TecnoCampus

Centres universitaris adscrits a la

upf. Universitat Pompeu Fabra Barcelona

Grado en Diseño y Producción de Videojuegos

Diseño y desarrollo de una herramienta para la creación de controladores de personaje en 1ª persona

MEMORIA

Xavier Lucena Pallarès

Tutor: Dr. Adso Fernández Baena

2021-2022

TecnoCampus Mataró-Maresme

Abstract

This project consists of the design and development of a tool for video game designers that allows them to create, edit and adjust 1st person character controllers for Walking Simulator, First Person Shooter and First Person Platformer video games. It has a graphical interface that facilitates its use, without the user needing access to its code. The development of the tool has been carried out in the Unity videogame engine, and different use cases have been elaborated to demonstrate its possibilities and applications in the mentioned genres.

Resum

Aquest treball consisteix en el disseny i el desenvolupament d'una eina per a dissenyadors de videojocs que els permeti crear, editar i ajustar controladors de personatge en 1a persona, per a videojocs de gènere *Walking Simulator, First Person Shooter* i *First Person Platformer*. Aquesta compta amb una interfície gràfica que en facilita l'ús, sense necessitat d'accés a codi per part de l'usuari. El desenvolupament de l'eina s'ha realitzat al motor de videojocs *Unity*, i s'han elaborat diferents casos d'ús per demostrar les seves possibilitats i aplicacions en els gèneres esmentats.

Resumen

Este trabajo consiste en el diseño y desarrollo de una herramienta para diseñadores de videojuegos que les permita crear, editar y ajustar controladores de personaje en 1ª persona, para videojuegos de género *Walking Simulator, First Person Shooter* y *First Person Platformer*. Esta cuenta con una interfaz gráfica que facilita su uso, sin necesidad de acceso a código por parte del usuario. El desarrollo de la herramienta se ha realizado en el motor de videojuegos *Unity*, y se han elaborado diferentes casos de uso para demostrar sus posibilidades y aplicaciones en los géneros mencionados.

Índice

Índice de figuras

Índice de tablas

1. Introducción

El desarrollo de videojuegos es un proceso complejo que implica la creación y unión de una gran cantidad de sistemas como, por ejemplo, el pipeline de creación de *assets*, el control de código fuente, o incluso la gestión de información en un proceso de traducción. Todos estos sistemas se deben adaptar a los tiempos de cada proyecto, por lo que la rapidez y la eficiencia son esenciales para su correcta ejecución, y es por esta razón que existen las herramientas.

Doran (2015) argumenta que las herramientas son muy usadas actualmente por estudios enfocados a la creación de videojuegos AAA, debido a la alta complejidad que su desarrollo presenta. Estas se crean con el objetivo de facilitar el desarrollo de videojuegos a diseñadores y artistas, reduciendo, el tedio de tareas repetitivas que suelen ser comunes en la creación de contenido para videojuegos, permitiéndoles acceder a la funcionalidad final sin tener que pasar por procesos innecesarios. Estas herramientas se pueden crear para todo tipo de finalidades, y un ejemplo de estas es la creación de controladores de personaje.

Según Hodent (2017) a la hora de crear videojuegos es importante crear controles que se sientan receptivos y disfrutables al utilizarlos, dando así al jugador una sensación de presencia en el mundo de juego. Esta sensación de juego está recogida en el concepto de *Game Feel*, el cual fue definido y matizado por Steve Swink en *Game Feel: A Game Designer's Guide to Virtual Sensation* (Swink, 2008). Todo videojuego tiene su *Game Feel* independientemente de su control (Swink, 2008) por lo que es necesario profundizar en el control de personajes.

Teniendo como base la importancia de las herramientas y el control de personajes, se ha desarrollado una herramienta que permite a diseñadores crear controladores de personaje en 1ª persona de manera rápida y eficiente, con una interfaz gráfica que prescinde de uso de código. Esta permite elegir para que género se pretende crear el controlador, cuyos controles se han excluido a teclado y ratón debido a la comodidad que estos presentan respecto a un mando de consola. Por último, la herramienta permite la personalización de *Game Feel* de algunas de las mecánicas de cada controlador, aumentando la posibilidad de resultados que esta puede

ofrecer. Su desarrollo se ha realizado con *Unity,* uno de los motores más usados en la industria del desarrollo de videojuegos.

Este trabajo detalla el proceso de creación de esta herramienta, desde la definición de los objetivos hasta la exposición de los resultados obtenidos. Para ello, primero, se estipulan estos objetivos y se detallan los fundamentos teóricos necesarios en un marco teórico. A continuación, se analizan dos referentes disponibles en el mercado con un propósito similar al de la herramienta de este trabajo. Posteriormente, se expone el diseño metodológico, donde se detallan las diferentes fases en las que se ha dividido el desarrollo. Y, por último, se muestran los resultados obtenidos y las conclusiones sacadas después del proceso.

2. Objetivos

Los objetivos de este trabajo se dividen en principales y secundarios. Estos se presentan a continuación.

2.1. Objetivo principal

El objetivo principal de este trabajo es el siguiente:

I. Desarrollar una herramienta que permita la creación de controladores de personaje en 1ª persona de diferentes géneros con *Game Feel* personalizable.

2.2. Objetivos secundarios

Los objetivos secundarios de este trabajo son los siguientes:

I. Analizar las características y mecánicas de control de personaje de videojuegos en 1ª persona de diferentes géneros.

II. Programar controladores de personaje en 1ª persona de diferentes géneros en el motor *Unity*.

III. Implementar una interfaz gráfica personalizada para la herramienta capaz de crear los controladores sin uso de código.

IV. Implementar la personalización de *Game Feel* en mecánicas específicas de cada controlador.

V. Validar las posibilidades de la herramienta con la creación de diferentes casos de uso.

3. Marco Teórico

El siguiente capítulo contiene toda la información necesaria para comprender los fundamentos teóricos que la fase de desarrollo requiere. Este está formado por los siguientes apartados: *Game Feel,* Cámara y control de personaje, y Creación de herramientas para diseñadores.

3.1. *Game Feel*

Game Feel (o sensación de juego) es un concepto que ha costado desarrollar y definir durante muchos años debido a las varias interpretaciones que ha tenido a lo largo de los años. En este apartado se define y clasifica el término *Game Feel* según el libro de Steve Swink *Game Feel: A Game Designer's Guide to Virtual Sensation* (Swink, 2008).

3.1.1. Definiendo Game Feel

El término *Juicy* (a partir de ahora jugoso/jugosidad) se conoció por primera vez a través del artículo *How to Prototype a Game in Under 7 Days* de Matt Kucic (Jonasson y Purho, 2012) y ha ido evolucionando con el paso de los años. Kucic (2005) define que un elemento jugoso de un videojuego responde a todo lo que hace el jugador, haciéndolo sentir poderoso y con el control del mundo de juego. Juul (2010) y Brown (2013) aumentan la definición, añadiendo que la jugosidad está ligada al *feedback* de las acciones del jugador, mejorando así su experiencia ofreciendo una sensación visceral satisfactoria. Fue Steve Swink quien quiso investigar el concepto de jugosidad y darle nombre propio.

Swink (2007) hizo aparecer el término *Game Feel* definiéndolo como un concepto que solo se puede conocer cuando se ha experimentado con anterioridad, como algo que se percibe de manera subconsciente, una combinación de vistas, sonidos y respuesta inmediata a la acción. Swink (2008) define en su libro *Game Feel: a game designer's guide to virtual sensation* (Swink, 2008) que el *Game Feel* es la sensación táctil y kinestésica de controlar en tiempo real un objeto virtual en un espacio simulado, con interacciones acentuadas por el pulido. En esta definición se encuentran los tres bloques principales que permiten crear experiencias jugables

con un buen *Game Feel*: control en tiempo real, espacio simulado y pulido (Swink, 2008). Para entender bien la definición hace falta definir cada uno de estos bloques.

3.1.1.1. Control en tiempo real

Swink (2008) define el control en tiempo real como el "flujo ininterrumpido de mando del jugador en el juego, resultando en un control preciso y continuo sobre un avatar en movimiento". (Traducido del autor, p.35). Lo define como una forma de interactividad entre ordenador (o consola) y usuario (o jugador) y esta, a su vez, la define como "el proceso cíclico entre dos o más agentes activos en el que cada agente escucha, piensa y habla de forma alternada" (Crawford, 2003) (traducido del autor, p.29). Con esta definición, Swink (2008) representa el bucle de interactividad entre jugador y ordenador tal y como se ve en la Fig. 3.1.

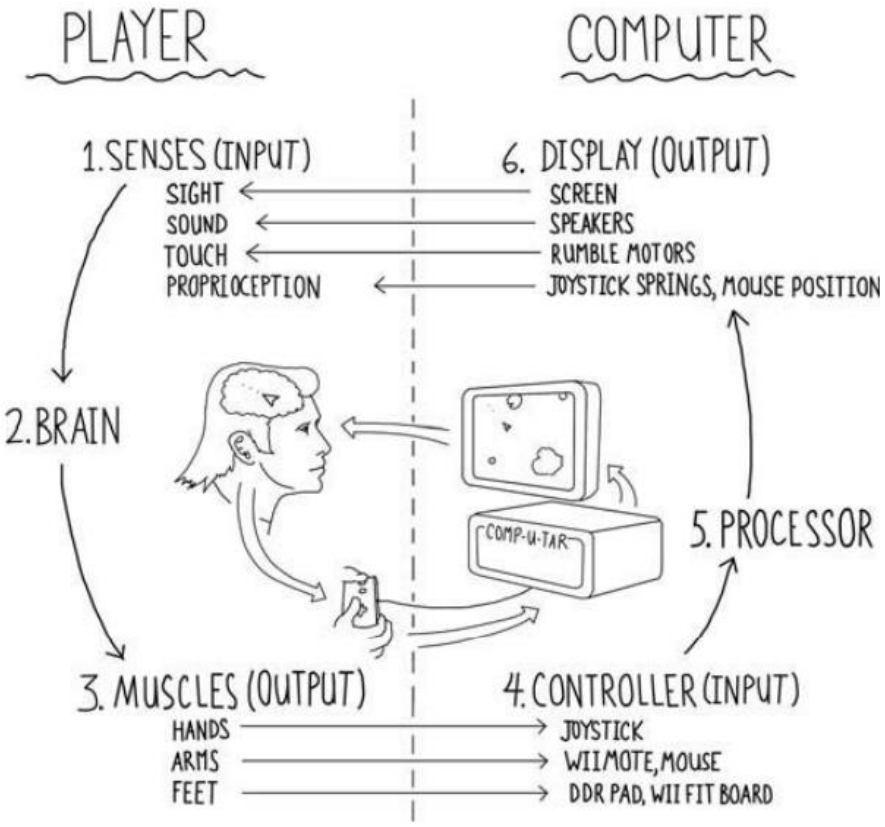

Figura 3.1. El bucle de interactividad entre jugador y ordenador. Fuente: Swink, 2008.

La mitad izquierda de la Fig. 3.1 representa el proceso por el que pasa el jugador para interactuar con el ordenador, y en este proceso actúan propiedades inalterables de la percepción humana (Swink, 2008). Card, Moran y Newell (1986)

estipulan que la mente está compuesta de tres procesadores parcialmente acoplados: el procesador perceptivo, el procesador cognitivo y el procesador motor. El procesador perceptivo recoge la entrada de los sentidos (punto 1 de la Fig. 3.1) y les busca sentido mediante la búsqueda de patrones y relaciones, después el procesador cognitivo decide qué hacer con la información obtenida y por último el procesador motor ordena a los músculos (punto 3 de la Fig. 3.1) la ejecución de esta decisión (Swink, 2008). Si bien la fisiología de cada persona puede hacer que cada procesador tenga diferentes velocidades de proceso, Card, Moran y Newell (1986) concluyen que el mínimo tiempo que necesita una persona para percibir el mundo es de 240 milisegundos, repartidos en los diferentes procesadores mentales de la siguiente manera:

- Procesador perceptivo: ~100 ms.
- Procesador cognitivo: ~70 ms.
- Procesador motor: ~70 ms.

Debido a que las propiedades perceptivas del ser humano son inalterables, el control en tiempo real depende enteramente del ordenador (mitad derecha de la Fig. 3.1). Swink (2008) define tres criterios que un ordenador debe mantener en el tiempo para dar la sensación de control en tiempo real:

- Impresión de movimiento: El número mínimo de fotogramas por segundo que el jugador necesita para mantener la impresión de movimiento debe ser mayor o igual a 10. A más fotogramas por segundo, mejor impresión de movimiento se tiene (Swink, 2008).
- Respuesta instantánea: En un videojuego la respuesta nunca será instantánea debido al tiempo de procesamiento (punto 5 de la Fig. 3.1) que tiene el ordenador, pero es importante que la respuesta se reciba antes de que acabe el ciclo de percepción. Es decir, se tiene que recibir respuesta antes de que pasen 240 milisegundos de haber realizado una entrada. Si hay una respuesta de 50 milisegundos, el jugador percibe que la respuesta ha sido instantánea, pero si hay una respuesta superior a 200 milisegundos, el jugador percibe una respuesta lenta (Swink, 2008).

- Continuidad de respuesta: El videojuego necesita actualizarse en tiempo real más rápido que el tiempo que tarda el jugador en usar el procesador perceptivo para ver el siguiente fotograma, es decir, el jugador debe poder realizar la siguiente entrada 100 milisegundos después de haber hecho la primera. Si no lo hace, el jugador nota retraso en sus acciones. Es por eso que para que haya sensación de control se tiene que ofrecer entrada continua al jugador, lo cual le permite modular una entrada en el tiempo (Swink, 2008).

Sin embargo, Swink (2008) argumenta que para que haya sensación de control en tiempo real es necesaria la existencia de un espacio simulado.

3.1.1.2. Espacio simulado

Según Schell (2008) todos los videojuegos ocurren en un espacio, y este es el círculo mágico del juego. Para Swink (2008) el espacio simulado se refiere a las interacciones físicas simuladas en el espacio virtual, las cuales deben existir de manera literal en el mundo y ser percibidas de forma activa por el jugador. Estas interacciones se manifiestan en el mundo del juego a partir de las colisiones entre los objetos del mundo y a partir del diseño de niveles. Las colisiones permiten al jugador interactuar con el mundo de la misma manera o de una manera similar a la realidad y el diseño de niveles provee al jugador de objetos con los que interactuar, moverse alrededor o usar de referencia para la impresión de velocidad (Swink, 2008).

3.1.1.3. Pulido

Por último, Swink (2008) habla de pulido, definiéndolo como cualquier efecto que mejora artificialmente la interacción entre los objetos del mundo sin cambiar ni alterar el núcleo de juego. Si se eliminan todos los efectos de pulido, la experiencia del jugador es menos convincente y, por lo tanto, menos atractiva. Ejemplos de pulido pueden ser efectos de partículas, animaciones de personajes u objetos, sonidos o efectos de cámara. Estos efectos de pulido ayudan a hacer más verosímil la naturaleza física de los objetos que se encuentran en el espacio simulado (Swink, 2008).

3.1.2. Clasificación de Game Feel

Una vez vista en detalle la definición de *Game Feel* ya es posible clasificar cada juego en un tipo de *Game Feel* distinto. Para ello Swink (2008) utiliza el sistema de clasificación gráfico que se ve en la Fig. 3.2.

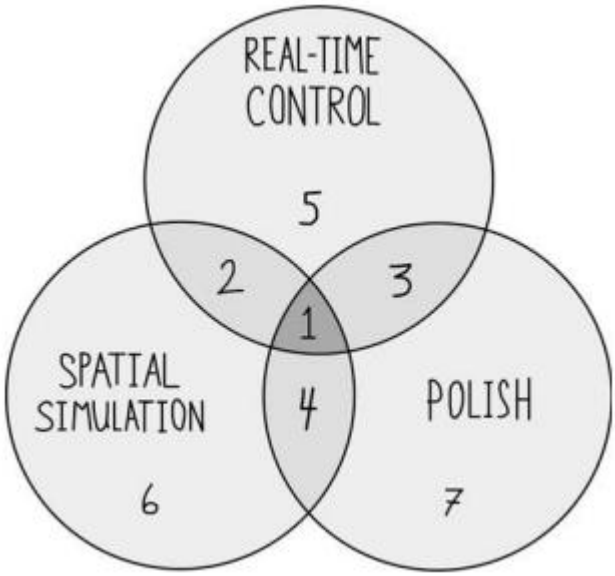

Figura 3.2. Sistema de clasificación de Game Feel. Fuente: Swink, 2008.

Este sistema de clasificación tiene 7 tipos de *Game Feel* y sus características son las siguientes:

1: Videojuegos que tienen todos los componentes especificados por la definición de *Game Feel.* Aquí se ubican la mayoría de videojuegos del mercado. Este recibe el nombre de *Game Feel* verdadero (Swink, 2008).

2: Videojuegos en los que se percibe control en tiempo real e interacción con espacio simulado, pero sin componentes de pulido. Normalmente, los videojuegos que salen al mercado tienen alguna parte de pulido, por lo que este caso no se suele dar (Swink, 2008).

3: Videojuegos en los que se perciben los efectos de pulido, pero no se percibe espacio simulado, y, por lo tanto, no se percibe control en tiempo real. Estos videojuegos no suelen existir, ya que pueden generar disonancia en el jugador (Swink, 2008).

4: Videojuegos en los que el espacio simulado se percibe de forma pasiva, sin control en tiempo real. Sin embargo, los efectos de pulido ayudan enfatizar las interacciones con el espacio simulado (Swink, 2008).

5: No existen videojuegos en los que haya solo control en tiempo real, ya que se requiere de espacio simulado para percibirlo (Swink, 2008).

6: Videojuegos en los que se percibe el espacio simulado de manera pasiva, sin ningún tipo de control en tiempo real o pulido (Swink, 2008).

7: Videojuegos con espacio simulado abstracto, sin colisiones. En estos juegos los efectos de pulido venden el peso de las interacciones, sin control en tiempo real (Swink, 2008).

3.1.3. Métricas de salida

Swink (2008) estipula que hay tres pasos esenciales para que el jugador reciba la respuesta de una entrada realizada con un dispositivo de entrada:

1. La señal de entrada es recibida por la consola u ordenador.
2. La señal de entrada es interpretada y filtrada.
3. La señal de entrada modula un parámetro del videojuego.

Cada señal de entrada está vinculada a un parámetro específico del videojuego, y este cambia en el tiempo según las decisiones que se hayan tomado en diseño. Estos parámetros pueden establecer una posición u orientación nuevas para un objeto del videojuego, añadir una fuerza o torque al mismo, cambiar variables del espacio simulado como puede ser su gravedad o incluso crear nuevas entidades, como por ejemplo balas (Swink, 2008).

Para mejor comprensión de cómo evoluciona la modulación de los parámetros en el tiempo, Swink (2008) propone usar el modelo ADSR (Fig. 3.3), el cual permite representar la modulación de un parámetro de un videojuego según su entrada específica. El modelo ADSR tiene sus orígenes en la empresa ARP después de que Robert Moog creara el sintetizador *Moog* y se suele usar principalmente en la industria de la música electrónica (Pinch y Trocco, 2004). Este consiste en un eje cartesiano en el que se ve la evolución de la modulación de un determinado

parámetro (Eje Y) en el tiempo (Eje X). Las siglas ADSR se corresponden a las diferentes fases en las que se modula un parámetro:

- Ataque (A): Tiempo en el que el parámetro pasa de su mínimo (normalmente 0) a su máximo después de que se haya recibido la señal de entrada.
- Decaimiento (D): Tiempo en el que el parámetro se regula, pasando de su máximo a un nivel de sostenimiento.
- Sostenimiento (S): Valor del parámetro (estipulado por la rama de diseño) que se mantiene fijo siempre y cuando la señal de entrada no se interrumpa. Esta fase es independiente del tiempo, y solo acaba cuando se deja de recibir la señal de entrada.
- Relajación (R): Tiempo en el que el parámetro pasa de su nivel de sostenimiento a su mínimo, después de que la señal de entrada deja de recibirse.

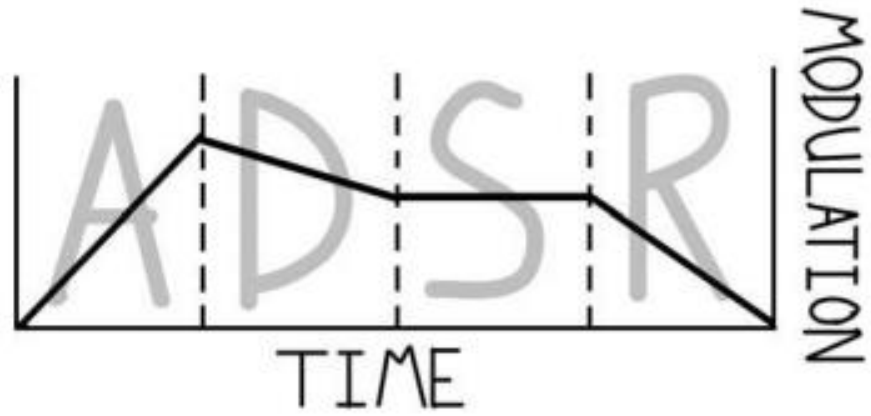

Figura 3.3. Gráfico esquemático del modelo ADSR. Fuente: Swink, 2008.

Con este modelo es posible saber de manera aproximada la sensación de control que tienen los jugadores sobre un parámetro en concreto (Swink, 2008).

3.1.3.1. Apuntes sobre el ataque

Según Swink (2008) el ataque es la fase más importante del modelo y la que hace sentir la receptividad del control. Normalmente, cuando hay un ataque largo, el jugador tiene una sensación de control floja, y en caso de que el jugador perciba que el control tarda más de 100 milisegundos en responder, la impresión de

respuesta instantánea deja de existir y se pierde la sensación de control en tiempo real (Fig. 3.4). Por otro lado, si se usa un ataque corto, el jugador percibe el control receptivo (Fig. 3.5).

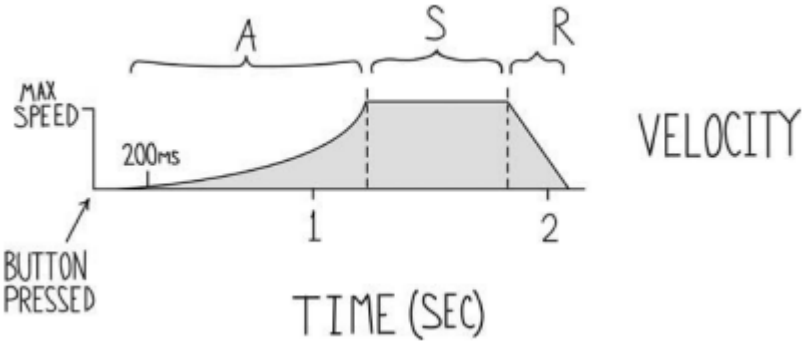

Figura 3.4. Modelo ADSR con un ataque largo. Fuente: Swink, 2008.

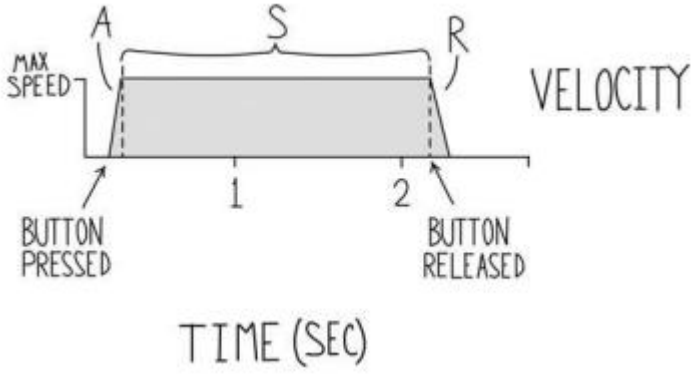

Figura 3.5. Modelo ADSR con un ataque corto. Fuente: Swink, 2008.

Sin embargo, aunque el ataque sea corto, es posible que la sensación de control sea demasiada rígida e inorgánica para el jugador. Para añadir una sensación de control orgánica es aconsejable convertir la fase de ataque en una curva (Swink, 2008) (Fig. 3.6).

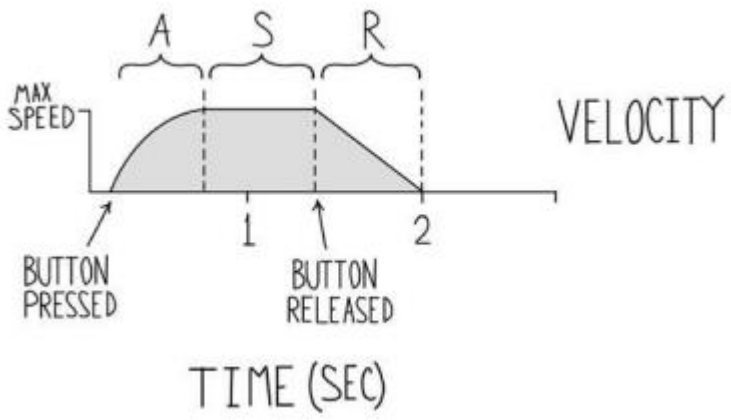

Figura 3.6. Modelo ADSR con una fase de ataque curva. Fuente: Swink, 2008.

3.1.3.2. Apuntes sobre el decaimiento

Swink (2008) estipula que la fase de decaimiento no suele existir en videojuegos, porque de ser así el parámetro solo llegaría al nivel máximo mediante la constante pulsación del botón en la fase de ataque (Fig. 3.7), lo cual puede ser ampliamente explotable a la vez que puede generar fatiga en las manos del jugador. La eliminación de la fase de decaimiento hace que la fase de sostenimiento tenga el valor máximo del parámetro (Swink, 2008).

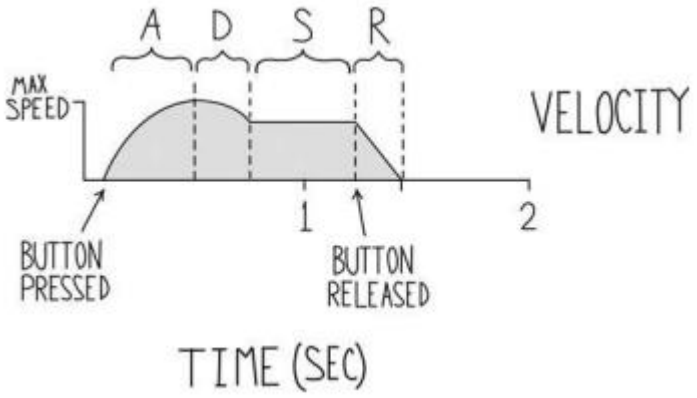

Figura 3.7. Modelo ADSR con fase de decaimiento. Fuente: Swink, 2008.

3.2. Cámara y control de personaje

La cámara es uno de los pilares fundamentales del videojuego. La cámara permite al jugador ver el espacio simulado y, sin ella, el bucle de interactividad se rompe debido a la desaparición de la visualización (punto 6 de la Fig. 3.1 en el subapartado Definiendo *Game Feel*) y, por ende, el control en tiempo real desaparece. Además, la cámara permite añadir efectos de pulido (sacudidas o cabeceos) que pueden ayudar a mejorar la experiencia del usuario y al *Game Feel* del videojuego.

3.2.1. Fundamentos básicos de una cámara virtual

La documentación de OpenGL, concretamente en el apartado *Camera* (s.f), expone que la cámara transforma las coordenadas de mundo en coordenadas de vista, las cuales son relativas a la posición y dirección de la cámara, por lo tanto, lo que la cámara ve está definido por su posición y rotación en el mundo. Poniendo la cámara principal que usa el motor de *Unity* como ejemplo (Fig. 3.8), se puede ver que el eje Z (azul) marca la dirección en la que mira la cámara, mientras que los ejes X e Y (rojo y verde, respectivamente) representan los ejes X e Y positivos del espacio de vista de cámara.

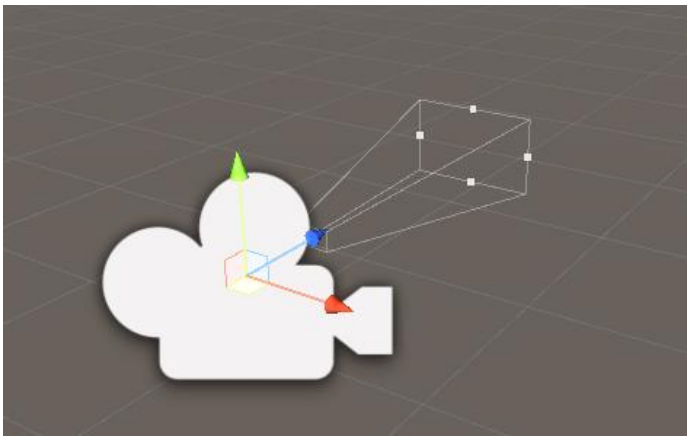

Figura 3.8. Visualización de los ejes de la cámara principal de Unity. Fuente: Unity.

A la hora de proyectar lo que ve una cámara se puede decidir si mostrarlo en proyección ortográfica o perspectiva (Fig. 3.9). Lakshminarayana (2019) explica que la proyección ortográfica es una forma de proyección paralela en la que las líneas de proyección son perpendiculares al plano de proyección, lo que hace que cada plano de la escena aparezca en transformación afín en la cámara. Por otro lado, la proyección en perspectiva da una impresión de profundidad, haciendo que los objetos más lejanos aparezcan más pequeños que los cercanos.

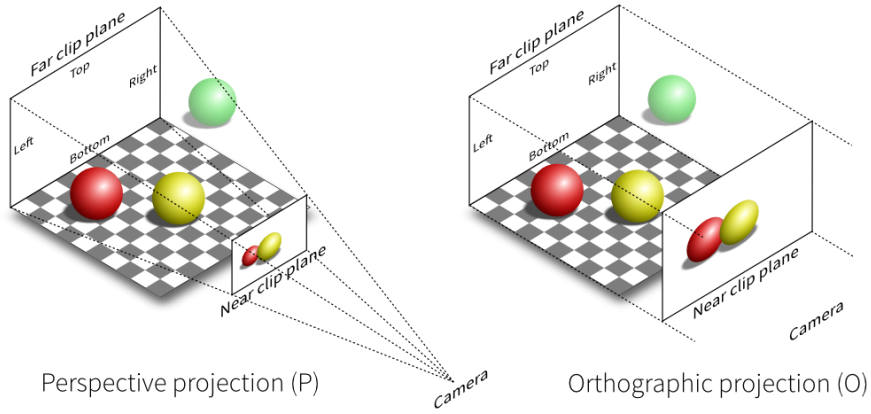

Figura 3.9. Representación de las proyecciones ortográfica y de perspectiva. Fuente: Lakshminarayana (2019)

Según Du, Liang, Xu, Wang y Gao (2012) una cámara virtual de OpenGL tiene los componentes que se ven en la Fig. 3.10.

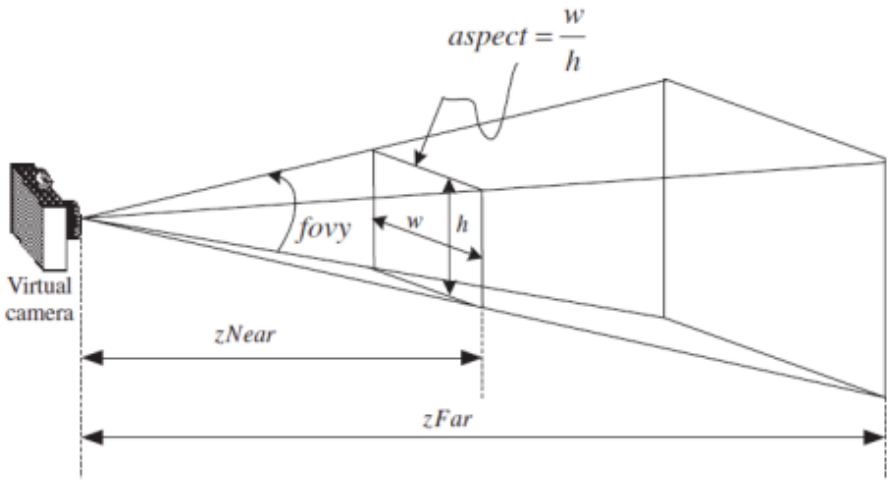

Figura 3.10. Componentes de una cámara virtual con proyección de perspectiva de OpenGL. Fuente: Du *et al,* (2012).

El *frustum* es el volumen que contiene todo lo que es visible por la cámara, el cual tiene forma de pirámide truncada en caso de que se use proyección de perspectiva y un prisma rectangular en caso de proyección ortográfica (Du *et al,* 2012).

El *fovy* especifica el ángulo del campo de visión en la dirección Y mientras que el llamado *aspect ratio* (anchura / altura) determina el campo de visión en la dirección X (Du *et al,* 2012).

Los llamados *Near* y *Far* son planos imaginarios que se encuentran a dos distancias de la cámara, siendo *zNear* la distancia entre la cámara y el *clipping plane* más cercano y el *zFar* la distancia entre la cámara y el *clipping plane* más lejano. Solo se ven en cámara los objetos que se sitúen entre estos dos planos (*Clipping Planes*, 2020).

En los videojuegos en primera persona la cámara se encuentra en una posición fija, por lo que el jugador nunca la mueve. Sin embargo, al situarse la cámara en los ojos del personaje es necesario que el jugador pueda rotarla para poder ver en detalle el espacio simulado. Para ello se cambia la dirección de la cámara según la entrada del jugador.

Los ángulos Euler son 3 valores que representan la rotación en tres dimensiones (Fig. 3.11). Estos son el *pitch, yaw* y *roll* (*Camera*, s.f).

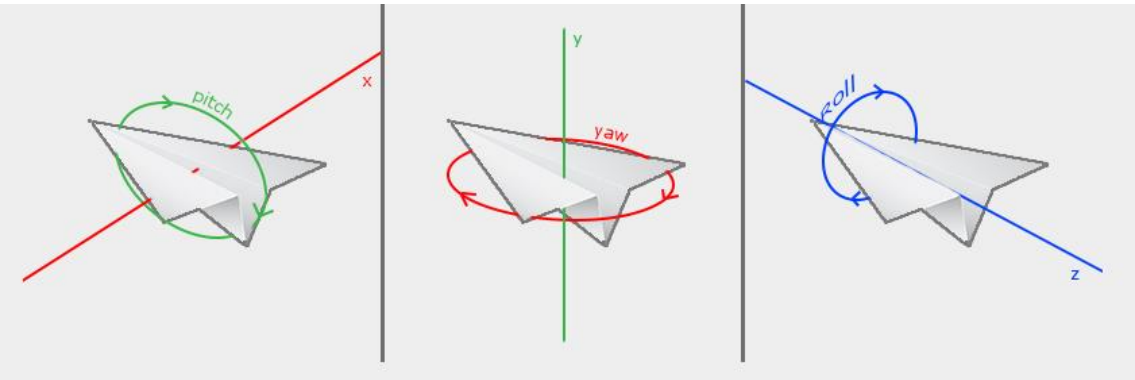

Figura 3.11. Representación gráfica de las rotaciones de los 3 ángulos Euler. Fuente: *Camera*, s.f.

El *pitch* (primera imagen de la Fig. 3.11) es el ángulo que determina si la cámara está mirando hacia arriba o abajo, el *yaw* (segunda imagen de la Fig. 3.11)

determina si la cámara está mirando hacia la izquierda o derecha y el *roll* (tercera imagen de la Fig. 3.11) representa la rotación en el eje Z de la cámara. En videojuegos el *roll* no se usa si no es en situaciones muy específicas, por lo que este se omite y solo se usan el *pitch* y el *yaw* (*Camera*, s.f).

Un *pitch* y *yaw* determinados se pueden convertir en un vector en tres dimensiones mediante las fórmulas de la Fig. 3.12. Este nuevo vector pasa a ser la nueva dirección de la cámara.

$$direction = new\ Vector3;$$

$$direction.x = \cos(yaw) * \cos(pitch);$$

$$direction.y = \sin(pitch);$$

$$direction.z = \sin(yaw) * \cos(pitch)$$

Figura 3.12. Fórmulas para la creación del nuevo vector dirección de la cámara. Fuente: (*Camera*, s.f).

3.2.2. Modelos de cámara

Un modelo de cámara es el punto de vista que tiene una cámara virtual de un videojuego a la hora de enseñar el espacio simulado, acompañado de las instrucciones sobre cómo la cámara debe moverse durante el tiempo de juego. La elección del modelo del cámara es parte del proceso de diseño (Adams, 2014).

Como explica Rogers (2014), el proceso de elección de cámara es uno de los más importantes a la hora de diseñar un videojuego junto al personaje y al control (estos tres componentes se consideran las tres Cs del diseño de juegos). También considera que es bastante común que un videojuego tenga más de un modelo de cámara, pero que es imprescindible que siempre haya uno principal y los demás sean secundarios y solo aparezcan en determinadas situaciones de juego.

3.2.2.1. Modelos estático y dinámico

Adams (2014) divide los modelos de cámara de forma general de la siguiente manera:

- Modelo de cámara estático: Modelo en el que la cámara enseña el espacio virtual desde una perspectiva fija. Aunque su uso era más frecuente en los videojuegos más tempranos, su uso sigue siendo común en la actualidad.
- Modelo de cámara dinámico: Modelo en el que la cámara se mueve reaccionando a las acciones del jugador. Su creación supuso un paso adelante en la industria del videojuego y su uso es mayoritario en la actualidad.

3.2.2.2. Primera Persona y Tercera Persona

A día de hoy la clasificación en géneros de los videojuegos es muy amplia, pero es frecuente separarlos según su vista de cámara. Las más usadas actualmente en videojuegos tridimensionales son la primera y tercera persona (Naftis, Tsatiris y Karpouzis, 2021).

La primera persona es la vista de cámara en la que la posición y orientación de la cámara corresponden a donde está el personaje controlado por el jugador y hacia donde está mirando, por lo que la cámara se posiciona en los ojos del personaje (Fig. 3.13) y el jugador gira la cabeza según su voluntad (Naftis, Tsatiris y Karpouzis, 2021). Según Adams (2014), por lo general no se suele ver el avatar del personaje, aunque en algunos casos se muestran sus manos.

Figura 3.13. Perspectiva en primera persona en el videojuego *Ghostrunner* de *One More Level*. Fuente: *Ghostrunnergame*, s.f.

La tercera persona es la vista que sitúa la cámara detrás del personaje (normalmente con una ligera inclinación hacia abajo para que el jugador pueda ver por dónde está caminando el personaje). Según Naftis, Tsatiris y Karpouzis (2021) esta es la vista que los diseñadores de videojuegos prefieren para videojuegos de acción, ya que esta muestra al personaje y parte de sus alrededores (Fig. 3.14). También dividen la tercera persona en tres subtipos: el primero usa diferentes cámaras esparcidas por el mundo que se activan y desactivan según la proximidad al jugador, el segundo usa una única cámara que sigue al personaje controlado por el jugador, y la última, basada en la segunda, ofrece al jugador la posibilidad de cambiar la orientación de la misma.

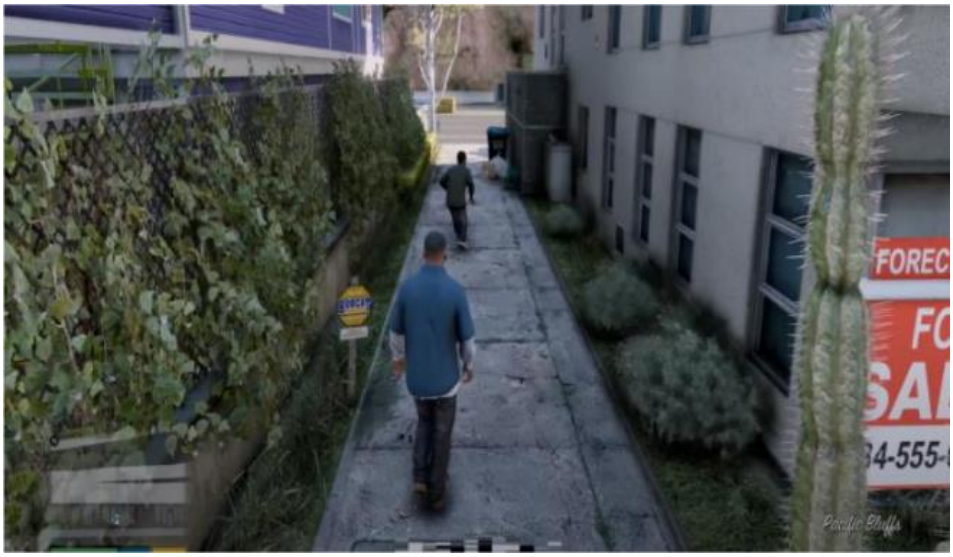

Figura 3.14. Perspectiva en tercera en el videojuego Grand Theft Auto 5 de Rockstar Games. Fuente: Naftis, Tsatiris y Karpouzis,

El uso de la primera persona tiene sus ventajas y desventajas. Algunas de ellas son las siguientes, expuestas por Adams (2014).

Ventajas:

- Al ser una vista de cámara en la que no se muestra el avatar, los costes de desarrollo se reducen al no tener que modelar un personaje principal entero.
- Al estar la cámara en completo control por el jugador no se requiere de ninguna IA que la controle.

- En caso de que haya mecánicas de disparo es más fácil apuntar debido a que el avatar no bloquea la visión y que la perspectiva del jugador es la misma que la del avatar.
- La primera persona permite interactuar fácilmente con el espacio simulado debido a la precisión que el modelo permite.

Desventajas:

- El uso de esta vista de cámara bloquea la posibilidad de usos de cámara cinemáticos.
- La visibilidad limitada que ofrece hace que ciertos tipos de movimientos gimnásticos sean más difíciles de ejecutar. Por ejemplo, es difícil calcular el momento exacto a la hora de saltar un vacío al no ver el suelo.
- Este modelo puede crear síntomas de cinetosis en jugadores que sean propensos a ello.

Por otro lado, Adams (2014) estipula que la tercera persona tiene la ventaja de mostrar al jugador la acción (en caso de que haya) de una manera más cinemática, viendo en todo momento al personaje, lo cual ofrece una gran cantidad de posibilidades a los diseñadores. Sin embargo, por este motivo y al encontrarse la cámara desvinculada físicamente del jugador, puede causar problemas de colisiones, por lo que se necesita un trabajo adicional de IA que otras vistas no presentan.

3.2.3. Control de personaje

El control de personaje es una de las 3Cs (al igual que la cámara) y es la más importante de todas (Rogers, 2014) ya que es la que permite la movilidad e interacción con el espacio simulado, además de ser uno de los tres bloques de la definición de *Game Feel* (ver subapartado Definiendo Game Feel). Al igual que la cámara, al control de personaje se le pueden añadir efectos de pulido (normalmente animaciones o efectos visuales) que ayuden a la experiencia de usuario y al *Game Feel* en general.

Adams (2014) define modelo de interacción como “la relación entre la entrada del jugador mediante el dispositivo de entrada y las acciones resultantes en el mundo de juego” (traducido del autor, p.259) y los divide en varios tipos:

- Basado en avatar: Las acciones del jugador consisten en controlar un solo personaje en el mundo de juego. El jugador actúa en el mundo mediante el avatar y puede influenciar las regiones del mundo en las que se ubica.
- Omnipresente: El jugador puede controlar diferentes partes del mundo a la vez.
- Basado en grupos: Es igual al que se basa en el avatar con la diferencia de que se controlan varios personajes a la vez.
- Concursante: El jugador responde preguntas y hace decisiones, como si se tratase de un concurso de televisión.
- Escritorio: Imita a un ordenador y el control de personaje se reduce a la interacción que este tiene con el ordenador.

Rogers (2014) enfatiza la importancia de un buen control mediante la ergonomía. La ergonomía, según la RAE (s.f.), es “el estudio de la adaptación de las máquinas, muebles y utensilios a la persona que los emplea habitualmente, para lograr una mayor comodidad y eficacia”. Rogers (2014) estipula que los controles se deben diseñar en función de la posición de la mano y recomienda hacer los controles accesibles para la mayoría de usuarios, ya sea añadiendo opciones de personalización o manteniéndose en los tipos de controles convencionales de cada género.

Según Rogers (2014) los controles de personaje se pueden dividir en dos tipos:

- Control relativo a la cámara: Los controles cambian según la dirección en la que el personaje mira a la cámara.
- Control relativo al personaje: El personaje siempre se mueve en la dirección que se ha recibido en el dispositivo de entrada, independientemente de donde esté mirando la cámara.

3.3. Creación de herramientas para diseñadores

Nicoll y Keogh (2019) definen un motor de videojuegos como “una herramienta de software que permite crear contenido digital interactivo en tiempo real, y un marco de código que permite que ese contenido se ejecute en diferentes plataformas”. (traducido del autor, p.9). Los motores de videojuegos normalmente se diseñan para gestionar tareas de bajo nivel como la renderización y el sistema de físicas, pero actualmente es habitual permitir su personalización a través de conjuntos de herramientas con el objetivo de facilitar el proceso de diseño (Nicoll y Keogh, 2019). Las herramientas (*tools* en inglés), permiten a desarrolladores crear nuevos entornos de complejidad variada (Juliani et al, 2018), además de reducir el tedio de tareas repetitivas que son comunes en el desarrollo de videojuegos (Tadres, 2015).

3.3.1. Unity

Unity es un motor de videojuegos lanzado por *Unity Technologies* en 2005. Principalmente se usa para el desarrollo de videojuegos (2D y 3D), pero también se usa para desarrollar aplicaciones o animaciones 3D gracias a su motor de renderización personalizado con el motor de físicas de *Nvidia PhysX* y Mono, la implementación *open source* de las librerías .NET de Microsoft (Craighead, Burke y Murphy, 2008).

Craighead, Burke y Murphy (2008) enumeran las ventajas que tiene Unity por encima de otros motores importantes en el mercado. Algunas son las siguientes:

- Documentación: *Unity* ofrece una documentación completa con ejemplos para su API entera.
- Comunidad de desarrolladores: Existe un foro donde se junta una comunidad de desarrolladores muy implicada en ayudar a posibles problemas que puedan tener los usuarios. Además de que *Unity Technologies* añade nuevas funcionalidades pedidas por los mismos usuarios.
- Editor de Unity: El editor de *Unity* es la interfaz gráfica de usuario que permite a los desarrolladores crear sus videojuegos. Este se compone de varias ventanas, siendo el navegador de proyecto, el inspector, la vista de juego, la

vista de escena y la jerarquía las más importantes (Haas, 2014). Aparte de las principales, Unity permite la creación de ventanas personalizadas. Unity tiene uno de los editores más fáciles de usar de la industria, ya que los objetos y hojas de código se pueden estructurar de forma rápida y sencilla con el sistema llamado *Drag-n-Drop* (Craighead, Burke y Murphy, 2008).

- Distribución multiplataforma: Todas las aplicaciones desarrolladas se pueden compilar en las plataformas más importantes y no hay ningún tipo de restricción al distribuirlas.

Aunque *Unity* tiene todos los componentes necesarios para hacer videojuegos, se puede dar el caso de que haya una funcionalidad que el proyecto necesite, pero no venga facilitada por el motor. En este caso, Unity provee una API de código que permite a los programadores desarrollar herramientas personalizables para cualquier tipo de funcionalidad (Tadres, 2015).

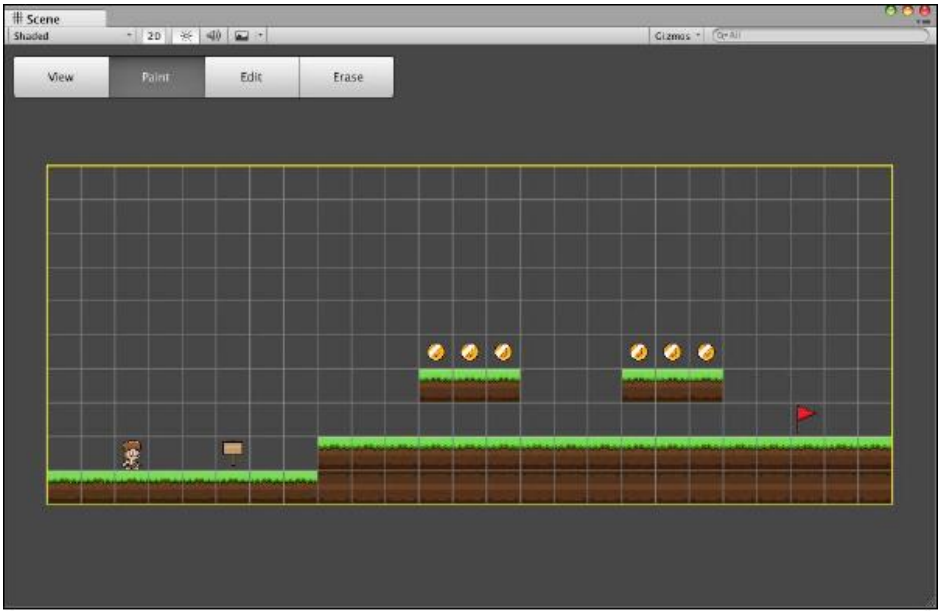

Figura 3.15. Ejemplo de herramienta de diseño de niveles para Unity. Fuente: Tadres, 2015.

3.3.2. *Scripting* de editor

A la hora de crear herramientas, *Unity* proporciona una API de *scripting* de editor que permite crearlas de manera integrada y rápida (Tadres, 2015). Un *script* de editor siempre tiene el *namespace UnityEditor* y sus funciones principales permiten

crear o modificar funcionalidades del editor predeterminado de *Unity*. En la Fig. 3.16 se puede ver un ejemplo de *script* de editor cuyo objetivo es el de crear un GameObject en la escena.

```
using UnityEngine;
using UnityEditor;

public class HelloWorld {

    [MenuItem ("GameObject/Create HelloWorld")]
    private static void CreateHelloWorldGameObject () {
        if(EditorUtility.DisplayDialog(
            "Hello World",
            "Do you really want to do this?",
            "Create",
            "Cancel")) {
            new GameObject("HelloWorld");
        }
    }
}
```

Figura 3.16. Script de editor diseñado para crear un GameObject en la escena. Fuente: Tadres, 2015.

En la misma Fig. 3.16 se pueden ver dos características que son únicas para este tipo de *scripts*:

- *MenuItem*: Atributo que permite añadir menús en el editor. En el caso del ejemplo se ha creado la opción de crear un *HelloWorld* en el menú de GameObject.

Figura 3.17. Representación visual de *MenuItem* en la interfaz de *Unity*. Fuente: Tadres, 2015.

- *DisplayDialog*: Método que permite crear una ventana emergente personalizada. En el caso del ejemplo esta ventana sirve para confirmar la creación del GameObject.

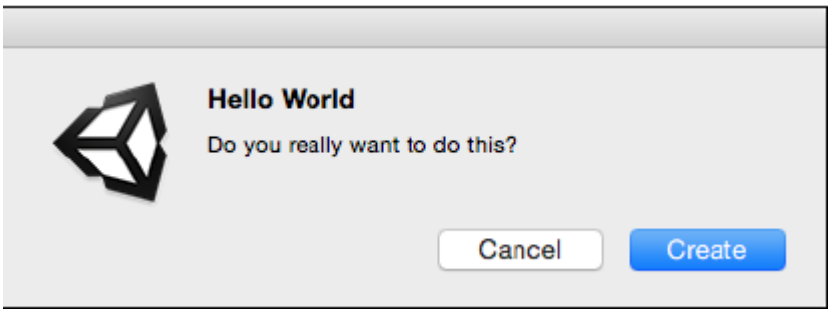

Figura 3.18. Representación visual de *DisplayDialog* en la interfaz de *Unity*. Fuente: Tadres, 2015.

Este tipo de *scripts* deben ir en una carpeta específica del proyecto llamada Editor para que *Unity* entienda que el script no forma parte del código fuente del juego. En caso de que no se quiera usar una carpeta, se pueden usar directrices de preprocesador como se ve en la Fig. 3.19. Todo código que se encuentre dentro de estas directrices no será usado en la *build* del videojuego y será leído solo en la fase de preprocesador (Tadres, 2015).

```
using UnityEngine;
#if UNITY_EDITOR
using UnityEditor;
#endif
public class HelloWorld {

    #if UNITY_EDITOR
    [MenuItem ("GameObject/Create HelloWorld")]
    private static void CreateHelloWorldGameObject () {
        if(EditorUtility.DisplayDialog(
            "Hello World",
            "Do you really want to do this?",
            "Create",
            "Cancel")) {
            new GameObject("HelloWorld");
        }
    }
    #endif

    // Add your video game code here
}
```

Figura 3.19. Ejemplo de *script* de editor con directrices de preprocesador (en negrita). Fuente: Tadres, 2015.

3.3.3. Inspectores personalizables

El inspector es la ventana de *Unity* que permite modificar y ver los atributos de cada *GameObject,* en este se pueden ver todos los componentes que están adjuntos en el mismo, ya sean colisionadores, hojas de código, pistas de sonido o físicas (Haas,

2014). *Unity* permite personalizar el inspector de las hojas de código para definir como se ven las propiedades y variables en el mismo (Tadres, 2015).

3.3.3.1. *CustomEditor*

Para poder personalizar inspectores es necesario crear una nueva clase que utilice el atributo *CustomEditor* del *namespace UnityEditor.* Este atributo necesita el tipo de la clase de la cual se quiere personalizar el inspector. En la Fig. 3.20 se puede ver el uso del mismo.

```
using UnityEngine;
using UnityEditor;

namespace RunAndJump.LevelCreator {
    [CustomEditor(typeof(Level))]
    public class LevelInspector : Editor {

    }
}
```

Figura 3.20. Uso del atributo *CustomEditor.* Fuente: Tadres, 2015.

Como se puede ver en la Fig. 3.20 es necesario heredar de la clase *Editor*, la cual permite usar los siguientes métodos de *Unity* (Tadres, 2015)*:*

- *On Enable*: Método que se llama cada vez que se selecciona el objeto con el inspector personalizado.
- *On Disable:* Método que se llama cuando se deja de ver el inspector personalizado.
- *On Destroy:* Método que se llama cuando el objeto con el inspector personalizado es eliminado de la escena.

La nueva clase necesita una variable que referencie al objeto del que se quiere personalizar el inspector para poder acceder a sus propiedades y manipularlas (Tadres, 2015).

3.3.3.2. Elementos de GUI

Para modificar la apariencia del inspector personalizado es necesario modificar la GUI (Interfaz Gráfica de Usuario), para ello se debe sobrescribir el método *OnInspectorGUI*, el cual permite, precisamente, cambiar esta apariencia (Tadres, 2015). Para ello se pueden utilizar las clases *EditorGUILayout*, *GUILayout y EditorUtility*, cuyos métodos permiten dibujar los elementos que se quieren poner en el editor. Algunos de los métodos que se pueden usar son los siguientes (Tadres, 2015):

- *LabelField:* Método que se usa para mostrar una etiqueta con texto, el cual puede tener una fuente personalizada (*EditorGUILayout*).
- *IntField / FloatField*: Método que se usa para mostrar una etiqueta con un número *float* o *int* (*EditorGUILayout*).
- *ObjectField:* Método que se usa para poder arrastrar objetos, ya sean de la escena o del proyecto (*EditorGUILayout*).
- *Button:* Método que permite dibujar un botón, el cual se puede usar para llamar a otros métodos (*GUILayout*).
- *DisplayDialog*: Método que permite crear ventanas emergentes (*EditorUtility*).

Un ejemplo de inspector personalizado se puede ver en la Fig. 3.21.

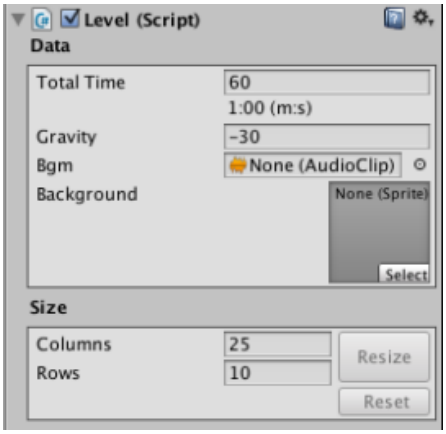

Figura 3.21. Ejemplo de inspector personalizado en el que se pueden ver ejemplos de *IntField*, *ObjectField* y *Button*. Fuente: Tadres (2015).

3.3.4. Ventanas personalizables

La mayoría de las interacciones que se tienen con *Unity* se hacen a través de ventanas, así que en caso de necesitar alguna herramienta que no se centre en objetos específicos, es más interesante usar ventanas personalizables, las cuales sirven como base para mostrar la GUI y contener las interacciones para poder realizar una funcionalidad específica (Tadres, 2015).

3.3.4.1. *EditorWindow*

La clase *EditorWindow* pertenece también al *namespace* de *Unity Editor* y permite crear ventanas personalizables (Tadres, 2015). Tal y como se ve en la Fig. 3.22, se requiere de una llamada al método *GetWindow* (Tadres, 2015).

```
public class PaletteWindow : EditorWindow {

  public static PaletteWindow instance;

  public static void ShowPalette () {
    instance = (PaletteWindow) EditorWindow.GetWindow
(typeof(PaletteWindow));
    instance.titleContent = new GUIContent("Palette");
  }
}
```

Figura 3.22. Ejemplo de creación de una ventana personalizable. Fuente: Tadres, 2015.

La clase *EditorWindow* dispone también de los mismos métodos que se pueden ver en el apartado anterior, Inspectores personalizables. Estos son el *OnEnable*, *OnDisable* y *OnDestroy* (Tadres, 2015).

3.3.4.2. Elementos de GUI

En el caso de las ventanas personalizables existe la opción de crear pestañas, aparte de todas las opciones que se han visto en el apartado anterior. Estas pestañas pueden servir para organizar contenido según su etiqueta o para agrupar funciones según su tipo (Tadres, 2015). El método que se usa para crear pestañas es el llamado *Toolbar* el cual recibe un *array* de tipo *string* con las etiquetas de todas

las pestañas que se quieren crear además de un *int* que indica que pestaña está seleccionada por defecto.

Un ejemplo de ventana personalizada se puede ver en la Fig. 3.23.

Figura 3.23. Ejemplo de ventana personalizada. Fuente: Tadres (2015).

4. Análisis de Referentes

En este capítulo se analiza el contenido de dos herramientas disponibles actualmente en la tienda de *assets* de *Unity*, cuya funcionalidad principal es la de crear controladores de personaje para videojuegos en primera persona.

4.1. *Modular First Person Controller*

El *Modular First Person Controller* (JeCase, 2021) es una herramienta creada por Jess Case lanzada en la tienda de *assets* de *Unity* el 8 de marzo de 2021 que permite la creación de un controlador en primera persona modular.

Al importar la herramienta se pueden encontrar tres archivos en su carpeta principal: un manual de instrucciones, una carpeta con el *script* donde se reúne todo el código y con un *prefab* con un personaje listo para ser usado, y, por último, una carpeta con una escena en la que se encuentra el mismo *prefab* acompañado de un escenario con el que interactuar para probar las funcionalidades que tiene el controlador.

El manual de instrucciones contiene toda información necesaria para entender el funcionamiento del controlador. La primera página es una introducción donde se explica el objetivo del controlador, se muestran los datos de contacto, el contenido del *pack* de importación de *Unity* y las instrucciones para el uso del mismo. Las siguientes páginas se usan para explicar el funcionamiento de cada una de las variables editables a través del inspector del *script* del controlador.

Al añadir el *prefab* en escena se crea un *GameObject* con el *script* del controlador y los componentes *Rigidbody, CharacterController* y *CapsuleCollider*.

El inspector del *script* de controlador es personalizado y está dividido en tres apartados (Fig. 4.1):

- Cámara: Contiene la cámara y todas las variables consideradas importantes para el controlador, como pueden ser el campo de visión, la sensibilidad o el ángulo máximo de rotación de la misma. Además, permite añadir como extra la capacidad de hacer *zoom* con un *toggle* que se puede activar en cualquier momento desde el editor.

- Movimiento: Contiene un conjunto de *toggles* que permiten activar y desactivar mecánicas y lo acompañan una serie de variables que permiten modificar su comportamiento.

- Cabeceo: Contiene la mecánica del cabeceo con sus propias variables para cambiar su comportamiento.

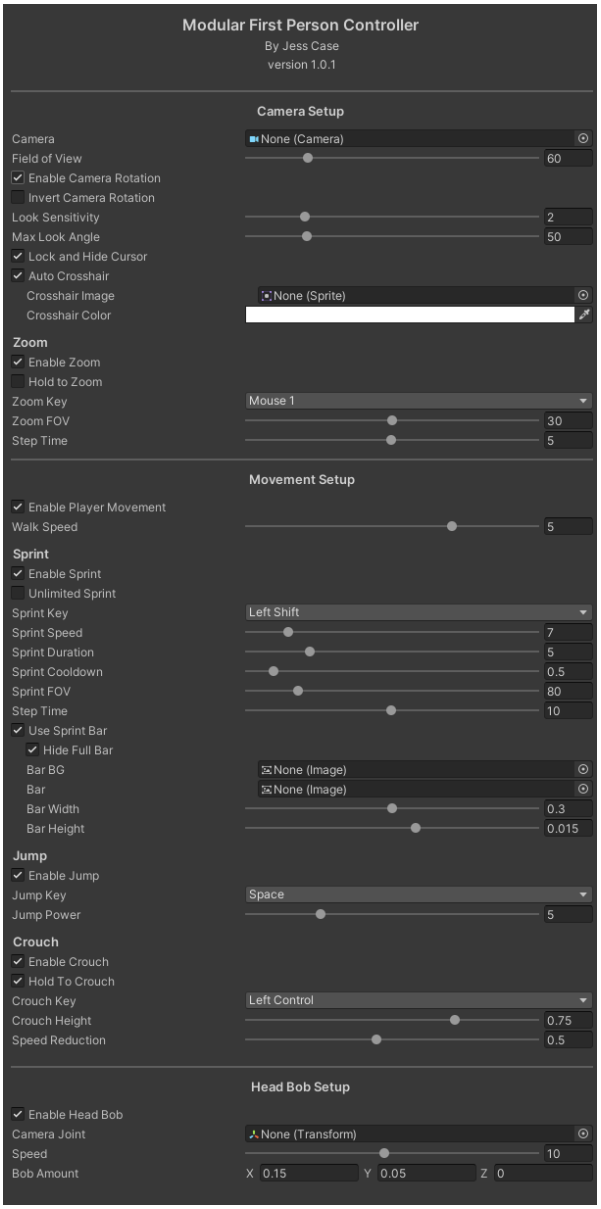

Figura 4.1. Captura del inspector del Modular First Person Controller.

Fuente: Elaboración propia.

Si bien todo funciona de forma correcta, se pueden observar problemas de legibilidad a la hora de querer cambiar una variable desde el inspector. Esto se ve en el apartado de movimiento (Fig. 4.1), lo cual puede ser un problema para los diseñadores a la hora de querer localizar y modificar variables de manera rápida.

A nivel interno, el código está estructurado en dos clases: una clase controlador y una clase que sobrescribe el inspector del editor de *Unity*. La clase que controla el inspector está formada principalmente por el método *OnInspectorGUI* y está debidamente estructurada por los apartados que lo forman gracias a la directriz de preprocesador #*region*, la cual puede agrupar fragmentos de código para facilitar la lectura. Las variables y principales métodos de *Unity* están recogidas también en esta directriz. A la hora de organizar el código de las mecánicas hay algunas que tienen su código distribuido por el *Update* y otras que lo tienen agrupado en métodos, lo cual puede causar confusión a otros programadores que quieran expandir el contenido de esta herramienta.

4.2. SUPER *Character Controller*

El *SUPER Character Controller* (Graves, 2019) es una herramienta creada por Aedan Graves lanzada en la tienda de *assets* de *Unity* el 8 de febrero de 2019 que permite la creación de controladores de personaje en primera y tercera persona con un amplio grupo de características personalizables.

Al importar el paquete de *Unity* se reciben dos carpetas: una contiene todos los archivos necesarios para ver una demo de lo que ofrece el controlador y la otra contiene *prefabs* de un personaje en primera y tercera persona, el manual de instrucciones y el *script* principal.

El manual de instrucciones contiene un total de siete secciones:

- Variables de inspector: Explicación detallada de todas las variables modificables en el inspector.
- Variables internas: Explicación detalla de todas las variables internas usadas en el *script* principal.
- Clases: Explicación de todas las clases y sus variables.

- Interfaces: Explicación de todas las interfaces creadas.
- Enums: Enumeración de todos los *enums* usados en el *script* principal.
- Funciones: Enumeración y explicación de todas las funciones que se encuentran en el *script* principal.

El usuario requiere saber de *Unity* para entender el manual y para preparar la herramienta en un personaje en la escena, por lo que puede ser complicado para usuarios que no estén acostumbrados al motor.

Para activar la herramienta se debe tener seleccionado un *GameObject* de la escena, acceder al menú *Component* (o directamente al *AddComponent*) y seleccionar el *SUPER Character Controller.*

El inspector personalizado del *SUPER Character Controller* está dividido en nueve secciones, cuatro de las cuales se pueden ver en la Fig. 4.2:

- Opciones de cámara: Permite activar el control de cámara, además de cambiar la perspectiva de la misma. En caso de que querer profundizar, el usuario puede acceder a las opciones avanzadas mediante el botón *show more*.
- Opciones de movimiento: Permite activar todas las mecánicas relacionadas con el movimiento que ofrece la herramienta, además de la opción de modificar los parámetros de sus variables.
- Aguante: Permite activar el sistema de aguante además de la opción de poder modificar sus variables.
- Cabeceo: Permite activar el cabeceo de cámara además de la opción de poder modificar sus variables.
- Estadísticas de supervivencia: Permite activar un conjunto de sistemas de vida junto con la opción de modificar sus variables.
- Opciones de los objetos interactivos: Permite asignar los parámetros de la mecánica de interactuar con los objetos, además de añadir qué objetos son interactivos.

- Configuración del *Animator:* Permite arrastrar el componente *animator* para su configuración.
- *Debuggers:* 4 botones que permiten activar *debugs* específicos del controlador.

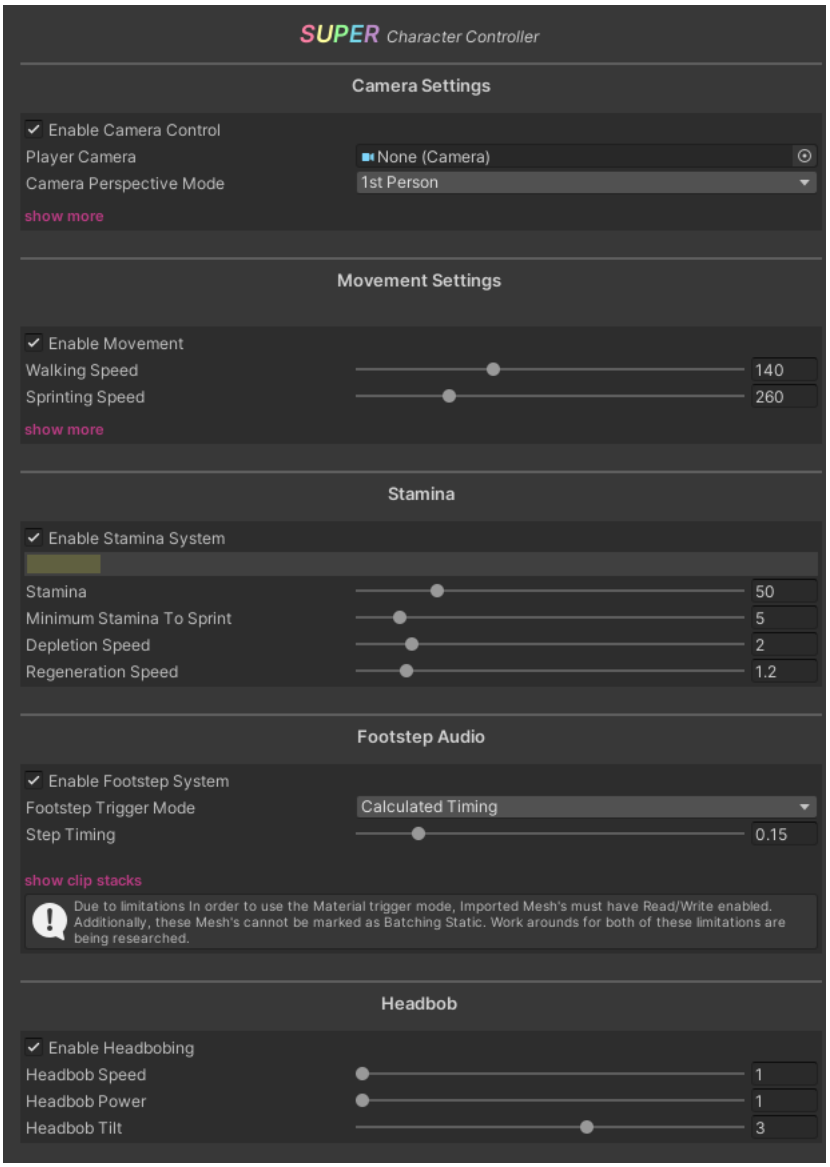

Figura 4.2. Captura del inspector del *SUPER Character Controller*. Fuente: Elaboración propia.

A nivel interno el código está compuesto por dos clases: la clase controlador y el inspector personalizado.

La clase controlador está separada por directrices de preprocesador *#region* que permiten organizar el código de manera leíble. En la primera región se encuentran todas las variables necesarias para el funcionamiento correcto de cada mecánica. La segunda región contiene los métodos propios de *Unity*: *Start, Update* y *OnTriggerEnter*. Por último, se encuentran todas las funciones necesarias para el funcionamiento adecuado del controlador.

El comienzo del inspector personalizado tiene un conjunto de variables creadas en el principio y un método *OnEnable* que permite configurar una serie de atributos. El resto de la clase es la sobreescritura del método *OnInspectorGUI*, el cual está dividido por directrices de preprocesador *#region*, que siguen la estructura de las secciones del inspector. Cada región usa métodos específicos de la clase *EditorGUILayout* y *GUILayout* para ofrecer representaciones visuales de las variables de todas las mecánicas en el inspector, mostrándose como *sliders*, *toggles*, textos y botones.

4.3. Tabla comparativa de referentes

La Tabla 1 muestra de manera resumida las características que tienen en común las dos herramientas.

Característica	***Modular First Person Controller***	***SUPER Character Controller***
Manual de instrucciones		
Demo		
Prefab		
Script único		

Uso de inspector personalizado		
Toggles de activación de mecánicas		
Separación visual entre mecánicas en el inspector		
Personalización de *Game Feel*		
Mecánicas no relacionadas con el control de personajes		
Uso de ventanas personalizadas		
Adición automática de componentes		
Uso de directrices de preprocesador para facilitar la lectura de código		

Tabla 1. Tabla comparativa de referentes. Fuente: Elaboración propia.

5. Diseño metodológico y cronograma

En este capítulo se explica la metodología de trabajo escogida para el desarrollo de la herramienta y como se ha usado para crear las diferentes fases de desarrollo por las que ha pasado el proyecto. Además, se detalla el contenido realizado en cada una y como se han repartido durante el tiempo de desarrollo de este trabajo.

5.1. Metodología de trabajo

Para este trabajo se usa la metodología (o modelo) en cascada (*waterfall* en inglés) creada por Winston W. Royce en 1970, la cual se ha utilizado ampliamente en el campo del desarrollo de software (McCornick, 2012). El modelo en cascada está diseñado para que el desarrollo se divida en diferentes fases (Fig. 5.1) y cada una de ellas permanece bloqueada hasta que no se haya completado la fase anterior, con el objetivo de que no haya solapamiento (Balaji y Murugaiyan, 2012).

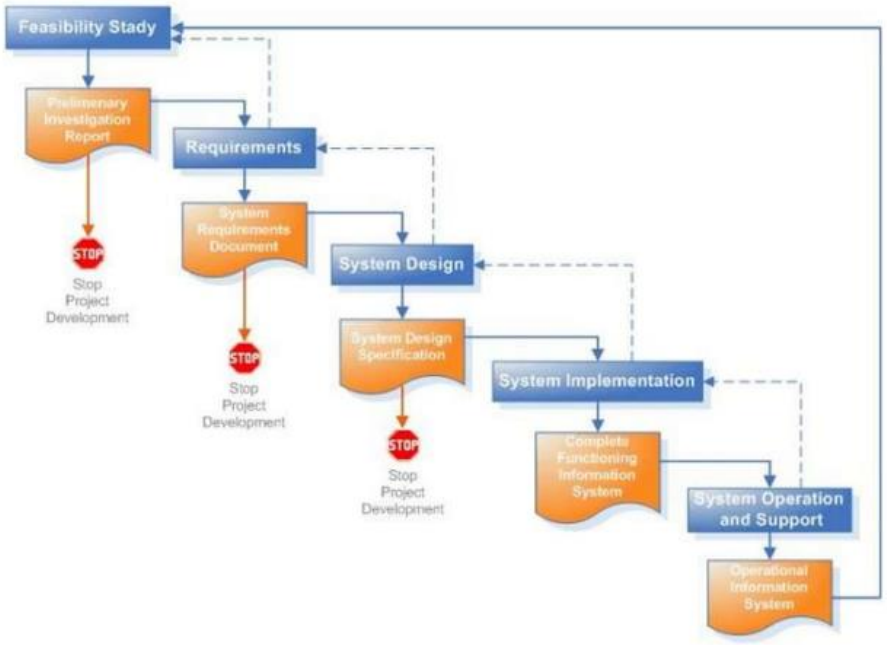

Figura 5.1. Ciclo del modelo en cascada. Fuente: McCornick, 2012.

5.2. Fases de Desarrollo

El desarrollo de la herramienta se ha dividido en cinco fases, cada una asociada a un objetivo secundario del trabajo. Siguiendo el modelo en cascada, cada una de

las fases ha estado bloqueada por la finalización de la fase anterior. De esta manera se ha podido realizar cada fase individualmente con la seguridad de que la anterior ha sido desarrollada al completo.

A continuación, se expone de forma resumida el contenido realizado en cada una de las cinco fases.

5.2.1. Fase 1: Requisitos de los controladores

En esta fase se han analizado las mecánicas relacionadas con el control de personaje de tres videojuegos, cada uno representando a un género diferente, siendo estos el *Walking Simulator*, el *First Person Shooter* y el *First Person Platformer*. A continuación, se han comparado las mecánicas analizadas para establecer cuáles son comunes y únicas en cada género con el fin de asignarlas a cada uno de los tres tipos de controlador desarrollados.

5.2.2. Fase 2: Creación de los controladores

En esta fase se ha diseñado un diagrama de clases en UML cuya estructura contiene los tres controladores de personaje, cada uno con sus mecánicas asignadas y analizadas en la fase anterior. Con la estructura de las clases organizada, se ha diseñado una estructura general de los *scripts* y se han estipulado los tipos de clase que cada uno tienen. Posteriormente, se ha diseñado la estructura para cada tipo de clase.

Una vez se ha realizado el diseño de *software*, se ha programado cada una de las mecánicas, creando así los tres controladores de personaje principales de la herramienta. A cada controlador se le ha creado un inspector personalizado, usando la clase *Editor* de *Unity*, que permite la edición de las variables de sus mecánicas en el editor, sin necesidad de acceso al código.

5.2.3. Fase 3: Creación de la interfaz gráfica

En esta fase se ha diseñado e implementado una ventana personalizada que dota a la herramienta de una interfaz gráfica que permite asignar los controladores de personaje, desarrollados en la fase anterior, a cualquier objeto de la escena de

Unity. Esta se ha programado con un *script* único, heredado de la clase *EditorWindow* de *Unity*, en el que se han implementado sus funcionalidades e interfaz. Por último, se han implementado mejoras de automatización que permiten un uso más rápido y efectivo de la herramienta.

5.2.4. Fase 4: Personalización de *Game Feel*

En esta fase se ha diseñado e implementado la posibilidad de modificar el *Game Feel* de ciertas mecánicas mediante la edición de gráficas ASR, las cuales permiten modular el valor de sus variables según el tiempo. Para añadir esta funcionalidad se han modificado los *scripts* creados en la fase 2: en los controladores de personaje se ha añadido la funcionalidad y en los inspectores personalizados se han añadido las gráficas para que se puedan modificar fácilmente desde el editor.

5.2.5. Fase 5: Creación de casos de uso

En esta fase el proceso de desarrollo de la herramienta ha finalizado y se han creado tres casos de uso de la misma, uno por cada controlador de personaje desarrollado. En cada caso se han expuesto todos los pasos a realizar para el correcto funcionamiento del controlador: la importación de *assets* y escenas, el uso de la ventana personalizada y la modificación de las mecánicas y parámetros en sus respectivos inspectores personalizados.

5.3. Cronograma

El cronograma recogido en la Tabla 2 representa de manera visual la duración de cada una de las fases de desarrollo del trabajo.

	Enero	Febrero	Marzo	Abril	Mayo	Junio
Marco Teórico						
Diseño Metodológico						
Anteproyecto						

Revisión de Anteproyecto						
Fase 1						
Fase 2						
Memoria Intermedia						
Análisis de Referentes						
Revisión de Memoria Intermedia						
Fase 3						
Fase 4						
Fase 5						
Conclusiones						
Memoria Final						

Tabla 2. Cronograma del trabajo. Fuente: Elaboración propia.

6. Resultados

En este capítulo se detallan los resultados obtenidos en cada una de las fases de desarrollo expuestas en el capítulo anterior. Este se compone de los siguientes apartados: Análisis de mecánicas, Creación de controladores, Creación de la interfaz gráfica, Personalización de *Game Feel* y Creación de casos de uso.

6.1. Análisis de mecánicas

Desde el principio del desarrollo se ha decidido que la herramienta permita a los diseñadores elegir un controlador por género, ya que, de esta manera, el controlador escogido por el usuario está limitado a las mecánicas comunes de su género.

En este apartado se analizan las mecánicas comunes y únicas de control de personaje de tres videojuegos que pertenecen a tres géneros distintos de videojuegos en primera persona. Los géneros analizados han sido el *Walking Simulator*, el *First Person Shooter* y el *First Person Platformer* debido a la importancia y popularidad que han conseguido en los últimos años.

6.1.1. *Walking Simulator*

Género de videojuegos enfocados principalmente en la narrativa cuyos principales objetivos suelen consistir en ir de un sitio a otro caminando de forma pausada, interactuando con el entorno o resolviendo puzles para avanzar en la historia. Este tipo de juegos están enfocados en el disfrute de la ambientación, narrativa y resolución de puzles. En este caso se analizan las mecánicas de control del *Walking Simulator Firewatch*.

6.1.1.1. *Firewatch*

Firewatch es un *Walking Simulator* desarrollado por el estudio Campo Santo cuya fecha de lanzamiento fue el 9 de febrero de 2016 (Campo Santo, 2016).

Figura 6.1. Captura de pantalla de *Firewatch* (Campo Santo, 2016). Fuente: Firewatchgame, 2022.

En la siguiente Tabla se encuentran descritas las mecánicas relacionadas con el control de personaje de *Firewatch*.

Mecánica	**Descripción**
Movimiento	Mecánica que permite al jugador moverse por el escenario con una velocidad lenta.
Correr	Mecánica que permite al jugador moverse con más rapidez de la habitual.
Cambio de altura / Salto	Esta mecánica permite al jugador interactuar con puntos específicos del escenario que se encuentran a una altura diferente de la normal y conseguir superarla mediante una animación automática.
Zoom	Esta mecánica permite al jugador hacer un zoom de la cámara en cualquier momento con el objetivo de ver más cerca objetos o lugares alejados.

Tabla 3. Tabla de mecánicas de *Firewatch*. Fuente: Elaboración propia.

6.1.2. *First Person Shooter*

Género de videojuegos centrado en la acción basado en el combate con armas de fuego cuyo objetivo suele ser el de limpiar zonas de enemigos sin morir. Suelen ir acompañados de una historia, pero normalmente esta pasa a un plano secundario, ya que este tipo de videojuegos están dirigidos a los jugadores que disfrutan de la acción y el frenetismo. En este caso se analizan las mecánicas de control del *First Person Shooter Doom Eternal*.

Es común dirigirse a este tipo de juegos con la abreviación FPS.

6.1.2.1. *Doom Eternal*

Doom Eternal es un *First Person Shooter* desarrollado por el estudio ID Software, cuya fecha de lanzamiento fue el 19 de marzo de 2020 (ID Software, 2020).

Figura 6.2. Captura de *Doom Eternal* (ID Software, 2020). Fuente: Slayersclub, s.f.

En la siguiente Tabla se encuentran descritas las mecánicas relacionadas con el control de personaje de *Doom Eternal*.

Mecánica	Descripción
Movimiento	Mecánica que permite al jugador moverse por el escenario con una velocidad moderada.

Salto	Esta mecánica permite al jugador hacer un salto y moverse verticalmente para acceder a sitios que se encuentran separados por un vacío o que se encuentran a diferente altura.
Dash	Mecánica que permite al jugador desplazarse horizontalmente de forma casi instantánea. Esta se puede usar para cubrir más distancia en los saltos o para esquivar ataques de enemigos. Esta habilidad tiene un número limitado de cargas, y cuando este llega a 0 se inicia un *cooldown* que la recarga.
Retroceso del arma	Esta mecánica da *feedback* visual al jugador de que ha disparado con un arma, mediante un movimiento de cámara.

Tabla 4. Tabla de mecánicas de DOOM Eternal. Fuente: Elaboración propia.

6.1.3. *First Person Platformer*

Género de videojuegos centrado en la acción y el movimiento basado en la superación de obstáculos del escenario para llegar al final del nivel. Normalmente están acompañados con desafíos de tiempo que se consiguen optimizando el uso de las mecánicas y movimiento. Este tipo de videojuegos están dirigidos a los jugadores que disfrutan del movimiento y de la sensación de control. En este caso se analizan las mecánicas de control del *First Person Platformer Ghostrunner*.

6.1.3.1. *Ghostrunner*

Ghostrunner es un *First Person Platformer* desarrollado por el estudio *One More Level* cuya fecha de lanzamiento fue el 27 de octubre de 2020 (*One More Level*, 2020).

Figura 6.3. Captura de pantalla de *Ghostrunner* (*One More Level*, 2020). Fuente: Ghostrunnergame, s.f.

En la siguiente Tabla se encuentran descritas las mecánicas relacionadas con el control de personaje de *Ghostrunner*.

Mecánica	**Descripción**
Movimiento	Mecánica que permite al jugador moverse por el escenario con una velocidad moderada.
Salto	Esta mecánica permite al jugador hacer un salto y moverse verticalmente para acceder a sitios que se encuentran separados por un vacío o que se encuentran a diferente altura.
Dash	Mecánica que permite al jugador desplazarse horizontalmente de forma casi instantánea. Esta se puede usar para cubrir más distancia en los saltos o para esquivar ataques de enemigos. Esta mecánica tiene un número limitado de cargas.
Deslizamiento / Agacharse	El jugador puede agacharse para pasar por sitios bajos que de otra manera no se podrían pasar. En caso de que el jugador tenga inercia, la acción de agacharse pasa a ser la de deslizarse y puede usarlo para ganar inercia e ir más deprisa.

Wall Run	El jugador puede saltar en una pared y correr por ella para avanzar por el nivel.
Gancho	El jugador puede transportarse de manera casi instantánea a un punto de gancho con el objetivo de cubrir grandes distancias o esquivar ataques de enemigos.

Tabla 5. Tabla de mecánicas de Ghostrunner. Fuente: Elaboración propia.

6.1.4. Tabla comparativa de mecánicas

La Tabla 6 muestra de manera resumida las mecánicas que tienen en común los tres géneros.

Mecánica	**Walking Simulator**	**FPS**	**FPP**
Movimiento			
Saltar			
Correr			
Dash			
Zoom			
Retroceso del arma			
Deslizamiento / Agacharse			
Wall Run			
Gancho			

Tabla 6. Tabla comparativa de mecánicas. Fuente: Elaboración propia.

6.2. Creación de controladores

En este apartado se muestra el proceso de construcción de los *scripts* que componen la herramienta. Primero se expone la estructura del código y la preparación de la escena de *Unity* para, a continuación, detallar la estructura general de los *scripts* y clases. Por último, se explica el desarrollo de cada una de las mecánicas de cada controlador.

Cada uno de los *scripts* creados se puede ver en el Anexo 1.

6.2.1. Diseño de software

La estructura del código puede verse en el diagrama de clases UML mostrado en la Fig. 6.4. Se ha decidido usar herencia debido a la existencia de mecánicas comunes entre los tres controladores, evitando así la repetición de código en cada controlador. Las clases que forman la estructura son las siguientes:

- *First Person Controller*: Contiene las mecánicas comunes de todos los controladores. Sus métodos son protegidos para que puedan ser heredados.
- *Walking Simulator Controller*: Controlador que contiene todas las mecánicas de un *Walking Simulator.* Hereda todas las mécanicas de *First Person Controller* y añade su mecánica única mediante un método privado.
- *Advanced First Person Controller*: Contiene las mecánicas comunes de los controladores de *FPS* y *FPP*. Hereda de *First Person Controller* para traspasar sus mecánicas y sus métodos son protegidos para que puedan ser heredados.
- *FPS Controller*: Controlador que contiene todas las mecánicas de un *First Person Shooter.* Hereda todas las mecánicas de *First Person Controller* y *Advanced First Person* Controller, y añade sus mecánicas únicas mediante métodos privados.
- *FPP Controller*: Controlador que contiene todas las mecánicas de un *First Person Platformer.* Hereda todas las mecánicas de *First Person Controller* y *Advanced First Person* Controller, y añade sus mecánicas únicas mediante métodos privados.

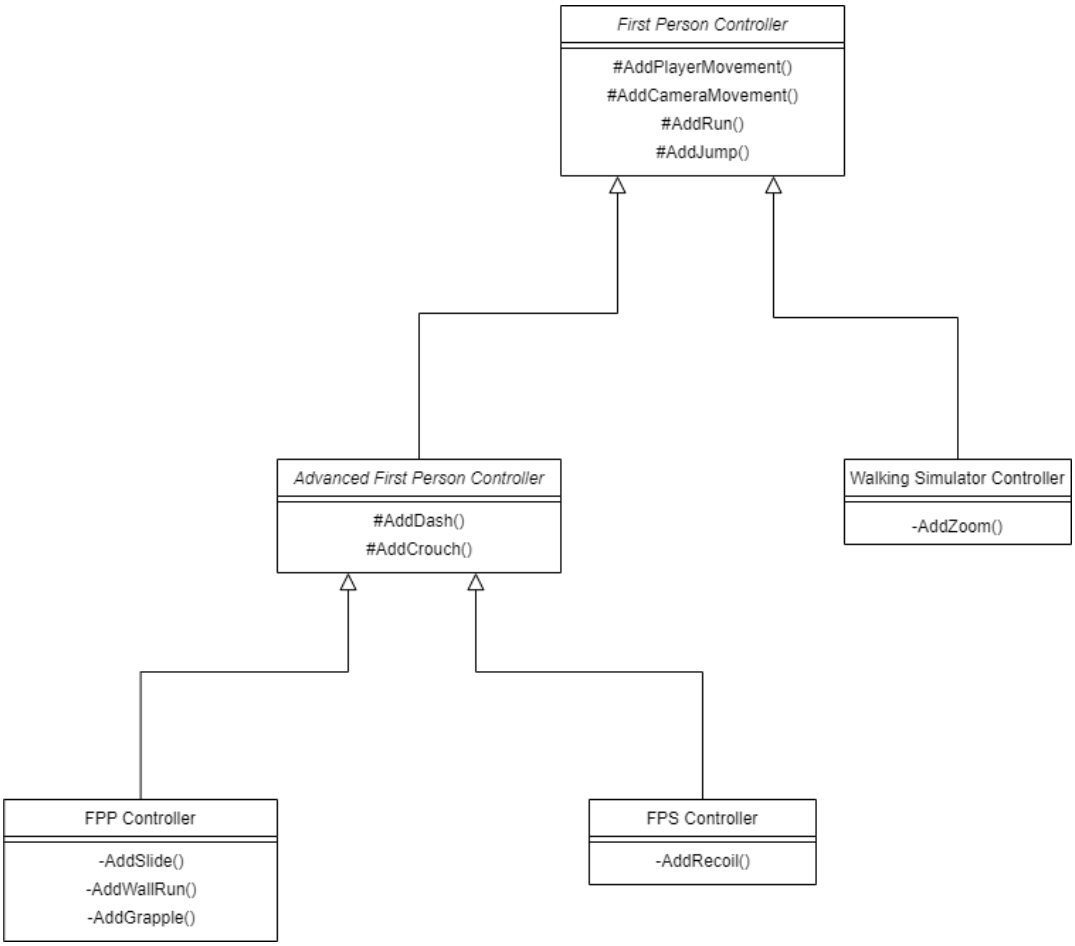

Figura 6.4. Diagrama de clases en formato UML. Fuente: Elaboración propia.

6.2.2. Preparación de la escena en *Unity*

Antes de empezar a desarrollar los *scripts,* se ha tenido que preparar una escena de *Unity* con los objetos necesarios para su funcionamiento. Primero de todo se han esparcido por la escena una serie de formas primitivas de *Unity* (planos y cubos), con la ayuda de la herramienta *ProBuilder* (*Unity Technologies*, s.f.) que han servido para crear un espacio simulado simple con el que el personaje puede interactuar. A continuación, se ha creado el personaje a controlar y se ha tomado la primera decisión de cara a su control, siendo esta la de elegir si el personaje es controlado por física usando un componente *Rigidbody* o por la clase *Character Controller*, diseñada específicamente para controles de personaje. Se ha usado la última no solo porque su funcionalidad es completamente afín a los objetivos de este proyecto, sino porque además es común la aparición de problemas al usar *Rigidbodies* controlados por el sistema de físicas del motor.

Por último, se ha asignado el objeto de cámara principal de la escena como hija del jugador debido a que el control del personaje y el control de cámara son completamente dependientes en videojuegos en primera persona.

En las Fig. 6.5 y 6.6 se puede ver la escena y la jerarquía de la misma, respectivamente.

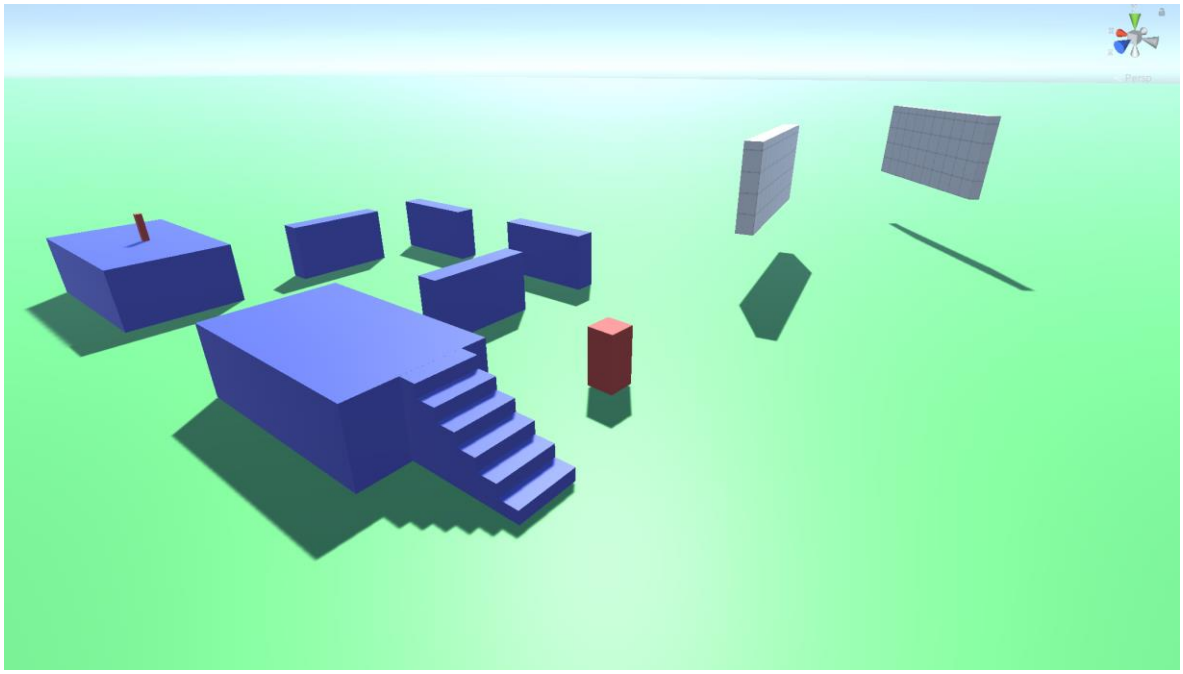

Figura 6.5. Captura de la escena de *Unity*. Fuente: Elaboración propia.

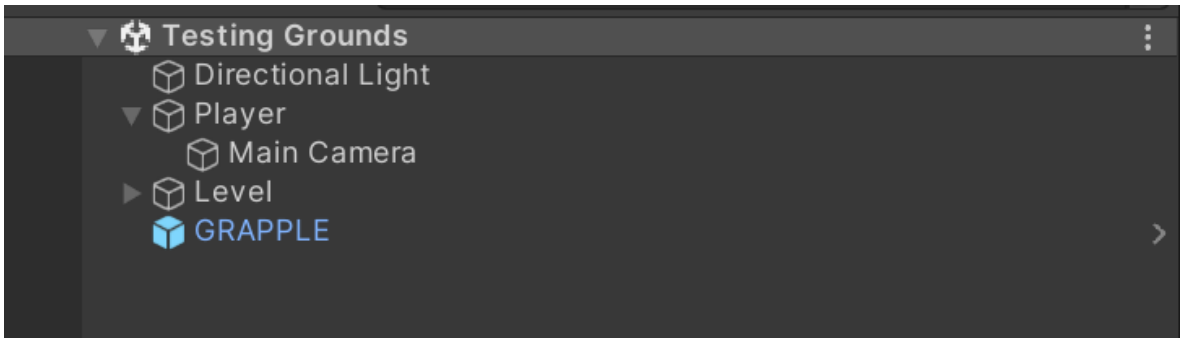

Figura 6.6. Captura de la jerarquía de la escena de *Unity*. Fuente: Elaboración propia.

6.2.3. Tipos de *scripts*

Se han creado tres tipos de *script* a lo largo del desarrollo:

- ***Script* de género:** Tipo de *script* que contiene el código necesario para crear un controlador del género de su nombre. Estos son el *Walking Simulator Controller*, el *FPS Controller* y el *FPP Controller.* Cada uno contiene las mecánicas de su propio género y nunca es padre de otro *script.*

- ***Script* auxiliar:** *Script* que contiene el código de mecánicas comunes entre géneros. Estos no son utilizados nunca como componente, ya que solo sirven para ser padres de los *scripts* de género. Estos son el *First Person Controller* y el *Advanced First Person Controller*.

- **Inspector personalizado:** *Script* que contiene el código necesario para personalizar la vista del inspector de *script* de género asignado. Estos son el *FPInspector*, *WSInspector*, *AFPInspector*, *FPSInspector* y *FPPInspector*.

6.2.4. Tipos de clase

Cada tipo de *script* tiene una clase, y hay dos tipos:

- **Clase Controlador:** En este tipo de clase se encuentran todas las variables y métodos necesarios para el buen funcionamiento de las mecánicas de cada controlador. Este tipo de clase solo se encuentra en los *scripts* género y auxiliares.

- **Inspector personalizado:** En esta clase se encuentran los métodos que permiten cambiar la interfaz del inspector de los *scripts* asignados, a través de las clases propias de *Unity*: *EditorGUILayout* y *Editor.* Esta clase recoge y muestra de forma personalizada en el inspector todas las variables públicas que los usuarios pueden modificar en cada controlador. Este tipo de clase solamente se encuentra en los *scripts* de inspector personalizado.

6.2.5. Estructura general de las clases

Después de crear los *scripts* se ha organizado internamente la clase que cada uno tiene. Todas las clases del mismo tipo siguen la misma estructura.

6.2.5.1. Clase Controlador

La clase controlador hereda de la clase padre que se ha estipulado en el diagrama de clases de la Fig. 6.4. Cada clase controlador sigue la siguiente estructura:

- **Activadores de mecánicas:** Al principio de la clase se crean los activadores de mecánicas. Estos son variables de tipo *bool* que permiten activar o desactivar las mecánicas de cada controlador de manera sencilla. Además, son traspasables a sus clases hijas para que las mecánicas puedan ser usadas por ellas.
- **Variables:** Listado de todas las variables necesarias para el funcionamiento adecuado de cada mecánica del controlador, también traspasables a sus clases hijas.
- ***Start y Update*:** los métodos propios de *Unity*. En el método *Start* se pueden aplicar las líneas de código necesarias a ejecutar antes del primer *Update*, y en el método *Update* se hace una comprobación constante del estado de los activadores de mecánicas, además de cualquier línea de código necesaria para el funcionamiento correcto del controlador. Estos métodos solo se encuentran en los *scripts* de género.
- **Métodos:** Funciones que contienen el código necesario para el buen funcionamiento de cada mecánica. Estas solo se llaman en caso de que sus activadores estén activos.

6.2.5.2. Inspector personalizado

Las clases de inspector personalizado siguen también la estructura por herencia del diagrama de clases de la Fig. 6.4. La estructura interna que siguen es la siguiente:

- **Variables:** Al principio de la clase se crean las variables que el inspector necesita. En este tipo de clases las variables usadas son del tipo *GUIStyle*, el cual permite personalizar diferentes estilos de texto a usar en el inspector.
- ***OnInspectorGUI()*:** Un *override* de este método permite reestablecer la UI del inspector y poder aplicar los cambios personalizados. Al principio del método se crea una variable del tipo de la clase controlador para poder

acceder a sus variables y representarlas visualmente en el inspector. A continuación, se llama a todos los métodos que contienen el código que cambia el apartado visual del inspector. Este método solo se encuentra en las clases de inspector personalizado asignadas a *scripts* de género.

- **Métodos:** Funciones que cambian el apartado visual del inspector. Dentro de estas se encuentran algunas de las funciones de la clase *EditorGUILayout*, como, por ejemplo: *BeginVertical* y *EndVertical* (permiten crear cuadros para distinguir entre las diferentes secciones del inspector), *LabelField*, *Toggle, FloatField, EnumPopup* y *Space* (permite crear un espacio vertical vacío). Estas funciones permiten vincularse a las variables de la clase controlador para que al ser modificadas en el inspector se modifique también en el código base de la clase controlador.

La clase *EditorGUILayout* no tiene una función para vincular variables de tipo *LayerMask*, por lo que ha sido necesario importar un *script* llamado *EditorTools* (*GamesDeveloper12*, 2015) que contiene una clase que permite usar la función *LayerMaskField*, la cual cubre la necesidad de poder vincular variables del tipo *LayerMask* en las clases de inspector personalizado.

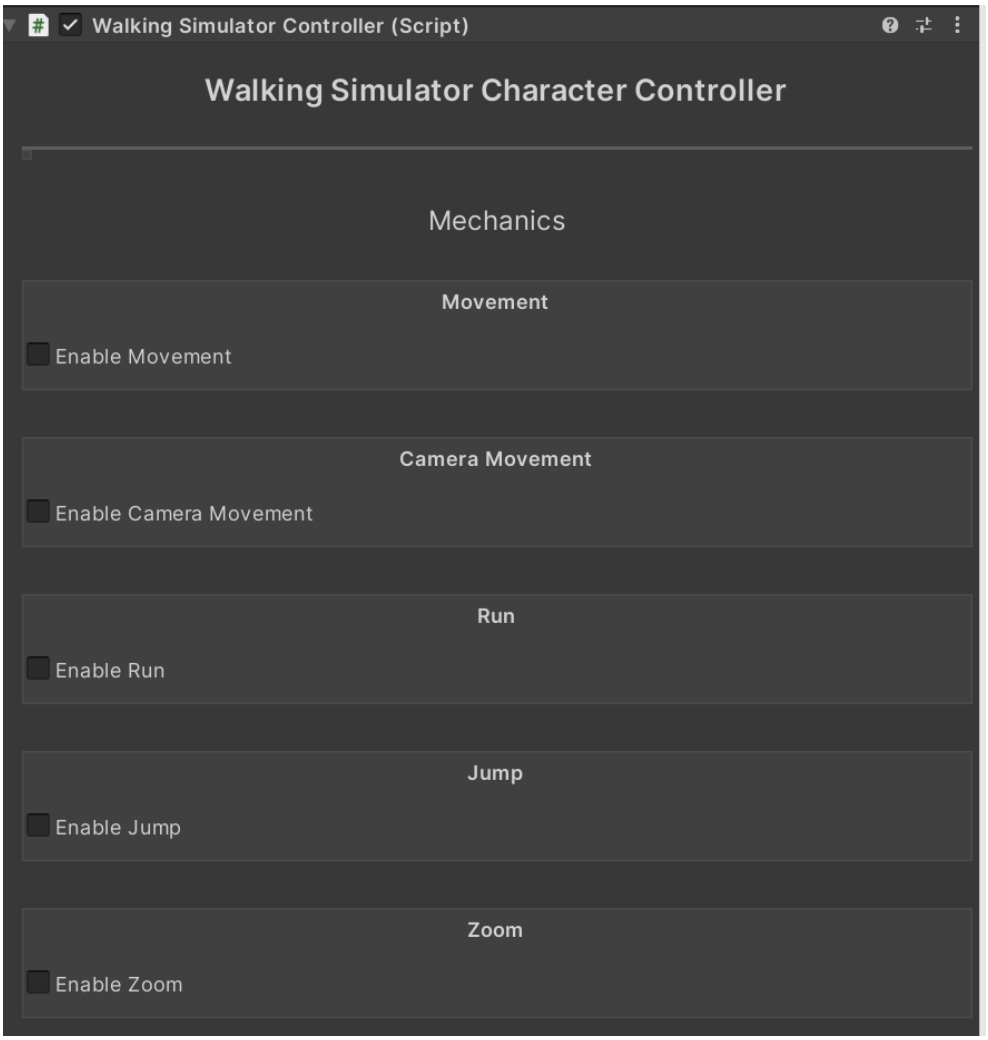

Figura 6.6. Inspector personalizado del controlador de *Walking Simulator*. Fuente: Elaboración propia.

6.2.6. Desarrollo de mecánicas

En este subapartado se explica el desarrollo de cada mecánica de cada una de las clases controlador.

6.2.6.1. *First Person Controller*

La clase controlador del *script First Person Controller* reúne las mecánicas de movimiento de la cámara y del personaje básicas en los videojuegos en primera persona, siendo estas las que se pueden ver en el diagrama de la Fig. 6.4.

Movimiento de personaje

Para el movimiento básico del personaje se han necesitado las siguientes variables:

- *walkSpeed*(float): Variable que contiene la velocidad por defecto del personaje.
- *movementSpeed*(float): Variable que se usa para asignar la velocidad del personaje en cualquier movimiento.
- *Gravity*(float): Variable que contiene el valor de la gravedad a aplicar al personaje.
- 4 variables de tipo *KeyCode* que permiten asignar el input de las 4 direcciones en las que se mueve el personaje en el eje horizontal (X y Z).

El método *AddPlayerMovement* gestiona todo el movimiento horizontal y vertical del personaje.

El movimiento horizontal depende del *input* que se reciba del jugador, cuando este se recibe se crea un *Vector3* con la dirección en la que mover al personaje, y este vector se pasa por parámetro a la función *Move* definida por el componente *CharacterController* que mueve al personaje según el vector que se le pasa por parámetro. Este vector se multiplica por la velocidad de movimiento guardada en la variable *movementSpeed* y por el *Time.deltaTime.*

El movimiento vertical se gestiona de forma interna mediante la misma función que el movimiento horizontal, con la diferencia de que el vertical es acelerado por la variable *gravity,* la cual se suma a la componente Y de un *Vector3* diferente al del movimiento horizontal*.* Sin embargo, este movimiento solo ocurre cuando el personaje no está tocando el suelo. Para saber si el personaje toca el suelo se ha creado un objeto llamado *GroundCheck* en la escena situado en la parte inferior del *CharacterController* que, a su vez, crea una esfera a su alrededor que detecta si hay colisión en su radio. En el caso de que haya colisión, el personaje pierde su movimiento vertical y se queda quieto.

Movimiento de cámara

Para el movimiento de cámara se han creado las siguientes variables:

- 2 variables de tipo *int* que guardan los valores de velocidad de movimiento de la cámara, una para el eje X y otra para el eje Y.

- 2 variables de tipo *int* que guardan los ángulos máximos de la cámara cuando esta mira hacia arriba o abajo.

El método *AddCameraMovement* gestiona el *pitch* y el *yaw* de la cámara.

El *yaw* tiene los valores de rotación del personaje en el eje Y, mientras que el *pitch* contiene el ángulo local de la rotación de la cámara, la cual está asignada como objeto público a través del inspector del *script*. A estos valores se les suma la velocidad de su respectivo eje (en el caso del *pitch* se resta) mediante las variables de velocidad de movimiento de la cámera y se multiplican por el *input* (recibido por el ratón a través de la función *Input.GetAxis*) y el *Time.deltaTime.*

En el caso del *pitch* es necesario hacer un *Clamp* entre los valores de las variables de los ángulos máximos para que el ángulo de la cámara no pase de esos valores.

Una vez se han calculado los valores de *pitch* y *yaw* estos se asignan a la rotación local de la cámara y a la rotación del personaje, respectivamente.

Carrera

Para que el personaje pueda correr se han creado 2 variables:

- *runSpeed*(float): Variable que contiene la velocidad de carrera del personaje.
- *runKey*(KeyCode): Variable que permite asignar el *input* de activación de la carrera.

El método *AddRun* gestiona cuando asignar la variable de *runSpeed* a la variable *movementSpeed* creada para guardar la velocidad del movimiento del personaje. Esta se asigna cuando se detecta la pulsación continua de la tecla asignada a la variable *runKey*.

Salto

Para el salto se han creado las siguientes variables:

- *jumpForce*(float): Variable que contiene la fuerza en la que salta el personaje.
- *jumpKey*(KeyCode): Variable que permite asignar el *input* de activación del salto.

El método *AddJump* contiene la gestión del salto, el cual comprueba la pulsación del botón de salto y confirma que el personaje está tocando el suelo. Si estas condiciones se cumplen, se le asigna al movimiento vertical en el eje Y el cálculo definido por la fórmula $vy = \sqrt{h * -2 * g}$ donde *h* es la variable *jumpForce* y *g* la variable *gravity*.

6.2.6.2. *Walking Simulator Controller*

La clase controlador del *script Walking Simulator* permite crear controladores para personajes de videojuegos del género. Esta clase hereda directamente de *First Person Controller* (Fig. 6.4) por lo que todos sus métodos y mecánicas pueden ser utilizados.

La única mecánica añadida a este controlador es la de hacer *zoom* con la cámara.

Zoom

Para el *zoom* se han creado las siguientes variables:

- *normalZoom*(float): Valor por defecto que corresponde al campo de visión por defecto de la cámara. Debe asignarse manualmente.
- *zoomValue*(float): Variable que contiene el *zoom* al que llega la cámara.
- *smooth*(float): Variable que contiene la suavidad a la que el *zoom* responde.
- *zoomKey*(KeyCode): Variable que permite asignar el *input* de activación del *zoom.*

El método *AddZoom* cambia el campo de visión (*fov*) de la cámara si se pulsa la tecla guardada en *zoomKey*. Para que este cambio ocurra de manera suave se hace un *lerp* desde el campo de visión actual, guardado en una variable auxiliar, al que se encuentra guardado en la variable *zoomValue* mediante la multiplicación del *smooth* por el *Time.deltaTime*. Cuando se deja de pulsar la tecla se hace la misma operación con la diferencia de que ahora el cambio se hace desde el campo de visión guardado en *zoomValue* hacia el que está guardado en la variable auxiliar.

6.2.6.3. *Advanced First Person Controller*

La clase controlador del *script Advanced First Person Controller* añade dos mecánicas nuevas a las que recibe por herencia de la clase controlador de *First Person Controller*. Estas dos mecánicas son el *dash* y la capacidad de agacharse.

Dash

Para realizar el *dash* se requiere de las siguientes variables:

- *dashSpeed*(float): Variable que contiene la velocidad del *dash*.
- *dashTime*(float): Variable que contiene el tiempo que dura el *dash*.
- *dashCooldown*(float): Variable que contiene el tiempo que ha de pasar hasta el siguiente *dash*.
- *dashKey*(KeyCode): Variable que permite asignar el *input* de activación del *dash.*

El método *AddDash()* detecta si el *input* de *dash* se ha activado y si el *cooldown* permite la activación se ejecuta una corutina que usa la función *Move* del *CharacterController* a la cual se le pasa por parámetro el movimiento horizontal del personaje multiplicado por la *dashSpeed* y el *Time.deltaTime*.

Agacharse

Para poder agacharse se necesitan las siguientes variables:

- *crouchSpeed*(float): Variable que guarda la velocidad de agacharse.
- *crouchHeight*(float): Variable que guarda la altura que tiene el *CharacterController* cuando el personaje está agachado.
- *standHeight*(float): Variable auxiliar que guarda la altura por defecto del *CharacterController*. Debe asignarse manualmente.
- *crouchKey* (KeyCode): Variable que permite asignar el *input* que permite agacharse.

El método *AddCrouch* controla si se pulsa la *crouchKey* y si el personaje se encuentra en el suelo, si esto se cumple, se realizan tres *Lerps*: uno para cambiar

la variable de altura del *CharacterController*, un segundo para cambiar el centro del *CharacterController* y un tercero para cambiar la posición Y de la cámara hacia abajo. La duración de los *Lerps* viene determinada por la *crouchSpeed*.

Cuando se deja de pulsar la tecla, las tres variables vuelven a sus valores originales.

6.2.6.4. *FPS Controller*

La clase controlador del *script First Person Controller* hereda todas las mecánicas del *Advanced First Person Controller* y añade una sola mecánica: el retroceso del arma.

Retroceso

Las variables que se necesitan para realizar el retroceso son las siguientes:

- 2 variables de tipo float que recogen la cantidad de retroceso en los ejes X e Y de la cámara.
- *shootKey*(KeyCode): Variable que asigna el *input* de disparo y permite activar el retroceso.

El método *AddRecoil* detecta si se ha pulsado la *shootKey* y crea dos variables de tipo *float* cuyos valores son calculados a partir de la fórmula $\frac{(Random.value - 0.5f)}{2} * recoilAmount$. Estos valores se restan al *pitch* (en valor absoluto) y al *yaw* de la cámara.

6.2.6.5. *FPP Controller*

La clase controlador del *script FPP Controller* hereda las mecánicas del *Advanced First Person Controller* y añade tres mecánicas: el deslizamiento, el *wallrun* y el gancho.

Deslizamiento

Las variables que se usan para el deslizamiento son las siguientes:

- *slideSpeed*(float): Variable que recoge la velocidad de deslizamiento.
- *maxSlideTime*(float): Variable que guarda el tiempo total de deslizamiento.
- *slideKey*(KeyCode): Variable que asigna el *input* de deslizamiento.

El método *AddSlide* comprueba que se presiona la tecla *slideKey*, si el movimiento horizontal es diferente de *Vector3.zero* y si está en el suelo. Si se cumplen las condiciones se replica el código de agacharse, además de añadir un *cooldown* que amplía la velocidad del personaje mientras el deslizamiento no pase del valor asignado en la variable *maxSlideTime*.

Cuando la tecla se deja de pulsar o cuando el tiempo de deslizamiento se acaba se restablecen todos los valore para volver al movimiento normal.

Wallrun

Las variables que se utilizan para el *wallrun* son las 2 siguientes:

- 1 variable de tipo *LayerMask* que guarda la capa en la que están todas las paredes en las que se puede aplicar esta mecánica.
- *wallCheckDistance*(float): Variable que recoge la distancia mínima en la que se detectan los objetos que tienen la capa de pared.

El método *AddWallRun* lanza dos *Raycast* en dirección a izquierda y derecha que detectan si el personaje se encuentra cerca de una pared. Si se detecta pared, el personaje no se encuentra en el suelo y el movimiento horizontal es mayor a cero, se produce el movimiento de *wallrun.* Antes de mover al personaje se desactiva la gravedad para que este no caiga y se calcula el vector dirección del movimiento a partir del producto vectorial entre la normal de la pared y el *transform.up* del personaje. Una vez se ha calculado este vector se usa la función *Move* en su dirección multiplicada por la velocidad del personaje.

Si el personaje ha recorrido toda la pared o se ha alejado de ella, la gravedad se restaura para volver a caer y volver al movimiento normal.

Gancho

Las variables que se usan para la mecánica del gancho son las 4 siguientes:

- *grappleSpeed*(float): Variable que guarda la velocidad a la que va el personaje mientras se engancha.
- *grappleDistance*(float): Variable que determina la distancia máxima a la que puede llegar el gancho.

- *grappleMask*(LayerMask): Variable que guarda la capa en la que están todas los objetos con los que puede interactuar el gancho.
- *grappleKey*(KeyCode): Variable que asigna el *input* del gancho.

El método *AddGrapple* lanza un *Raycast* desde la posición de la cámara mediante su vector *forward* con distancia *grappleDistance* para detectar objetos guardados en la *grappleMask* en los que poder engancharse. Si se detecta un objeto y se detecta el *input* de *grappleKey,* se guarda el *RaycastHit* en una variable *Vector3*. A continuación se crea un vector desde el personaje hasta el punto del *RaycastHit* que se convierte en el vector a pasar por parámetro en la función *Move*. Esta dirección se multiplica por la velocidad guardada en *grappleSpeed* y *Time.deltaTime* para el funcionamiento adecuado de la mecánica.

6.3. Creación de la interfaz gráfica

Después de la creación de los tres controladores de personaje, se ha desarrollado un nuevo *script* cuyo objetivo es el de crear una interfaz gráfica a través de una ventana que simplifique y facilite el proceso inicial de asignación de componentes al personaje al que se quiere añadir el controlador. Es a partir de este punto que la herramienta ha pasado a llamarse *First Person Controller Creator*.

6.3.1. Funcionamiento de la ventana

La ventana se puede acceder mediante el menú *Tools*, situado en la parte superior izquierda del editor de *Unity*. La ventana se llama *First Person Controller Creator*, igual que la herramienta, y al abrirla, el usuario debe añadir el *GameObject* que se usa como personaje controlable y elegir el tipo de controlador que le quiere aplicar. Cada tipo de controlador tiene enumeradas sus mecánicas disponibles.

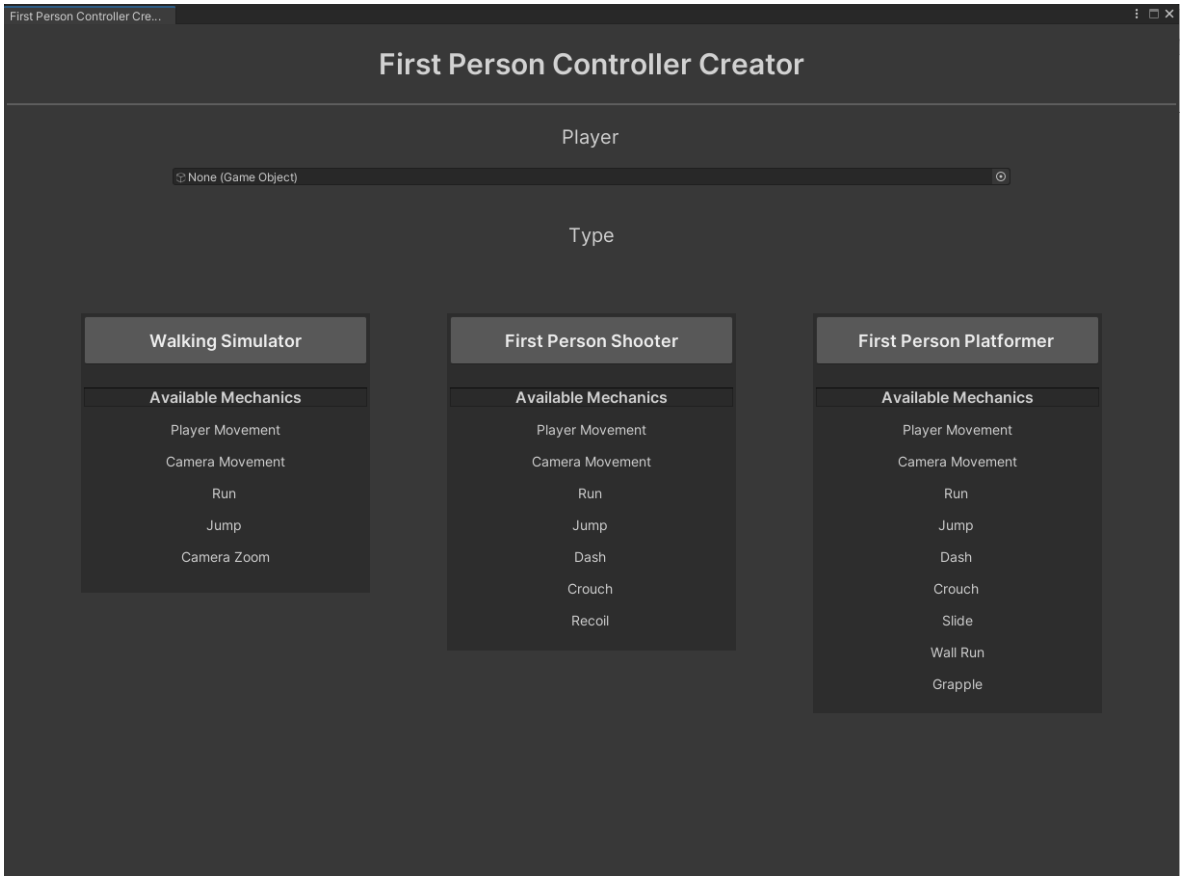

Figura 6.7. Ventana *First Person Controller Creator*. Fuente: Elaboración propia.

Cuando se añade el personaje y se selecciona el tipo de controlador que se quiere aplicar, aparece un botón de confirmación en la parte inferior de la ventana, que, al ser seleccionado, cierra la ventana y añade el *script* deseado como componente al *GameObject* del personaje, además de un *CharacterController*, si este aún no se ha asignado.

6.3.2. Desarrollo de la ventana

El contenido de la ventana se reúne en una clase llamada *CharacterControllerCreatorWindow*, la cual hereda de la clase *EditorWindow.* Al principio de la clase se reúnen todas las variables de tipo *GUIStyle*, para personalizar diferentes estilos de texto a usar en la ventana, además de las variables necesarias para el funcionamiento correcto de la herramienta. Para que esta sea accesible por el menú *Tools* ha sido necesario crear un *MenuItem,* que permite crear el mismo menú *Tools* y el botón de apertura de la herramienta.

El *script* tiene los siguientes cuatro métodos:

- *ShowWindow()*: Método estático en el que se llama a la función *GetWindow* para vincular el uso de la herramienta al *script* *CharacterControllerCreatorWindow* y crear la ventana.

- *OnGUI()*: Método en el que se edita el contenido visual de la herramienta que, a su vez, usa los siguientes métodos de la clase *EditorGUILayout* y *GUILayout*: *Label*, *Object Field*, *BeginHorizontal, EndHorizontal*, *BeginArea*, *EndArea, Button* y *Space*.

- *SetStyles*(): Método que inicializa todas las variables de tipo *GUIStyle*. Este se llama al principio del método *OnGUI()*.

- *AddScript():* Método que gestiona la adición de componentes en el *GameObject* del personaje mediante el botón de confirmación. Se deben cumplir dos condiciones para que aparezca este botón: el campo de *GameObject* de personaje debe estar lleno y el tipo de controlador debe haberse seleccionado. Cuando esto se cumple y se confirma la selección, se añaden como componentes al *GameObject* de personaje un *CharacterController* y un *script* del tipo de controlador que se ha escogido. Este método se llama al final del método *OnGUI()*.

6.3.3. Otras mejoras de la herramienta

Para facilitar el proceso inicial de la herramienta se han añadido dos mejoras en los inspectores personalizados de los *scripts* de género:

- Reposición de cámara: Cuando se añade un controlador como componente, este busca la cámara principal de la escena y la posiciona como hija del *GameObject* que actúa como personaje.

- Creación automática del *GroundCheck*: En caso de que el *GameObject* del personaje no tenga ningún *GroundCheck* como hijo en la jerarquía, se crea uno automáticamente y se sitúa en la parte inferior del *CharacterController*.

Estos procesos se realizan en el método *OnInspectorGUI()* de los inspectores personalizados de *scripts* de género.

6.4. Personalización de *Game Feel*

Una vez finalizada la ventana, se ha añadido la posibilidad de personalizar el *Game Feel* de mecánicas específicas disponibles en los controladores. Esto se ha hecho añadiendo la posibilidad de modificar las métricas de salida, a partir de la implementación de curvas ASR en el inspector que permiten modular los parámetros de las mecánicas en el tiempo.

6.4.1. Creación e implementación de curvas ASR

Para la creación de curvas editables, *Unity* ofrece las *AnimationCurve*, que consisten en ventanas con un sistema de coordenadas cartesiano cuyo eje Y representa el valor del parámetro a modificar y el eje X el tiempo. El usuario puede añadir *Keyframes* que sirven como puntos de inflexión de la curva. Para modificar el parámetro según los valores de la curva se usa la función *Evaluate(float time)* la cual devuelve el parámetro de la curva según el tiempo que se le pase por parámetro.

Para implementar las curvas ASR se han tomado dos decisiones, la primera ha sido la eliminación del decaimiento, ya que este no tiene un uso común en los videojuegos. La segunda decisión ha sido separar los demás estados de la curva (Ataque, Sostenimiento y Relajación) en tres curvas diferentes. Esto se ha realizado de esta manera debido a que *Unity* no permite la asignación de *presets* (o tipo de curva) a partes específicas de la curva, por lo que, si se asigna uno, este afecta a toda la curva. Si se crean tres curvas diferentes para cada estado, se le puede asignar un *preset* diferente a cada una sin que este afecte a los demás estados, lo cual permite al usuario asignar directamente un tipo de curva (línea recta, sigmoide, …) a cada uno de los estados por separado.

Al separar la curva en tres, ha sido necesaria la creación de una máquina de estados que controle el estado actual y las condiciones de cambio a cada una de las tres curvas.

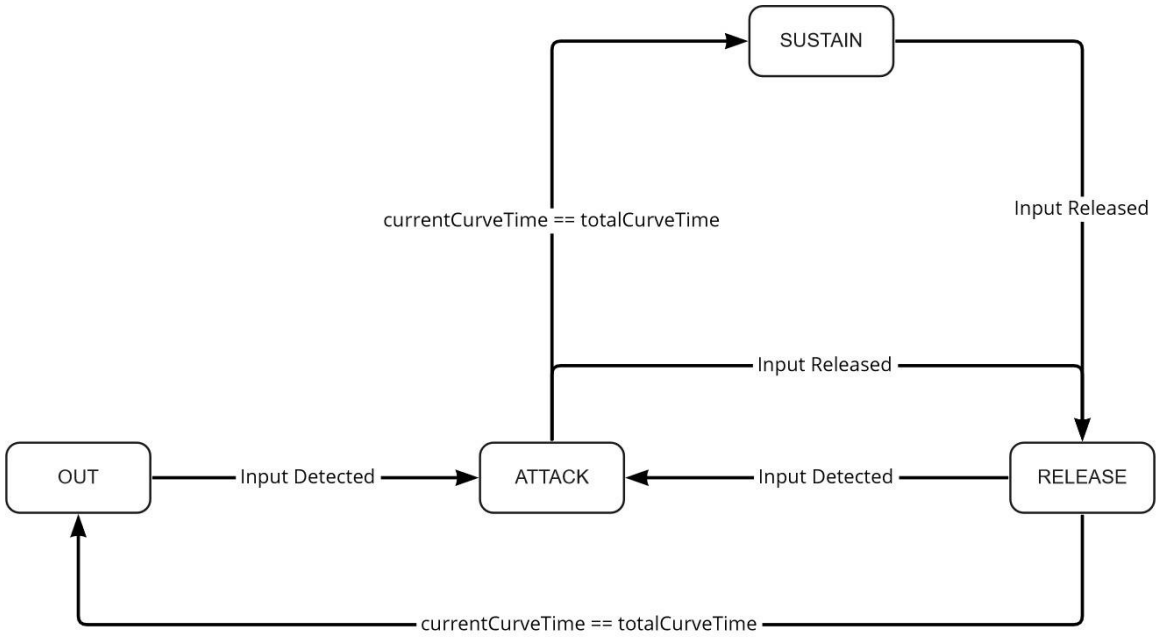

Figura 6.8. Representación gráfica de la máquina de estados usada la gestión de curvas ASR. Fuente: Elaboración propia.

Tal y como se ve en la Fig. 6.8 la máquina de estados se encuentra por defecto en el estado *OUT*, esperando a que se realice el *input* de la mecánica. Cuando esto ocurre se realiza un cambio de estado a *ATTACK* y el parámetro se modifica según la curva de ataque. Si esta se completa, se realiza un cambio de estado a *SUSTAIN* en el que se espera a que el jugador suelte el *input* para pasar al estado *RELEASE* (también se puede llegar al estado *RELEASE* si se suelta el *input* en el estado *ATTACK*). Mientras el estado *RELEASE* está activo, se reproduce la curva de relajación y, cuando esta se acaba, se vuelve al estado inicial, o a la curva de ataque en caso de volver a detectarse el *input*.

Este proceso se realiza por cada una de las mecánicas activas que tengan la posibilidad de habilitar las curvas ASR.

6.4.2. Modificaciones en la clase controlador

Para añadir el funcionamiento de las curvas ASR a las clases controlador primero se ha creado una función accesible por todos los *scripts* de género en la clase *FirstPersonController* llamada *ManageASR()* que recibe los siguientes parámetros:

- Párametro a modificar en el tiempo.
- *Bool* que devuelve el estado del *input* (pulsado, o no pulsado).
- Las tres *AnimationCurve*, ataque, sostenimiento y relajación.
- *Enum* que indica el estado actual de la máquina de estados (*Out, Attack, Sustain* y *Release*).
- *Float* que determina el tiempo transcurrido en la curva.

A partir de estos se ha hecho la máquina de estados mediante un *switch* según el estado actual que ejecuta las acciones y comprueba las condiciones para cada uno de los estados expuestos en el subapartado anterior.

Para cada clase controlador situada en un *script* de género se ha creado una variable por cada parámetro que necesita la función *ManageASR()*, además de un *bool* que determina si se usan las curvas ASR o la funcionalidad que viene por defecto. En caso de que este *bool* esté activo se llama a la función en el método que contiene el funcionamiento de la mecánica y, en el caso de que no lo esté, se activa el funcionamiento por defecto de la mecánica.

6.4.3. Modificaciones en los inspectores personalizados

Para que el usuario pueda modificar las curvas en el inspector se han tenido que ampliar los métodos de las clases de inspector personalizado. A estos métodos se les ha añadido un *Toggle* que determina si el *bool* que activa el ASR de la mecánica en la clase controlador está activo o no. En caso de que lo esté, se muestran las curvas de ataque, sostenimiento y relajación en el inspector con el método *CurveField* de la clase *EditorGUILayout*. Para determinar los valores mínimos y máximos de los parámetros, y los tiempos totales de cada curva se han creado tres funciones (*CreateAttack, CreateSustain, CreateRelease*) en las que se establecen los valores mínimos y máximos del parámetro en cada curva, y sus tiempos totales.

Estos valores están vinculados a las variables de la clase controlador, por lo que, si las variables se cambian en el inspector, también lo hacen en la curva. De esta manera, al editar la curva, el usuario solo puede cambiar su forma. Estas funciones se llaman en el método *OnInspectorGUI().*

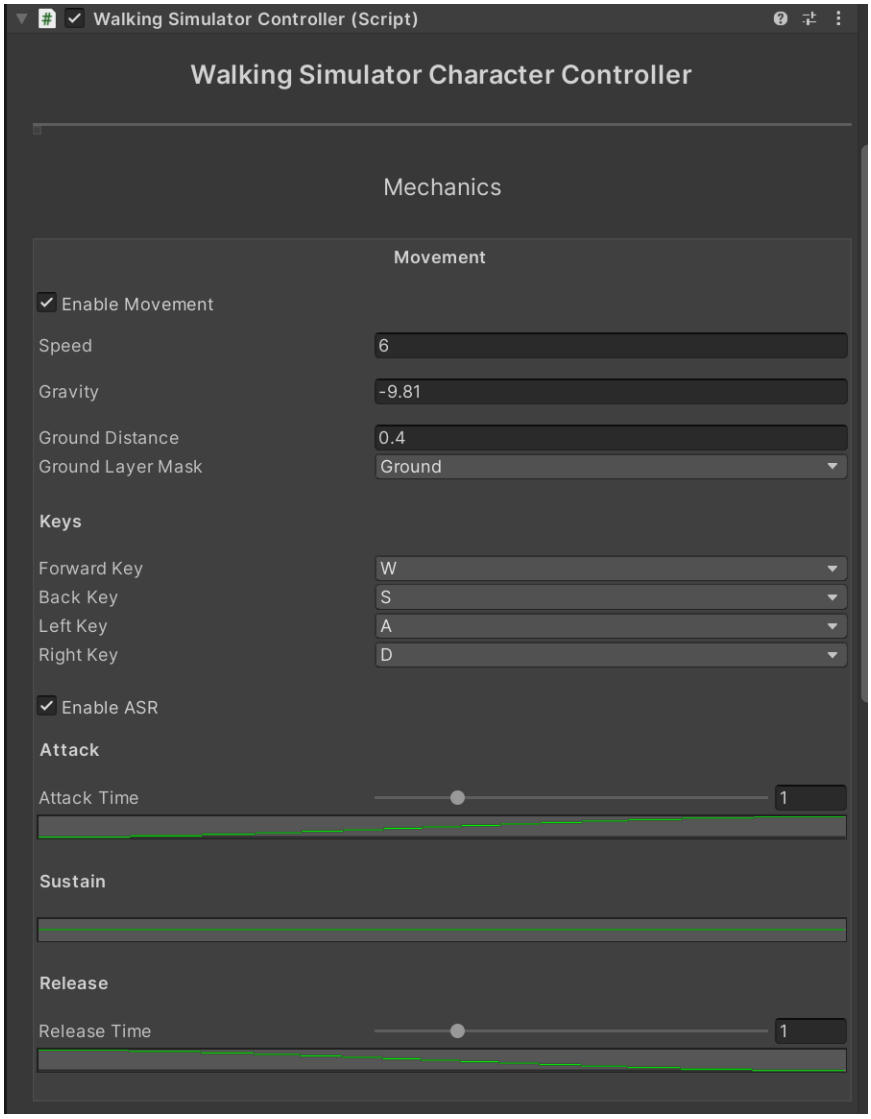

Figura 6.9. Fragmento del inspector de Walking Simulator con ASR implementado. Fuente: Elaboración propia.

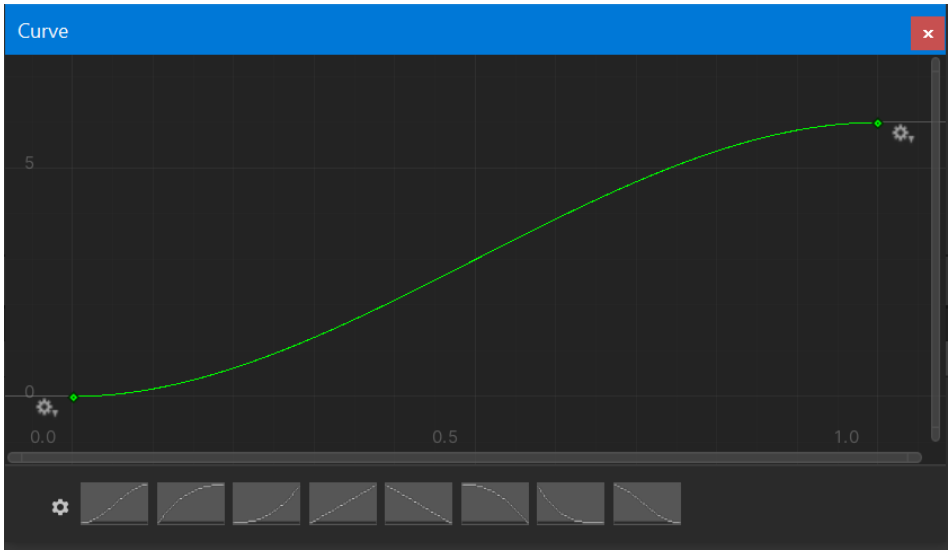

Figura 6.10. Ejemplo de ventana de una *AnimationCurve* de ataque.

Fuente: Elaboración propia.

6.5. Creación de casos de uso

Una vez finalizada la inclusión de curvas ASR, la herramienta se ha considerado acabada y, por último, se han creado un total de tres casos de uso para mostrar sus posibilidades y utilidad. Cada caso de uso muestra la creación de un controlador para uno de los géneros que permite la herramienta.

Estos se pueden probar en el ejecutable incluido en el Anexo 3.

6.5.1. Caso 1: *Walking Simulator*

Para crear este caso se ha usado una escena incorporada en los *assets* del *package* llamado *Apartment* Kit (Brick Project Studio, 2018). A continuación, se han creado el personaje a controlar y una cámara, y se han colocado en un punto específico de la escena.

Figura 6.11. Captura de la escena del package Apartment Kit. Fuente: Brick Project Studio, 2018.

Para asignar el controlador en el personaje se ha accedido al menú *Tools* y se ha abierto el *First Person Controller Creator* (Fig. 6.7). Se han asignado el *GameObject* del personaje y el género *Walking Simulator*, y por último se ha pulsado el botón de creación.

Cuando se ha abierto el inspector del *Walking Simulator Controller*, la cámara principal de la escena ha pasado a ser hija del personaje y, además, se ha creado un objeto vacío en la parte inferior del personaje que permite comprobar si este está tocando el suelo. Una vez está todo listo, se han activado todas las mecánicas del controlador con los *Toggles* de activación del inspector. Para cada una de estas se han asignado los siguientes valores:

- Movimiento:
    - Velocidad: Al tratarse de un espacio limitado se ha decidido poner una velocidad de personaje de 2 u/s.
    - Gravedad: Se ha aumentado la gravedad a -19.81 u/s^2 para evitar saltos muy elevados.
    - Suelo: La distancia para detectar suelo es de 0.5 unidades y se ha asignado la *LayerMask Ground,* donde se encuentran todos los objetos de la escena que se consideran suelo.

- *Input*: Se han asignado las teclas W, A, S, D como las cuatro teclas de dirección.
- ASR*:* Las gráficas ASR de la velocidad de movimiento se han adecuado para que el ataque sea orgánico y dure un segundo, mientras que la relajación es más lineal y dura la mitad de tiempo. De esta manera se consigue una sensación de peso al empezar a caminar y un detenimiento más rápido al dejar de caminar.

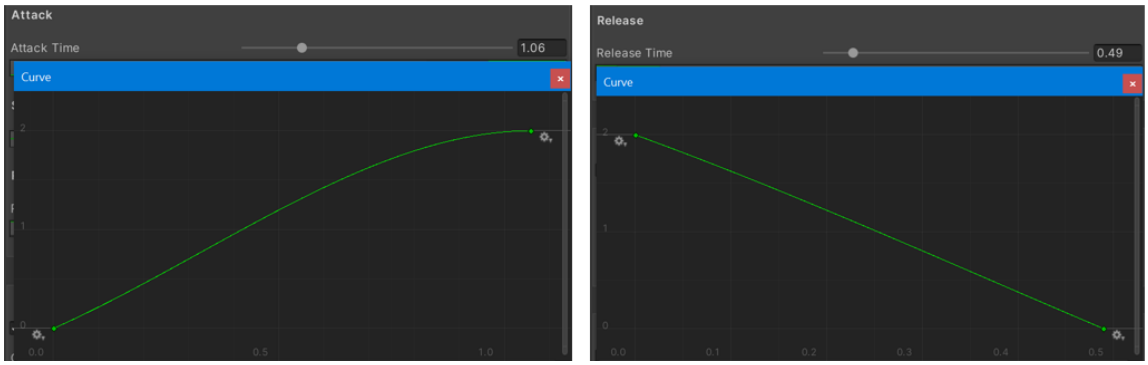

Figura 6.12. Ataque y Relajación de la velocidad de movimiento en el caso 1. Fuente: Elaboración propia.

- Movimiento de cámara:
  - Velocidad de cámara: Se ha optado por una velocidad de cámara reducida a 300 debido al ritmo pausado que caracteriza los *Walking Simulator.*
  - Ángulo de cámara: El valor mínimo y máximo para el ángulo de cámara es de 80º.
- Carrera:
  - Velocidad: Se ha doblado la velocidad normal del personaje, en este caso 4 u/s.
  - *Input:* Se ha asignado la tecla *Left Shift* por estándares de industria.
- Salto:
  - Fuerza de salto: Se ha decidido poner una fuerza de salto de 1 unidad debido a las limitaciones del escenario.

- *Input*: Se ha asignado la tecla espacio por estándares de industria.

- *Zoom*:
  - *Zoom* por defecto: Se ha vinculado manualmente el valor por defecto del campo de visión de la cámara, en este caso 90.
  - *Zoom*: Se ha asignado una cantidad de *zoom* de 20.
  - Suavidad: En caso de no usar ASR se ha asignado una suavidad de 5.
  - ASR: Las gráficas ASR de *zoom* se han adecuado para que el ataque y la relajación tengan forma sigmoidal, aportando suavidad, y tengan una diferencia de medio segundo entre ellos, siendo más duradero el ataque que la relajación.

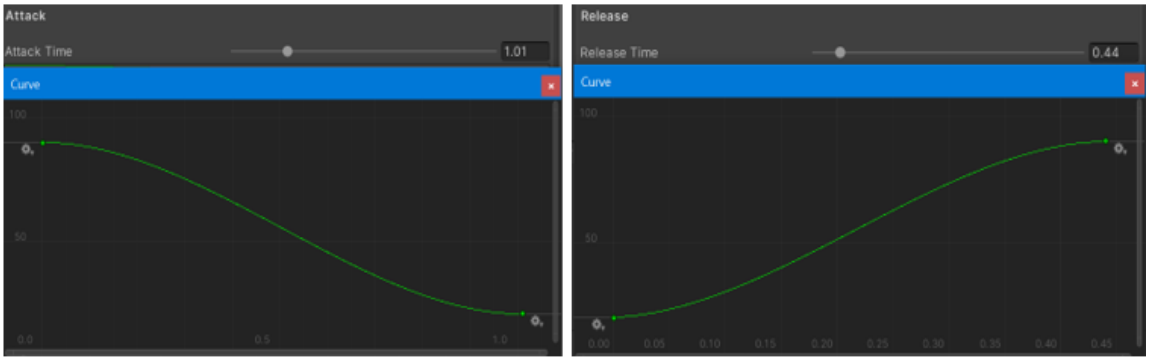

Figura 6.13. Ataque y Relajación del zoom en el caso 1.

Fuente: Elaboración Propia.

Con estos valores se ha conseguido un control de personaje lento y preciso, propio de los videojuegos de género *Walking Simulator*. El movimiento tiene la velocidad adecuada para moverse por el escenario, en el movimiento y la carrera. Además, el salto tiene la fuerza justa como para no chocar con el techo o para subirse a sitios donde no se debe.

6.5.2. Caso 2: *First Person Shooter*

Para crear este caso se ha usado una escena incorporada en los *assets* del *package* llamado *RPG/FPS Game Assets for PC/Mobile (Industrial Set v2.0)* (Kutsenko, 2017).

Figura 6.14. Captura de la escena del package RPG/FPS Game Assets for PC/Mobile (Industrial Set v2.0). Fuente: Kutsenko, 2021.

El proceso inicial para crear este caso es idéntico al del caso 1 con la excepción de seleccionar el género *First Person Shooter* (Fig. 6.6) antes de crear el controlador, y la de meter un objeto que sirve de arma como hijo de la cámara para que estas se muevan a la vez. En este caso las mecánicas tienen los siguientes valores:

- Movimiento:
    - Velocidad: Al tratarse de un espacio extenso se ha decidido poner una velocidad de personaje de 6 u/s.
    - Gravedad: Se ha aumentado la gravedad a -29 u/s^2 para evitar saltos muy elevados y para tener una sensación de gravedad verosímil.
    - Suelo: La distancia para detectar suelo es de 0.5 unidades y se ha asignado la *LayerMask Ground,* donde se encuentran todos los objetos de la escena que se consideran suelo.

- *Input*: Se han asignado las teclas W, A, S, D como las cuatro teclas de dirección.
- ASR*:* Las gráficas ASR de velocidad de movimiento se han adecuado para que el ataque y la relajación sean curvas y orgánicas, aportando suavidad. Sin embargo, los tiempos en este caso se han hecho más lentos para aportar sensación de peso en el ataque (1.5 s) y sensación de mantenimiento de inercia en la relajación (1 s).

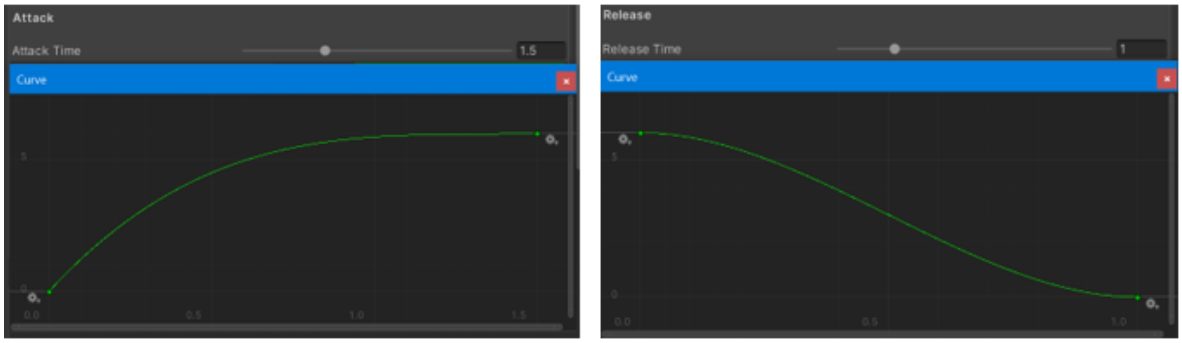

Figura 6.15. Ataque y Relajación de la velocidad de movimiento en el caso 2. Fuente: Elaboración propia.

- Movimiento de cámara:
  - Velocidad de cámara: Se ha optado por una velocidad de cámara de 500 debido al ritmo más rápido que caracteriza los *First Person Shooter*.
  - Ángulo de cámara: El valor mínimo y máximo para el ángulo de cámara es de 80º.
- Salto:
  - Fuerza de salto: Se ha decidido poner una fuerza de salto de 2 unidades debido a las limitaciones del escenario.
  - *Input*: Se ha asignado la tecla espacio por estándares de industria.
- *Dash*:
  - Velocidad: Se ha asignado un valor de 19 u/s a la velocidad debido a la necesidad de moverse a una velocidad elevada en poco tiempo.

- Tiempo: Se ha asignado un valor pequeño de 0.3 segundos para compensar la velocidad elevada.
- *Cooldown:* Se ha puesto un *cooldown* de 1 segundo para evitar la posibilidad de hacerlo continuamente.
- *Input:* Se ha asignado la tecla *Left Shift* por estándares de industria.

- Agacharse:
  - Altura normal: Se ha vinculado manualmente el valor por defecto de la altura del personaje, en este caso 2.
  - Altura mientras el personaje está agachado: Se ha asignado un valor de 0.5 unidades de altura para que se note claramente la diferencia de altura.
  - Velocidad: En caso de no usar ASR se ha asignado una velocidad de agachado de 10.
  - *Input*: Se ha asignado la tecla *Left Control* por estándares de industria.
  - ASR: Las gráficas ASR de agacharse se han adecuado para que el ataque y la relajación tengan forma sigmoidal, aportando suavidad, y tengan una diferencia de un cuarto de segundo entre ellos, 0.5 y 0.25 segundos, respectivamente. Estos valores son pequeños debido a la inmediatez de recuperación que se necesita en juegos frenéticos como un *First Person Shooter*.

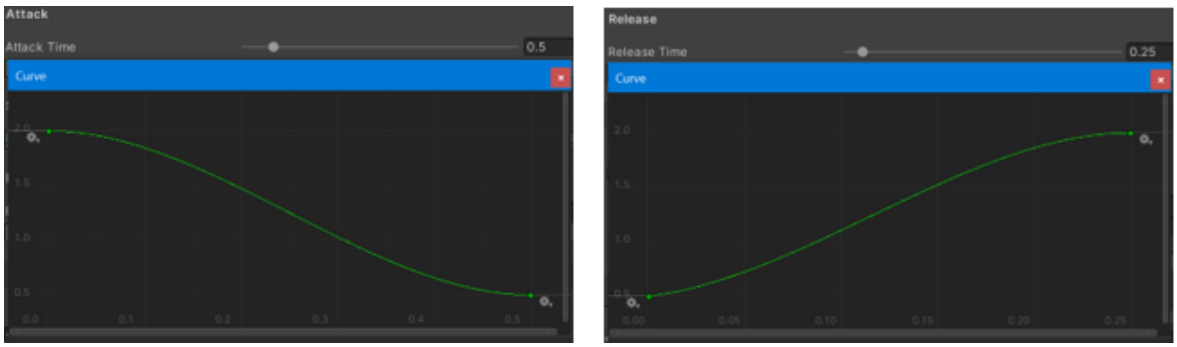

Figura 6.16. Ataque y Relajación de la altura del personaje en el caso 2.

Fuente: Elaboración propia.

- Retroceso:
    - Cantidad de retroceso: La cantidad de retroceso de los ejes X e Y es de 0.5 unidades.
    - *Input*: Se ha asignado el botón izquierdo del ratón por estándares de industria.

Con estos valores se ha conseguido un control de personaje moderadamente rápido y pesado, propio de los videojuegos de género *First Person Shooter*. El movimiento tiene la velocidad adecuada para moverse por el escenario. Además, el salto tiene la fuerza justa como para que el jugador no suba a sitios donde no debe. El *Dash* es lo suficientemente rápido y disponible como para que la ausencia de carrera no se note. Por último, el agachamiento tiene la velocidad adecuada como para que el proceso no sea lento.

6.5.3. Caso 3: *First Person Platformer*

Para crear este caso se ha usado una escena incorporada en los *assets* del *package Sun Temple* (Sandro T, 2018).

Figura 6.17. Captura de la escena del package Sun Temple.

Fuente: Sandro T, 2018.

El proceso inicial para crear este caso es idéntico al del caso 1 y 2 con la excepción de seleccionar el género *First Person Platformer* (Fig. 6.6) antes de crear el controlador. En este caso las mecánicas tienen los siguientes valores:

- Movimiento:
    - Velocidad: Al tratarse de un espacio extenso y de un género basado en la movilidad y la rapidez se ha decidido poner una velocidad de personaje de 13 u/s.
    - Gravedad: Se ha aumentado la gravedad a -29 u/s^2 para evitar saltos muy elevados y para tener una sensación de gravedad verosímil.
    - Suelo: La distancia para detectar suelo es de 0.5 unidades y se ha asignado la *LayerMask Ground,* donde se encuentran todos los objetos de la escena que se consideran suelo.
    - *Input*: Se han asignado las teclas W, A, S, D como las cuatro teclas de dirección.
    - ASR*:* Las gráficas ASR de velocidad de movimiento se han adecuado para que el ataque y la relajación sean sigmoides, aportando suavidad. Sin embargo, los tiempos en este caso se han hecho más lentos para aportar sensación de peso en el ataque (1 s) y sensación de mantenimiento de inercia en la relajación (0.5 s).

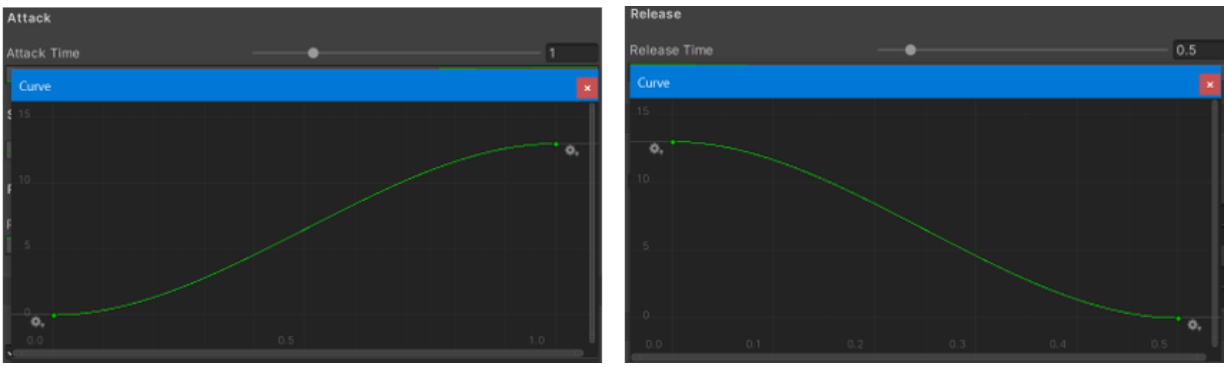

Figura 6.18. Ataque y Relajación de la velocidad de movimiento en el caso 3. Fuente: Elaboración propia.

- Movimiento de cámara:
    - Velocidad de cámara: Se ha optado por una velocidad de cámara de 500 debido al ritmo más rápido que caracteriza los *First Person Platformer*.
    - Ángulo de cámara: El valor mínimo y máximo para el ángulo de cámara es de 80º.
- Salto:
    - Fuerza de salto: Se ha decidido poner una fuerza de salto de 4 unidades debido al tamaño del escenario.
    - *Input*: Se ha asignado la tecla espacio por estándares de industria.
- *Dash*:
    - Velocidad: Se ha asignado un valor de 19 u/s a la velocidad debido a la necesidad de moverse a una velocidad elevada en poco tiempo.
    - Tiempo: Se ha asignado un valor pequeño de 0.5 segundos para compensar la velocidad elevada.
    - *Cooldown:* Se ha puesto un *cooldown* de 0.5 segundos para evitar la posibilidad de hacerlo continuamente, aunque es significativamente baja para mantener la sensación de rapidez.
    - *Input:* Se ha asignado la tecla *Left Shift* por estándares de industria.
- Agacharse:
    - Altura normal: Se ha vinculado manualmente el valor por defecto de la altura del personaje, en este caso 2.
    - Altura mientras el personaje está agachado: Se ha asignado un valor de 0.5 unidades de altura.
    - Velocidad: En caso de no usar ASR se ha asignado una velocidad de agachado de 7.
    - *Input*: Se ha asignado la tecla *C* por estándares de industria.

- ASR: Las gráficas ASR de agacharse se han adecuado para que el ataque y la relajación tengan forma sigmoidal, y tengan una diferencia de un cuarto de segundo entre ellos, 0.5 y 0.25 segundos, respectivamente. Estos tiempos son pequeños por las demandas de rapidez del género.

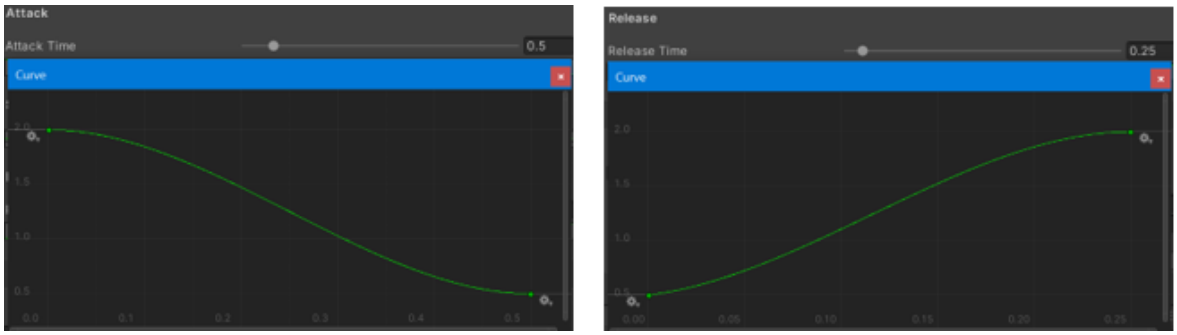

Figura 6.19. Ataque y Relajación de la altura del personaje en el caso 3.
Fuente: Elaboración propia.

- Deslizamiento:
  - Tiempo: Se ha establecido un tiempo de deslizamiento de 0.7 segundos con el objetivo de simular la fricción con el suelo.
  - Velocidad: Se ha aumentado la velocidad hasta 16, para asegurar que se percibe un cambio.
  - *Input*: Se ha asignado la tecla *Left Control* por estándares de industria.
- *Wall Run*:
  - Paredes: La distancia para detectar una pared es de 1 unidade y se ha asignado la *LayerMask Wall,* donde se encuentran todos los objetos de la escena que son paredes.
- Gancho:
  - Velocidad: Se ha asignado una velocidad de 14 u/s, un punto medio entre la velocidad normal y la de deslizamiento.
  - Distancia y paredes: La distancia máxima para usar el gancho es de 20 unidades, y se ha asignado la *LayerMask Wall,* donde se encuentran todos los objetos de la escena que son paredes.

- *Input*: Se ha asignado la tecla *E* debido a su fácil accesibilidad y cercanía con las teclas de movimiento.

Con estos valores se ha conseguido un control de personaje rápido y preciso, propio de los videojuegos de género *First Person Platformer*. El movimiento tiene la velocidad adecuada para que sea satisfactorio moverse por el escenario. Además, el salto tiene la fuerza justa como para que el jugador pueda combinar la velocidad con la mecánica del gancho y desplazarse rápidamente. El *Dash* es lo suficientemente rápido y disponible como para que la ausencia de carrera no se note. Por último, el deslizamiento aporta una buena sensación de juego, añadiendo velocidad en suelo horizontal y en suelo inclinado.

Por cado uno de los casos de uso se ha creado un *prefab* del personaje para que el usuario tenga fácil acceso a un controlador de cada género con los parámetros ajustados.

Una vez se han validado las posibilidades de la herramienta con los casos de uso se ha creado un paquete de *Unity* (el cual está incluido en el Anexo 2) con todos los *scripts* necesarios para su funcionamiento, además de los *prefabs* de cada caso de uso, el manual de instrucciones (incluido en el Anexo 6) y una escena de demostración en la que el usuario tiene acceso rápido a uno de los *prefabs*, con el fin de que pueda probarlo y editarlo como quiera, y asegurarse de que la herramienta se ajusta a sus necesidades.

7. Conclusiones

En este último capítulo se hace una valoración final sobre los objetivos cumplidos y los resultados obtenidos. Además, se proponen unas líneas futuras partiendo de las limitaciones del producto final hacia sus posibles mejoras de cara al futuro.

7.1. Valoración de los resultados

Este trabajo ha conseguido cumplir su objetivo principal de crear una herramienta capaz de crear controladores de personaje en 1ª persona de diferentes géneros con *Game Feel* personalizable. A esta se le ha puesto el nombre de *First Person Controller Creator*. Su desarrollo se ha realizado con éxito con el cumplimiento de los objetivos secundarios planteados.

El primer objetivo secundario cumplido ha sido el de analizar las mecánicas relacionadas con el control de personaje de videojuegos en 1ª persona de diferentes géneros. Para ello se han enumerado las mecánicas de un representante de cada género y se han comparado entre ellos con el fin de encontrar las mecánicas que son comunes y únicas en cada uno.

El segundo objetivo secundario ha sido el que más tiempo ha llevado cumplir, este ha consistido en desarrollar los controladores de personaje en 1ª persona. El análisis previo de mecánicas ha servido para realizar una estructura de clases con herencia que ha permitido la creación de controladores con mecánicas similares sin repetición de líneas de código. De esta manera, los controladores se han separado según su género y se han desarrollado todas las mecánicas que cada uno necesita. Además, se han creado los inspectores personalizados para cada controlador que permiten la edición de variables de cada mecánica de forma sencilla desde el editor de *Unity*.

El tercer objetivo secundario ha consistido en permitir al usuario crear y asignar los controladores de forma rápida y efectiva a partir de una interfaz gráfica en forma de ventana de *Unity*. Para ello se ha utilizado una clase integrada en el motor que ha permitido crear esta ventana con una interfaz personalizada.

El cuarto objetivo secundario ha consistido en aplicar una manera de personalizar el *Game Feel* en mecánicas específicas de cada controlador. Si bien el *Game Feel*

es un concepto muy amplio y abstracto, se ha conseguido cumplir el objetivo mediante la opción de editar curvas ASR en los inspectores personalizados con la ayuda de herramientas que el propio motor *Unity* da a sus usuarios.

El último objetivo que se ha cumplido ha sido el de crear tres casos de uso para la herramienta. Por cada uno de los géneros se ha creado una escena en *Unity* con un controlador diferente, enseñando así la utilidad de la herramienta además de sus posibilidades.

La Tabla 7 compara los dos referentes analizados en el capítulo 4, siendo estos el *Modular First Person Controller* (JeCase, 2021) y el *SUPER Character Controller* (Graves, 2019), con la herramienta desarrollada, *First Person Controller Creator*.

Característica	***Modular First Person Controller* (JeCase, 2021)**	***SUPER Character Controller* (Graves, 2019)**	***First Person Controller Creator***
Manual de instrucciones			
Demo			
Prefab			
Script único			
Uso de inspector personalizado			
Toggles de activación de mecánicas			
Separación visual entre mecánicas en el inspector			
Personalización de *Game Feel*			

Mecánicas no relacionadas con el control de personajes			
Uso de ventanas personalizadas			
Adición automática de componentes			
Uso de directrices de preprocesador para facilitar la lectura de código			

Tabla 7. Tabla comparativa de referentes y producto final. Fuente: Elaboración propia.

En cuanto a contenido, la herramienta creada es similar a los referentes, ya que todas tienen un manual de instrucciones, una escena de demostración y *prefabs* con controladores ya creados. En cuanto a interfaz, la herramienta también es bastante parecida a los referentes, ya que todas usan inspectores personalizados con sus variables editables y *Toggles* de activación de mecánicas. Sin embargo, la disposición de estos se ha separado por mecánicas, pareciéndose más al *SUPER Character Controller* y alejándose del otro. Además, la herramienta creada se separa de los dos referentes por tener una ventana personalizada que automatiza el proceso de asignación de componentes.

Por otro lado, el *First Person Controller Creator* tiene una funcionalidad única entre los tres referentes, la personalización de *Game Feel*, aunque El *SUPER Character Controller* lo compensa añadiendo la funcionalidad única de añadir mecánicas no relacionadas con el control de personaje.

A nivel interno de código, la herramienta creada usa directrices de preprocesador para facilitar la lectura del código, lo cual es común a los demás referentes. La principal diferencia es que los referentes han sido desarrollados con un *script* único en el que se reúne todo el código y la herramienta desarrollada utiliza una estructura de clases con herencia para dividir las mecánicas entre los principales géneros para los que se ha desarrollado, creando así un *script* de controlador por género.

7.2. Líneas futuras

Con todos los objetivos cumplidos se ha conseguido crear una herramienta completa y funcional. Sin embargo, esta presenta ciertas limitaciones y varios apartados a mejorar, tanto en el código como en la propia herramienta.

En cuanto a código hay algunas mecánicas cuyo desarrollo podría haberse hecho de forma más eficiente y pulida. Además, la implementación en código del modelo ASR en las mecánicas se podría haber desarrollado con una mejor estructura y modularidad. Por último, hubiera sido interesante haber creado una máquina de estados para los diferentes estados en los que el personaje se puede encontrar. De esta manera, las variables comunes entre mecánicas, como puede ser la velocidad de movimiento, se hubieran podido controlar mejor.

En cuanto a la herramienta, se puede mejorar la interfaz de usuario para que sea más intuitiva y visual, añadiendo más colores o mejorando el posicionamiento de ciertos botones o campos. En un futuro la herramienta se puede mejorar añadiendo más géneros y, por ende, más mecánicas de las que tiene. Además de que se podrían añadir mecánicas que no estén relacionadas con el movimiento, como por ejemplo sistemas de vida, interacción con objetos del espacio simulado o incluso sistemas de daño para los géneros que lo requieran.

Con estas mejoras incluidas, se plantea la posibilidad de añadir la herramienta en la *Unity Asset Store* para su uso comercial, o personal, de forma gratuita.

8. Referencias

8.1. Bibliografía

Adams, E. (2014). *Fundamentals of game design* (3ª ed.). Pearson Education.

Balaji, S., & Murugaiyan, M. S. (2012). Waterfall vs. V-Model vs. Agile: A comparative study on SDLC. *International Journal of Information Technology and Business Management*, *2*(1), 26-30. Recuperado de: https://mediaweb.saintleo.edu/Courses/COM430/M2Readings/WATEERFALLVs%20V-MODEL%20Vs%20AGILE%20A%20COMPARATIVE%20STUDY%20ON%20SDLC.pdf

Brown, M. (2013). *How to Design for the Gut.* Recuperado de: https://uxmag.com/articles/how-to-design-for-the-gut

Camera. (s.f). Recuperado de: https://learnopengl.com/Getting-started/Camera

Card, S., Moran, T., & Newell, A. (1986). The model human processor- An engineering model of human performance. *Handbook of perception and human performance.*, *2*(45–1). Recuperado de: http://iihm.imag.fr/blanch/ens/2009-2010/M1/IHM1/cours/UIR-1986-05-Card.pdf

Clipping Planes. (2020). Recuperado de: https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2020/ENU/Maya-Rendering/files/GUID-D69C23DA-ECFB-4D95-82F5-81118ED41C95-htm.html

Craighead, Jeffrey & Burke, Jenny & Murphy, Robin. (2007). Using the Unity Game Engine to Develop SARGE: A Case Study. Computer. 4552. Recuperado de: https://www.researchgate.net/publication/265284198_Using_the_Unity_Game_Engine_to_Develop_SARGE_A_Case_Study

Crawford, C. (2003). *Chris Crawford on Interactive Storytelling*. New Riders.

Doran, J. (2015). Prefacio. En A. Tadres, *Extending Unity with Editor Scripting.* Packt Publishing Ltd.

Du, Xiaodong & Liang, Bin & Xu, Wenfu & Wang, Xueqian & Gao, Xuehai. (2012). *A semi-physical simulation system for binocular vision guided rendezvous*. 853-858. DOI: 10.1109/ICARCV.2012.6485269

FirewatchGame. (2022). Recuperado de: http://www.firewatchgame.com/

Ghostrunnergame. (s.f). Recuperado de: https://ghostrunnergame.com/home-es/

Haas, J. (2014). A history of the unity game engine. *Diss. WORCESTER POLYTECHNIC INSTITUTE*. Recuperado de: https://digital.wpi.edu/concern/student_works/tx31qh96p

Hodent, C. (2017). *The gamer's brain: How neuroscience and UX can impact video game design*. CRC Press.

Jonasson, M, Purho P. (2012). *Juice it or lose it.* [Vídeo]. Recuperado de: https://www.gdcvault.com/play/1016487/Juice-It-or-Lose

Juliani, A., Berges, V. P., Teng, E., Cohen, A., Harper, J., Elion, C., ... & Lange, D. (2018). Unity: A general platform for intelligent agents. Recuperado de: https://arxiv.org/abs/1809.02627

Juul, J. (2010). *A casual revolution: Reinventing video games and their players*. MIT press.

Kienitz, P. (27 de febrero de 2017). *The pros and cons of Waterfall Software Development*. DCSL. Recuperado de: https://www.dcsl.com/pros-cons-waterfall-software-development/

Kucic, M. (2005). *How to Prototype a Game in Under 7 Days*. Recuperado de: https://www.gamedeveloper.com/disciplines/how-to-prototype-a-game-in-under-7-days

Lakshminarayana, K. (2019, 10 de Diciembre). What is the difference between orthographic and perspective projections in engineering drawing? [Comentario en un foro en línea]. Recuperado de: https://www.quora.com/What-is-the-difference-between-orthographic-and-perspective-projections-in-engineering-drawing

McCormick, M. (2012). *Waterfall vs. Agile methodology*. Recuperado de: http://www.mccormickpcs.com/images/Waterfall_vs_Agile_Methodology.pdf

Naftis, M., Tsatiris, G., & Karpouzis, K. (2021). How Camera Placement Affects Gameplay in Video Games. Recuperado de: https://arxiv.org/abs/2109.03750#:~:text=The%20analysis%20of%20the%20results,they%20should%20jump%20across%20from.

Nicoll, B., & Keogh, B. (2019). The Unity game engine and the circuits of cultural software. En *The Unity game engine and the circuits of cultural software*. Palgrave Pivot, Cham. DOI: https://doi.org/10.1007/978-3-030-25012-6

Pinch, T., Trocco F. (2004). *Analog Days: The Invention and Impact of the Moog Synthesizer*. Harvard University Press.

Real Academia Española. (s.f.). *Diccionario de la lengua española*. Recuperado en 4 de febrero de 2022, de: https://dle.rae.es/ergonom%C3%ADa

Rogers, S. (2014). *Level Up! The guide to great video game design*. John Wiley & Sons.

Schell, J. (2008). *The Art of Game Design: A book of lenses*. DOI: https://doi.org/10.1201/b22101

SlayersClub. (s.f). Recuperado de: https://slayersclub.bethesda.net/es/media#6Oyq4J9EfewxnJKW06pMoM

Swink, S. (23 de noviembre de 2007). *Game Feel: The Secret Ingredient.* Game Developer. Recuperado de: https://www.gamedeveloper.com/design/game-feel-the-secret-ingredient

Swink, S. (2008). *Game feel: a game designer's guide to virtual sensation*. CRC Press.

Tadres, A. (2015). *Extending Unity with Editor Scripting*. Packt Publishing Ltd.

8.2. Ludografía

Campo Santo, (2016). Firewatch (PC) [Videojuego]. Portland, OR: Panic.

ID Software, (2020). DOOM Eternal (PC) [Videojuego]. Rockville, MD: Bethesda Softworks.

One More Level, (2020). Ghostrunner (PC) [Videojuego]. Milán, Italia: 505 Games.

8.3. *Software* de terceros

Brick Project Studio. (2018). Apartment Kit (3.3) [Software]. Recuperado de: https://assetstore.unity.com/packages/3d/props/apartment-kit-124055

GamesDeveloper12. (2015). EditorTools (1.0) [Software]. Recuperado de: https://answers.unity.com/questions/1073094/custom-inspector-layer-mask-variable.html

Graves, A. (2019). SUPER Character Controller (2022.4.23) [Software]. Recuperado de: https://assetstore.unity.com/packages/tools/game-toolkits/super-character-controller-135316

JeCase. (2021). Modular First Person Controller (1.0.1) [Software]. Recuperado de: https://assetstore.unity.com/packages/3d/characters/modular-first-person-controller-189884

Kutsenko, D. (2017). RPG/FPS Game Assets for PC/Mobile (Industrial Set v2.0) (2.0) [Software]. Recuperado de: https://assetstore.unity.com/packages/3d/environments/industrial/rpg-fps-game-assets-for-pc-mobile-industrial-set-v2-0-86679

Sandro T. (2018). Sun Temple (1.0) [Software]. Recuperado de: https://assetstore.unity.com/packages/3d/environments/sun-temple-115417

Unity Technologies. (s.f). Probuilder (5.0.4) [Software]. Recuperado de: https://unity.com/es/features/probuilder

TecnoCampus

Centres universitaris adscrits a la

upf. Universitat Pompeu Fabra Barcelona

Grado en Diseño y Producción de Videojuegos

Diseño y desarrollo de una herramienta para la creación de controladores de personaje en 1ª persona

ANEXOS

Xavier Lucena Pallarès

Tutor: Dr. Adso Fernández Baena

2021-2022

TecnoCampus Mataró-Maresme

Índice

1. Anexos

A continuación, se enumeran todos los anexos que pertenecen al trabajo, además de la ruta para acceder a ellos en la carpeta de *Google Drive* y su formato.

1.1. Anexo 1: Proyecto de *Unity* y código fuente

Este anexo recoge el proyecto de *Unity* donde se ha realizado la herramienta incluyendo el código fuente que la forma.

- Nombre: HerramientaControladores_Proyecto
- Formato: Carpeta comprimida (zip)
- Ruta: Lucena_Pallares_HerramientaControladores_TFG\Anexos\Anexo 1

El código fuente se encuentra en la siguiente ruta dentro del proyecto de *Unity*:

- First Person Controller Creator\Scripts

1.2. Anexo 2: Paquete con la herramienta

Este anexo tiene la herramienta en formato de paquete de *Unity* para poder ser importada en cualquier proyecto.

- Nombre: First Person Controller Creator
- Formato: Paquete de *Unity* (unitypackage)
- Ruta: Lucena_Pallares_HerramientaControladores_TFG\Anexos\Anexo 2

Las instrucciones para importarlo se encuentran en el Anexo 5**.**

1.3. Anexo 3: Ejecutable

Este anexo tiene un ejecutable donde se pueden probar los tres casos de uso.

- Nombre: HerramientaControladores_Ejecutable
- Formato: Carpeta comprimida (zip)
- Ruta: Lucena_Pallares_HerramientaControladores_TFG\Anexos\Anexo 3

El ejecutable se encuentra dentro de la carpeta comprimida con el siguiente nombre y formato:

- Nombre: TFG
- Formato: Archivo ejecutable(exe)

1.4. Anexo 4: Vídeos demostrativos

Este anexo contiene 4 vídeos demostrativos en los que se enseña el funcionamiento de la herramienta.

1.4.1. Vídeo 1: Importación y funcionalidad

Este video muestra el proceso de importación de la herramienta y la creación de un controlador de personaje funcional.

- Nombre: Vídeo 1 - Importación de herramienta y creación de controlador
- Formato: Vídeo (mp4)
- Ruta: Lucena_Pallares_HerramientaControladores_TFG\Anexos\Anexo 4

1.4.2. Vídeo 2: *Walking Simulator*

Este video muestra el proceso de creación del primer caso de uso, el de un controlador del género *Walking Simulator*.

- Nombre: Vídeo 2 - Caso de uso 1- Walking Simulator
- Formato: Vídeo (mp4)
- Ruta: Lucena_Pallares_HerramientaControladores_TFG\Anexos\Anexo 4

1.4.3. Vídeo 3: FPS

Este video muestra el proceso de creación del segundo caso de uso, el de un controlador del género *First Person Shooter*.

- Nombre: Vídeo 3 - Caso de uso 2 - FPS
- Formato: Vídeo (mp4)
- Ruta: Lucena_Pallares_HerramientaControladores_TFG\Anexos\Anexo 4

1.4.4. Vídeo 4: FPP

Este video muestra el proceso de creación del tercer caso de uso, el de un controlador del género *First Person Platformer*.

- Nombre: Vídeo 4 - Caso de uso 3 - FPP
- Formato: Vídeo (mp4)
- Ruta: Lucena_Pallares_HerramientaControladores_TFG\Anexos\Anexo 4

1.5. Anexo 5: Documento Técnico

Este anexo contiene el documento técnico en el que se muestra el *software* utilizado, las herramientas y material de terceros, y las instrucciones para importar la herramienta en *Unity*.

- Nombre: HerramientaControladores_DocumentoTécnico
- Formato: Texto (pdf)
- Ruta: Lucena_Pallares_HerramientaControladores_TFG\Anexos\Anexo 5

1.6. Anexo 6: Manual de instrucciones

Este anexo contiene el manual de instrucciones de la herramienta.

- Nombre: HerramientaControladores_ManualInstrucciones
- Formato: Texto (pdf)
- Ruta: Lucena_Pallares_HerramientaControladores_TFG\Anexos\Anexo 6