COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

ENTRENADOR DEL PERSONAL DE MANTENIMIENTO DE LA MINIFÁBRICA DE ICAI BASADO EN REALIDAD VIRTUAL

Autor: Borja Rojo González

Director: Jose Antonio Rodríguez Mondejar

Madrid

Julio de 2019

AUTORIZACIÓN PARA LA DIGITALIZACIÓN, DEPÓSITO Y DIVULGACIÓN EN RED DE PROYECTOS FIN DE GRADO, FIN DE MÁSTER, TESINAS O MEMORIAS DE BACHILLERATO

1º. Declaración de la autoría y acreditación de la misma.

El autor D. Borja Rojo González

DECLARA ser el titular de los derechos de propiedad intelectual de la obra: ENTRENADOR DEL PERSONAL DE MANTENIMIENTO DE LA MINIFÁBRICA DE ICAI BASADO EN REALIDAD VIRTUAL, que ésta es una obra original, y que ostenta la condición de autor en el sentido que otorga la Ley de Propiedad Intelectual.

2º. Objeto y fines de la cesión.

Con el fin de dar la máxima difusión a la obra citada a través del Repositorio institucional de la Universidad, el autor **CEDE** a la Universidad Pontificia Comillas, de forma gratuita y no exclusiva, por el máximo plazo legal y con ámbito universal, los derechos de digitalización, de archivo, de reproducción, de distribución y de comunicación pública, incluido el derecho de puesta a disposición electrónica, tal y como se describen en la Ley de Propiedad Intelectual. El derecho de transformación se cede a los únicos efectos de lo dispuesto en la letra a) del apartado siguiente.

3º. Condiciones de la cesión y acceso

Sin perjuicio de la titularidad de la obra, que sigue correspondiendo a su autor, la cesión de derechos contemplada en esta licencia habilita para:

a) Transformarla con el fin de adaptarla a cualquier tecnología que permita incorporarla a internet y hacerla accesible; incorporar metadatos para realizar el registro de la obra e incorporar "marcas de agua" o cualquier otro sistema de seguridad o de protección.
b) Reproducirla en un soporte digital para su incorporación a una base de datos electrónica, incluyendo el derecho de reproducir y almacenar la obra en servidores, a los efectos de garantizar su seguridad, conservación y preservar el formato.
c) Comunicarla, por defecto, a través de un archivo institucional abierto, accesible de modo libre y gratuito a través de internet.
d) Cualquier otra forma de acceso (restringido, embargado, cerrado) deberá solicitarse expresamente y obedecer a causas justificadas.
e) Asignar por defecto a estos trabajos una licencia Creative Commons.
f) Asignar por defecto a estos trabajos un HANDLE (URL *persistente)*.

4º. Derechos del autor.

El autor, en tanto que titular de una obra tiene derecho a:

a) Que la Universidad identifique claramente su nombre como autor de la misma
b) Comunicar y dar publicidad a la obra en la versión que ceda y en otras posteriores a través de cualquier medio.
c) Solicitar la retirada de la obra del repositorio por causa justificada.
d) Recibir notificación fehaciente de cualquier reclamación que puedan formular terceras personas en relación con la obra y, en particular, de reclamaciones relativas a los derechos de propiedad intelectual sobre ella.

5º. Deberes del autor.

El autor se compromete a:

a) Garantizar que el compromiso que adquiere mediante el presente escrito no infringe ningún derecho de terceros, ya sean de propiedad industrial, intelectual o cualquier otro.
b) Garantizar que el contenido de las obras no atenta contra los derechos al honor, a la intimidad y a la imagen de terceros.
c) Asumir toda reclamación o responsabilidad, incluyendo las indemnizaciones por daños, que pudieran ejercitarse contra la Universidad por terceros que vieran infringidos sus derechos e

intereses a causa de la cesión.

d) Asumir la responsabilidad en el caso de que las instituciones fueran condenadas por infracción de derechos derivada de las obras objeto de la cesión.

6º. Fines y funcionamiento del Repositorio Institucional.

La obra se pondrá a disposición de los usuarios para que hagan de ella un uso justo y respetuoso con los derechos del autor, según lo permitido por la legislación aplicable, y con fines de estudio, investigación, o cualquier otro fin lícito. Con dicha finalidad, la Universidad asume los siguientes deberes y se reserva las siguientes facultades:

- La Universidad informará a los usuarios del archivo sobre los usos permitidos, y no garantiza ni asume responsabilidad alguna por otras formas en que los usuarios hagan un uso posterior de las obras no conforme con la legislación vigente. El uso posterior, más allá de la copia privada, requerirá que se cite la fuente y se reconozca la autoría, que no se obtenga beneficio comercial, y que no se realicen obras derivadas.
- La Universidad no revisará el contenido de las obras, que en todo caso permanecerá bajo la responsabilidad exclusive del autor y no estará obligada a ejercitar acciones legales en nombre del autor en el supuesto de infracciones a derechos de propiedad intelectual derivados del depósito y archivo de las obras. El autor renuncia a cualquier reclamación frente a la Universidad por las formas no ajustadas a la legislación vigente en que los usuarios hagan uso de las obras.
- La Universidad adoptará las medidas necesarias para la preservación de la obra en un futuro.
- La Universidad se reserva la facultad de retirar la obra, previa notificación al autor, en supuestos suficientemente justificados, o en caso de reclamaciones de terceros.

Madrid, a 12 de Julio de 2019

ACEPTA

Fdo………………………………………………

Motivos para solicitar el acceso restringido, cerrado o embargado del trabajo en el Repositorio Institucional:

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título ………...…………ENTRENADOR DEL PERSONAL DE MANTENIMIENTO DE LA MINIFÁBRICA DE ICAI BASADO EN REALIDAD VIRTUAL………………………………………………………….……

…………………………………………………………………………………………

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico ……2018-2019………… es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: Borja Rojo González Fecha: 12/ 06/ 2019

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Jose Antonio Rodríguez Mondejar Fecha: 15/ 07/ 2019

mondejar@comillas.edu

Firmado digitalmente por mondejar@comillas.edu
Nombre de reconocimiento (DN): cn=mondejar@comillas.edu
Fecha: 2019.07.16 09:00:52 +02'00'

GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

ENTRENADOR DEL PERSONAL DE MANTENIMIENTO DE LA MINIFÁBRICA DE ICAI BASADO EN REALIDAD VIRTUAL

Autor: Borja Rojo González

Director: Jose Antonio Rodríguez Mondejar

Madrid

Julio de 2019

ENTRENADOR DEL PERSONAL DE MANTENIMIENTO DE LA MINIFÁBRICA DE ICAI BASADO EN REALIDAD VIRTUAL

Autor: Rojo González, Borja

Director: Rodríguez Mondejar, Jose Antonio

Entidad Colaboradora: ICAI – Universidad Pontificia de Comillas

RESUMEN DEL PROYECTO

Introducción

La realidad está formada por aquello que existe de forma auténtica o verdadera. Virtual, por su parte, es lo que cuenta con la virtud de generar un efecto, pero que no se concreta en lo físico. Podría decirse, por tanto, que la idea de realidad virtual enfrenta dos conceptos que resultan opuestos. Sin embargo, la noción es de uso habitual para referirse a aquel entorno informático que representa, de manera digital, algo que simula ser real. Mediante diversos equipos y programas informáticos, la realidad virtual genera una simulación de la realidad [1].

A pesar de que el desarrollo de la RV se ha visto principalmente impulsado por su uso en el área de entretenimiento, la industria está aprovechando esta tecnología para aplicarla a lo que se denomina la cuarta revolución industrial o Industria 4.0.
El ámbito de la formación de técnicos especializados en la manipulación de maquinaria industrial es una de las aplicaciones de la tecnología de Realidad Virtual. Igual que es posible generar entornos donde visualizar las máquinas funcionando de manera real, es posible permitir a los usuarios interactuar con ellas para la resolución de averías o incidencias.
Simplemente con unas gafas de realidad virtual es posible colocar a un operario en un entorno virtual donde se visualice una máquina que ha detenido su producción para que este compruebe su funcionamiento y solucione la incidencia tal y como si se la encontrara en el entorno real de fábrica.
Así la Realidad Virtual se sitúa cercana al ámbito del training ofreciendo a las empresas industriales una herramienta mucho más óptima y económica para la formación de sus trabajadores. Pues, gracias a ella, se elimina el coste del técnico que debe hacer la formación a cada trabajador, se optimizan los tiempos y se evita tener que parar la producción para formar a estos trabajadores o permitirles que accedan a las máquinas para practicar su uso.
En este sentido, más idónea incluso, es su aplicación para el entrenamiento en situaciones
de riesgo. Una de las empresas que ya ha usado esta tecnología es 3M que, a través de una simulación virtual, muestra a los trabajadores de la construcción los riesgos de trabajar sin las correctas medidas de seguridad. En este caso, al ahorro de costes antes mencionados, se debe añadir la ventaja de evitar poner en riesgo a los trabajadores de forma innecesaria [2].

El objetivo de este proyecto es desarrollar una aplicación que permita la formación de posibles técnicos de mantenimiento de la minifábrica de ICAI y estudiar las posibles ventajas de la aplicación en caso de ser desarrollada en torno a una planta que todavía no ha sido o está en proceso de ser construida. La aplicación situará al usuario en un entorno virtual que simula el laboratorio de sistemas digitales de ICAI. Una vez dentro,

se expondrán diferentes averías que el usuario tendría que identificar basándose en el funcionamiento de la planta. Con el uso de este tipo de simuladores se pretende:
Formar técnicos: El objetivo principal de la aplicación es ayudar en la formación de técnicos, simulando el entorno en el que trabajarán, sin necesidad de parar la planta y sin que los errores cometidos conlleven consecuencia alguna en la planta. La ventaja de la realidad virtual es que es mucho más inmersiva que un simulador en ordenador, y, por lo tanto, cuando el técnico comience a trabajar en la planta real, ese cambio sería mucho menos drástico, ya que estaría familiarizado con el entorno.
Seguridad: La minifábrica de ICAI consta de varios brazos robóticos. Los motores que controlan estos brazos son muy potentes, y, en caso de avería, un técnico no formado podría verse involucrado en un accidente laboral debido a la falta de experiencia. De este modo, los técnicos novatos pueden realizar las prácticas sin riesgo de lesiones u otros accidentes.
Agilizar la puesta en marcha de la planta: En caso de ser una planta nueva, mediante el uso de la realidad virtual, se puede formar a los operarios mientras que la planta está construyéndose, de modo que cuando esté terminada, los mismos puedan incorporarse inmediatamente, ya que habrían practicado en un entorno virtual, que posee las mismas características y dimensiones que la planta real. Se ahorrarán tiempos muertos, pudiendo comenzar la producción en el menor tiempo.
Resolver problemas de diseño: Al tratar con un espacio virtual, durante la formación de los técnicos se podría detectar con más facilidad fallos en el diseño de las instalaciones, de modo que antes de empezar a construir el complejo, se puedan hacer los cambios pertinentes.

Metodología

Para realizar el proyecto se ha hecho uso de Oculus Rift, Unity y Steam VR. Oculus Rift es un sistema de realidad virtual que permite visualizar espacios generados digitalmente, ofreciendo una sensación inmersiva en el entorno digital. Unity es el motor de desarrollo de videojuegos en el que se ha desarrollado la aplicación. Para poder implementar Oculus Rift en Unity se ha utilizado Steam VR, una aplicación que ofrece un SDK con códigos para facilitar la implementación de la realidad virtual en Unity.

En esta versión de la aplicación se han desarrollado dos tipos de fallos o averías de la planta, el sensor de movimiento y el gatillo. La aplicación sitúa al usuario en el laboratorio de ICAI. Aleatoriamente ocurrirá un fallo en la planta, que el usuario tendrá que identificar y detectar. Se contarán los fallos y los aciertos que el operario haya tenido. Además, para posteriores versiones de la aplicación, se ha desarrollado una plantilla de código C# para facilitar la incorporación de nuevas averías en el programa.

En primer lugar, se desarrolló la ‘escena’. Las ‘escenas’ en Unity son los entornos en los que el usuario va a interaccionar con la aplicación, en este caso el laboratorio de sistemas digitales de ICAI. Éste se ha modelado utilizando un modelo de la planta importado anteriormente en Unity. También se ha modelado el laboratorio en sí para favorecer la sensación de inmersión, utilizando modelos importados de sitios web como 3D Warehouse o Unity Asset Store. En la escena se especifican como interactúan los diferentes objetos entre ellos, añadiendo componentes como *colliders*,*scripts* y *animations* entre otros.

Por otro lado, se han desarrollado los scripts que hacen funcionar los diferentes dispositivos y establecen el sistema de comunicaciones entre los objetos. Tanto el sensor de movimiento como el gatillo contienen *scripts* que funcionan de forma independiente. Éstos se comunican a su vez con el gestor de fallos (Failure Manager). El gestor de fallos es el encargado de coordinar todos los elementos. Al inicio de la aplicación, el gestor de fallos inicia una cuenta atrás. Cuando llega a 0, se establece aleatoriamente una avería. El gestor almacena el ID de la avería seleccionada. A continuación, el usuario procede a investigar la incidencia y selecciona el objeto que cree que no funciona correctamente. Una vez seleccionado, el gestor de fallos comprueba si es el objeto correcto o no. En caso negativo, se suma un fallo. En caso afirmativo, se suma un acierto y se reinicia la cuenta atrás.

Resultados y conclusiones

Unity es un motor centrado en la creación de aplicaciones de entretenimiento, y por lo tanto, carece de herramientas que faciliten el desarrollo de aplicaciones más técnicas, como compatibilidad directa con modelos en otros formatos como '.dwg' o '.sldasm'. Aun así, Unity Technologies está trabajando con grandes empresas, especialmente del sector del automóvil para solucionar los problemas.
Con el desarrollo de este proyecto, se demuestra, que, a pesar de las carencias técnicas de Unity, se pueden crear aplicaciones de simulación industriales, especialmente en el área de formación de personal. El uso de simuladores en cualquier entorno facilita el aprendizaje y la formación. En el caso de una gran fábrica industrial, el mantenimiento de la planta es de especial importancia, especialmente si los beneficios se cuentan por horas. Para formar correctamente a los técnicos haría falta parar la producción o esperar a que se produjese en una avería, lo que se traduce en importantes pérdidas económicas. No solo eso, sino que también, según el tipo de planta, se podría comprometer la seguridad de los técnicos. El uso de simuladores de este tipo permite al personal llegar a la planta real con la formación necesaria para afrontar el problema.
La gran ventaja de la realidad virtual frente a los simuladores convencionales es que no hace falta una réplica física de la máquina/dispositivo que se vaya a reparar. Se puede generar una réplica virtual que funcione de la misma manera que lo haría la real, a partir de los planos de este. De este modo hay un ahorro en el coste de producción del equipo, con la ventaja de que el entorno virtual se puede actualizar, y en caso de que se cambie algún componente de la planta de la fábrica, se podría hacer en el entorno virtual de la misma manera.
Algunas empresas, como PixoGroup, han desarrollado aplicaciones de formación, que permiten el uso simultáneo de la aplicación de varios operarios a la vez, de modo que se puedan entrenar dinámicas en las que se requiera más de una persona, desarrollando un método de formación que ayuda a los alumnos a retener mejor lo que aprenden durante la formación.
En conclusión, los programas de entrenamiento en Realidad Virtual son una realidad presente que, aunque todavía no se haya establecido completamente, está experimentando
un gran crecimiento. Grandes empresas industriales están invirtiendo en este tipo de aplicaciones, convirtiéndose la Realidad Virtual en uno de los principales impulsores hacia la implementación de la Industria 4.0.
El futuro de este tipo de aplicaciones dependería de cómo se desarrolle la Realidad

Virtual. En el caso de programas de formación, se podrían generar sistemas de enseñanza
que formasen y examinasen a los operarios, mediante un sistema de tutoriales y lecciones
virtuales. En cuanto a aplicaciones directamente relacionadas con el mantenimiento, sería de especial interés ser capaces de conectar directamente la planta en sí con la aplicación, de modo que los movimientos y estado de la planta física se reflejen en la virtual, así el operario podría diagnosticar los problemas reales de la planta sin necesidad
de estar, ni si quiera, en el mismo edificio; o poder controlar la planta desde la realidad virtual, minimizando el contacto físico entre operario y maquinaria. Otro de los posibles desarrollos es el diseño de las plantas, mediante la creación de modelos a partir de planos
de CAD, poder visualizar y testear una planta de manera virtual, para comprobar
su funcionamiento y distribución. Ésto sería posible mediante la creación, en caso de Unity, de modelos 'prefabs' que se puedan reciclar para ser utilizados en diferentes proyectos, ahorrando costes de diseño y programación. Por otro lado, alejado de la industria manufacturera, se podrían diseñar aplicaciones que simulasen incidentes en centros donde los accidentes son inusuales pero críticos, como el caso de las centrales nucleares. En estas centrales es difícil entrenar operarios de manera realista, ya que las incidencias son mínimas, pero a la vez son muy peligrosas. Por esto, una formación previa
en un simulador de realidad virtual podría, en casos extremos, prevenir catástrofes.

Referencias

[1] Julián Pérez Porto y María Merino. Definición de realidad virtual. 2013. Recuperado el 1 de Julio de 2019. url:definicion.de/realidad-virtual.
[2] Amparo Caballero. Industria 4.0 a trav_es de Realidad Virtual y Realidad Aumentada. 2017. Recuperado el 1 de Julio de 2019. url: http://www.innoarea.com/industria-4-0-a-traves-de-realidad-virtual-y-realidad-aumentada/.

VIRTUAL REALITY BASED MAINTENANCE TECHNICIAN TRAINER FOR ICAI'S MINIFACTORY.

Author: Rojo González, Borja

Director: Rodríguez Mondejar, Jose Antonio

Collaborating entity: ICAI – Universidad Pontificia de Comillas

PROJECT SUMMARY

Introduction

Reaility is formed by all that is real or existent. Virtual, on the other hand, is what has the virtue of generating an effect that does not materialize in the physical world. It could be said, that the idea of virtual reality faces two opposite concepts. However, the term is often used to refer to something that simulates reality. By using different gear and programs, virtual reality generates a simulation of the real world [1].

Even though VR has been developed thanks to its use in the entertainment industry, the technical industry is taking advantage of it by applying it to what it is called the 4^{th} industrial revolution or Smart Industry. The training of specialized technicians in the handling of industrial machinery is one of the applications of virtual reality in the industry. In the same way that it is possible to generate virtual industry environments where working machines can be visualized, it is possible to allow users to interact with those machines to learn how to solve faults or failures. With a VR Headset it is possible to place the technician in a digital environment where he visualizes the failing machine that has stopped working, so he can check it and solve the problem as if he was in the real factory or plant.

This way, Virtual Reality is a great application in the training ambit, offering industrial companies a much more optimal and economical tool to train their employees. The cost of the trainer is eliminated; production times are optimized; and the stop of the production to train the technicians and allow them access to the machines is avoided. In this sense, the use of VR in occupational hazards is even better. 3M is one of the companies, that using VR, shows construction workers the risks of working without taking safety measures. In this case, to the cost savings advantages previously mentioned, the advantage of avoiding risking the workers unnecessarily must be added to the list [2].

The main purpose of this project is to create an application that allows the training of possible maintenance technicians at ICAI's minifactory and study the advantages of the application, in case of being developed for a factory that has not yet or is being built. The application will place the user in a virtual environment that replicates the digital systems laboratory of ICAI. Once inside, the technician will be exposed to different faults, that he will have to identify. With the use of this type of simulators, the intention is to:
Train technicians: The purpose of the application is to develop a training program for technicians that simulates the plant/factory where they will work, without the need of pausing the production, and without the risk that errors would entail in the plan. The advantage of VR is that it is much more immersive than traditional simulators, so when

the technician starts working in the factory, the change would be less harsh, as he would be already familiarized with the environment.
Safety: ICAI's minifactory is equipped with four robotic arms. The motors that move these robots are very powerful and, in case of fault, a non-formed technician could be injured due to lack of experience. This way, unformed technicians can practice without any kind of risk.
Speed up the start-up of the plan: By using VR, employees can be trained while the factory is being built, so when it is finished, the technicians will be able to work immediately.
Solve design problems: Since the plan has been modelled in 3D, it will be easier to detect problems in the design of the plant, so that before building the facility, this problem can be solved.

Methodology

In order to carry out this project Oculus Rift, Unity and Steam VR had been used as the main tools of development. Oculus Rift is a headset that allows the use of virtual reality in digital environments. Unity is the game engine where the application has been developed. Finally, Steam VR allows VR to be implemented in Unity, by the use of an SDK.

In this version of the application, two types of faults have been developed. The movement sensor and the stopping device. The application places the user in the digital systems laboratory at ICAI. Randomly the different devices will start failing, the user will have to detect and identify the source of the fault. Hits and misses will be scored. Additionally, a script template has been developed, in order to ease the implementation of faults in the future.

On the one hand, the 'scene' was created. 'Scenes' in Unity are the environments where the user interacts with the application, in this case, the digital systems laboratory of ICAI. It has been modelled using the model of the plant imported previously in Unity. The laboratory has also been modelled using models imported from websites such as 3D Warehouse or Unity Asset Store. In the 'scene' interactions between objects are coordinated by the addition of components such as colliders, scripts or animations.

On the other hand, scripts that make the different devices work and objects communicate between them had been developed. Both the movement sensor and the stopping device contain scripts that work independently. They communicate with the 'Failure Manager'. The Failure Manager coordinates all the objects in the scene. At the beginning of the application, it starts a countdown. Once it reaches 0, a fault is stablished randomly. The manager stored the fault's ID. Then, the user investigates the fault and selects the object that he believes is failing. Once selected, the failure manager checks if it is the correct one. If so, it adds a hit and restarts the countdown, else, it will add a miss.

Results and conclusions

Unity is an engine mainly used in the creation of entertainment applications, and therefore, it lacks many of the tools that would ease the development of more technical applications, such as the compatibility with formats such as '.dwg' or '.sldasm'. Even

so, Unity Technologies is working with big companies, especially from the automotive industry to solve these problems.
This project has demonstrated that, despite de lack of technical tools in Unity, industry simulation applications can be created, especially in the training ambit. The use of simulators eases training and learning. In the case of a big industrial factory, the plant maintenance is very important, especially when the benefits are counted by hours. To properly train technicians the pause of the production is needed. This translates in economic losses. Not only that, but the safety of the employees is compromised. The use of VR simulators allows staff to start working with enough knowledge to face any problem.
The great advantage of VR over traditional simulators is that there is no need to create a replica of the machine. A virtual replica can be generated based on the plans. This way, there is a cost saving in the production of the simulator, and in case of any update in the machinery, it could be done the same way.
Some companies, such as PixoGroup, have developed training programs that allow the simulatenous use by differente users of the application, allowing trainees to work together.
In conclusion, VR training programs are a present reaility, even though this technology is not completely stablished, it is experiencing a great growth. Big industrial companies are investing in this type of applications. Becoming Virtual Reality one the boosters of the Smart Industry.

References

[1] Julián Pérez Porto y María Merino. Definición de realidad virtual. 2013. Retrieved July 1, 2019. url:definicion.de/realidad-virtual.
[2] Amparo Caballero. Industria 4.0 a trav_es de Realidad Virtual y Realidad Aumentada. 2017. Retrieved July 1, 2019. url: http://www.innoarea.com/industria-4-0-a-traves-de-realidad-virtual-y-realidad-aumentada/.

Índice general

Parte I

Memoria

Capítulo 1

Introducción y estado del arte

La realidad está formada por aquello que existe de forma auténtica o verdadera. Virtual, por su parte, es lo que cuenta con la virtud de generar un efecto, pero que no se concreta en lo físico. Podría decirse, por tanto, que la idea de realidad virtual enfrenta dos conceptos que resultan opuestos. Sin embargo, la noción es de uso habitual para referirse a aquel entorno informático que representa, de manera digital, algo que simula ser real. Mediante diversos equipos y programas informáticos, la realidad virtual genera una simulación de la realidad. Para ello, se hace uso de un casco con un visor que proyecta las imágenes, además de contar con diversos sensores para poder interactuar con el entorno[15].

En la actualidad, esta tecnología está siendo desarrollada en diversos campos, como pueda ser la educación, entretenimiento, software, videojuegos o medicina. En la industria se están desarrollando numerosas aplicaciones tanto de realidad virtual como aumentada, especialmente en el área de simulación y entrenamiento, con objetivos tales como: formación, resolución de problemas o mejorar las medidas de seguridad. Con el desarrollo de la industria 4.0, el consumo de realidad virtual aplicada a la industria es actualmente del 40 % [19]. Es por ello que las empresas deben actualizarse y adaptarse a este tipo de tecnologías que están cobrando cada año más importancia.

En este trabajo de fin de grado se pretende crear una aplicación que permita formar a técnicos de mantenimiento sin necesidad de interactuar físicamente con la planta. La aplicación se centrará en la planta de la minifábrica de ICAI. Ésta consta de una pequeña línea de montaje con varios brazos robóticos. La aplicación situará al técnico en la planta, donde se expondrá a diferentes fallos comunes, que tendrá que identificar y señalar.

1.1. Historia

Los verdaderos orígenes de la realidad virtual se remontan dos siglos atrás, concretamente al año 1844, cuando Charles Wheatstone creó el estereoscopio, un medio que consiste en obtener dos fotografías prácticamente idénticas, cuya única diferencia radica en el punto de toma de la imagen, lo que provoca que sean observadas por cada ojo de forma independiente, lo que hace que el cerebro las mezcle en una sola creando un objeto tridimensional. Esta técnica sería la base de los primeros visores de realidad virtual, e incluso a día de hoy siguen utilizándose dichos patrones.

Figura 1.1: Estereoscopio

Fuente: caminosantiago360.com[6]

Los verdaderos primeros pasos llegaron en el año 1958. La empresa *Philco Corporation* ideó un sistema de realidad virtual que fue diseñado para generar entornos artificiales a los que podían acceder personas mediante un casco que captaba los movimientos que los usuarios realizaban con sus cabezas. Con el paso de los años se crearon dispositivos más eficientes, e incluso la NASA apoyó a varias empresas en el desarrollo de diversos dispositivos con el objetivo de crear ambientes interactivos con los que se pudiera captar el movimiento de cualquier parte del cuerpo, incorporando además sustanciosas mejoras visuales, como entornos tridimensionales y estereoscópicos. El primer casco verdaderamente VR llegaría en 1961, construido por Corneau y Bryan, empleados de *Philco Corporation*.
El concepto en sí de 'Realidad Virtual' nace en el año 1965, cuando Ivan Sutherland, considerado como uno de los pioneros de la informática, publicó un artículo llamado *The Ultimate Display*, en el que explicaba el concepto que los científicos llevaban años tratando de asentar. El propio Sutherland desarrollaría dos años después el primer programa diseñado para crear mundos virtuales con imágenes en 3D[13].
En 2007 Google introdujo lo que se podría considerar como el inicio de realidad virtual moderna, con *Google Street View*, un servicio que nos muestra vistas panorámicas de gran parte de las carreteras de nuestro planeta. En 2010, Palmer Luckey diseñó el primer prototipo de Oculus Rift, que es en la actualidad uno de los sistemas de RV de referencia.
Actualmente otras grandes empresas tecnológicas como Facebook, Apple, Amazon, Microsoft, Sony o Samsung han abierto departamentos centrados en realidad virtual y aumentada. En 2016 la empresa HTC lanzó al mercado el casco HTC VIVE, que a día de hoy está compitiendo por el liderazgo del sector con Oculus Rift[4].

1.2. Funcionamiento

Para hacer uso de la realidad virtual se requieren de dos elementos fundamentales: un equipo informático, y unas lentes o casco. El equipo puede ser un ordenador, un móvil o una tablet. Por otro lado, las lentes VR son por lo general productos diseñados por empresas, que pueden ir desde un simple dispositivo construido en cartón, hasta cascos completamente independientes, como puede ser el recientemente lanzado Oculus Go, que incluye el equipo informático y es completamente portátil. Las gafas/casco de realidad virtual más utilizadas actualmente son Oculus Rift y HTC Vive.
Las características principales por las que se diferencian los diferentes sistemas de RV son:

- **Resolución:** La cantidad de píxeles que tiene la pantalla. En realidad virtual es de especial importancia, puesto que los ojos están muy cerca de la pantalla, por tanto, a mayor cantidad de píxeles, mayor será la calidad de la imagen.

- **Tasa de refresco:** Mide la fluidez de las imágenes en pantalla. Es la cantidad de veces que se actualiza la imagen en el tiempo. Es también importante, ya que al controlar los movimientos con la cabeza, éstos son a menudo bruscos y rápidos, por lo que una tasa de refresco baja podría producir mareos.

- **Ángulo de visión:** Campo de visión que ofrecen las gafas.

- **Sensores:** Son necesarios para registrar los movimientos, conocer la posición del usuario y poder interactuar con el entorno. Los hay internos (incorporados en el casco) y externos (normalmente cámaras).

- **Área de rastreo:** Superficie dentro de la cual los movimientos son registrados por los sensores[20].

1.3. Uso de la realidad virtual en la industria

A pesar de que el desarrollo de la RV se ha visto principalmente impulsado por su uso en el área de entretenimiento, la industria está aprovechando esta tecnología para aplicarla a lo que se denomina la cuarta revolución industrial o Industria 4.0.
Hay sectores industriales donde el prototipado de productos requiere de altos costes de inversión; es necesario crear físicamente los productos para que se puedan apreciar sus características y analizar pormenorizadamente su diseño.
Un ejemplo de industria que requiere de estos procesos de alto coste es la automovilística. Hasta el momento era necesario construir, aunque con calidades inferiores, el modelo que luego se iba a comercializar para analizar su uso real.
La Realidad Virtual es la herramienta perfecta para reducir los costes derivados de las tareas de prototipado. Esta tecnología permite crear una simulación casi real del futuro producto donde se visualizarán todas las características del mismo como si se tuviera

físicamente delante de los ojos. Esta creación virtual, además, permite probar diferentes opciones de acabado sin inversión en prototipado.
BMW ha sido una de las primeras marcas en hacer uso de la Realidad Virtual para visualizar, entre otros aspectos, nuevos diseños de interiores de los vehículos, así como otras características físicas de sus coches. Estas simulaciones sirven para comprobar aspectos como la visibilidad interior del coche o si el tablero de instrumentos y controles se ve adecuadamente desde la posición del conductor. Esta tecnología ha permitido a la compañía alemana tomar decisiones sin tener que recurrir a los costosos túneles de simulación utilizados anteriormente para el prototipado.

Figura 1.2: Aplicación de realidad virtual de BMW

Fuente: zerolight.com[10]

El ámbito de la formación de técnicos especializados en la manipulación de maquinaria industrial es otra de las aplicaciones de la tecnología de Realidad Virtual. Igual que es posible generar entornos donde visualizar las máquinas funcionando de manera real, es posible permitir a los usuarios interactuar con ellas para la resolución de averías o incidencias.
Simplemente con unas gafas de realidad virtual es posible colocar a un operario en un entorno virtual donde se visualice una máquina que ha detenido su producción para que este compruebe su funcionamiento y solucione la incidencia tal y como si se la encontrara en el entorno real de fábrica.
Así la Realidad Virtual se sitúa cercana al ámbito del *training* ofreciendo a las empresas industriales una herramienta mucho más óptima y económica para la formación de sus trabajadores. Pues, gracias a ella, se elimina el coste del técnico que debe hacer la formación a cada trabajador, se optimizan los tiempos y se evita tener que parar la producción para formar a estos trabajadores o permitirles que accedan a las máquinas para practicar su uso.
En este sentido, más idónea incluso, es su aplicación para el entrenamiento en situacio-

nes de riesgo. Una de las empresas que ya ha usado esta tecnología es 3M que, a través de una simulación virtual, muestra a los trabajadores de la construcción los riesgos de trabajar sin las correctas medidas de seguridad. En este caso, al ahorro de costes antes mencionados, se debe añadir la ventaja de evitar poner en riesgo a los trabajadores de forma innecesaria[3].

Figura 1.3: Aplicación de realidad virtual para seguridad de 3M

Fuente: 3m.com[21]

Capítulo 2

Objetivos del proyecto

El objetivo de este proyecto es desarrollar una aplicación que permita la formación de posibles técnicos de mantenimiento de la minifábrica de ICAI y estudiar las posibles ventajas de la aplicación en caso de ser desarrollada en torno a una planta que todavía no ha sido o está en proceso de ser construida. La aplicación situará al usuario en un entorno virtual que simula el laboratorio de sistemas digitales de ICAI. Una vez dentro, se expondrán diferentes averías que el usuario tendrá que identificar basándose en el funcionamiento de la planta. Con el uso de este tipo de simuladores se pretende:

- **Formar técnicos:** El objetivo principal de la aplicación es ayudar en la formación de técnicos, simulando el entorno en el que trabajarán, sin necesidad de parar la planta y sin que los errores cometidos conlleven consecuencia alguna en la planta. La ventaja de la realidad virtual es que es mucho más inmersiva que un simulador en ordenador, y, por lo tanto, cuando el técnico comience a trabajar en la planta real, ese cambio será mucho menos drástico, ya que estará familiarizado con el entorno.

- **Seguridad:** La minifábrica de ICAI consta de varios brazos robóticos. Los motores que controlan estos brazos son muy potentes, y, en caso de avería, un técnico no formado podría verse involucrado en un accidente laboral debido a la falta de experiencia. De este modo, los técnicos novatos pueden realizar las prácticas sin riesgo de lesiones u otros accidentes.

- **Agilizar la puesta en marcha de la planta:** En caso de ser una planta nueva, mediante el uso de la realidad virtual, se puede formar a los operarios mientras que la planta está construyéndose, de modo que cuando esté terminada, los mismos puedan incorporarse inmediatamente, ya que habrán practicado en un entorno virtual, que posee las mismas características y dimensiones que la planta real. Se ahorrarán tiempos muertos, pudiendo comenzar la producción en el menor tiempo.

- **Resolver problemas de diseño:** Al tratar con un espacio virtual, durante la formación de los técnicos se podrá detectar con más facilidad fallos en el diseño de las instalaciones, de modo que antes de empezar a construir el complejo, se puedan hacer los cambios pertinentes.

Capítulo 3

Recursos

Para crear una aplicación de este tipo, se necesitan principalmente dos elementos: el dispositivo de realidad virtual y un motor gráfico. El motor gráfico es el encargado de dar forma a la aplicación y el proyecto en sí, en este caso Unity. Por otro lado, como visor de realidad virtual, se ha escogido Oculus Rift, con el que se podrá visualizar la aplicación como si se tratase de un entorno físico. Para poder coordinar la aplicación y el visor, se ha hecho uso de Steam VR, una aplicación que permite la inclusión de realidad virtual en Unity, y que ofrece diversas herramientas para facilitar la programación de RV en el motor de desarrollo.

3.1. Unity

Unity es un motor de desarrollo para la creación de juegos y contenido 3D interactivo con las características que es completamente integrado y que ofrece innumerables funcionalidades para facilitar el desarrollo de este tipo de aplicaciones. Pertenece a la empresa Unity Technologies, fundada en 2004 en Copenhague, Dinamarca.
Actualmente, es la herramienta más utilizada por los creadores de experiencias XR (Realidad aumentada y virtual). Utilizada por estudios de juegos, agencias creativas como Weiden+Kennedy, la NASA e incluso Google. Es por ello que se ha optado por este software frente a otros similares como Unreal Engine. Otra de las ventajas es la comunidad que ha desarrollado, ofreciendo grandes cantidades de información en foros propios y externos.
Unity se puede usar de forma gratuita en el entorno educativo. El precio del software para uso profesional es de 125$\mes. Posee un plan de suscripción de 25$\mes para aficionados, siendo por esto la preferida de los desarrolladores independientes[17].

Figura 3.1: Logo de Unity

Fuente: Unity Technologies[17]

3.2. Steam VR

Steam VR es una herramienta que permite ejecutar aplicaciones en realidad virtual. Pertenece a la compañía Valve que comenzó su desarrollo en 2014. Comenzó como un modo experimental que soportaba y utilizaba Oculus Rift.
Actualmente es una de las principales plataformas para ejecutar aplicaciones en la realidad virtual, ofreciendo una configuración rápida. Además, posee un SDK de Unity para la creación de experiencias VR. Este kit de desarrollo facilita código, herramientas y modelos de Unity para poder incorporar fácilmente el equipo de VR en la aplicación[16].

Figura 3.2: Logo de Steam VR

Fuente: Valve[16]

3.3. Oculus Rift

El dispositivo de VR utilizado es Oculus Rift. Desarrollado por la empresa Oculus, actualmente pertenece a Facebook. Empezó como un proyecto en *Kickstarter.com* que consiguió recaudar 2.5 millones de dólares[12]. El equipo que se utiliza en este proyecto es la primera versión lanzada dirigida al consumidor(Oculus Rift CV1). Actualmente la compañía ha lanzado dos productos nuevos, el primero, una actualización del Oculus Rift tradicional, y el segundo, Oculus Go, que no requiere de un ordenador. Oculus Rift CV1 se compone de dos pantallas OLED de 1080x1200 cada una(2160x1200 en total), con un ángulo de visión de 110º. El casco incluye unos auriculares. En la caja, además, se incluyen dos mandos (uno para cada mano) y dos sensores. Para poder ejecutarlo se recomienda un ordenador con:

Windows 10
Tarjeta gráfica Nvidia GTX1060/AMD Radeon R9 290 o superior
CPU Intel i5-4590 equivalente o superior
Memoria 8Gb RAM
Salida de vídeo HDMI 1.3
Puertos USB 3x USB 3.0 y 1x USB 2.0[11]

Figura 3.3: Logo de Oculus

Fuente: Oculus[11]

Capítulo 4

Diseño de la escena

En Unity se denomina 'escena' al espacio con el que el usuario va a interactuar. Éste solo puede estar en una escena al mismo tiempo, pudiendo moverse entre diferentes escenas dentro de la misma aplicación. Para este caso sólo se ha hecho uso de una escena, el laboratorio. En la escena se colocarán todos los objetos y componentes que se vayan a ejecutar, y se establecerán las relaciones entre los mismos. En primer lugar, se genera el espacio, al que se le añaden los modelos 3D del laboratorio. A continuación, se les asigna a cada modelo los componentes necesarios y se establecen las relaciones entre los mismos, ya sea mediante código u otros elementos de Unity.

El primer paso para empezar a trabajar en Unity es generar la escena 'física' en la que se va a desarrollar la escena. En este caso, el escenario es el Laboratorio de Sistemas Digitales. Para importar modelos, en primer lugar, hay que arrastrar los objetos a la ventana de la escena. Una vez arrastrado el modelo a la escena, este se convierte en un *Game Object* (Objeto de juego). Los *Game Objects* son instancias que pueden contener diferentes componentes. Dependiendo de los componentes que se le agreguen se comportará de una forma u otra. Estos componentes se pueden ver en la ventana 'inspector' a la derecha. Los *Game Objects* importados poseen todos el componente *Transform*, que indica la posición y la escala del objeto en la escena. Además, se pueden añadir otros componentes como *colliders* (permite interactuar físicamente con otros *Game Objects*), animaciones, scripts, materiales y *renderers* entre otros. (ver Figura 4.1).

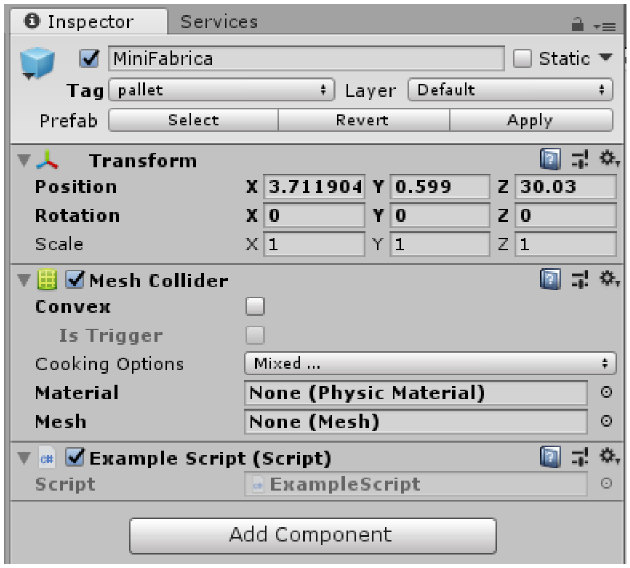

Figura 4.1: Ventana del inspector de Unity.

En este capítulo se ofrece una descripción más detallada de los modelos principales. Otros modelos como las mesas, setas de emergencia y puertas no aparecen desarrollados, ya que solo funcionan como objetos que ayudan a la inmersión y no poseen función específica dentro del programa, conteniendo su instancia de *Game Object* los componentes básicos para que aparezcan en escena.

4.1. Planta

El diseño y modelado de la planta se ha extraído del trabajo fin de máster de Carlos Álvarez Vereterra[7], que importó el modelo a su vez de otros trabajos fin de grado y máster. El modelo original estaba creado en AutoCAD, éste utiliza un formato de archivo con la extensión '.dwg' (DrawWinG, o plano) para almacenar sus archivos 3D. Este formato desarrollado por Autodesk no es compatible con Unity, como se puede comprobar en la documentación oficial. Esto implica que no se puede soportar directamente, sino que será necesario convertirlo a otra extensión. Entre los formatos compatibles se encuentra '.fbx' un formato desarrollado también por Autodesk. AutoCAD permite exportar directamente a este formato, así que esta fue la primera opción. Sin embargo, el modelo en '.fbx' visto a través de la realidad virtual genera sensación de mareo, impidiendo que se pueda usar el sistema de RV por más de unos segundos. Visto el problema de rendimiento, es necesario buscar una alternativa.

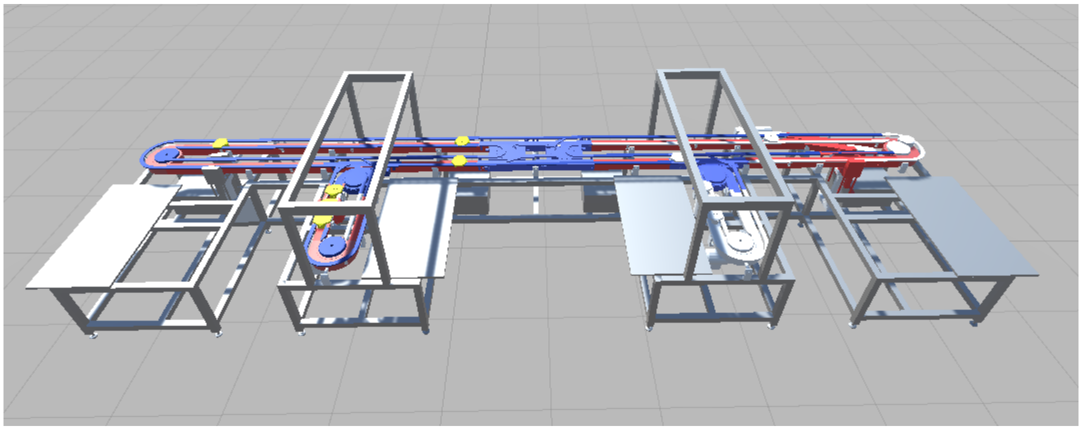

Figura 4.2: Modelo de la planta en Unity.

Este problema de rendimiento se debe a que AutoCAD no es un software de modelado 3D para videojuegos, sino un software CAD para aplicaciones de diseño industrial donde la precisión y la fidelidad son cruciales, sobre todo en campos como el prototipado rápido o impresión 3D.
Una alternativa de Autodesk, la misma compañía que desarrolla AutoCAD, es utilizar 3DS Max, un programa especializado en modelado y animación para videojuegos.

3DS Max no es un programa gratuito, sino que tiene un modelo de negocio de suscripción. Sin embargo, para esta aplicación se ha utilizado la versión de prueba de 30 días. Al estar desarrollado por Autodesk es compatible con archivos '.dwg', por lo que no será necesario realizar ninguna conversión para abrirlo[8].
Para este proyecto no se han realizado más modificaciones más allá de los materiales de los modelos, para los cuales se han utilizado colores básicos para poder diferenciar las diferentes partes de la planta. A continuación, se ofrece una breve descripción de los componentes de la planta:

- **Transformar:** Indica posición y escala de la planta en la escena. En este caso posición (4.42,0.0750,-3.352) y escala (1,1,1). La escala se ha dejado intacta, porque la planta estaba diseñada con las medidas reales, sirviendo de referencia para ajustar la escala de los demás objetos de la escena.

- **Scripts:** En este caso tres scripts: *UIElement, Interactable, Button.*

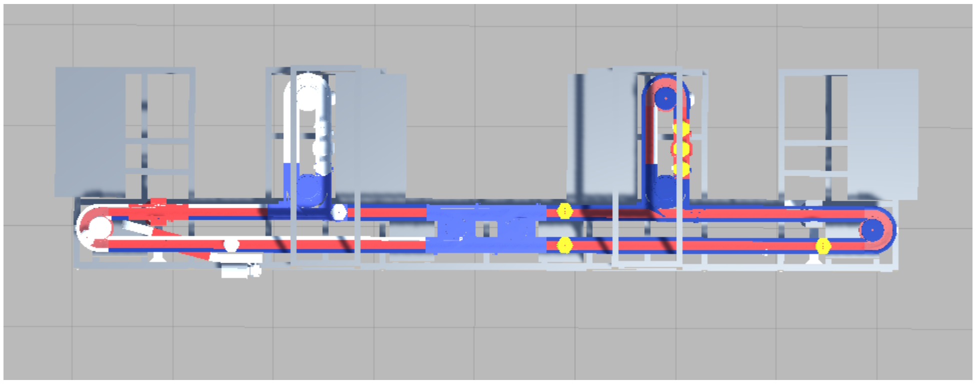

Figura 4.3: Vista en planta del modelo de la planta en Unity.

4.1.1. Robot ABB IRB 120

Al igual que la planta, el modelo del robot ha sido importado del trabajo fin de máster de Carlos Álvarez Vereterra[7]. La importación del modelo 3D del robot ABB IRB 120 es bastante sencilla. ABB proporciona el modelo 3D en su página web[2], en varios formatos. Utilizando el formato *Google Sketchup*, Unity es capaz de reconocer el modelo 3D directamente, sin necesidad de convertirlo a otro formato. El único problema encontrado con ese modelo es la escala. En la ventada 2D del editor de Unity, es muy complicado tener una sensación de escala realista. Al visualizar el modelo en 3D utilizando el Oculus Rift por primera vez, éste medía más de 100 metros de alto. En las opciones de importación del modelo de Unity , es necesario hacer la conversión de pulgadas a metros, haciendo los siguientes cambios:

- Unit conversion: 1 Meters = 0,03937008.

- Scale factor: 0; 001.

- Use file Scale: activado. Una vez hechas estas correcciones, el modelo ya estaría a escala 1:1, lo cual es fácil de comprobar utilizando un HMD de realidad virtual como el Oculus Rift[9].

Figura 4.4: Modelo robot ABB IRB 120

4.1.2. Sensor movimiento y gatillo

Para este proyecto se modelaron dos objetos específicos, el sensor de movimiento y el gatillo, que se encuentran en la planta real (Ver 4.5) El primero se encarga de detectar cuándo un pallet se encuentra cerca del mismo, y el segundo es un dispositivo que al activarse bloquea el movimiento del pallet. Estos objetos han sido modelados con formas primitivas de Unity.

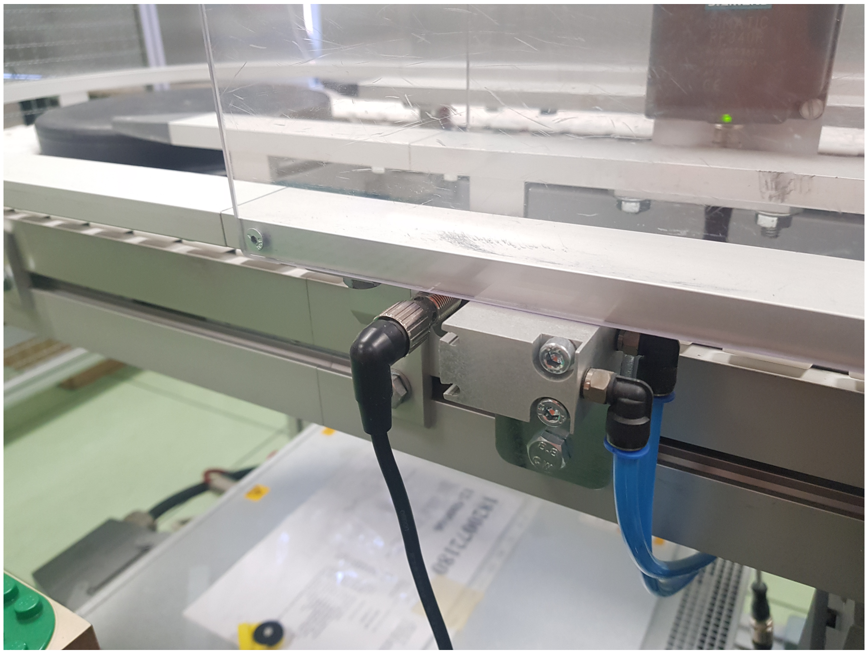

Figura 4.5: Sensor de movimiento (izquierda) y gatillo (derecha).

El sensor de movimiento está formado por una cápsula (cuerpo del sensor), una esfera (dispositivo del LED), una luz de punto (luz del LED) y un objeto sin render(invisible), que es el que se encarga de detectar si un pallet ha pasado por el punto especificado. Ver Figura 4.6.

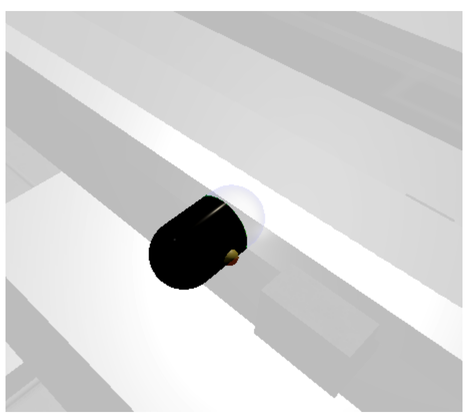

Figura 4.6: Modelo en Unity del sensor de movimiento.

El gatillo se ha modelado con dos cubos. Uno que hace de la parte visible del dispositivo, y otro que hace de gatillo para bloquear el pallet. Al activarse, el gatillo se desplaza, de modo que el componente *Collider* del pallet, choca contra el *Collider* del gatillo, impidiendo que pueda seguir desplazándose. Ver Figura 4.7

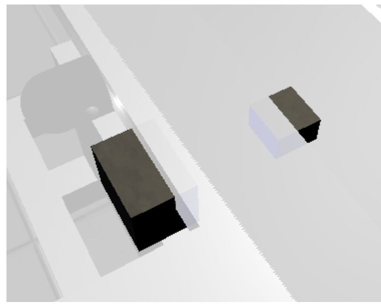

Figura 4.7: Modelo en Unity del gatillo.

Además, se ha añadido un modelo de las válvulas de aire situadas en la parte inferior, con un LED que se ilumina indicando qué válvula está funcionando, y cuando está funcionando.Ver Figura 4.8.

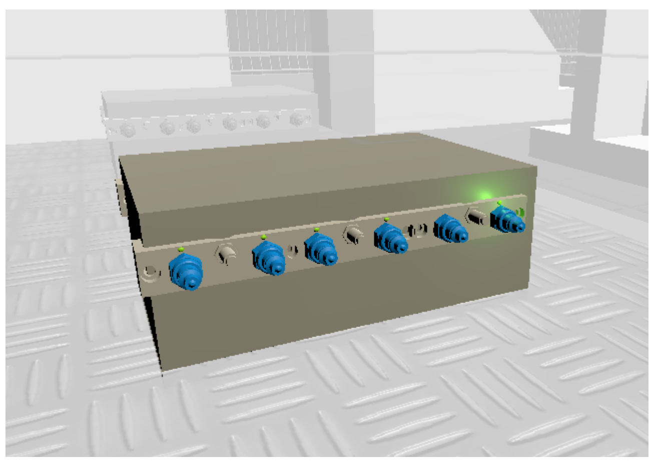

Figura 4.8: Modelo de las válvulas con LED encendido.

4.2. Laboratorio

El laboratorio se ha intentado modelar de la manera más fiel a la realidad. Para ello se tomaron medidas aproximadas de las dimensiones del laboratorio, y se ajustaron según la escala de la planta. El objetivo de los simuladores no es dar una representación exacta del entorno, sino ofrecer la sensación al usuario de que se encuentra en ese espacio, por ello se tomó cierta libertad en las medidas.
La estructura del laboratorio (paredes y suelo) está modelada con formas básicas ofrecidas por Unity, en concreto planos y cubos. La ventaja de estas formas es que, al ser simples, contienen un número limitado de polígonos, lo que facilita el *render* y por lo

tanto mejora el rendimiento de la aplicación, ofreciendo mayor número de fotogramas por segundo. Es por esto, que hay que controlar el detalle y la complejidad de los objetos que se importan en la escena, ya que la potencia de la tarjeta gráfica del ordenador es limitada. Si se quiere mayor nivel de detalle en los modelos, se requerirá de una tarjeta gráfica más potente.
El plano del suelo y las paredes están compuestas del ya comentado componente Transformar, así como de *Colliders* y *Renderers*. *Collider* es en este caso, hace que los objetos no puedan atravesarse. De este modo, el jugador no puede salir del espacio delimitado. Además, en el caso del suelo, hace que los objetos no se caigan al vacío. El componente *Renderer* aparece generalmente en todos los objetos de la escena, permite que el objeto en sí aparezca en pantalla. Ver Figura 4.9.

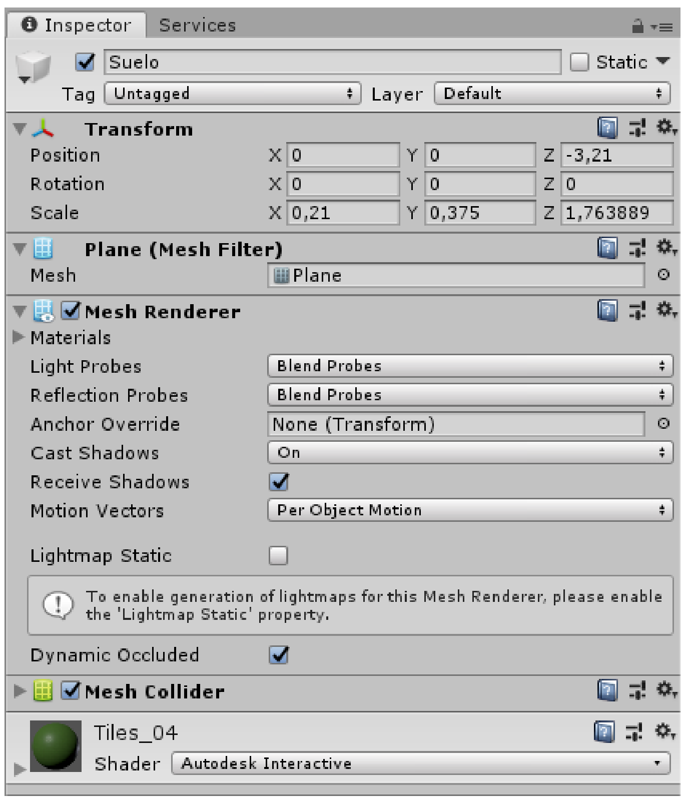

Figura 4.9: Ventana inspector del plano del suelo.

Los otros objetos como mesas de laboratorio, ordenadores, ventanas, rejas y demás, han sido importados de *3D Warehouse*[1]. Éste es un catálogo online, perteneciente a SkecthUp (Google), en el que los usuarios suben modelos creados por ellos. Todos los autores de los modelos utilizados están citados en el apéndice A.Para posicionar los diferentes objetos se ha hecho uso de nuevo del componente *Transformar*, alterando la posición y, en algunos casos, la escala, para ajustarse lo mejor posible a la realidad.

4.2.1. Luces

En Unity, normalmente se ilumina la escena con una luz direccional, diferentes luces de punto y de foco. Las luces direccionales son aquellas que emiten con rayos paralelos y uniformemente distribuidos, iluminando todo de forma homogénea (como el Sol). Las luces de punto son aquellas en las que todos los rayos de luz se originan desde un mismo punto (como una bombilla). Por otro lado en las luces de foco los rayos se originan desde un punto, pero se proyectan en forma de cono. [5].
Para este proyecto se ha utilizado una luz direccional y un foco. El motivo de utilizar tan pocas luces es el rendimiento de la tarjeta gráfica. Según la documentación de Unity[18] cuantas más luces haya, mayor será la reducción de fotogramas por segundo (FPS). Al usar el visor de realidad virtual, se experimenta un retraso en la actualización de las imágenes, resultando en fragmentos negros o imágenes deformadas que causan mareo. Una de las posibles soluciones sería modificar la forma en que se iluminan los objetos de la escena, pero esto está fuera del alcance de este trabajo.

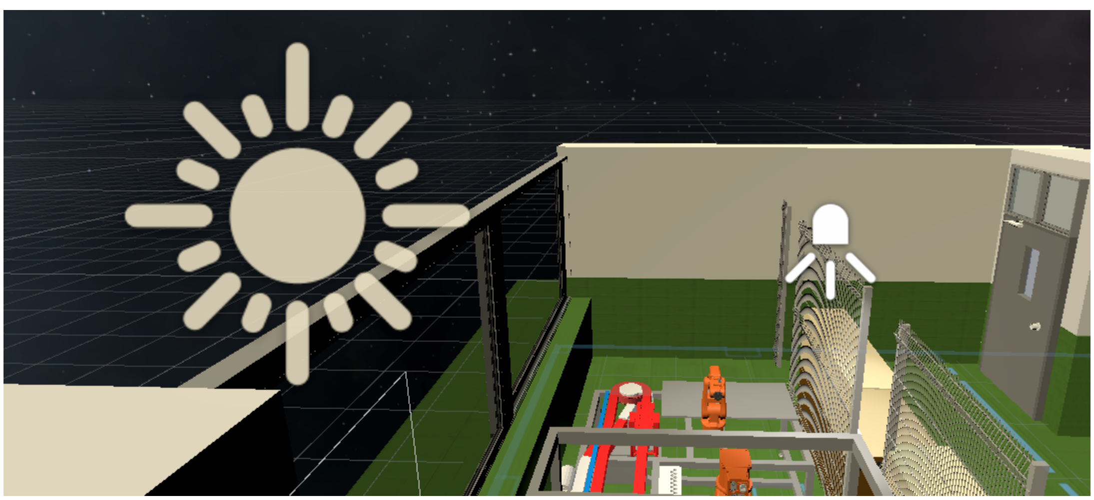

Figura 4.10: Vista de los gizmos de las luces.

Al principio se utilizó un techo en el modelo del laboratorio, que impedía que la luz direccional entrase, por lo que se tuvieron que añadir numerosas luces interiores de punto y foco. Al añadir estas luces se experimentó el problema comentado anteriormente. La solución final fue quitar el techo del laboratorio, y aprovechar la luz direccional, además de añadir otra luz para iluminar la pared izquierda del laboratorio, que quedaba en penumbra. De este modo se consiguió una iluminación correcta sin comprometer el número de FPS.

4.2.2. Interfaz de contadores

En una de las columnas se ha añadido un objeto de Unity tipo *canvas*. Este objeto permite representar texto en la escena. Este *canvas* muestra al usuario el número de aciertos, el número de fallos y el tiempo que queda hasta la siguiente avería(Figura 4.11). Este tiempo se puede modificar en el script FailureManager.cs(9.4.3). Si el tiempo está a 0, significa que la planta está en fallo. El texto se actualiza cada fotograma, ya que está asociado a FailureManager.cs. Se explica con más detalles en la sección 5.1.

Figura 4.11: Interfaz de usuario para recuento de aciertos, fallos y temporizador.

Capítulo 5

Diseño del código

En este capítulo se explica cómo se ha diseñado el código. Unity hace uso de programación orientada a objetos. El funcionamiento de la aplicación gira en torno al gestor de fallos. Éste se encarga de coordinar los demás elementos de la escena, específicamente, los dispositivos que son susceptibles de avería. Éstos últimos funcionan cada uno de forma independiente, pero supervisados por el gestor. Los objetos se comunican directamente a través de 'mensajes' que llegan al gestor de fallos, y éste responde según la situación.

5.1. Gestor de fallos

Figura 5.1: Modelo que representa el gestor de fallos en la escena.

El gestor de fallos es el encargado de coordinar todos los eventos que se dan en la escena. Se ha designado uno de los ordenadores como el *Game Object* que representa el gestor de fallos(Ver Figura 5.1). Éste contiene el código FailureManager.cs (9.4.3). Este código contiene campos públicos modificables en los que añadir el número de dispositivos susceptibles de fallos así como campos para cada uno de esos objetos(Ver Figura 5.2).

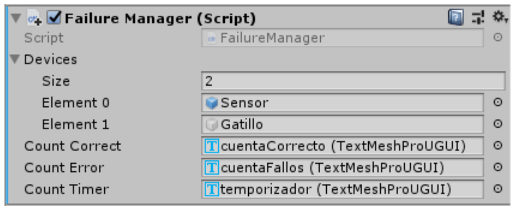

Figura 5.2: Ventana inspector del modelo del gestor de fallos.

1. Variables:

   - devices[]: Vector de *Game Objects* en el que se almacenan los objetos susceptibles de avería. Variable pública que se puede modificar directamente desde Unity (tamaño y *Game Objects* que contiene).
   - n: Índice de devices[] de la avería actual.
   - failure_ id: Almacena el ID de la avería actual.
   - object_ id: ID del objeto que ha sido pulsado.
   - OnFailure: Booleano que indica si la planta está actualmente en estado de avería o funcionando correctamente.
   - TimeLeft: Tiempo hasta que se produzca el siguiente fallo en la planta.
   - c_ correct: Contador de número de aciertos.
   - c_ error: Contador de número de errores.
   - countCorrect: Variable del tipo *TextMeshPro(TMP)*, convierte el número de errores en texto *TMP*, para ser mostrado en la escena.
   - countError: Igual que countCorrect, pero cuenta errores.
   - countTimer: igual que countCorrect, pero muestra el tiempo restante hasta siguiente avería.

2. Funciones:

   - Update(): Función que se llama cada fotograma. Se encarga de actualizar las variables del tipo *TMP*.
   - FailureTimer(): Función temporizador que realiza la cuenta atrás hasta la siguiente avería.
   - FailureSet(): Se encarga de seleccionar aleatoriamente una de las averías especificadas en devices[], y envía al elemento elegido el mensaje para activar su función de avería.
   - OptionCheck(int id): Recibe el ID de un objeto pulsado por el usuario y lo compara con el ID de la avería actual. En caso de ser el mismo, envía el mensaje correspondiente al objeto averiado para que vuelva a su estado normal. En caso contrario llama a OptionFalse().
   - OptionFalse(): Suma un error.
   - StoreFailureID(int id): Recibe el ID de la avería actual y lo almacena en failure_ id.

5.2. Dispositivos susceptibles de fallo

El sensor de movimiento y el gatillo son los elementos de este proyecto susceptibles de avería. Ambos se han codificado utilizando la plantilla incluída en la sección 5.5. En esta sección se explica el código referente exclusivamente a la función de avería de los dispositivos. Para ver las otras funciones, hay que dirigirse a la sección 5.5.

5.2.1. Sensor de movimiento

El sensor de movimiento es un dispositivo situado al lado de la cinta (Ver figura 4.6), que detecta cuando un pallet está adyacente. Cuando pasa un pallet, el LED del sensor se ilumina. En ocasiones el sensor no responde bien y no detecta el paso del pallet. Para identificar esta avería hay que fijarse en dicha luz. Si el sensor se encuentra fallando, el LED no se iluminará al paso del pallet. El código de este dispositivo, MovementSensor.cs (9.4.4) se encuentra en el *Game Object* '*sensor*' que pertenece a su vez al *Game Object* '*movement sensor*'.

1. Variables:
   - myLight: Luz del sensor
2. Funciones:
   - Start(): Función que se llama al inicio de la escena. Apaga la luz del LED.
   - Update(): Se llama en cada fotograma. Esta función hace que el objeto del sensor identifique qué clase de objeto está pasando al lado. Si es un pallet el sensor de iluminará en caso de funcionar normalmente. En caso de avería, el LED no se encenderá.
   - LightOff(): Apaga la luz.
   - LigthOn(): Enciende la luz.

5.2.2. Gatillo

Este dispositivo se encarga de bloquear el paso del pallet en un punto de la cinta transportadora(Figura 4.7). Funciona a través de una válvula de aire comprimido. A veces el gatillo se atasca, por lo que a pesar de estar la válvula en funcionamiento el gatillo no se acciona. Para identificar esta avería hay que activar el gatillo. En la planta real, el gatillo se acciona mediante código. Como no se puede meter el mismo código en Unity, se ha añadido un interruptor manual dentro de la planta virtual que acciona el gatillo como si se hubiese accionado por código. Si se acciona el interruptor y el pallet no se para en el lugar indicado, significa que está fallando. Además, se ha incluido un LED en la válvula(Figura 4.8), que se encuentra debajo de las cintas transportadoras, que también indica cuando debería estar actuando el gatillo(encendido) y cuando no (apagado). El archivo de código es StoppingDevice.cs (9.4.8).

1. Variables:

- belt: Cinta sobre la que está situado el gatillo. Se especifica en el editor de Unity.
- myLight: Luz situada en la válvula (Figura 4.8)
- button: Booleano que indica si el interruptor ha sido accionado o no.
- active: Booleano que indica si el gatillo está bloqueando la cinta.

2. Funciones:

- Start(): Función que se llama al inicio de la escena. Apaga la luz de la válvula.
- Update(): Se llama en cada fotograma. Controla el funcionamiento del gatillo. Según el gatillo esté activo o no, modificará la posición del mismo para bloquear la cinta, siempre y cuando no se encuentre en estado de avería.
- OnCollisionEnter() y OnCollisionExit(): Estas funciones son auxiliares, evitan movimientos indeseados del pallet cuando los *colliders* de los objetos están en contacto.
- DeviceActivated(): Activa el gatillo y enciende la luz de la válvula.
- DeviceOff(): Desactiva el gatillo y apaga la luz de la válvula.

5.3. Objetos pulsables activos y pasivos

Para poder interactuar con los objetos desde la realidad virtual es necesario incluir diferentes scripts. Se han diferenciado dos tipos de objetos pulsables: los activos y los pasivos. Los activos son aquellos que tienen algún tipo de función, en este caso son el sensor de movimiento, el gatillo y el interruptor del gatillo. El resto de objetos que se pueden pulsar son pasivos, como los robots o la planta.
Para que se pueda interactuar con los objetos se necesitan tres scripts, proporcionados por Steam VR[16]: Interactable.cs (9.5.7), Button.cs(9.4.10) y UIElement.cs(9.5.20). Basta con arrastrar los scripts al *Game Object* correspondiente. Hay que tener en cuenta que el *Game Object* debe tener un componente *Collider*, que delimitará la zona a la que hay que acercar la mano para que se pueda pulsar el objeto. Finalmente, en el código UIElement.cs (9.5.20) hay que especificar qué función activa el objeto cuando es pulsado. Para ello, este mismo código ofrece unos campos en los que podemos especificar la función (pública) del *Game Object* (no tiene por qué ser el mismo) que se ha de llamar(Figura 5.3).

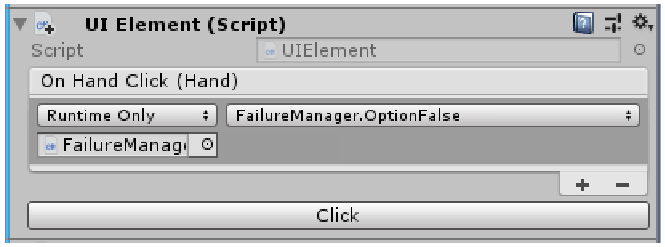

Figura 5.3: Vista del código UIElement.cs en el inspector.

Con estos códigos se crea una interfaz, que al acercar la mano sobre un objeto, el gatillo del mando y el objeto se resaltan en amarillo, de forma que el usuario sepa qué objeto está enfocando y qué botón pulsar. Ver Figura 5.4.

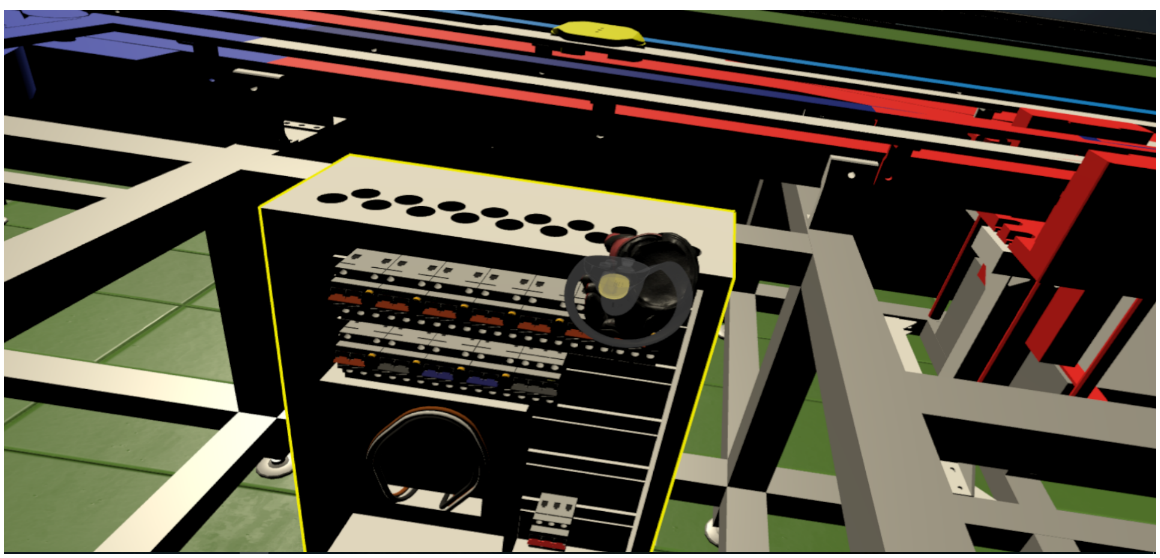

Figura 5.4: Ventana inspector del modelo del gestor de fallos.

Los scripts UIElement.cs del gatillo y el sensor están configurados para llamar a la función CheckOption() de FailureManager.cs(9.4.3). Los demás objetos llaman directamente a la función OptionFalse(). En el caso del interruptor del gatillo, éste llama a la función ButtonSwicth() de SDSwitch.cs(9.4.7), para activar/desactivar el dispositivo.

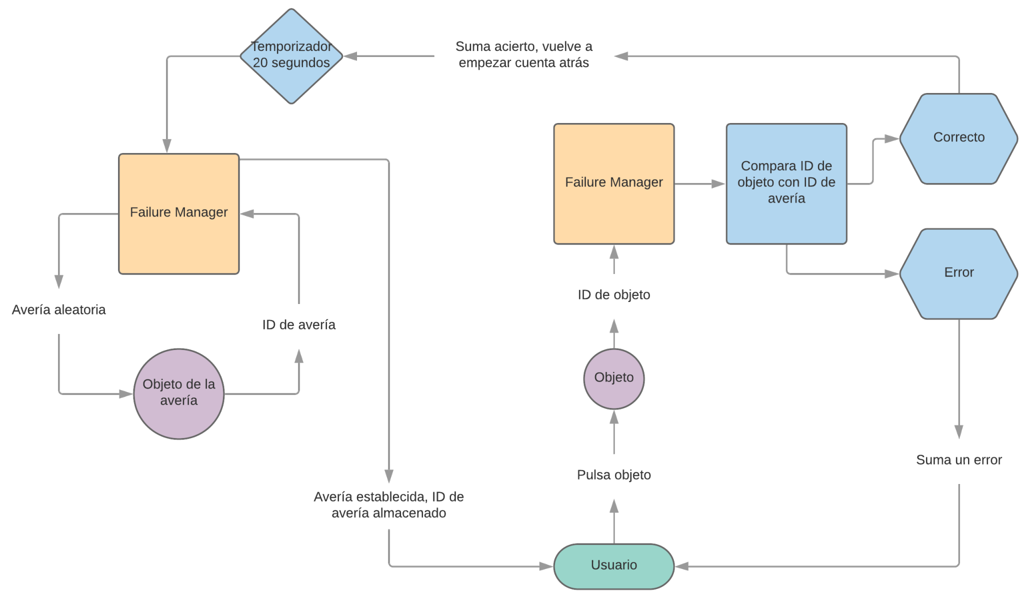

Figura 5.5: Diagrama resumen de las comunicaciones entre objetos.

5.4. Movimiento de las cintas

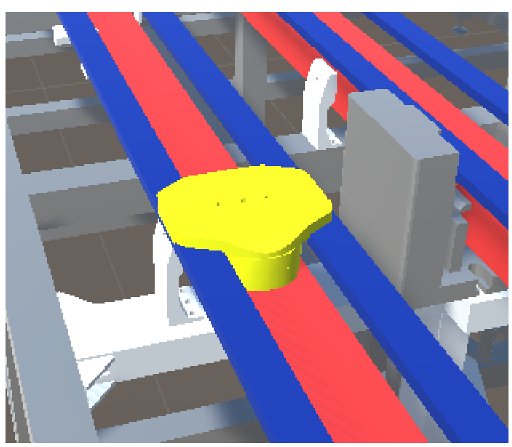

Figura 5.6: Pallet colocado en las cintas en Unity.

Para mover el pallet se hace uso de dos scripts para cada tipo de cinta. Para la cinta recta, se utiliza Belt.cs(9.4.1):

1. Variables:

   - endpoint: Punto hacia donde se transporta el objeto.

- speed: Velocidad de la cinta.
- active: Booleano que indica si está activa la cinta o no.

2. Funciones:

   - OnTriggerStay(Collider other): Al entrar en contacto con el collider de otro objeto, mueve dicho objeto hacia delante.
   - StopMovement(): Para la cinta.
   - IniMovement(): Inicia movimiento de la cinta.

Por otro lado, para la cinta curva, se han utilizado las ecuaciones de la circunferencia. En este script, CircularBelt.cs(9.4.2) se han declarado variables públicas como el centro de giro y el radio, para poder ser modificadas desde el editor de Unity y ajustadas a la forma de la cinta de la planta. El script permite un movimiento de rotación y traslación, aunque el de rotación se ha desactivado para este proyecto, por motivos de coordinación el otro movimiento y con la forma de la cinta. En caso de querer activarlo, simplemente hace falta establecer una velocidad de rotación desde el editor de Unity. El script contiene:

1. Variables:

   - center: Centro de giro.
   - radius: Radio de giro.
   - traslationSpeed: Velocidad de traslación.
   - posX: Posición en plano X del objeto a transportar.
   - posZ: Posición en plano Z del objeto a transportar.
   - angle: Ángulo de inicio.
   - ini_ angle: Ángulo de reinicio.
   - rotationSpeed: Velocidad de rotación.

2. Funciones:

   - OnTriggerStay(Collider other): Al entrar en contacto con el collider de otro objeto, mueve dicho objeto en una trayectoria circular de rotación y traslación.
   - OnCollisionExit(): Resetea el ángulo cuando el pallet sale de la cinta transportadora.

Al ejecutarse estos scripts, el pallet, debido a la posición de los Colliders de las cintas, acumulaba un pequeño error en cada vuelta que hacía que se cayese. Para corregir este error, se añadió el código Pallet.cs(9.4.5) levanta el pallet de forma imperceptible en cada vuelta, de modo que cada vez que desciende cierta altura, se corrige su posición vertical a la que tenía inicialmente.

5.5. Añadir nuevos fallos

Debido al carácter de continuidad de este proyecto, se decidió añadir una plantilla de código para poder añadir nuevas averías, de modo que funcionen con el código actual de FailureManager.cs(5.1). La plantilla se encuentra en la sección 9.4.6. Esta plantilla contiene:

1. Variables:
   - failure_ manager: GameObject público. Hay que seleccionar el Game Object que contiene el código FailureManager.cs. Ver Figura 5.7.
   - id: ID del fallo. Es el ID que se enviará a FailureManager.cs para que lo almacene/compare.
   - fail: Booleano que indica si el dispositivo está fallando o no.
2. Funciones:
   - SensorFailure(): Activa la avería y llama a FailIDSet().
   - FailureDetected(): Desactiva el fallo.
   - FailIDSet(): Envía el ID del objeto a FailureManager.cs para que lo almacene.
   - SendID(): En caso de ser pulsado, envía el ID a FailureManager.cs para que compare con la avería actual.

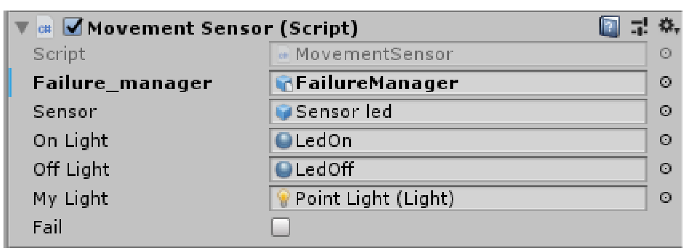

Figura 5.7: Vista del script de la plantilla en el inspector.

Las funciones anteriores están diseñadas para funcionar con FailureManager.cs, es importante no cambiar el nombre de las mismas, ya que al usar la función de unity 'SendMessage', cualquier cambio en el nombre, haría que se perdiera el mensaje entre los objetos. El código correspondiente a la avería tendrá que funcionar conforme a la variable '*fail*'. De modo que si es '*true*' el dispositivo esté fallando. También hay que asegurarse de que el ID no coincide con ninguno de los otros(Ver cuadro 6.1), lo que podría dar lugar a falsos aciertos.
Una vez se tenga diseñado el objeto, hay que añadir los scripts mencionados en la sección 5.3. En UIElement se seleccionará la función propia del objeto SendID()(Ver Figura 5.8). Finalmente, hay que añadir en los campos del inspector de FailureManager.cs en el *Game Object* 'Failure Manager', el nuevo *Game Object*, como se puede ver en la figura 5.2.

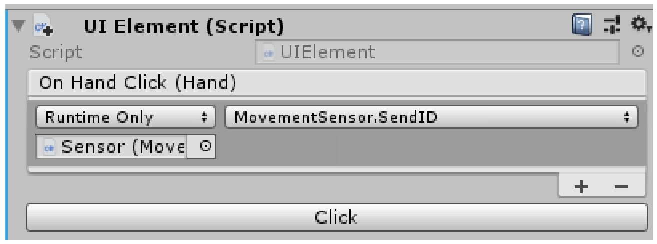

Figura 5.8: Vista del script de UIElement.cs en el inspector.

Dispositivo	GameObject	ID
Sensor movimiento	Sensor	101
Gatillo	Gatillo	102
Nuevo dispositivo	Nuevo GameObject	XXX

Cuadro 5.1: Tabla de objetos susceptibles de fallo.

Listing 5.1: Script plantilla para añadir nuevos dispositivos susceptibles de falllo

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

public class PlantillaFallo : MonoBehaviour
{

    //Gestor de fallos.
    [SerializeField]
    public GameObject failure_manager;

    //ID del fallo.
    int id = 000;


    //Fallo en el sensor
    public bool fail = false;


    // Start is called before the first frame update
    void Start()
    {

    }

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

    }
```

```


    //Funcion: SensorFailure()
    //Descripcion: Activa la averia y envía el ID a Failure Manager para almacenarlo.
    void SensorFailure()
    {

        fail = true;
        FailIDSet();
    }

    //Funcion: Failure detected()
    //Descripcion: Desactiva la averia.
    void FailureDetected()
    {
        //Desactiva el fallo
        fail = false;
    }

    //Función: FailIDSet()
    //Descripción: Envía el ID del fallo a Failure Manager para que lo almacene.
    public void FailIDSet()
    {
        failure_manager.SendMessage("StoreFailureID", id);
    }

    //Función: SendID()
    //Descripción: Envía el id del fallo a Failure Manager para comprobar si es el correcto.
    public void SendID()
    {
        failure_manager.SendMessage("OptionCheck", id);
    }

}
```

Capítulo 6

Implementación en RV

La implementación de la RV en Unity se ha realizado a través de SteamVR. En un principio se comenzó utilizando el SDK propio de Oculus para Unity, pero presentaba problemas de compatibilidad entre actualizaciones de Unity y el propio *SDK*. Adicionalmente, SteamVR ofrece un mayor número de herramientas y permite la compatibilidad con otros dispositivos a parte de Oculus Rift, como el HTC Vive.

Figura 6.1: Representación de las manos y los mandos dentro de la aplicación.

6.1. Player

Player es el nombre del *Game Object* que representa al jugador en la escena. Es un 'prefab' de SteamVR que incluye todos los scripts necesarios para una implementación básica de la RV en Unity. Éste *Game Object* está compuesto a su vez de otros *Game*

Objects entre los que se incluyen las manos, el cuerpo y la cabeza(cámara).
Cada uno de los *Game Objects* contiene uno o varios scripts de SteamVR que en conjunto, compatibilizan el hardware y el software. De este modo los sensores detectan el visor de realidad virtual y los mandos, y a través de SteamVR lo llevan a Unity. Los scripts utilizados de SteamVR están recogidos en la sección 9.5.

6.2. Controles

Oculus Rift incluye dos mandos, uno para cada mano(Figura 6.2). Dentro de la escena, aparecen representados los mandos en sí, además de las manos del usuario, que imitan los movimientos de las manos reales, para facilitar el uso de los controles(Figura 6.1). Para esta aplicación se han configurado los mandos de forma simétrica, de modo que el botón en uno de los mandos realiza la misma acción que el botón correspondiente en el otro mando. Los controles en esta aplicación tienen dos funciones principales: movimiento y selección de objetos.

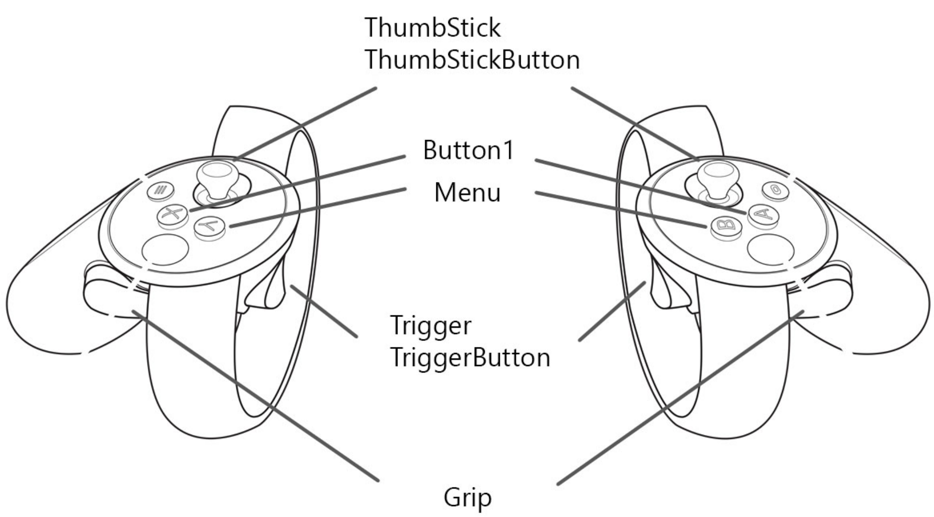

Figura 6.2: Controles Oculus Rift

Fuente: Documentación Unity[18]

Para moverse dentro del laboratorio virtual, se ha incluido un sistema de 'teletransporte'. Este sistema funciona a través de scripts de SteamVR. Es una forma fácil y segura de desplazarse en el entorno virtual, permitiendo avanzar grandes distancias sin moverse del sitio. Se ha optado por este sistema debido principalmente al límite de espacio que hay en el laboratorio, además de la longitud del cable del visor. Otra forma de movimiento que se probó fue el tradicional por *joysticks* utilizados en videojuegos

FPS(*First Person Shooter*). El principal problema que presenta este tipo de movimiento, es que, en realidad virtual, acaba resultando en mareos y pérdidas de equilibrio, quedando por lo tanto descartado.
Para activar el sistema de teletransporte, hay que pulsar cualquiera de los dos *joysticks*. Una vez pulsado aparecerá un haz parabólico desde el mando, y en el otro extremo un círculo iluminado(Figura 6.3). Al dejar de pulsar el *joysctick*, el usuario se moverá al punto marcado por el círculo.

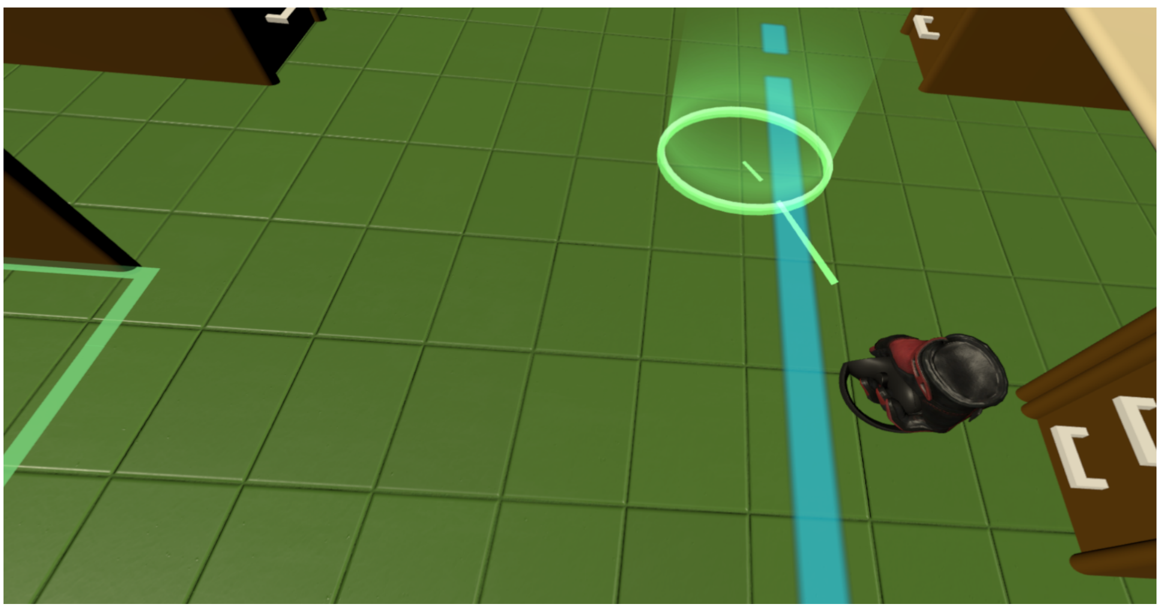

Figura 6.3: Sistema de teletransporte.

Para implementar este código, se tuvieron que agregar los scripts correpondientes al *Game Object* del jugador. Además, se tuvo que designar el área en el que el usuario puede moverse, mediante TeleportArea.cs(9.5.17). Para ello se asigna el script a un plano superpuesto con el plano del suelo del laboratorio(Figura 6.4).

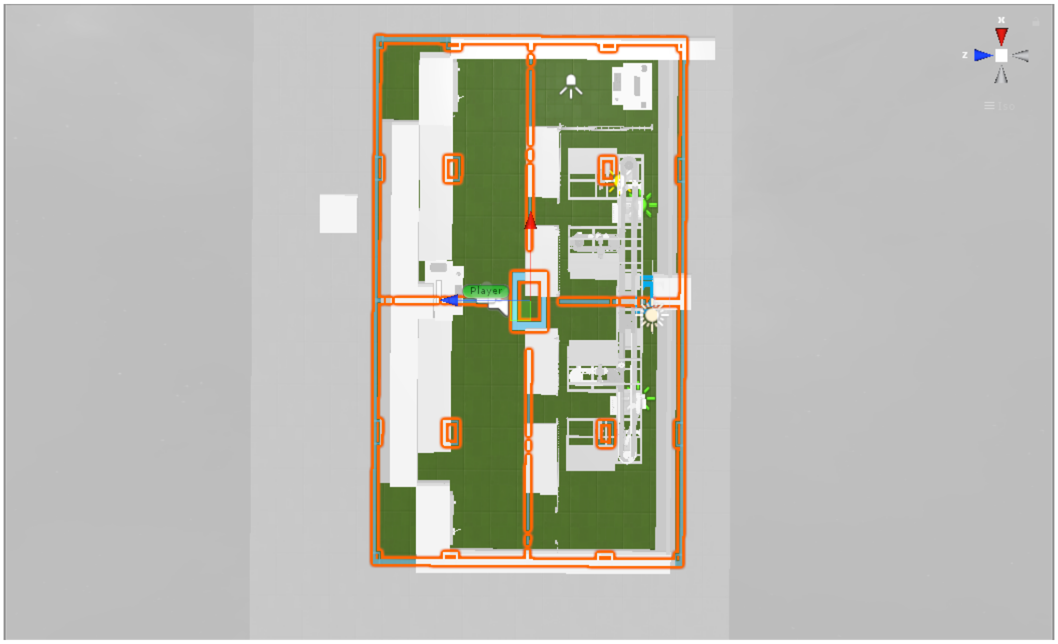

Figura 6.4: Área de teletransporte ajustada al tamaño del laboratorio.

La selección de objetos permite al usuario interactuar con la aplicación, pudiendo seleccionar el objeto que crea que está averiado(Ver sección 5.3). Para seleccionar un objeto el usuario debe acercar la mano al objeto en cuestión, éste se resaltará en color amarillo. Para seleccionar ese objeto hay que pulsar el gatillo/*trigger* del mando con el que se está tocando el objeto deseado.

Capítulo 7

Puesta en marcha de Oculus Rift

1. Conectar el visor y los sensores al ordenador: Es recomendable conectar los USB del visor y los sensores a puertos USB 3.0 para disminuir la latencia. El HDMI del visor se ha de conectar al HDMI de la tarjeta gráfica, en caso contrario,no funcionará correctamente. Una vez conectados, se abrirá la aplicación de Oculus en el ordenador, si no se ha abierto, hay que abrirla manualmente. La aplicación hará de guía para continuar configurando el dispositivo.

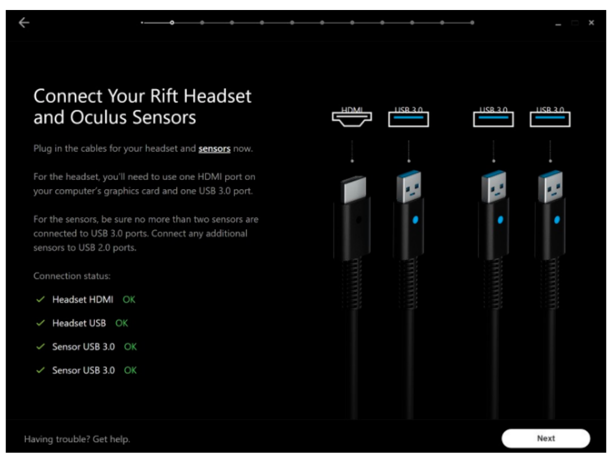

Figura 7.1: Primer paso.

2. Conectar mandos inalámbricos: Para encender los mandos hay que pulsar durante unos segundos el botón 'B' y 'oculus' a la vez. Hacer lo mismo con los botones correspondientes en el otro mando.

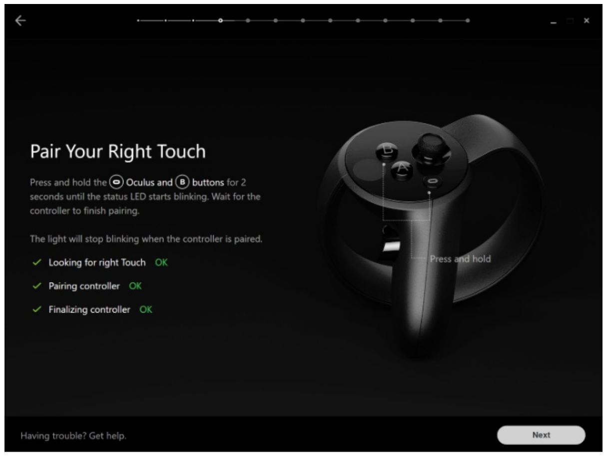

Figura 7.2: Segundo paso.

3. Posicionar los sensores: Hay que colocar los sensores de modo que cubran todo el área que se va a usar. La distancia entre los sensores recomendada es de 1 a 1,5 metros. Una vez colocados, seguimos los pasos de la pantalla, que verificará si se cubre el área mínima, y nos dirá si hay que modificar la posición de alguno de los sensores.

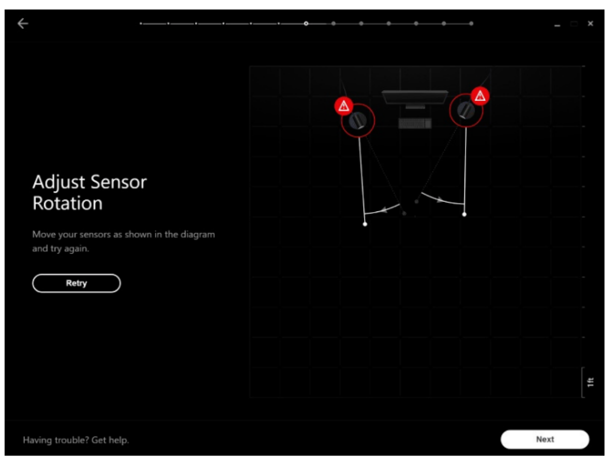

Figura 7.3: Tercer paso.

4. Delimitar el área de juego: En este paso hay que 'pintar' tal como indica la pantalla, el área que vayamos a utilizar. Este paso es importante, ya que colocará una malla virtual, que aparecerá cuando estemos muy próximos a ella, para evitar golpear a otras personas u objetos.

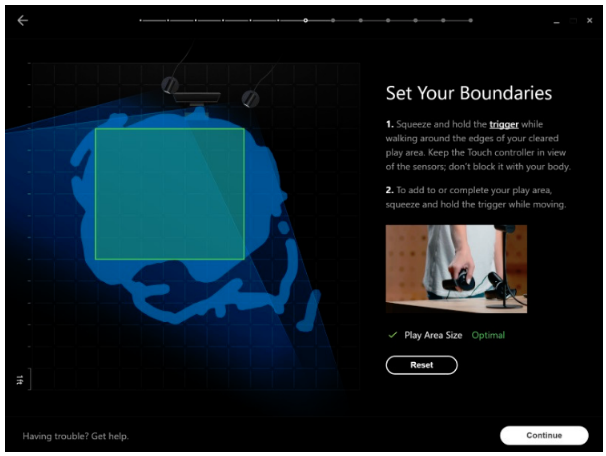

Figura 7.4: Cuarto paso.

5. Ajustar el visor a la cabeza y la distancia de las lentes.

6. Finalmente, hay que abrir SteamVR si no se ha abierto ya, y asegurarse de que todos los elementos han sido vinculados (aparecen en una ventana pequeña en la parte derecha inferior de la pantalla). Una vez hecho esto, ya se puede iniciar la aplicación .exe del proyecto.

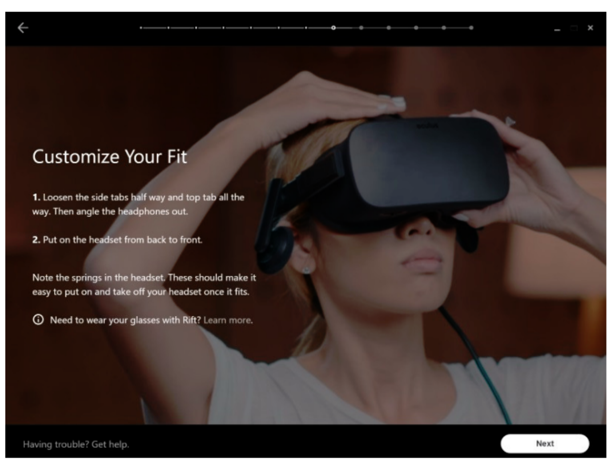

Figura 7.5: Quinto paso.

Capítulo 8

Conclusiones y desarrollos futuros

Unity es un motor centrado en la creación de aplicaciones de entretenimiento, y por lo tanto, carece de herramientas que faciliten el desarrollo de aplicaciones más técnicas, como compatibilidad directa con modelos en otros formatos como '.dwg' o '.sldasm'. Aun así, Unity Technologies está trabajando con grandes empresas, especialmente del sector del automóvil para solucionar los problemas.
Con el desarrollo de este proyecto, se demuestra, que, a pesar de las carencias técnicas de Unity, se pueden crear aplicaciones de simulación industriales, especialmente en el área de formación de personal. El uso de simuladores en cualquier entorno facilita el aprendizaje y la formación. En el caso de una gran fábrica industrial, el mantenimiento de la planta es de especial importancia, especialmente si los beneficios se cuentan por horas. Para formar correctamente a los técnicos haría falta parar la producción o esperar a que se produjese en una avería, lo que se traduce en importantes pérdidas económicas. No solo eso, sino que también, según el tipo de planta, se podría comprometer la seguridad de los técnicos. El uso de simuladores de este tipo permite al personal llegar a la planta real con la formación necesaria para afrontar el problema.
La gran ventaja de la realidad virtual frente a los simuladores convencionales es que no hace falta una réplica física de la máquina/dispositivo que se vaya a reparar. Se puede generar una réplica virtual que funcione de la misma manera que lo haría la real, a partir de los planos de este. De este modo hay un ahorro en el coste de producción del equipo, con la ventaja de que el entorno virtual se puede actualizar, y en caso de que se cambie algún componente de la planta de la fábrica, se podría hacer en el entorno virtual de la misma manera.
Algunas empresas, como PixoGroup[14], han desarrollado aplicaciones de formación que permiten el uso simultáneo de la aplicación de varios operarios a la vez, de modo que se puedan entrenar dinámicas en las que se requiera más de una persona, desarrollando un método de formación que ayuda a los alumnos a retener mejor lo que aprenden durante la formación.
En conclusión, los programas de entrenamiento en Realidad Virtual son una realidad presente que, aunque todavía no se haya establecido completamente, está experimentando un gran crecimiento. Grandes empresas industriales están invirtiendo en este tipo

de aplicaciones, convirtiéndose la Realidad Virtual en uno de los principales impulsores hacia la implementación de la Industria 4.0.

El futuro de este tipo de aplicaciones dependerá de cómo se desarrolle la Realidad Virtual. En el caso de programas de formación, se podrían generar sistemas de enseñanza que formasen y examinasen a los operarios, mediante un sistema de tutoriales y lecciones virtuales. En cuanto a aplicaciones directamente relacionadas con el mantenimiento, sería de especial interés ser capaces de conectar directamente la planta en sí con la aplicación, de modo que los movimientos y estado de la planta física se reflejen en la virtual, así operario podría diagnosticar los problemas reales de la planta sin necesidad de estar, ni si quiera, en el mismo edificio; o poder controlar la planta desde la realidad virtual, minimizando el contacto físico entre operario y maquinaria. Otro de los posibles desarrollos es el diseño de las plantas, mediante la creación de modelos a partir de planos de CAD, poder visualizar y testear una planta de manera virtual, para comprobar su funcionamiento y distribución. Ésto sería posible mediante la creación, en caso de Unity, de modelos 'prefabs' que se puedan reciclar para ser utilizados en diferentes proyectos, ahorrando costes de diseño y programación. Por otro lado, alejado de la industria manufacturera, se podrían diseñar aplicaciones que simulasen incidentes en centros donde los accidentes son inusuales pero críticos, como el caso de las centrales nucleares. En estas centrales es difícil entrenar operarios de manera realista, ya que las incidencias son mínimas, pero a la vez son muy peligrosas. Por esto, una formación previa en un simulador de realidad virtual podría, en casos extremos, prevenir catástrofes.

Parte II

Presupuesto

Capítulo 9

Presupuesto del proyecto

9.1. Ingeniería

Tarea	**Horas**	**Precio/hora[€/h]**	**Precio[€]**
Documentación	30	20.00	600.00
Anteproyecto	55	20.00	1100.00
Diseño	25	25.00	625.00
Programación	180	30	5400.00
TOTAL			7725.00

Cuadro 9.1: Presupuesto de ingeniería

9.2. Hardware y software

Hardware	**Precio[€]**
Oculus Rift	499.00
Ordenador RV	1400.00
Ordenador personal	800.00
TOTAL	2699.00

Cuadro 9.2: Presupuesto de Hardware.

Software	**Precio[€]**
Oculus VR	Gratuito
Steam VR	Gratuito
Unity	Gratuito
TOTAL	0.00

Cuadro 9.3: Presupuesto de Software.

9.3. Total

Sección	**Precio[€]**
Hardware	2699.00
Software	0.00
Ingeniería	7725.00
TOTAL	10424.00

Cuadro 9.4: Presupuesto total del proyecto.

Parte III

Código fuente

9.4. Código de Unity

9.4.1. Belt.cs

```
//Nombre: Belt.cs
//Descripcion: Mueve los objetos de la cinta recta hacia delante
//Entradas: speed(velocidad de mov.), enpoint(punto final de trayecto)
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Belt : MonoBehaviour {

    //Punto final
    public Transform endpoint;
    //Velocidad
    public float speed;
    //Activo
    bool active = true;

    private void Update()
    {

    }

    private void OnTriggerStay(Collider other)
    {
        //Activa el movimiento

        //Debug.Log("Triggered");
        Move(other);

    }

    private void Move(Collider other)
    {
        //Mueve el objeto hacia delante

        //Debug.Log("Moving");
        if (active)
        {
            other.transform.position = Vector3.MoveTowards(other.transform.position, endpoint.position, speed * Time.deltaTime);
        }
    }

    void StopMovement()
    {
        active = false;
    }
    void IniMovement()
    {
```

```
        active = true;
    }

}
```

9.4.2. CircularBelt.cs

```
//Nombre: CircularBelt.cs
//Descripicion: Movimiento circular para la cinta transportadora.
    Movimiento de rotación y traslación.
//Entradas: Center, radius, traslationSpeed, angle, iniAngle,
    rotationSpeed
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CircularBelt : MonoBehaviour {

    //Centro de traslacion
    [SerializeField]
    Transform Center;

    //Radio de traslacion. Aviso: Ajustar si se reescala la planta.
    [SerializeField]
    float radius = 2f;

    //Velocidad de traslacion
    [SerializeField]
    float traslationSpeed = 2f;

    float posX, posZ;

    //Angulo inicio
    [SerializeField]
    float angle = -90f;

    //Angulo de reseteo
    [SerializeField]
    float iniAngle = -90f;

    //Velocida de rotacion
    [SerializeField]
    float rotationSpeed = 0.5f;

  // Use this for initialization
  void Start () {

  }

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

```

```
    }

    private void OnTriggerStay(Collider other)
    {
        //Cálculo de posición
        float posY = other.gameObject.transform.position.y;
        posX = Center.position.x + Mathf.Cos(angle) * radius;
        posZ = Center.position.z + Mathf.Sin(angle) * radius;

        //Transformación de posición
        other.transform.position = new Vector3(posX, posY, posZ);

        //Actualización del ángulo de traslación
        angle = angle + Time.deltaTime * traslationSpeed;

        //Debug.Log(angle);

        //Giro de rotación
        other.transform.Rotate(Vector3.forward, rotationSpeed*Time.
            deltaTime);



    }

    private void OnCollisionExit(Collision collision)
    {
        //Reseteo del ángulo al entrar un objeto nuevo.
        angle = iniAngle;
    }


}
```

9.4.3. FailureManager.cs

```
//Nombre: FailureManager.cs
//Descripción: Escoge fallo de forma aleatoria. Administra las
    interacciones con los objetos, temporizador, aciertos y fallos.
//Entradas:

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

public class FailureManager : MonoBehaviour
{
    //GameObjects que contienen los scripts con la plantilla de fallo.
    [SerializeField]
    public GameObject[] devices;
```

```

    //Índice de la avería actual, dependerá del orden en el que se hayan
       introducido los GameObjects en el campo anterior.
    int n;

    //ID del fallo actual
    int failure_id;

    //ID del objeto pulsado
    int object_id;

    //Indicador si la planta está en estado de fallo
    bool OnFailure = false;

    //Tiempo hasta siguiente fallo en la planta
    float TimeLeft = 20.0f;
    //Cuenta de aciertos y fallos
    int c_correct=0;
    int c_error=0;
    int timeLeft_i = 0;
    //IU para fallos, aciertos y temporizador.
    public TMP_Text countCorrect;
    public TMP_Text countError;
    public TMP_Text countTimer;


    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        FailureTimer();
        countCorrect.text = "ACIERTOS: " + c_correct.ToString();
        countError.text = "FALLOS: " + c_error.ToString();
        countTimer.text = "TIEMPO: " + timeLeft_i.ToString();
    }

    //Función: FailureTimer()
    //Descripción: Temporizador hasta siguiente avería.
    //Entradas: -

    void FailureTimer()
    {
        //Si la planta no se encuentra en estado de fallo, se inicia el
           temporizador.
        if (!OnFailure)
        {
            TimeLeft -= Time.deltaTime;
            Debug.Log(TimeLeft);
            timeLeft_i = (int)TimeLeft;
```

```

            if (TimeLeft < 0)
            {
                FailureSet();
                TimeLeft = 20.0f;
            }
        }
    }

    //Función:FailureSet()
    //Descripción: Selección aleatoria del fallo
    //Entradas: -

    void FailureSet()
    {

        OnFailure = true;
        n = Random.Range(0, devices.Length );
        Debug.Log("Length=" + devices.Length);
        devices[n].SendMessage("SensorFailure");
        Debug.Log("Fallo establecido =" + n);


    }

    //Función:OptionCheck(int id)
    //Descripción: Comprueba si el objeto seleccionado es el fallo. Si es
         correcto, desactiva la avería.
    //Entradas: id

    void OptionCheck(int id)
    {
        if (id == failure_id)
        {
            OnFailure = false;
            devices[n].SendMessage("FailureDetected");
            Debug.Log("Averia detectada");
            if(timeLeft_i==0)
                c_correct++;
            //Mensaje de acierto
        }
        else
        {
            OptionFalse();
        }
    }

    //Función:OptionFalse()
    //Descripción: Se llama cuando el objeto seleccionado no es el
        correcto. Suma un fallo.
    //Entradas: -

    public void OptionFalse()
    {
```

```
        //Mensaje de fallo
        Debug.Log("Opción falsa");
        if(timeLeft_i==0)
            c_error++;
    }

    //Función: StoreFailureID(int id)
    //Descripción: Almacena el ID del dispositivo que está fallando
    //Entradas: id

    void StoreFailureID(int id)
    {
        failure_id = id;
    }





}
```

9.4.4. MovementSensor.cs

```
//Nombre: MovementSensor.cs
//Descripcion: Detecta el movimiento de pallets al pasar por el sensor.
    Activa y desactiva el fallo en el sensor.
//Entradas: Luz, sensor*, materiales sensor.
//*Objeto que contiene la luz del sensor

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

public class MovementSensor : MonoBehaviour {

    //Gestor de fallos
    [SerializeField]
    public GameObject failure_manager;
    //ID del fallo
    int id = 101;
    //Objeto que contiene la luz
    public GameObject sensor;
    //Materiales del objeto en encendido/apagado
    public Material onLight;
    public Material offLight;
    //Luz
    public Light myLight;

    //Fallo en el sensor
    public bool fail = false;


```

```
// Use this for initialization
void Start () {
      //Empieza con luz apagada
      LightOff();
}

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


      RaycastHit hit;

      //Debug.DrawRay(transform.position, -Vector3.up, Color.blue);

      //Si el objeto es un pallet y no hay error en el sensor,
      //se enciende la luz. En caso contrario se apaga.
      if (Physics.Raycast(transform.position, -Vector3.up, out hit
          ,100.0f))
      {
          //Debug.Log("Objeto detectado");

          if (hit.transform.gameObject.tag == "pallet" && fail==false)
          {
              //Debug.Log("Sensor activado");
              LightOn();

          }
          else
          {
              LightOff();
          }
      }
}

  //Funcion: LightOn()
  //Descripcion: Enciende la luz del sensor.
  void LightOn()
  {
      //Cambio de material del objeto LED
      sensor.GetComponent<Renderer>().material = onLight;
      //Luz encendida
      myLight.enabled = true;

  }

  //Funcion: LightOff()
  //Descripcion: Apaga la luz del sensor.
  void LightOff()
  {
      //Cambio de material del objeto LED
      sensor.GetComponent<Renderer>().material = offLight;
      //Luz apagada
      myLight.enabled = false;
  }
```

```

    //Funcion: SensorFailure()                    PLANTILLA
    //Descripcion: Activa la averia y envía el ID a Failure Manager para
        almacenarlo.
    void SensorFailure()
    {

        fail = true;
        FailIDSet();
        Debug.Log("Avería sensor movimiento");
    }

    //Funcion: Failure detected()              PLANTILLA
    //Descripcion: Desactiva la averia.
    void FailureDetected()
    {
        //Desactiva el fallo
        fail = false;
    }

    //Función: FailIDSet()                        PLANTILLA
    //Descripción: Envía el ID del fallo a Failure Manager para que lo
        almacene.
    public void FailIDSet()
    {
        failure_manager.SendMessage("StoreFailureID", id);
    }

    //Función: SendID()                           PLANTILLA
    //Descripción: Envía el id del fallo a Failure Manager para comprobar
         si es el correcto.
    public void SendID()
    {
        failure_manager.SendMessage("OptionCheck", id);
    }
}
```

9.4.5. Pallet.cs

```
//Nombre: Pallet.cs
//Descripción: Corrige rotacion del pallet y la altura.
//Entradas: -
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Pallet : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

```

```
    }

    // Update is called once per frame
    void Update()
    {
        //Comprobar rotación del pallet
        if (transform.rotation.z != 0 || transform.rotation.x!=-90)
        {
            //Corrige rotación del pallet
            transform.eulerAngles = new Vector3(-90, 0, 0);
        }

        if (transform.position.y < 0.83f)
        {
            Debug.Log("Subir");
            while (transform.position.y < 0.83f)
            {
                transform.Translate(Vector3.up * Time.deltaTime, Space.
                    World);
            }
        }
    }
}
```

9.4.6. Plantilla Fallo.cs

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

public class PlantillaFallo : MonoBehaviour
{

    //Gestor de fallos.
    [SerializeField]
    public GameObject failure_manager;

    //ID del fallo.
    int id = 000;


    //Fallo en el sensor
    public bool fail = false;


    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
```

```
    void Update()
    {

    }


    //Funcion: SensorFailure()
    //Descripcion: Activa la averia y envía el ID a Failure Manager para
       almacenarlo.
    void SensorFailure()
    {

        fail = true;
        FailIDSet();
    }

    //Funcion: Failure detected()
    //Descripcion: Desactiva la averia.
    void FailureDetected()
    {
        //Desactiva el fallo
        fail = false;
    }

    //Función: FailIDSet()
    //Descripción: Envía el ID del fallo a Failure Manager para que lo
       almacene.
    public void FailIDSet()
    {
        failure_manager.SendMessage("StoreFailureID", id);
    }

    //Función: SendID()
    //Descripción: Envía el id del fallo a Failure Manager para comprobar
        si es el correcto.
    public void SendID()
    {
        failure_manager.SendMessage("OptionCheck", id);
    }

}
```

9.4.7. SDSwitch.cs

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

public class SDSwitch : MonoBehaviour
{
    //Stopping device
    [SerializeField]
```

```
    GameObject device;

    bool active = false;


    // Start is called before the first frame update
    void Start()
    {

    }

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

    }

    public void ButtonSwitch()
    {
        active = !active;
        if (active)
        {
            device.SendMessage("DeviceActivated");
            transform.Rotate(0, 0, 90);
            Debug.Log("Activated");
        }
        else if (!active)
        {
            device.SendMessage("DeviceOff");
            transform.Rotate(0, 0, -90);
            Debug.Log("Off");
        }

    }
}
```

9.4.8. StoppingDevice.cs

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

public class StoppingDevice : MonoBehaviour
{

    //Gestor de fallos.
    [SerializeField]
    public GameObject failure_manager;

    //ID del fallo.
    int id = 102;

```

```
    //Cinta sobre la que está el gatillo
    [SerializeField]
    GameObject belt;

    //Luz de la válvula
    [SerializeField]
    Light myLight;

    //Indica la pulsación del botón que activa el gatillo.
    public bool button = false;

    //Fallo
    public bool fail = false;
    //Active False: no bloquea el paso   True: Bloquea el paso
    public bool active = false;
    // Start is called before the first frame update
    void Start()
    {
        myLight.enabled = false;
    }

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

        if (active && fail == false)
        {

            transform.localPosition = new Vector3(0, -0.09f, -2.05f);


        } else if (!active && fail == false)
        {
            transform.localPosition = Vector3.zero;

        }else
        {
            //Si hay fallo, el gatillo se queda en la misma posición.
            transform.localPosition = transform.localPosition;
        }
    }

    //Funciones OnCollisionEnter/OnCollisionExit
    //Descripción: Evita movimientos no deseados en el pallet cuando el
        gatillo está activado
    //Entradas: collision
    private void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.tag == "pallet")
        {
            collision.rigidbody.velocity = Vector3.zero;
            belt.SendMessage("StopMovement");
        }

```

```
    }
    private void OnCollisionExit(Collision collision)
    {
        //Reanuda el movimiento de la cinta
        if (collision.gameObject.tag == "pallet")
        {
            belt.SendMessage("IniMovement");
        }
    }


    //Función: DeviceActivated()
    //Descripción: Activa el gatillo
    //Entradas: -
    void DeviceActivated()
    {
        active = true;
        myLight.enabled = true;
    }

    //Función: DeviceOff()
    //Descripción:Desactiva el gatillo.
    //Entradas: -
    void DeviceOff()
    {
        active = false;
        myLight.enabled = false;
    }




    //Funcion: SensorFailure()
    //Descripcion: Activa la averia y envía el ID a Failure Manager para
       almacenarlo.
    void SensorFailure()
    {

        fail = true;
        FailIDSet();
        //Debug.Log("Averia gatillo");
    }

    //Funcion: Failure detected()
    //Descripcion: Desactiva la averia.
    void FailureDetected()
    {
        //Desactiva el fallo
        fail = false;
    }

    //Función: FailIDSet()
    //Descripción: Envía el ID del fallo a Failure Manager para que lo
       almacene.
```

```
    public void FailIDSet()
    {
        failure_manager.SendMessage("StoreFailureID", id);
    }

    //Función: SendID()
    //Descripción: Envía el id del fallo a Failure Manager para comprobar
         si es el correcto.
    public void SendID()
    {
        failure_manager.SendMessage("OptionCheck", id);
        Debug.Log("ID enviado =" + id);
    }


}
```

9.4.9. ToggleLight.cs

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

public class ToggleLight : MonoBehaviour {

  // Use this for initialization
  void Start () {

  }

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

  }
    //void LightOn()
    //{
    //    //Luz encendida
    //   enabled = true;

    //}

    //void LightOff()
    //{
    //    //Luz encendida
    //    enabled = false;
    //}
}
```

9.4.10. Button.cs

```
using System;
using System.Collections;
using UnityEngine.Events;
using UnityEngine.EventSystems;
using UnityEngine.Serialization;

namespace UnityEngine.UI
{
    /// <summary>
    /// A standard button that sends an event when clicked.
    /// </summary>
    [AddComponentMenu("UI/Button", 30)]
    public class Button : Selectable, IPointerClickHandler,
       ISubmitHandler
    {
        [Serializable]
        /// <summary>
        /// Function definition for a button click event.
        /// </summary>
        public class ButtonClickedEvent : UnityEvent {}

        // Event delegates triggered on click.
        [FormerlySerializedAs("onClick")]
        [SerializeField]
        private ButtonClickedEvent m_OnClick = new ButtonClickedEvent();

        protected Button()
        {}

        /// <summary>
        /// UnityEvent that is triggered when the button is pressed.
        /// Note: Triggered on MouseUp after MouseDown on the same object
           .
        /// </summary>
        ///<example>
        ///<code>
        /// using UnityEngine;
        /// using UnityEngine.UI;
        /// using System.Collections;
        ///
        /// public class ClickExample : MonoBehaviour
        /// {
        ///     public Button yourButton;
        ///
        ///     void Start()
        ///     {
        ///         Button btn = yourButton.GetComponent<Button>();
        ///         btn.onClick.AddListener(TaskOnClick);
        ///     }
        ///
        ///     void TaskOnClick()
        ///     {
        ///         Debug.Log("You have clicked the button!");
```

```
        ///     }
        /// }
        ///</code>
        ///</example>
        public ButtonClickedEvent onClick
        {
            get { return m_OnClick; }
            set { m_OnClick = value; }
        }

        private void Press()
        {
            if (!IsActive() || !IsInteractable())
                return;

            UISystemProfilerApi.AddMarker("Button.onClick", this);
            m_OnClick.Invoke();
        }

        /// <summary>
        /// Call all registered IPointerClickHandlers.
        /// Register button presses using the IPointerClickHandler. You can also use it to tell what type of click happened (left, right etc.).
        /// Make sure your Scene has an EventSystem.
        /// </summary>
        /// <param name="eventData">Pointer Data associated with the event. Typically by the event system.</param>
        /// <example>
        /// <code>
        /// //Attatch this script to a Button GameObject
        /// using UnityEngine;
        /// using UnityEngine.EventSystems;
        ///
        /// public class Example : MonoBehaviour, IPointerClickHandler
        /// {
        ///     //Detect if a click occurs
        ///     public void OnPointerClick(PointerEventData pointerEventData)
        ///     {
        ///             //Use this to tell when the user right-clicks on the Button
        ///         if (pointerEventData.button == PointerEventData.InputButton.Right)
        ///         {
        ///             //Output to console the clicked GameObject's name and the following message. You can replace this with your own actions for when clicking the GameObject.
        ///             Debug.Log(name + " Game Object Right Clicked!");
        ///         }
        ///
        ///         //Use this to tell when the user left-clicks on the Button
```

```
        ///         if (pointerEventData.button == PointerEventData.InputButton.Left)
        ///         {
        ///             Debug.Log(name + " Game Object Left Clicked!");
        ///         }
        ///     }
        /// }
        /// </code>
        /// </example>

        public virtual void OnPointerClick(PointerEventData eventData)
        {
            if (eventData.button != PointerEventData.InputButton.Left)
                return;

            Press();
        }

        /// <summary>
        /// Call all registered ISubmitHandler.
        /// </summary>
        /// <param name="eventData">Associated data with the event. Typically by the event system.</param>
        /// <remarks>
        /// This detects when a Button has been selected via a "submit" key you specify (default is the return key).
        ///
        /// To change the submit key, either:
        ///
        /// 1. Go to Edit->Project Settings->Input.
        ///
        /// 2. Next, expand the Axes section and go to the Submit section if it exists.
        ///
        /// 3. If Submit doesnt exist, add 1 number to the Size field. This creates a new section at the bottom. Expand the new section and change the Name field to Submit.
        ///
        /// 4. Change the Positive Button field to the key you want (e.g. space).
        ///
        ///
        /// Or:
        ///
        /// 1. Go to your EventSystem in your Project
        ///
        /// 2. Go to the Inspector window and change the Submit Button field to one of the sections in the Input Manager (e.g. "Submit"), or create your own by naming it what you like, then following the next few steps.
        ///
        /// 3. Go to Edit->Project Settings->Input to get to the Input Manager.
        ///
```

```
        /// 4. Expand the Axes section in the Inspector window. Add 1 to
            the number in the Size field. This creates a new section at
            the bottom.
        ///
        /// 5. Expand the new section and name it the same as the name
            you inserted in the Submit Button field in the EventSystem.
            Set the Positive Button field to the key you want (e.g. space
            )
        /// </remarks>

        public virtual void OnSubmit(BaseEventData eventData)
        {
            Press();

            // if we get set disabled during the press
            // don't run the coroutine.
            if (!IsActive() || !IsInteractable())
                return;

            DoStateTransition(SelectionState.Pressed, false);
            StartCoroutine(OnFinishSubmit());
        }

        private IEnumerator OnFinishSubmit()
        {
            var fadeTime = colors.fadeDuration;
            var elapsedTime = 0f;

            while (elapsedTime < fadeTime)
            {
                elapsedTime += Time.unscaledDeltaTime;
                yield return null;
            }

            DoStateTransition(currentSelectionState, false);
        }
    }
}
```

9.5. Código de Steam VR

9.5.1. BodyCollider.cs

```
//======= Copyright (c) Valve Corporation, All rights reserved.
    ===============
//
// Purpose: Collider dangling from the player's head
//
//
    =============================================================================
```

```

using UnityEngine;
using System.Collections;

namespace Valve.VR.InteractionSystem
{
  //-------------------------------------------------------------------------
  [RequireComponent( typeof( CapsuleCollider ) )]
  public class BodyCollider : MonoBehaviour
  {
    public Transform head;

    private CapsuleCollider capsuleCollider;

    //-------------------------------------------------
    void Awake()
    {
      capsuleCollider = GetComponent<CapsuleCollider>();
    }


    //-------------------------------------------------
    void FixedUpdate()
    {
      float distanceFromFloor = Vector3.Dot( head.localPosition, Vector3.up );
      capsuleCollider.height = Mathf.Max( capsuleCollider.radius, distanceFromFloor );
      transform.localPosition = head.localPosition - 0.5f * distanceFromFloor * Vector3.up;
    }
  }
}
```

9.5.2. ControllerButtonHints.cs

```
//======= Copyright (c) Valve Corporation, All rights reserved. ===============
//
// Purpose: Displays text and button hints on the controllers
//
//=============================================================================

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

```

namespace Valve.VR.InteractionSystem
{
  //
      -------------------------------------------------------------------------

  public class ControllerButtonHints : MonoBehaviour
  {
    public Material controllerMaterial;
    public Color flashColor = new Color( 1.0f, 0.557f, 0.0f );
    public GameObject textHintPrefab;

        public SteamVR_Action_Vibration hapticFlash = SteamVR_Input.
            GetAction<SteamVR_Action_Vibration>("Haptic");

        [Header( "Debug" )]
    public bool debugHints = false;

    private SteamVR_RenderModel renderModel;
    private Player player;

    private List<MeshRenderer> renderers = new List<MeshRenderer>();
    private List<MeshRenderer> flashingRenderers = new List<MeshRenderer
        >();
    private float startTime;
    private float tickCount;

    private enum OffsetType
    {
      Up,
      Right,
      Forward,
      Back
    }

    //Info for each of the buttons
    private class ActionHintInfo
    {
      public string componentName;
      public List<MeshRenderer> renderers;
      public Transform localTransform;

      //Text hint
      public GameObject textHintObject;
      public Transform textStartAnchor;
      public Transform textEndAnchor;
      public Vector3 textEndOffsetDir;
      public Transform canvasOffset;

      public Text text;
      public TextMesh textMesh;
      public Canvas textCanvas;
      public LineRenderer line;

```

```
		public float distanceFromCenter;
		public bool textHintActive = false;
	}

	private Dictionary<ISteamVR_Action_In_Source, ActionHintInfo>
	   actionHintInfos;
	private Transform textHintParent;

	private int colorID;

	public bool initialized { get; private set; }
	private Vector3 centerPosition = Vector3.zero;

	SteamVR_Events.Action renderModelLoadedAction;

		protected SteamVR_Input_Sources inputSource;

		//-------------------------------------------------
		void Awake()
	{
		renderModelLoadedAction = SteamVR_Events.RenderModelLoadedAction(
		   OnRenderModelLoaded );
		colorID = Shader.PropertyToID( "_Color" );
	}


	//-------------------------------------------------
	void Start()
	{
		player = Player.instance;
	}


	//-------------------------------------------------
	private void HintDebugLog( string msg )
	{
		if ( debugHints )
		{
			Debug.Log("<b>[SteamVR Interaction]</b> Hints: " + msg );
		}
	}


	//-------------------------------------------------
	void OnEnable()
	{
		renderModelLoadedAction.enabled = true;
	}


	//-------------------------------------------------
	void OnDisable()
	{
		renderModelLoadedAction.enabled = false;
```

```
      Clear();
    }


    //--------------------------------------------------
    private void OnParentHandInputFocusLost()
    {
      //Hide all the hints when the controller is no longer the primary attached object
      HideAllButtonHints();
      HideAllText();
    }


        public virtual void SetInputSource(SteamVR_Input_Sources newInputSource)
        {
            inputSource = newInputSource;
            if (renderModel != null)
                renderModel.SetInputSource(newInputSource);
        }

        //--------------------------------------------------
        // Gets called when the hand has been initialized and a render model has been set
        //--------------------------------------------------
        private void OnHandInitialized(int deviceIndex)
        {
      //Create a new render model for the controller hints
      renderModel = new GameObject( "SteamVR_RenderModel" ).AddComponent<SteamVR_RenderModel>();
      renderModel.transform.parent = transform;
      renderModel.transform.localPosition = Vector3.zero;
      renderModel.transform.localRotation = Quaternion.identity;
      renderModel.transform.localScale = Vector3.one;

            renderModel.SetInputSource(inputSource);
            renderModel.SetDeviceIndex(deviceIndex);

            if ( !initialized )
      {
        //The controller hint render model needs to be active to get accurate transforms for all the individual components
        renderModel.gameObject.SetActive( true );
      }
    }


        private Dictionary<string, Transform> componentTransformMap = new Dictionary<string, Transform>();

        //--------------------------------------------------
        void OnRenderModelLoaded(SteamVR_RenderModel renderModel, bool succeess)
```

```
        {
            //Only initialize when the render model for the controller hints has been loaded
            if (renderModel == this.renderModel)
            {
                //Debug.Log("<b>[SteamVR Interaction]</b> OnRenderModelLoaded: " + this.renderModel.renderModelName);
                if (initialized)
                {
                    Destroy(textHintParent.gameObject);
                    componentTransformMap.Clear();
                    flashingRenderers.Clear();
                }

                renderModel.SetMeshRendererState(false);

                StartCoroutine(DoInitialize(renderModel));
            }
        }
        private IEnumerator DoInitialize(SteamVR_RenderModel renderModel)
        {
            while (renderModel.initializedAttachPoints == false)
                yield return null;

            textHintParent = new GameObject("Text Hints").transform;
            textHintParent.SetParent(this.transform);
            textHintParent.localPosition = Vector3.zero;
            textHintParent.localRotation = Quaternion.identity;
            textHintParent.localScale = Vector3.one;

            //Get the button mask for each component of the render model

            var renderModels = OpenVR.RenderModels;
            if (renderModels != null)
            {
                string renderModelDebug = "";

                if (debugHints)
                    renderModelDebug = "Components for render model " + renderModel.index;

                for (int childIndex = 0; childIndex < renderModel.transform.childCount; childIndex++)
                {
                    Transform child = renderModel.transform.GetChild(childIndex);

                    if (componentTransformMap.ContainsKey(child.name))
                    {
                        if (debugHints)
                            renderModelDebug += "\n\t!    Child component already exists with name: " + child.name;
```

```
                }
                else
                    componentTransformMap.Add(child.name, child);

                if (debugHints)
                    renderModelDebug += "\n\t" + child.name + ".";
            }

            //Uncomment to show the button mask for each component of the render model
            HintDebugLog(renderModelDebug);
        }

        actionHintInfos = new Dictionary<ISteamVR_Action_In_Source, ActionHintInfo>();

        for (int actionIndex = 0; actionIndex < SteamVR_Input.actionsNonPoseNonSkeletonIn.Length; actionIndex++)
        {
            ISteamVR_Action_In action = SteamVR_Input.actionsNonPoseNonSkeletonIn[actionIndex];

            if (action.GetActive(inputSource))
                CreateAndAddButtonInfo(action, inputSource);
        }

        ComputeTextEndTransforms();

        initialized = true;

        //Set the controller hints render model to not active
        renderModel.SetMeshRendererState(true);
        renderModel.gameObject.SetActive(false);
    }


//-------------------------------------------------
private void CreateAndAddButtonInfo(ISteamVR_Action_In action, SteamVR_Input_Sources inputSource)
{
  Transform buttonTransform = null;
  List<MeshRenderer> buttonRenderers = new List<MeshRenderer>();

        StringBuilder buttonDebug = new StringBuilder();
        buttonDebug.Append("Looking for action: ");

        buttonDebug.AppendLine(action.GetShortName());

        buttonDebug.Append("Action localized origin: ");
        buttonDebug.AppendLine(action.GetLocalizedOrigin(inputSource));

        string actionComponentName = action.GetRenderModelComponentName(inputSource);
```

```

            if (componentTransformMap.ContainsKey(actionComponentName))
            {
                buttonDebug.AppendLine(string.Format("Found component: {0} for {1}", actionComponentName, action.GetShortName()));
                Transform componentTransform = componentTransformMap[actionComponentName];

                buttonTransform = componentTransform;

                buttonDebug.AppendLine(string.Format("Found componentTransform: {0}. buttonTransform: {1}", componentTransform, buttonTransform));

                buttonRenderers.AddRange(componentTransform.GetComponentsInChildren<MeshRenderer>());
            }
            else
            {
                buttonDebug.AppendLine(string.Format("Can't find component transform for action: {0}. Component name: \"{1}\"", action.GetShortName(), actionComponentName));
            }

            buttonDebug.AppendLine(string.Format("Found {0} renderers for {1}", buttonRenderers.Count, action.GetShortName()));

        foreach ( MeshRenderer renderer in buttonRenderers )
        {
                buttonDebug.Append("\t");
                buttonDebug.AppendLine(renderer.name);
        }

        HintDebugLog( buttonDebug.ToString() );

        if ( buttonTransform == null )
        {
          HintDebugLog( "Couldn't find buttonTransform for " + action.GetShortName());
          return;
        }

        ActionHintInfo hintInfo = new ActionHintInfo();
        actionHintInfos.Add( action, hintInfo );

        hintInfo.componentName = buttonTransform.name;
        hintInfo.renderers = buttonRenderers;

        //Get the local transform for the button
            for (int childIndex = 0; childIndex < buttonTransform.childCount; childIndex++)
            {
```

```
              Transform child = buttonTransform.GetChild(childIndex);
              if (child.name == SteamVR_RenderModel.k_localTransformName)
                  hintInfo.localTransform = child;
          }

    OffsetType offsetType = OffsetType.Right;

          /*
          switch ( buttonID )
    {
      case EVRButtonId.k_EButton_SteamVR_Trigger:
        {
          offsetType = OffsetType.Right;
        }
        break;
      case EVRButtonId.k_EButton_ApplicationMenu:
        {
          offsetType = OffsetType.Right;
        }
        break;
      case EVRButtonId.k_EButton_System:
        {
          offsetType = OffsetType.Right;
        }
        break;
      case Valve.VR.EVRButtonId.k_EButton_Grip:
        {
          offsetType = OffsetType.Forward;
        }
        break;
      case Valve.VR.EVRButtonId.k_EButton_SteamVR_Touchpad:
        {
          offsetType = OffsetType.Up;
        }
        break;
    }
          */

          //Offset for the text end transform
          switch ( offsetType )
    {
      case OffsetType.Forward:
        hintInfo.textEndOffsetDir = hintInfo.localTransform.forward;
        break;
      case OffsetType.Back:
        hintInfo.textEndOffsetDir = -hintInfo.localTransform.forward;
        break;
      case OffsetType.Right:
        hintInfo.textEndOffsetDir = hintInfo.localTransform.right;
        break;
      case OffsetType.Up:
        hintInfo.textEndOffsetDir = hintInfo.localTransform.up;
        break;
```

```
        }

        //Create the text hint object
        Vector3 hintStartPos = hintInfo.localTransform.position + (
            hintInfo.localTransform.forward * 0.01f );
        hintInfo.textHintObject = GameObject.Instantiate( textHintPrefab,
            hintStartPos, Quaternion.identity ) as GameObject;
        hintInfo.textHintObject.name = "Hint_" + hintInfo.componentName + "
            _Start";
        hintInfo.textHintObject.transform.SetParent( textHintParent );
        hintInfo.textHintObject.layer = gameObject.layer;
        hintInfo.textHintObject.tag = gameObject.tag;

        //Get all the relevant child objects
        hintInfo.textStartAnchor = hintInfo.textHintObject.transform.Find(
            "Start" );
        hintInfo.textEndAnchor = hintInfo.textHintObject.transform.Find( "
            End" );
        hintInfo.canvasOffset = hintInfo.textHintObject.transform.Find( "
            CanvasOffset" );
        hintInfo.line = hintInfo.textHintObject.transform.Find( "Line" ).
            GetComponent<LineRenderer>();
        hintInfo.textCanvas = hintInfo.textHintObject.
            GetComponentInChildren<Canvas>();
        hintInfo.text = hintInfo.textCanvas.GetComponentInChildren<Text>();
        hintInfo.textMesh = hintInfo.textCanvas.GetComponentInChildren<
            TextMesh>();

        hintInfo.textHintObject.SetActive( false );

        hintInfo.textStartAnchor.position = hintStartPos;

        if ( hintInfo.text != null )
        {
          hintInfo.text.text = hintInfo.componentName;
        }

        if ( hintInfo.textMesh != null )
        {
          hintInfo.textMesh.text = hintInfo.componentName;
        }

        centerPosition += hintInfo.textStartAnchor.position;

        // Scale hint components to match player size
        hintInfo.textCanvas.transform.localScale = Vector3.Scale( hintInfo.
            textCanvas.transform.localScale, player.transform.localScale );
        hintInfo.textStartAnchor.transform.localScale = Vector3.Scale(
            hintInfo.textStartAnchor.transform.localScale, player.transform
            .localScale );
        hintInfo.textEndAnchor.transform.localScale = Vector3.Scale(
            hintInfo.textEndAnchor.transform.localScale, player.transform.
            localScale );
```

```
      hintInfo.line.transform.localScale = Vector3.Scale( hintInfo.line.
          transform.localScale, player.transform.localScale );
    }


    //-------------------------------------------------
    private void ComputeTextEndTransforms()
    {
      //This is done as a separate step after all the ButtonHintInfos
          have been initialized
      //to make the text hints fan out appropriately based on the button'
          s position on the controller.

      centerPosition /= actionHintInfos.Count;
      float maxDistanceFromCenter = 0.0f;

      foreach ( var hintInfo in actionHintInfos )
      {
        hintInfo.Value.distanceFromCenter = Vector3.Distance( hintInfo.
            Value.textStartAnchor.position, centerPosition );

        if ( hintInfo.Value.distanceFromCenter > maxDistanceFromCenter )
        {
          maxDistanceFromCenter = hintInfo.Value.distanceFromCenter;
        }
      }

      foreach ( var hintInfo in actionHintInfos )
      {
        Vector3 centerToButton = hintInfo.Value.textStartAnchor.position
            - centerPosition;
        centerToButton.Normalize();

        centerToButton = Vector3.Project( centerToButton, renderModel.
            transform.forward );

        //Spread out the text end positions based on the distance from
            the center
        float t = hintInfo.Value.distanceFromCenter /
            maxDistanceFromCenter;
        float scale = hintInfo.Value.distanceFromCenter * Mathf.Pow( 2,
            10 * ( t - 1.0f ) ) * 20.0f;

        //Flip the direction of the end pos based on which hand this is
        float endPosOffset = 0.1f;

        Vector3 hintEndPos = hintInfo.Value.textStartAnchor.position + (
            hintInfo.Value.textEndOffsetDir * endPosOffset ) + (
            centerToButton * scale * 0.1f );

                if (SteamVR_Utils.IsValid(hintEndPos))
                {
                    hintInfo.Value.textEndAnchor.position = hintEndPos;

```

```
                    hintInfo.Value.canvasOffset.position = hintEndPos;
                }
                else
                {
                    Debug.LogWarning("<b>[SteamVR Interaction]</b>
                        Invalid end position for: " + hintInfo.Value.
                        textStartAnchor.name, hintInfo.Value.
                        textStartAnchor.gameObject);
                }
            hintInfo.Value.canvasOffset.localRotation = Quaternion.identity;
        }
    }


    //-------------------------------------------------
    private void ShowButtonHint( params ISteamVR_Action_In_Source[]
        actions )
    {
        renderModel.gameObject.SetActive( true );

        renderModel.GetComponentsInChildren<MeshRenderer>( renderers );
        for ( int i = 0; i < renderers.Count; i++ )
        {
            Texture mainTexture = renderers[i].material.mainTexture;
            renderers[i].sharedMaterial = controllerMaterial;
            renderers[i].material.mainTexture = mainTexture;

            // This is to poke unity into setting the correct render queue
                for the model
            renderers[i].material.renderQueue = controllerMaterial.shader.
                renderQueue;
        }

        for ( int i = 0; i < actions.Length; i++ )
        {
            if ( actionHintInfos.ContainsKey( actions[i] ) )
            {
                ActionHintInfo hintInfo = actionHintInfos[actions[i]];
                foreach ( MeshRenderer renderer in hintInfo.renderers )
                {
                    if ( !flashingRenderers.Contains( renderer ) )
                    {
                        flashingRenderers.Add( renderer );
                    }
                }
            }
        }

        startTime = Time.realtimeSinceStartup;
        tickCount = 0.0f;
    }


    //-------------------------------------------------
```

```
	private void HideAllButtonHints()
	{
	  Clear();

	        if (renderModel != null && renderModel.gameObject != null)
	      renderModel.gameObject.SetActive( false );
	}


	//-------------------------------------------------
	private void HideButtonHint( params ISteamVR_Action_In_Source[]
	    actions )
	{
	  Color baseColor = controllerMaterial.GetColor( colorID );
	  for ( int i = 0; i < actions.Length; i++ )
	  {
	    if ( actionHintInfos.ContainsKey(actions[i] ) )
	    {
	      ActionHintInfo hintInfo = actionHintInfos[actions[i]];
	      foreach ( MeshRenderer renderer in hintInfo.renderers )
	      {
	        renderer.material.color = baseColor;
	        flashingRenderers.Remove( renderer );
	      }
	    }
	  }

	  if ( flashingRenderers.Count == 0 )
	  {
	    renderModel.gameObject.SetActive( false );
	  }
	}




	//-------------------------------------------------
	private bool IsButtonHintActive(ISteamVR_Action_In_Source action )
	{
	  if ( actionHintInfos.ContainsKey(action) )
	  {
	    ActionHintInfo hintInfo = actionHintInfos[action];
	    foreach ( MeshRenderer buttonRenderer in hintInfo.renderers )
	    {
	      if ( flashingRenderers.Contains( buttonRenderer ) )
	      {
	        return true;
	      }
	    }
	  }

	  return false;
	}

```

```

	//-------------------------------------------------
	private IEnumerator TestButtonHints()
	{
		while ( true )
		{
				for (int actionIndex = 0; actionIndex < SteamVR_Input.actionsNonPoseNonSkeletonIn.Length; actionIndex++)
				{
					ISteamVR_Action_In action = SteamVR_Input.actionsNonPoseNonSkeletonIn[actionIndex];
					if (action.GetActive(inputSource))
					{
						ShowButtonHint(action);
						yield return new WaitForSeconds(1.0f);
					}
					yield return null;
				}
		}
	}


	//-------------------------------------------------
	private IEnumerator TestTextHints()
	{
		while ( true )
			{
				for (int actionIndex = 0; actionIndex < SteamVR_Input.actionsNonPoseNonSkeletonIn.Length; actionIndex++)
				{
					ISteamVR_Action_In action = SteamVR_Input.actionsNonPoseNonSkeletonIn[actionIndex];
					if (action.GetActive(inputSource))
					{
						ShowText(action, action.GetShortName());
						yield return new WaitForSeconds(3.0f);
					}
					yield return null;
				}

			HideAllText();
			yield return new WaitForSeconds( 3.0f );
		}
	}


	//-------------------------------------------------
	void Update()
	{
		if ( renderModel != null && renderModel.gameObject.activeInHierarchy && flashingRenderers.Count > 0 )
		{
			Color baseColor = controllerMaterial.GetColor( colorID );

```

```
      float flash = ( Time.realtimeSinceStartup - startTime ) * Mathf.PI * 2.0f;
      flash = Mathf.Cos( flash );
      flash = Util.RemapNumberClamped( flash, -1.0f, 1.0f, 0.0f, 1.0f );

      float ticks = ( Time.realtimeSinceStartup - startTime );
      if ( ticks - tickCount > 1.0f )
      {
        tickCount += 1.0f;
                  hapticFlash.Execute(0, 0.005f, 0.005f, 1, inputSource);
      }

      for ( int i = 0; i < flashingRenderers.Count; i++ )
      {
        Renderer r = flashingRenderers[i];
        r.material.SetColor( colorID, Color.Lerp( baseColor, flashColor, flash ) );
      }

      if ( initialized )
      {
        foreach ( var hintInfo in actionHintInfos )
        {
          if ( hintInfo.Value.textHintActive )
          {
            UpdateTextHint( hintInfo.Value );
          }
        }
      }
    }
  }


  //-------------------------------------------------
  private void UpdateTextHint( ActionHintInfo hintInfo )
  {
    Transform playerTransform = player.hmdTransform;
    Vector3 vDir = playerTransform.position - hintInfo.canvasOffset.position;

    Quaternion standardLookat = Quaternion.LookRotation( vDir, Vector3.up );
    Quaternion upsideDownLookat = Quaternion.LookRotation( vDir, playerTransform.up );

    float flInterp;
    if ( playerTransform.forward.y > 0.0f )
    {
      flInterp = Util.RemapNumberClamped( playerTransform.forward.y, 0.6f, 0.4f, 1.0f, 0.0f );
    }
    else
```

```
    {
      flInterp = Util.RemapNumberClamped( playerTransform.forward.y, -0.8f, -0.6f, 1.0f, 0.0f );
    }

    hintInfo.canvasOffset.rotation = Quaternion.Slerp( standardLookat, upsideDownLookat, flInterp );

    Transform lineTransform = hintInfo.line.transform;

    hintInfo.line.useWorldSpace = false;
    hintInfo.line.SetPosition( 0, lineTransform.InverseTransformPoint( hintInfo.textStartAnchor.position ) );
    hintInfo.line.SetPosition( 1, lineTransform.InverseTransformPoint( hintInfo.textEndAnchor.position ) );
  }


  //-------------------------------------------------
  private void Clear()
  {
    renderers.Clear();
    flashingRenderers.Clear();
  }


  //-------------------------------------------------
  private void ShowText(ISteamVR_Action_In_Source action, string text, bool highlightButton = true )
      {
          if ( actionHintInfos.ContainsKey(action) )
          {
              ActionHintInfo hintInfo = actionHintInfos[action];
      hintInfo.textHintObject.SetActive( true );
      hintInfo.textHintActive = true;

      if ( hintInfo.text != null )
      {
        hintInfo.text.text = text;
      }

      if ( hintInfo.textMesh != null )
      {
        hintInfo.textMesh.text = text;
      }

      UpdateTextHint( hintInfo );

      if ( highlightButton )
      {
        ShowButtonHint(action);
      }

      renderModel.gameObject.SetActive( true );
```

```
		}
	}


	//-------------------------------------------------
	private void HideText(ISteamVR_Action_In_Source action)
	{
		if ( actionHintInfos.ContainsKey(action) )
		{
			ActionHintInfo hintInfo = actionHintInfos[action];
			hintInfo.textHintObject.SetActive( false );
			hintInfo.textHintActive = false;

			HideButtonHint(action);
		}
	}


	//-------------------------------------------------
	private void HideAllText()
	{
            if (actionHintInfos != null)
            {

                foreach (var hintInfo in actionHintInfos)
                {
                    hintInfo.Value.textHintObject.SetActive(false);
                    hintInfo.Value.textHintActive = false;
                }

                HideAllButtonHints();
            }
	}


	//-------------------------------------------------
	private string GetActiveHintText(ISteamVR_Action_In_Source action )
	{
		if ( actionHintInfos.ContainsKey(action) )
		{
			ActionHintInfo hintInfo = actionHintInfos[action];
			if ( hintInfo.textHintActive )
			{
				return hintInfo.text.text;
			}
		}
		return string.Empty;
	}


	//-------------------------------------------------
	// These are the static functions which are used to show/hide the
	   hints
	//-------------------------------------------------
```

```
	//-------------------------------------------------
	private static ControllerButtonHints GetControllerButtonHints( Hand hand )
	{
		if ( hand != null )
		{
			ControllerButtonHints hints = hand.GetComponentInChildren<ControllerButtonHints>();
			if ( hints != null && hints.initialized )
			{
				return hints;
			}
		}

		return null;
	}


	//-------------------------------------------------
	public static void ShowButtonHint( Hand hand, params ISteamVR_Action_In_Source[] actions )
	{
		ControllerButtonHints hints = GetControllerButtonHints( hand );
		if ( hints != null )
		{
			hints.ShowButtonHint( actions );
		}
	}


	//-------------------------------------------------
	public static void HideButtonHint( Hand hand, params ISteamVR_Action_In_Source[] actions )
	{
		ControllerButtonHints hints = GetControllerButtonHints( hand );
		if ( hints != null )
		{
			hints.HideButtonHint( actions );
		}
	}


	//-------------------------------------------------
	public static void HideAllButtonHints( Hand hand )
	{
		ControllerButtonHints hints = GetControllerButtonHints( hand );
		if ( hints != null )
		{
			hints.HideAllButtonHints();
		}
	}

```

```
//-------------------------------------------------
public static bool IsButtonHintActive( Hand hand,
    ISteamVR_Action_In_Source action )
{
  ControllerButtonHints hints = GetControllerButtonHints( hand );
  if ( hints != null )
  {
    return hints.IsButtonHintActive(action);
  }

  return false;
}


//-------------------------------------------------
public static void ShowTextHint( Hand hand, ISteamVR_Action_In_Source
     action, string text, bool highlightButton = true )
    {
        ControllerButtonHints hints = GetControllerButtonHints( hand
            );
  if ( hints != null )
  {
    hints.ShowText(action, text, highlightButton );
  }
}


//-------------------------------------------------
public static void HideTextHint( Hand hand, ISteamVR_Action_In_Source
     action)
{
  ControllerButtonHints hints = GetControllerButtonHints( hand );
  if ( hints != null )
  {
    hints.HideText(action);
  }
}


//-------------------------------------------------
public static void HideAllTextHints( Hand hand )
{
  ControllerButtonHints hints = GetControllerButtonHints( hand );
  if ( hints != null )
  {
    hints.HideAllText();
  }
}


//-------------------------------------------------
public static string GetActiveHintText( Hand hand,
    ISteamVR_Action_In_Source action)
{
```

```
      ControllerButtonHints hints = GetControllerButtonHints( hand );
      if ( hints != null )
      {
        return hints.GetActiveHintText(action);
      }

      return string.Empty;
    }
  }
}
```

9.5.3. ControllerHoverHighlight.cs

```
//======= Copyright (c) Valve Corporation, All rights reserved.
    ===============
//
// Purpose: Highlights the controller when hovering over interactables
//
//
    =============================================================================

using UnityEngine;
using System.Collections;

namespace Valve.VR.InteractionSystem
{
    //
        -------------------------------------------------------------------------
    public class ControllerHoverHighlight : MonoBehaviour
    {
        public Material highLightMaterial;
        public bool fireHapticsOnHightlight = true;

        protected Hand hand;

        protected RenderModel renderModel;

        protected SteamVR_Events.Action renderModelLoadedAction;

        protected void Awake()
        {
            hand = GetComponentInParent<Hand>();
        }

        protected void OnHandInitialized(int deviceIndex)
        {
            GameObject renderModelGameObject = GameObject.Instantiate(
                hand.renderModelPrefab);
            renderModelGameObject.transform.parent = this.transform;
            renderModelGameObject.transform.localPosition = Vector3.zero;
```

```
            renderModelGameObject.transform.localRotation = Quaternion.identity;
            renderModelGameObject.transform.localScale = hand.renderModelPrefab.transform.localScale;


            renderModel = renderModelGameObject.GetComponent<RenderModel>();

            renderModel.SetInputSource(hand.handType);
            renderModel.OnHandInitialized(deviceIndex);
            renderModel.SetMaterial(highLightMaterial);

            hand.SetHoverRenderModel(renderModel);
            renderModel.onControllerLoaded += RenderModel_onControllerLoaded;
            renderModel.Hide();
        }

        private void RenderModel_onControllerLoaded()
        {
            renderModel.Hide();
        }


        //-------------------------------------------------
        protected void OnParentHandHoverBegin(Interactable other)
        {
            if (!this.isActiveAndEnabled)
            {
                return;
            }

            if (other.transform.parent != transform.parent)
            {
                ShowHighlight();
            }
        }


        //-------------------------------------------------
        private void OnParentHandHoverEnd(Interactable other)
        {
            HideHighlight();
        }


        //-------------------------------------------------
        private void OnParentHandInputFocusAcquired()
        {
            if (!this.isActiveAndEnabled)
            {
                return;
            }
```

```

            if (hand.hoveringInteractable && hand.hoveringInteractable.
                transform.parent != transform.parent)
            {
                ShowHighlight();
            }
        }


        //-------------------------------------------------
        private void OnParentHandInputFocusLost()
        {
            HideHighlight();
        }


        //-------------------------------------------------
        public void ShowHighlight()
        {
            if (renderModel == null)
            {
                return;
            }

            if (fireHapticsOnHightlight)
            {
                hand.TriggerHapticPulse(500);
            }

            renderModel.Show();
        }


        //-------------------------------------------------
        public void HideHighlight()
        {
            if (renderModel == null)
            {
                return;
            }

            if (fireHapticsOnHightlight)
            {
                hand.TriggerHapticPulse(300);
            }

            renderModel.Hide();
        }
    }
}
```

9.5.4. Hand.cs

```
//======= Copyright (c) Valve Corporation, All rights reserved.
    ===============
//
// Purpose: The hands used by the player in the vr interaction system
//
//
    ===========================================================================

using UnityEngine;
using System;
using System.Collections;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using UnityEngine.Events;
using System.Threading;

namespace Valve.VR.InteractionSystem
{
    //
        -----------------------------------------------------------------------
    // Links with an appropriate SteamVR controller and facilitates
    // interactions with objects in the virtual world.
    //
        -----------------------------------------------------------------------
    public class Hand : MonoBehaviour
    {
        // The flags used to determine how an object is attached to the
            hand.
        [Flags]
        public enum AttachmentFlags
        {
            SnapOnAttach = 1 << 0, // The object should snap to the
                position of the specified attachment point on the hand.
            DetachOthers = 1 << 1, // Other objects attached to this hand
                 will be detached.
            DetachFromOtherHand = 1 << 2, // This object will be detached
                 from the other hand.
            ParentToHand = 1 << 3, // The object will be parented to the
                hand.
            VelocityMovement = 1 << 4, // The object will attempt to move
                 to match the position and rotation of the hand.
            TurnOnKinematic = 1 << 5, // The object will not respond to
                external physics.
            TurnOffGravity = 1 << 6, // The object will not respond to
                external physics.
            AllowSidegrade = 1 << 7, // The object is able to switch from
                 a pinch grab to a grip grab. Decreases likelyhood of a
                good throw but also decreases likelyhood of accidental
                drop
        };
```

```
        public const AttachmentFlags defaultAttachmentFlags = AttachmentFlags.ParentToHand |
                                                              AttachmentFlags.DetachOthers |
                                                              AttachmentFlags.DetachFromOtherHand |
                                                              AttachmentFlags.TurnOnKinematic |
                                                              AttachmentFlags.SnapOnAttach;

        public Hand otherHand;
        public SteamVR_Input_Sources handType;

        public SteamVR_Behaviour_Pose trackedObject;

        public SteamVR_Action_Boolean grabPinchAction = SteamVR_Input.GetAction<SteamVR_Action_Boolean>("GrabPinch");

        public SteamVR_Action_Boolean grabGripAction = SteamVR_Input.GetAction<SteamVR_Action_Boolean>("GrabGrip");

        public SteamVR_Action_Vibration hapticAction = SteamVR_Input.GetAction<SteamVR_Action_Vibration>("Haptic");

        public SteamVR_Action_Boolean uiInteractAction = SteamVR_Input.GetAction<SteamVR_Action_Boolean>("InteractUI");

        public bool useHoverSphere = true;
        public Transform hoverSphereTransform;
        public float hoverSphereRadius = 0.05f;
        public LayerMask hoverLayerMask = -1;
        public float hoverUpdateInterval = 0.1f;

        public bool useControllerHoverComponent = true;
        public string controllerHoverComponent = "tip";
        public float controllerHoverRadius = 0.075f;

        public bool useFingerJointHover = true;
        public SteamVR_Skeleton_JointIndexEnum fingerJointHover = SteamVR_Skeleton_JointIndexEnum.indexTip;
        public float fingerJointHoverRadius = 0.025f;

        [Tooltip("A transform on the hand to center attached objects on")]
```

```
        public Transform objectAttachmentPoint;

        public Camera noSteamVRFallbackCamera;
        public float noSteamVRFallbackMaxDistanceNoItem = 10.0f;
        public float noSteamVRFallbackMaxDistanceWithItem = 0.5f;
        private float noSteamVRFallbackInteractorDistance = -1.0f;

        public GameObject renderModelPrefab;
        protected List<RenderModel> renderModels = new List<RenderModel
            >();
        protected RenderModel mainRenderModel;
        protected RenderModel hoverhighlightRenderModel;

        public bool showDebugText = false;
        public bool spewDebugText = false;
        public bool showDebugInteractables = false;

        public struct AttachedObject
        {
            public GameObject attachedObject;
            public Interactable interactable;
            public Rigidbody attachedRigidbody;
            public CollisionDetectionMode collisionDetectionMode;
            public bool attachedRigidbodyWasKinematic;
            public bool attachedRigidbodyUsedGravity;
            public GameObject originalParent;
            public bool isParentedToHand;
            public GrabTypes grabbedWithType;
            public AttachmentFlags attachmentFlags;
            public Vector3 initialPositionalOffset;
            public Quaternion initialRotationalOffset;
            public Transform attachedOffsetTransform;
            public Transform handAttachmentPointTransform;
            public Vector3 easeSourcePosition;
            public Quaternion easeSourceRotation;
            public float attachTime;

            public bool HasAttachFlag(AttachmentFlags flag)
            {
                return (attachmentFlags & flag) == flag;
            }
        }

        private List<AttachedObject> attachedObjects = new List<
            AttachedObject>();

        public ReadOnlyCollection<AttachedObject> AttachedObjects
        {
            get { return attachedObjects.AsReadOnly(); }
        }

        public bool hoverLocked { get; private set; }

        private Interactable _hoveringInteractable;
```

```

        private TextMesh debugText;
        private int prevOverlappingColliders = 0;

        private const int ColliderArraySize = 16;
        private Collider[] overlappingColliders;

        private Player playerInstance;

        private GameObject applicationLostFocusObject;

        private SteamVR_Events.Action inputFocusAction;

        public bool isActive
        {
            get
            {
                if (trackedObject != null)
                    return trackedObject.isActive;

                return this.gameObject.activeInHierarchy;
            }
        }

        public bool isPoseValid
        {
            get
            {
                return trackedObject.isValid;
            }
        }


        //-------------------------------------------------
        // The Interactable object this Hand is currently hovering over
        //-------------------------------------------------
        public Interactable hoveringInteractable
        {
            get { return _hoveringInteractable; }
            set
            {
                if (_hoveringInteractable != value)
                {
                    if (_hoveringInteractable != null)
                    {
                        if (spewDebugText)
                            HandDebugLog("HoverEnd " +
                                _hoveringInteractable.gameObject);
                        _hoveringInteractable.SendMessage("OnHandHoverEnd
                           ", this, SendMessageOptions.
                           DontRequireReceiver);

                        //Note: The _hoveringInteractable can change
                           after sending the OnHandHoverEnd message so
```

```
we need to check it again before broadcasting this message
                        if (_hoveringInteractable != null)
                        {
                            this.BroadcastMessage("OnParentHandHoverEnd", _hoveringInteractable, SendMessageOptions.DontRequireReceiver); // let objects attached to the hand know that a hover has ended
                        }
                    }

                    _hoveringInteractable = value;

                    if (_hoveringInteractable != null)
                    {
                        if (spewDebugText)
                            HandDebugLog("HoverBegin " + _hoveringInteractable.gameObject);
                        _hoveringInteractable.SendMessage("OnHandHoverBegin", this, SendMessageOptions.DontRequireReceiver);

                        //Note: The _hoveringInteractable can change after sending the OnHandHoverBegin message so we need to check it again before broadcasting this message
                        if (_hoveringInteractable != null)
                        {
                            this.BroadcastMessage("OnParentHandHoverBegin", _hoveringInteractable, SendMessageOptions.DontRequireReceiver); // let objects attached to the hand know that a hover has begun
                        }
                    }
                }
            }
        }


        //-------------------------------------------------
        // Active GameObject attached to this Hand
        //-------------------------------------------------
        public GameObject currentAttachedObject
        {
            get
            {
                CleanUpAttachedObjectStack();

                if (attachedObjects.Count > 0)
                {
                    return attachedObjects[attachedObjects.Count - 1].attachedObject;
```

```
                }

                return null;
            }
        }

        public AttachedObject? currentAttachedObjectInfo
        {
            get
            {
                CleanUpAttachedObjectStack();

                if (attachedObjects.Count > 0)
                {
                    return attachedObjects[attachedObjects.Count - 1];
                }

                return null;
            }
        }

        public SteamVR_Behaviour_Skeleton skeleton
        {
            get
            {
                if (mainRenderModel != null)
                    return mainRenderModel.GetSkeleton();

                return null;
            }
        }

        public void ShowController(bool permanent = false)
        {
            if (mainRenderModel != null)
                mainRenderModel.SetControllerVisibility(true, permanent);

            if (hoverhighlightRenderModel != null)
                hoverhighlightRenderModel.SetControllerVisibility(true, permanent);
        }

        public void HideController(bool permanent = false)
        {
            if (mainRenderModel != null)
                mainRenderModel.SetControllerVisibility(false, permanent);

            if (hoverhighlightRenderModel != null)
                hoverhighlightRenderModel.SetControllerVisibility(false, permanent);
        }

        public void ShowSkeleton(bool permanent = false)
```

```
        {
            if (mainRenderModel != null)
                mainRenderModel.SetHandVisibility(true, permanent);

            if (hoverhighlightRenderModel != null)
                hoverhighlightRenderModel.SetHandVisibility(true, permanent);
        }

        public void HideSkeleton(bool permanent = false)
        {
            if (mainRenderModel != null)
                mainRenderModel.SetHandVisibility(false, permanent);

            if (hoverhighlightRenderModel != null)
                hoverhighlightRenderModel.SetHandVisibility(false, permanent);
        }

        public bool HasSkeleton()
        {
            return mainRenderModel != null && mainRenderModel.GetSkeleton() != null;
        }

        public void Show()
        {
            SetVisibility(true);
        }

        public void Hide()
        {
            SetVisibility(false);
        }

        public void SetVisibility(bool visible)
        {
            if (mainRenderModel != null)
                mainRenderModel.SetVisibility(visible);
        }

        public void SetSkeletonRangeOfMotion(EVRSkeletalMotionRange newRangeOfMotion, float blendOverSeconds = 0.1f)
        {
            for (int renderModelIndex = 0; renderModelIndex < renderModels.Count; renderModelIndex++)
            {
                renderModels[renderModelIndex].SetSkeletonRangeOfMotion(newRangeOfMotion, blendOverSeconds);
            }
        }

        public void SetTemporarySkeletonRangeOfMotion(SkeletalMotionRangeChange temporaryRangeOfMotionChange, float
```

```
        blendOverSeconds = 0.1f)
{
    for (int renderModelIndex = 0; renderModelIndex <
        renderModels.Count; renderModelIndex++)
    {
        renderModels[renderModelIndex].
            SetTemporarySkeletonRangeOfMotion(
            temporaryRangeOfMotionChange, blendOverSeconds);
    }
}

public void ResetTemporarySkeletonRangeOfMotion(float
    blendOverSeconds = 0.1f)
{
    for (int renderModelIndex = 0; renderModelIndex <
        renderModels.Count; renderModelIndex++)
    {
        renderModels[renderModelIndex].
            ResetTemporarySkeletonRangeOfMotion(blendOverSeconds)
            ;
    }
}

public void SetAnimationState(int stateValue)
{
    for (int renderModelIndex = 0; renderModelIndex <
        renderModels.Count; renderModelIndex++)
    {
        renderModels[renderModelIndex].SetAnimationState(
            stateValue);
    }
}

public void StopAnimation()
{
    for (int renderModelIndex = 0; renderModelIndex <
        renderModels.Count; renderModelIndex++)
    {
        renderModels[renderModelIndex].StopAnimation();
    }
}


//-------------------------------------------------
// Attach a GameObject to this GameObject
//
// objectToAttach - The GameObject to attach
// flags - The flags to use for attaching the object
// attachmentPoint - Name of the GameObject in the hierarchy of
    this Hand which should act as the attachment point for this
    GameObject
//-------------------------------------------------
public void AttachObject(GameObject objectToAttach, GrabTypes
    grabbedWithType, AttachmentFlags flags =
```

```
            defaultAttachmentFlags, Transform attachmentOffset = null)
        {
            AttachedObject attachedObject = new AttachedObject();
            attachedObject.attachmentFlags = flags;
            attachedObject.attachedOffsetTransform = attachmentOffset;
            attachedObject.attachTime = Time.time;

            if (flags == 0)
            {
                flags = defaultAttachmentFlags;
            }

            //Make sure top object on stack is non-null
            CleanUpAttachedObjectStack();

            //Detach the object if it is already attached so that it can
                get re-attached at the top of the stack
            if(ObjectIsAttached(objectToAttach))
                DetachObject(objectToAttach);

            //Detach from the other hand if requested
            if (attachedObject.HasAttachFlag(AttachmentFlags.
                DetachFromOtherHand))
            {
                if (otherHand != null)
                    otherHand.DetachObject(objectToAttach);
            }

            if (attachedObject.HasAttachFlag(AttachmentFlags.DetachOthers
                ))
            {
                //Detach all the objects from the stack
                while (attachedObjects.Count > 0)
                {
                    DetachObject(attachedObjects[0].attachedObject);
                }
            }

            if (currentAttachedObject)
            {
                currentAttachedObject.SendMessage("OnHandFocusLost", this
                    , SendMessageOptions.DontRequireReceiver);
            }

            attachedObject.attachedObject = objectToAttach;
            attachedObject.interactable = objectToAttach.GetComponent<
                Interactable>();
            attachedObject.handAttachmentPointTransform = this.transform;

            if (attachedObject.interactable != null)
            {
                if (attachedObject.interactable.attachEaseIn)
                {
```

```
                attachedObject.easeSourcePosition = attachedObject.
                    attachedObject.transform.position;
                attachedObject.easeSourceRotation = attachedObject.
                    attachedObject.transform.rotation;
                attachedObject.interactable.snapAttachEaseInCompleted
                     = false;
            }

            if (attachedObject.interactable.
                useHandObjectAttachmentPoint)
                attachedObject.handAttachmentPointTransform =
                    objectAttachmentPoint;

            if (attachedObject.interactable.hideHandOnAttach)
                Hide();

            if (attachedObject.interactable.hideSkeletonOnAttach &&
                mainRenderModel != null && mainRenderModel.
                displayHandByDefault)
                HideSkeleton();

            if (attachedObject.interactable.hideControllerOnAttach &&
                 mainRenderModel != null && mainRenderModel.
                displayControllerByDefault)
                HideController();

            if (attachedObject.interactable.handAnimationOnPickup !=
                0)
                SetAnimationState(attachedObject.interactable.
                    handAnimationOnPickup);

            if (attachedObject.interactable.setRangeOfMotionOnPickup
                != SkeletalMotionRangeChange.None)
                SetTemporarySkeletonRangeOfMotion(attachedObject.
                    interactable.setRangeOfMotionOnPickup);

        }

        attachedObject.originalParent = objectToAttach.transform.
            parent != null ? objectToAttach.transform.parent.
            gameObject : null;

        attachedObject.attachedRigidbody = objectToAttach.
            GetComponent<Rigidbody>();
        if (attachedObject.attachedRigidbody != null)
        {
            if (attachedObject.interactable.attachedToHand != null)
                //already attached to another hand
            {
                //if it was attached to another hand, get the flags
                    from that hand

                for (int attachedIndex = 0; attachedIndex <
                    attachedObject.interactable.attachedToHand.
```

```
                        attachedObjects.Count; attachedIndex++)
                    {
                        AttachedObject attachedObjectInList = attachedObject.interactable.attachedToHand.attachedObjects[attachedIndex];
                        if (attachedObjectInList.interactable == attachedObject.interactable)
                        {
                            attachedObject.attachedRigidbodyWasKinematic = attachedObjectInList.attachedRigidbodyWasKinematic;
                            attachedObject.attachedRigidbodyUsedGravity = attachedObjectInList.attachedRigidbodyUsedGravity;
                            attachedObject.originalParent = attachedObjectInList.originalParent;
                        }
                    }
                }
                else
                {
                    attachedObject.attachedRigidbodyWasKinematic = attachedObject.attachedRigidbody.isKinematic;
                    attachedObject.attachedRigidbodyUsedGravity = attachedObject.attachedRigidbody.useGravity;
                }
            }

            attachedObject.grabbedWithType = grabbedWithType;

            if (attachedObject.HasAttachFlag(AttachmentFlags.ParentToHand))
            {
                //Parent the object to the hand
                objectToAttach.transform.parent = this.transform;
                attachedObject.isParentedToHand = true;
            }
            else
            {
                attachedObject.isParentedToHand = false;
            }

            if (attachedObject.HasAttachFlag(AttachmentFlags.SnapOnAttach))
            {
                if (attachedObject.interactable != null && attachedObject.interactable.skeletonPoser != null && HasSkeleton())
                {
                    SteamVR_Skeleton_PoseSnapshot pose = attachedObject.interactable.skeletonPoser.GetBlendedPose(skeleton);

                    //snap the object to the center of the attach point
```

```
                    objectToAttach.transform.position = this.transform.TransformPoint(pose.position);
                    objectToAttach.transform.rotation = this.transform.rotation * pose.rotation;

                    attachedObject.initialPositionalOffset = attachedObject.handAttachmentPointTransform.InverseTransformPoint(objectToAttach.transform.position);
                    attachedObject.initialRotationalOffset = Quaternion.Inverse(attachedObject.handAttachmentPointTransform.rotation) * objectToAttach.transform.rotation;
                }
                else
                {
                    if (attachmentOffset != null)
                    {
                        //offset the object from the hand by the positional and rotational difference between the offset transform and the attached object
                        Quaternion rotDiff = Quaternion.Inverse(attachmentOffset.transform.rotation) * objectToAttach.transform.rotation;
                        objectToAttach.transform.rotation = attachedObject.handAttachmentPointTransform.rotation * rotDiff;

                        Vector3 posDiff = objectToAttach.transform.position - attachmentOffset.transform.position;
                        objectToAttach.transform.position = attachedObject.handAttachmentPointTransform.position + posDiff;
                    }
                    else
                    {
                        //snap the object to the center of the attach point
                        objectToAttach.transform.rotation = attachedObject.handAttachmentPointTransform.rotation;
                        objectToAttach.transform.position = attachedObject.handAttachmentPointTransform.position;
                    }

                    Transform followPoint = objectToAttach.transform;

                    attachedObject.initialPositionalOffset = attachedObject.handAttachmentPointTransform.InverseTransformPoint(followPoint.position);
                    attachedObject.initialRotationalOffset = Quaternion.Inverse(attachedObject.
```

```
                handAttachmentPointTransform.rotation) * followPoint.rotation;
        }
    }
    else
    {
        if (attachedObject.interactable != null && attachedObject.interactable.skeletonPoser != null && HasSkeleton())
        {
            attachedObject.initialPositionalOffset = attachedObject.handAttachmentPointTransform.InverseTransformPoint(objectToAttach.transform.position);
            attachedObject.initialRotationalOffset = Quaternion.Inverse(attachedObject.handAttachmentPointTransform.rotation) * objectToAttach.transform.rotation;
        }
        else
        {
            if (attachmentOffset != null)
            {
                //get the initial positional and rotational offsets between the hand and the offset transform
                Quaternion rotDiff = Quaternion.Inverse(attachmentOffset.transform.rotation) * objectToAttach.transform.rotation;
                Quaternion targetRotation = attachedObject.handAttachmentPointTransform.rotation * rotDiff;
                Quaternion rotationPositionBy = targetRotation * Quaternion.Inverse(objectToAttach.transform.rotation);

                Vector3 posDiff = (rotationPositionBy * objectToAttach.transform.position) - (rotationPositionBy * attachmentOffset.transform.position);

                attachedObject.initialPositionalOffset = attachedObject.handAttachmentPointTransform.InverseTransformPoint(attachedObject.handAttachmentPointTransform.position + posDiff);
                attachedObject.initialRotationalOffset = Quaternion.Inverse(attachedObject.handAttachmentPointTransform.rotation) * (attachedObject.handAttachmentPointTransform.rotation * rotDiff);
            }
            else
            {
```

```
                        attachedObject.initialPositionalOffset = attachedObject.handAttachmentPointTransform.InverseTransformPoint(objectToAttach.transform.position);
                        attachedObject.initialRotationalOffset = Quaternion.Inverse(attachedObject.handAttachmentPointTransform.rotation) * objectToAttach.transform.rotation;
                    }
                }
            }



            if (attachedObject.HasAttachFlag(AttachmentFlags.TurnOnKinematic))
            {
                if (attachedObject.attachedRigidbody != null)
                {
                    attachedObject.collisionDetectionMode = attachedObject.attachedRigidbody.collisionDetectionMode;
                    if (attachedObject.collisionDetectionMode == CollisionDetectionMode.Continuous)
                        attachedObject.attachedRigidbody.collisionDetectionMode = CollisionDetectionMode.Discrete;

                    attachedObject.attachedRigidbody.isKinematic = true;
                }
            }

            if (attachedObject.HasAttachFlag(AttachmentFlags.TurnOffGravity))
            {
                if (attachedObject.attachedRigidbody != null)
                {
                    attachedObject.attachedRigidbody.useGravity = false;
                }
            }

            if (attachedObject.interactable != null && attachedObject.interactable.attachEaseIn)
            {
                attachedObject.attachedObject.transform.position = attachedObject.easeSourcePosition;
                attachedObject.attachedObject.transform.rotation = attachedObject.easeSourceRotation;
            }

            attachedObjects.Add(attachedObject);

            UpdateHovering();

```

```
            if (spewDebugText)
                HandDebugLog("AttachObject " + objectToAttach);
            objectToAttach.SendMessage("OnAttachedToHand", this, SendMessageOptions.DontRequireReceiver);
        }

        public bool ObjectIsAttached(GameObject go)
        {
            for (int attachedIndex = 0; attachedIndex < attachedObjects.Count; attachedIndex++)
            {
                if (attachedObjects[attachedIndex].attachedObject == go)
                    return true;
            }

            return false;
        }

        public void ForceHoverUnlock()
        {
            hoverLocked = false;
        }

        //-------------------------------------------------
        // Detach this GameObject from the attached object stack of this Hand
        //
        // objectToDetach - The GameObject to detach from this Hand
        //-------------------------------------------------
        public void DetachObject(GameObject objectToDetach, bool restoreOriginalParent = true)
        {
            int index = attachedObjects.FindIndex(l => l.attachedObject == objectToDetach);
            if (index != -1)
            {
                if (spewDebugText)
                    HandDebugLog("DetachObject " + objectToDetach);

                GameObject prevTopObject = currentAttachedObject;


                if (attachedObjects[index].interactable != null)
                {
                    if (attachedObjects[index].interactable.hideHandOnAttach)
                        Show();

                    if (attachedObjects[index].interactable.hideSkeletonOnAttach && mainRenderModel != null && mainRenderModel.displayHandByDefault)
                        ShowSkeleton();

```

```
                if (attachedObjects[index].interactable.hideControllerOnAttach && mainRenderModel != null && mainRenderModel.displayControllerByDefault)
                    ShowController();

                if (attachedObjects[index].interactable.handAnimationOnPickup != 0)
                    StopAnimation();

                if (attachedObjects[index].interactable.setRangeOfMotionOnPickup != SkeletalMotionRangeChange.None)
                    ResetTemporarySkeletonRangeOfMotion();
            }

            Transform parentTransform = null;
            if (attachedObjects[index].isParentedToHand)
            {
                if (restoreOriginalParent && (attachedObjects[index].originalParent != null))
                {
                    parentTransform = attachedObjects[index].originalParent.transform;
                }

                if (attachedObjects[index].attachedObject != null)
                {
                    attachedObjects[index].attachedObject.transform.parent = parentTransform;
                }
            }

            if (attachedObjects[index].HasAttachFlag(AttachmentFlags.TurnOnKinematic))
            {
                if (attachedObjects[index].attachedRigidbody != null)
                {
                    attachedObjects[index].attachedRigidbody.isKinematic = attachedObjects[index].attachedRigidbodyWasKinematic;
                    attachedObjects[index].attachedRigidbody.collisionDetectionMode = attachedObjects[index].collisionDetectionMode;
                }
            }

            if (attachedObjects[index].HasAttachFlag(AttachmentFlags.TurnOffGravity))
            {
                if (attachedObjects[index].attachedObject != null)
                {
                    if (attachedObjects[index].attachedRigidbody != null)
```

```
                        attachedObjects[index].attachedRigidbody.
                            useGravity = attachedObjects[index].
                            attachedRigidbodyUsedGravity;
                    }
                }

                if (attachedObjects[index].interactable != null &&
                    attachedObjects[index].interactable.
                    handFollowTransform && HasSkeleton())
                {
                    skeleton.transform.localPosition = Vector3.zero;
                    skeleton.transform.localRotation = Quaternion.
                        identity;
                }

                if (attachedObjects[index].attachedObject != null)
                {
                    if (attachedObjects[index].interactable == null || (
                        attachedObjects[index].interactable != null &&
                        attachedObjects[index].interactable.isDestroying
                        == false))
                        attachedObjects[index].attachedObject.SetActive(
                            true);

                    attachedObjects[index].attachedObject.SendMessage("
                        OnDetachedFromHand", this, SendMessageOptions.
                        DontRequireReceiver);
                }

                attachedObjects.RemoveAt(index);

                CleanUpAttachedObjectStack();

                GameObject newTopObject = currentAttachedObject;

                hoverLocked = false;


                //Give focus to the top most object on the stack if it
                    changed
                if (newTopObject != null && newTopObject != prevTopObject
                    )
                {
                    newTopObject.SetActive(true);
                    newTopObject.SendMessage("OnHandFocusAcquired", this,
                         SendMessageOptions.DontRequireReceiver);
                }
            }

            CleanUpAttachedObjectStack();

            if (mainRenderModel != null)
                mainRenderModel.MatchHandToTransform(mainRenderModel.
                    transform);
```

```
            if (hoverhighlightRenderModel != null)
                hoverhighlightRenderModel.MatchHandToTransform(hoverhighlightRenderModel.transform);
        }


        //-------------------------------------------------
        // Get the world velocity of the VR Hand.
        //-------------------------------------------------
        public Vector3 GetTrackedObjectVelocity(float timeOffset = 0)
        {
            if (trackedObject == null)
            {
                Vector3 velocityTarget, angularTarget;
                GetUpdatedAttachedVelocities(currentAttachedObjectInfo.Value, out velocityTarget, out angularTarget);
                return velocityTarget;
            }

                if (isActive)
            {
                if (timeOffset == 0)
                    return Player.instance.trackingOriginTransform.TransformVector(trackedObject.GetVelocity());
                else
                {
                    Vector3 velocity;
                    Vector3 angularVelocity;

                    trackedObject.GetVelocitiesAtTimeOffset(timeOffset, out velocity, out angularVelocity);
                    return Player.instance.trackingOriginTransform.TransformVector(velocity);
                }
            }

            return Vector3.zero;
        }


        //-------------------------------------------------
        // Get the world space angular velocity of the VR Hand.
        //-------------------------------------------------
        public Vector3 GetTrackedObjectAngularVelocity(float timeOffset = 0)
        {
            if (trackedObject == null)
            {
                Vector3 velocityTarget, angularTarget;
                GetUpdatedAttachedVelocities(currentAttachedObjectInfo.Value, out velocityTarget, out angularTarget);
                return angularTarget;
            }

```

```
            if (isActive)
            {
                if (timeOffset == 0)
                    return Player.instance.trackingOriginTransform.
                        TransformDirection(trackedObject.
                        GetAngularVelocity());
                else
                {
                    Vector3 velocity;
                    Vector3 angularVelocity;

                    trackedObject.GetVelocitiesAtTimeOffset(timeOffset,
                        out velocity, out angularVelocity);
                    return Player.instance.trackingOriginTransform.
                        TransformDirection(angularVelocity);
                }
            }

            return Vector3.zero;
        }

        public void GetEstimatedPeakVelocities(out Vector3 velocity, out
            Vector3 angularVelocity)
        {
            trackedObject.GetEstimatedPeakVelocities(out velocity, out
                angularVelocity);
            velocity = Player.instance.trackingOriginTransform.
                TransformVector(velocity);
            angularVelocity = Player.instance.trackingOriginTransform.
                TransformDirection(angularVelocity);
        }


        //-------------------------------------------------
        private void CleanUpAttachedObjectStack()
        {
            attachedObjects.RemoveAll(l => l.attachedObject == null);
        }


        //-------------------------------------------------
        protected virtual void Awake()
        {
            inputFocusAction = SteamVR_Events.InputFocusAction(
                OnInputFocus);

            if (hoverSphereTransform == null)
                hoverSphereTransform = this.transform;

            if (objectAttachmentPoint == null)
                objectAttachmentPoint = this.transform;

            applicationLostFocusObject = new GameObject("
                _application_lost_focus");
```

```
            applicationLostFocusObject.transform.parent = transform;
            applicationLostFocusObject.SetActive(false);

            if (trackedObject == null)
            {
                trackedObject = this.gameObject.GetComponent<
                    SteamVR_Behaviour_Pose>();

                if (trackedObject != null)
                    trackedObject.onTransformUpdatedEvent +=
                        OnTransformUpdated;
            }
        }

        protected virtual void OnDestroy()
        {
            if (trackedObject != null)
            {
                trackedObject.onTransformUpdatedEvent -=
                    OnTransformUpdated;
            }
        }

        protected virtual void OnTransformUpdated(SteamVR_Behaviour_Pose
            updatedPose, SteamVR_Input_Sources updatedSource)
        {
            HandFollowUpdate();
        }

        //-------------------------------------------------
        protected virtual IEnumerator Start()
        {
            // save off player instance
            playerInstance = Player.instance;
            if (!playerInstance)
            {
                Debug.LogError("<b>[SteamVR Interaction]</b> No player
                    instance found in Hand Start()");
            }

            // allocate array for colliders
            overlappingColliders = new Collider[ColliderArraySize];

            // We are a "no SteamVR fallback hand" if we have this camera
                set
            // we'll use the right mouse to look around and left mouse to
                interact
            // - don't need to find the device
            if (noSteamVRFallbackCamera)
            {
                yield break;
            }

```

```
            //Debug.Log( "<b>[SteamVR Interaction]</b> Hand - 
                initializing connection routine" );

            while (true)
            {
                if (isPoseValid)
                {
                    InitController();
                    break;
                }

                yield return null;
            }
        }


        //-------------------------------------------------
        protected virtual void UpdateHovering()
        {
            if ((noSteamVRFallbackCamera == null) && (isActive == false))
            {
                return;
            }

            if (hoverLocked)
                return;

            if (applicationLostFocusObject.activeSelf)
                return;

            float closestDistance = float.MaxValue;
            Interactable closestInteractable = null;

            if (useHoverSphere)
            {
                float scaledHoverRadius = hoverSphereRadius * Mathf.Abs(
                    SteamVR_Utils.GetLossyScale(hoverSphereTransform));
                CheckHoveringForTransform(hoverSphereTransform.position,
                    scaledHoverRadius, ref closestDistance, ref
                    closestInteractable, Color.green);
            }

            if (useControllerHoverComponent && mainRenderModel != null &&
                 mainRenderModel.IsControllerVisibile())
            {
                float scaledHoverRadius = controllerHoverRadius * Mathf.
                    Abs(SteamVR_Utils.GetLossyScale(this.transform));
                CheckHoveringForTransform(mainRenderModel.
                    GetControllerPosition(controllerHoverComponent),
                    scaledHoverRadius / 2f, ref closestDistance, ref
                    closestInteractable, Color.blue);
            }

```

```
            if (useFingerJointHover && mainRenderModel != null && mainRenderModel.IsHandVisibile())
            {
                float scaledHoverRadius = fingerJointHoverRadius * Mathf.Abs(SteamVR_Utils.GetLossyScale(this.transform));
                CheckHoveringForTransform(mainRenderModel.GetBonePosition((int)fingerJointHover), scaledHoverRadius / 2f, ref closestDistance, ref closestInteractable, Color.yellow);
            }

            // Hover on this one
            hoveringInteractable = closestInteractable;
        }

        protected virtual bool CheckHoveringForTransform(Vector3 hoverPosition, float hoverRadius, ref float closestDistance, ref Interactable closestInteractable, Color debugColor)
        {
            bool foundCloser = false;

            // null out old vals
            for (int i = 0; i < overlappingColliders.Length; ++i)
            {
                overlappingColliders[i] = null;
            }

            int numColliding = Physics.OverlapSphereNonAlloc(hoverPosition, hoverRadius, overlappingColliders, hoverLayerMask.value);

            if (numColliding == ColliderArraySize)
                Debug.LogWarning("<b>[SteamVR Interaction]</b> This hand is overlapping the max number of colliders: " + ColliderArraySize + ". Some collisions may be missed. Increase ColliderArraySize on Hand.cs");

            // DebugVar
            int iActualColliderCount = 0;

            // Pick the closest hovering
            for (int colliderIndex = 0; colliderIndex < overlappingColliders.Length; colliderIndex++)
            {
                Collider collider = overlappingColliders[colliderIndex];

                if (collider == null)
                    continue;

                Interactable contacting = collider.GetComponentInParent<Interactable>();

                // Yeah, it's null, skip
                if (contacting == null)
```

```
                    continue;

                // Ignore this collider for hovering
                IgnoreHovering ignore = collider.GetComponent<IgnoreHovering>();
                if (ignore != null)
                {
                    if (ignore.onlyIgnoreHand == null || ignore.onlyIgnoreHand == this)
                    {
                        continue;
                    }
                }

                // Can't hover over the object if it's attached
                bool hoveringOverAttached = false;
                for (int attachedIndex = 0; attachedIndex < attachedObjects.Count; attachedIndex++)
                {
                    if (attachedObjects[attachedIndex].attachedObject == contacting.gameObject)
                    {
                        hoveringOverAttached = true;
                        break;
                    }
                }
                if (hoveringOverAttached)
                    continue;

                // Occupied by another hand, so we can't touch it
                if (otherHand && otherHand.hoveringInteractable == contacting)
                    continue;

                // Best candidate so far...
                float distance = Vector3.Distance(contacting.transform.position, hoverPosition);
                if (distance < closestDistance)
                {
                    closestDistance = distance;
                    closestInteractable = contacting;
                    foundCloser = true;
                }
                iActualColliderCount++;
            }

            if (showDebugInteractables && foundCloser)
            {
                Debug.DrawLine(hoverPosition, closestInteractable.transform.position, debugColor, .05f, false);
            }

            if (iActualColliderCount > 0 && iActualColliderCount != prevOverlappingColliders)
```

```
            {
                prevOverlappingColliders = iActualColliderCount;

                if (spewDebugText)
                    HandDebugLog("Found " + iActualColliderCount + "
                        overlapping colliders.");
            }

            return foundCloser;
        }


        //-------------------------------------------------
        protected virtual void UpdateNoSteamVRFallback()
        {
            if (noSteamVRFallbackCamera)
            {
                Ray ray = noSteamVRFallbackCamera.ScreenPointToRay(Input.
                    mousePosition);

                if (attachedObjects.Count > 0)
                {
                    // Holding down the mouse:
                    // move around a fixed distance from the camera
                    transform.position = ray.origin +
                        noSteamVRFallbackInteractorDistance * ray.
                        direction;
                }
                else
                {
                    // Not holding down the mouse:
                    // cast out a ray to see what we should mouse over

                    // Don't want to hit the hand and anything underneath
                         it
                    // So move it back behind the camera when we do the
                        raycast
                    Vector3 oldPosition = transform.position;
                    transform.position = noSteamVRFallbackCamera.
                        transform.forward * (-1000.0f);

                    RaycastHit raycastHit;
                    if (Physics.Raycast(ray, out raycastHit,
                        noSteamVRFallbackMaxDistanceNoItem))
                    {
                        transform.position = raycastHit.point;

                        // Remember this distance in case we click and
                            drag the mouse
                        noSteamVRFallbackInteractorDistance = Mathf.Min(
                            noSteamVRFallbackMaxDistanceNoItem,
                            raycastHit.distance);
                    }
                    else if (noSteamVRFallbackInteractorDistance > 0.0f)
```

```
                    {
                        // Move it around at the distance we last had a hit
                        transform.position = ray.origin + Mathf.Min( noSteamVRFallbackMaxDistanceNoItem, noSteamVRFallbackInteractorDistance) * ray.direction;
                    }
                    else
                    {
                        // Didn't hit, just leave it where it was
                        transform.position = oldPosition;
                    }
                }
            }
        }


        //-------------------------------------------------
        private void UpdateDebugText()
        {
            if (showDebugText)
            {
                if (debugText == null)
                {
                    debugText = new GameObject("_debug_text").AddComponent<TextMesh>();
                    debugText.fontSize = 120;
                    debugText.characterSize = 0.001f;
                    debugText.transform.parent = transform;

                    debugText.transform.localRotation = Quaternion.Euler(90.0f, 0.0f, 0.0f);
                }

                if (handType == SteamVR_Input_Sources.RightHand)
                {
                    debugText.transform.localPosition = new Vector3(-0.05f, 0.0f, 0.0f);
                    debugText.alignment = TextAlignment.Right;
                    debugText.anchor = TextAnchor.UpperRight;
                }
                else
                {
                    debugText.transform.localPosition = new Vector3(0.05f, 0.0f, 0.0f);
                    debugText.alignment = TextAlignment.Left;
                    debugText.anchor = TextAnchor.UpperLeft;
                }

                debugText.text = string.Format(
                    "Hovering: {0}\n" +
                    "Hover Lock: {1}\n" +
                    "Attached: {2}\n" +
```

```
                    "Total Attached: {3}\n" +
                    "Type: {4}\n",
                    (hoveringInteractable ? hoveringInteractable.
                        gameObject.name : "null"),
                    hoverLocked,
                    (currentAttachedObject ? currentAttachedObject.name :
                         "null"),
                    attachedObjects.Count,
                    handType.ToString());
            }
            else
            {
                if (debugText != null)
                {
                    Destroy(debugText.gameObject);
                }
            }
        }


        //-------------------------------------------------
        protected virtual void OnEnable()
        {
            inputFocusAction.enabled = true;

            // Stagger updates between hands
            float hoverUpdateBegin = ((otherHand != null) && (otherHand.
                GetInstanceID() < GetInstanceID())) ? (0.5f *
                hoverUpdateInterval) : (0.0f);
            InvokeRepeating("UpdateHovering", hoverUpdateBegin,
                hoverUpdateInterval);
            InvokeRepeating("UpdateDebugText", hoverUpdateBegin,
                hoverUpdateInterval);
        }


        //-------------------------------------------------
        protected virtual void OnDisable()
        {
            inputFocusAction.enabled = false;

            CancelInvoke();
        }


        //-------------------------------------------------
        protected virtual void Update()
        {
            UpdateNoSteamVRFallback();

            GameObject attachedObject = currentAttachedObject;
            if (attachedObject != null)
            {
```

```
                attachedObject.SendMessage("HandAttachedUpdate", this, SendMessageOptions.DontRequireReceiver);
            }

            if (hoveringInteractable)
            {
                hoveringInteractable.SendMessage("HandHoverUpdate", this, SendMessageOptions.DontRequireReceiver);
            }
        }

        /// <summary>
        /// Returns true when the hand is currently hovering over the interactable passed in
        /// </summary>
        public bool IsStillHovering(Interactable interactable)
        {
            return hoveringInteractable == interactable;
        }

        protected virtual void HandFollowUpdate()
        {
            GameObject attachedObject = currentAttachedObject;
            if (attachedObject != null)
            {
                if (currentAttachedObjectInfo.Value.interactable != null)
                {
                    SteamVR_Skeleton_PoseSnapshot pose = null;

                    if (currentAttachedObjectInfo.Value.interactable.skeletonPoser != null && HasSkeleton())
                    {
                        pose = currentAttachedObjectInfo.Value.interactable.skeletonPoser.GetBlendedPose(skeleton);
                    }

                    if (currentAttachedObjectInfo.Value.interactable.handFollowTransform)
                    {
                        Quaternion targetHandRotation;
                        Vector3 targetHandPosition;

                        if (pose == null)
                        {
                            Quaternion offset = Quaternion.Inverse(this.transform.rotation) * currentAttachedObjectInfo.Value.handAttachmentPointTransform.rotation;
                            targetHandRotation = currentAttachedObjectInfo.Value.interactable.transform.rotation * Quaternion.Inverse(offset);

```

```
                    Vector3 worldOffset = (this.transform.position - currentAttachedObjectInfo.Value.handAttachmentPointTransform.position);
                    Quaternion rotationDiff = mainRenderModel.GetHandRotation() * Quaternion.Inverse(this.transform.rotation);
                    Vector3 localOffset = rotationDiff * worldOffset;
                    targetHandPosition = currentAttachedObjectInfo.Value.interactable.transform.position + localOffset;
                }
                else
                {
                    Transform objectT = currentAttachedObjectInfo.Value.attachedObject.transform;
                    Vector3 oldItemPos = objectT.position;
                    Quaternion oldItemRot = objectT.transform.rotation;
                    objectT.position = TargetItemPosition(currentAttachedObjectInfo.Value);
                    objectT.rotation = TargetItemRotation(currentAttachedObjectInfo.Value);
                    Vector3 localSkelePos = objectT.InverseTransformPoint(transform.position);
                    Quaternion localSkeleRot = Quaternion.Inverse(objectT.rotation) * transform.rotation;
                    objectT.position = oldItemPos;
                    objectT.rotation = oldItemRot;

                    targetHandPosition = objectT.TransformPoint(localSkelePos);
                    targetHandRotation = objectT.rotation * localSkeleRot;
                }

                if (mainRenderModel != null)
                    mainRenderModel.SetHandRotation(targetHandRotation);
                if (hoverhighlightRenderModel != null)
                    hoverhighlightRenderModel.SetHandRotation(targetHandRotation);

                if (mainRenderModel != null)
                    mainRenderModel.SetHandPosition(targetHandPosition);
                if (hoverhighlightRenderModel != null)
                    hoverhighlightRenderModel.SetHandPosition(targetHandPosition);
            }
        }
```

```
            }
        }

        protected virtual void FixedUpdate()
        {
            if (currentAttachedObject != null)
            {
                AttachedObject attachedInfo = currentAttachedObjectInfo.
                    Value;
                if (attachedInfo.attachedObject != null)
                {
                    if (attachedInfo.HasAttachFlag(AttachmentFlags.
                        VelocityMovement))
                    {
                        if(attachedInfo.interactable.attachEaseIn ==
                            false || attachedInfo.interactable.
                            snapAttachEaseInCompleted)
                            UpdateAttachedVelocity(attachedInfo);

                        /*if (attachedInfo.interactable.
                            handFollowTransformPosition)
                        {
                            skeleton.transform.position =
                                TargetSkeletonPosition(attachedInfo);
                            skeleton.transform.rotation = attachedInfo.
                                attachedObject.transform.rotation *
                                attachedInfo.skeletonLockRotation;
                        }*/
                    }else
                    {
                        if (attachedInfo.HasAttachFlag(AttachmentFlags.
                            ParentToHand))
                        {
                            attachedInfo.attachedObject.transform.
                                position = TargetItemPosition(
                                attachedInfo);
                            attachedInfo.attachedObject.transform.
                                rotation = TargetItemRotation(
                                attachedInfo);
                        }
                    }


                    if (attachedInfo.interactable.attachEaseIn)
                    {
                        float t = Util.RemapNumberClamped(Time.time,
                            attachedInfo.attachTime, attachedInfo.
                            attachTime + attachedInfo.interactable.
                            snapAttachEaseInTime, 0.0f, 1.0f);
                        if (t < 1.0f)
                        {
                            if (attachedInfo.HasAttachFlag(
                                AttachmentFlags.VelocityMovement))
                            {
```

```
                                attachedInfo.attachedRigidbody.velocity =
                                     Vector3.zero;
                                attachedInfo.attachedRigidbody.
                                    angularVelocity = Vector3.zero;
                            }
                            t = attachedInfo.interactable.
                                snapAttachEaseInCurve.Evaluate(t);
                            attachedInfo.attachedObject.transform.
                                position = Vector3.Lerp(attachedInfo.
                                easeSourcePosition, TargetItemPosition(
                                attachedInfo), t);
                            attachedInfo.attachedObject.transform.
                                rotation = Quaternion.Lerp(attachedInfo.
                                easeSourceRotation, TargetItemRotation(
                                attachedInfo), t);
                        }
                        else if (!attachedInfo.interactable.
                            snapAttachEaseInCompleted)
                        {
                            attachedInfo.interactable.gameObject.
                                SendMessage("
                                OnThrowableAttachEaseInCompleted", this,
                                SendMessageOptions.DontRequireReceiver);
                            attachedInfo.interactable.
                                snapAttachEaseInCompleted = true;
                        }
                    }
                }
            }
        }

        protected const float MaxVelocityChange = 10f;
        protected const float VelocityMagic = 6000f;
        protected const float AngularVelocityMagic = 50f;
        protected const float MaxAngularVelocityChange = 20f;

        protected void UpdateAttachedVelocity(AttachedObject
            attachedObjectInfo)
        {
            Vector3 velocityTarget, angularTarget;
            bool success = GetUpdatedAttachedVelocities(
                attachedObjectInfo, out velocityTarget, out angularTarget
                );
            if (success)
            {
                float scale = SteamVR_Utils.GetLossyScale(
                    currentAttachedObjectInfo.Value.
                    handAttachmentPointTransform);
                float maxAngularVelocityChange = MaxAngularVelocityChange
                     * scale;
                float maxVelocityChange = MaxVelocityChange * scale;

                attachedObjectInfo.attachedRigidbody.velocity = Vector3.
                    MoveTowards(attachedObjectInfo.attachedRigidbody.
```

```
                    velocity, velocityTarget, maxVelocityChange);
                attachedObjectInfo.attachedRigidbody.angularVelocity =
                    Vector3.MoveTowards(attachedObjectInfo.
                    attachedRigidbody.angularVelocity, angularTarget,
                    maxAngularVelocityChange);
            }
        }

        protected Vector3 TargetItemPosition(AttachedObject
            attachedObject)
        {
            if (attachedObject.interactable != null && attachedObject.
                interactable.skeletonPoser != null && HasSkeleton())
            {
                Vector3 tp = attachedObject.handAttachmentPointTransform.
                    InverseTransformPoint(transform.TransformPoint(
                    attachedObject.interactable.skeletonPoser.
                    GetBlendedPose(skeleton).position));
                //tp.x *= -1;
                return currentAttachedObjectInfo.Value.
                    handAttachmentPointTransform.TransformPoint(tp);
            }
            else
            {
                return currentAttachedObjectInfo.Value.
                    handAttachmentPointTransform.TransformPoint(
                    attachedObject.initialPositionalOffset);
            }
        }

        protected Quaternion TargetItemRotation(AttachedObject
            attachedObject)
        {
            if (attachedObject.interactable != null && attachedObject.
                interactable.skeletonPoser != null && HasSkeleton())
            {
                Quaternion tr = Quaternion.Inverse(attachedObject.
                    handAttachmentPointTransform.rotation) * (transform.
                    rotation * attachedObject.interactable.skeletonPoser.
                    GetBlendedPose(skeleton).rotation);
                return currentAttachedObjectInfo.Value.
                    handAttachmentPointTransform.rotation * tr;
            }
            else
            {
                return currentAttachedObjectInfo.Value.
                    handAttachmentPointTransform.rotation *
                    attachedObject.initialRotationalOffset;
            }
        }

        protected bool GetUpdatedAttachedVelocities(AttachedObject
            attachedObjectInfo, out Vector3 velocityTarget, out Vector3
            angularTarget)
```

```
        {
            bool realNumbers = false;


            float velocityMagic = VelocityMagic;
            float angularVelocityMagic = AngularVelocityMagic;

            Vector3 targetItemPosition = TargetItemPosition(
                attachedObjectInfo);
            Vector3 positionDelta = (targetItemPosition -
                attachedObjectInfo.attachedRigidbody.position);
            velocityTarget = (positionDelta * velocityMagic * Time.
                deltaTime);

            if (float.IsNaN(velocityTarget.x) == false && float.
                IsInfinity(velocityTarget.x) == false)
            {
                if (noSteamVRFallbackCamera)
                    velocityTarget /= 10; //hacky fix for fallback

                realNumbers = true;
            }
            else
                velocityTarget = Vector3.zero;


            Quaternion targetItemRotation = TargetItemRotation(
                attachedObjectInfo);
            Quaternion rotationDelta = targetItemRotation * Quaternion.
                Inverse(attachedObjectInfo.attachedObject.transform.
                rotation);


            float angle;
            Vector3 axis;
            rotationDelta.ToAngleAxis(out angle, out axis);

            if (angle > 180)
                angle -= 360;

            if (angle != 0 && float.IsNaN(axis.x) == false && float.
                IsInfinity(axis.x) == false)
            {
                angularTarget = angle * axis * angularVelocityMagic *
                    Time.deltaTime;

                if (noSteamVRFallbackCamera)
                    angularTarget /= 10; //hacky fix for fallback

                realNumbers &= true;
            }
            else
                angularTarget = Vector3.zero;

```

```
            return realNumbers;
        }


        //-------------------------------------------------
        protected virtual void OnInputFocus(bool hasFocus)
        {
            if (hasFocus)
            {
                DetachObject(applicationLostFocusObject, true);
                applicationLostFocusObject.SetActive(false);
                UpdateHovering();
                BroadcastMessage("OnParentHandInputFocusAcquired",
                    SendMessageOptions.DontRequireReceiver);
            }
            else
            {
                applicationLostFocusObject.SetActive(true);
                AttachObject(applicationLostFocusObject, GrabTypes.
                    Scripted, AttachmentFlags.ParentToHand);
                BroadcastMessage("OnParentHandInputFocusLost",
                    SendMessageOptions.DontRequireReceiver);
            }
        }

        //-------------------------------------------------
        protected virtual void OnDrawGizmos()
        {
            if (useHoverSphere && hoverSphereTransform != null)
            {
                Gizmos.color = Color.green;
                float scaledHoverRadius = hoverSphereRadius * Mathf.Abs(
                    SteamVR_Utils.GetLossyScale(hoverSphereTransform));
                Gizmos.DrawWireSphere(hoverSphereTransform.position,
                    scaledHoverRadius/2);
            }

            if (useControllerHoverComponent && mainRenderModel != null &&
                 mainRenderModel.IsControllerVisibile())
            {
                Gizmos.color = Color.blue;
                float scaledHoverRadius = controllerHoverRadius * Mathf.
                    Abs(SteamVR_Utils.GetLossyScale(this.transform));
                Gizmos.DrawWireSphere(mainRenderModel.
                    GetControllerPosition(controllerHoverComponent),
                    scaledHoverRadius/2);
            }

            if (useFingerJointHover && mainRenderModel != null &&
                mainRenderModel.IsHandVisibile())
            {
                Gizmos.color = Color.yellow;
                float scaledHoverRadius = fingerJointHoverRadius * Mathf.
                    Abs(SteamVR_Utils.GetLossyScale(this.transform));
```

```
                Gizmos.DrawWireSphere(mainRenderModel.GetBonePosition((
                    int)fingerJointHover), scaledHoverRadius/2);
            }
        }


        //-------------------------------------------------
        private void HandDebugLog(string msg)
        {
            if (spewDebugText)
            {
                Debug.Log("<b>[SteamVR Interaction]</b> Hand (" + this.
                    name + "): " + msg);
            }
        }


        //-------------------------------------------------
        // Continue to hover over this object indefinitely, whether or
            not the Hand moves out of its interaction trigger volume.
        //
        // interactable - The Interactable to hover over indefinitely.
        //-------------------------------------------------
        public void HoverLock(Interactable interactable)
        {
            if (spewDebugText)
                HandDebugLog("HoverLock " + interactable);
            hoverLocked = true;
            hoveringInteractable = interactable;
        }


        //-------------------------------------------------
        // Stop hovering over this object indefinitely.
        //
        // interactable - The hover-locked Interactable to stop hovering
            over indefinitely.
        //-------------------------------------------------
        public void HoverUnlock(Interactable interactable)
        {
            if (spewDebugText)
                HandDebugLog("HoverUnlock " + interactable);

            if (hoveringInteractable == interactable)
            {
                hoverLocked = false;
            }
        }

        public void TriggerHapticPulse(ushort microSecondsDuration)
        {
            float seconds = (float)microSecondsDuration / 1000000f;
            hapticAction.Execute(0, seconds, 1f / seconds, 1, handType);
        }
```

```
        public void TriggerHapticPulse(float duration, float frequency, float amplitude)
        {
            hapticAction.Execute(0, duration, frequency, amplitude, handType);
        }

        public void ShowGrabHint()
        {
            ControllerButtonHints.ShowButtonHint(this, grabGripAction); //todo: assess
        }

        public void HideGrabHint()
        {
            ControllerButtonHints.HideButtonHint(this, grabGripAction); //todo: assess
        }

        public void ShowGrabHint(string text)
        {
            ControllerButtonHints.ShowTextHint(this, grabGripAction, text);
        }

        public GrabTypes GetGrabStarting(GrabTypes explicitType = GrabTypes.None)
        {
            if (explicitType != GrabTypes.None)
            {
                if (noSteamVRFallbackCamera)
                {
                    if (Input.GetMouseButtonDown(0))
                        return explicitType;
                }

                if (explicitType == GrabTypes.Pinch && grabPinchAction.GetStateDown(handType))
                    return GrabTypes.Pinch;
                if (explicitType == GrabTypes.Grip && grabGripAction.GetStateDown(handType))
                    return GrabTypes.Grip;
            }
            else
            {
                if (noSteamVRFallbackCamera)
                {
                    if (Input.GetMouseButtonDown(0))
                        return GrabTypes.Grip;
                }

                if (grabPinchAction.GetStateDown(handType))
                    return GrabTypes.Pinch;
```

```
                if (grabGripAction.GetStateDown(handType))
                    return GrabTypes.Grip;
            }

            return GrabTypes.None;
        }

        public GrabTypes GetGrabEnding(GrabTypes explicitType = GrabTypes.None)
        {
            if (explicitType != GrabTypes.None)
            {
                if (noSteamVRFallbackCamera)
                {
                    if (Input.GetMouseButtonUp(0))
                        return explicitType;
                }

                if (explicitType == GrabTypes.Pinch && grabPinchAction.GetStateUp(handType))
                    return GrabTypes.Pinch;
                if (explicitType == GrabTypes.Grip && grabGripAction.GetStateUp(handType))
                    return GrabTypes.Grip;
            }
            else
            {
                if (noSteamVRFallbackCamera)
                {
                    if (Input.GetMouseButtonUp(0))
                        return GrabTypes.Grip;
                }

                if (grabPinchAction.GetStateUp(handType))
                    return GrabTypes.Pinch;
                if (grabGripAction.GetStateUp(handType))
                    return GrabTypes.Grip;
            }

            return GrabTypes.None;
        }

        public bool IsGrabEnding(GameObject attachedObject)
        {
            for (int attachedObjectIndex = 0; attachedObjectIndex < attachedObjects.Count; attachedObjectIndex++)
            {
                if (attachedObjects[attachedObjectIndex].attachedObject == attachedObject)
                {
                    return IsGrabbingWithType(attachedObjects[attachedObjectIndex].grabbedWithType) == false;
                }
            }
```

```
            return false;
        }

        public bool IsGrabbingWithType(GrabTypes type)
        {
            if (noSteamVRFallbackCamera)
            {
                if (Input.GetMouseButton(0))
                    return true;
            }

            switch (type)
            {
                case GrabTypes.Pinch:
                    return grabPinchAction.GetState(handType);

                case GrabTypes.Grip:
                    return grabGripAction.GetState(handType);

                default:
                    return false;
            }
        }

        public bool IsGrabbingWithOppositeType(GrabTypes type)
        {
            if (noSteamVRFallbackCamera)
            {
                if (Input.GetMouseButton(0))
                    return true;
            }

            switch (type)
            {
                case GrabTypes.Pinch:
                    return grabGripAction.GetState(handType);

                case GrabTypes.Grip:
                    return grabPinchAction.GetState(handType);

                default:
                    return false;
            }
        }

        public GrabTypes GetBestGrabbingType()
        {
            return GetBestGrabbingType(GrabTypes.None);
        }

        public GrabTypes GetBestGrabbingType(GrabTypes preferred, bool
            forcePreference = false)
        {
```

```
            if (noSteamVRFallbackCamera)
            {
                if (Input.GetMouseButton(0))
                    return preferred;
            }

            if (preferred == GrabTypes.Pinch)
            {
                if (grabPinchAction.GetState(handType))
                    return GrabTypes.Pinch;
                else if (forcePreference)
                    return GrabTypes.None;
            }
            if (preferred == GrabTypes.Grip)
            {
                if (grabGripAction.GetState(handType))
                    return GrabTypes.Grip;
                else if (forcePreference)
                    return GrabTypes.None;
            }

            if (grabPinchAction.GetState(handType))
                return GrabTypes.Pinch;
            if (grabGripAction.GetState(handType))
                return GrabTypes.Grip;

            return GrabTypes.None;
        }


        //-------------------------------------------------
        private void InitController()
        {
            if (spewDebugText)
                HandDebugLog("Hand " + name + " connected with type " +
                    handType.ToString());

            bool hadOldRendermodel = mainRenderModel != null;
            EVRSkeletalMotionRange oldRM_rom = EVRSkeletalMotionRange.
                WithController;
            if(hadOldRendermodel)
                oldRM_rom = mainRenderModel.GetSkeletonRangeOfMotion;


            foreach (RenderModel r in renderModels)
            {
                if (r != null)
                    Destroy(r.gameObject);
            }

            renderModels.Clear();

            GameObject renderModelInstance = GameObject.Instantiate(
                renderModelPrefab);
```

```
            renderModelInstance.layer = gameObject.layer;
            renderModelInstance.tag = gameObject.tag;
            renderModelInstance.transform.parent = this.transform;
            renderModelInstance.transform.localPosition = Vector3.zero;
            renderModelInstance.transform.localRotation = Quaternion.identity;
            renderModelInstance.transform.localScale = renderModelPrefab.transform.localScale;

            //TriggerHapticPulse(800);  //pulse on controller init

            int deviceIndex = trackedObject.GetDeviceIndex();

            mainRenderModel = renderModelInstance.GetComponent<RenderModel>();
            renderModels.Add(mainRenderModel);

            if (hadOldRendermodel)
                mainRenderModel.SetSkeletonRangeOfMotion(oldRM_rom);

            this.BroadcastMessage("SetInputSource", handType, SendMessageOptions.DontRequireReceiver); // let child objects know we've initialized
            this.BroadcastMessage("OnHandInitialized", deviceIndex, SendMessageOptions.DontRequireReceiver); // let child objects know we've initialized
        }

        public void SetRenderModel(GameObject prefab)
        {
            renderModelPrefab = prefab;

            if (mainRenderModel != null && isPoseValid)
                InitController();
        }

        public void SetHoverRenderModel(RenderModel hoverRenderModel)
        {
            hoverhighlightRenderModel = hoverRenderModel;
            renderModels.Add(hoverRenderModel);
        }

        public int GetDeviceIndex()
        {
            return trackedObject.GetDeviceIndex();
        }
    }


    [System.Serializable]
    public class HandEvent : UnityEvent<Hand> { }


#if UNITY_EDITOR
```

```
    //
    -------------------------------------------------------------------------
    [UnityEditor.CustomEditor(typeof(Hand))]
    public class HandEditor : UnityEditor.Editor
    {
        //-------------------------------------------------
        // Custom Inspector GUI allows us to click from within the UI
        //-------------------------------------------------
        public override void OnInspectorGUI()
        {
            DrawDefaultInspector();

            /*
            Hand hand = (Hand)target;

            if (hand.otherHand)
            {
                if (hand.otherHand.otherHand != hand)
                {
                    UnityEditor.EditorGUILayout.HelpBox("The otherHand of
                         this Hand's otherHand is not this Hand.",
                        UnityEditor.MessageType.Warning);
                }

                if (hand.handType == SteamVR_Input_Sources.LeftHand &&
                    hand.otherHand && hand.otherHand.handType !=
                    SteamVR_Input_Sources.RightHand)
                {
                    UnityEditor.EditorGUILayout.HelpBox("This is a left
                        Hand but otherHand is not a right Hand.",
                        UnityEditor.MessageType.Warning);
                }

                if (hand.handType == SteamVR_Input_Sources.RightHand &&
                    hand.otherHand && hand.otherHand.handType !=
                    SteamVR_Input_Sources.LeftHand)
                {
                    UnityEditor.EditorGUILayout.HelpBox("This is a right
                        Hand but otherHand is not a left Hand.",
                        UnityEditor.MessageType.Warning);
                }

                if (hand.handType == SteamVR_Input_Sources.Any && hand.
                    otherHand && hand.otherHand.handType !=
                    SteamVR_Input_Sources.Any)
                {
                    UnityEditor.EditorGUILayout.HelpBox("This is an any-
                        handed Hand but otherHand is not an any-handed
                        Hand.", UnityEditor.MessageType.Warning);
                }
            }
            */ //removing for now because it conflicts with other input
                sources (trackers and such)
```

```
        }
    }
#endif
}
```

9.5.5. IgnoreTeleportTrace.cs

```
//======= Copyright (c) Valve Corporation, All rights reserved.
    ===============
//
// Purpose: Allows the teleport arc trace to pass through any colliders
    on this
//      object
//
//
    =============================================================================

using UnityEngine;
using System.Collections;

namespace Valve.VR.InteractionSystem
{
  //
    -------------------------------------------------------------------------
  public class IgnoreTeleportTrace : MonoBehaviour
  {
  }
}
```

9.5.6. InputModule.cs

```
//======= Copyright (c) Valve Corporation, All rights reserved.
    ===============
//
// Purpose: Makes the hand act as an input module for Unity's event
    system
//
//
    =============================================================================

using UnityEngine;
using System.Collections;
using UnityEngine.EventSystems;

namespace Valve.VR.InteractionSystem
{
```

```
//
    -----------------------------------------------------------------------
public class InputModule : BaseInputModule
{
  private GameObject submitObject;

  //-------------------------------------------------
  private static InputModule _instance;
  public static InputModule instance
  {
    get
    {
      if ( _instance == null )
        _instance = GameObject.FindObjectOfType<InputModule>();

      return _instance;
    }
  }


  //-------------------------------------------------
  public override bool ShouldActivateModule()
  {
    if ( !base.ShouldActivateModule() )
      return false;

    return submitObject != null;
  }


  //-------------------------------------------------
  public void HoverBegin( GameObject gameObject )
  {
    PointerEventData pointerEventData = new PointerEventData(
        eventSystem );
    ExecuteEvents.Execute( gameObject, pointerEventData, ExecuteEvents.
        pointerEnterHandler );
  }


  //-------------------------------------------------
  public void HoverEnd( GameObject gameObject )
  {
    PointerEventData pointerEventData = new PointerEventData(
        eventSystem );
    pointerEventData.selectedObject = null;
    ExecuteEvents.Execute( gameObject, pointerEventData, ExecuteEvents.
        pointerExitHandler );
  }


  //-------------------------------------------------
  public void Submit( GameObject gameObject )
```

```
    {
      submitObject = gameObject;
    }


    //-------------------------------------------------
    public override void Process()
    {
      if ( submitObject )
      {
        BaseEventData data = GetBaseEventData();
        data.selectedObject = submitObject;
        ExecuteEvents.Execute( submitObject, data, ExecuteEvents.
            submitHandler );

        submitObject = null;
      }
    }
  }
}
```

9.5.7. Interactable.cs

```
//======= Copyright (c) Valve Corporation, All rights reserved.
    ===============
//
// Purpose: This object will get hover events and can be attached to the
    hands
//
//
    =============================================================================

using UnityEngine;
using UnityEngine.Events;
using System.Collections;
using System.Collections.Generic;

namespace Valve.VR.InteractionSystem
{
    //
        -------------------------------------------------------------------------
    public class Interactable : MonoBehaviour
    {
        [Tooltip("Activates an action set on attach and deactivates on
            detach")]
        public SteamVR_ActionSet activateActionSetOnAttach;

        [Tooltip("Hide the whole hand on attachment and show on detach")]
        public bool hideHandOnAttach = true;

```

```
        [Tooltip("Hide the skeleton part of the hand on attachment and
            show on detach")]
        public bool hideSkeletonOnAttach = false;

        [Tooltip("Hide the controller part of the hand on attachment and
            show on detach")]
        public bool hideControllerOnAttach = false;

        [Tooltip("The integer in the animator to trigger on pickup. 0 for
             none")]
        public int handAnimationOnPickup = 0;

        [Tooltip("The range of motion to set on the skeleton. None for no
             change.")]
        public SkeletalMotionRangeChange setRangeOfMotionOnPickup =
            SkeletalMotionRangeChange.None;

        public delegate void OnAttachedToHandDelegate(Hand hand);
        public delegate void OnDetachedFromHandDelegate(Hand hand);

        public event OnAttachedToHandDelegate onAttachedToHand;
        public event OnDetachedFromHandDelegate onDetachedFromHand;


        [Tooltip("Specify whether you want to snap to the hand's object
            attachment point, or just the raw hand")]
        public bool useHandObjectAttachmentPoint = true;

        public bool attachEaseIn = false;
        [HideInInspector]
        public AnimationCurve snapAttachEaseInCurve = AnimationCurve.
            EaseInOut(0.0f, 0.0f, 1.0f, 1.0f);
        public float snapAttachEaseInTime = 0.15f;

        public bool snapAttachEaseInCompleted = false;


        // [Tooltip("The skeleton pose to apply when grabbing. Can only
            set this or handFollowTransform.")]
        [HideInInspector]
        public SteamVR_Skeleton_Poser skeletonPoser;

        [Tooltip("Should the rendered hand lock on to and follow the
            object")]
        public bool handFollowTransform= true;


        [Tooltip("Set whether or not you want this interactible to
            highlight when hovering over it")]
        public bool highlightOnHover = true;
        protected MeshRenderer[] highlightRenderers;
        protected MeshRenderer[] existingRenderers;
        protected GameObject highlightHolder;
        protected SkinnedMeshRenderer[] highlightSkinnedRenderers;
```

```
        protected SkinnedMeshRenderer[] existingSkinnedRenderers;
        protected static Material highlightMat;
        [Tooltip("An array of child gameObjects to not render a highlight for. Things like transparent parts, vfx, etc.")]
        public GameObject[] hideHighlight;


        [System.NonSerialized]
        public Hand attachedToHand;

        [System.NonSerialized]
        public Hand hoveringHand;

        public bool isDestroying { get; protected set; }
        public bool isHovering { get; protected set; }
        public bool wasHovering { get; protected set; }


        private void Awake()
        {
            skeletonPoser = GetComponent<SteamVR_Skeleton_Poser>();
        }

        protected virtual void Start()
        {
            highlightMat = (Material)Resources.Load("SteamVR_HoverHighlight", typeof(Material));

            if (highlightMat == null)
                Debug.LogError("<b>[SteamVR Interaction]</b> Hover Highlight Material is missing. Please create a material named 'SteamVR_HoverHighlight' and place it in a Resources folder");

            if (skeletonPoser != null)
            {
                if (useHandObjectAttachmentPoint)
                {
                    //Debug.LogWarning("<b>[SteamVR Interaction]</b> SkeletonPose and useHandObjectAttachmentPoint both set at the same time. Ignoring useHandObjectAttachmentPoint.");
                    useHandObjectAttachmentPoint = false;
                }
            }
        }

        protected virtual bool ShouldIgnoreHighlight(Component component)
        {
            return ShouldIgnore(component.gameObject);
        }

        protected virtual bool ShouldIgnore(GameObject check)
        {
```

```
            for (int ignoreIndex = 0; ignoreIndex < hideHighlight.Length; ignoreIndex++)
            {
                if (check == hideHighlight[ignoreIndex])
                    return true;
            }

            return false;
        }

        protected virtual void CreateHighlightRenderers()
        {
            existingSkinnedRenderers = this.GetComponentsInChildren<SkinnedMeshRenderer>(true);
            highlightHolder = new GameObject("Highlighter");
            highlightSkinnedRenderers = new SkinnedMeshRenderer[existingSkinnedRenderers.Length];

            for (int skinnedIndex = 0; skinnedIndex < existingSkinnedRenderers.Length; skinnedIndex++)
            {
                SkinnedMeshRenderer existingSkinned = existingSkinnedRenderers[skinnedIndex];

                if (ShouldIgnoreHighlight(existingSkinned))
                    continue;

                GameObject newSkinnedHolder = new GameObject("SkinnedHolder");
                newSkinnedHolder.transform.parent = highlightHolder.transform;
                SkinnedMeshRenderer newSkinned = newSkinnedHolder.AddComponent<SkinnedMeshRenderer>();
                Material[] materials = new Material[existingSkinned.sharedMaterials.Length];
                for (int materialIndex = 0; materialIndex < materials.Length; materialIndex++)
                {
                    materials[materialIndex] = highlightMat;
                }

                newSkinned.sharedMaterials = materials;
                newSkinned.sharedMesh = existingSkinned.sharedMesh;
                newSkinned.rootBone = existingSkinned.rootBone;
                newSkinned.updateWhenOffscreen = existingSkinned.updateWhenOffscreen;
                newSkinned.bones = existingSkinned.bones;

                highlightSkinnedRenderers[skinnedIndex] = newSkinned;
            }

            MeshFilter[] existingFilters = this.GetComponentsInChildren<MeshFilter>(true);
            existingRenderers = new MeshRenderer[existingFilters.Length];
```

```
            highlightRenderers = new MeshRenderer[existingFilters.Length];

            for (int filterIndex = 0; filterIndex < existingFilters.Length; filterIndex++)
            {
                MeshFilter existingFilter = existingFilters[filterIndex];
                MeshRenderer existingRenderer = existingFilter.GetComponent<MeshRenderer>();

                if (existingFilter == null || existingRenderer == null || ShouldIgnoreHighlight(existingFilter))
                    continue;

                GameObject newFilterHolder = new GameObject("FilterHolder");
                newFilterHolder.transform.parent = highlightHolder.transform;
                MeshFilter newFilter = newFilterHolder.AddComponent<MeshFilter>();
                newFilter.sharedMesh = existingFilter.sharedMesh;
                MeshRenderer newRenderer = newFilterHolder.AddComponent<MeshRenderer>();

                Material[] materials = new Material[existingRenderer.sharedMaterials.Length];
                for (int materialIndex = 0; materialIndex < materials.Length; materialIndex++)
                {
                    materials[materialIndex] = highlightMat;
                }
                newRenderer.sharedMaterials = materials;

                highlightRenderers[filterIndex] = newRenderer;
                existingRenderers[filterIndex] = existingRenderer;
            }
        }

        protected virtual void UpdateHighlightRenderers()
        {
            if (highlightHolder == null)
                return;

            for (int skinnedIndex = 0; skinnedIndex < existingSkinnedRenderers.Length; skinnedIndex++)
            {
                SkinnedMeshRenderer existingSkinned = existingSkinnedRenderers[skinnedIndex];
                SkinnedMeshRenderer highlightSkinned = highlightSkinnedRenderers[skinnedIndex];

                if (existingSkinned != null && highlightSkinned != null && attachedToHand == false)
                {
```

```
                    highlightSkinned.transform.position = existingSkinned.transform.position;
                    highlightSkinned.transform.rotation = existingSkinned.transform.rotation;
                    highlightSkinned.transform.localScale = existingSkinned.transform.lossyScale;
                    highlightSkinned.localBounds = existingSkinned.localBounds;
                    highlightSkinned.enabled = isHovering && existingSkinned.enabled && existingSkinned.gameObject.activeInHierarchy;

                    int blendShapeCount = existingSkinned.sharedMesh.blendShapeCount;
                    for (int blendShapeIndex = 0; blendShapeIndex < blendShapeCount; blendShapeIndex++)
                    {
                        highlightSkinned.SetBlendShapeWeight(blendShapeIndex, existingSkinned.GetBlendShapeWeight(blendShapeIndex));
                    }
                }
                else if (highlightSkinned != null)
                    highlightSkinned.enabled = false;

            }

            for (int rendererIndex = 0; rendererIndex < highlightRenderers.Length; rendererIndex++)
            {
                MeshRenderer existingRenderer = existingRenderers[rendererIndex];
                MeshRenderer highlightRenderer = highlightRenderers[rendererIndex];

                if (existingRenderer != null && highlightRenderer != null && attachedToHand == false)
                {
                    highlightRenderer.transform.position = existingRenderer.transform.position;
                    highlightRenderer.transform.rotation = existingRenderer.transform.rotation;
                    highlightRenderer.transform.localScale = existingRenderer.transform.lossyScale;
                    highlightRenderer.enabled = isHovering && existingRenderer.enabled && existingRenderer.gameObject.activeInHierarchy;
                }
                else if (highlightRenderer != null)
                    highlightRenderer.enabled = false;
            }
        }

        /// <summary>
```

```
        /// Called when a Hand starts hovering over this object
        /// </summary>
        protected virtual void OnHandHoverBegin(Hand hand)
        {
            wasHovering = isHovering;
            isHovering = true;

            hoveringHand = hand;

            if (highlightOnHover == true)
            {
                CreateHighlightRenderers();
                UpdateHighlightRenderers();
            }
        }


        /// <summary>
        /// Called when a Hand stops hovering over this object
        /// </summary>
        private void OnHandHoverEnd(Hand hand)
        {
            wasHovering = isHovering;
            isHovering = false;

            if (highlightOnHover && highlightHolder != null)
                Destroy(highlightHolder);
        }

        protected virtual void Update()
        {
            if (highlightOnHover)
            {
                UpdateHighlightRenderers();

                if (isHovering == false && highlightHolder != null)
                    Destroy(highlightHolder);
            }
        }


        protected float blendToPoseTime = 0.1f;
        protected float releasePoseBlendTime = 0.2f;

        protected virtual void OnAttachedToHand(Hand hand)
        {
            if (activateActionSetOnAttach != null)
                activateActionSetOnAttach.Activate(hand.handType);

            if (onAttachedToHand != null)
            {
                onAttachedToHand.Invoke(hand);
            }

```

```
            if (skeletonPoser != null && hand.skeleton != null)
            {
                hand.skeleton.BlendToPoser(skeletonPoser, blendToPoseTime
                    );
            }

            attachedToHand = hand;
        }

        protected virtual void OnDetachedFromHand(Hand hand)
        {
            if (activateActionSetOnAttach != null)
            {
                if (hand.otherHand == null || hand.otherHand.
                    currentAttachedObjectInfo.HasValue == false ||
                    (hand.otherHand.currentAttachedObjectInfo.Value.
                        interactable != null &&
                     hand.otherHand.currentAttachedObjectInfo.Value.
                         interactable.activateActionSetOnAttach != this.
                         activateActionSetOnAttach))
                {
                    activateActionSetOnAttach.Deactivate(hand.handType);
                }
            }

            if (onDetachedFromHand != null)
            {
                onDetachedFromHand.Invoke(hand);
            }


            if (skeletonPoser != null)
            {
                if (hand.skeleton != null)
                    hand.skeleton.BlendToSkeleton(releasePoseBlendTime);
            }

            attachedToHand = null;
        }

        protected virtual void OnDestroy()
        {
            isDestroying = true;

            if (attachedToHand != null)
            {
                attachedToHand.DetachObject(this.gameObject, false);
                attachedToHand.skeleton.BlendToSkeleton(0.1f);
            }

            if (highlightHolder != null)
                Destroy(highlightHolder);

        }
```

```
        protected virtual void OnDisable()
        {
            isDestroying = true;

            if (attachedToHand != null)
            {
                attachedToHand.ForceHoverUnlock();
            }

            if (highlightHolder != null)
                Destroy(highlightHolder);
        }
    }
}
```

9.5.8. Player.cs

```
//======= Copyright (c) Valve Corporation, All rights reserved.
    ===============
//
// Purpose: Player interface used to query HMD transforms and VR hands
//
//
    =============================================================================

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

namespace Valve.VR.InteractionSystem
{
  //
      -------------------------------------------------------------------------
  // Singleton representing the local VR player/user, with methods for
      getting
  // the player's hands, head, tracking origin, and guesses for various
      properties.
  //
      -------------------------------------------------------------------------
  public class Player : MonoBehaviour
  {
    [Tooltip( "Virtual transform corresponding to the meatspace tracking
        origin. Devices are tracked relative to this." )]
    public Transform trackingOriginTransform;

    [Tooltip( "List of possible transforms for the head/HMD, including
        the no-SteamVR fallback camera." )]
```

```
    public Transform[] hmdTransforms;

    [Tooltip( "List of possible Hands, including no-SteamVR fallback Hands." )]
    public Hand[] hands;

    [Tooltip( "Reference to the physics collider that follows the player's HMD position." )]
    public Collider headCollider;

    [Tooltip( "These objects are enabled when SteamVR is available" )]
    public GameObject rigSteamVR;

    [Tooltip( "These objects are enabled when SteamVR is not available, or when the user toggles out of VR" )]
    public GameObject rig2DFallback;

    [Tooltip( "The audio listener for this player" )]
    public Transform audioListener;

        [Tooltip("This action lets you know when the player has placed the headset on their head")]
        public SteamVR_Action_Boolean headsetOnHead = SteamVR_Input.GetBooleanAction("HeadsetOnHead");

    public bool allowToggleTo2D = true;


    //-------------------------------------------------
    // Singleton instance of the Player. Only one can exist at a time.
    //-------------------------------------------------
    private static Player _instance;
    public static Player instance
    {
      get
      {
        if ( _instance == null )
        {
          _instance = FindObjectOfType<Player>();
        }
        return _instance;
      }
    }


    //-------------------------------------------------
    // Get the number of active Hands.
    //-------------------------------------------------
    public int handCount
    {
      get
      {
        int count = 0;
        for ( int i = 0; i < hands.Length; i++ )
```

```
        {
          if ( hands[i].gameObject.activeInHierarchy )
          {
            count++;
          }
        }
        return count;
      }
    }


    //-------------------------------------------------
    // Get the i-th active Hand.
    //
    // i - Zero-based index of the active Hand to get
    //-------------------------------------------------
    public Hand GetHand( int i )
    {
      for ( int j = 0; j < hands.Length; j++ )
      {
        if ( !hands[j].gameObject.activeInHierarchy )
        {
          continue;
        }

        if ( i > 0 )
        {
          i--;
          continue;
        }

        return hands[j];
      }

      return null;
    }


    //-------------------------------------------------
    public Hand leftHand
    {
      get
      {
        for ( int j = 0; j < hands.Length; j++ )
        {
          if ( !hands[j].gameObject.activeInHierarchy )
          {
            continue;
          }

          if ( hands[j].handType != SteamVR_Input_Sources.LeftHand)
          {
            continue;
          }
```

```
          return hands[j];
        }

        return null;
      }
    }


    //-------------------------------------------------
    public Hand rightHand
    {
      get
      {
        for ( int j = 0; j < hands.Length; j++ )
        {
          if ( !hands[j].gameObject.activeInHierarchy )
          {
            continue;
          }

          if ( hands[j].handType != SteamVR_Input_Sources.RightHand)
          {
            continue;
          }

          return hands[j];
        }

        return null;
      }
    }

        //-------------------------------------------------
        // Get Player scale. Assumes it is scaled equally on all axes.
        //-------------------------------------------------

        public float scale
        {
            get
            {
                return transform.lossyScale.x;
            }
        }


        //-------------------------------------------------
        // Get the HMD transform. This might return the fallback camera transform if SteamVR is unavailable or disabled.
        //-------------------------------------------------
        public Transform hmdTransform
    {
      get
      {
```

```
                if (hmdTransforms != null)
                {
                    for (int i = 0; i < hmdTransforms.Length; i++)
                    {
                        if (hmdTransforms[i].gameObject.activeInHierarchy)
                            return hmdTransforms[i];
                    }
                }
            return null;
        }
    }


    //-------------------------------------------------
    // Height of the eyes above the ground - useful for estimating player height.
    //-------------------------------------------------
    public float eyeHeight
    {
        get
        {
            Transform hmd = hmdTransform;
            if ( hmd )
            {
                Vector3 eyeOffset = Vector3.Project( hmd.position - trackingOriginTransform.position, trackingOriginTransform.up );
                return eyeOffset.magnitude / trackingOriginTransform.lossyScale.x;
            }
            return 0.0f;
        }
    }


    //-------------------------------------------------
    // Guess for the world-space position of the player's feet, directly beneath the HMD.
    //-------------------------------------------------
    public Vector3 feetPositionGuess
    {
        get
        {
            Transform hmd = hmdTransform;
            if ( hmd )
            {
                return trackingOriginTransform.position + Vector3.ProjectOnPlane( hmd.position - trackingOriginTransform.position, trackingOriginTransform.up );
            }
            return trackingOriginTransform.position;
        }
    }
```

```


    //-------------------------------------------------
    // Guess for the world-space direction of the player's hips/torso. This is effectively just the gaze direction projected onto the floor plane.
    //-------------------------------------------------
    public Vector3 bodyDirectionGuess
    {
      get
      {
        Transform hmd = hmdTransform;
        if ( hmd )
        {
          Vector3 direction = Vector3.ProjectOnPlane( hmd.forward, trackingOriginTransform.up );
          if ( Vector3.Dot( hmd.up, trackingOriginTransform.up ) < 0.0f )
          {
            // The HMD is upside-down. Either
            // -The player is bending over backwards
            // -The player is bent over looking through their legs
            direction = -direction;
          }
          return direction;
        }
        return trackingOriginTransform.forward;
      }
    }


    //-------------------------------------------------
    private void Awake()
    {
      if ( trackingOriginTransform == null )
      {
        trackingOriginTransform = this.transform;
      }
    }


    //-------------------------------------------------
    private IEnumerator Start()
    {
      _instance = this;

            while (SteamVR.initializedState == SteamVR.InitializedStates.None || SteamVR.initializedState == SteamVR.InitializedStates.Initializing)
                yield return null;

      if ( SteamVR.instance != null )
      {
        ActivateRig( rigSteamVR );
      }
```

```
      else
      {
#if !HIDE_DEBUG_UI
        ActivateRig( rig2DFallback );
#endif
      }
        }

        protected virtual void Update()
        {
            if (SteamVR.initializedState != SteamVR.InitializedStates.InitializeSuccess)
                return;

            if (headsetOnHead != null)
            {
                if (headsetOnHead.GetStateDown(SteamVR_Input_Sources.Head))
                {
                    Debug.Log("<b>SteamVR Interaction System</b> Headset placed on head");
                }
                else if (headsetOnHead.GetStateUp(SteamVR_Input_Sources.Head))
                {
                    Debug.Log("<b>SteamVR Interaction System</b> Headset removed");
                }
            }
        }

    //-------------------------------------------------
    void OnDrawGizmos()
    {
      if ( this != instance )
      {
        return;
      }

      //NOTE: These gizmo icons don't work in the plugin since the icons need to exist in a specific "Gizmos"
      //    folder in your Asset tree. These icons are included under Core/Icons. Moving them into a
      //    "Gizmos" folder should make them work again.

      Gizmos.color = Color.white;
      Gizmos.DrawIcon( feetPositionGuess, "vr_interaction_system_feet.png" );

      Gizmos.color = Color.cyan;
      Gizmos.DrawLine( feetPositionGuess, feetPositionGuess + trackingOriginTransform.up * eyeHeight );

      // Body direction arrow
```

```
      Gizmos.color = Color.blue;
      Vector3 bodyDirection = bodyDirectionGuess;
      Vector3 bodyDirectionTangent = Vector3.Cross(
          trackingOriginTransform.up, bodyDirection );
      Vector3 startForward = feetPositionGuess + trackingOriginTransform.
          up * eyeHeight * 0.75f;
      Vector3 endForward = startForward + bodyDirection * 0.33f;
      Gizmos.DrawLine( startForward, endForward );
      Gizmos.DrawLine( endForward, endForward - 0.033f * ( bodyDirection
          + bodyDirectionTangent ) );
      Gizmos.DrawLine( endForward, endForward - 0.033f * ( bodyDirection
          - bodyDirectionTangent ) );

      Gizmos.color = Color.red;
      int count = handCount;
      for ( int i = 0; i < count; i++ )
      {
        Hand hand = GetHand( i );

        if ( hand.handType == SteamVR_Input_Sources.LeftHand)
        {
          Gizmos.DrawIcon( hand.transform.position, "
              vr_interaction_system_left_hand.png" );
        }
        else if ( hand.handType == SteamVR_Input_Sources.RightHand)
        {
          Gizmos.DrawIcon( hand.transform.position, "
              vr_interaction_system_right_hand.png" );
        }
        else
        {
                    /*
          Hand.HandType guessHandType = hand.currentHandType;

          if ( guessHandType == Hand.HandType.Left )
          {
            Gizmos.DrawIcon( hand.transform.position, "
                vr_interaction_system_left_hand_question.png" );
          }
          else if ( guessHandType == Hand.HandType.Right )
          {
            Gizmos.DrawIcon( hand.transform.position, "
                vr_interaction_system_right_hand_question.png" );
          }
          else
          {
            Gizmos.DrawIcon( hand.transform.position, "
                vr_interaction_system_unknown_hand.png" );
          }
                    */
        }
      }
    }

```

```

    //------------------------------------------------
    public void Draw2DDebug()
    {
      if ( !allowToggleTo2D )
        return;

      if ( !SteamVR.active )
        return;

      int width = 100;
      int height = 25;
      int left = Screen.width / 2 - width / 2;
      int top = Screen.height - height - 10;

      string text = ( rigSteamVR.activeSelf ) ? "2D Debug" : "VR";

      if ( GUI.Button( new Rect( left, top, width, height ), text ) )
      {
        if ( rigSteamVR.activeSelf )
        {
          ActivateRig( rig2DFallback );
        }
        else
        {
          ActivateRig( rigSteamVR );
        }
      }
    }


    //------------------------------------------------
    private void ActivateRig( GameObject rig )
    {
      rigSteamVR.SetActive( rig == rigSteamVR );
      rig2DFallback.SetActive( rig == rig2DFallback );

      if ( audioListener )
      {
        audioListener.transform.parent = hmdTransform;
        audioListener.transform.localPosition = Vector3.zero;
        audioListener.transform.localRotation = Quaternion.identity;
      }
    }


    //------------------------------------------------
    public void PlayerShotSelf()
    {
      //Do something appropriate here
    }
  }
}
```

9.5.9. SteamVR_ ActivateActionSetOnLoad.cs

```
//======= Copyright (c) Valve Corporation, All rights reserved.
    ===============

using UnityEngine;
using System.Collections;

namespace Valve.VR
{
    /// <summary>
    /// Automatically activates an action set on Start() and deactivates
        the set on OnDestroy(). Optionally deactivating all other sets as
         well.
    /// </summary>
    public class SteamVR_ActivateActionSetOnLoad : MonoBehaviour
    {
        public SteamVR_ActionSet actionSet = SteamVR_Input.GetActionSet("
            default");

        public SteamVR_Input_Sources forSources = SteamVR_Input_Sources.
            Any;

        public bool disableAllOtherActionSets = false;

        public bool activateOnStart = true;
        public bool deactivateOnDestroy = true;


        private void Start()
        {
            if (actionSet != null && activateOnStart)
            {
                //Debug.Log(string.Format("[SteamVR] Activating {0}
                    action set.", actionSet.fullPath));
                actionSet.Activate(forSources, 0,
                    disableAllOtherActionSets);
            }
        }

        private void OnDestroy()
        {
            if (actionSet != null && deactivateOnDestroy)
            {
                //Debug.Log(string.Format("[SteamVR] Deactivating {0}
                    action set.", actionSet.fullPath));
                actionSet.Deactivate(forSources);
            }
        }
    }
}
```

9.5.10. SteamVR_ Behaviour.cs

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

#if UNITY_2017_2_OR_NEWER
    using UnityEngine.XR;
#else
using XRSettings = UnityEngine.VR.VRSettings;
using XRDevice = UnityEngine.VR.VRDevice;
#endif

namespace Valve.VR
{
    public class SteamVR_Behaviour : MonoBehaviour
    {
        private const string openVRDeviceName = "OpenVR";
        public static bool forcingInitialization = false;

        private static SteamVR_Behaviour _instance;
        public static SteamVR_Behaviour instance
        {
            get
            {
                if (_instance == null)
                {
                    Initialize(false);
                }

                return _instance;
            }
        }

        public bool initializeSteamVROnAwake = true;

        public bool doNotDestroy = true;

        [HideInInspector]
        public SteamVR_Render steamvr_render;


        private static bool initializing = false;
        public static void Initialize(bool forceUnityVRToOpenVR = false)
        {
            if (_instance == null && initializing == false)
            {
                initializing = true;
                GameObject steamVRObject = null;

                if (forceUnityVRToOpenVR)
```

```
                forcingInitialization = true;

            SteamVR_Render renderInstance = GameObject.FindObjectOfType<SteamVR_Render>();
            if (renderInstance != null)
                steamVRObject = renderInstance.gameObject;

            SteamVR_Behaviour behaviourInstance = GameObject.FindObjectOfType<SteamVR_Behaviour>();
            if (behaviourInstance != null)
                steamVRObject = behaviourInstance.gameObject;

            if (steamVRObject == null)
            {
                GameObject objectInstance = new GameObject("[SteamVR]");
                _instance = objectInstance.AddComponent<SteamVR_Behaviour>();
                _instance.steamvr_render = objectInstance.AddComponent<SteamVR_Render>();
            }
            else
            {
                behaviourInstance = steamVRObject.GetComponent<SteamVR_Behaviour>();
                if (behaviourInstance == null)
                    behaviourInstance = steamVRObject.AddComponent<SteamVR_Behaviour>();

                if (renderInstance != null)
                    behaviourInstance.steamvr_render = renderInstance;
                else
                {
                    behaviourInstance.steamvr_render = steamVRObject.GetComponent<SteamVR_Render>();
                    if (behaviourInstance.steamvr_render == null)
                        behaviourInstance.steamvr_render = steamVRObject.AddComponent<SteamVR_Render>();
                }

                _instance = behaviourInstance;
            }

            if (behaviourInstance != null && behaviourInstance.doNotDestroy)
                GameObject.DontDestroyOnLoad(behaviourInstance.transform.root.gameObject);

            initializing = false;
        }
    }

```

```
        protected void Awake()
        {
            if (initializeSteamVROnAwake && forcingInitialization ==
                false)
                InitializeSteamVR();
        }

        public void InitializeSteamVR(bool forceUnityVRToOpenVR = false)
        {
            if (forceUnityVRToOpenVR)
            {
                forcingInitialization = true;

                if (initializeCoroutine != null)
                    StopCoroutine(initializeCoroutine);

                if (XRSettings.loadedDeviceName == openVRDeviceName)
                    EnableOpenVR();
                else
                    initializeCoroutine = StartCoroutine(
                        DoInitializeSteamVR(forceUnityVRToOpenVR));
            }
            else
            {
                SteamVR.Initialize(false);
            }
        }

        private Coroutine initializeCoroutine;

#if UNITY_2018_3_OR_NEWER
        private bool loadedOpenVRDeviceSuccess = false;
        private IEnumerator DoInitializeSteamVR(bool forceUnityVRToOpenVR
             = false)
        {
            XRDevice.deviceLoaded += XRDevice_deviceLoaded;
            XRSettings.LoadDeviceByName(openVRDeviceName);
            while (loadedOpenVRDeviceSuccess == false)
            {
                yield return null;
            }
            XRDevice.deviceLoaded -= XRDevice_deviceLoaded;
            EnableOpenVR();
        }

        private void XRDevice_deviceLoaded(string deviceName)
        {
            if (deviceName == openVRDeviceName)
            {
                loadedOpenVRDeviceSuccess = true;
            }
            else
            {
```

```
                Debug.LogError("<b>[SteamVR]</b> Tried to async load: " +
                     openVRDeviceName + ". Loaded: " + deviceName);
                loadedOpenVRDeviceSuccess = true; //try anyway
            }
        }
#else
        private IEnumerator DoInitializeSteamVR(bool forceUnityVRToOpenVR
             = false)
        {
            XRSettings.LoadDeviceByName(openVRDeviceName);
            yield return null;
            EnableOpenVR();
        }
#endif

        private void EnableOpenVR()
        {
            XRSettings.enabled = true;
            SteamVR.Initialize(false);
            initializeCoroutine = null;
            forcingInitialization = false;
        }

#if UNITY_2017_1_OR_NEWER
        protected void OnEnable()
        {
        Application.onBeforeRender += OnBeforeRender;
            SteamVR_Events.System(EVREventType.VREvent_Quit).Listen(
                OnQuit);
        }
        protected void OnDisable()
        {
        Application.onBeforeRender -= OnBeforeRender;
            SteamVR_Events.System(EVREventType.VREvent_Quit).Remove(
                OnQuit);
        }
      protected void OnBeforeRender()
        {
            PreCull();
        }
#else
        protected void OnEnable()
        {
            Camera.onPreCull += OnCameraPreCull;
            SteamVR_Events.System(EVREventType.VREvent_Quit).Listen(
                OnQuit);
        }
        protected void OnDisable()
        {
            Camera.onPreCull -= OnCameraPreCull;
            SteamVR_Events.System(EVREventType.VREvent_Quit).Remove(
                OnQuit);
        }
        protected void OnCameraPreCull(Camera cam)
```

```
        {
            if (!cam.stereoEnabled)
                return;

            PreCull();
        }
#endif

        protected static int lastFrameCount = -1;
        protected void PreCull()
        {
            // Only update poses on the first camera per frame.
            if (Time.frameCount != lastFrameCount)
            {
                lastFrameCount = Time.frameCount;

                SteamVR_Input.OnPreCull();
            }
        }

        protected void FixedUpdate()
        {
            SteamVR_Input.FixedUpdate();
        }

        protected void LateUpdate()
        {
            SteamVR_Input.LateUpdate();
        }

        protected void Update()
        {
            SteamVR_Input.Update();
        }

        protected void OnQuit(VREvent_t vrEvent)
        {
#if UNITY_EDITOR
            UnityEditor.EditorApplication.isPlaying = false;
#else
        Application.Quit();
#endif
        }
    }
}
```

9.5.11. SteamVR_ Behaviour_ Pose.cs

```
//======= Copyright (c) Valve Corporation, All rights reserved.
    ===============

using System;
```

```
using System.Threading;
using UnityEngine;
using UnityEngine.Events;
using Valve.VR;

namespace Valve.VR
{
    /// <summary>
    /// This component simplifies the use of Pose actions. Adding it to a gameobject will auto set that transform's position and rotation every update to match the pose.
    /// Advanced velocity estimation is handled through a buffer of the last 30 updates.
    /// </summary>
    public class SteamVR_Behaviour_Pose : MonoBehaviour
    {
        public SteamVR_Action_Pose poseAction = SteamVR_Input.GetAction<SteamVR_Action_Pose>("Pose");

        [Tooltip("The device this action should apply to. Any if the action is not device specific.")]
        public SteamVR_Input_Sources inputSource;

        [Tooltip("If not set, relative to parent")]
        public Transform origin;

        /// <summary>Returns whether or not the current pose is in a valid state</summary>
        public bool isValid { get { return poseAction[inputSource].poseIsValid; } }

        /// <summary>Returns whether or not the pose action is bound and able to be updated</summary>
        public bool isActive { get { return poseAction[inputSource].active; } }


        /// <summary>This Unity event will fire whenever the position or rotation of this transform is updated.</summary>
        public SteamVR_Behaviour_PoseEvent onTransformUpdated;

        /// <summary>This Unity event will fire whenever the position or rotation of this transform is changed.</summary>
        public SteamVR_Behaviour_PoseEvent onTransformChanged;

        /// <summary>This Unity event will fire whenever the device is connected or disconnected</summary>
        public SteamVR_Behaviour_Pose_ConnectedChangedEvent onConnectedChanged;

        /// <summary>This Unity event will fire whenever the device's tracking state changes</summary>
        public SteamVR_Behaviour_Pose_TrackingChangedEvent onTrackingChanged;
```

```
        /// <summary>This Unity event will fire whenever the device's deviceIndex changes</summary>
        public SteamVR_Behaviour_Pose_DeviceIndexChangedEvent onDeviceIndexChanged;


        /// <summary>This C# event will fire whenever the position or rotation of this transform is updated.</summary>
        public UpdateHandler onTransformUpdatedEvent;

        /// <summary>This C# event will fire whenever the position or rotation of this transform is changed.</summary>
        public ChangeHandler onTransformChangedEvent;

        /// <summary>This C# event will fire whenever the device is connected or disconnected</summary>
        public DeviceConnectedChangeHandler onConnectedChangedEvent;

        /// <summary>This C# event will fire whenever the device's tracking state changes</summary>
        public TrackingChangeHandler onTrackingChangedEvent;

        /// <summary>This C# event will fire whenever the device's deviceIndex changes</summary>
        public DeviceIndexChangedHandler onDeviceIndexChangedEvent;


        [Tooltip("Can be disabled to stop broadcasting bound device status changes")]
        public bool broadcastDeviceChanges = true;

        protected int deviceIndex = -1;

        protected SteamVR_HistoryBuffer historyBuffer = new SteamVR_HistoryBuffer(30);


        protected virtual void Start()
        {
            if (poseAction == null)
            {
                Debug.LogError("<b>[SteamVR]</b> No pose action set for this component");
                return;
            }

            CheckDeviceIndex();

            if (origin == null)
                origin = this.transform.parent;
        }

        protected virtual void OnEnable()
```

```
        {
            SteamVR.Initialize();

            if (poseAction != null)
            {
                poseAction[inputSource].onUpdate +=
                    SteamVR_Behaviour_Pose_OnUpdate;
                poseAction[inputSource].onDeviceConnectedChanged +=
                    OnDeviceConnectedChanged;
                poseAction[inputSource].onTrackingChanged +=
                    OnTrackingChanged;
                poseAction[inputSource].onChange +=
                    SteamVR_Behaviour_Pose_OnChange;
            }
        }

        protected virtual void OnDisable()
        {
            if (poseAction != null)
            {
                poseAction[inputSource].onUpdate -=
                    SteamVR_Behaviour_Pose_OnUpdate;
                poseAction[inputSource].onDeviceConnectedChanged -=
                    OnDeviceConnectedChanged;
                poseAction[inputSource].onTrackingChanged -=
                    OnTrackingChanged;
                poseAction[inputSource].onChange -=
                    SteamVR_Behaviour_Pose_OnChange;
            }

            historyBuffer.Clear();
        }

        private void SteamVR_Behaviour_Pose_OnUpdate(SteamVR_Action_Pose
            fromAction, SteamVR_Input_Sources fromSource)
        {
            UpdateHistoryBuffer();

            UpdateTransform();

            if (onTransformUpdated != null)
                onTransformUpdated.Invoke(this, inputSource);
            if (onTransformUpdatedEvent != null)
                onTransformUpdatedEvent.Invoke(this, inputSource);
        }
        protected virtual void UpdateTransform()
        {
            CheckDeviceIndex();

            if (origin != null)
            {
                transform.position = origin.transform.TransformPoint(
                    poseAction[inputSource].localPosition);
```

```
                transform.rotation = origin.rotation * poseAction[inputSource].localRotation;
            }
            else
            {
                transform.localPosition = poseAction[inputSource].localPosition;
                transform.localRotation = poseAction[inputSource].localRotation;
            }
        }

        private void SteamVR_Behaviour_Pose_OnChange(SteamVR_Action_Pose fromAction, SteamVR_Input_Sources fromSource)
        {
            if (onTransformChanged != null)
                onTransformChanged.Invoke(this, fromSource);
            if (onTransformChangedEvent != null)
                onTransformChangedEvent.Invoke(this, fromSource);
        }

        protected virtual void OnDeviceConnectedChanged(SteamVR_Action_Pose changedAction, SteamVR_Input_Sources changedSource, bool connected)
        {
            CheckDeviceIndex();

            if (onConnectedChanged != null)
                onConnectedChanged.Invoke(this, inputSource, connected);
            if (onConnectedChangedEvent != null)
                onConnectedChangedEvent.Invoke(this, inputSource, connected);
        }

        protected virtual void OnTrackingChanged(SteamVR_Action_Pose changedAction, SteamVR_Input_Sources changedSource, ETrackingResult trackingChanged)
        {
            if (onTrackingChanged != null)
                onTrackingChanged.Invoke(this, inputSource, trackingChanged);
            if (onTrackingChangedEvent != null)
                onTrackingChangedEvent.Invoke(this, inputSource, trackingChanged);
        }

        protected virtual void CheckDeviceIndex()
        {
            if (poseAction[inputSource].active && poseAction[inputSource].deviceIsConnected)
            {
                int currentDeviceIndex = (int)poseAction[inputSource].trackedDeviceIndex;

```

```
                if (deviceIndex != currentDeviceIndex)
                {
                    deviceIndex = currentDeviceIndex;

                    if (broadcastDeviceChanges)
                    {
                        this.gameObject.BroadcastMessage("SetInputSource", inputSource, SendMessageOptions.DontRequireReceiver);
                        this.gameObject.BroadcastMessage("SetDeviceIndex", deviceIndex, SendMessageOptions.DontRequireReceiver);
                    }

                    if (onDeviceIndexChanged != null)
                        onDeviceIndexChanged.Invoke(this, inputSource, deviceIndex);
                    if (onDeviceIndexChangedEvent != null)
                        onDeviceIndexChangedEvent.Invoke(this, inputSource, deviceIndex);
                }
            }
        }

        /// <summary>
        /// Returns the device index for the device bound to the pose.
        /// </summary>
        public int GetDeviceIndex()
        {
            if (deviceIndex == -1)
                CheckDeviceIndex();

            return deviceIndex;
        }

        /// <summary>Returns the current velocity of the pose (as of the last update)</summary>
        public Vector3 GetVelocity()
        {
            return poseAction[inputSource].velocity;
        }

        /// <summary>Returns the current angular velocity of the pose (as of the last update)</summary>
        public Vector3 GetAngularVelocity()
        {
            return poseAction[inputSource].angularVelocity;
        }

        /// <summary>Returns the velocities of the pose at the time specified. Can predict in the future or return past values.</summary>
        public bool GetVelocitiesAtTimeOffset(float secondsFromNow, out Vector3 velocity, out Vector3 angularVelocity)
```

```
        {
            return poseAction[inputSource].GetVelocitiesAtTimeOffset(secondsFromNow, out velocity, out angularVelocity);
        }

        /// <summary>Uses previously recorded values to find the peak speed of the pose and returns the corresponding velocity and angular velocity</summary>
        public void GetEstimatedPeakVelocities(out Vector3 velocity, out Vector3 angularVelocity)
        {
            int top = historyBuffer.GetTopVelocity(10, 1);

            historyBuffer.GetAverageVelocities(out velocity, out angularVelocity, 2, top);
        }

        protected int lastFrameUpdated;
        protected void UpdateHistoryBuffer()
        {
            int currentFrame = Time.frameCount;
            if (lastFrameUpdated != currentFrame)
            {
                historyBuffer.Update(poseAction[inputSource].localPosition, poseAction[inputSource].localRotation, poseAction[inputSource].velocity, poseAction[inputSource].angularVelocity);
                lastFrameUpdated = currentFrame;
            }
        }

        /// <summary>
        /// Gets the localized name of the device that the action corresponds to.
        /// </summary>
        /// <param name="localizedParts">
        /// <list type="bullet">
        /// <item><description>VRInputString_Hand - Which hand the origin is in. E.g. "Left Hand"</description></item>
        /// <item><description>VRInputString_ControllerType - What kind of controller the user has in that hand.E.g. "Vive Controller"</description></item>
        /// <item><description>VRInputString_InputSource - What part of that controller is the origin. E.g. "Trackpad"</description></item>
        /// <item><description>VRInputString_All - All of the above. E.g. "Left Hand Vive Controller Trackpad"</description></item>
        /// </list>
        /// </param>
        public string GetLocalizedName(params EVRInputStringBits[] localizedParts)
        {
            if (poseAction != null)
```

```
                return poseAction.GetLocalizedOriginPart(inputSource,
                    localizedParts);
            return null;
        }

        public delegate void ActiveChangeHandler(SteamVR_Behaviour_Pose
            fromAction, SteamVR_Input_Sources fromSource, bool active);
        public delegate void ChangeHandler(SteamVR_Behaviour_Pose
            fromAction, SteamVR_Input_Sources fromSource);
        public delegate void UpdateHandler(SteamVR_Behaviour_Pose
            fromAction, SteamVR_Input_Sources fromSource);
        public delegate void TrackingChangeHandler(SteamVR_Behaviour_Pose
             fromAction, SteamVR_Input_Sources fromSource,
            ETrackingResult trackingState);
        public delegate void ValidPoseChangeHandler(
            SteamVR_Behaviour_Pose fromAction, SteamVR_Input_Sources
            fromSource, bool validPose);
        public delegate void DeviceConnectedChangeHandler(
            SteamVR_Behaviour_Pose fromAction, SteamVR_Input_Sources
            fromSource, bool deviceConnected);
        public delegate void DeviceIndexChangedHandler(
            SteamVR_Behaviour_Pose fromAction, SteamVR_Input_Sources
            fromSource, int newDeviceIndex);
    }
}
```

9.5.12. SteamVR_ Fade.cs

```
//======= Copyright (c) Valve Corporation, All rights reserved.
    ===============
//
// Purpose: CameraFade script adapted to work with SteamVR.
//
// Usage: Add to your top level SteamVR_Camera (the one with
    ApplyDistoration
//      checked) and drag a reference to this component into
    SteamVR_Camera
//      RenderComponents list.  Then call the static helper function
//      SteamVR_Fade.Start with the desired color and duration.
//      Use a duration of zero to set the start color.
//
// Example: Fade down from black over one second.
//      SteamVR_Fade.Start(Color.black, 0);
//      SteamVR_Fade.Start(Color.clear, 1);
//
// Note:  This component is provided to fade out a single camera layer's
//      scene view.  If instead you want to fade the entire view, use:
//      SteamVR_Fade.View(Color.black, 1);
//      (Does not affect the game view, however.)
//
//
    ============================================================================
```

```

using UnityEngine;
using Valve.VR;

namespace Valve.VR
{
    public class SteamVR_Fade : MonoBehaviour
    {
        private Color currentColor = new Color(0, 0, 0, 0); // default starting color: black and fully transparent
        private Color targetColor = new Color(0, 0, 0, 0);  // default target color: black and fully transparent
        private Color deltaColor = new Color(0, 0, 0, 0);   // the delta-color is basically the "speed / second" at which the current color should change
        private bool fadeOverlay = false;

        static public void Start(Color newColor, float duration, bool fadeOverlay = false)
        {
            SteamVR_Events.Fade.Send(newColor, duration, fadeOverlay);
        }

        static public void View(Color newColor, float duration)
        {
            var compositor = OpenVR.Compositor;
            if (compositor != null)
                compositor.FadeToColor(duration, newColor.r, newColor.g, newColor.b, newColor.a, false);
        }

#if TEST_FADE_VIEW
  void Update()
  {
    if (Input.GetKeyDown(KeyCode.Space))
    {
      SteamVR_Fade.View(Color.black, 0);
      SteamVR_Fade.View(Color.clear, 1);
    }
  }
#endif

        public void OnStartFade(Color newColor, float duration, bool fadeOverlay)
        {
            if (duration > 0.0f)
            {
                targetColor = newColor;
                deltaColor = (targetColor - currentColor) / duration;
            }
            else
            {
                currentColor = newColor;
```

```
            }
        }

        static Material fadeMaterial = null;
        static int fadeMaterialColorID = -1;

        void OnEnable()
        {
            if (fadeMaterial == null)
            {
                fadeMaterial = new Material(Shader.Find("Custom/SteamVR_Fade"));
                fadeMaterialColorID = Shader.PropertyToID("fadeColor");
            }

            SteamVR_Events.Fade.Listen(OnStartFade);
            SteamVR_Events.FadeReady.Send();
        }

        void OnDisable()
        {
            SteamVR_Events.Fade.Remove(OnStartFade);
        }

        void OnPostRender()
        {
            if (currentColor != targetColor)
            {
                // if the difference between the current alpha and the desired alpha is smaller than delta-alpha * deltaTime, then we're pretty much done fading:
                if (Mathf.Abs(currentColor.a - targetColor.a) < Mathf.Abs(deltaColor.a) * Time.deltaTime)
                {
                    currentColor = targetColor;
                    deltaColor = new Color(0, 0, 0, 0);
                }
                else
                {
                    currentColor += deltaColor * Time.deltaTime;
                }

                if (fadeOverlay)
                {
                    var overlay = SteamVR_Overlay.instance;
                    if (overlay != null)
                    {
                        overlay.alpha = 1.0f - currentColor.a;
                    }
                }
            }

            if (currentColor.a > 0 && fadeMaterial)
            {
```

```
                    fadeMaterial.SetColor(fadeMaterialColorID, currentColor);
                    fadeMaterial.SetPass(0);
                    GL.Begin(GL.QUADS);

                    GL.Vertex3(-1, -1, 0);
                    GL.Vertex3(1, -1, 0);
                    GL.Vertex3(1, 1, 0);
                    GL.Vertex3(-1, 1, 0);
                    GL.End();
                }
            }
        }
}
```

9.5.13. SteamVR_ Overlay.cs

```
//======= Copyright (c) Valve Corporation, All rights reserved.
    ===============
//
// Purpose: Displays 2d content on a large virtual screen.
//
//
    ============================================================================


using UnityEngine;
using System.Collections;
using Valve.VR;

namespace Valve.VR
{
    public class SteamVR_Overlay : MonoBehaviour
    {
        public Texture texture;
        public bool curved = true;
        public bool antialias = true;
        public bool highquality = true;

        [Tooltip("Size of overlay view.")]
        public float scale = 3.0f;

        [Tooltip("Distance from surface.")]
        public float distance = 1.25f;

        [Tooltip("Opacity"), Range(0.0f, 1.0f)]
        public float alpha = 1.0f;

        public Vector4 uvOffset = new Vector4(0, 0, 1, 1);
        public Vector2 mouseScale = new Vector2(1, 1);
        public Vector2 curvedRange = new Vector2(1, 2);

```

```
        public VROverlayInputMethod inputMethod = VROverlayInputMethod.
            None;

        static public SteamVR_Overlay instance { get; private set; }

        static public string key { get { return "unity:" + Application.
            companyName + "." + Application.productName; } }

        private ulong handle = OpenVR.k_ulOverlayHandleInvalid;

        void OnEnable()
        {
            var overlay = OpenVR.Overlay;
            if (overlay != null)
            {
                var error = overlay.CreateOverlay(key, gameObject.name,
                    ref handle);
                if (error != EVROverlayError.None)
                {
                    Debug.Log("<b>[SteamVR]</b> " + overlay.
                        GetOverlayErrorNameFromEnum(error));
                    enabled = false;
                    return;
                }
            }

            SteamVR_Overlay.instance = this;
        }

        void OnDisable()
        {
            if (handle != OpenVR.k_ulOverlayHandleInvalid)
            {
                var overlay = OpenVR.Overlay;
                if (overlay != null)
                {
                    overlay.DestroyOverlay(handle);
                }

                handle = OpenVR.k_ulOverlayHandleInvalid;
            }

            SteamVR_Overlay.instance = null;
        }

        public void UpdateOverlay()
        {
            var overlay = OpenVR.Overlay;
            if (overlay == null)
                return;

            if (texture != null)
            {
                var error = overlay.ShowOverlay(handle);
```

```
            if (error == EVROverlayError.InvalidHandle || error == EVROverlayError.UnknownOverlay)
            {
                if (overlay.FindOverlay(key, ref handle) != EVROverlayError.None)
                    return;
            }

            var tex = new Texture_t();
            tex.handle = texture.GetNativeTexturePtr();
            tex.eType = SteamVR.instance.textureType;
            tex.eColorSpace = EColorSpace.Auto;
            overlay.SetOverlayTexture(handle, ref tex);

            overlay.SetOverlayAlpha(handle, alpha);
            overlay.SetOverlayWidthInMeters(handle, scale);
            overlay.SetOverlayAutoCurveDistanceRangeInMeters(handle, curvedRange.x, curvedRange.y);

            var textureBounds = new VRTextureBounds_t();
            textureBounds.uMin = (0 + uvOffset.x) * uvOffset.z;
            textureBounds.vMin = (1 + uvOffset.y) * uvOffset.w;
            textureBounds.uMax = (1 + uvOffset.x) * uvOffset.z;
            textureBounds.vMax = (0 + uvOffset.y) * uvOffset.w;
            overlay.SetOverlayTextureBounds(handle, ref textureBounds);

            var vecMouseScale = new HmdVector2_t();
            vecMouseScale.v0 = mouseScale.x;
            vecMouseScale.v1 = mouseScale.y;
            overlay.SetOverlayMouseScale(handle, ref vecMouseScale);

            var vrcam = SteamVR_Render.Top();
            if (vrcam != null && vrcam.origin != null)
            {
                var offset = new SteamVR_Utils.RigidTransform(vrcam.origin, transform);
                offset.pos.x /= vrcam.origin.localScale.x;
                offset.pos.y /= vrcam.origin.localScale.y;
                offset.pos.z /= vrcam.origin.localScale.z;

                offset.pos.z += distance;

                var t = offset.ToHmdMatrix34();
                overlay.SetOverlayTransformAbsolute(handle, SteamVR.settings.trackingSpace, ref t);
            }

            overlay.SetOverlayInputMethod(handle, inputMethod);

            if (curved || antialias)
                highquality = true;

            if (highquality)
```

```
                {
                    overlay.SetHighQualityOverlay(handle);
                    overlay.SetOverlayFlag(handle, VROverlayFlags.Curved,
                         curved);
                    overlay.SetOverlayFlag(handle, VROverlayFlags.RGSS4X,
                         antialias);
                }
                else if (overlay.GetHighQualityOverlay() == handle)
                {
                    overlay.SetHighQualityOverlay(OpenVR.
                        k_ulOverlayHandleInvalid);
                }
            }
            else
            {
                overlay.HideOverlay(handle);
            }
        }

        public bool PollNextEvent(ref VREvent_t pEvent)
        {
            var overlay = OpenVR.Overlay;
            if (overlay == null)
                return false;

            var size = (uint)System.Runtime.InteropServices.Marshal.
                SizeOf(typeof(Valve.VR.VREvent_t));
            return overlay.PollNextOverlayEvent(handle, ref pEvent, size)
                ;
        }

        public struct IntersectionResults
        {
            public Vector3 point;
            public Vector3 normal;
            public Vector2 UVs;
            public float distance;
        }

        public bool ComputeIntersection(Vector3 source, Vector3 direction
            , ref IntersectionResults results)
        {
            var overlay = OpenVR.Overlay;
            if (overlay == null)
                return false;

            var input = new VROverlayIntersectionParams_t();
            input.eOrigin = SteamVR.settings.trackingSpace;
            input.vSource.v0 = source.x;
            input.vSource.v1 = source.y;
            input.vSource.v2 = -source.z;
            input.vDirection.v0 = direction.x;
            input.vDirection.v1 = direction.y;
            input.vDirection.v2 = -direction.z;
```

```

            var output = new VROverlayIntersectionResults_t();
            if (!overlay.ComputeOverlayIntersection(handle, ref input, ref output))
                return false;

            results.point = new Vector3(output.vPoint.v0, output.vPoint.v1, -output.vPoint.v2);
            results.normal = new Vector3(output.vNormal.v0, output.vNormal.v1, -output.vNormal.v2);
            results.UVs = new Vector2(output.vUVs.v0, output.vUVs.v1);
            results.distance = output.fDistance;
            return true;
        }
    }
}
```

9.5.14. SteamVR_ Render.cs

```
//======= Copyright (c) Valve Corporation, All rights reserved. ===============
//
// Purpose: Handles rendering of all SteamVR_Cameras
//
//=============================================================================

using UnityEngine;
using System.Collections;
using Valve.VR;


namespace Valve.VR
{
    public class SteamVR_Render : MonoBehaviour
    {
        public SteamVR_ExternalCamera externalCamera;
        public string externalCameraConfigPath = "externalcamera.cfg";

        public static EVREye eye { get; private set; }

        public static SteamVR_Render instance { get { return SteamVR_Behaviour.instance.steamvr_render; } }

        static private bool isQuitting;
        void OnApplicationQuit()
        {
            isQuitting = true;
            SteamVR.SafeDispose();
        }

```

```
        static public void Add(SteamVR_Camera vrcam)
        {
            if (!isQuitting)
                instance.AddInternal(vrcam);
        }

        static public void Remove(SteamVR_Camera vrcam)
        {
            if (!isQuitting && instance != null)
                instance.RemoveInternal(vrcam);
        }

        static public SteamVR_Camera Top()
        {
            if (!isQuitting)
                return instance.TopInternal();

            return null;
        }

        private SteamVR_Camera[] cameras = new SteamVR_Camera[0];

        void AddInternal(SteamVR_Camera vrcam)
        {
            var camera = vrcam.GetComponent<Camera>();
            var length = cameras.Length;
            var sorted = new SteamVR_Camera[length + 1];
            int insert = 0;
            for (int i = 0; i < length; i++)
            {
                var c = cameras[i].GetComponent<Camera>();
                if (i == insert && c.depth > camera.depth)
                    sorted[insert++] = vrcam;

                sorted[insert++] = cameras[i];
            }
            if (insert == length)
                sorted[insert] = vrcam;

            cameras = sorted;
        }

        void RemoveInternal(SteamVR_Camera vrcam)
        {
            var length = cameras.Length;
            int count = 0;
            for (int i = 0; i < length; i++)
            {
                var c = cameras[i];
                if (c == vrcam)
                    ++count;
            }
            if (count == 0)
                return;
```

```
            var sorted = new SteamVR_Camera[length - count];
            int insert = 0;
            for (int i = 0; i < length; i++)
            {
                var c = cameras[i];
                if (c != vrcam)
                    sorted[insert++] = c;
            }

            cameras = sorted;
        }

        SteamVR_Camera TopInternal()
        {
            if (cameras.Length > 0)
                return cameras[cameras.Length - 1];

            return null;
        }

        public TrackedDevicePose_t[] poses = new TrackedDevicePose_t[
            OpenVR.k_unMaxTrackedDeviceCount];
        public TrackedDevicePose_t[] gamePoses = new TrackedDevicePose_t
            [0];

        static private bool _pauseRendering;
        static public bool pauseRendering
        {
            get { return _pauseRendering; }
            set
            {
                _pauseRendering = value;

                var compositor = OpenVR.Compositor;
                if (compositor != null)
                    compositor.SuspendRendering(value);
            }
        }

        private WaitForEndOfFrame waitForEndOfFrame = new
            WaitForEndOfFrame();

        private IEnumerator RenderLoop()
        {
            while (Application.isPlaying)
            {
                yield return waitForEndOfFrame;

                if (pauseRendering)
                    continue;

                var compositor = OpenVR.Compositor;
                if (compositor != null)
```

```
                {
                    if (!compositor.CanRenderScene())
                        continue;

                    compositor.SetTrackingSpace(SteamVR.settings.trackingSpace);
                }

                var overlay = SteamVR_Overlay.instance;
                if (overlay != null)
                    overlay.UpdateOverlay();

                RenderExternalCamera();
            }
        }

        void RenderExternalCamera()
        {
            if (externalCamera == null)
                return;

            if (!externalCamera.gameObject.activeInHierarchy)
                return;

            var frameSkip = (int)Mathf.Max(externalCamera.config.frameSkip, 0.0f);
            if (Time.frameCount % (frameSkip + 1) != 0)
                return;

            // Keep external camera relative to the most relevant vr camera.
            externalCamera.AttachToCamera(TopInternal());

            externalCamera.RenderNear();
            externalCamera.RenderFar();
        }

        float sceneResolutionScale = 1.0f, timeScale = 1.0f;

        private void OnInputFocus(bool hasFocus)
        {
            if (SteamVR.active == false)
                return;

            if (hasFocus)
            {
                if (SteamVR.settings.pauseGameWhenDashboardVisible)
                {
                    Time.timeScale = timeScale;
                }

                SteamVR_Camera.sceneResolutionScale = sceneResolutionScale;
            }
```

```
            else
            {
                if (SteamVR.settings.pauseGameWhenDashboardVisible)
                {
                    timeScale = Time.timeScale;
                    Time.timeScale = 0.0f;
                }

                sceneResolutionScale = SteamVR_Camera.sceneResolutionScale;
                SteamVR_Camera.sceneResolutionScale = 0.5f;
            }
        }

        private string GetScreenshotFilename(uint screenshotHandle, EVRScreenshotPropertyFilenames screenshotPropertyFilename)
        {
            var error = EVRScreenshotError.None;
            var capacity = OpenVR.Screenshots.GetScreenshotPropertyFilename(screenshotHandle, screenshotPropertyFilename, null, 0, ref error);
            if (error != EVRScreenshotError.None && error != EVRScreenshotError.BufferTooSmall)
                return null;
            if (capacity > 1)
            {
                var result = new System.Text.StringBuilder((int)capacity);
                OpenVR.Screenshots.GetScreenshotPropertyFilename(screenshotHandle, screenshotPropertyFilename, result, capacity, ref error);
                if (error != EVRScreenshotError.None)
                    return null;
                return result.ToString();
            }
            return null;
        }

        private void OnRequestScreenshot(VREvent_t vrEvent)
        {
            var screenshotHandle = vrEvent.data.screenshot.handle;
            var screenshotType = (EVRScreenshotType)vrEvent.data.screenshot.type;

            if (screenshotType == EVRScreenshotType.StereoPanorama)
            {
                string previewFilename = GetScreenshotFilename(screenshotHandle, EVRScreenshotPropertyFilenames.Preview);
                string VRFilename = GetScreenshotFilename(screenshotHandle, EVRScreenshotPropertyFilenames.VR);

                if (previewFilename == null || VRFilename == null)
                    return;
```

```

                // Do the stereo panorama screenshot
                // Figure out where the view is
                GameObject screenshotPosition = new GameObject("screenshotPosition");
                screenshotPosition.transform.position = SteamVR_Render.Top().transform.position;
                screenshotPosition.transform.rotation = SteamVR_Render.Top().transform.rotation;
                screenshotPosition.transform.localScale = SteamVR_Render.Top().transform.lossyScale;
                SteamVR_Utils.TakeStereoScreenshot(screenshotHandle, screenshotPosition, 32, 0.064f, ref previewFilename, ref VRFilename);

                // and submit it
                OpenVR.Screenshots.SubmitScreenshot(screenshotHandle, screenshotType, previewFilename, VRFilename);
            }
        }

        private EVRScreenshotType[] screenshotTypes = new EVRScreenshotType[] { EVRScreenshotType.StereoPanorama };

        private void OnEnable()
        {
            StartCoroutine(RenderLoop());
            SteamVR_Events.InputFocus.Listen(OnInputFocus);
            SteamVR_Events.System(EVREventType.VREvent_RequestScreenshot).Listen(OnRequestScreenshot);

#if UNITY_2017_1_OR_NEWER
        Application.onBeforeRender += OnBeforeRender;
#else
            Camera.onPreCull += OnCameraPreCull;
#endif

            if (SteamVR.initializedState == SteamVR.InitializedStates.InitializeSuccess)
                OpenVR.Screenshots.HookScreenshot(screenshotTypes);
            else
                SteamVR_Events.Initialized.AddListener(OnSteamVRInitialized);
        }

        private void OnSteamVRInitialized(bool success)
        {
            if (success)
                OpenVR.Screenshots.HookScreenshot(screenshotTypes);
        }

        private void OnDisable()
        {
            StopAllCoroutines();
```

```
            SteamVR_Events.InputFocus.Remove(OnInputFocus);
            SteamVR_Events.System(EVREventType.VREvent_RequestScreenshot).Remove(OnRequestScreenshot);

#if UNITY_2017_1_OR_NEWER
        Application.onBeforeRender -= OnBeforeRender;
#else
            Camera.onPreCull -= OnCameraPreCull;
#endif

            if (SteamVR.initializedState != SteamVR.InitializedStates.InitializeSuccess)
                SteamVR_Events.Initialized.RemoveListener(OnSteamVRInitialized);
        }

        private void Awake()
        {
            if (externalCamera == null && System.IO.File.Exists(externalCameraConfigPath))
            {
                var prefab = Resources.Load<GameObject>("SteamVR_ExternalCamera");
                var instance = Instantiate(prefab);
                instance.gameObject.name = "External Camera";

                externalCamera = instance.transform.GetChild(0).GetComponent<SteamVR_ExternalCamera>();
                externalCamera.configPath = externalCameraConfigPath;
                externalCamera.ReadConfig();
            }
        }

        public void UpdatePoses()
        {
            var compositor = OpenVR.Compositor;
            if (compositor != null)
            {
                compositor.GetLastPoses(poses, gamePoses);
                SteamVR_Events.NewPoses.Send(poses);
                SteamVR_Events.NewPosesApplied.Send();
            }
        }

#if UNITY_2017_1_OR_NEWER
      void OnBeforeRender()
        {
            if (SteamVR.active == false)
                return;

            if (SteamVR.settings.IsPoseUpdateMode(SteamVR_UpdateModes.OnPreCull))
            {
                UpdatePoses();
```

```
            }
        }
#else
        void OnCameraPreCull(Camera cam)
        {
            if (SteamVR.active == false)
                return;

#if UNITY_2017_1_OR_NEWER
    if (cam.cameraType != CameraType.VR)
      return;
#else
            //custom code
            if (!cam.stereoEnabled) //if not main camera (stereoEnabled
                isn't perfect, but it is the fast/easiest way to check
                this in Unity 5.4)
            {
                return;
            }
#endif
            // Only update poses on the first camera per frame.
            if (Time.frameCount != lastFrameCount)
            {
                lastFrameCount = Time.frameCount;

                if (SteamVR.settings.IsPoseUpdateMode(SteamVR_UpdateModes
                    .OnPreCull))
                {
                    UpdatePoses();
                }
            }
        }
        static int lastFrameCount = -1;
#endif

        void Update()
        {
            if (SteamVR.active == false)
                return;

            UpdatePoses();

            // Dispatch any OpenVR events.
            var system = OpenVR.System;
            if (system != null)
            {
                var vrEvent = new VREvent_t();
                var size = (uint)System.Runtime.InteropServices.Marshal.
                    SizeOf(typeof(VREvent_t));
                for (int i = 0; i < 64; i++)
                {
                    if (!system.PollNextEvent(ref vrEvent, size))
                        break;

```

```
                    switch ((EVREventType)vrEvent.eventType)
                    {
                        case EVREventType.VREvent_InputFocusCaptured: // another app has taken focus (likely dashboard)
                            if (vrEvent.data.process.oldPid == 0)
                            {
                                SteamVR_Events.InputFocus.Send(false);
                            }
                            break;
                        case EVREventType.VREvent_InputFocusReleased: // that app has released input focus
                            if (vrEvent.data.process.pid == 0)
                            {
                                SteamVR_Events.InputFocus.Send(true);
                            }
                            break;
                        case EVREventType.VREvent_ShowRenderModels:
                            SteamVR_Events.HideRenderModels.Send(false);
                            break;
                        case EVREventType.VREvent_HideRenderModels:
                            SteamVR_Events.HideRenderModels.Send(true);
                            break;
                        default:
                            SteamVR_Events.System((EVREventType)vrEvent.eventType).Send(vrEvent);
                            break;
                    }
                }
            }

            // Ensure various settings to minimize latency.
            Application.targetFrameRate = -1;
            Application.runInBackground = true; // don't require companion window focus
            QualitySettings.maxQueuedFrames = -1;
            QualitySettings.vSyncCount = 0; // this applies to the companion window

            if (SteamVR.settings.lockPhysicsUpdateRateToRenderFrequency && Time.timeScale > 0.0f)
            {
                var vr = SteamVR.instance;
                if (vr != null)
                {
                    var timing = new Compositor_FrameTiming();
                    timing.m_nSize = (uint)System.Runtime.InteropServices.Marshal.SizeOf(typeof(Compositor_FrameTiming));
                    vr.compositor.GetFrameTiming(ref timing, 0);

                    Time.fixedDeltaTime = Time.timeScale / vr.hmd_DisplayFrequency;
                }
            }
```

```
        }
    }
}
```

9.5.15. Teleport.cs

```
//======= Copyright (c) Valve Corporation, All rights reserved.
    ===============
//
// Purpose: Handles all the teleport logic
//
//
    =============================================================================

using UnityEngine;
using UnityEngine.Events;
using System.Collections;

namespace Valve.VR.InteractionSystem
{
  //
      -------------------------------------------------------------------------
  public class Teleport : MonoBehaviour
    {
        public SteamVR_Action_Boolean teleportAction = SteamVR_Input.
            GetAction<SteamVR_Action_Boolean>("Teleport");

        public LayerMask traceLayerMask;
    public LayerMask floorFixupTraceLayerMask;
    public float floorFixupMaximumTraceDistance = 1.0f;
    public Material areaVisibleMaterial;
    public Material areaLockedMaterial;
    public Material areaHighlightedMaterial;
    public Material pointVisibleMaterial;
    public Material pointLockedMaterial;
    public Material pointHighlightedMaterial;
    public Transform destinationReticleTransform;
    public Transform invalidReticleTransform;
    public GameObject playAreaPreviewCorner;
    public GameObject playAreaPreviewSide;
    public Color pointerValidColor;
    public Color pointerInvalidColor;
    public Color pointerLockedColor;
    public bool showPlayAreaMarker = true;

    public float teleportFadeTime = 0.1f;
    public float meshFadeTime = 0.2f;

    public float arcDistance = 10.0f;

```

```
    [Header( "Effects" )]
    public Transform onActivateObjectTransform;
    public Transform onDeactivateObjectTransform;
    public float activateObjectTime = 1.0f;
    public float deactivateObjectTime = 1.0f;

    [Header( "Audio Sources" )]
    public AudioSource pointerAudioSource;
    public AudioSource loopingAudioSource;
    public AudioSource headAudioSource;
    public AudioSource reticleAudioSource;

    [Header( "Sounds" )]
    public AudioClip teleportSound;
    public AudioClip pointerStartSound;
    public AudioClip pointerLoopSound;
    public AudioClip pointerStopSound;
    public AudioClip goodHighlightSound;
    public AudioClip badHighlightSound;

    [Header( "Debug" )]
    public bool debugFloor = false;
    public bool showOffsetReticle = false;
    public Transform offsetReticleTransform;
    public MeshRenderer floorDebugSphere;
    public LineRenderer floorDebugLine;

    private LineRenderer pointerLineRenderer;
    private GameObject teleportPointerObject;
    private Transform pointerStartTransform;
    private Hand pointerHand = null;
    private Player player = null;
    private TeleportArc teleportArc = null;

    private bool visible = false;

    private TeleportMarkerBase[] teleportMarkers;
    private TeleportMarkerBase pointedAtTeleportMarker;
    private TeleportMarkerBase teleportingToMarker;
    private Vector3 pointedAtPosition;
    private Vector3 prevPointedAtPosition;
    private bool teleporting = false;
    private float currentFadeTime = 0.0f;

    private float meshAlphaPercent = 1.0f;
    private float pointerShowStartTime = 0.0f;
    private float pointerHideStartTime = 0.0f;
    private bool meshFading = false;
    private float fullTintAlpha;

    private float invalidReticleMinScale = 0.2f;
    private float invalidReticleMaxScale = 1.0f;
    private float invalidReticleMinScaleDistance = 0.4f;
    private float invalidReticleMaxScaleDistance = 2.0f;
```

```
    private Vector3 invalidReticleScale = Vector3.one;
    private Quaternion invalidReticleTargetRotation = Quaternion.identity;

    private Transform playAreaPreviewTransform;
    private Transform[] playAreaPreviewCorners;
    private Transform[] playAreaPreviewSides;

    private float loopingAudioMaxVolume = 0.0f;

    private Coroutine hintCoroutine = null;

    private bool originalHoverLockState = false;
    private Interactable originalHoveringInteractable = null;
    private AllowTeleportWhileAttachedToHand allowTeleportWhileAttached = null;

    private Vector3 startingFeetOffset = Vector3.zero;
    private bool movedFeetFarEnough = false;

    SteamVR_Events.Action chaperoneInfoInitializedAction;

    // Events

    public static SteamVR_Events.Event< float > ChangeScene = new SteamVR_Events.Event< float >();
    public static SteamVR_Events.Action< float > ChangeSceneAction( UnityAction< float > action ) { return new SteamVR_Events.Action< float >( ChangeScene, action ); }

    public static SteamVR_Events.Event< TeleportMarkerBase > Player = new SteamVR_Events.Event< TeleportMarkerBase >();
    public static SteamVR_Events.Action< TeleportMarkerBase > PlayerAction( UnityAction< TeleportMarkerBase > action ) { return new SteamVR_Events.Action< TeleportMarkerBase >( Player, action ); }

    public static SteamVR_Events.Event< TeleportMarkerBase > PlayerPre = new SteamVR_Events.Event< TeleportMarkerBase >();
    public static SteamVR_Events.Action< TeleportMarkerBase > PlayerPreAction( UnityAction< TeleportMarkerBase > action ) { return new SteamVR_Events.Action< TeleportMarkerBase >( PlayerPre, action ); }

    //-------------------------------------------------
    private static Teleport _instance;
    public static Teleport instance
    {
      get
      {
        if ( _instance == null )
        {
          _instance = GameObject.FindObjectOfType<Teleport>();
        }
```

```

      return _instance;
    }
  }


  //-------------------------------------------------
  void Awake()
      {
          _instance = this;

    chaperoneInfoInitializedAction = ChaperoneInfo.InitializedAction(
        OnChaperoneInfoInitialized );

    pointerLineRenderer = GetComponentInChildren<LineRenderer>();
    teleportPointerObject = pointerLineRenderer.gameObject;

    int tintColorID = Shader.PropertyToID( "_TintColor" );
    fullTintAlpha = pointVisibleMaterial.GetColor( tintColorID ).a;

    teleportArc = GetComponent<TeleportArc>();
    teleportArc.traceLayerMask = traceLayerMask;

    loopingAudioMaxVolume = loopingAudioSource.volume;

    playAreaPreviewCorner.SetActive( false );
    playAreaPreviewSide.SetActive( false );

    float invalidReticleStartingScale = invalidReticleTransform.
        localScale.x;
    invalidReticleMinScale *= invalidReticleStartingScale;
    invalidReticleMaxScale *= invalidReticleStartingScale;
  }


  //-------------------------------------------------
  void Start()
      {
          teleportMarkers = GameObject.FindObjectsOfType<
              TeleportMarkerBase>();

    HidePointer();

    player = InteractionSystem.Player.instance;

    if ( player == null )
    {
      Debug.LogError("<b>[SteamVR Interaction]</b> Teleport: No Player
          instance found in map.");
      Destroy( this.gameObject );
      return;
    }

    CheckForSpawnPoint();
```

```

		Invoke( "ShowTeleportHint", 5.0f );
	}


	//-------------------------------------------------
	void OnEnable()
	{
		chaperoneInfoInitializedAction.enabled = true;
		OnChaperoneInfoInitialized(); // In case it's already initialized
	}


	//-------------------------------------------------
	void OnDisable()
	{
		chaperoneInfoInitializedAction.enabled = false;
		HidePointer();
	}


	//-------------------------------------------------
	private void CheckForSpawnPoint()
	{
		foreach ( TeleportMarkerBase teleportMarker in teleportMarkers )
		{
			TeleportPoint teleportPoint = teleportMarker as TeleportPoint;
			if ( teleportPoint && teleportPoint.playerSpawnPoint )
			{
				teleportingToMarker = teleportMarker;
				TeleportPlayer();
				break;
			}
		}
	}


	//-------------------------------------------------
	public void HideTeleportPointer()
	{
		if ( pointerHand != null )
		{
			HidePointer();
		}
	}


	//-------------------------------------------------
	void Update()
	{
		Hand oldPointerHand = pointerHand;
		Hand newPointerHand = null;

		foreach ( Hand hand in player.hands )
```

```
        {
          if ( visible )
          {
            if ( WasTeleportButtonReleased( hand ) )
            {
              if ( pointerHand == hand ) //This is the pointer hand
              {
                TryTeleportPlayer();
              }
            }
          }

          if ( WasTeleportButtonPressed( hand ) )
          {
            newPointerHand = hand;
          }
        }

        //If something is attached to the hand that is preventing teleport
        if ( allowTeleportWhileAttached && !allowTeleportWhileAttached.teleportAllowed )
        {
          HidePointer();
        }
        else
        {
          if ( !visible && newPointerHand != null )
          {
            //Begin showing the pointer
            ShowPointer( newPointerHand, oldPointerHand );
          }
          else if ( visible )
          {
            if ( newPointerHand == null && !IsTeleportButtonDown( pointerHand ) )
            {
              //Hide the pointer
              HidePointer();
            }
            else if ( newPointerHand != null )
            {
              //Move the pointer to a new hand
              ShowPointer( newPointerHand, oldPointerHand );
            }
          }
        }

        if ( visible )
        {
          UpdatePointer();

          if ( meshFading )
          {
            UpdateTeleportColors();
```

```
        }

        if ( onActivateObjectTransform.gameObject.activeSelf && Time.time - pointerShowStartTime > activateObjectTime )
        {
            onActivateObjectTransform.gameObject.SetActive( false );
        }
    }
    else
    {
        if ( onDeactivateObjectTransform.gameObject.activeSelf && Time.time - pointerHideStartTime > deactivateObjectTime )
        {
            onDeactivateObjectTransform.gameObject.SetActive( false );
        }
    }
}


//-------------------------------------------------
private void UpdatePointer()
{
    Vector3 pointerStart = pointerStartTransform.position;
    Vector3 pointerEnd;
    Vector3 pointerDir = pointerStartTransform.forward;
    bool hitSomething = false;
    bool showPlayAreaPreview = false;
    Vector3 playerFeetOffset = player.trackingOriginTransform.position - player.feetPositionGuess;

    Vector3 arcVelocity = pointerDir * arcDistance;

    TeleportMarkerBase hitTeleportMarker = null;

    //Check pointer angle
    float dotUp = Vector3.Dot( pointerDir, Vector3.up );
    float dotForward = Vector3.Dot( pointerDir, player.hmdTransform.forward );
    bool pointerAtBadAngle = false;
    if ( ( dotForward > 0 && dotUp > 0.75f ) || ( dotForward < 0.0f && dotUp > 0.5f ) )
    {
        pointerAtBadAngle = true;
    }

    //Trace to see if the pointer hit anything
    RaycastHit hitInfo;
    teleportArc.SetArcData( pointerStart, arcVelocity, true, pointerAtBadAngle );
    if ( teleportArc.DrawArc( out hitInfo ) )
    {
        hitSomething = true;
        hitTeleportMarker = hitInfo.collider.GetComponentInParent<TeleportMarkerBase>();
```

```
      }

      if ( pointerAtBadAngle )
      {
        hitTeleportMarker = null;
      }

      HighlightSelected( hitTeleportMarker );

      if ( hitTeleportMarker != null ) //Hit a teleport marker
      {
        if ( hitTeleportMarker.locked )
        {
          teleportArc.SetColor( pointerLockedColor );
#if (UNITY_5_4)
          pointerLineRenderer.SetColors( pointerLockedColor,
              pointerLockedColor );
#else
          pointerLineRenderer.startColor = pointerLockedColor;
          pointerLineRenderer.endColor = pointerLockedColor;
#endif
          destinationReticleTransform.gameObject.SetActive( false );
        }
        else
        {
          teleportArc.SetColor( pointerValidColor );
#if (UNITY_5_4)
          pointerLineRenderer.SetColors( pointerValidColor,
              pointerValidColor );
#else
          pointerLineRenderer.startColor = pointerValidColor;
          pointerLineRenderer.endColor = pointerValidColor;
#endif
          destinationReticleTransform.gameObject.SetActive(
              hitTeleportMarker.showReticle );
        }

        offsetReticleTransform.gameObject.SetActive( true );

        invalidReticleTransform.gameObject.SetActive( false );

        pointedAtTeleportMarker = hitTeleportMarker;
        pointedAtPosition = hitInfo.point;

        if ( showPlayAreaMarker )
        {
          //Show the play area marker if this is a teleport area
          TeleportArea teleportArea = pointedAtTeleportMarker as
              TeleportArea;
          if ( teleportArea != null && !teleportArea.locked &&
              playAreaPreviewTransform != null )
          {
            Vector3 offsetToUse = playerFeetOffset;

```

```
            //Adjust the actual offset to prevent the play area marker from moving too much
            if ( !movedFeetFarEnough )
            {
              float distanceFromStartingOffset = Vector3.Distance( playerFeetOffset, startingFeetOffset );
              if ( distanceFromStartingOffset < 0.1f )
              {
                offsetToUse = startingFeetOffset;
              }
              else if ( distanceFromStartingOffset < 0.4f )
              {
                offsetToUse = Vector3.Lerp( startingFeetOffset, playerFeetOffset, ( distanceFromStartingOffset - 0.1f ) / 0.3f );
              }
              else
              {
                movedFeetFarEnough = true;
              }
            }

            playAreaPreviewTransform.position = pointedAtPosition + offsetToUse;

            showPlayAreaPreview = true;
          }
        }

        pointerEnd = hitInfo.point;
      }
      else //Hit neither
      {
        destinationReticleTransform.gameObject.SetActive( false );
        offsetReticleTransform.gameObject.SetActive( false );

        teleportArc.SetColor( pointerInvalidColor );
#if (UNITY_5_4)
        pointerLineRenderer.SetColors( pointerInvalidColor, pointerInvalidColor );
#else
        pointerLineRenderer.startColor = pointerInvalidColor;
        pointerLineRenderer.endColor = pointerInvalidColor;
#endif
        invalidReticleTransform.gameObject.SetActive( !pointerAtBadAngle );

        //Orient the invalid reticle to the normal of the trace hit point
        Vector3 normalToUse = hitInfo.normal;
        float angle = Vector3.Angle( hitInfo.normal, Vector3.up );
        if ( angle < 15.0f )
        {
          normalToUse = Vector3.up;
        }
```

```
        invalidReticleTargetRotation = Quaternion.FromToRotation( Vector3
            .up, normalToUse );
        invalidReticleTransform.rotation = Quaternion.Slerp(
            invalidReticleTransform.rotation,
            invalidReticleTargetRotation, 0.1f );

        //Scale the invalid reticle based on the distance from the player
        float distanceFromPlayer = Vector3.Distance( hitInfo.point,
            player.hmdTransform.position );
        float invalidReticleCurrentScale = Util.RemapNumberClamped(
            distanceFromPlayer, invalidReticleMinScaleDistance,
            invalidReticleMaxScaleDistance, invalidReticleMinScale,
            invalidReticleMaxScale );
        invalidReticleScale.x = invalidReticleCurrentScale;
        invalidReticleScale.y = invalidReticleCurrentScale;
        invalidReticleScale.z = invalidReticleCurrentScale;
        invalidReticleTransform.transform.localScale =
            invalidReticleScale;

        pointedAtTeleportMarker = null;

        if ( hitSomething )
        {
          pointerEnd = hitInfo.point;
        }
        else
        {
          pointerEnd = teleportArc.GetArcPositionAtTime( teleportArc.
              arcDuration );
        }

        //Debug floor
        if ( debugFloor )
        {
          floorDebugSphere.gameObject.SetActive( false );
          floorDebugLine.gameObject.SetActive( false );
        }
      }

      if ( playAreaPreviewTransform != null )
      {
        playAreaPreviewTransform.gameObject.SetActive(
            showPlayAreaPreview );
      }

      if ( !showOffsetReticle )
      {
        offsetReticleTransform.gameObject.SetActive( false );
      }

      destinationReticleTransform.position = pointedAtPosition;
      invalidReticleTransform.position = pointerEnd;
      onActivateObjectTransform.position = pointerEnd;
      onDeactivateObjectTransform.position = pointerEnd;
```

```
      offsetReticleTransform.position = pointerEnd - playerFeetOffset;

      reticleAudioSource.transform.position = pointedAtPosition;

      pointerLineRenderer.SetPosition( 0, pointerStart );
      pointerLineRenderer.SetPosition( 1, pointerEnd );
    }


    //-------------------------------------------------
    void FixedUpdate()
    {
      if ( !visible )
      {
        return;
      }

      if ( debugFloor )
      {
        //Debug floor
        TeleportArea teleportArea = pointedAtTeleportMarker as
            TeleportArea;
        if ( teleportArea != null )
        {
          if ( floorFixupMaximumTraceDistance > 0.0f )
          {
            floorDebugSphere.gameObject.SetActive( true );
            floorDebugLine.gameObject.SetActive( true );

            RaycastHit raycastHit;
            Vector3 traceDir = Vector3.down;
            traceDir.x = 0.01f;
            if ( Physics.Raycast( pointedAtPosition + 0.05f * traceDir,
                traceDir, out raycastHit, floorFixupMaximumTraceDistance,
                 floorFixupTraceLayerMask ) )
            {
              floorDebugSphere.transform.position = raycastHit.point;
              floorDebugSphere.material.color = Color.green;
#if (UNITY_5_4)
              floorDebugLine.SetColors( Color.green, Color.green );
#else
              floorDebugLine.startColor = Color.green;
              floorDebugLine.endColor = Color.green;
#endif
              floorDebugLine.SetPosition( 0, pointedAtPosition );
              floorDebugLine.SetPosition( 1, raycastHit.point );
            }
            else
            {
              Vector3 rayEnd = pointedAtPosition + ( traceDir *
                  floorFixupMaximumTraceDistance );
              floorDebugSphere.transform.position = rayEnd;
              floorDebugSphere.material.color = Color.red;
#if (UNITY_5_4)
```

```
                floorDebugLine.SetColors( Color.red, Color.red );
#else
                floorDebugLine.startColor = Color.red;
                floorDebugLine.endColor = Color.red;
#endif
                floorDebugLine.SetPosition( 0, pointedAtPosition );
                floorDebugLine.SetPosition( 1, rayEnd );
              }
            }
          }
        }
    }


    //-------------------------------------------------
    private void OnChaperoneInfoInitialized()
    {
      ChaperoneInfo chaperone = ChaperoneInfo.instance;

      if ( chaperone.initialized && chaperone.roomscale )
      {
        //Set up the render model for the play area bounds

        if ( playAreaPreviewTransform == null )
        {
          playAreaPreviewTransform = new GameObject( "
              PlayAreaPreviewTransform" ).transform;
          playAreaPreviewTransform.parent = transform;
          Util.ResetTransform( playAreaPreviewTransform );

          playAreaPreviewCorner.SetActive( true );
          playAreaPreviewCorners = new Transform[4];
          playAreaPreviewCorners[0] = playAreaPreviewCorner.transform;
          playAreaPreviewCorners[1] = Instantiate( playAreaPreviewCorners
              [0] );
          playAreaPreviewCorners[2] = Instantiate( playAreaPreviewCorners
              [0] );
          playAreaPreviewCorners[3] = Instantiate( playAreaPreviewCorners
              [0] );

          playAreaPreviewCorners[0].transform.parent =
              playAreaPreviewTransform;
          playAreaPreviewCorners[1].transform.parent =
              playAreaPreviewTransform;
          playAreaPreviewCorners[2].transform.parent =
              playAreaPreviewTransform;
          playAreaPreviewCorners[3].transform.parent =
              playAreaPreviewTransform;

          playAreaPreviewSide.SetActive( true );
          playAreaPreviewSides = new Transform[4];
          playAreaPreviewSides[0] = playAreaPreviewSide.transform;
          playAreaPreviewSides[1] = Instantiate( playAreaPreviewSides[0]
              );
```

```
            playAreaPreviewSides[2] = Instantiate( playAreaPreviewSides[0] );
            playAreaPreviewSides[3] = Instantiate( playAreaPreviewSides[0] );

            playAreaPreviewSides[0].transform.parent = playAreaPreviewTransform;
            playAreaPreviewSides[1].transform.parent = playAreaPreviewTransform;
            playAreaPreviewSides[2].transform.parent = playAreaPreviewTransform;
            playAreaPreviewSides[3].transform.parent = playAreaPreviewTransform;
        }

        float x = chaperone.playAreaSizeX;
        float z = chaperone.playAreaSizeZ;

        playAreaPreviewSides[0].localPosition = new Vector3( 0.0f, 0.0f, 0.5f * z - 0.25f );
        playAreaPreviewSides[1].localPosition = new Vector3( 0.0f, 0.0f, -0.5f * z + 0.25f );
        playAreaPreviewSides[2].localPosition = new Vector3( 0.5f * x - 0.25f, 0.0f, 0.0f );
        playAreaPreviewSides[3].localPosition = new Vector3( -0.5f * x + 0.25f, 0.0f, 0.0f );

        playAreaPreviewSides[0].localScale = new Vector3( x - 0.5f, 1.0f, 1.0f );
        playAreaPreviewSides[1].localScale = new Vector3( x - 0.5f, 1.0f, 1.0f );
        playAreaPreviewSides[2].localScale = new Vector3( z - 0.5f, 1.0f, 1.0f );
        playAreaPreviewSides[3].localScale = new Vector3( z - 0.5f, 1.0f, 1.0f );

        playAreaPreviewSides[0].localRotation = Quaternion.Euler( 0.0f, 0.0f, 0.0f );
        playAreaPreviewSides[1].localRotation = Quaternion.Euler( 0.0f, 180.0f, 0.0f );
        playAreaPreviewSides[2].localRotation = Quaternion.Euler( 0.0f, 90.0f, 0.0f );
        playAreaPreviewSides[3].localRotation = Quaternion.Euler( 0.0f, 270.0f, 0.0f );

        playAreaPreviewCorners[0].localPosition = new Vector3( 0.5f * x - 0.25f, 0.0f, 0.5f * z - 0.25f );
        playAreaPreviewCorners[1].localPosition = new Vector3( 0.5f * x - 0.25f, 0.0f, -0.5f * z + 0.25f );
        playAreaPreviewCorners[2].localPosition = new Vector3( -0.5f * x + 0.25f, 0.0f, -0.5f * z + 0.25f );
        playAreaPreviewCorners[3].localPosition = new Vector3( -0.5f * x + 0.25f, 0.0f, 0.5f * z - 0.25f );

```

```
            playAreaPreviewCorners[0].localRotation = Quaternion.Euler( 0.0f,
                 0.0f, 0.0f );
            playAreaPreviewCorners[1].localRotation = Quaternion.Euler( 0.0f,
                 90.0f, 0.0f );
            playAreaPreviewCorners[2].localRotation = Quaternion.Euler( 0.0f,
                 180.0f, 0.0f );
            playAreaPreviewCorners[3].localRotation = Quaternion.Euler( 0.0f,
                 270.0f, 0.0f );

            playAreaPreviewTransform.gameObject.SetActive( false );
        }
    }


    //-------------------------------------------------
    private void HidePointer()
    {
        if ( visible )
        {
            pointerHideStartTime = Time.time;
        }

        visible = false;
        if ( pointerHand )
        {
            if ( ShouldOverrideHoverLock() )
            {
                //Restore the original hovering interactable on the hand
                if ( originalHoverLockState == true )
                {
                    pointerHand.HoverLock( originalHoveringInteractable );
                }
                else
                {
                    pointerHand.HoverUnlock( null );
                }
            }

            //Stop looping sound
            loopingAudioSource.Stop();
            PlayAudioClip( pointerAudioSource, pointerStopSound );
        }
        teleportPointerObject.SetActive( false );

        teleportArc.Hide();

        foreach ( TeleportMarkerBase teleportMarker in teleportMarkers )
        {
            if ( teleportMarker != null && teleportMarker.markerActive &&
                 teleportMarker.gameObject != null )
            {
                teleportMarker.gameObject.SetActive( false );
            }
        }
```

```

		destinationReticleTransform.gameObject.SetActive( false );
		invalidReticleTransform.gameObject.SetActive( false );
		offsetReticleTransform.gameObject.SetActive( false );

		if ( playAreaPreviewTransform != null )
		{
			playAreaPreviewTransform.gameObject.SetActive( false );
		}

		if ( onActivateObjectTransform.gameObject.activeSelf )
		{
			onActivateObjectTransform.gameObject.SetActive( false );
		}
		onDeactivateObjectTransform.gameObject.SetActive( true );

		pointerHand = null;
	}


	//-------------------------------------------------
	private void ShowPointer( Hand newPointerHand, Hand oldPointerHand )
	{
		if ( !visible )
		{
			pointedAtTeleportMarker = null;
			pointerShowStartTime = Time.time;
			visible = true;
			meshFading = true;

			teleportPointerObject.SetActive( false );
			teleportArc.Show();

			foreach ( TeleportMarkerBase teleportMarker in teleportMarkers )
			{
				if ( teleportMarker.markerActive && teleportMarker.
					ShouldActivate( player.feetPositionGuess ) )
				{
					teleportMarker.gameObject.SetActive( true );
					teleportMarker.Highlight( false );
				}
			}

			startingFeetOffset = player.trackingOriginTransform.position -
				player.feetPositionGuess;
			movedFeetFarEnough = false;

			if ( onDeactivateObjectTransform.gameObject.activeSelf )
			{
				onDeactivateObjectTransform.gameObject.SetActive( false );
			}
			onActivateObjectTransform.gameObject.SetActive( true );

			loopingAudioSource.clip = pointerLoopSound;
```

```
			loopingAudioSource.loop = true;
			loopingAudioSource.Play();
			loopingAudioSource.volume = 0.0f;
		}


		if ( oldPointerHand )
		{
			if ( ShouldOverrideHoverLock() )
			{
				//Restore the original hovering interactable on the hand
				if ( originalHoverLockState == true )
				{
					oldPointerHand.HoverLock( originalHoveringInteractable );
				}
				else
				{
					oldPointerHand.HoverUnlock( null );
				}
			}
		}

		pointerHand = newPointerHand;

		if ( visible && oldPointerHand != pointerHand )
		{
			PlayAudioClip( pointerAudioSource, pointerStartSound );
		}

		if ( pointerHand )
		{
			pointerStartTransform = GetPointerStartTransform( pointerHand );

			if ( pointerHand.currentAttachedObject != null )
			{
				allowTeleportWhileAttached = pointerHand.currentAttachedObject.
					GetComponent<AllowTeleportWhileAttachedToHand>();
			}

			//Keep track of any existing hovering interactable on the hand
			originalHoverLockState = pointerHand.hoverLocked;
			originalHoveringInteractable = pointerHand.hoveringInteractable;

			if ( ShouldOverrideHoverLock() )
			{
				pointerHand.HoverLock( null );
			}

			pointerAudioSource.transform.SetParent( pointerStartTransform );
			pointerAudioSource.transform.localPosition = Vector3.zero;

			loopingAudioSource.transform.SetParent( pointerStartTransform );
			loopingAudioSource.transform.localPosition = Vector3.zero;
		}
```

```
	}


	//-------------------------------------------------
	private void UpdateTeleportColors()
	{
		float deltaTime = Time.time - pointerShowStartTime;
		if ( deltaTime > meshFadeTime )
		{
			meshAlphaPercent = 1.0f;
			meshFading = false;
		}
		else
		{
			meshAlphaPercent = Mathf.Lerp( 0.0f, 1.0f, deltaTime /
				meshFadeTime );
		}

		//Tint color for the teleport points
		foreach ( TeleportMarkerBase teleportMarker in teleportMarkers )
		{
			teleportMarker.SetAlpha( fullTintAlpha * meshAlphaPercent,
				meshAlphaPercent );
		}
	}


	//-------------------------------------------------
	private void PlayAudioClip( AudioSource source, AudioClip clip )
	{
		source.clip = clip;
		source.Play();
	}


	//-------------------------------------------------
	private void PlayPointerHaptic( bool validLocation )
	{
		if ( pointerHand != null )
		{
			if ( validLocation )
			{
				pointerHand.TriggerHapticPulse( 800 );
			}
			else
			{
				pointerHand.TriggerHapticPulse( 100 );
			}
		}
	}


	//-------------------------------------------------
	private void TryTeleportPlayer()
```

```
    {
      if ( visible && !teleporting )
      {
        if ( pointedAtTeleportMarker != null && pointedAtTeleportMarker.
            locked == false )
        {
          //Pointing at an unlocked teleport marker
          teleportingToMarker = pointedAtTeleportMarker;
          InitiateTeleportFade();

          CancelTeleportHint();
        }
      }
    }


    //-------------------------------------------------
    private void InitiateTeleportFade()
    {
      teleporting = true;

      currentFadeTime = teleportFadeTime;

      TeleportPoint teleportPoint = teleportingToMarker as TeleportPoint;
      if ( teleportPoint != null && teleportPoint.teleportType ==
          TeleportPoint.TeleportPointType.SwitchToNewScene )
      {
        currentFadeTime *= 3.0f;
        Teleport.ChangeScene.Send( currentFadeTime );
      }

      SteamVR_Fade.Start( Color.clear, 0 );
      SteamVR_Fade.Start( Color.black, currentFadeTime );

      headAudioSource.transform.SetParent( player.hmdTransform );
      headAudioSource.transform.localPosition = Vector3.zero;
      PlayAudioClip( headAudioSource, teleportSound );

      Invoke( "TeleportPlayer", currentFadeTime );
    }


    //-------------------------------------------------
    private void TeleportPlayer()
    {
      teleporting = false;

      Teleport.PlayerPre.Send( pointedAtTeleportMarker );

      SteamVR_Fade.Start( Color.clear, currentFadeTime );

      TeleportPoint teleportPoint = teleportingToMarker as TeleportPoint;
      Vector3 teleportPosition = pointedAtPosition;

```

```
      if ( teleportPoint != null )
      {
        teleportPosition = teleportPoint.transform.position;

        //Teleport to a new scene
        if ( teleportPoint.teleportType == TeleportPoint.TeleportPointType.SwitchToNewScene )
        {
          teleportPoint.TeleportToScene();
          return;
        }
      }

      // Find the actual floor position below the navigation mesh
      TeleportArea teleportArea = teleportingToMarker as TeleportArea;
      if ( teleportArea != null )
      {
        if ( floorFixupMaximumTraceDistance > 0.0f )
        {
          RaycastHit raycastHit;
          if ( Physics.Raycast( teleportPosition + 0.05f * Vector3.down, Vector3.down, out raycastHit, floorFixupMaximumTraceDistance, floorFixupTraceLayerMask ) )
          {
            teleportPosition = raycastHit.point;
          }
        }
      }

      if ( teleportingToMarker.ShouldMovePlayer() )
      {
        Vector3 playerFeetOffset = player.trackingOriginTransform.position - player.feetPositionGuess;
        player.trackingOriginTransform.position = teleportPosition + playerFeetOffset;
      }
      else
      {
        teleportingToMarker.TeleportPlayer( pointedAtPosition );
      }

      Teleport.Player.Send( pointedAtTeleportMarker );
    }


    //-------------------------------------------------
    private void HighlightSelected( TeleportMarkerBase hitTeleportMarker )
    {
      if ( pointedAtTeleportMarker != hitTeleportMarker ) //Pointing at a new teleport marker
      {
        if ( pointedAtTeleportMarker != null )
```

```
      {
        pointedAtTeleportMarker.Highlight( false );
      }

      if ( hitTeleportMarker != null )
      {
        hitTeleportMarker.Highlight( true );

        prevPointedAtPosition = pointedAtPosition;
        PlayPointerHaptic( !hitTeleportMarker.locked );

        PlayAudioClip( reticleAudioSource, goodHighlightSound );

        loopingAudioSource.volume = loopingAudioMaxVolume;
      }
      else if ( pointedAtTeleportMarker != null )
      {
        PlayAudioClip( reticleAudioSource, badHighlightSound );

        loopingAudioSource.volume = 0.0f;
      }
    }
    else if ( hitTeleportMarker != null ) //Pointing at the same
        teleport marker
    {
      if ( Vector3.Distance( prevPointedAtPosition, pointedAtPosition )
           > 1.0f )
      {
        prevPointedAtPosition = pointedAtPosition;
        PlayPointerHaptic( !hitTeleportMarker.locked );
      }
    }
  }


  //-------------------------------------------------
  public void ShowTeleportHint()
  {
    CancelTeleportHint();

    hintCoroutine = StartCoroutine( TeleportHintCoroutine() );
  }


  //-------------------------------------------------
  public void CancelTeleportHint()
  {
    if ( hintCoroutine != null )
          {
              ControllerButtonHints.HideTextHint(player.leftHand,
                  teleportAction);
              ControllerButtonHints.HideTextHint(player.rightHand,
                  teleportAction);

```

```
			StopCoroutine( hintCoroutine );
			hintCoroutine = null;
		}

		CancelInvoke( "ShowTeleportHint" );
	}


	//-------------------------------------------------
	private IEnumerator TeleportHintCoroutine()
	{
		float prevBreakTime = Time.time;
		float prevHapticPulseTime = Time.time;

		while ( true )
		{
			bool pulsed = false;

			//Show the hint on each eligible hand
			foreach ( Hand hand in player.hands )
			{
				bool showHint = IsEligibleForTeleport( hand );
				bool isShowingHint = !string.IsNullOrEmpty(
					ControllerButtonHints.GetActiveHintText( hand,
					teleportAction) );
				if ( showHint )
				{
					if ( !isShowingHint )
					{
						ControllerButtonHints.ShowTextHint( hand, teleportAction, "
							Teleport" );
						prevBreakTime = Time.time;
						prevHapticPulseTime = Time.time;
					}

					if ( Time.time > prevHapticPulseTime + 0.05f )
					{
						//Haptic pulse for a few seconds
						pulsed = true;

						hand.TriggerHapticPulse( 500 );
					}
				}
				else if ( !showHint && isShowingHint )
				{
					ControllerButtonHints.HideTextHint( hand, teleportAction);
				}
			}

			if ( Time.time > prevBreakTime + 3.0f )
			{
				//Take a break for a few seconds
				yield return new WaitForSeconds( 3.0f );

```

```
          prevBreakTime = Time.time;
        }

        if ( pulsed )
        {
          prevHapticPulseTime = Time.time;
        }

        yield return null;
      }
    }


    //-------------------------------------------------
    public bool IsEligibleForTeleport( Hand hand )
    {
      if ( hand == null )
      {
        return false;
      }

      if ( !hand.gameObject.activeInHierarchy )
      {
        return false;
      }

      if ( hand.hoveringInteractable != null )
      {
        return false;
      }

      if ( hand.noSteamVRFallbackCamera == null )
      {
        if ( hand.isActive == false)
        {
          return false;
        }

        //Something is attached to the hand
        if ( hand.currentAttachedObject != null )
        {
          AllowTeleportWhileAttachedToHand
              allowTeleportWhileAttachedToHand = hand.
              currentAttachedObject.GetComponent<
              AllowTeleportWhileAttachedToHand>();

          if ( allowTeleportWhileAttachedToHand != null &&
              allowTeleportWhileAttachedToHand.teleportAllowed == true )
          {
            return true;
          }
          else
          {
            return false;
```

```
				}
			}
		}

		return true;
	}


	//-------------------------------------------------
	private bool ShouldOverrideHoverLock()
	{
		if ( !allowTeleportWhileAttached || allowTeleportWhileAttached.
			overrideHoverLock )
		{
			return true;
		}

		return false;
	}


	//-------------------------------------------------
	private bool WasTeleportButtonReleased( Hand hand )
	{
		if ( IsEligibleForTeleport( hand ) )
		{
			if ( hand.noSteamVRFallbackCamera != null )
			{
				return Input.GetKeyUp( KeyCode.T );
			}
			else
                {
                    return teleportAction.GetStateUp(hand.handType);

                    //return hand.controller.GetPressUp(
                        SteamVR_Controller.ButtonMask.Touchpad );
                }
		}

		return false;
	}

	//-------------------------------------------------
	private bool IsTeleportButtonDown( Hand hand )
	{
		if ( IsEligibleForTeleport( hand ) )
		{
			if ( hand.noSteamVRFallbackCamera != null )
			{
				return Input.GetKey( KeyCode.T );
			}
			else
                {
                    return teleportAction.GetState(hand.handType);
```

```
				}
			}

			return false;
		}


		//-------------------------------------------------
		private bool WasTeleportButtonPressed( Hand hand )
		{
			if ( IsEligibleForTeleport( hand ) )
			{
				if ( hand.noSteamVRFallbackCamera != null )
				{
					return Input.GetKeyDown( KeyCode.T );
				}
				else
                {
                    return teleportAction.GetStateDown(hand.handType);

                    //return hand.controller.GetPressDown( SteamVR_Controller.ButtonMask.Touchpad );
				}
			}

			return false;
		}


		//-------------------------------------------------
		private Transform GetPointerStartTransform( Hand hand )
		{
			if ( hand.noSteamVRFallbackCamera != null )
			{
				return hand.noSteamVRFallbackCamera.transform;
			}
			else
			{
				return hand.transform;
			}
		}
	}
}
```

9.5.16. TeleportArc.cs

```
//======= Copyright (c) Valve Corporation, All rights reserved. ===============
//
// Purpose: Displays the arc lines for teleporting and does the traces
//
```

```
//
    ============================================================================

using UnityEngine;

namespace Valve.VR.InteractionSystem
{
    //
        -------------------------------------------------------------------------
    public class TeleportArc : MonoBehaviour
    {
        public int segmentCount = 60;
        public float thickness = 0.01f;

        [Tooltip("The amount of time in seconds to predict the motion of
            the projectile.")]
        public float arcDuration = 3.0f;

        [Tooltip("The amount of time in seconds between each segment of
            the projectile.")]
        public float segmentBreak = 0.025f;

        [Tooltip("The speed at which the line segments of the arc move.")
            ]
        public float arcSpeed = 0.2f;

        public Material material;

        [HideInInspector]
        public int traceLayerMask = 0;

        //Private data
        private LineRenderer[] lineRenderers;
        private float arcTimeOffset = 0.0f;
        private float prevThickness = 0.0f;
        private int prevSegmentCount = 0;
        private bool showArc = true;
        private Vector3 startPos;
        private Vector3 projectileVelocity;
        private bool useGravity = true;
        private Transform arcObjectsTransfrom;
        private bool arcInvalid = false;
        private float scale = 1;


        //-------------------------------------------------
        void Start()
        {
            arcTimeOffset = Time.time;
        }


```

```
        //--------------------------------------------------
        void Update()
        {
            //scale arc to match player scale
            scale = Player.instance.transform.lossyScale.x;
            if (thickness != prevThickness || segmentCount != prevSegmentCount)
            {
                CreateLineRendererObjects();
                prevThickness = thickness;
                prevSegmentCount = segmentCount;
            }
        }



        //--------------------------------------------------
        private void CreateLineRendererObjects()
        {
            //Destroy any existing line renderer objects
            if (arcObjectsTransfrom != null)
            {
                Destroy(arcObjectsTransfrom.gameObject);
            }

            GameObject arcObjectsParent = new GameObject("ArcObjects");
            arcObjectsTransfrom = arcObjectsParent.transform;
            arcObjectsTransfrom.SetParent(this.transform);

            //Create new line renderer objects
            lineRenderers = new LineRenderer[segmentCount];
            for (int i = 0; i < segmentCount; ++i)
            {
                GameObject newObject = new GameObject("LineRenderer_" + i);
                newObject.transform.SetParent(arcObjectsTransfrom);

                lineRenderers[i] = newObject.AddComponent<LineRenderer>();

                lineRenderers[i].receiveShadows = false;
                lineRenderers[i].reflectionProbeUsage = UnityEngine.Rendering.ReflectionProbeUsage.Off;
                lineRenderers[i].lightProbeUsage = UnityEngine.Rendering.LightProbeUsage.Off;
                lineRenderers[i].shadowCastingMode = UnityEngine.Rendering.ShadowCastingMode.Off;
                lineRenderers[i].material = material;
#if (UNITY_5_4)
                lineRenderers[i].SetWidth(thickness, thickness);
#else
        lineRenderers[i].startWidth = thickness * scale;
        lineRenderers[i].endWidth = thickness * scale;
#endif
```

```
                lineRenderers[i].enabled = false;
            }
        }


        //-------------------------------------------------
        public void SetArcData(Vector3 position, Vector3 velocity, bool
            gravity, bool pointerAtBadAngle)
        {
            startPos = position;
            projectileVelocity = velocity;
            useGravity = gravity;

            if (arcInvalid && !pointerAtBadAngle)
            {
                arcTimeOffset = Time.time;
            }
            arcInvalid = pointerAtBadAngle;
        }


        //-------------------------------------------------
        public void Show()
        {
            showArc = true;
            if (lineRenderers == null)
            {
                CreateLineRendererObjects();
            }
        }


        //-------------------------------------------------
        public void Hide()
        {
            //Hide the line segments if they were previously being shown
            if (showArc)
            {
                HideLineSegments(0, segmentCount);
            }
            showArc = false;
        }


        //-------------------------------------------------
        // Draws each segment of the arc individually
        //-------------------------------------------------
        public bool DrawArc(out RaycastHit hitInfo)
        {
            float timeStep = arcDuration / segmentCount;

            float currentTimeOffset = (Time.time - arcTimeOffset) *
                arcSpeed;

```

```
            //Reset the arc time offset when it has gone beyond a segment
                length
            if (currentTimeOffset > (timeStep + segmentBreak))
            {
                arcTimeOffset = Time.time;
                currentTimeOffset = 0.0f;
            }

            float segmentStartTime = currentTimeOffset;

            float arcHitTime = FindProjectileCollision(out hitInfo);

            if (arcInvalid)
            {
                //Only draw first segment
                lineRenderers[0].enabled = true;
                lineRenderers[0].SetPosition(0, GetArcPositionAtTime(0.0f
                    ));
                lineRenderers[0].SetPosition(1, GetArcPositionAtTime(
                    arcHitTime < timeStep ? arcHitTime : timeStep));

                HideLineSegments(1, segmentCount);
            }
            else
            {
                //Draw the first segment outside the loop if needed
                int loopStartSegment = 0;
                if (segmentStartTime > segmentBreak)
                {
                    float firstSegmentEndTime = currentTimeOffset -
                        segmentBreak;
                    if (arcHitTime < firstSegmentEndTime)
                    {
                        firstSegmentEndTime = arcHitTime;
                    }
                    DrawArcSegment(0, 0.0f, firstSegmentEndTime);

                    loopStartSegment = 1;
                }

                bool stopArc = false;
                int currentSegment = 0;
                if (segmentStartTime < arcHitTime)
                {
                    for (currentSegment = loopStartSegment;
                        currentSegment < segmentCount; ++currentSegment)
                    {
                        //Clamp the segment end time to the arc duration
                        float segmentEndTime = segmentStartTime +
                            timeStep;
                        if (segmentEndTime >= arcDuration)
                        {
                            segmentEndTime = arcDuration;
                            stopArc = true;
```

```
                        }

                        if (segmentEndTime >= arcHitTime)
                        {
                            segmentEndTime = arcHitTime;
                            stopArc = true;
                        }

                        DrawArcSegment(currentSegment, segmentStartTime,
                            segmentEndTime);

                        segmentStartTime += timeStep + segmentBreak;

                        //If the previous end time or the next start time
                             is beyond the duration then stop the arc
                        if (stopArc || segmentStartTime >= arcDuration ||
                             segmentStartTime >= arcHitTime)
                        {
                            break;
                        }
                    }
                }
                else
                {
                    currentSegment--;
                }

                //Hide the rest of the line segments
                HideLineSegments(currentSegment + 1, segmentCount);
            }

            return arcHitTime != float.MaxValue;
        }


        //-------------------------------------------------
        private void DrawArcSegment(int index, float startTime, float
            endTime)
        {
            lineRenderers[index].enabled = true;
            lineRenderers[index].SetPosition(0, GetArcPositionAtTime(
                startTime));
            lineRenderers[index].SetPosition(1, GetArcPositionAtTime(
                endTime));
        }


        //-------------------------------------------------
        public void SetColor(Color color)
        {
            for (int i = 0; i < segmentCount; ++i)
            {
#if (UNITY_5_4)
                lineRenderers[i].SetColors(color, color);
```

```
#else
        lineRenderers[i].startColor = color;
        lineRenderers[i].endColor = color;
#endif
            }
        }


        //-------------------------------------------------
        private float FindProjectileCollision(out RaycastHit hitInfo)
        {
            float timeStep = arcDuration / segmentCount;
            float segmentStartTime = 0.0f;

            hitInfo = new RaycastHit();

            Vector3 segmentStartPos = GetArcPositionAtTime(segmentStartTime);
            for (int i = 0; i < segmentCount; ++i)
            {
                float segmentEndTime = segmentStartTime + timeStep;
                Vector3 segmentEndPos = GetArcPositionAtTime(segmentEndTime);

                if (Physics.Linecast(segmentStartPos, segmentEndPos, out hitInfo, traceLayerMask))
                {
                    if (hitInfo.collider.GetComponent<IgnoreTeleportTrace>() == null)
                    {
                        Util.DrawCross(hitInfo.point, Color.red, 0.5f);
                        float segmentDistance = Vector3.Distance(segmentStartPos, segmentEndPos);
                        float hitTime = segmentStartTime + (timeStep * (hitInfo.distance / segmentDistance));
                        return hitTime;
                    }
                }

                segmentStartTime = segmentEndTime;
                segmentStartPos = segmentEndPos;
            }

            return float.MaxValue;
        }


        //-------------------------------------------------
        public Vector3 GetArcPositionAtTime(float time)
        {
            Vector3 gravity = useGravity ? Physics.gravity : Vector3.zero;

```

```
            Vector3 arcPos = startPos + ((projectileVelocity * time) + (0.5f * time * time) * gravity) * scale;
            return arcPos;
        }


        //------------------------------------------------
        private void HideLineSegments(int startSegment, int endSegment)
        {
            if (lineRenderers != null)
            {
                for (int i = startSegment; i < endSegment; ++i)
                {
                    lineRenderers[i].enabled = false;
                }
            }
        }
    }
}
```

9.5.17. TeleportArea

```
//======= Copyright (c) Valve Corporation, All rights reserved. ===============
//
// Purpose: An area that the player can teleport to
//
//=============================================================================

using UnityEngine;
#if UNITY_EDITOR
using UnityEditor;
#endif

namespace Valve.VR.InteractionSystem
{
  //-------------------------------------------------------------------------
  public class TeleportArea : TeleportMarkerBase
  {
    //Public properties
    public Bounds meshBounds { get; private set; }

    //Private data
    private MeshRenderer areaMesh;
    private int tintColorId = 0;
    private Color visibleTintColor = Color.clear;
    private Color highlightedTintColor = Color.clear;
    private Color lockedTintColor = Color.clear;
```

```
    private bool highlighted = false;

    //-------------------------------------------------
    public void Awake()
    {
      areaMesh = GetComponent<MeshRenderer>();

      tintColorId = Shader.PropertyToID( "_TintColor" );

      CalculateBounds();
    }


    //-------------------------------------------------
    public void Start()
    {
      visibleTintColor = Teleport.instance.areaVisibleMaterial.GetColor( tintColorId );
      highlightedTintColor = Teleport.instance.areaHighlightedMaterial.GetColor( tintColorId );
      lockedTintColor = Teleport.instance.areaLockedMaterial.GetColor( tintColorId );
    }


    //-------------------------------------------------
    public override bool ShouldActivate( Vector3 playerPosition )
    {
      return true;
    }


    //-------------------------------------------------
    public override bool ShouldMovePlayer()
    {
      return true;
    }


    //-------------------------------------------------
    public override void Highlight( bool highlight )
    {
      if ( !locked )
      {
        highlighted = highlight;

        if ( highlight )
        {
          areaMesh.material = Teleport.instance.areaHighlightedMaterial;
        }
        else
        {
          areaMesh.material = Teleport.instance.areaVisibleMaterial;
        }
```

```
		}
	}


	//-------------------------------------------------
	public override void SetAlpha( float tintAlpha, float alphaPercent )
	{
		Color tintedColor = GetTintColor();
		tintedColor.a *= alphaPercent;
		areaMesh.material.SetColor( tintColorId, tintedColor );
	}


	//-------------------------------------------------
	public override void UpdateVisuals()
	{
		if ( locked )
		{
			areaMesh.material = Teleport.instance.areaLockedMaterial;
		}
		else
		{
			areaMesh.material = Teleport.instance.areaVisibleMaterial;
		}
	}


	//-------------------------------------------------
	public void UpdateVisualsInEditor()
	{
            if (Teleport.instance == null)
                return;

		areaMesh = GetComponent<MeshRenderer>();

		if ( locked )
		{
			areaMesh.sharedMaterial = Teleport.instance.areaLockedMaterial;
		}
		else
		{
			areaMesh.sharedMaterial = Teleport.instance.areaVisibleMaterial;
		}
	}


	//-------------------------------------------------
	private bool CalculateBounds()
	{
		MeshFilter meshFilter = GetComponent<MeshFilter>();
		if ( meshFilter == null )
		{
			return false;
		}
```

```

      Mesh mesh = meshFilter.sharedMesh;
      if ( mesh == null )
      {
        return false;
      }

      meshBounds = mesh.bounds;
      return true;
    }


    //-------------------------------------------------
    private Color GetTintColor()
    {
      if ( locked )
      {
        return lockedTintColor;
      }
      else
      {
        if ( highlighted )
        {
          return highlightedTintColor;
        }
        else
        {
          return visibleTintColor;
        }
      }
    }
  }


#if UNITY_EDITOR
  //-------------------------------------------------------------------------
  [CustomEditor( typeof( TeleportArea ) )]
  public class TeleportAreaEditor : Editor
  {
    //-------------------------------------------------
    void OnEnable()
    {
      if ( Selection.activeTransform != null )
      {
        TeleportArea teleportArea = Selection.activeTransform.
            GetComponent<TeleportArea>();
        if ( teleportArea != null )
        {
          teleportArea.UpdateVisualsInEditor();
        }
      }
    }
```

```
		//-------------------------------------------------
		public override void OnInspectorGUI()
		{
			DrawDefaultInspector();

			if ( Selection.activeTransform != null )
			{
				TeleportArea teleportArea = Selection.activeTransform.
				    GetComponent<TeleportArea>();
				if ( GUI.changed && teleportArea != null )
				{
					teleportArea.UpdateVisualsInEditor();
				}
			}
		}
	}
#endif
}
```

9.5.18. TeleportMarkerBase.cs

```
//======= Copyright (c) Valve Corporation, All rights reserved.
    ===============
//
// Purpose: Base class for all the objects that the player can teleport
    to
//
//
    =============================================================================

using UnityEngine;

namespace Valve.VR.InteractionSystem
{
	//
	    -----------------------------------------------------------------------
	public abstract class TeleportMarkerBase : MonoBehaviour
	{
		public bool locked = false;
		public bool markerActive = true;

		//-------------------------------------------------
		public virtual bool showReticle
		{
			get
			{
				return true;
			}
```

```
    }


    //-------------------------------------------------
    public void SetLocked( bool locked )
    {
      this.locked = locked;

      UpdateVisuals();
    }


    //-------------------------------------------------
    public virtual void TeleportPlayer( Vector3 pointedAtPosition )
    {
    }


    //-------------------------------------------------
    public abstract void UpdateVisuals();

    //-------------------------------------------------
    public abstract void Highlight( bool highlight );

    //-------------------------------------------------
    public abstract void SetAlpha( float tintAlpha, float alphaPercent );

    //-------------------------------------------------
    public abstract bool ShouldActivate( Vector3 playerPosition );

    //-------------------------------------------------
    public abstract bool ShouldMovePlayer();
  }
}
```

9.5.19. TeleportPoint.cs

```
//======= Copyright (c) Valve Corporation, All rights reserved. ===============
//
// Purpose: Single location that the player can teleport to
//
//=============================================================================

using UnityEngine;
using UnityEngine.UI;
#if UNITY_EDITOR
using UnityEditor;
#endif

```

```
namespace Valve.VR.InteractionSystem
{
  //
    -------------------------------------------------------------------------------

  public class TeleportPoint : TeleportMarkerBase
  {
    public enum TeleportPointType
    {
      MoveToLocation,
      SwitchToNewScene
    };

    //Public variables
    public TeleportPointType teleportType = TeleportPointType.
       MoveToLocation;
    public string title;
    public string switchToScene;
    public Color titleVisibleColor;
    public Color titleHighlightedColor;
    public Color titleLockedColor;
    public bool playerSpawnPoint = false;

    //Private data
    private bool gotReleventComponents = false;
    private MeshRenderer markerMesh;
    private MeshRenderer switchSceneIcon;
    private MeshRenderer moveLocationIcon;
    private MeshRenderer lockedIcon;
    private MeshRenderer pointIcon;
    private Transform lookAtJointTransform;
    private new Animation animation;
    private Text titleText;
    private Player player;
    private Vector3 lookAtPosition = Vector3.zero;
    private int tintColorID = 0;
    private Color tintColor = Color.clear;
    private Color titleColor = Color.clear;
    private float fullTitleAlpha = 0.0f;

    //Constants
    private const string switchSceneAnimation = "switch_scenes_idle";
    private const string moveLocationAnimation = "move_location_idle";
    private const string lockedAnimation = "locked_idle";


    //-------------------------------------------------
    public override bool showReticle
    {
      get
      {
        return false;
      }
    }
```

```


    //-------------------------------------------------
    void Awake()
    {
      GetRelevantComponents();

      animation = GetComponent<Animation>();

      tintColorID = Shader.PropertyToID( "_TintColor" );

      moveLocationIcon.gameObject.SetActive( false );
      switchSceneIcon.gameObject.SetActive( false );
      lockedIcon.gameObject.SetActive( false );

      UpdateVisuals();
    }


    //-------------------------------------------------
    void Start()
    {
      player = Player.instance;
    }


    //-------------------------------------------------
    void Update()
    {
      if ( Application.isPlaying )
      {
        lookAtPosition.x = player.hmdTransform.position.x;
        lookAtPosition.y = lookAtJointTransform.position.y;
        lookAtPosition.z = player.hmdTransform.position.z;

        lookAtJointTransform.LookAt( lookAtPosition );
      }
    }


    //-------------------------------------------------
    public override bool ShouldActivate( Vector3 playerPosition )
    {
      return ( Vector3.Distance( transform.position, playerPosition ) >
          1.0f );
    }


    //-------------------------------------------------
    public override bool ShouldMovePlayer()
    {
      return true;
    }

```

```
    //-------------------------------------------------
    public override void Highlight( bool highlight )
    {
      if ( !locked )
      {
        if ( highlight )
        {
          SetMeshMaterials( Teleport.instance.pointHighlightedMaterial,
              titleHighlightedColor );
        }
        else
        {
          SetMeshMaterials( Teleport.instance.pointVisibleMaterial,
              titleVisibleColor );
        }
      }

      if ( highlight )
      {
        pointIcon.gameObject.SetActive( true );
        animation.Play();
      }
      else
      {
        pointIcon.gameObject.SetActive( false );
        animation.Stop();
      }
    }


    //-------------------------------------------------
    public override void UpdateVisuals()
    {
      if ( !gotReleventComponents )
      {
        return;
      }

      if ( locked )
      {
        SetMeshMaterials( Teleport.instance.pointLockedMaterial,
            titleLockedColor );

        pointIcon = lockedIcon;

        animation.clip = animation.GetClip( lockedAnimation );
      }
      else
      {
        SetMeshMaterials( Teleport.instance.pointVisibleMaterial,
            titleVisibleColor );

        switch ( teleportType )
```

```
        {
          case TeleportPointType.MoveToLocation:
            {
              pointIcon = moveLocationIcon;

              animation.clip = animation.GetClip( moveLocationAnimation );
            }
            break;
          case TeleportPointType.SwitchToNewScene:
            {
              pointIcon = switchSceneIcon;

              animation.clip = animation.GetClip( switchSceneAnimation );
            }
            break;
        }
      }

      titleText.text = title;
    }


    //-------------------------------------------------
    public override void SetAlpha( float tintAlpha, float alphaPercent )
    {
      tintColor = markerMesh.material.GetColor( tintColorID );
      tintColor.a = tintAlpha;

      markerMesh.material.SetColor( tintColorID, tintColor );
      switchSceneIcon.material.SetColor( tintColorID, tintColor );
      moveLocationIcon.material.SetColor( tintColorID, tintColor );
      lockedIcon.material.SetColor( tintColorID, tintColor );

      titleColor.a = fullTitleAlpha * alphaPercent;
      titleText.color = titleColor;
    }


    //-------------------------------------------------
    public void SetMeshMaterials( Material material, Color textColor )
    {
      markerMesh.material = material;
      switchSceneIcon.material = material;
      moveLocationIcon.material = material;
      lockedIcon.material = material;

      titleColor = textColor;
      fullTitleAlpha = textColor.a;
      titleText.color = titleColor;
    }


    //-------------------------------------------------
```

```
	public void TeleportToScene()
	{
		if ( !string.IsNullOrEmpty( switchToScene ) )
		{
			Debug.Log("<b>[SteamVR Interaction]</b> TeleportPoint: Hook up your level loading logic to switch to new scene: " + switchToScene );
		}
		else
		{
			Debug.LogError("<b>[SteamVR Interaction]</b> TeleportPoint: Invalid scene name to switch to: " + switchToScene );
		}
	}


	//-------------------------------------------------
	public void GetRelevantComponents()
	{
		markerMesh = transform.Find( "teleport_marker_mesh" ).GetComponent<MeshRenderer>();
		switchSceneIcon = transform.Find( "teleport_marker_lookat_joint/teleport_marker_icons/switch_scenes_icon" ).GetComponent<MeshRenderer>();
		moveLocationIcon = transform.Find( "teleport_marker_lookat_joint/teleport_marker_icons/move_location_icon" ).GetComponent<MeshRenderer>();
		lockedIcon = transform.Find( "teleport_marker_lookat_joint/teleport_marker_icons/locked_icon" ).GetComponent<MeshRenderer>();
		lookAtJointTransform = transform.Find( "teleport_marker_lookat_joint" );

		titleText = transform.Find( "teleport_marker_lookat_joint/teleport_marker_canvas/teleport_marker_canvas_text" ).GetComponent<Text>();

		gotReleventComponents = true;
	}


	//-------------------------------------------------
	public void ReleaseRelevantComponents()
	{
		markerMesh = null;
		switchSceneIcon = null;
		moveLocationIcon = null;
		lockedIcon = null;
		lookAtJointTransform = null;
		titleText = null;
	}


	//-------------------------------------------------
```

```
    public void UpdateVisualsInEditor()
    {
      if ( Application.isPlaying )
      {
        return;
      }

      GetRelevantComponents();

      if ( locked )
      {
        lockedIcon.gameObject.SetActive( true );
        moveLocationIcon.gameObject.SetActive( false );
        switchSceneIcon.gameObject.SetActive( false );

        markerMesh.sharedMaterial = Teleport.instance.pointLockedMaterial;
        lockedIcon.sharedMaterial = Teleport.instance.pointLockedMaterial;

        titleText.color = titleLockedColor;
      }
      else
      {
        lockedIcon.gameObject.SetActive( false );

        markerMesh.sharedMaterial = Teleport.instance.pointVisibleMaterial;
        switchSceneIcon.sharedMaterial = Teleport.instance.pointVisibleMaterial;
        moveLocationIcon.sharedMaterial = Teleport.instance.pointVisibleMaterial;

        titleText.color = titleVisibleColor;

        switch ( teleportType )
        {
          case TeleportPointType.MoveToLocation:
            {
              moveLocationIcon.gameObject.SetActive( true );
              switchSceneIcon.gameObject.SetActive( false );
            }
            break;
          case TeleportPointType.SwitchToNewScene:
            {
              moveLocationIcon.gameObject.SetActive( false );
              switchSceneIcon.gameObject.SetActive( true );
            }
            break;
        }
      }

      titleText.text = title;

```

```
			ReleaseRelevantComponents();
		}
	}


#if UNITY_EDITOR
	//-------------------------------------------------------------------------
	[CustomEditor( typeof( TeleportPoint ) )]
	public class TeleportPointEditor : Editor
	{
		//-------------------------------------------------
		void OnEnable()
		{
			if ( Selection.activeTransform )
			{
				TeleportPoint teleportPoint = Selection.activeTransform.GetComponent<TeleportPoint>();
                if (teleportPoint != null)
				    teleportPoint.UpdateVisualsInEditor();
			}
		}


		//-------------------------------------------------
		public override void OnInspectorGUI()
		{
			DrawDefaultInspector();

			if ( Selection.activeTransform )
			{
				TeleportPoint teleportPoint = Selection.activeTransform.GetComponent<TeleportPoint>();
				if ( GUI.changed )
				{
					teleportPoint.UpdateVisualsInEditor();
				}
			}
		}
	}
#endif
}
```

9.5.20. UIElement.cs

```
//======= Copyright (c) Valve Corporation, All rights reserved. ===============
//
// Purpose: UIElement that responds to VR hands and generates UnityEvents
//
```

```
//
    ================================================================================

using UnityEngine;
using UnityEngine.Events;
using UnityEngine.UI;
using System;

namespace Valve.VR.InteractionSystem
{
  //
      ----------------------------------------------------------------------------

  [RequireComponent( typeof( Interactable ) )]
  public class UIElement : MonoBehaviour
  {
    public CustomEvents.UnityEventHand onHandClick;

        protected Hand currentHand;

    //-------------------------------------------------
    protected virtual void Awake()
    {
      Button button = GetComponent<Button>();
      if ( button )
      {
        button.onClick.AddListener( OnButtonClick );
      }
    }


    //-------------------------------------------------
    protected virtual void OnHandHoverBegin( Hand hand )
    {
      currentHand = hand;
      InputModule.instance.HoverBegin( gameObject );
      ControllerButtonHints.ShowButtonHint( hand, hand.uiInteractAction);
    }


        //-------------------------------------------------
        protected virtual void OnHandHoverEnd( Hand hand )
    {
      InputModule.instance.HoverEnd( gameObject );
      ControllerButtonHints.HideButtonHint( hand, hand.uiInteractAction);
      currentHand = null;
    }


        //-------------------------------------------------
        protected virtual void HandHoverUpdate( Hand hand )
    {
```

```
      if ( hand.uiInteractAction != null && hand.uiInteractAction.GetStateDown(hand.handType) )
      {
        InputModule.instance.Submit( gameObject );
        ControllerButtonHints.HideButtonHint( hand, hand.uiInteractAction );
      }
    }


        //-------------------------------------------------
        protected virtual void OnButtonClick()
    {
      onHandClick.Invoke( currentHand );
    }
  }

#if UNITY_EDITOR
  //-------------------------------------------------------------------------
  [UnityEditor.CustomEditor( typeof( UIElement ) )]
  public class UIElementEditor : UnityEditor.Editor
  {
    //-------------------------------------------------
    // Custom Inspector GUI allows us to click from within the UI
    //-------------------------------------------------
    public override void OnInspectorGUI()
    {
      DrawDefaultInspector();

      UIElement uiElement = (UIElement)target;
      if ( GUILayout.Button( "Click" ) )
      {
        InputModule.instance.Submit( uiElement.gameObject );
      }
    }
  }
#endif
}
```

Apéndice A

Assets importados

Nombre	**Tipo**	**Descripción**	**Autor**	**Fuente**
Electric+lab	Model	Mesa de laboratorio	'Men in Black'	3D Warehouse
metal+fence+audrius	Model	Jaula de seguridad	'Storboda1980'	3D Warehouse
door+classroom	Model	Puerta	'Wong Yee Woon G.'	3D Warehouse
AS-120mm	Model	Ventana	'pizzi_lace'	3D Warehouse
Siemens-Sirius-ACT	Model	Seta emergencia	'Chris W.'	3D Warehouse
computer+desk	Model	Mesa con ordenador	'Paztecarms'	3D Warehouse
sikringskap	Model	Cuadro eléctrico	'Kai'	3D Warehouse
B0201	Model	Modelo de pruebas	'Press Start'	Unity Asset Store
Starfield Skybox	Skybox	Cielo	'Nightsoundgames'	Unity Asset Store
PBR Materials light	Material	Texturas	'Pixel Processor'	Unity Asset Store
Free architectural textures	Material	Texturas	'Crazytextures'	Unity Asset Store
IRB120	Model	Robot	'ABB'	abb.com

Cuadro A.1: Assets importados de fuentes externas.

Apéndice B

Imágenes adicionales de la aplicación

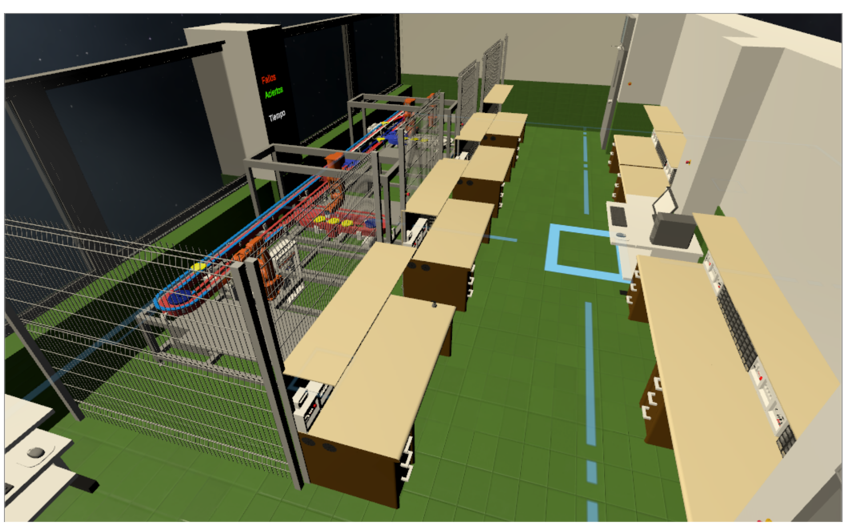

Figura B.1: Vista del laboratorio general.

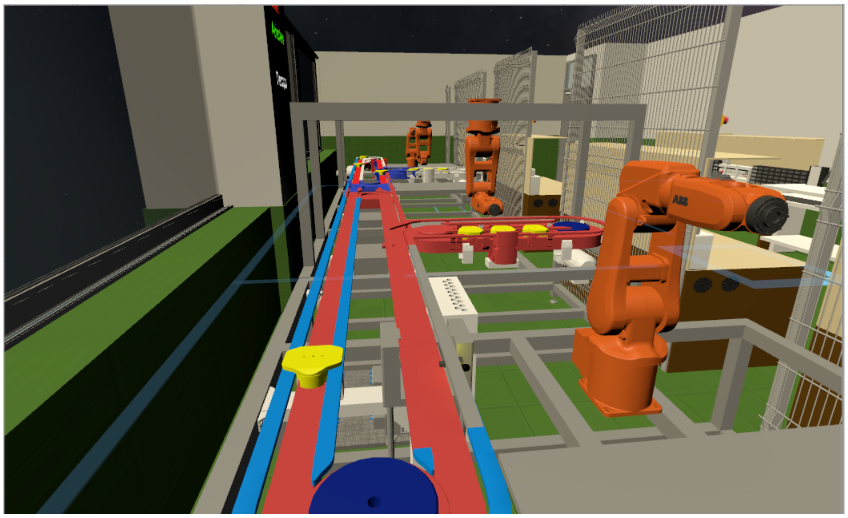

Figura B.2: Vista de la planta.

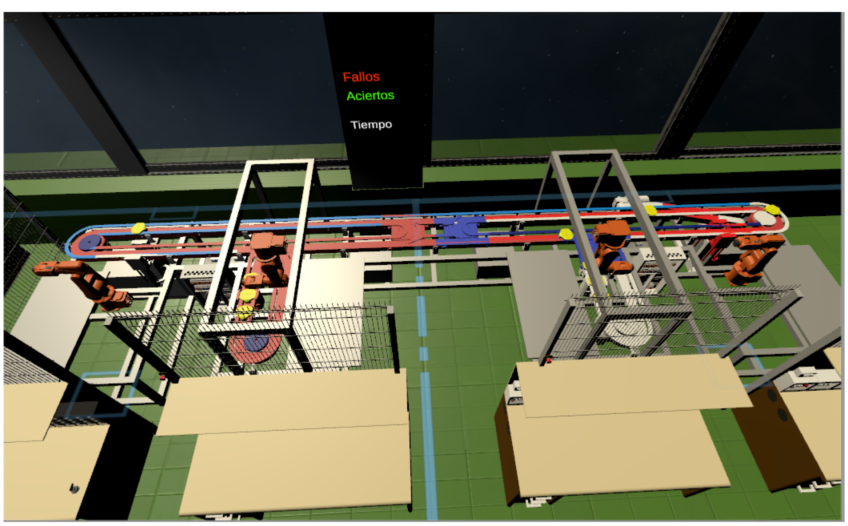

Figura B.3: Vista de la planta desde arriba.

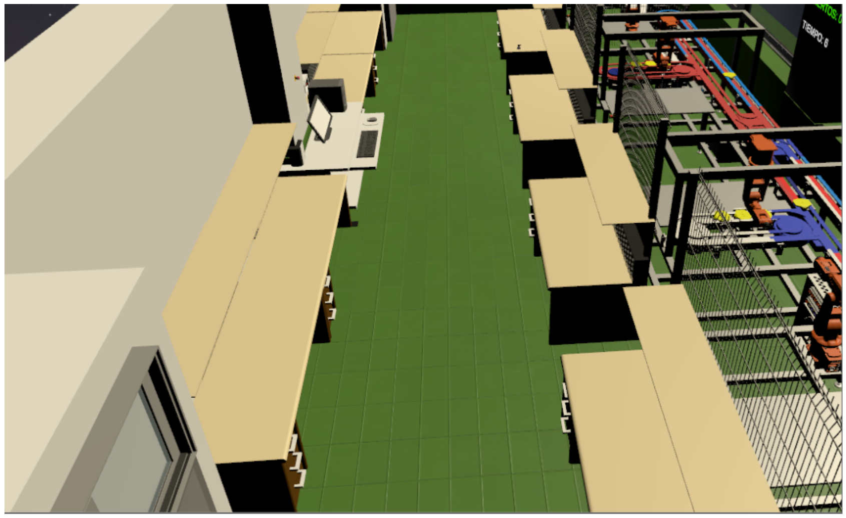

Figura B.4: Vista de la zona de trabajo.

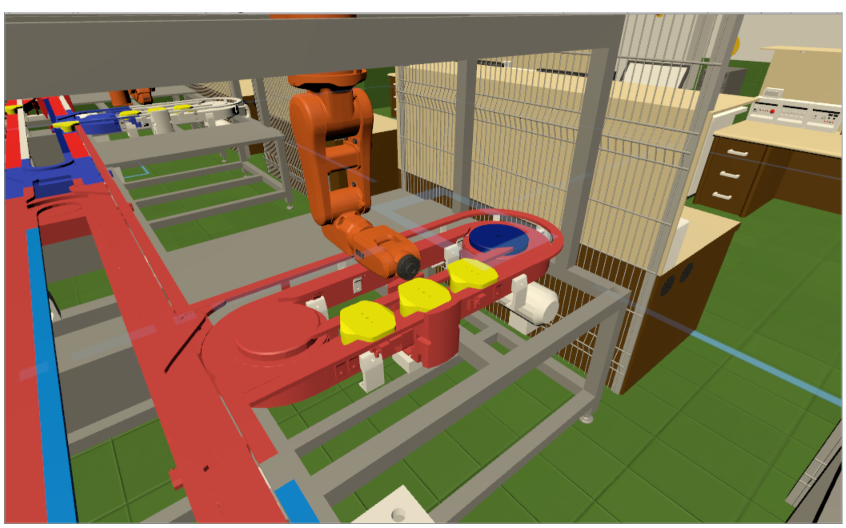

Figura B.5: Vista del robot ABB en la planta.

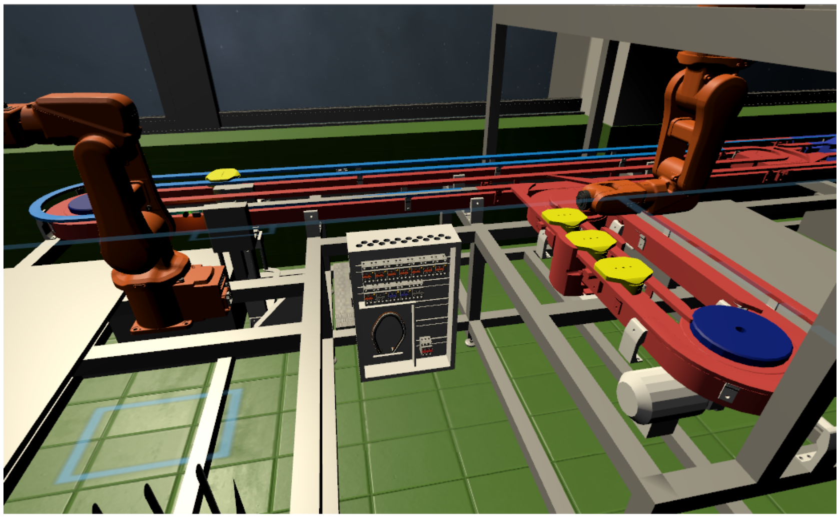

Figura B.6: Detalle de la zona izquierda de la planta.

Bibliografía

[1] *3D Warehouse*. 2019. Último acceso 01/07/19. URL: https://3dwarehouse.sketchup.com/.

[2] *ABB*. 2019. Último acceso 01/07/19. URL: https://new.abb.com/es.

[3] Amparo Caballero. *Industria 4.0 a través de Realidad Virtual y Realidad Aumentada*. 2017. Último acceso 01/07/19. URL: http://www.innoarea.com/industria-4-0-a-traves-de-realidad-virtual-y-realidad-aumentada/.

[4] *Historia de la realidad virtual*. Último acceso 30/06/19. URL: https://es.wikipedia.org/wiki/Historia_de_la_realidad_virtual.

[5] Joseph Hocking. *Unity in Action*. 20 Baldwin Road, Shelter Island, NY 11964: Manning, 2015. ISBN: 9781617292323.

[6] Iraltacamino. *Historia de la realidad virtual*. 2016. Último acceso 01/07/19. URL: http://caminosantiago360.com/historia-la-realidad-virtual/.

[7] Carlos Álvarez Vereterra. Trabajo Fin de Máster. *Uso de la realidad virtual para el entrenamiento del personal de operación y mantenimiento: aplicación a la minifábrica ICAI*. Madrid, 2018.

[8] Carlos Álvarez Vereterra. Trabajo Fin de Máster. "Uso de la realidad virtual para el entrenamiento del personal de operación y mantenimiento: aplicación a la minifábrica ICAI". En: Madrid: ICAI, 2018. Cap. 3, págs. 18-25.

[9] Carlos Álvarez Vereterra. Trabajo Fin de Máster. "Uso de la realidad virtual para el entrenamiento del personal de operación y mantenimiento: aplicación a la minifábrica ICAI". En: Madrid: ICAI, 2018. Cap. 3, pág. 18.

[10] Jack Nove. *ZeroLight, BMW, HTC VIVE and NVIDIA Team Up for Virtual Reality Showcase at CES 2019*. 2019. Último acceso 01/07/19. URL: https://zerolight.com/news/press-releases/zerolight-bmw-htc-vive-and-nvidia-team-up-for-virtual-reality-showcase-at-ces-2019.

[11] *Oculus*. 2019. Último acceso 01/07/19. URL: https://www.oculus.com/rift/?locale=es_ES#oui-csl-rift-games=mages-tale.

[12] *Oculus Rift*. Último acceso 02/07/19. URL: https://es.wikipedia.org/wiki/Oculus_Rift.

[13] José L. Ortega. *Un repaso a la historia de la realidad virtual*. 2016.Último acceso 01/07/19. URL: https://es.ign.com/realidad-virtual/109691/feature/un-repaso-a-la-historia-de-la-realidad-virtual.

[14] *Pixo Group*. 2019. Último acceso 08/07/19. URL: https://pixogroup.com/.

[15] Julián Pérez Porto y María Merino. *Definición de realidad virtual*. 2013. Último acceso 01/07/19. URL: definicion.de/realidad-virtual.

[16] *Steam VR*. 2019. Último acceso 01/07/19. URL: https://www.steamvr.com/en/.

[17] *Unity*. 2019. Último acceso 01/07/19. URL: https://unity3d.com/es/unity.

[18] *Unity Documentation*. 2019. Último acceso 01/07/19. URL: https://docs.unity3d.com/ScriptReference/.

[19] Cámara Valencia. *Caminar con éxito hacia la Industria 4.0: Capítulo 18 - Realidad Aumentada y Virtual*. 2018. Último acceso 01/07/19. URL: https://ticnegocios.camaravalencia.com.

[20] Mundo Virtual. *¿Qué es la realidad virtual?* Último acceso 01/07/19. URL: mundo-virtual.com/que-es-la-realidad-virtual.

[21] *Welcome to the Evolution of Safety Training: Virtual Reality*. 2019. Último acceso 01/07/19. URL: https://www.3m.com/3M/en_US/worker-health-safety-us/3m-ppe-training/virtual-reality/.