



Universidad
de Alcalá

TRABAJO DE FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

ESCUELA POLITÉCNICA SUPERIOR

Interactividad y métodos de control en
videojuegos: el caso de la visualización en tercera
persona

Adrián Montesinos González

Tutor: Antonio Moratilla Ocaña.

2024

A todos los que me apoyaron para llegar hasta aquí.

Resumen

Se han estudiado los sistemas de cámaras en el contexto de la visualización en tercera persona y cómo interactúan estos con los esquemas de control; se han estudiado las dificultades que se encuentran al implementar estos sistemas y en su interacción con los esquemas de control; y se han desarrollado componentes software para el motor de videojuegos Unity que permitan diseñar e iterar en sistemas de cámaras en tercera persona, dando solución a dichas dificultades.

Además se presenta una escena de demostración, prototipada con los componentes desarrollados, en la que se puede controlar a un personaje a través de varias cámaras en tercera persona, mostrando la interacción entre las cámaras y el esquemas de control, y demostrando la utilidad de los componentes desarrollados.

Palabras clave: Cámaras en tercera persona, esquemas de control de juegos, motores de videojuegos, Unity.

Abstract

Camera systems in the context of third-person visualization and their interaction with control schemes have been studied; the hardships encountered in implementing these systems and their interactions with input schemes have also been studied; and software components for the Unity game engine have been developed to allow the design and iteration of third-person camera systems, providing solutions to these hardships.

In addition, a demonstration scene is presented, prototyped with the components here developed, in which a character can be controlled through multiple third-person cameras, showing the interaction between the cameras and the control scheme, and demonstrating the usefulness of the developed components.

Keywords: Third-person cameras, game control schemes, video game engines, Unity.

Índice general

Resumen	vii
Abstract	ix
Índice general	xi
Índice de figuras	xiii
Índice de tablas	xv
1 Introducción	1
1.1 Objetivos	1
2 Estudio teórico	3
2.1 Nomenclatura de motores de videojuegos	3
2.2 Estudio de la literatura	4
2.2.1 Real Time Cameras: a Guide for Game Designers and Developers	4
2.2.2 50 Game Camera Mistakes	7
2.3 Estado del arte	8
2.3.1 Unity	8
2.3.1.1 Cámaras básicas	9
2.3.1.2 Cinemachine	10
2.3.1.3 Esquemas de control	12
2.3.2 Unreal Engine	12
2.3.2.1 Cámaras básicas	12
2.3.2.2 Esquemas de control	13
2.4 Casos de estudio	13
2.4.1 Journey	13
2.4.2 Nier: Automata	14
2.4.3 Lost Planet 2	15
2.5 Otros métodos de entrada	16
2.6 Comparación y lecciones	16
3 Diseño y desarrollo de la solución	19
3.1 Requisitos y consideraciones	19
3.2 Arquitectura e implementación	21
3.2.1 Virtual Camera Parameters	22
3.2.2 Virtual Camera	23
3.2.3 Main Camera Virtualizer	24
3.2.4 Orbit Camera Constraint	24

3.2.5 Orbit Camera Whiskers	27
3.2.6 Movement Compass	28
3.2.7 Drag Camera Input	30
3.2.8 Clases auxiliares	31
4 Prueba y evaluación	33
4.1 Desarrollo de la escena	33
4.2 Demostración	35
5 Conclusión	45
6 Coste del proyecto	47
Bibliografía	49
Apéndice A Elementos adicionales entregables	51
Apéndice B Herramientas	53

Índice de figuras

2.1	Portada de “Real Time Cameras”	5
2.2	Captura de “50 Game Camera Mistakes”	7
2.3	Número de títulos en Steam agrupados por motor de videojuegos. Nótese que solo Unity, Unreal, y en menor medida Godot y XNA son usados para juegos en 3D.	9
2.4	Logo de Unity	9
2.5	<i>Gizmos</i> de una <i>Free Look Camera</i>	11
2.6	Logo de Unreal Engine	13
2.7	Captura del videojuego Journey. Se muestra una escena con una cámara orbital y un desfase horizontal extremo para maximizar las vistas en un momento de poco movimiento.	14
2.8	Captura del videojuego Nier: Automata. Se muestra una escena de acción con una cámara orbital centrada totalmente en el personaje jugador.	15
2.9	Captura de Lost Planet 2. Se muestra una cámara orbital con un desfase horizontal fijo, y la cámara apuntando el propio pivote de la cámara, en lugar del personaje, para poder apuntar al fondo con la mirilla.	15
3.1	Diagrama de clases de <code>VirtualCameraParameters</code>	22
3.2	Diagrama de clases de <code>VirtualCamera</code>	23
3.3	Captura del inspector de <code>VirtualCamera</code>	23
3.4	Diagrama de clases de <code>MainCameraVirtualizer</code>	24
3.5	Captura del inspector de <code>MainCameraVirtualizer</code>	25
3.6	Diagrama de clases de <code>OrbitCameraConstraint</code>	26
3.7	Captura del inspector de <code>OrbitCameraConstraint</code>	26
3.8	Diagrama de clases de <code>OrbitCameraWhiskers</code>	27
3.9	Captura del inspector de <code>OrbitCameraWhiskers</code>	28
3.10	Diagrama de clases de <code>MovementCompass</code>	29
3.11	Captura del inspector de <code>MovementCompass</code>	29
3.12	Diagrama de clases de <code>DragCameraInput</code>	30
3.13	Captura del inspector de <code>DragCameraInput</code>	30
3.14	Diagrama de clases de <code>ReinEditorUtils</code>	31
4.1	Vista panorámica de la escena de la demostración.	34
4.2	Gizmos de la cámara orbital en tercera persona.	34
4.3	Gizmos de los <i>whiskers</i> de la cámara orbital.	35
4.4	Cámara orbital funcionando.	36
4.5	Gizmo del compás de movimiento del personaje.	37
4.6	Gizmos de la cámara previniendo la colisión con una pared.	37
4.7	Cámara con <i>whiskers</i> en una posición inestable.	38
4.8	Cámara habiendo sido empujada por el impulso de los <i>whiskers</i>	38

4.9	Gizmos de los <i>whiskers</i> de la cámara en situación inestable.	39
4.10	Cámara con <i>whiskers</i> dejada por el usuario en una posición estable contra la pared; los impulsos de cada lado se anulan.	39
4.11	Personaje rodeando una esquina.	40
4.12	Cámara ajustándose para mantener la línea de visión con el personaje mientras rodea la esquina.	40
4.13	Cámara con una línea de visión despejada.	41
4.14	Cámara permitiendo que los pilares rompan la línea de visión con el personaje.	41
4.15	Perspectiva de la cámara fija al otro lado de los pilares.	42
4.16	Perspectiva de la cámara fija permitiendo que los pilares rompan la línea de visión con el personaje.	43
4.17	La línea de visión de la cámara del fondo está obstruida por la geometría.	43
4.18	La línea de visión de la cámara del fondo se despeja, realizándose una transición suave a esta.	44
4.19	Gizmo del compás de movimiento del personaje al salir de la pasarela; el corte de cámara ha bloqueado la orientación.	44
6.1	Suelo programador junior en Madrid, por hora, según Indeed.	47

Índice de tablas

2.1 Funcionalidades de cámaras en tercera persona	18
3.1 Funcionalidades de cámaras en tercera persona	20
6.1 Coste de mano de obra	47
6.2 Coste de hardware	48
6.3 Precios de licencias	48
6.4 Inversión total en el proyecto	48

Capítulo 1

Introducción

La visualización en tercera persona es un elemento fundamental en muchos videojuegos y experiencias interactivas. En estos, el usuario o jugador controla un personaje o entidad desde una perspectiva externa, lo que permite ver tanto al personaje como su entorno.

Los videojuegos en tercera persona se han popularizado debido a la facilidad que otorgan al jugador para entender la interacción entre su personaje y el entorno, y sus amplias perspectivas que permiten a los desarrolladores mostrar mejor el arte y la narrativa del juego. Encontramos que, casi todos los años durante los últimos 10, al menos la mitad de los juegos más vendidos son en tercera persona [1], y de los 50 juegos más vendidos de la historia, 26 tienen una perspectiva en tercera persona [2].

Un elemento importante de los juegos con visualización en tercera persona es el control de la cámara virtual por parte del jugador. En estos juegos, la cámara virtual se desplaza a través de la propia escena del juego, y es a menudo controlada por el propio jugador con funciones tanto de visibilidad como de apuntado, como en juegos de acción.

El jugador, además, según su plataforma, puede emplear distintos dispositivos de control, y distintos juegos pueden requerir distintos esquemas de control. Es importante, entonces, que los sistemas de cámaras virtuales y los esquemas de control estén diseñados de manera que permitan al jugador tener una experiencia de juego satisfactoria independientemente de los dispositivos de control.

Este trabajo entonces se centra en el análisis de las necesidades técnicas de la visualización e interacción en tercera persona en videojuegos y experiencias interactivas, y el desarrollo de componentes para Unity que ayuden a implementar tanto sistemas de cámaras virtuales como esquemas de control que aborden las necesidades estudiadas.

1.1 Objetivos

El principal objetivo de este trabajo es el desarrollo de componentes para Unity, en el lenguaje de programación C#, que permitan el fácil desarrollo de sistemas de cámaras y esquemas de control para videojuegos en tercera persona.

Para ello, se plantean los siguientes objetivos específicos:

- Estudiar los fundamentos sobre la visualización en 3D y los sistemas de cámaras virtuales, así como los esquemas de control de los juegos en tercera persona y su estrecha relación con las cámaras.

- Estudiar las dificultades técnicas de distintos juegos en tercera persona, buscando sobre todo aquellos que presenten sistemas de cámaras o esquemas de control interesantes desde un punto de vista técnico.
- Desarrollar una variedad de componentes para Unity, altamente integrados con el editor, que permitan la creación y configuración de sistemas de cámaras complejos, y su integración con distintos esquemas de control solucionando las dificultades estudiadas.
- Probar los componentes desarrollados en una escena de Unity con una escena de demostración que permita alternar entre distintos planos de visualización, permitiendo evaluar la eficacia de los componentes desarrollados.

Capítulo 2

Estudio teórico

Para poder desarrollar un sistema de cámaras y esquemas de control para visualizaciones en tercera persona, primero estudiaremos sus fundamentos, las necesidades de la industria y las implementaciones presentes actualmente en esta.

Es importante señalar que, debido a la amplitud del tema, este trabajo no explicará en detalle los fundamentos proceso de *renderizado*, el cual podría merecer su propio trabajo, ni se profundizará en las representaciones matemáticas utilizadas para representar distintas transformaciones espaciales, como los *cuaterniones* y las *coordenadas homogéneas*, quedando dentro del ámbito del trabajo solo su uso para la representación de cámaras y objetos en el espacio 3D.

Si se desea estudiar en más profundidad estas representaciones y transformaciones, se recomienda leer el tema 3 del libro “Applied Geometry for Computer Graphics and CAD” de Duncan Marsh [3], especialmente las secciones 3.1, 3.2, 3.3.2 y 3.5. Puesto que las coordenadas homogéneas pueden ser difíciles de entender sin una intuición visual, se recomienda también el artículo “Explaining Homogeneous Coordinates & Projective Geometry” del blog del programador de gráficos 3D Tom Dalling [4].

2.1 Nomenclatura de motores de videojuegos

Vamos a explicar primero algunos términos que se encuentran habitualmente al empezar a trabajar con herramientas de desarrollo de videojuegos, y que son relevantes para la visualización 3D y el desarrollo de sistemas de cámaras en videojuegos.

Un **motor de videojuegos** se refiere a un conjunto de aplicaciones y librerías de software que permiten el desarrollo de videojuegos u otras aplicaciones 3D, como simulaciones o visualizaciones.

Un **objeto** es cualquier elemento que tenga una posición en un espacio vectorial \mathbb{R}^3 , una orientación en el mismo espacio, generalmente representado como un *cuaternion unitario*, y una identidad única que lo distingue del resto de objetos siendo considerados.

Los cuaterniones son estructuras algo complicadas de explicar de forma breve: son una extensión de los números complejos con tres unidades imaginarias en lugar de una. De cara a este trabajo, solo necesitamos entender que los cuaterniones unitarios representan las rotaciones de un vector en el espacio \mathbb{R}^3 , y que son preferidos a otras representaciones porque el producto (no commutativo) de dos cuaterniones modela la composición ordenada de las rotaciones que representan. Como se ha indicado previamente, puede leerse el tema 3 de “Applied Geometry for Computer Graphics and CAD” [3] para más detalles.

Los objetos pueden tener un **objeto padre**, en cuyo caso su posición y orientación están definidas de forma relativa a las del objeto padre y no respecto al origen del espacio vectorial. Aun así, normalmente nos referiremos a las coordenadas absolutas de un objeto, no las relativas al padre, salvo cuando se indique lo contrario.

El término usado para los objetos varía mucho de un motor de videojuegos a otro: Unity los llama *game objects*, Unreal Engine los llama *actors*, Godot los llama nodos... En este trabajo usamos la nomenclatura de Unity debido a que lo empleamos para el desarrollo.

Una **escena** es el conjunto de objetos al mismo tiempo en un mismo espacio. Esta suele representarse como una estructura jerárquica que muestra las relaciones padre-hijo entre los objetos de la escena. Los objetos que no tienen un padre se representan en la raíz.

Un **componente** es cualquier conjunto de atributos que se asocian a un objeto. Generalmente, estos son instancias de una clase, en el sentido de la programación orientada a objetos, cuyos atributos públicos pueden ser editados de forma visual desde el editor del motor de videojuegos. Los componentes suelen además implementar una serie de métodos *callback* que el motor invoca para que los componentes reaccionen al ciclo de vida del objeto u otros eventos; en estos métodos los componentes pueden obtener referencias otros componentes para acceder a sus miembros públicos.

Un **componente de cámara** se refiere a un componente que represente algún modelo matemático de una cámara. Cuando un objeto tiene un componente de cámara asociado, se le llama **objeto cámara**, en el contexto de una escena, simplemente **cámara**.

Un **gizmo** es una representación visual de las propiedades geométricas, pero no visuales, de un objeto. Por ejemplo, un *gizmo* de cámara puede mostrar un icono de cámara en la posición y un tronco de pirámide hacia la orientación de la cámara, representando su campo de visión.

Un **frame** se refiere a cada paso de la simulación de la escena. Este paso tiene, por lo general, una duración variable, por lo que es necesaria una variable que represente el tiempo transcurrido desde el *frame* anterior para poder integrar cualquier fórmula diferencial.

2.2 Estudio de la literatura

Se han estudiado distintas fuentes de información de fuentes de la industria sobre sistemas de cámaras en videojuegos y su interacción con el esquema de control.

2.2.1 Real Time Cameras: a Guide for Game Designers and Developers

“Real Time Cameras: a Guide for Game Designers and Developers” [5], escrito por Mark Haigh-Hutchinson, conocido sobre todo por su trabajo en los sistemas de cámaras de la serie de juegos Metroid Prime [6].

Este libro nos da los fundamentos básicos de la representación por ordenador de los sistemas de cámaras en motores de videojuegos, del diseño cámaras en tercera persona, de los conceptos básicos para modelarlos matemáticamente, y de su implementación.

Entendemos que el objetivo de una cámara 3D es modelar el concepto de una cámara real, de manera que el software pueda luego *renderizar* la geometría de una escena dando una imagen equivalente. Esto requiere elegir un modelo matemático con el que calcular la óptica para las cámaras.

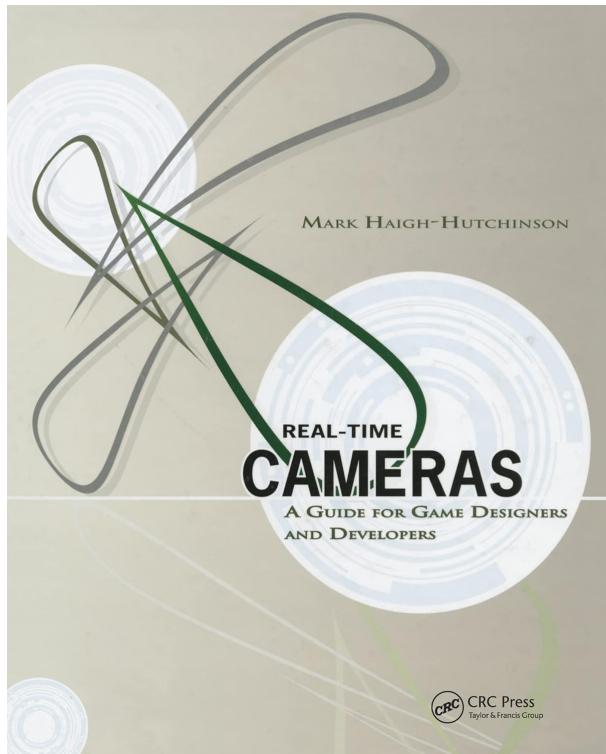


Figura 2.1: Portada de “Real Time Cameras”.

Algunos modelos de cámara, sobre todo en producción de películas, intentan recrear la difracción de lentes reales u otros efectos ópticos, pero *renderizar* estos en tiempo real está fuera del alcance del hardware actual. Por tanto, para visualizaciones como videojuegos usaremos modelos matemáticos simplificados, que nos permitan calcular la perspectiva de la cámara de forma eficiente.

Para proyectar la geometría de la escena sobre una cámara, los sistemas de renderizado usan una representación basada en *coordenadas homogéneas*. Explicar estas y las matemáticas del espacio proyectivo queda fuera del ámbito de este trabajo, pero basta explicar que las coordenadas homogéneas nos permiten representar la proyección en perspectiva de un punto en el espacio 3D a un punto en la pantalla como una transformación lineal sobre este espacio de coordenadas.

Más concretamente, la transformación de estas cámaras se representa como el producto de dos matrices en $M_4(\mathbb{R})$:

- La **matriz de vista** V , que traslada la geometría de la escena desde su posición a la posición relativa a la cámara.
- La **matriz de proyección** o matriz de perspectiva P , que proyecta con perspectiva la geometría de la escena.

Si se visualiza esta proyección al revés, sobre el espacio euclídeo de la escena, se obtiene un tronco de pirámide con el vértice en la cámara y la base al fondo de la escena. Este tronco de pirámide se llama **tronco de visión** (*view frustum* en inglés).

Por tanto, los parámetros de una cámara 3D son los que permitan definir estas dos matrices.

La matriz de vista V se puede derivar fácilmente de la posición y orientación del objeto de cámara, por lo que un componente de cámara solo necesita representar los parámetros restantes para modelar P .

Para calcular P necesitamos definir estos otros parámetros:

- Los ángulos de **campo de visión** (*field of view*) en inglés, de la forma $f_{h,v} \in \mathbb{R} : 0 < f_{h,v} < \pi/2$, que determinan los ángulos de la base del tronco de visión. Puesto que la relación de aspecto está condicionada a las dimensiones de la ventana, no a la propia cámara, podemos definir uno de los ángulos como una función del otro y de la relación de aspecto de la ventana.

Entonces solo necesitamos especificar uno de los dos ángulos para modelar el campo de visión. Puesto que se suele preferir que las pantallas más alargadas muestren más contenido en vez de menos, se suele fijar el ángulo sobre la dimensión más corta de la ventana, para que el ángulo en la dimensión más larga sea variable según el dispositivo.

En PC y consola, los juegos suelen presentarse en orientación apaisada, por lo que los videojuegos suelen preferir fijar el ángulo de campo de visión vertical, f_v , como haremos nosotros en este trabajo y como hace por convención Unity.

- Los planos de corte del tronco de visión, llamados **plano cercano** y **plano lejano** (*near plane* y *far plane* en inglés). Estos se representan como dos distancias $d_n, d_f \in \mathbb{R} : 0 < d_n < d_f$ desde el vértice del tronco de visión.

El uso de estos planos de corte implica que la geometría que interseque estos planos se verá cortada. Esto incita a usar valores muy pequeños de d_n y valores muy grandes para d_f , pero debe tenerse cuidado, porque extender demasiado la profundidad del tronco de visión puede producir errores debidos al uso de valores en coma flotante en la implementación de los renderer.

Por tanto, componente de cámara es aquel objeto de la escena que contenga, al menos, los valores (f_v, d_n, d_f) ; esto es, que contenga un ángulo de cámara vertical y dos distancias de planos de corte.

Puesto que los motores de videojuegos tienen su propio componente cámara que, contra nuestras necesidades, acopla estos valores con muchos otros parámetros de *renderizado*, es común llamar componente de **cámara virtual** a un componente no nativo del motor que también represente estos parámetros.

Podemos entonces definir un sistema de cámaras como una colección de cámaras virtuales en una escena, cada una con sus propios parámetros de cámara, y uno o más componentes controladores que modifiquen estos parámetros, aplicando restricciones y comportamientos específicos como transiciones entre cámaras o seguimiento de personajes.

Para este trabajo, nos interesan entonces específicamente los sistemas de cámaras virtuales que permitan definir cámaras en tercera persona, y los componentes controladores que permitan definir esquemas de control para estas cámaras.

Una cámara en tercera persona es aquella que muestra al personaje desde una posición apartada del personaje, generalmente detrás y ligeramente por encima. Estas cámaras permiten al jugador ver la relación espacial del personaje con el entorno, y son comunes en videojuegos donde la navegación es importante.

Hay distintas restricciones de cámara que se pueden usar para visualizar un personaje en tercera persona. La más sencilla es la **cámara estacionaria o cámara fija**, que simplemente apunta al personaje desde una posición no necesariamente estática, pero independiente del personaje. Estas suelen tener transiciones suaves o cortes de cámara para alternar entre ellas.

Pero las verdaderas dificultades técnicas de implementar una cámara en tercera persona derivan de la necesidad del jugador de controlar la cámara, y de la necesidad de que la cámara siga al personaje dentro de la geometría de la escena.

Esta interacción dinámica entre el personaje, la cámara y la escena se modela como una **cámara orbital**, que sigue al personaje a unas distancias predeterminadas, y permite al jugador cambiar los ángulos de cámara alrededor del personaje.

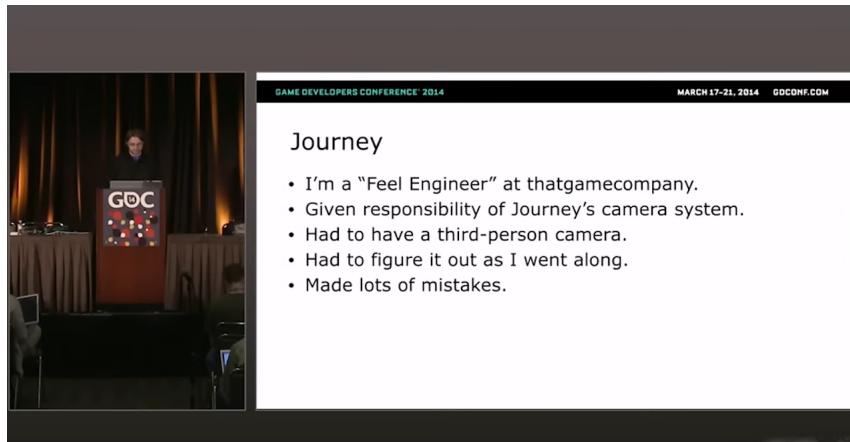


Figura 2.2: Captura de “50 Game Camera Mistakes”.

Puesto que el personaje puede navegar por la escena, la cámara orbital debe también navegar la escena, evitando colisiones con la geometría de esta. Un sistema de cámaras en tercera persona debe entonces definir mecanismos para evitar estas colisiones: tanto de forma predictiva, evitando que la cámara se acerque a la geometría de la escena innecesariamente, como de forma reactiva, ajustando la posición de la cámara para evitar colisiones cuando estas sean inminentes.

2.2.2 50 Game Camera Mistakes

“50 Game Camera Mistakes” [7] es una charla de John Nesky grabada durante la *Game Developers Conference* 2014 sobre las lecciones aprendidas durante como ingeniero tras los sistemas de cámaras virtuales en los juegos de Thatgamecompany, sobre todo del videojuego de exploración en tercera persona *Journey*.

La charla ofrece una lista de funcionalidades a tener en cuenta, descritas respecto a diferentes problemas a evitar, tanto desde el punto de vista del diseño de juegos y niveles como del desarrollo técnico de sistemas de cámaras. Para este trabajo vamos a destilar las conclusiones que se pueden sacar sobre funcionalidades y características técnicas.

Nótese que esta charla comienza recomendando el libro de Mark Haigh-Hutchinson.

Una de sus primeras advertencias es en contra del uso de cuaterniones o coordenadas globales para la persistencia del estado de la cámara orbital: Nesky argumenta que estas representaciones tienen más grados de libertad de los que tiene una cámara orbital, lo que puede llevar a implementar accidentalmente rotaciones de cámara que no deberían ser posibles.

Se identifica que el estado variable de una cámara debería ser unos 7 valores escalares: hasta 3 ángulos de Euler, una distancia entre la cámara y el pivote de la órbita, dos desfases del pivote relativos a la orientación de la cámara, y el ángulo de campo de visión.

Otra recomendación de Nesky es implementar un modificador de cámara que haga uso de *whiskers*: filas de *raycasts* cuyo objetivo es detectar cuándo se va a romper la línea de visión entre el personaje (pivote) y la cámara, por ejemplo, al girar una esquina, y ejercer de forma predictiva una fuerza que gire la cámara para despejar la línea de visión preventivamente.

Sin embargo, también advierte que el uso de fuerzas como esta, que empujen la cámara automáticamente, puede interferir los intentos del jugador de controlar la cámara. Sugiere considerar las fuerzas del jugador sobre la cámara como de prioridad superior, y anular las fuerzas menos prioritarias cuando entren en conflicto. Para este propósito, se considera por separado cada grado de libertad de la cámara.

También advierte que esta prioridad del control manual no debe aplicarse cuando la cámara llega a colisionar con la geometría de la escena: Si bien los *whiskers* protegen la línea de visión de colisiones por los lados, una cámara que ya tiene una colisión debe ajustarse siempre a la geometría del escenario, proyectándola hacia esta con *sphere casts* para buscar el punto más alejado que pueda preservar la línea de visión en el mismo ángulo de cámara.

Otra funcionalidad destacada es permitir que cierta geometría específica de la escena no se tenga en cuenta al calcular las roturas de la línea de visión. Esto es útil para evitar comportamientos inestables cuando se cruzan por delante elementos estrechos que solo cortan un instante el ángulo de visión, como pilares.

Una característica importante de los sistemas de cámara de Nesky es que la cámara orbital soporta distancias y ángulos de campo de visión como una función del ángulo vertical de la cámara. Esto es útil para los artistas de cámara, ya que los planos desde arriba requieren más distancia para ver el suelo, mientras que los de planos desde abajo requieren más campo de visión para ver mejor la escena junto al personaje.

Se resalta también de la necesidad de introducir y manejar cortes de cámara entre cámaras orbitales y fijas. Es importante que los cortes de cámara preserven la dirección de movimiento del personaje que tenía antes de cortar, para evitar que el ángulo de movimiento del personaje cambie sin que el jugador tenga tiempo de adaptarse, o peor, creando inestabilidad en las transiciones.

Finalmente, y esto es importante para este trabajo, se advierte que es mala idea intentar implementar un *constraint solver* que resuelva todas las necesidades que los diseñadores puedan tener.

Por un lado, Nesky explica, el proceso de diseñar un juego requiere que los diseñadores puedan trabajar con reglas específicas y predecibles, para lo que es mejor simplemente iterar manualmente en el diseño de un sistema de cámaras. Por otro lado, un sistema general suele resultar impredecible a la hora de depurar, y es preferible que el desarrollador tenga constancia de cómo se resuelven las relaciones entre las distintas restricciones.

2.3 Estado del arte

Estudiamos ahora las herramientas de software existentes en motores de videojuegos comerciales para implementar sistemas de cámaras en videojuegos, centrándonos en el caso de uso de la visualización en tercera persona.

Si bien no hay muchos datos sobre las cuotas de mercado de los motores de videojuegos 3D en la industria en general, podemos emplear la base de datos abierta de *SteamDB* [8] para estimar la cuota de mercado en el mercado de videojuegos de PC. Observamos claramente que el mercado está dominado por dos grandes motores de videojuegos comerciales: Unity (47398 títulos) y Unreal Engine (13627 títulos). Una inspección de los títulos que usan estos motores nos muestra que Unity es preferido por los estudios de videojuegos independientes, pequeña y mediana empresa, mientras que Unreal Engine es preferido por los estudios de videojuegos con equipos y presupuestos más grandes.

Entonces estudiaremos, en concreto, las herramientas ofrecidas por Unity y Unreal Engine para implementar sistemas de cámaras en videojuegos.

2.3.1 Unity

Unity [9], desarrollado por Unity Technologies, es el motor de videojuegos comercial con mayor cuota de mercado, especialmente en la pequeña y mediana empresa. Unity es un motor genérico y flexible que



Figura 2.3: Número de títulos en Steam agrupados por motor de videojuegos. Nótese que solo Unity, Unreal, y en menor medida Godot y XNA son usados para juegos en 3D.



Figura 2.4: Logo de Unity

permite el desarrollo de juegos y simulaciones mediante un editor visual, en el que componentes escritos en C# se pueden adjuntar a objetos virtuales de la escena.

2.3.1.1 Cámaras básicas

En Unity, una cámara debe definirse con el componente Camera [10], que se añade a un objeto en la escena. Cada instancia del componente define los parámetros comunes a una cámara 3D que hemos estudiado, además de parámetros relacionados con el renderizado de la escena. En cada momento, solo una cámara puede ser la cámara principal que se *renderizará* en pantalla, determinada por la API Camera.main; por defecto, esta será la primera cámara con la etiqueta MainCamera.

Esto nos da una forma primitiva de alternar entre distintas cámaras, además de poder manipularla añadiendo nuestros propios componentes, pero no es suficiente por sí mismo para especificar un sistema de cámaras virtuales. Además, *instanciar* distintas cámaras obliga a duplicar también en el editor la configuración de renderizado, lo que debería ser un aspecto separado del componente.

Por tanto, la estrategia habitual para una visualización en tercera persona es definir una única cámara principal con un componente controlador que sitúe el objeto cámara en la posición y orientación correctas según el sistema de cámaras, y usar otros componentes para representar las cámaras virtuales y otros elementos del sistema.

Unity intenta ofrecer un componente implementar la funcionalidad de mirar automáticamente a un objeto objetivo, *Look At Constraint*. Si bien es utilizable, su configuración es más compleja de lo necesario, y no tiene opción para copiar el vector “arriba” de un objetivo, lo que permitiría cámaras para personajes inclinados (por ejemplo, andando por una pared o techo).

2.3.1.2 Cinemachine

Esto es lo que hace, precisamente, Cinemachine, un paquete de componentes para Unity con el que implementar sistemas de cámaras virtuales [11]. Inicialmente desarrollado para el videojuego *Homeworld: Deserts of Kharak*, Cinemachine fue adquirido por Unity y distribuido con su motor como paquete instalable desde Unity 2017.2.

Notablemente, Unity recibió un premio Emmy de tecnología e ingeniería por su trabajo en Cinemachine, además de otras herramientas, para la producción de los cortos *Baymax Dreams* en colaboración con Disney.

La principal característica de Cinemachine es que sus cámaras virtuales imitan los movimientos de los equipos de cámara reales, y el sistema de cámaras imita las decisiones de un operador de cámara. Si bien esta función es más práctica para la elaboración de escenas cinematográficas, muchos de sus componentes son también aptos para implementar cámaras virtuales para visualizaciones en tercera persona.

El componente más importante de Cinemachine es *Camera Brain*, que se añade al objeto cámara principal para que esta sea controlada por el sistema de cámaras de Cinemachine. Cinemachine solo soporta un sistema de cámaras por escena, seleccionando la cámara activada más recientemente o con mayor prioridad y modificando la cámara principal y sus parámetros para imitar a la seleccionada.

El componente básico para crear escenas en Cinemachine es *Virtual Camera*. Este se adjunta a un objeto desde el editor para marcarlo como una cámara virtual, y permite definir los parámetros comunes a las cámaras virtuales que ya hemos estudiado. Además, este componente permite definir dos objetos para calcular la posición y orientación de la cámara, denominados *Follow* y *Look At*; generalmente estos dos serán el personaje del jugador y su punto de mira, respectivamente.

Sin embargo, aunque definamos estos dos objetos, el objeto cámara virtual seguirá siendo estático. Para que el sistema de cámaras mueva automáticamente la cámara, debemos configurar en el componente distintos algoritmos que modifiquen su comportamiento.

Por un lado, encontramos algoritmos de posición; de interés son:

- El algoritmo *Transposer* recalcula la posición de la cámara virtual sumando un desfase al vector posición de *Follow* y opcionalmente aplicando un efecto amortiguador a los movimientos de la cámara. Este efecto amortiguador lo encontramos en casi todos los algoritmos de posición.
- El algoritmo *Orbital Transposer* implementa parte de la clásica cámara orbital. Similar a *Transposer*, este suma un desfase al vector posición de *Follow*, pero además aplica una rotación a este desfase alrededor de *Follow*, controlado directamente por eje del método de entrada del jugador.
- El algoritmo *Tracked Dolly* implementa parte de la clásica cámara sobre raíles. Un parámetro adicional *Path* permite seleccionar la ruta de la cámara, un objeto que tenga el componente *Path* o *Smooth Path*, el cual define una curva interpolada sobre la que se busca la posición más cercana. La diferencia entre *Path* y *Smooth Path* es que este segundo calcula automáticamente las tangentes de la curva resultante.

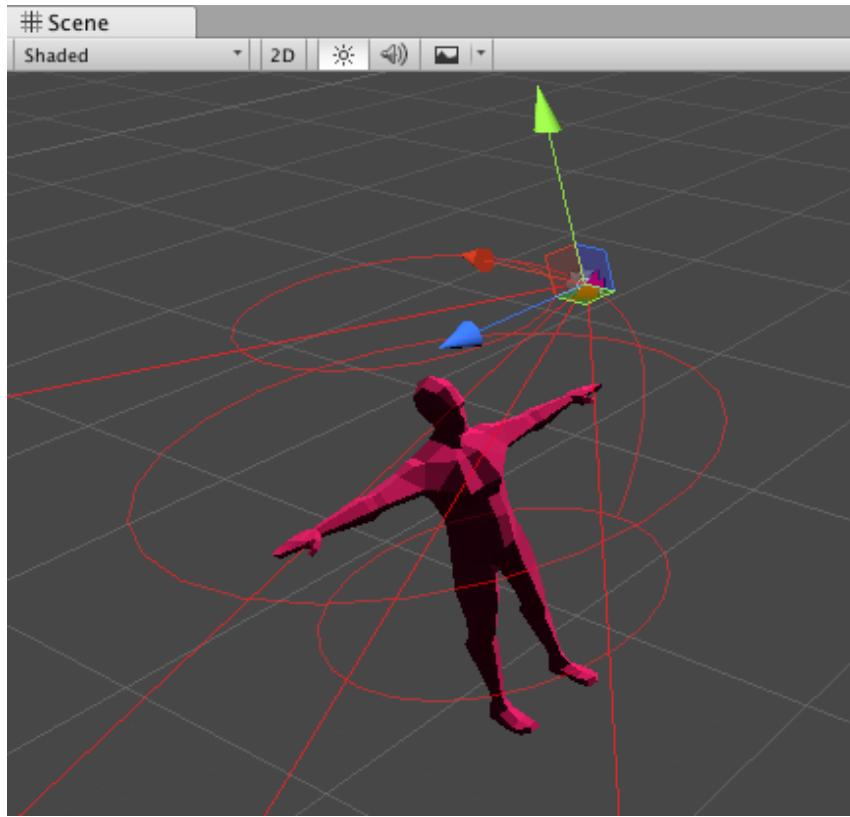


Figura 2.5: *Gizmos de una Free Look Camera*

Por otro lado, tenemos algoritmos de apuntado; de interés son los algoritmos *Composer* y *Group Composer*, que apuntan la cámara a la posición de *Look At*. Estos definen además parámetros de amortiguación de orientación, desfase del objetivo, y regiones muertas para ignorar pequeños movimientos en pantalla. En el caso de *Group Composer* se hace uso de un objeto complementario con un componente *Target Group* que lista múltiples elementos a tener en cuenta simultáneamente.

Sobre estos comportamientos, Cinemachine implementa componentes compuestos más complejos. De especial interés para nosotros es el componente *Free Look Camera*, que define tres cámaras orbitales a diferentes alturas, cada una con su propio conjunto de algoritmos y parámetros, conectados por una curva vertical en la que va montada la verdadera cámara virtual, y a lo largo de la cual la cámara interpola los parámetros de las tres cámaras orbitales. Esto permite configurar visualmente una cámara con distintos desfases y orientaciones a distintas alturas.

Cinemachine tiene también una funcionalidad para que los scripts puedan introducir vibraciones en las cámaras, para simular efectos como explosiones o imitar el movimiento de un operador de cámara corriendo.

Estos impulsos están compuestos de una señal de vibración, efectivamente una curva en 6 dimensiones (3 espaciales, 3 de orientación), y un par de componentes *Source* y *Listener* que propagan esta señal y la aplican en la cámara objetivo.

Si bien los parámetros que definen este impulso en el tiempo modelan propiedades físicas de la vibración (amplitud, dirección, generador de envolvente), su propagación por el medio no implica ninguna simulación, simplemente se especifica un radio y una función de disipación que se evalúa sin tener en cuenta la geometría de la escena.

Es importante mencionar que todos estos componentes muestran una visualización en el editor de las órbitas, rutas y volúmenes de todas las cámaras, y permiten configurar sus parámetros en tiempo real. Además, los scripts del sistema de cámara aplican sus algoritmos durante el modo edición, lo cual no sucede por defecto en los scripts de Unity.

Una limitación de las cámaras virtuales de Cinemachine es que, al imitar el movimiento de un operador de cámara, su algoritmo intenta evitar ciertos movimientos poco naturales como los planos inclinados o en picado, por lo que para implementar ciertas cámaras virtuales es necesario controlar manualmente el cambio de cámara.

De forma más general, se aprecia que el enfoque de estos componentes es permitir a diseñadores crear cámaras complejas sin código, cayendo en la trampa descrita por Nesky de llegar a implementar un *constraint solver* genérico. Esto puede ser práctico para escenas cinematográficas, el punto fuerte de Cinemachine, pero esto hace sus componentes más difíciles de controlar a la hora de iterar en una cámara dinámica para el *gameplay*.

2.3.1.3 Esquemas de control

Cinemachine no ofrece funcionalidad para manejar la interacción entre el esquema de control del juego y su sistema de cámaras, salvo por una primitiva forma de controlar el algoritmo *Orbital Transposer* directamente desde un eje de entrada definido en Unity (p. ej. ratón o stick derecho del mando).

En general, Unity no ofrece ninguna funcionalidad especial para manejar la interacción entre el esquema de control del juego y el sistema de cámaras, como podría ser proyectar la dirección de movimiento del personaje según la cámara y la dirección de entrada del jugador.

Lo que se ofrece, es un sistema para definir distintos botones y ejes abstractos que representen la entrada del jugador, y un sistema para configurar cómo un teclado, ratón o mando se proyecta sobre estas acciones. Esta abstracción de ejes y botones permite ignorar las diferencias entre dispositivos de entrada, dejando de lado el caso de la entrada táctil.

2.3.2 Unreal Engine

Unreal Engine [12], desarrollado por Epic Games, es el motor de videojuegos comercial con la segunda mayor cuota de mercado, liderando sobre todo en las grandes empresas.

2.3.2.1 Cámaras básicas

Unreal provee dos componentes básicos para implementar cámaras en videojuegos controladas por el jugador: UCameraComponent y USpringArmComponent [13].

UCameraComponent es el componente que define una cámara nativa en Unreal. Al igual que la cámara de Unity, este componente empareja mucho más estado que solo los necesarios para definir una cámara virtual, sino que también incluye parámetros de *renderizado* y postprocesado.

En Unreal los objetos de la escena no se pueden componer libremente, sino que es necesario definir una clase de C++ que herede de AActor y tenga los componentes como campos. Por ello, Unreal provee también una implementación por defecto de objetos cámara, ACameraActor que se puede añadir directamente a la escena.

El otro componente, USpringArmComponent, es un componente que se añade a un objeto cámara para definir una restricción orbital a un objeto, como una cámara. Este componente soporta detección



Figura 2.6: Logo de Unreal Engine

de colisiones con la geometría de la escena, y permite definir un desfase del pivote y una distancia fija a este.

Sin embargo, este componente representa los ángulos con la misma representación que el motor, un cuaternión, lo que le impide tener funcionalidades como impulsos en cada eje o definir la distancia en función de la orientación vertical de la cámara.

2.3.2.2 Esquemas de control

Al igual que en Unity, encontramos que Unreal Engine no ofrece ninguna funcionalidad especial para manejar la interacción entre el esquema de control del juego y su sistema de cámaras.

Y al igual que en Unity, Unreal ofrece un sistema de entrada abstracto, *Input Mapping Contexts*, que permite definir distintos botones y ejes abstractos que representen la entrada del jugador, y un sistema para configurar cómo un teclado, ratón o mando se proyecta sobre estas acciones.

Sin embargo, encontramos que Unreal, al estar diseñado para ser programado mediante *Blueprints*, su lenguaje de programación visual, además de mediante C++, ofrece funciones de más alto nivel para combinar direcciones y orientaciones de forma más sencilla, y para aplicar cambios sobre los objetos con transiciones suaves, lo que simplifica la implementación de estos componentes de entrada, control y cámaras.

2.4 Casos de estudio

Vamos también a estudiar las cámaras de distintos juegos con visualización en tercera persona y su interacción con sus esquemas de control. Para esto, se han analizado juegos con comportamientos de cámara interesantes desde un punto de vista técnico.

2.4.1 Journey

Journey [14] es un juego de aventuras para consolas y, más recientemente, PC. Este juego presenta una cámara orbital elevada, con distintos desfases y ángulos de cámara a distintas alturas, y gran variedad de comportamientos de cámara dinámicos.



Figura 2.7: Captura del videojuego Journey. Se muestra una escena con una cámara orbital y un desfase horizontal extremo para maximizar las vistas en un momento de poco movimiento.

En Journey, cuando la línea de visión de la cámara orbital está a punto de romperse por la geometría del escenario, la cámara da un empujón a un lado para evitar la colisión. Esta característica distingue entre la geometría que interfiere con línea de visión, como paredes, y la que no, como pilares y dunas.

En ocasiones, el jugador en Journey atraviesa barreras opacas como cascadas de arena. En estos casos, se produce un corte de cámara, generalmente a una cámara fija con muy distinta orientación, pero la dirección de movimiento del personaje se mantiene fija mientras no cambie, permitiendo al jugador mantener el control del personaje tras el corte de cámara.

Este juego, además de usar el stick derecho para controlar la cámara, como es costumbre, también permitía hacer uso del giroscopio de los mandos “Sixaxis” de PlayStation 3 para girar la cámara a los lados, una característica heredada del juego anterior de la misma compañía, Flower, el cual se controla casi enteramente con el giroscopio.

2.4.2 Nier: Automata

Nier: Automata [15] es un juego de acción en tercera persona para PC y consolas que hace uso de gran variedad de cámaras y esquemas de control. La mayor parte del tiempo, este juego presenta una cámara orbital algo elevada, controlada por el stick derecho, donde el personaje jugador se mueve en el plano horizontal relativo a la orientación de la cámara.

Sin embargo, muchas secuencias y habilidades hacen que la cámara alterne a cámaras restringidas a planos en picado o laterales, y restringiendo incluso la geometría de control del personaje sin interrumpir la acción ni el control sobre el personaje.

Por ejemplo, el jugador puede estar luchando en una gran sala, y el juego cambiar desde una cámara orbital hasta una cámara en picado, y un plano de control fijo alineados con el norte, para mostrar más enemigos en pantalla. O entrar en un pasillo y cambiar a una cámara lateral, restringiendo el plano de control horizontal a una sola línea, eliminando una dimensión al control del personaje.



Figura 2.8: Captura del videojuego Nier: Automata. Se muestra una escena de acción con una cámara orbital centrada totalmente en el personaje jugador.



Figura 2.9: Captura de Lost Planet 2. Se muestra una cámara orbital con un desfase horizontal fijo, y la cámara apuntando el propio pivote de la cámara, en lugar del personaje, para poder apuntar al fondo con la mirilla.

2.4.3 Lost Planet 2

Lost Planet 2 [16] es un juego de disparos en tercera persona para PC y consolas. Este juego presenta una cámara orbital centrada en el jugador, con desfase hacia el lateral y algo elevada respecto al personaje jugador, y con una cruceta en el centro de la pantalla para apuntar con el arma.

El jugador se mueve sobre el plano horizontal relativo a la orientación de la cámara. Pero, respecto a la cámara, este juego tiene, por defecto, esquemas de control diferentes según uses ratón o mando.

En el caso del ratón, la cámara se controla como en otros títulos de su género: La cruceta se mantiene fija en la pantalla, y mover el ratón mueve la rotación de la cámara.

Pero con mando, por defecto, en Lost Planet el stick derecho no mueve la orientación de la cámara orbital, sino que mueve la cruceta: La pantalla tiene un área por la cual la cruceta puede moverse sin rotar la propia cámara. Es solo cuando la cruceta llega al límite de esta área, y el jugador sigue empujando el stick, que esta “arrastra” la orientación de la cámara hasta que suelta, dejando la cruceta en la misma posición relativa a la pantalla.

Esto permite que la cruceta tenga menor sensibilidad que la rotación de la cámara, permitiendo apuntado fino en una región de la pantalla. Inusualmente, y quizás para compensar el tramo de menor sensibilidad, el juego reserva los botones LB y RB (sobre los gatillos) para girar la cámara 90 grados rápidamente.

Además, el juego tiene una función de *sprint*, durante la cual el desfase de cámara se vuelve más bajo y la órbita pasa a tener un radio más limitado.

2.5 Otros métodos de entrada

Para que un jugador pueda controlar su personaje y la cámara de un videojuego, es necesario que el videojuego pueda recibir información de sus dispositivos de entrada. Las distintas configuraciones de dispositivos y los distintos sistemas de cámaras dan lugar a distintos esquemas de control.

Normalmente, los videojuegos en tercera persona permiten al jugador controlar una cámara orbital mediante un *stick* analógico, en el caso de usar mando, o con el ratón, en el caso de usar teclado y ratón. En ambos casos, el jugador puede controlar la dirección de la cámara mediante un vector de dirección que se obtiene del dispositivo de entrada.

Si embargo, algunos juegos, o algunos jugadores, pueden emplear métodos de entrada distintos para controlar la cámara. Estudiamos algunos ejemplos:

- Como se ha visto en Journey, el giroscopio de un mando se puede usar para controlar la cámara. Este método de entrada también se emplea en la franquicia de juegos Splatoon [17], de Nintendo, en los que el giroscopio se usa para apuntar con el arma.
- En el lado académico, tenemos “A Fitts’ Law Evaluation of Video Game Controllers: Thumbstick, Touchpad and Gyrosensor” [18], que también estudia los diferentes métodos de entrada disponibles en el mando *Steam Controller* de Valve, concluyendo que los jugadores son más precisos usando *touchpad* y, en segundo lugar, giroscopio, que con el *stick* analógico.
- La mirada se puede emplear para controlar una experiencia en tercera persona. En “Gaze as a Navigation and Control Mechanism in Third-Person Shooter Video Games” [19], se estudia el uso de un *eye tracker* para controlar un videojuego de disparos en tercera persona con una cámara con perspectiva en picado, tanto para controlar al personaje como para apuntar, concluyendo que la mirada es más precisa para apuntar que un stick de mando habitual.
- En “Exploratory Design of a Hands-free Video Game Controller for a Quadriplegic Individual” [20], se estudia el uso de reconocimiento de gestos faciales para controlar videojuegos. El jugador puede mover al personaje mediante la proyección de determinados gestos estudiados a entradas de teclado, permitiendo a un jugador tetrapléjico jugar a juegos de diferentes géneros, en especial un juego en tercera persona en el que navegar por una zona natural.

2.6 Comparación y lecciones

Aprendemos que no existe una única solución que satisfaga las necesidades de todos los juegos, así que nuestros componentes deben ser piezas básicas con las que los diseñadores puedan hacer prototipos sobre los que iterar para crear sistemas de cámara más complejos.

Aprendemos también que diferentes juegos pueden emplear al mismo tiempo diferentes tipos de cámaras en función de la situación, mediante transiciones suaves o cortes de cámara. Esto no solo tiene una finalidad artística, sino que puede ser necesario para facilitar la interacción entre el jugador y el juego.

Esta postura es respaldada por estudios académicos como “How Camera Placement Affects Gameplay in Video Games” [21], en el que se crea un juego con diferentes retos en los que los jugadores pueden elegir la cámara que prefieran, y se concluye que, aunque todos los retos son compatibles con todas las cámaras, siempre hay una o dos cámaras más favorables para cada reto, siendo la tercera persona la más práctica en general.

Por tanto, nuestros sistemas de cámaras deben permitir alternar entre distintas cámaras y lidiar con las transiciones o cortes de cámara necesarios, así como las dificultades de interacción que puedan surgir al alternar entre cámaras.

Además, aprendemos que todos los motores de videojuegos comerciales abstraen la entrada del jugador de forma similar, mediante un concepto de ejes y botones. Y el estudio de otros dispositivos de entrada diferentes no parece indicar que esta abstracción sea incompatible con dispositivos de entrada más experimentales.

Por tanto, consideraremos que esta abstracción de ejes y botones es suficiente para definir un esquema de control de cámara en tercera persona, y que la dificultad de la interacción entre el esquema de control y el sistema de cámaras no reside en el método de entrada concreto, sino en la interacción entre el jugador y dicha abstracción.

Observamos entonces varias funcionalidades comunes para la implementación de un sistema de cámaras en tercera persona:

- Soportar múltiples cámaras virtuales entre las que poder alternar.
- Una restricción de cámara que apunte directamente a un objetivo.
- Un modelo de cámara virtual *orbital* con desfase en el pivote en relación con la orientación de cámara, que pueda apuntar la cámara o al propio pivote, para usar una mirilla; o a un objetivo, normalmente el mismo personaje que se orbita.
- Un modelo de cámaras con cortes de cámara y transiciones suaves.
- Un mecanismo para aplicar fuerzas o impulsos sobre la cámara orbital.
- Mecanismos que mantengan la línea de visión de las cámaras, permitiendo excepciones especificadas por los diseñadores.
- Un mecanismo para proyectar la dirección del método de entrada del jugador a una dirección en la escena, en función de la orientación de la cámara.
- Un mecanismo para bloquear la dirección del personaje cuando se producen cortes de cámara, para evitar entradas inestables.
- Un mecanismo para controlar el apuntado de la cámara y al mismo tiempo una retícula de apuntado para juegos de disparos.
- Un mecanismo que modifique la orientación de la cámara para atraer la atención del jugador a un punto de interés.

Tabla 2.1: Funcionalidades de cámaras en tercera persona

Alternar entre múltiples cámaras virtuales
Restricción de cámara de apuntado a un objetivo
Cámara orbital con desfase en el pivote
Cortes de cámara
Transiciones suaves de cámara
Impulsos sobre la cámara orbital
Preservación de la línea de visión
Proyección de la dirección del método de entrada del jugador
Bloqueo de dirección en corte de cámara
Apuntado y posición de retícula
Punto de interés

Capítulo 3

Diseño y desarrollo de la solución

Con todo lo aprendido, se ha procedido a diseñar y desarrollar una serie de componentes para algún motor de videojuegos que implementen un conjunto razonable de funcionalidades para el prototipado de cámaras en tercera persona. Estos componentes deben ser ajustables y acoplables: los diseñadores deben poder componer comportamientos complejos con nuestros componentes y con scripts compatibles que usen nuestras APIs para extender sus capacidades.

3.1 Requisitos y consideraciones

Antes de poder desarrollar ningún componente, primero debemos decidir para qué motor de videojuegos se van a implementar. He decidido emplear el motor Unity por dos motivos principales:

- Primero, que Unity es el más popular de los motores en 3D [8], sobre todo en el ámbito de los estudios independientes y de pequeño tamaño, que se pueden beneficiar más de un sistema de cámaras como el que se propone en este trabajo.
- Y segundo, que tengo experiencia previa desarrollando varios tipos de aplicaciones con el lenguaje de programación C# y, en menor medida, con Unity. Unreal, en cambio, requiere escribir C++ poco ortodoxo, aumentado con un preprocesador [22] y una jerarquía de clases con recolección de basura [23]; o bien aprender *Blueprint*, un lenguaje de programación visual [24].

Unreal entonces habría supuesto una carga adicional de trabajo que me distraería del objetivo principal de este TFG, que es el diseño y desarrollo de componentes de cámaras en tercera persona, y las habilidades que obtendría no serían tan transferibles a otros campos de la industria como el desarrollo en C#.

Con esto, podemos fijar que nuestros componentes se desarrollarán en C# para Unity, y deberán integrarse con sus sistemas de recursos y su editor visual de escenas para facilitar su uso a los diseñadores.

Si bien nuestros componentes no dependen directamente del método de entrada del jugador, ya que resuelven problemas de interactividad relacionados con el sistema de cámaras, se tiene presente que los componentes deben ser compatibles y aplicables a cualquier método de entrada, ya sea teclado, ratón, mando o cualquier otro dispositivo capaz de modelar entradas de eje, y han sido probados tanto en configuraciones de teclado y ratón como en configuraciones de mando.

Puesto que hay una gran variedad de posibles combinaciones de comportamientos de cámara, y para acotar el ámbito del trabajo, se ha decidido implementar los componentes altamente acoplables que satisfacen la siguiente lista de funcionalidades de cámaras en tercera persona en Unity:

- Cámaras virtuales genéricas
- “Virtualización” de la cámara principal nativa de Unity: un componente que hace de conector entre Unity y nuestras cámaras virtuales. Este lanza eventos de actualización y corte de cámara, cuyos *listener* son configurables desde el editor visual, y que mantiene una colección de las cámaras virtuales activas para que se puedan activar y desactivar por código desde distintos componentes sin coordinación.
- Restricciones de cámara orbital, con desfase de pivote, *sphere casts* para corregir colisiones y API para aplicar impulsos desde otros componentes.
- Restricciones de objetivo de mira.
- *Whiskers* que empujen la cámara para evitar predictivamente romper la línea de visión.
- Excepciones a la rotura de línea de visión mediante integrado con la característica de capas de Unity.
- Conjuntos de cámaras fijas con selección de mejor toma y transiciones suaves entre ellas.
- “Brújula” de movimiento que proyecte la entrada de movimiento del personaje al plano de control de este.
- Bloqueo de dirección movimiento en cortes de cámara, para prevenir movimiento inestable en los cortes de cámara.
- Mecanismo para configurar retículas con áreas de movimiento libre en la pantalla.
- Edición y previsualización en tiempo real de todas las cámaras, con *gizmos* que representen el estado de los componentes.

Comparamos las funcionalidades de cámara en tercera persona implementadas con las estudiadas:

Tabla 3.1: Funcionalidades de cámaras en tercera persona

Funcionalidad	Implementada
Alternar entre múltiples cámaras virtuales	Sí
Restricción de cámara de apuntado a un objetivo	Sí
Cámara orbital con desfase en el pivote	Sí
Cortes de cámara	Sí
Transiciones suaves de cámara	Sí
Impulsos sobre la cámara orbital	Sí
Preservación de la línea de visión	Sí
Proyección de la dirección del método de entrada del jugador	Sí
Bloqueo de dirección en corte de cámara	Sí
Apuntado y posición de retícula	Sí
Punto de interés	No

Se considera que estas funcionalidades son suficientes como para componer escenas elaboradas durante el prototipado, y los campos y APIs expuestas suficientes para implementar sistemas más complejos si fuera necesario. Al mismo tiempo, se considera que estas funcionalidades resuelven los principales problemas que aparecen al interactuar el jugador con un sistema de cámaras en tercera persona, y que son adaptables a distintos esquemas de control.

La única funcionalidad que no se ha implementado es la de orientar la cámara hacia un punto de interés. Esta función se podría implementar sobre la API de impulsos de cámara, del mismo modo que se implementa la función de línea de visión, pero no se ha implementado porque suele requerir una integración más profunda con el diseño de niveles del juego, lo que aumentaba la carga de trabajo para desarrollar la demostración de los componentes.

Se ha considerado que los componentes no deberían tener dependencias adicionales más que la API de Unity, y, dado el motivo académico, mucho menos usar sistemas de cámaras ya existentes como base del proyecto.

Para evaluar la utilidad de los componentes implementados, se ha desarrollado una escena de demostración muestra todos estos componentes interaccionando con mínimo código extra.

3.2 Arquitectura e implementación

Nuestros componentes para Unity se implementan en C#, siguiendo la API y arquitectura de este motor, detallados en el manual oficial [25]. Un componente de Unity se implementa como una clase que hereda de MonoBehaviour, y añade campos y métodos que Unity invocará en tiempo de juego o de edición. Unity llama a estos métodos “mensajes”, y los invoca en diferentes momentos del ciclo de vida de un objeto o de la escena.

Muchos de los componentes definen elementos geométricos como campos de visión, órbitas, curvas y rayos. Para poder previsualizarlos en tiempo real mientras se editan los parámetros, estos componentes implementan el mensaje `OnDrawGizmosSelected`, desde el cual se dibujan sobre la escena los *gizmos* que representan dicha geometría.

Puesto que muchos de los componentes implementados modifican también la posición y orientación de las cámaras virtuales de la escena, se han implementado de tal manera que estos atributos también se actualicen en tiempo de edición, anotando estas clases con el atributo de metadatos `ExecuteAlways`. Esto permite que los componentes tengan un ciclo de vida durante la edición de la escena, y no solo en tiempo de juego.

Algunos componentes tienen como dependencia que otros componentes estén presentes en el mismo objeto, para alterar sus campos o invocar alguna API que expongan. Para facilitar el uso de estos componentes, estos declaran todas sus dependencias con el atributo de metadatos `RequireComponent`, que añade automáticamente cualquier dependencia faltante al añadir el componente.

Además, los componentes se integran con los menús del editor visual, haciendo uso del atributo `MenuItem` para declarar ítems en el submenú de crear objetos. Esto permite crear directamente nuevo objetos con estos componentes de cámara ya añadidos.

Un detalle a destacar es que la API de Unity suele representar los ángulos en grados, por lo que se ha optado por seguir esta convención en los componentes implementados.

Se detallan aquí los componentes y otros tipos de datos implementados:

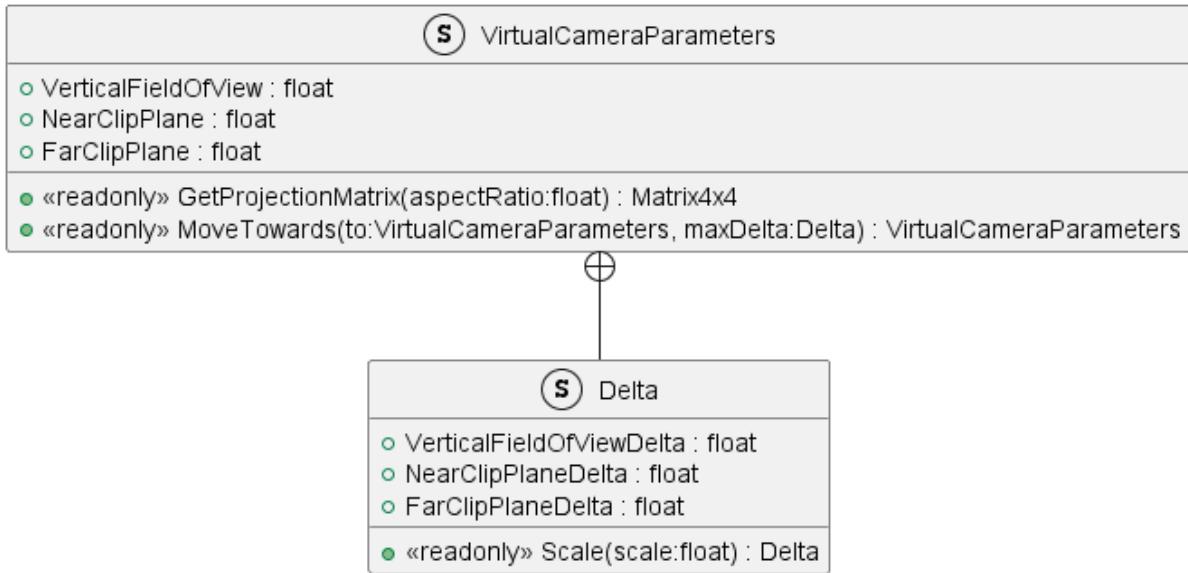


Figura 3.1: Diagrama de clases de `VirtualCameraParameters`.

3.2.1 Virtual Camera Parameters

`Rein.Cameras.VirtualCameraParameters` representa los parámetros comunes de toda cámara virtual. Estos se han extraído a campos en un tipo de datos separado, para poder copiarlos u operar con ellos en conjunto.

Estos campos son:

- `VerticalFieldOfView`, que representa el ángulo de visión vertical de la cámara; recordemos que el ángulo horizontal se ajusta según la relación de aspecto de la ventana. Unity trabaja con ángulos en grados, y este ángulo debe cumplir que $-90 < f_v < 90$ (recordamos que trabajamos en grados), así que este campo tiene un atributo de metadatos `Range (-89, 89)`.
- `NearClipPlane`, que representa el plano de corte cercano del tronco de campo de visión de la cámara.
- `FarClipPlane`, que representa el plano de lejano cercano del mismo campo de visión.

Estos parámetros son suficientes para recrear la matriz de proyección de la perspectiva en función de la relación de aspecto de la ventana, como se evidencia por el método `GetProjectionMatrix` que implementa precisamente esto.

Este tipo de datos implementa además un método `MoveTowards` que implementa una función para interpolar entre dos valores `VirtualCameraParameters` con un factor máximo `maxDelta`. Para esto se implementa para esto además un tipo de datos `VirtualCameraParameters.Delta` que representa los mismos valores que `VirtualCameraParameters` pero sin límites de rango en el editor, ya que representan diferencias o velocidades, y un método `Scale` para poder escalar un delta con valores de velocidad por un escalar de tiempo y devolver un delta con valores de diferencia del *frame*.

Para poder serializar campos de componentes con estos parámetros y editarlos desde el editor visual, se ha marcado este tipo como `struct` y marcado con el atributo de metadatos `Serializable`. En cualquier caso, nos convenía declararlo `struct` porque es un tipo pequeño, y los tipos de datos por valor de .NET (*structs*) no implican alojamiento de memoria, sino copias in-situ.

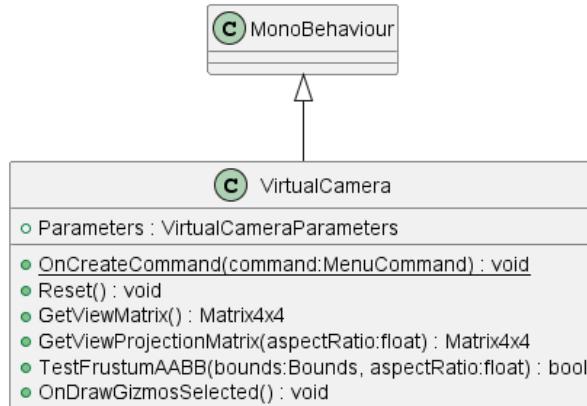


Figura 3.2: Diagrama de clases de VirtualCamera.

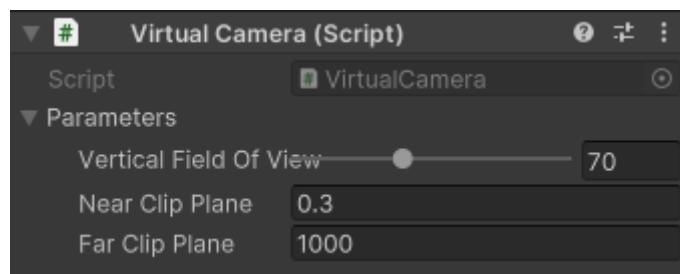


Figura 3.3: Captura del inspector de VirtualCamera.

3.2.2 Virtual Camera

`Rein.Cameras.VirtualCamera` es un componente que marca el objeto al que se añade como una cámara virtual. Este componente es una cámara libre, sin comportamiento inherente: Solo presenta su propia copia de los parámetros de una cámara, y permite previsualizar en el editor visual su campo de visión.

Este componente no procesa mensajes en tiempo de juego. Solo tiene un único campo, `Parameters`, de tipo `CameraParameters`, que puede ser configurado desde el editor o asignado desde otros componentes que especifiquen modelos de cámara más específicos.

Al seleccionar un objeto con `VirtualCamera` en el editor de escenas, este reacciona en tiempo de edición al mensaje `OnDrawGizmosSelected`, dibujando *gizmos* que representan sobre la escena el tronco de visión de la cámara.

`VirtualCamera` tiene todos los parámetros necesarios para calcular las matrices de vista y proyección a partir de la relación de aspecto de la ventana, como se evidencia por los métodos `GetViewMatrix` y `GetViewProjectionMatrix`.

En Unity, estas matrices las calculará el motor durante la fase de renderizado, leyendo los parámetros del objeto marcado como cámara principal, pero también es útil generarlas por separado para, por ejemplo, tests de colisión de un objeto con el tronco de visión de la cámara. Esta funcionalidad es precisamente la implementada en el método `TestFrustumAABB`.

Este componente puede añadirse simplemente a un objeto existente de la escena, o crearse directamente usando el ítem de menú `GameObject/Rein/Virtual Camera` que se ha implementado, el cual añade a la escena un nuevo objeto con componente `VirtualCamera` ya añadido.

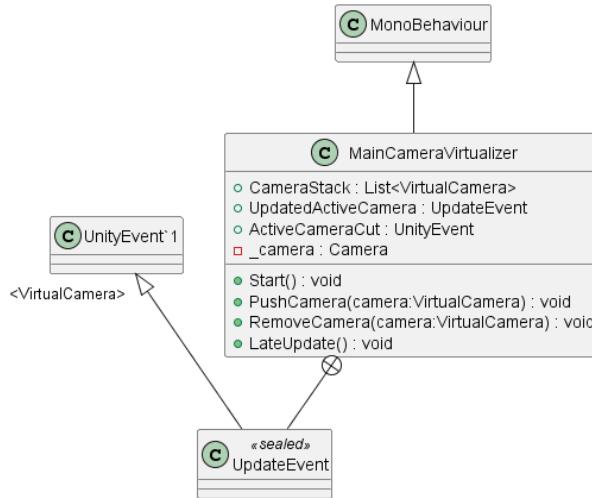


Figura 3.4: Diagrama de clases de `MainCameraVirtualizer`.

3.2.3 Main Camera Virtualizer

El componente `Rein.Cameras.MainCameraVirtualizer` se añade a un objeto con un componente `Camera` nativo de Unity, y copia a esta cámara el estado de nuestra cámara virtual activa todos los *frames*.

Para seleccionar la cámara virtual, `MainCameraVirtualizer` define un campo `CameraStack`, de tipo `System.Collections.Generic.List<VirtualCamera>` y una API con dos métodos, `PushCamera` y `RemoveCamera`. Al configurar, desde el editor o desde la API, una o más cámaras, `MainCameraVirtualizer` usará siempre la última de la colección.

Esta estructura similar a una pila se usa para que distintos pares de métodos *enter/exit* puedan cambiar la cámara actual y volver a la anterior automáticamente y aunque los pares *enter/exit* se resuelvan en distinto orden.

`UpdatedActiveCamera` y `ActiveCameraCut` hacen uso además de dos eventos usando `UnityEvent`, el sistema nativo de eventos de Unity. `UpdatedActiveCamera` se lanza todos los *frames* durante `LateUpdate` con la cámara actual como argumento, mientras que `ActiveCameraCut` se lanza cuando se ha cambiado la cámara virtual activa de este componente, produciendo un corte de cámara.

La posición y orientación del objeto actual serán sobrescritas con las de la `VirtualCamera` actual, y sus `CameraParameters` también serán copiadas al componente `Camera` nativo de Unity en este objeto.

`MainCameraVirtualizer` implementa el mensaje de Unity `LateUpdate`, en el cual se realizan todas estas copias de estado.

3.2.4 Orbit Camera Constraint

`Rein.Cameras.OrbitCameraConstraint` es el mayor componente de este trabajo. Implementa los parámetros para una cámara orbital, incluyendo:

- Selección por separado de los objetivos de posición, de apuntado de la cámara y de inclinación de la cámara.
- Distancia al objetivo y ángulos de campo de visión, variables en función del ángulo vertical de la cámara e integrados con el editor visual de curvas de Unity.

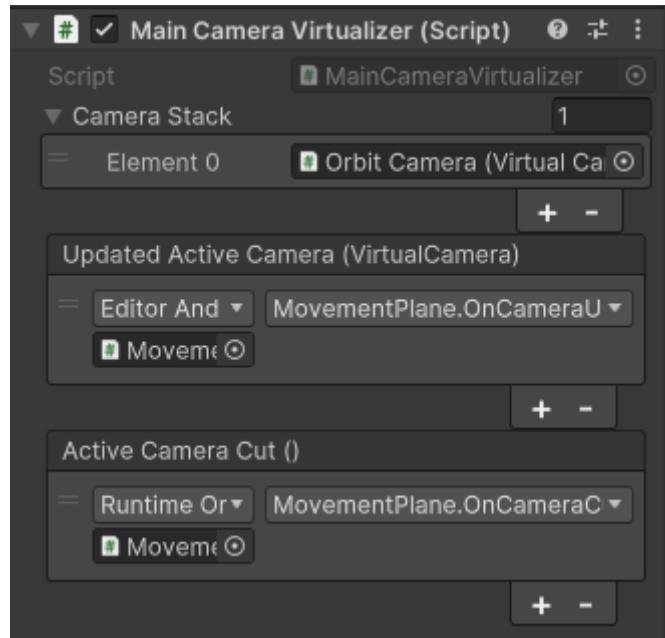


Figura 3.5: Captura del inspector de MainCameraVirtualizer.

- Detección y corrección de colisiones con la geometría del escenario mediante *sphere casts*, haciendo también uso de máscaras de *layers* de Unity para permitir excepciones en la geometría del escenario.
- Desfase del pivote en espacio de cámara.
- Una API de acumulación de impulsos de cámara, por *frame*, con dos niveles de prioridad por cada grado de libertad. Esto permite que los impulsos del jugador tengan preferencia sobre los impulsos generados para evitar colisiones de cámara.
- *Gizmos* dinámicos que permiten previsualizar la órbita actual y la función de distancia mientras se editan.
- Previsualización de todos los comportamientos del componente en tiempo de edición, no solo en tiempo de juego.

Los campos LookAtTarget y OrbitTarget definen los objetivos de mira y de posición de la cámara, respectivamente. OrbitTarget debe ser no-nulo para que la cámara funcione, mientras que LookAtTarget es opcional, usando por defecto la posición del pivote, lo que es también más deseable para implementar sistemas de apuntado en juegos de disparos.

El campo OrbitOffset define el desfase del pivote en espacio de cámara: este desfase se aplica respecto a OrbitTarget relativo al ángulo horizontal de la cámara, para poder colocar la cámara sobre el personaje o a un lado y despejar así la línea de visión.

Los campos Distance y VerticalFieldOfView definen dos curvas como función del ángulo vertical de la cámara: una de distancia y otra de campo de visión. Estas curvas usan la clase AnimationCurve de Unity, que permite definir curvas de manera visual en el editor y guardarlas como *assets* compartidos de Unity en el proyecto.

Los campos HorizontalAngle y VerticalAngle definen el ángulo actual de la cámara respecto a OrbitTarget. VerticalAngle tiene un atributo [Range (-89, 89)] para limitar el ángulo vertical de la cámara en el editor, limitación que también se aplica en el mensaje LateUpdate de Unity.

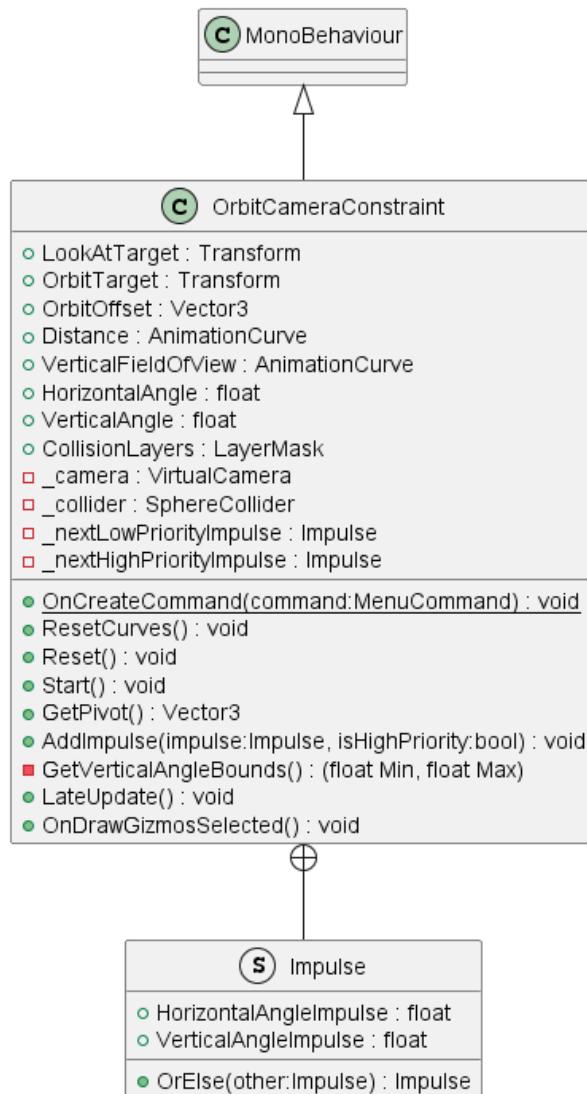


Figura 3.6: Diagrama de clases de OrbitCameraConstraint.

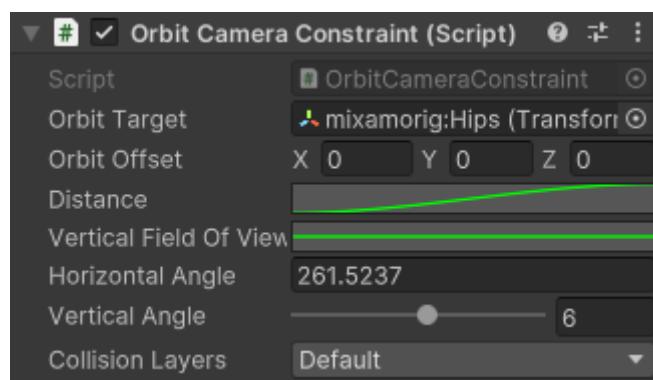


Figura 3.7: Captura del inspector de OrbitCameraConstraint.

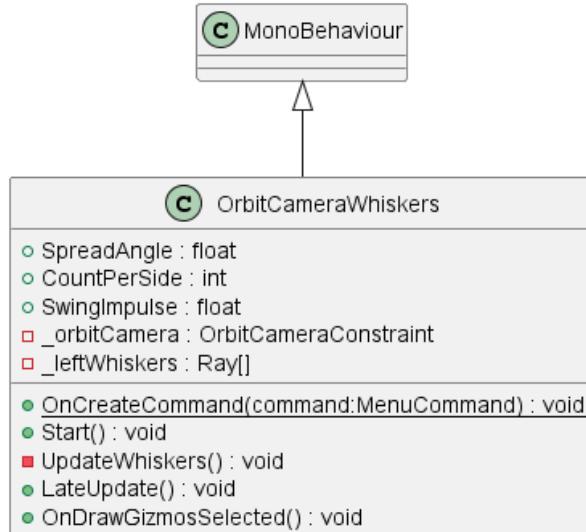


Figura 3.8: Diagrama de clases de `OrbitCameraWhiskers`.

El campo `CollisionLayers` especifica las capas (*layers*) de Unity que deben ser consideradas para las colisiones de cámara, permitiendo excluir partes de la geometría del escenario, como pilares o el propio personaje.

Finalmente, se implementa una API `AddImpulse` que permite que otros componentes apliquen impulsos sobre esta cámara, con dos niveles de prioridad para diferenciar impulsos del jugador e impulsos de otros componentes.

El tipo de datos `OrbitCameraConstraint.Impulse` representa un impulso sobre los parámetros de la cámara. Los campos privados `_nextLowPriorityImpulse` y `_nextHighPriorityImpulse` acumulan los impulsos de baja y alta prioridad, respectivamente.

En el mensaje `LateUpdate` de Unity, se acumulan todos los impulsos, se actualizan los ángulos de cámara, y se calculan el pivote y la distancia de la cámara, y se actualizan la posición y orientación.

Durante este proceso, se realizan *sphere casts* para detectar si la cámara colisiona con el escenario, en cuyo caso se reduce la distancia para mantener la línea de visión en el mismo ángulo de cámara.

Como otros componentes, `OrbitCameraConstraint` implementa el mensaje de Unity `OnDrawGizmosSelected`, que dibuja sobre la escena los *gizmos* con la órbita actual y la curva de distancia para el ángulo actual.

Este componente se añade a cualquier objeto que también tenga nuestra `VirtualCamera` y un `SphereCollider` de Unity; también pueden crearse directamente usando el ítem de menú `GameObject/Rein/Orbit Camera` que se ha implementado, el cual añade a la escena un nuevo objeto con componente `OrbitCamera` y todas sus dependencias.

3.2.5 Orbit Camera Whiskers

El componente `Rein.Cameras.OrbitCameraWhiskers` implementa detección y corrección predictiva de colisiones con la geometría entre una cámara orbital y el pivote, mediante el uso de “whiskers”, filas de *raycasts* lanzados desde el pivote hacia los lados.

`OrbitCameraWhiskers` se añade a un objeto que también tenga `OrbitCameraConstraint`, y hace uso de su API de impulsos de cámara para generar fuerzas que empujen la cámara a los lados al girar



Figura 3.9: Captura del inspector de OrbitCameraWhiskers.

esquinas o pegarse a paredes, para evitar que la visión quede obstruida o encerrada entre el personaje y la pared.

El campo `SpreadAngle` define el ángulo de apertura de los *whiskers*, que se lanzan hacia los laterales de la cámara orbital. Tiene un atributo [Range (0, 180)] para limitar el ángulo en el editor.

El campo `CountPerSide` define la cantidad de *whiskers* que se lanzan a cada lado de la cámara. Este campo se usa para calcular el ángulo entre cada *whisker* y el siguiente.

El campo `SwingImpulse` define el impulso máximo total que aplica cada lado de la cámara. El impulso máximo de cada *whisker* se calcula dividiendo el impulso máximo total entre el número de *whiskers* de cada lado.

Los impulsos de cámara generados por `OrbitCameraWhiskers` son de prioridad baja, por lo que dejan de hacer efecto mientras el jugador controla manualmente la cámara.

En el mensaje `LateUpdate`, `OrbitCameraWhiskers` lanza *raycasts* a cada lado de la cámara hasta la distancia de la propia cámara, y genera impulsos proporcionales a la distancia desde la cámara a la que han hecho contacto con la geometría, por lo que el impulso crece según crece la necesidad de mover la cámara.

Puesto que los impulsos se aplican a cada lado, se puede dar la situación en la que los *whiskers* de ambos lados se anulen entre sí. Esto es deseable, pues suele significar que el jugador ha puesto deliberadamente la cámara en paralelo a la normal de una pared, y no podemos elegir a qué lado mover la cámara.

Al igual que el resto de componentes, `OrbitCameraWhiskers` responde al mensaje `OnDrawGizmosSelected` para dibujar *gizmos* sobre la escena actual, que representan los rayos de los *whiskers* y cambian de color cuando entran en contacto con el escenario, para ayudar a depurar su uso.

Además de añadir a un objeto que ya tenga el componente `OrbitCameraConstraint`, una cámara orbital con *whiskers* puede añadirse directamente a la escena usando el ítem de menú `GameObject/Rein/Orbit Camera (Whiskers)` que se ha implementado, el cual crea un nuevo objeto con `OrbitCameraWhiskers` y todas sus dependencias.

3.2.6 Movement Compass

`Rein.Inputs.MovementCompass` es un componente que proyecta el movimiento del personaje sobre plano de movimiento del personaje, en un ángulo relativo a la dirección de la cámara.

Este componente no se añade al objeto del personaje ni de la cámara, sino que existe como un objeto independiente en la escena. La orientación de este objeto define el plano sobre el que se mueve el personaje, generalmente horizontal.

Este campo tiene dos métodos, `OnCameraUpdate` y `OnCameraCut`, que deben ser conectados con los eventos generados por `MainCameraVirtualizer` para que el componente funcione correctamente.

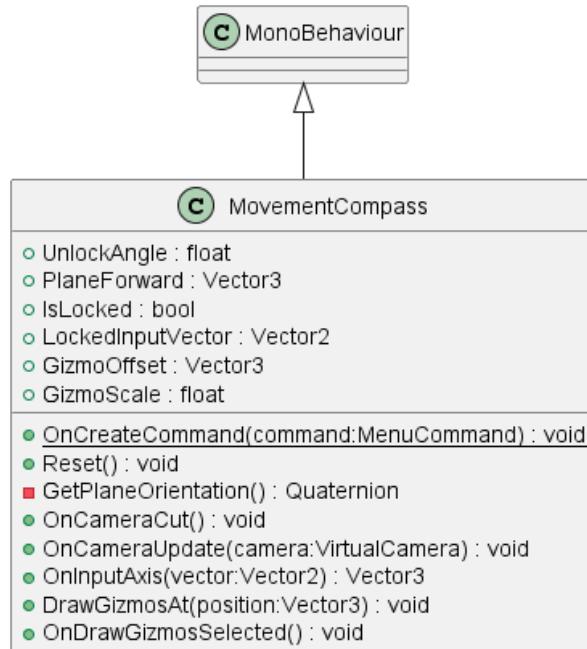


Figura 3.10: Diagrama de clases de MovementCompass.

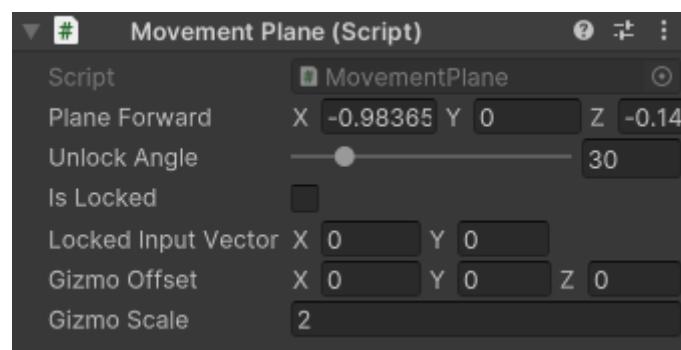


Figura 3.11: Captura del inspector de MovementCompass.

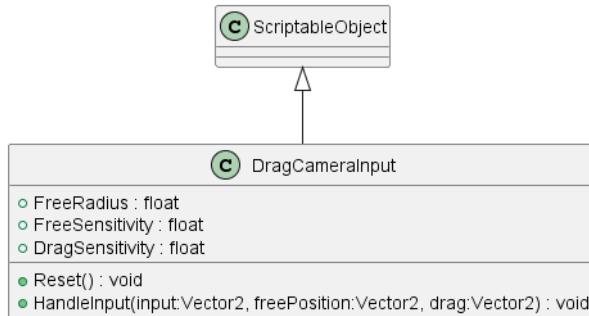


Figura 3.12: Diagrama de clases de DragCameraInput.



Figura 3.13: Captura del inspector de DragCameraInput.

`OnCameraUpdate` actualiza el ángulo del compás en relación con la cámara activa actual, mientras que `OnCameraCut` bloquea temporalmente el ángulo del compás cuando se realiza un corte de cámara, para evitar que la dirección del compás entre u bucle errático de un *frame* al siguiente.

El único campo de este componente que es necesario editar a mano es `UnlockAngle`, que define la diferencia de ángulo que el usuario debe realizar en su método de entrada para que se desbloquee el ángulo del compás.

También se pueden editar los campos `GizmoOffset` y `GizmoScale` para configurar la apariencia del *gizmo* de este componente. `MovementCompass` implementa el mensaje `OnDrawGizmosSelected` para dibujar un *gizmo* en la escena que representa el plano de movimiento y la dirección actual del compás.

El resto de campos se actualizan solos. El ángulo del compás de movimiento está controlado por el campo `PlaneForward`, que representa la última dirección de la cámara sobre el plano de movimiento. Los campos `IsLocked` y `LockedInputVector` representan el estado de bloqueo de entrada actual y la dirección de entrada en la que el compás se encontraría bloqueado.

Para usar este componente, es necesario que el script que mueva al personaje añada una referencia al `MovementCompass` y use su API `OnInputAxis`. Esta toma la última dirección de entrada del usuario, ya sea teclado o mando; desbloquea si es necesario el compás, y proyecta la entrada a una dirección sobre el plano del compás. Si el compás está bloqueado, esta dirección proyectada no cambiará hasta que se desbloquee.

3.2.7 Drag Camera Input

`Rein.Inputs.DragCameraInput` especifica un área de la pantalla en la que el usuario puede mover una retícula libremente. Cuando la retícula alcanza los bordes del área, la cámara es arrastrada en la dirección correspondiente.

`DragCameraInput` no es un componente que se añada a un objeto, sino un `ScriptableObject` de Unity: una clase que se puede serializar, guardarla como `_asset`, *instanciarla* y configurarla desde el editor visual, y además se puede usar como dependencia de otros componentes.

Esta clase tiene tres campos públicos editables desde el editor visual:



Figura 3.14: Diagrama de clases de `ReinEditorUtils`.

- `FreeRadius`, que define el radio del área de la pantalla en la que el usuario puede mover la retícula libremente. Nótese que este “radio” es una distancia en cada eje desde en centro, así que el área es realmente un cuadrado. Esto se hace porque los ejes de entrada, sea ratón o mando, suelen considerarse de manera independiente.
- `FreeSensitivity`, que define la sensibilidad de la retícula en el área de movimiento libre. Este valor se multiplica por la magnitud del vector de entrada obtener la velocidad de la retícula.
- `DragSensitivity`, que define la sensibilidad de la cámara al arrastrar el borde del área de movimiento libre. Este valor se multiplica por la magnitud del vector de entrada para obtener la velocidad de la cámara.

Para usar esta clase, se añade como dependencia de un componente y se asigna una instancia desde el editor visual. Entonces dicho componente puede usar la API `HandleInput`, que dado un vector de un dispositivo de entrada (por ejemplo, stick derecho de un mando), actualiza la posición libre de la retícula, pasada como parámetro, y devuelve un vector de arrastre de cámara que se puede usar para generar un impulso en la cámara orbital.

3.2.8 Clases auxiliares

Como parte de la implementación de los ítems de menú que crean nuevos objetos ya configurados en la escena, una clase estática `ReinEditorUtils` presenta un método auxiliar `AddGameObjectByCommand` que, dado el contexto del editor visual en el que se ejecuta el comando, crea un nuevo objeto básico de Unity (`GameObject`) en la escena, registra esta acción en la pila de acciones que se pueden deshacer (en el sentido de `Ctrl+Z`), y selecciona el objeto recién creado.

Capítulo 4

Prueba y evaluación

Los componentes de cámaras y entrada desarrollados están pensados para ser piezas reutilizables en el desarrollo de una visualización mayor. Por lo tanto, para evaluar la funcionalidad de los componentes desarrollados se ha creado en Unity una escena de demostración.

4.1 Desarrollo de la escena

La escena tiene una composición sencilla, con un personaje controlable por el usuario y una serie de objetos que interactúan con los componentes de cámara de distintos modos.

El modelo 3D del personaje es “Ninja character asset for Unity” de Raquel García Guillem [26], disponible como *asset* en la plataforma itch.io.

Para que el usuario pueda controlar el movimiento de este personaje, se ha creado un componente *Player Controller* que recibe la entrada del usuario, las proyecta mediante su referencia al único objeto con el componente *Movement Compass*, y aplica velocidades al componente nativo de Unity *Animator* que tiene el personaje, para controlar su animación de andar.

Este personaje tiene a su alrededor una cámara orbital en tercera persona. Esta cámara se ha configurado con un componente *Orbit Camera Constraint*, configurado con una curva de distancia que da planos más panorámicos desde la altura.

Para que el usuario pueda controlar la cámara orbital, se ha creado un componente *Orbit Camera Controller* que recibe la entrada del usuario y aplica impulsos al componente *Orbit Camera Constraint* la API ya definida.

Esta cámara está equipada con un componente *Orbit Camera Whiskers*, que detecta predictivamente la presencia de obstáculos en la línea de visión de la cámara y aplica impulsos para alejar la cámara de estos.

Se ha añadido además a la escena además un elemento de interfaz gráfica: una retícula. Se ha usado otro *asset* gratuito, “Crosshair Pack” de KennyNL [27].

Esta retícula está controlada por el *Orbit Camera Controller*, y funciona como en el juego Lost Planet: Se mueve libremente por la pantalla hasta llegar al borde de un área invisible, punto en el que empieza a “arrastrar” los ángulos de la cámara orbital.

Para que la retícula apunte de forma estable, la cámara orbital se ha configurado sin un objetivo *Look At*, en cuyo caso se apunta al pivote, que se encuentra desfasado al lado.

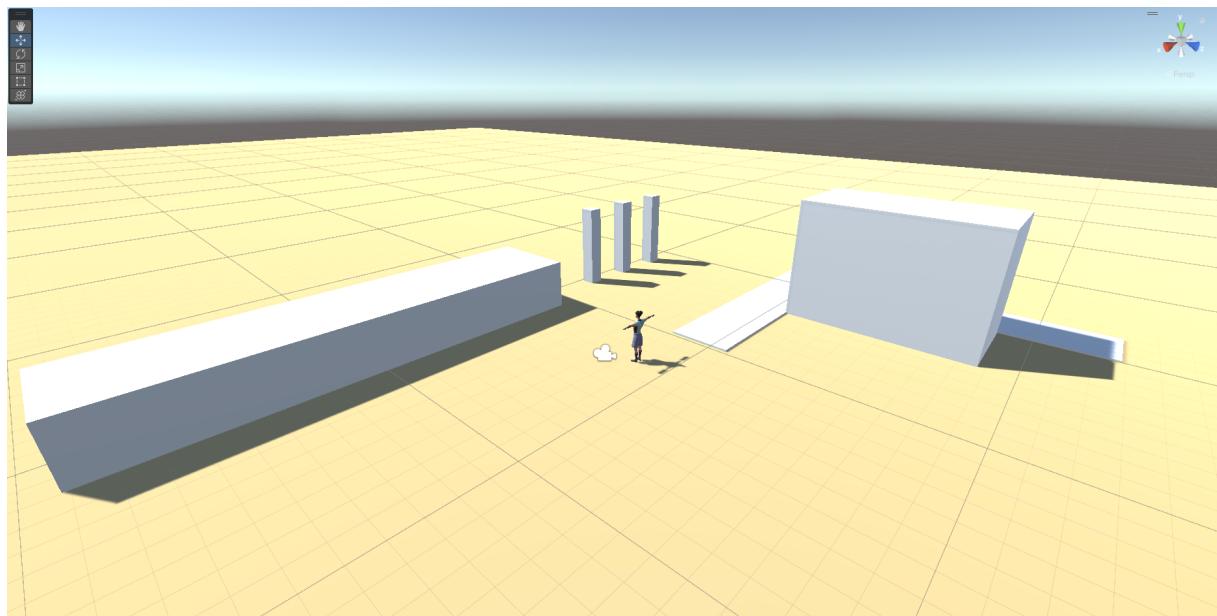


Figura 4.1: Vista panorámica de la escena de la demostración.

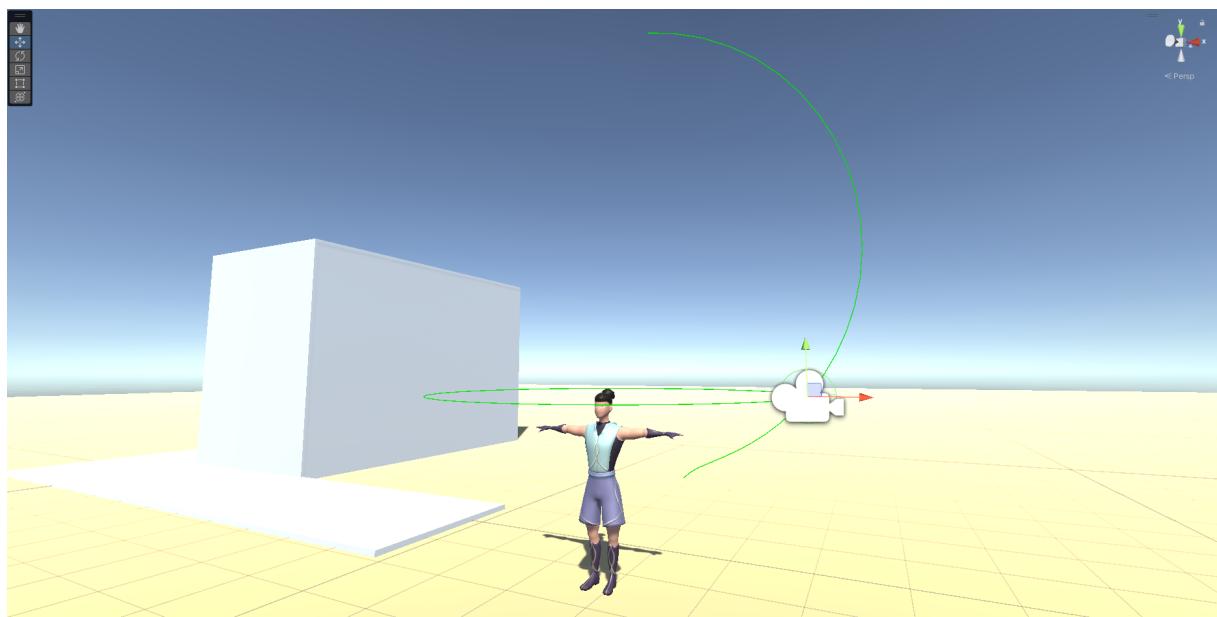


Figura 4.2: Gizmos de la cámara orbital en tercera persona.

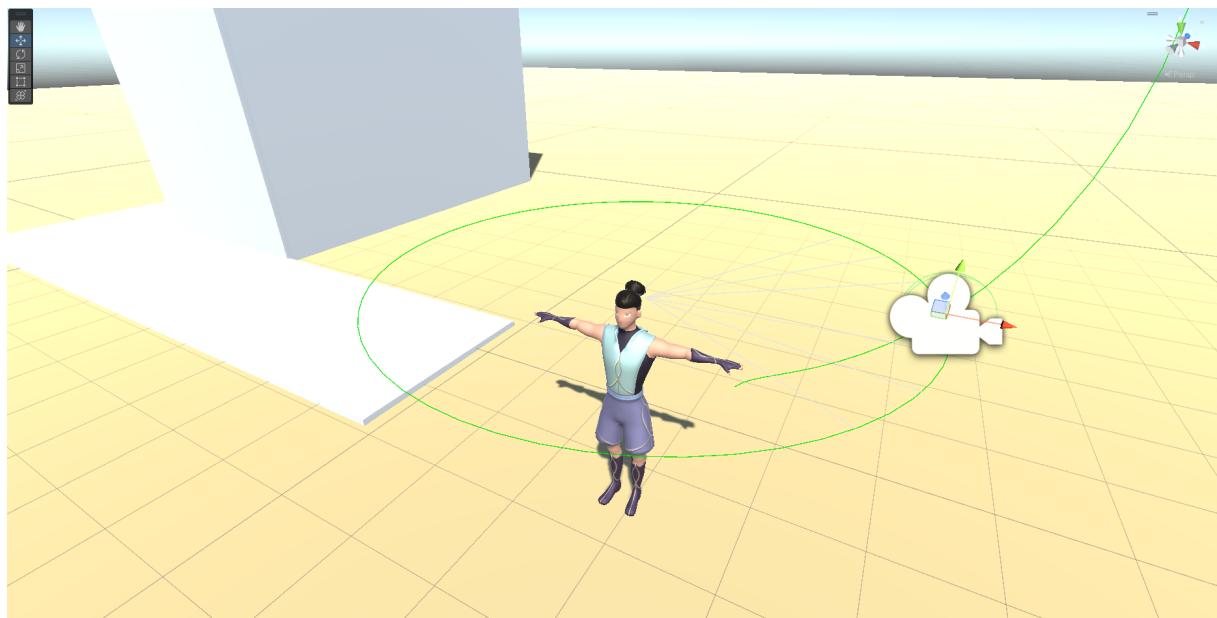


Figura 4.3: Gizmos de los *whiskers* de la cámara orbital.

Esta retícula solo puede verse mientras la cámara orbital está activa; al cambiar a las cámaras fijas, esta desaparece.

Para demostrar la funcionalidad de excepciones de rotura de la línea de visión, se ha creado una nueva capa de Unity *Ignore line-of-sight* que se ha aplicado unos pilares puestos en la escena, y al modelo del personaje para evitar que la cámara colisione con el propio personaje. Obviamente se han configurado los componentes para respetar esta capa.

La demostración tiene una sección con una pasarela formada por dos segmentos en L: al andar sobre ellos la demo corta a un conjunto de cámaras fijas con transiciones.

Esto ha requerido crear dos pequeños componentes más. Por un lado, *Contact Camera Trigger* simplemente coloca sobre la cima de la colección de cámaras de *Main Camera Virtualizer* el conjunto de cámaras fijas a activar.

Por otro lado, por detalles de la implementación del componente nativo de Unity *Character Controller*, que hereda de *Collider*, pero no genera sus mismos mensajes, se ha tenido que añadir un objeto hijo al personaje con un segundo *Collider* que sí lance mensajes colisión de Unity. Para sincronizar automáticamente la configuración de este *Collider* con la del *Character Controller* se ha creado un componente *Character Trigger Collider*.

La entrada del juego está configurada para ser controlada con teclado y ratón o con mando; se ha probado con un mando de PlayStation 4. Si se prueba, otros mandos pueden necesitar otro *preset* de ejes de entrada: estos se pueden cambiar en el menú Edit, Project Settings, Input Manager. El *preset* para el mando de PS4 está disponible como un de los *assets* de la demostración.

4.2 Demostración

Una vez preparada la escena, podemos probar el funcionamiento de los componentes desarrollados. Al iniciar la escena, el usuario se encuentra con el personaje y la cámara orbital en tercera persona. Activando los gizmos de Unity en la escena de juego (arriba a la derecha del panel) y seleccionando un objeto en el editor es posible comprobar el estado de los gizmos durante la demostración.



Figura 4.4: Cámara orbital funcionando.

Movernos por la escena mientras se controla la cámara valida la funcionalidad básica de *Main Camera Virtualizer*, *Orbit Camera Constraint* y *Look At Camera Constraint*. Podemos comprobar que la cámara sigue al personaje, que la cámara orbita alrededor del personaje, y que la cámara se orienta hacia este.

Podemos comprobar además la funcionalidad básica del componente *Movement Compass*, ya que el personaje se mueve sobre el plano horizontal en el mismo sentido que la cámara, incluso mientras esta gira.

Al acercarnos a una gran pared, podemos comprobar que acercar la cámara a esta no la atraviesa, sino que la cámara se desplaza hacia adelante lo suficiente para evitar la colisión y mantener la línea de visión.

También podemos comprobar que, al acercar la cámara en ángulo oblicuo a la pared, *Orbit Camera Whiskers* aleja automáticamente la cámara de la pared mientras el usuario no ejerza control sobre la cámara; en ningún momento la cámara se opone al control del usuario.

En cambio, colocar la cámara perpendicular a la pared (paralela a la normal de su superficie) hace que las fuerzas en cada dirección de los *whiskers* se anulen, y la cámara se mantiene estable.

También podemos andar hacia una esquina de la pared y comprobar que, gracias a los *whiskers*, la cámara se ajusta automáticamente para mantener la línea de visión con el personaje mientras se rodea la esquina.

Sin embargo, si acercamos el personaje a los pilares, podemos comprobar que la cámara permite que los pilares rompan la línea de visión con el personaje, ya que no se consideran obstáculos por la capa que tienen asignada.

Andar sobre la pasarela en forma de L permite probar además los componentes de cámaras fijas, *Fixed Camera Set* y *Fixed Camera Path*, y la funcionalidad de cortes de cámara de *Main Camera Virtualizer* y *Movement Compass*.

Al entrar a caminar en la pasarela desde la posición inicial del personaje en la escena, la cámara orbital corta a una cámara fija al otro lado de los pilares. Este corte produce un evento de corte de cámara que hace que *Movement Compass* bloquee el ángulo del movimiento del personaje mientras que la dirección de la entrada del usuario no cambie significativamente.

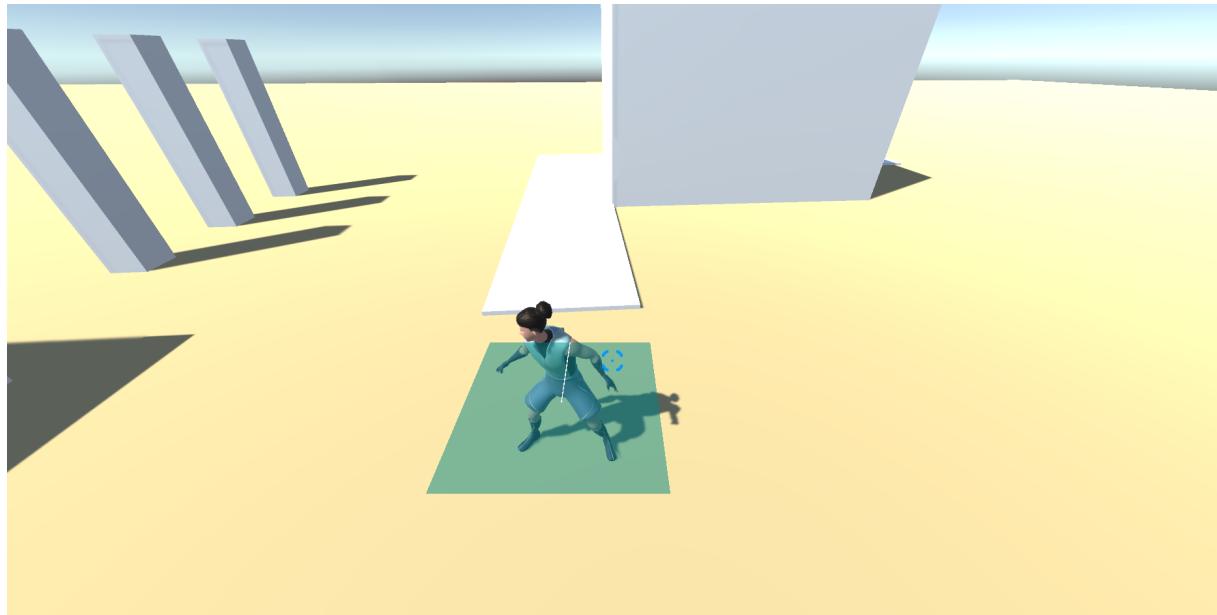


Figura 4.5: Gizmo del compás de movimiento del personaje.



Figura 4.6: Gizmos de la cámara previniendo la colisión con una pared.

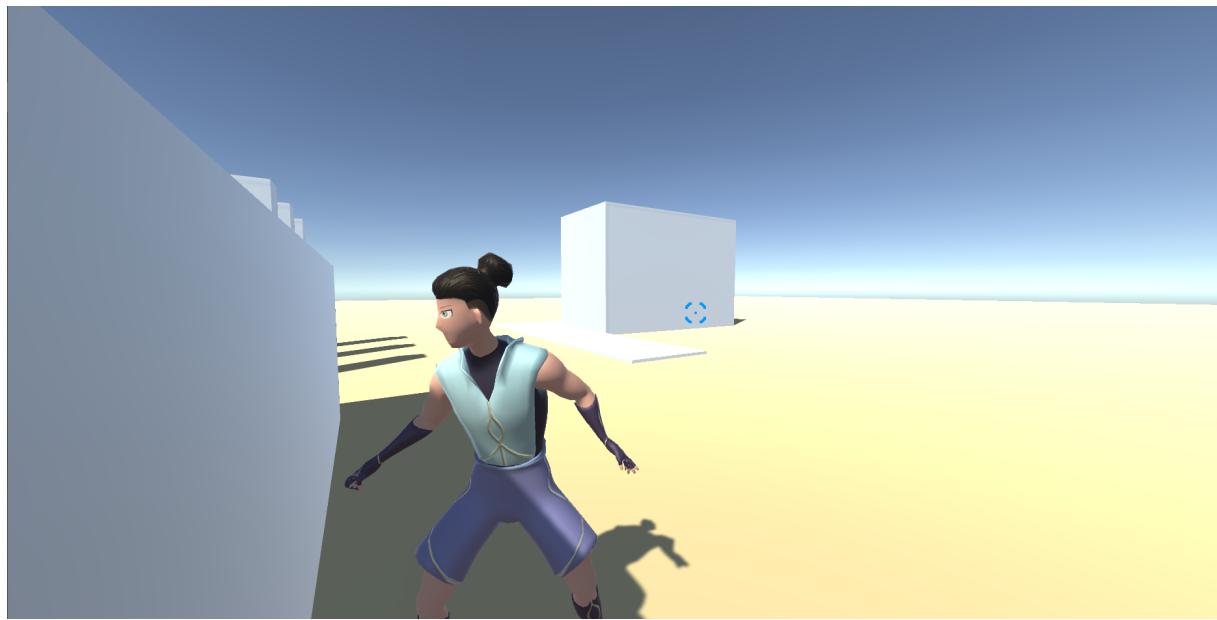


Figura 4.7: Cámara con *whiskers* en una posición inestable.



Figura 4.8: Cámara habiendo sido empujada por el impulso de los *whiskers*.

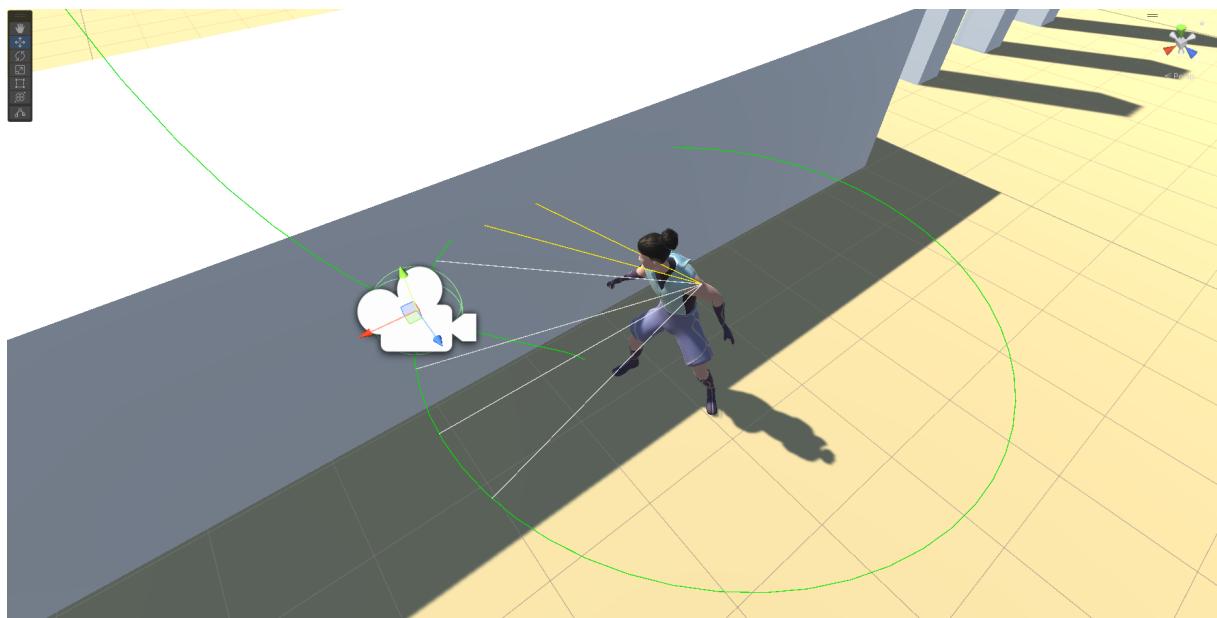


Figura 4.9: Gizmos de los *whiskers* de la cámara en situación inestable.



Figura 4.10: Cámara con *whiskers* dejada por el usuario en una posición estable contra la pared; los impulsos de cada lado se anulan.

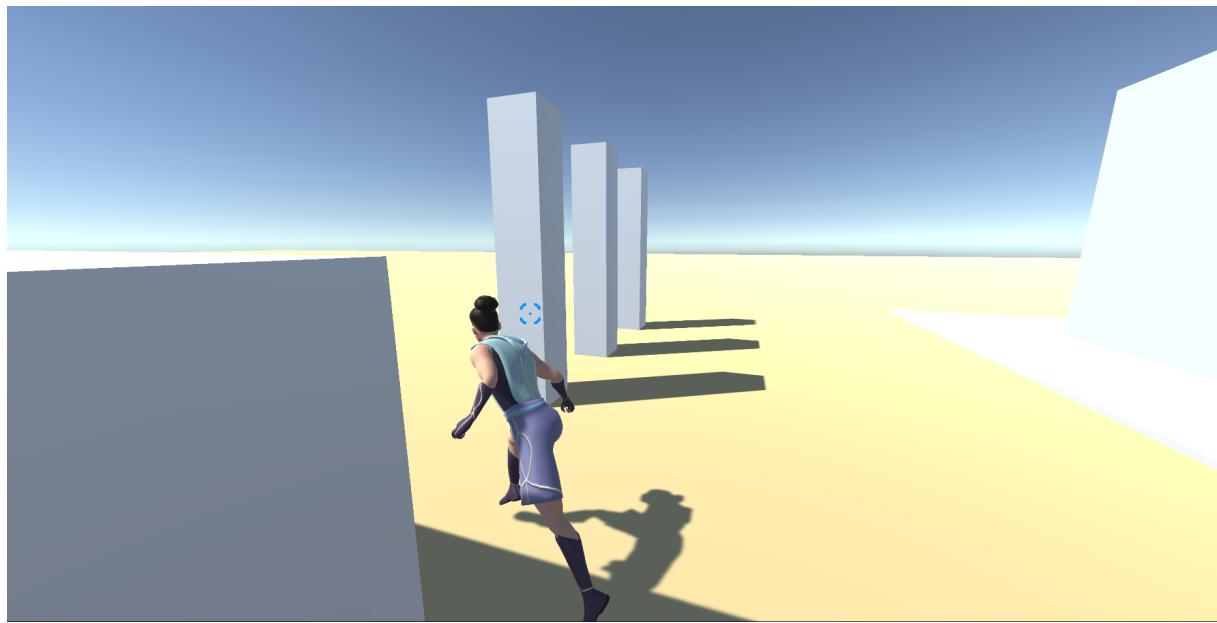


Figura 4.11: Personaje rodeando una esquina.

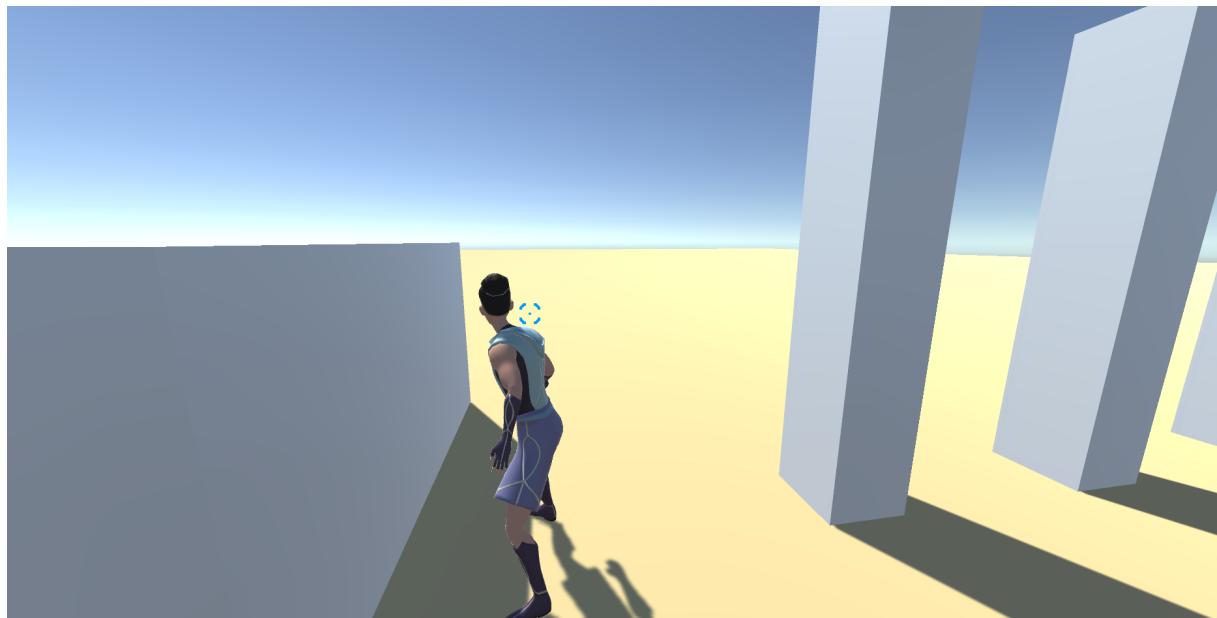


Figura 4.12: Cámara ajustándose para mantener la línea de visión con el personaje mientras rodea la esquina.

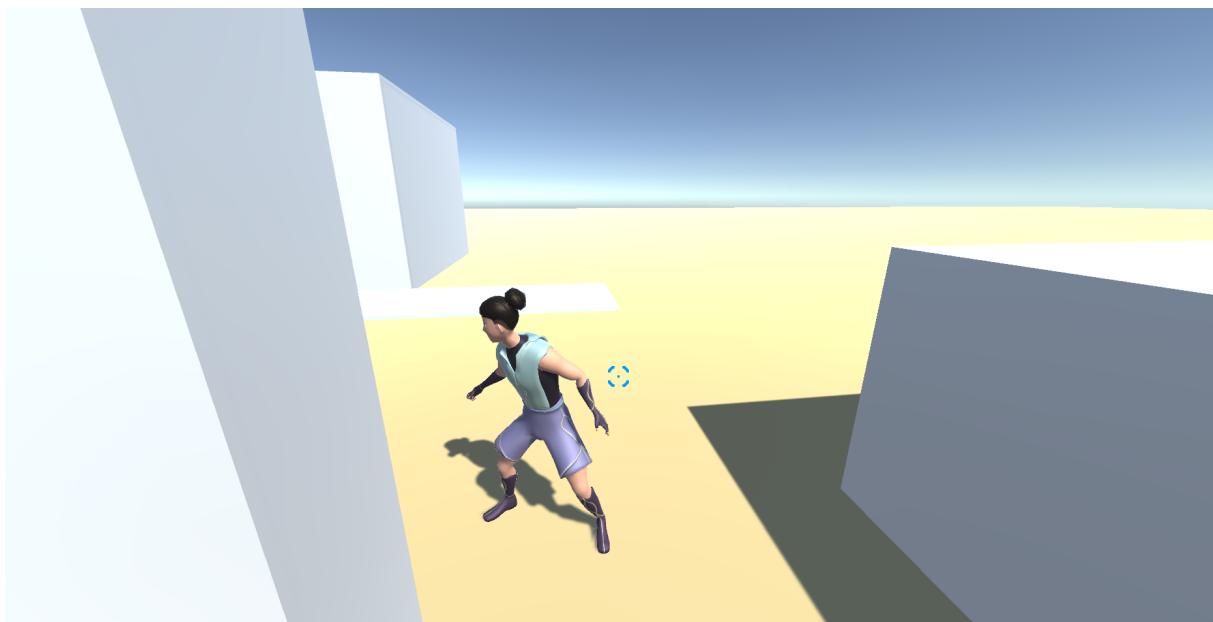


Figura 4.13: Cámara con una línea de visión despejada.



Figura 4.14: Cámara permitiendo que los pilares rompan la línea de visión con el personaje.

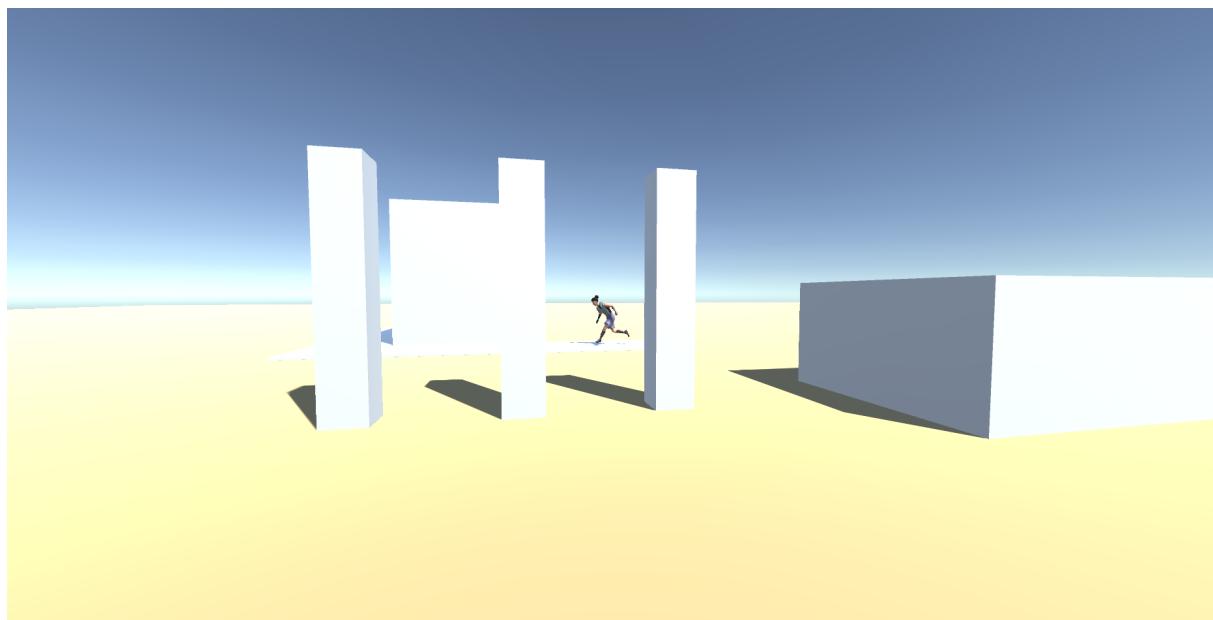


Figura 4.15: Perspectiva de la cámara fija al otro lado de los pilares.

Este bloqueo permite al usuario seguir moviéndose en el mismo sentido que llevaba antes del corte de cámara, en vez de crear un movimiento errático al cambiar de dirección de repente e iniciar un bucle accidental de cortes de cámara, entrando y saliendo de la pasarela.

Atravesando la pasarela, podemos comprobar que a los pilares se les sigue permitiendo interrumpir deliberadamente la línea de visión, en vez de cambiar a otra cámara fija con una peor toma.

Aún en la pasarela, comprobamos que, al cambiar el sentido de la dirección o dejar de mover el personaje, el bloqueo se libera, volviendo a interpretarse la entrada de forma relativa al ángulo de la cámara, en este caso fija.

Al alcanzar la esquina de la pasarela, la segunda cámara, ubicada al fondo, pasa a tener línea de visión con el personaje, y puesto que tiene mejor toma se puede ver cómo una se produce una transición suave entre ambas cámaras en vez de un corte, ya que pertenecen al mismo *Fixed Camera Set*.

Podemos entonces salir de la pasarela, volviendo a producirse un corte de cámara a la cámara orbital inicial. Este corte de nuevo bloquea la dirección de entrada, permitiendo seguir moviéndose en la misma dirección que llevábamos antes del corte, aunque la cámara orbital tenga otra orientación diferente.

Cambiar de dirección, o soltar de nuevo, libera el bloqueo de dirección de compás, devolviendo el control normal del personaje.

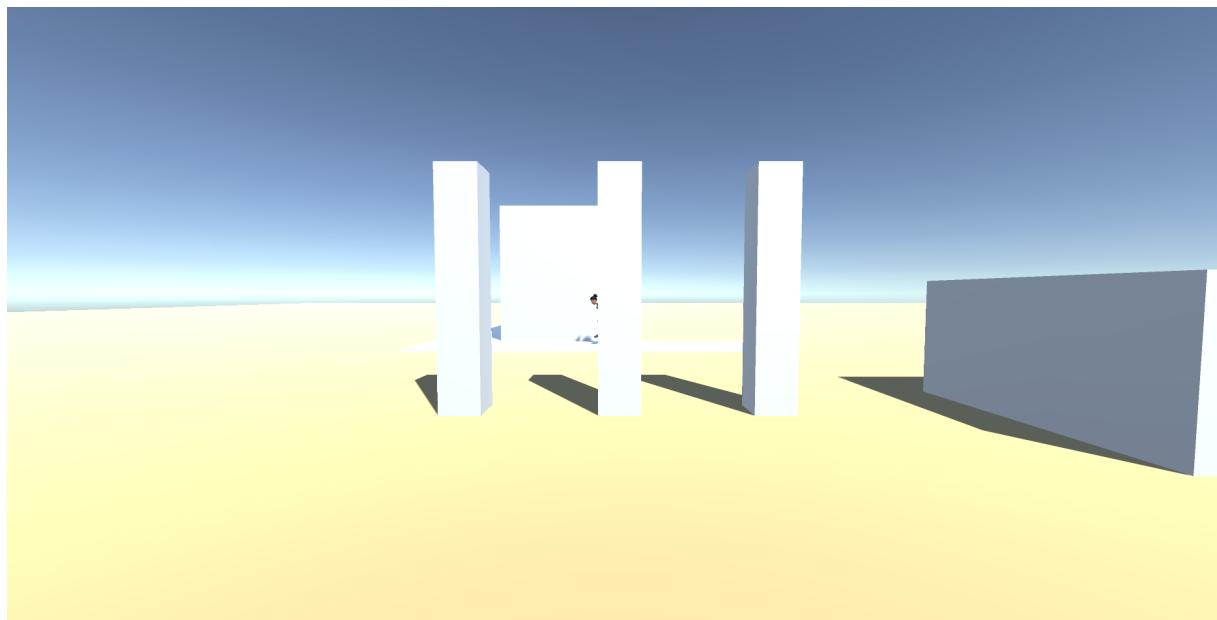


Figura 4.16: Perspectiva de la cámara fija permitiendo que los pilares rompan la línea de visión con el personaje.

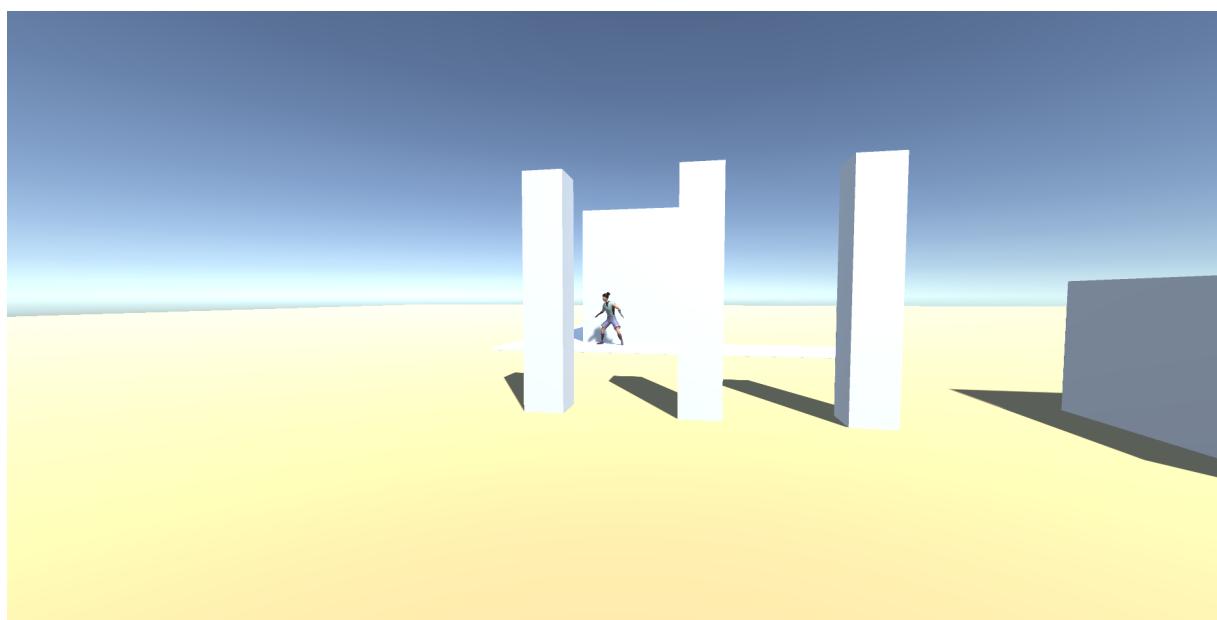


Figura 4.17: La línea de visión de la cámara del fondo está obstruida por la geometría.

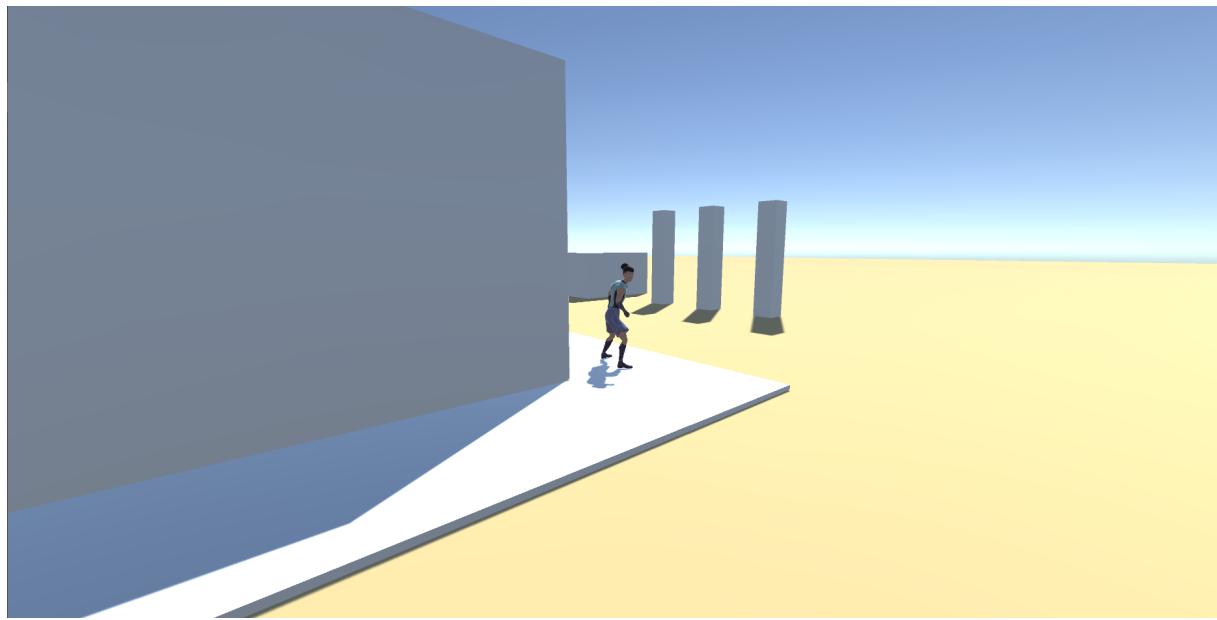


Figura 4.18: La línea de visión de la cámara del fondo se despeja, realizándose una transición suave a esta.

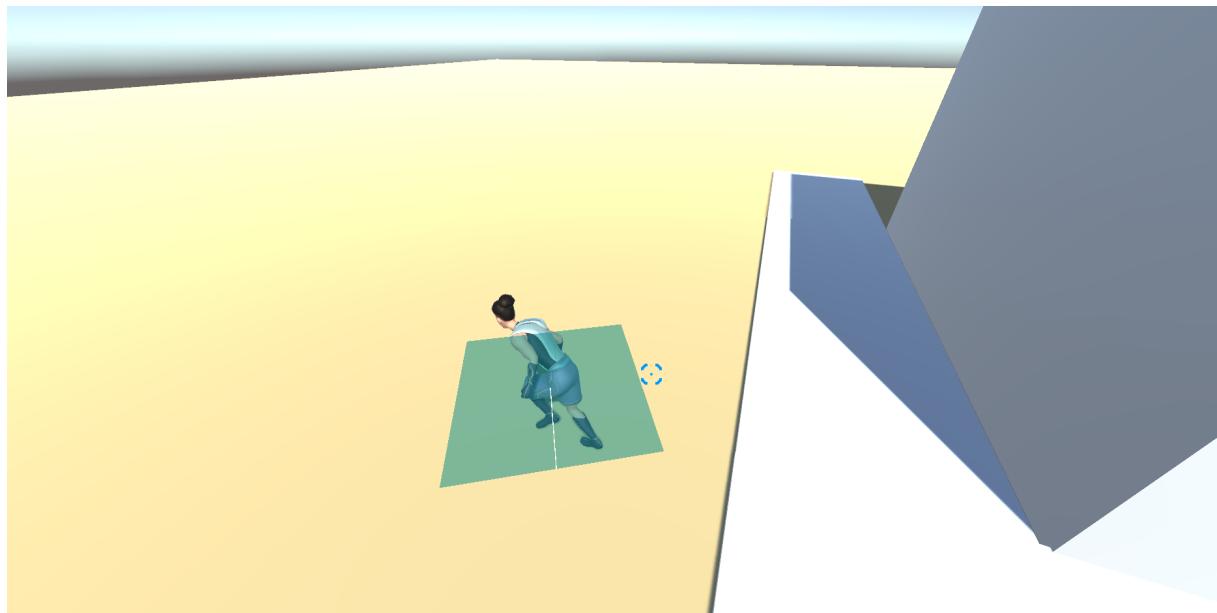


Figura 4.19: Gizmo del compás de movimiento del personaje al salir de la pasarela; el corte de cámara ha bloqueado la orientación.

Capítulo 5

Conclusión

En este trabajo se han estudiado los sistemas de cámaras en el contexto de la visualización en tercera persona y la interactividad del usuario, y se ha diseñado componentes para Unity que permiten a desarrolladores de videojuegos implementar sistemas de cámaras de tercera persona de manera sencilla y que resuelvan los problemas de interactividad encontrados al implementar esquemas de control para estas cámaras. Además, se ha implementado una demostración de todos los componentes desarrollados, que muestra cómo se puede utilizar el sistema de cámaras para prototipar el sistema de cámaras de un videojuego.

En general, se ha logrado el objetivo de diseñar e implementar componentes reutilizables para el caso de uso estudiado y el autor considera que son una buena base para la implementación de sistemas de cámaras de tercera persona más complejos.

Como trabajo futuro, podrían explorarse otros modificadores de cámara, como puntos de interés que, automáticamente, desfasen el pivote de la cámara para mantener en plano un segundo objetivo. Estos podrían implementarse sobre la API de impulsos que se ha desarrollado para este trabajo.

Además, podrían investigarse transiciones de cámaras que se asienten en puntos intermedios de la ruta entre las cámaras del conjunto, llevando estos componentes al ámbito de las cámaras “sobre raíles”. También se podría explorar cómo realizar transiciones entre la cámara orbital y las cámaras fijas en lugar de cortes de cámara, para prototipar transiciones cinematográficas.

Capítulo 6

Coste del proyecto

En este apartado calculamos cuál habría sido la inversión económica de desarrollar el proyecto si se hubiera llevado a cabo en un entorno industrial en lugar de académico.

Consideramos primero el gasto de la mano de obra, un desarrollador *junior* durante unas 200 horas de trabajo distribuidas a lo largo de unos dos meses como parte de un contrato indefinido existente, por un sueldo de 14,19 €/hora tras comprobar los salarios medios en la Comunidad de Madrid en distintas fuentes online como Indeed [28], y teniendo en cuenta las contribuciones a la Seguridad Social [29] que corresponden al trabajo realizado.

Tabla 6.1: Coste de mano de obra

Concepto	Horas	Coste/hora	Coste total
Sueldo bruto programador junior	200	14,19 €	2838,00 €
Contingencias comunes (23,60 %)			669,77 €
Desempleo (5,50 %)			156,09 €
FOGASA (0,20 %)			5,68 €
Formación profesional (0,60 %)			17,03 €
Total			3686,57 €

Hace falta, por supuesto, un ordenador para el desarrollador capaz de ejecutar las herramientas necesarias para el proyecto. Tras considerar un presupuesto de unos 2300 € para adquirir un ordenador y periféricos, o un portátil, a amortizar en 4 años.



Figura 6.1: Suelo programador junior en Madrid, por hora, según Indeed.

Tabla 6.2: Coste de hardware

Concepto	Coste total	Amortización	Coste amortizado (2 meses)
Ordenador	2300,00 €	4 años	95,83 €

Consideramos también que se habrán empleado licencias del software comercial utilizado: Unity por un lado, y por otro, la extensión “C# Dev Kit para Visual Studio Code”, la cual depende de una licencia de Visual Studio 2022.

Tabla 6.3: Precios de licencias

Concepto	Precio mensual	Coste total (2 meses)
Licencia Unity Pro	170,00 €	340,00 €
Licencia Visual Studio 2022	41,98 €	83,96 €
Total		423,96 €

Podemos entonces calcular la inversión total del proyecto:

Tabla 6.4: Inversión total en el proyecto

Concepto	Coste total
Mano de obra	3686,57 €
Hardware	95,83 €
Licencias	423,96 €
Total	4206,36 €

Bibliografía

- [1] Wikipedia, *List of best-selling video games in the United States by year*, https://en.wikipedia.org/wiki/List_of_best-selling_video_games_in_the_United_States_by_year, [Online].
- [2] Wikipedia, *List of best-selling video games*, https://en.wikipedia.org/wiki/List_of_best-selling_video_games, [Online].
- [3] D. Marsh, *Applied Geometry for Computer Graphics and CAD* (Springer Undergraduate Mathematics Series), eng, 2nd ed. 2005. 2005, 1 online resource (360 p.) ISBN: 1-84628-109-1.
- [4] T. Dalling, *Explaining Homogeneous Coordinates & Projective Geometry*, <https://www.tomdalling.com/blog/modern-opengl/explaining-homogenous-coordinates-and-projective-geometry/>, [Online], 2014.
- [5] M. Haigh-Hutchinson, *Real Time Cameras : a Guide for Game Designers and Developers*. eng. 2009, 1 online resource (530 p.) ISBN: 1-000-06508-1.
- [6] Wikipedia, *Mark Haigh-Hutchinson*, https://en.wikipedia.org/wiki/Mark_Haigh-Hutchinson, [Online].
- [7] J. Nesky, *50 Game Camera Mistakes*, <https://www.youtube.com/watch?v=C7307qRmlMI>, [Online], 2014.
- [8] SteamDB, *What are games built with and what technologies do they use?*, <https://steamdb.info/tech/>, [Online].
- [9] Unity Technologies, *Unity*, <https://unity.com/>, [Software].
- [10] Unity Technologies, *Unity - Manual: Camera component*, <https://docs.unity3d.com/Manual/class-Camera.html>, [Online].
- [11] Unity Technologies, *Cinemachine Documentation*, <https://docs.unity3d.com/Packages/com.unity.cinemachine@2.3/manual/index.html>, [Online].
- [12] Epic Games, *Unreal Engine*, <https://www.unrealengine.com/>, [Software].
- [13] Epic Games, *Player-Controlled Cameras*, https://dev.epicgames.com/documentation/en-us/unreal-engine/quick-start-guide-to-player-controlled-cameras-in-unreal-engine-cpp?application_version=5.3, [Online].
- [14] Thatgamecompany, *Journey*, [Videojuego], 2012.
- [15] PlatinumGames Inc., *Nier: Automata*, [Videojuego], 2017.
- [16] Capcom, *Lost Planet 2*, [Videojuego], 2010.
- [17] Nintendo, *Splatoon*, [Serie de videojuegos].

- [18] A. Ramcharitar y R. J. Teather, “A Fitts’ Law Evaluation of Video Game Controllers: Thumstick, Touchpad and Gyrosensor”, en *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, ép. CHI EA ’17, Denver, Colorado, USA: Association for Computing Machinery, 2017, págs. 2860-2866, ISBN: 9781450346566. DOI: [10.1145/3027063.3053213](https://doi.org/10.1145/3027063.3053213). dirección: <https://doi.org/10.1145/3027063.3053213>.
- [19] P. Isbej y F. J. Gutierrez, “Gaze as a Navigation and Control Mechanism in Third-Person Shooter Video Games”, en *HCI in Games: Experience Design and Game Mechanics*, X. Fang, ed., Cham: Springer International Publishing, 2021, págs. 45-56, ISBN: 978-3-030-77277-2.
- [20] A. Taheri, Z. Weissman y M. Sra, “Exploratory Design of a Hands-free Video Game Controller for a Quadriplegic Individual”, en *Proceedings of the Augmented Humans International Conference 2021*, ép. AHs ’21, Rovaniemi, Finland: Association for Computing Machinery, 2021, págs. 131-140, ISBN: 9781450384285. DOI: [10.1145/3458709.3458946](https://doi.org/10.1145/3458709.3458946). dirección: <https://doi.org/10.1145/3458709.3458946>.
- [21] M. Naftis, G. Tsatiris y K. Karpouzis, *How Camera Placement Affects Gameplay in Video Games*, 2021. arXiv: [2109.03750 \[cs.MM\]](https://arxiv.org/abs/2109.03750). dirección: <https://arxiv.org/abs/2109.03750>.
- [22] M. Noland, *Unreal Property System (Reflection)*, <https://www.unrealengine.com/es-ES/blog/unreal-property-system-reflection>, [Online], 2014.
- [23] Epic Games, *Objects*, <https://dev.epicgames.com/documentation/en-us/unreal-engine/objects-in-unreal-engine>, [Online].
- [24] Epic Games, *Blueprints Visual Scripting*, <https://dev.epicgames.com/documentation/en-us/unreal-engine/blueprints-visual-scripting-in-unreal-engine>, [Online].
- [25] Unity Technologies, *Unity User Manual 2022.3 (LTS)*, <https://docs.unity3d.com/2022.3/Documentation/Manual/UnityManual.html>, [Online], 2022.
- [26] R. G. Guillem, *Ninja character asset for Unity*, <https://raquelgg00.itch.io/ninja-character-asset-for-unity>, [Online], 2021.
- [27] KennyNL, *Crosshair Pack*, <https://kenney-assets.itch.io/crosshair-pack>, [Online].
- [28] Indeed, *Sueldo de Programador/a junior en Comunidad de Madrid*, <https://es.indeed.com/career/programador-junior/salaries/Comunidad-de-Madrid>, [Online].
- [29] Seguridad Social, Gobierno de España, *Bases y tipos de cotización 2024*, <https://www.seg-social.es/wps/portal/wss/internet/Trabajadores/CotizacionRecaudacionTrabajadores/36537>, [Online].

Apéndice A

Elementos adicionales entregables

Este anexo documenta los elementos que no son entregables como parte de este documento, pero que se han desarrollado como parte de este TFG.

El código de los componentes de Unity desarrollados, así como el código y *assets* de la demostración se pueden encontrar en el repositorio que se ha creado en GitHub para esto proyecto, disponible en la URL <https://github.com/amongonz/TFG>.

El código de los componentes de Unity se encuentra en la carpeta Project/Assets/TFG Components del repositorio, mientras que Project/Assets/Demo contiene el código y *assets* de la demostración. Se encontrará también una copia de los recursos de terceros utilizados en la demostración, bajo la carpeta Project/Assets/Third Party Assets.

Una copia de este documento en formato PDF debería encontrarse también en la raíz del repositorio, bajo el nombre TFG.pdf.

Apéndice B

Herramientas

Las herramientas necesarias para la elaboración del proyecto han sido:

- PC
- Sistema operativo Windows 10
- Mando de PlayStation 4
- Unity 2022.3 LTS (*Long Term Support*)
- Editor de código Visual Studio Code
- Extensión “C# Dev Kit” para Visual Studio Code
- Extensión “Unity” para Visual Studio Code
- Git
- MikTeX
- TeXstudio
- Pandoc
- PlantUML



Universidad
de Alcalá

2024