cámara, pero en cambio no tendrá limitación con el número de mandos conectados.

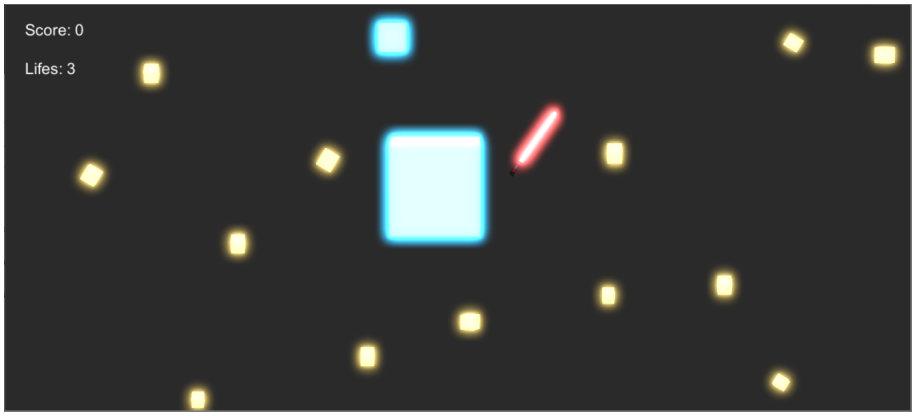

Figura 4.13: Captura de pantalla de las modificaciones finales del proyecto *Lightcubes*.

4.2 Segundo Prototipo - Space Duel

En este prototipo se ha buscado utilizar el mando con un enfoque distinto al experimento anterior. En *Lightcubes*, se utilizaba el *PSMove* junto a la cámara y se replicaban las acciones de este en el entorno virtual. Si se desplazaba el mando hacia la derecha, el objeto en pantalla imitaba este movimiento. En esta caso, se va a suprimir la cámara y se van a utilizar únicamente los valores internos del mando para simular una palanca de mando. Además, ya que no se requiere de la cámara y por lo tanto de los métodos que detectaban la posición del mando, el rendimiento esta menos limitado.

La idea de *Space Duel* consiste en un duelo entre dos jugadores en el que cada uno pilotará una nave y tendrá que derribar al rival tres veces. Además, el escenario contendrá una serie de asteroides de distinta forma y tamaño que dificultarán el movimiento de las naves.

Se utilizó el proyecto de *UniMove* como base ya que contenía todo lo necesario para desarrollar un juego en el que solo se utilizasen los valores internos del mando. También, se descargó un modelo de nave espacial en la *Unity Asset Store* y seguidamente se empezó a estudiar como replicar el movimiento de un *joystick*.

Para utilizar el *PSMove* como palanca de mandos la mejor forma es situarlo en perpendicular a la palma de una de las mandos y utilizar la otra para moverlo en la dirección deseada. En primer lugar, los controles del mando que trae *UniMove* por defecto estaban invertidos respecto a la orientación de la nave espacial. Una vez cambiados, se ha probado a trasladar la orientación del mando directamente a la nave. Esto no ha funcionado por una serie de razones. Primeramente, implementar el movimiento de esta forma impedía la realización de *loopings* ya que si se inclinaba el *joystick* con esta intención la nave simplemente replicaba la orientación del mando pero paraba cuando estaba en la misma posición. El segundo problema era que para rotar la nave en su eje 'z' (Figura 4.14) se tenía que girar el mando sobre la mano como si fuese un destornillador y esto no era nada ergonómico. Además, si se paraba a mitad el mando se quedaba al revés y por tanto los movimientos quedaban invertidos.

La solución a este problema era utilizar el método *transform.Rotate(Vector 3)*. Este rota el objeto utilizando los valores del vector que le pasas como parámetro. En este caso, el vector esta formado por los valores 'x' e 'y' de la orientación del *PSMove* multiplicados por diez para que el movimiento fuese más rápido. Como he dicho en el párrafo anterior, usar el valor 'z' del mando no es ergonómico, por lo tanto este se modificará con los botones cuadrado y triángulo del mando. El cuadrado rotará la nave hacia la izquierda y el triángulo hacia la derecha. El control de la nave quedaba entonces de la siguiente

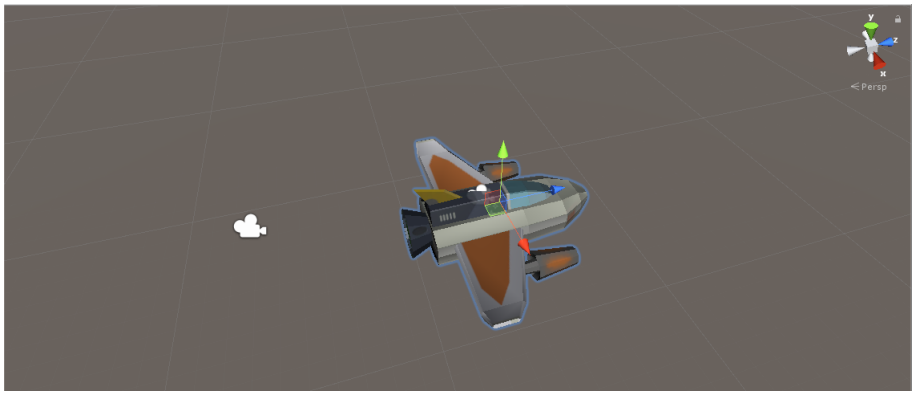

Figura 4.14: Captura de pantalla del modelo 3D de la nave y la orientación de sus ejes.

manera: inclinar el mando hacia delante baja la nave, hacia atrás la sube, inclinar hacia derecha e izquierda gira la nave en esa dirección respectivamente y los botones cuadrado y triángulo para rotar en el eje 'z'. El código para generar este comportamiento ha quedado de tal que así:

```
1         if (move.GetButtonDown(PSMoveButton.Square))
2         {
3             Zrotation = 1;
4         }
5         if (move.GetButtonDown(PSMoveButton.Triangle))
6         {
7             Zrotation = -1;
8         }
9         if (move.GetButtonUp(PSMoveButton.Triangle) || move.GetButtonUp(
              PSMoveButton.Square))
10        {
11            Zrotation = 0;
12        }
13        transform.Rotate(move.Orientation.x * 10f, move.Orientation.y * 10f,
              Zrotation);
```

El siguiente paso es implementar el disparo de la nave espacial. Para eso se han creado dos generadores de balas cada uno sitiado en uno de los cañones del modelo 3D. Para accionarlos se utilizará el *trigger* del mando para simular los controles de se utilizan normalmente en las naves de ciencia ficción. Al soltar el botón, el disparo cesará.

Para generar un escenario se utilizará una *skybox*. Una skybox es [50] «son una envoltura alrededor de su escena entera que muestra cómo el mundo se vería más allá de su geometría.». Se han encontrado unos que simulan el espacio y son los que se van a importar al proyecto. Asimismo, se van a añadir unos modelos de asteroides que se han encontrado por la *Unity Asset Store* y se van a distribuir por el mapa variando su tamaño para que el escenario quede menos monótono (Figura 4.15). Por último, se han añadido unos límites al mapa ya que el jugador podía volar indefinidamente hacia una dirección. Para hacerlo más justo, se ha añadido un mensaje de alerta para informar al jugador que se está acercando a este límite. Si la nave colisiona con el borde del mapa será destruida.

Ahora que el escenario ya esta acabado se le van a añadir unas mejoras a la nave. Primero, se va a implementar una mecánica para cambiar la vista de la nave como se hace en los juegos de este género. Se han añadido 4 vistas a la nave: vista por defecto, vista interior, vista superior y vista alejada. Se podrá iterar sobre estas utilizando el botón equis. Además, facilitará al jugador moverse por el mapa y esquivar los obstáculos.

Figura 4.15: Captura de pantalla de la escena con el *skybox* y los asteroides añadidos.

También, se ha añadido un modo súper velocidad que se activa pulsando el botón círculo. En este momento, no hay limitación de disparo ni de turbo, esta es la siguiente modificación que se va a implementar. Se empezará añadiendo una interfaz de usuario en la que se mostrarán el número de vidas y la cantidad de balas y turbo restante. Un jugador perderá la partida si se queda sin vidas. Las balas se recargarán automáticamente al cabo de tres segundos cuando se hayan acabado y el turbo se llenará constantemente, pero se gastará mucho más rápido.

Seguidamente, se ha duplicado la nave espacial y su interfaz de usuario para crear al segundo jugador. Para implementar la pantalla partida se han modificado las cámaras de cada nave, para que ocupasen solo la mitad de la pantalla. Como se tienen varias vistas, se han tenido que modificar los atributos de todas ellas. Además, para facilitar la localización de la nave rival, se ha creado una pirámide que siempre apunta a esta. Utilizando el método *LookRotation(Vector* 3*)* al que se le pasa la posición de la otra nave como argumento continuamente.

Se han añadido unas particulas de explosión, utilizando el *particle system*, cuando la nave pierde una vida. Ya sea cuando haya colisionado con un asteroide o haya recibido un disparo enemigo. Hasta ahora, la nave simplemente desaparecía y esto no mostraba suficiente información al usuario. Al añadir las particulas, se aprecia mejor que se ha perdido una vida. Al ser destruida, la nave volverá a aparecer en la posición en la que ha empezado la partida. También, para aprovechar todas las opciones que ofrece el *PSMove* y añadir inmersión al juego, se ha hecho que este vibre cuando la nave está disparando y cuando sea destruida.

Por último, para seguir con la temática de *Star Wars* se ha importado el modelo 3D de la nave *Arc*-170. Se han cambiado los generadores de balas y las distintas cámaras para que cuadrasen con esta nave y se ha dado por termindo el prototipo. El la figura 4.16, se puede ver como ha quedado este prototipo finalmente.

Como conclusión, ha quedado un juego bastante completo que aprovecha la ergonomía del *PSMove* para utilizarse como palanca de mando. No hay problemas de rendimiento y es una forma alternativa a desarrollar juegos con este dispositivo. Al utilizar únicamente los valores internos del *PSMove*, no se requiere cámara y no hay limitaciones de número de mandos conectados. En este caso, solamente se han utilizado dos porque no se disponían de más. La diferencia en el rendimiento entre usar uno o dos mandos ha sido inapreciable. En consecuencia, se puede suponer que conectar más mandos no lo afectará en exceso.

Figura 4.16: Captura de pantalla del prototipo acabado. Arriba, pantalla durante el juego. Ambas naves con la cámara por defecto y la de la izquierda, disparando. Abajo, pantalla de victoria/-derrota y demostración de las diferentes cámaras: a la izquierda cámara interior y a la derecha cámara superior.

4.3 Tercer Prototipo - Lightcubes Trinus VR

La idea de este prototipo es llegar a la imitación completa de los dispositivos de realidad virtual. En los dos anteriores, se ha experimentado con las funcionalidades del *PS-Move* pero estas se acercaban más hacia dispositivos de interacción avanzados que hacia dispositivos de realidad virtual. En primer lugar, estos prototipos no utilizaban *HMD* o casco de realidad virtual, que es un componente imprescindible en los productos de realidad virtual actuales.

Para conseguir este comportamiento junto al *PSMove*, se va a utilizar el servicio de *Trinus VR*. «*Trinus* es una solución software que enlaza el ordenador (o consola) y el teléfono para jugar a juegos en *VR*» [51], es decir ofrece la posibilidad de usar la realidad virtual sin la necesidad de un dispositivo de alta gama. Además, existe un *plug-in* para que se pueda utilizar este servicio directamente en *Unity*.

El *plug-in* se puede descargar directamente de la *Unity Asset Store*, en este caso lo importaremos en un nuevo proyecto basado en *Lightcubes*. Pero viene con algunos fallos, al parecer no se ha actualizado para ser compatible con la última versión de *Unity*, por lo tanto hay que modificar algunos métodos que se utilizan por la referencia más nueva. Estos métodos desfasados aparecen en la consola como errores y en esta se explican por cuales se deben sustituir. También, hay que cambiar algunas opciones dentro de la configuración del proyecto, que vienen explicadas en la guía de instalación y uso [51]. Entre estas, hay que volver a versiones anteriores de *Scrit Runtime*, que permite decidir que versión de *.NET* se quiere utilizar, y de *API Compatibility*, que le especifica al compilador como debe compilar el proyecto. Aunque estén obsoletas, se pueden utilizar sin problema y no hacen conflicto con la *PSMoveApi* ni con el proyecto de *Lightcubes*.

En *Unity*, un *prefab* es una manera de almacenar un objeto de *Unity* con todas sus propiedades para poder utilizarse cuando se necesite. El *plug-in* de *Trinus VR* incluye varios *prefabs* que se pueden añadir a la escena para empezar a utilizar el servicio. Los imprescindibles son *Trinus UI* y *SampleTrinusOnlyUIManager*, que se encargan de la interfaz de usuario y sus opciones, *TrinusManager*, que se encarga de todo el procesamiento del servicio como conectar el ordenador y el teléfono y *TrinusCamera*, que es la cámara que replica los movimientos de cabeza que hace el usuario. Sin embargo, este último no lo se utilizará ya que en el proyecto ya se tiene implementada una cámara. Así pues, se añadirá el *script TrinusCamera* directamente a esta cámara.

Una vez añadidos los necesarios, el proyecto ya está preparado. El siguiente paso, es descargar la aplicación el el *smartphone*, en este caso se instalará para *Android*. Aunque, *Trinus* ofrece diferentes opciones para conectar el móvil con el ordenador, primeramente se probará a enlazarlos mediante la misma red wifi.

Se ha ejecutado el proyecto y ha aparecido la interfaz de *Trinus* que se divide en tres pasos. Primero, una advertencia de que hay que tener instalada la aplicación para dispositivos móviles. Segundo, te explica las maneras de conectar el ordenador con el móvil. Tercero y último, empezar a jugar o salir del juego. Al seleccionar la opción de jugar, ha aparecido una cuarta ventana indicando que si no se conecta con el móvil automáticamente, hay que escribir la dirección *IP* de este manualmente. Esta dirección se muestra en la aplicación del *smartphone*, después de escribirla se muestra en el móvil la pantalla de la figura 4.17. Se puede apreciar que la pantalla se ha partido en dos para que utilizando un dispositivo como las *Google Cardboard* se pueda apreciar ese efecto de profundidad que usa la realidad virtual. La calibración del mando y el proceso de enlazar el móvil con el ordenador se pueden ejecutar en cualquier orden, incluso a la vez, esto permite que el proceso de empezar el juego no se vea ralentizado.

En esta primera prueba, no se ha utilizado un casco para colocar el móvil ya que solamente se quería comprobar que la conexión con *Trinus* funcionaba correctamente. No obstante, se han encontrado algunos problemas. La cámara se había rotado en el eje 'z' y para ver la pantalla de juego había que posicionar el *smartphone* con un ángulo incómodo. También, en una de las pruebas la conexión se ha caído, pero al volver a activar la aplicación móvil de *Trinus* se ha recuperado y ha seguido funcionando con normalidad. Se va a bloquear la rotación del eje 'z' ya que entorpece la experiencia de juego. Para esto, se ha buscado en el *script TrinusProcessor.cs* el método que se encarga de trasladar el movimiento del móvil al de la cámara del juego y se ha modificado el valor de la 'z' para que siempre sea cero.

Seguidamente, se han utilizado las gafas de la figura 4.18 para probar el servicio *Trinus* en su totalidad. Este ha respondido bastante bien, el rendimiento ha sido mejor de lo que se esperaba y aún no siendo óptimo se puede jugar. El movimiento en el eje 'x' no funciona adecuadamente ya que su posición inicial no es constante, a veces la cámara queda a la izquierda de los elementos en pantalla y otras a la derecha. En este proyecto, no se van a implementar elementos en todas las direcciones, únicamente los que se van a ver delante, por esto se ha decidido bloquear el movimiento del eje 'x' también. De esta forma, únicamente el movimiento vertical moverá la pantalla de juego.

Uno de los problemas que ocurren, es que nunca se detecta la conexión a la primera y siempre hay que añadir la dirección *IP* manualmente. Por ello, se va a probar el otro modo de conexión que permite *Trinus*, la conexión por USB. Para activarla, hay que modificar los ajustes que vienen por defecto en la aplicación móvil. En un principio, se creía que se enviaba la información necesaria por USB, sin embargo la solución que ofrece este

[7] Imagen extraída de https://saudi.souq.com/sa-en/vr-box-virtual-reality-3d-glass-for-3d-games-and-3d-movies-i/

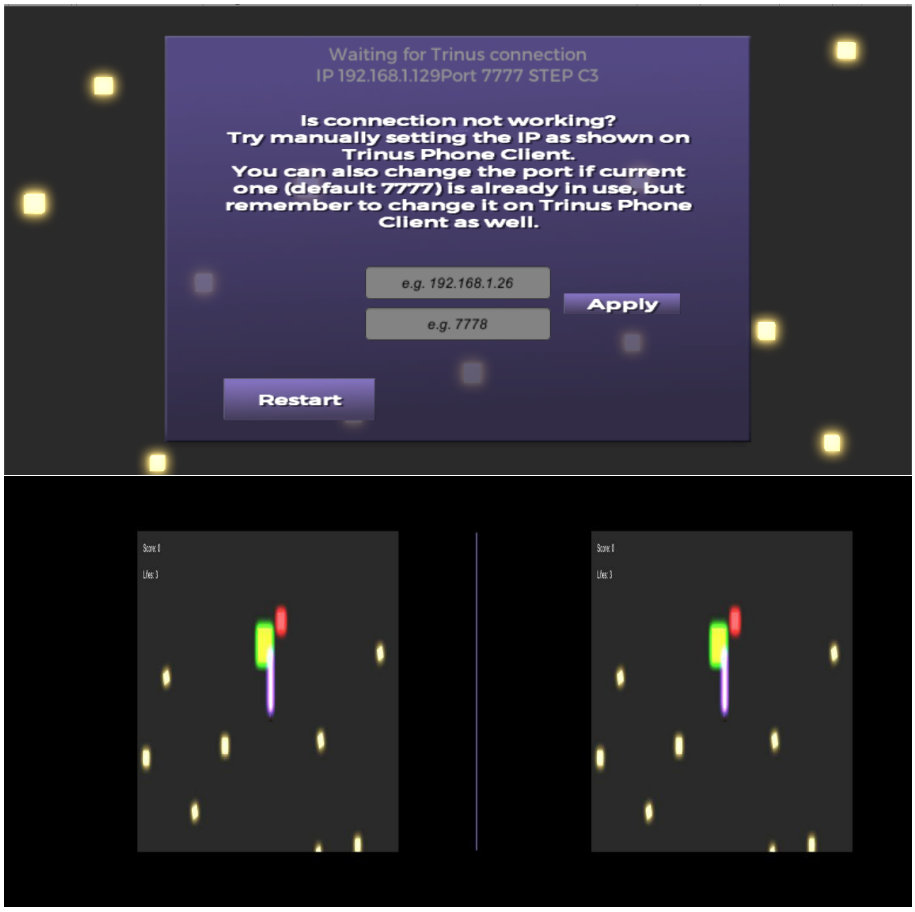

Figura 4.17: Captura de pantalla del proyecto *Lightcubes VR*. Arriba, pantalla para escribir la dirección *IP* manualmente. Abajo, vista en del proyecto en *smartphone*.

Figura 4.18: Gafas de realidad virtual para *smartphone* que se han utilizado en este proyecto. [7].

modo consiste en compartir el internet del teléfono móvil con el ordenador por USB. Con esto se asegura que ambos están conectados a la misma red. El rendimiento del juego no se ha visto afectado utilizando esta conexión, la única mejora ha sido que no ha hecho falta meter la dirección *IP* manualmente. Igualmente, al tener el *smartphone* conectado por USB se limita la libertad del usuario para mover la cabeza en todas las direcciones. Por lo tanto, se ha decidido descartar está opción de conexión y se va a buscar una manera de automatizar el proceso de introducir la dirección *IP*.

Como se ha podido observar durante todas las pruebas que se han hecho, la dirección *IP* ha sido siempre la misma, por ello se va a modificar el método que se encarga de leer y comprobar esta dirección y forzar a que siempre utilice la misma. Esta solución no es la más óptima ya que solo va a funcionar con este proyecto y con esta red wifi, pero será una manera de demostrar que este proceso sí se puede automatizar. El método que se encarga de obtener la dirección *IP* una vez se ha introducido es *ipEnteredOnConnection(string ip)*, se va a modificar para que el valor del *string* para que sea el que se quiere. Después, se hará que este método se ejecute una vez se haya presione el botón de empezar el juego. Hecho esto, el proceso ha quedado automatizado y funcional.

En conclusión, utilizar el *PSMove* junto con el servicio de *Trinus VR* ha ofrecido una aproximación más cercana a trabajar con los dispositivos de realidad virtual que están actualmente en el mercado. Existen algunas limitaciones como que el mando deba estar en el campo de visión de la cámara y que se requiera de una conexión estable a internet. No obstante, es un producto viable que se puede utilizar para el desarrollo de juegos, con la única desventaja de que si se utilizan dos mandos el rendimiento no es bueno y hay que buscar alternativas como limitar el número de llamadas al método de detectar la posición de los mandos.

4.4 Cuarto Prototipo - YogaMove

Un *serious game* o juego serio, como se ha explicado en la introducción, es un juego cuyo fin no es totalmente lúdico, sino que se utiliza en sectores más formales cómo pueden ser la medicina o la educación. En estos, cada vez se utilizan más las nuevas tecnologías para mejorar algunos tratamientos. Por esto, se ha decido desarrollar el último prototipo enfocándose en la industria de los *serious games*. En este caso, se creará un prototipo basándose en *Wii fit* (Figura 4.19) en el que se hacían estiramientos y ejercicios, pero mucho más sencillo. La idea es replicar los movimientos que se muestran en pantalla y al terminar enseñarle al jugador la precisión con la que los ha hecho.

Para empezar, se ha creado un nuevo proyecto *Unity* y se han añadido un modelo 3D y unas animaciones obtenidas de *Mixamo* [9]. *Mixamo* es una página web que ofrece de manera gratuita modelos y animaciones humanoides. Una vez importadas correctamente, se han enlazado utilizando el *animator* de *Unity* para que se reproduzca una detrás de otra. Seguidamente, se han creado tres cubos de diferentes tamaño, uno dentro de otro, que replican el movimiento de la mano de la figura. Esto se ha hecho, para que cada uno sumase una puntuación distinta al jugador. Cuanto más próximo se estuviese del cubo interior, más puntuación ya que este es el que imitaba la mano con más precisión.

El siguiente paso es implementar el movimiento del *PSMove*, se han utilizado las mismas técnicas que en *Lightcubes*, pero se ha bloqueado el movimiento del eje 'z' ya que se quiere trabajar en dos dimensiones únicamente. Para el cursor se ha utilizado una esfera.

[8] Imagen extraída de https://www.amazon.com/Nintendo-Wii-Balance-Board-Meter/dp/B00FE8WKPQ
[9] https://www.mixamo.com/#/

Figura 4.19: Imagen de ejercicio propuesto en el juego *wii fit* [8].

Para que el jugador puntúe, como se ha mencionado anteriormente, se quiere utilizar los tres cubos. La idea era detectar si había colisión con alguno de ellos y si la había detectar cual era el cubo más interior que se estaba detectando y sumar la puntuación correspondiente. El problema, es que *Unity* no permite detectar colisiones dentro de colisiones, es decir únicamente se detectaba el cubo exterior ya que los otros estaban en el interior de este como se muestra en la figura 4.20. Por lo tanto, se ha cambiado esta mecánica. Ahora, se utiliza solamente el cubo interior y el cálculo de puntuación utiliza la posición del cubos y la esfera en vez de las colisiones. Se calcula la posición de ambos en todo momento y se saca una puntuación respecto a lo cercanas que sean. Se han establecido tres rangos de distancia y cada uno con una puntuación distinta. Para obtener un porcentaje, en cada iteración se sumará la puntuación máxima que se puede obtener y al final se comparará está con la puntuación que ha obtenido el jugador. Esta modificación se puede apreciar en la figura 4.20.

El juego repetirá el ciclo de animación tres veces y al terminar mostrar la puntuación obtenida en pantalla. Además, se ha añadido un mensaje para que el jugador decida cuándo empezar la partida mediante la pulsación del botón *start* (Figura 4.21).

Como conclusión, aún siendo un juego muy sencillo, se muestra que el *PSMove* también se puede utilizar en este ámbito, en este caso se ha hecho un *serious game* basado en el deporte y la rehabilitación, pero con la tecnología disponible se podría ampliar a otras como simulador de operaciones en quirófano o reparación de vehículos.

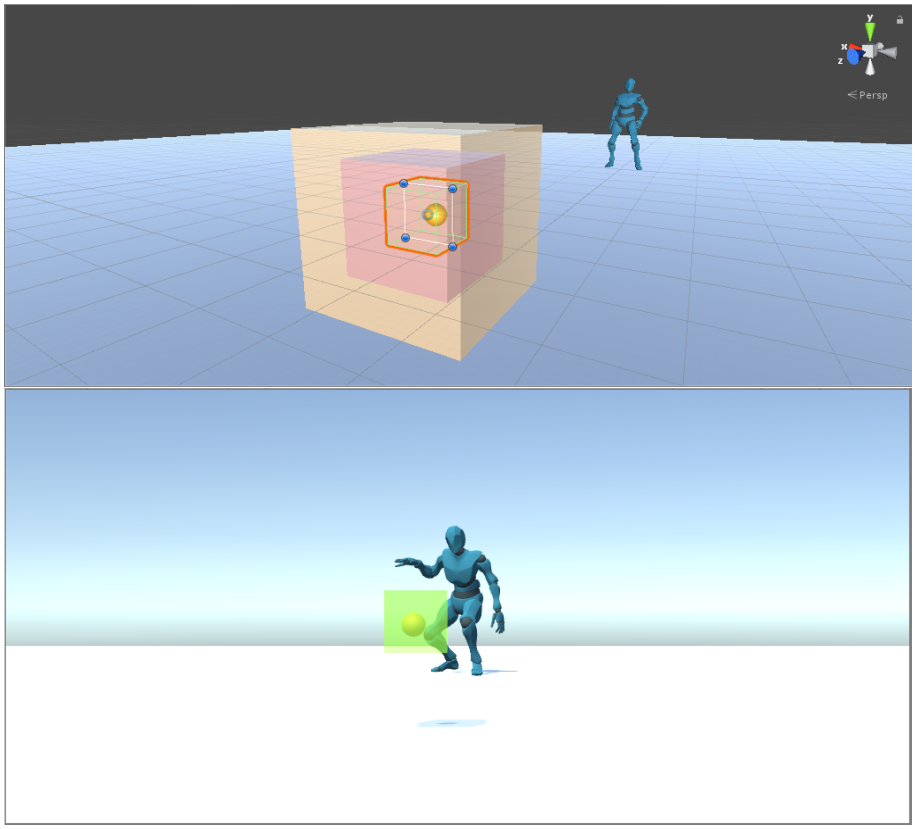

Figura 4.20: Capturas de pantalla del proyecto *YogaMove*. Arriba, primera prueba de cálculo de puntuación utilizando tres cubos. Abajo, implementación final utilizando únicamente el cubo interior y comparando su posición con la esfera.

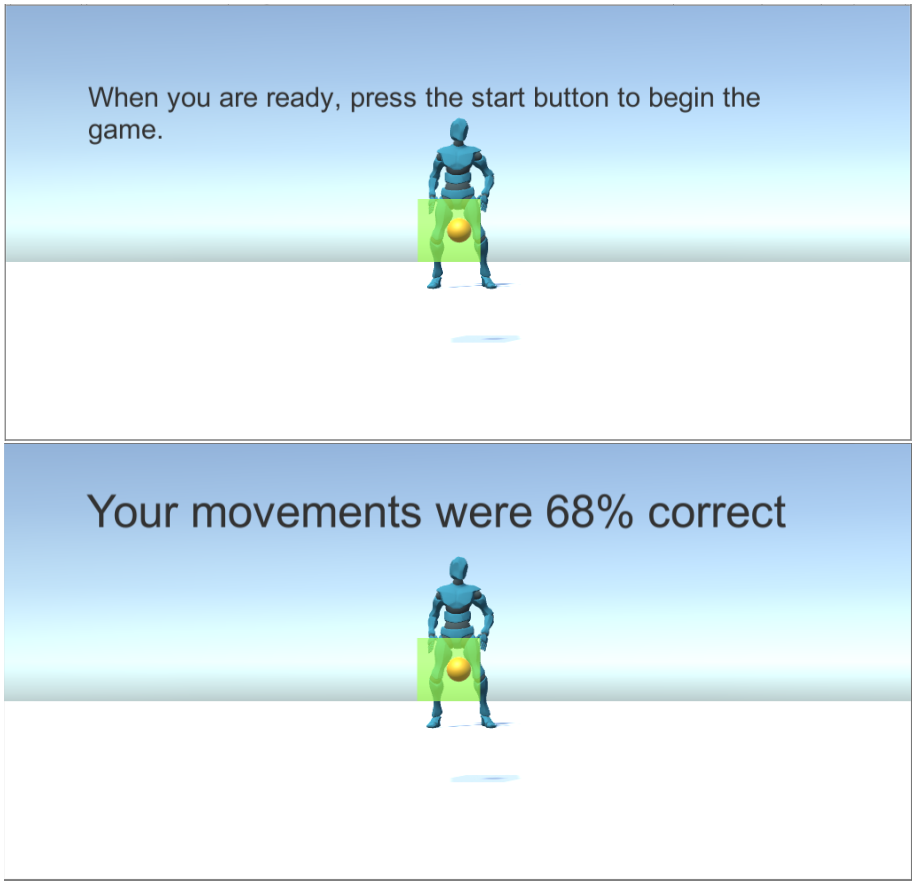

Figura 4.21: Capturas de pantalla del proyecto *YogaMove*. Arriba, mensaje de inicio. Abajo, mensaje final con la puntuación conseguida.

CAPÍTULO 5
Conclusiones

Para finalizar este proyecto se va a evaluar si los objetivos propuestos en la sección 1.1 se han cumplido. En primer lugar, se ha presentado la evolución de los dispositivos de interacción y la realidad virtual en el ámbito de los videojuegos, empezando por cómo se jugaba al *Pong* hasta llegar a los aparatos más novedosos que hay en el mercado actualmente. Mencionando los aspectos más importantes que se han ido mejorando o eliminando. Seguidamente, se han analizado los competidores más directos como el *WiiMote* y el *HTC Vive*.

Para conectar el *PSMove* al ordenador, se han barajado varias librerías y se evaluado los puntos positivos y negativos de cada una. Finalmente, se ha decidido utilizar *PSMoveApi* ya que ofrecía una mayor flexibilidad. Una vez hecho esto, se ha expuesto el proceso de instalación tanto en *MacOS* como en *Windows*.

En el proceso de desarrollo de los prototipos, se ha experimentado con el amplio abanico de posibilidades que ofrece este dispositivo. En primer lugar, se ha probado a utilizar para la funcionalidad que se desarrolló en su lanzamiento, un sistema de detección del mando que replica los movimientos del usuario. Después, se ha probado a desarrollar un juego que utilizase únicamente los valores internos del *PSMove* y para lograr esto se ha imitado el comportamiento de un *joystick*. Posteriormente, se ha utilizado la herramienta *Trinus VR* para generar un casco de realidad virtual con el *smartphone*. Por último, se ha enfocado más hacia el área de juegos serios desarrollando un juego de ejercicios.

Como conclusión a este proyecto, el mando *PSMove* ofrece la posibilidad de desarrollar juegos enfocados a la realidad virtual hasta cierto grado. Esto se debe principalmente a las limitaciones respecto al rendimiento que ocurren cuando se conecta más de un mando. En los dispositivos actuales se ha convertido en estándar utilizar dos mandos y en este aspecto el *PSMove* se queda atrás. Además, el uso del *smartphone* como *Head-Mounted display* no está al nivel de los cascos que se utilizan específicamente para esto. No obstante, el potencial del *PSMove* como dispositivo de interacción avanzado es alto. Se pueden experimentar con muchas mecánicas interesantes que utilicen las funcionalidades del mando para hacer juegos muy diferentes a los que se han desarrollado en este proyecto. En juegos que no requieran de movimientos muy fluidos o precisos, se pueden utilizar dos mandos sin ningún problema ya que se pueden limitar las llamadas a la detección del mando para mantener un buen rendimiento. En definitiva, el *PSMove* es un dispositivo perfecto para prototipar conceptos y mecánicas para juegos en realidad virtual cuando no se dispone de alguna herramienta más puntera.

En las asignaturas de 'Entornos de desarrollo de videojuegos' y '*Animation and Design of Videogames*' sí que se había utilizado *Unity*. Pero, en este proyecto, se ha podido experimentar más a fondo las capacidades que este ofrece y se ha obtenido un mayor conocimiento de la herramienta. Además, nunca se había probado a desarrollar juegos que

no utilizasen los dispositivos tradicionales, como el ratón y teclado, y se ha conseguido despertar un gran interés hacia este tipo de tecnologías que no se tenía antes de empezar con este proyecto.

Bibliografía

[1] Howard Rheingol. *Virtual Reality*. Summit Books, New York, 1991.

[2] Definición dispositivos de interacción humana o HID (human interface device). Disponible en https://www.philips.es/c-f/XC000010682/%C2%BFqu%C3%A9-es-el-dispositivo-de-interfaz-humano-%C2%BFlo-admite-mi-dispositivo-philips.

[3] Entrada de realidad aumentada en Wikipedia. Disponible en https://en.wikipedia.org/wiki/Augmented_reality.

[4] Entrada de Wikipedia del videojuego Pong. Disponible en https://en.wikipedia.org/wiki/Pong.

[5] Entrada de Wikipedia de la palanca de mando o joystick. Disponible en https://es.wikipedia.org/wiki/Palanca_de_mando.

[6] Entrada de Wikipedia de la consola Atari 2600. Disponible en https://en.wikipedia.org/wiki/Atari_2600.

[7] Entrada de Wikipedia de la cruceta o D-Pad. Disponible en https://es.wikipedia.org/wiki/Cruceta.

[8] Entrada de Wikipedia de Gunpei Yokoi. Disponible en https://es.wikipedia.org/wiki/Gunpei_Yokoi.

[9] Entrada de Wikipedia de la consola Nintendo Entertaiment System. Disponible en https://es.wikipedia.org/wiki/Nintendo_Entertainment_System.

[10] Entrada de Wikipedia de la consola Super Nintedo. Disponible en https://es.wikipedia.org/wiki/Super_Nintendo.

[11] Entrada de Wikipedia de la consola Virtual Boy. Disponible en https://en.wikipedia.org/wiki/Virtual_Boy.

[12] Entrada de Wikipedia de la consola Nintendo 64. Disponible en https://en.wikipedia.org/wiki/Nintendo_64.

[13] Entrada de Wikipedia de la consola PlayStation. Disponible en https://en.wikipedia.org/wiki/PlayStation_(console).

[14] Entrada de Wikipedia de la serie de videojuegos Dance Dance Revolution. Disponible en https://es.wikipedia.org/wiki/Dance_Dance_Revolution_(serie).

[15] Entrada de Wikipedia de la consola PlayStation. Disponible en https://en.wikipedia.org/wiki/PlayStation_(console).

[16] Entrada de Wikipedia de la consola Dreamcast. Disponible en https://es.wikipedia.org/wiki/Dreamcast.

[17] Entrada de Wikipedia de la cámara EyeToy. Disponible en https://en.wikipedia.org/wiki/EyeToy.

[18] Entrada de Wikipedia de la cámara Kinect. Disponible en https://es.wikipedia.org/wiki/Kinect.

[19] Entrada de Wikipedia de la consola Nintendo DS. Disponible en https://en.wikipedia.org/wiki/Nintendo_DS.

[20] Entrada de Wikipedia de la consola Nintendo 3DS. Disponible en https://es.wikipedia.org/wiki/Nintendo_3DS.

[21] Entrada de Wikipedia de la serie de juegos Singstar. Disponible en https://es.wikipedia.org/wiki/SingStar_(serie).

[22] Entrada de Wikipedia de la serie de la franquicia de juegos Guitar Hero. Disponible en https://es.wikipedia.org/wiki/Guitar_Hero.

[23] Entrada de Wikipedia de la consola Nintendo Wii. Disponible en https://es.wikipedia.org/wiki/Wii.

[24] Entrada de Wikipedia de la consola Nintendo Wii U. Disponible en https://es.wikipedia.org/wiki/Wii_U.

[25] Entrada de Wikipedia del mando PlayStation Move. Disponible en https://es.wikipedia.org/wiki/PlayStation_Move.

[26] Entrada de Wikipedia del visor de realidad virtual PlayStation VR. Disponible en https://es.wikipedia.org/wiki/PlayStation_VR.

[27] Entrada de Wikipedia de la compañia Oculus VR. Disponible en https://es.wikipedia.org/wiki/Oculus_VR.

[28] Entrada de Wikipedia del dispositivo Oculus Rift. Disponible en https://en.wikipedia.org/wiki/Oculus_Rift.

[29] Entrada de Wikipedia de las gafas de realidad virtual HTC Vive. Disponible en https://es.wikipedia.org/wiki/HTC_Vive.

[30] Página oficial del Xbox Adaptive Controller. Disponible en https://www.microsoft.com/es-es/p/xbox-adaptive-controller/8nsdbhz1n3d8.

[31] Flujo de trabajo de los Assets (Asset Workflow). Disponible en https://docs.unity3d.com/es/current/Manual/AssetWorkflow.html.

[32] Página principal de la librería PSMoveApi. Disponible en https://github.com/thp/psmoveapi.

[33] Página principal de la librería PSMove-Unity5. Disponible en https://github.com/HipsterSloth/psmove-unity5.

[34] Página principal del servicio PSMoveSerive. Disponible en https://github.com/psmoveservice/PSMoveService.

[35] Guía de instalación oficial de la PSMoveApi. Disponible en https://psmoveapi.readthedocs.io/en/latest/build.html#building-on-macos-10-12.

[36] Guía de instalación en MacOS de la PSMoveApi. Disponible en https://docs.google.com/document/d/16oB5jpbCpeRWb0o7w0b9bFbT3vEzWGqEPESWZUSCPuU/edit.

[37] Página oficial del gestor de paquetes para MacOS Homebrew. Disponible en https://brew.sh/index_es.

[38] Entrada del manual de Windows sobre el símbolo del sistema para desarrolladores de Visual Studio. Disponible en https://docs.microsoft.com/es-es/dotnet/framework/tools/developer-command-prompt-for-vs.

[39] Entrada de MinGW en Wikipedia. Disponible en https://es.wikipedia.org/wiki/MinGW.

[40] Documentación sobre la opción TESTSIGNIN de Windows. Disponible en https://technet.microsoft.com/en-us/ff553484(v=vs.96).

[41] Anuncio sobre la versión oficial de Unity para linux. Disponible en https://blogs.unity3d.com/2019/05/30/announcing-the-unity-editor-for-linux/.

[42] Página web oficial de UniMove. Disponible en http://www.copenhagengamecollective.org/projects/unimove/.

[43] Página web oficial de UniMoveX. Disponible en https://eric.itomura.org/unimovex/.

[44] Foro de Unity donde se pregunta como utilizar las .dylib en Unity. Disponible en https://answers.unity.com/questions/23615/how-to-make-unity-find-dylib-files.html.

[45] Entrada de XCode en Wikipedia. Disponible en https://es.wikipedia.org/wiki/Xcode.

[46] Documentación de Apple sobre los archivos Information Property List. Disponible en https://developer.apple.com/library/archive/documentation/General/Reference/InfoPlistKeyReference/Articles/AboutInformationPropertyListFiles.html.

[47] Página oficial del juego Beatsaber. Disponible en https://beatsaber.com/.

[48] Entrada del manual de Unity sobre los Animation Events. Disponible en https://docs.unity3d.com/Manual/animeditor-AnimationEvents.html.

[49] Entrada del manual de Unity sobre las escenas. Disponible en https://docs.unity3d.com/es/current/Manual/CreatingScenes.html.

[50] Entrada del manual de Unity sobre los Skyboxes. Disponible en https://docs.unity3d.com/es/current/Manual/class-Skybox.html.

[51] Manual de uso de Trinus VR para Unity. Disponible en https://usermanual.wiki/Pdf/trinusUnityInstructions.1232441561/view.