



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN DISEÑO Y DESARROLLO DE VIDEOJUEGOS

Curso Académico 2022/2023

Trabajo Fin de Grado

**DISEÑO Y DESARROLLO DE UN SISTEMA DE COMBATE HACK AND
SLASH: GESTIÓN E IMPLEMENTACIÓN DE ANIMACIONES EN
“ARCANIMA: MIST OF OBLIVION”**

Autor: Marcos Agudo Alarcón

Directores: Julio Guillén García
María Zapata Cáceres



©2023 Marcos Agudo Alarcón
Algunos derechos reservados

Este documento se distribuye bajo la licencia "Atribución- CompartirIgual 4.0
Internacional" de Creative Commons,
disponible en: <https://creativecommons.org/licenses/by-sa/4.0/deed.es>



*“Qué larga ha sido la noche
qué frío y qué oscuridad
pero regresa la vida
y aquí te vuelvo a cantar
Cuántas penitas que he guardao’
dentro del corazón mío
pero aquí estamos de nuevo
que hoy mi copla
ha renacio”*

A Enrique, Mario y Mireya,
por ser el tridente artístico que ha dado luz, color y vida a Arcanima.

Resumen

En este trabajo se lleva a cabo el estudio e implementación de un sistema de combate perteneciente al género de juegos Hack 'n' Slash, profundizando en todos sus subsistemas como la locomoción, la gestión de animaciones y la sensación de juego que trasmite al jugador. Este proyecto se estructura dentro del desarrollo de "Arcanima: Mist of Oblivion", un videojuego en el que se aplican todas las técnicas estudiadas para lograr una jugabilidad satisfactoria sobre la que se articulará toda la experiencia del juego.

Palabras Clave: sistema de combate, gamefeel, animación, desarrollo de videojuegos, Hack 'n' Slash, Unity Engine.

Abstract

In this work, the study and implementation of a combat system belonging to the Hack 'n' Slash game genre is carried out, delving into all its subsystems such as locomotion, animation management and the feeling of gameplay that it transmits to the player. This project is structured within the development of "Arcanima: Mist of Oblivion", a videogame in which all the techniques studied are applied to achieve a satisfactory gameplay on which the whole experience of the game will be articulated.

Keywords: combat system, gamefeel, animation, game development, Hack 'n' Slash, Unity Engine.

Índice de contenidos

Índice de contenidos	6
Índice de ilustraciones	9
Índice de tablas	13
Índice de <i>snippets</i>	14
Glosario	15
Capítulo 1 Introducción	17
1.1 Introducción y orígenes del Hack ‘n’ Slash.....	17
1.2 Importancia de la animación en un sistema de combate.....	19
1.3 Técnicas de animación.....	20
1.4 Estado del arte.....	24
Capítulo 2 Objetivos	28
2.1 Descripción del problema.....	28
2.2 Objetivos.....	29
2.3 Estudio de alternativas.....	29
<i>Unity</i>	29
<i>Unreal Engine</i>	30
2.4 Metodología empleada.....	31
2.5 Planificación.....	33
2.6 Contexto, alcance e integrantes del proyecto.....	36
Capítulo 3 Marco teórico	38
3.1 Teoría de diseño de combate.....	38
<i>Anatomía de un ataque</i>	38
<i>Vocabulario y mecánicas comunes</i>	39
<i>Cadenas de ataques y combos</i>	41
3.2 <i>Gamefeel</i>	42
<i>¿Qué es el gamefeel?</i>	42
<i>Principios del gamefeel</i>	43
<i>Técnicas de gamefeel</i>	44
<i>Casos de estudio</i>	46
3.3 Principios de animación.....	49
3.4 Animación en Unity.....	53
<i>Animation Clips</i>	53



	<i>Animator Controllers</i>	56
	<i>Componente Animator</i>	63
	<i>Retargeting de animaciones</i>	63
3.5	Root Motion y Script Motion	64
Capítulo 4	Diseño de juego	67
4.1	Sistema de combate	67
	<i>Concentración y voluntad</i>	67
	<i>Sistema Ánima</i>	68
	<i>Flujo de combate</i>	69
	<i>Sistema de fijado</i>	70
4.2	Locomoción	71
4.3	Mecánicas	71
	<i>Atacar</i>	71
	<i>Dash</i>	72
	<i>Bloquear y Contraatacar</i>	72
	<i>Concentrarse</i>	73
	<i>Marcar</i>	75
4.4	Personajes	76
	<i>Juno</i>	76
	<i>Soldado raso</i>	77
	<i>Soldado blindado</i>	78
	<i>Autómata volador de ataque</i>	79
	<i>Raj</i>	80
	<i>Nyx</i>	81
Capítulo 5	Descripción informática	83
5.1	Análisis	83
	<i>Extracción de Requisitos</i>	83
5.2	Diseño	86
	<i>Modelo de Dominio</i>	86
	<i>Casos de uso</i>	86
	<i>Diagramas de caso de uso</i>	90
	<i>Diagramas de secuencia</i>	91
	<i>Diagramas de colaboración</i>	92
	<i>Diagrama de clases</i>	93
5.3	Implementación	94
	<i>Sistema de locomoción</i>	94



<i>Sistema de combate</i>	116
<i>Gamefeel</i>	146
<i>Sistema de fijado</i>	152
5.4 Workflow	156
Capítulo 6 Validación	158
6.1 Resultado final	158
6.2 Evaluación de sistemas	159
<i>Playtest de navegación</i>	159
<i>Playtest de combate</i>	161
<i>Alpha Playtest</i>	162
<i>Playtests en ferias</i>	163
Capítulo 7 Conclusiones	165
7.1 Logros alcanzados	165
7.2 Lecciones aprendidas	167
7.3 Líneas futuras	168
Bibliografía	169
Ludografía	176
Anexo I Diagrama de clases	178
Anexo II Animation Events	181
Anexo III State Machine Behaviours	185

Índice de ilustraciones

Ilustración 1- Dos juegos Hack 'n' Slash con 30 años de diferencia: Golden Axe (1989) a la izq. y Devil May Cry 5 (2019) a la dcha.	18
Ilustración 2 – Línea de tiempo de la evolución del género Hack 'n' Slash.....	19
Ilustración 3-- Frame Data de Shulk en Super Smash Bros Ultimate [13].....	20
Ilustración 4 -- Spritesheet utilizada en el juego Cuphead cuando el jugador recorre el mapa del mundo	21
Ilustración 5 -- Ejemplo de personaje 2D preparado para realizar animación esquelética.....	22
Ilustración 6 -- Ejemplo de diferentes blendshapes que definen deformaciones para la boca.....	23
Ilustración 7 -- Comparación del uso de IK: a la izq. sin IK, en el medio con Foot IK, a la derecha con Hand IK [19].....	24
Ilustración 8 -- El videojuego de lucha medieval "For Honor" utiliza Learned Motion Matching en sus animaciones [25].....	26
Ilustración 9-- Sección del diagrama de Gantt del proyecto.....	32
Ilustración 10 - Fases de un ataque: Wind-up, Attack, Recovery (de izq. a dcha.)	38
Ilustración 11- Modo de práctica en Bayonetta 3, en rojo las cadenas de ataques disponibles.....	41
Ilustración 12- Ejemplo de una ADSR.....	44
Ilustración 13- Ejemplos de Easing Functions [46].....	45
Ilustración 14- Dos ejemplos de "estirar y encoger" en las animaciones de los personajes del juego Overwatch: Zenyatta (arriba) y Cassidy (abajo)	50
Ilustración 15 - Diferentes visualizaciones de una animación en la ventana Animation: con curvas (arriba) o con keyframes (abajo)	53
Ilustración 16 - Ventana de configuración de animaciones humanoides	54
Ilustración 17 - Ejemplo de uso de curvas (arriba) y eventos (abajo) en una animación	55
Ilustración 18 - Ventana de modificación de la máscara	56
Ilustración 19 - Ejemplo de Animator Controller	57
Ilustración 20- Ejemplo de un Estado.....	57
Ilustración 21- Ejemplo de Blend Tree	58
Ilustración 22- Ejemplo de transición.....	59
Ilustración 23- Orden de prioridad de las transiciones establecido dentro de un estado.....	60
Ilustración 24- Orden de prioridad según la Interruption Source.....	60
Ilustración 25- Ejemplo de condiciones	61
Ilustración 26- Ejemplo de algunos parámetros	61
Ilustración 27 - Opciones de configuración de una capa	62
Ilustración 28 - Componente Animator.....	63
Ilustración 29- Diagrama del marco de trabajo de animación en Unity [57].....	63
Ilustración 30- Diferentes modelos usando la misma animación mediante Animation Retargeting [60]	64
Ilustración 31- Componente Character Controller.....	65
Ilustración 32- Componente RigidBody.....	65
Ilustración 33- Fortalezas y debilidades de las ánimas	69
Ilustración 34- Flujo de combate.....	70
Ilustración 35 - Interfaz modo concentración	74
Ilustración 36- Cartel anunciando la técnica del soldado raso, "Carga de Valor"	74
Ilustración 37- Soldado blindado marcado	75



Ilustración 38- Personaje Juno	76
Ilustración 39- Combo graph Juno	77
Ilustración 40- Personaje Soldado Raso	77
Ilustración 41- Combo graph Soldado Raso	78
Ilustración 42- Personaje Soldado Blindado.....	78
Ilustración 43- Combo graph Soldado Blindado.....	78
Ilustración 44- Personaje Autómata Volador de Ataque	79
Ilustración 45- Combo graph Automata Volador de Ataque.....	79
Ilustración 46- Personaje Raj.....	80
Ilustración 47- Combo graph Raj.....	80
Ilustración 48- Personaje Nyx.....	81
Ilustración 49- Combo graph Nyx.....	82
Ilustración 50- Modelo de Dominio	86
Ilustración 51- Diagrama de casos de uso.....	90
Ilustración 52- Diagrama de secuencia de realizar un dash.....	91
Ilustración 53- Diagrama de secuencia de empezar a bloquear	91
Ilustración 54- Diagrama de secuencia de personaje dañado	92
Ilustración 55- Diagrama de colaboración de realizar un ataque	92
Ilustración 56- Diagrama de colaboración de escalar salientes	93
Ilustración 57- Diagrama de colaboración de personaje dañado	93
Ilustración 58- Objeto de pruebas que simula un personaje	94
Ilustración 59- Ejemplo de uso del Trail Renderer, el cual dibuja la trayectoria del movimiento	95
Ilustración 60- Level Gym utilizado para probar el movimiento	95
Ilustración 61- Componente CharacterMovement.....	96
Ilustración 62 - Diagrama de clases de los desplazamientos	102
Ilustración 63- ScriptableObjects de los diferentes tipos de desplazamientos: horizontal, vertical y parabólico (de izq. a dcha.)	102
Ilustración 64- Parámetros de configuración del sprint.....	103
Ilustración 65 - Método GetMovementDirection con Inputted Movement Modifiers.....	104
Ilustración 66- Componente JumpFreeFallingHandler.....	105
Ilustración 67- ScriptableObjects del primer y segundo salto del jugador. El primero basado en distancia (arriba) y el segundo en tiempo (abajo).....	105
Ilustración 68 - Doble salto sin y con buffer.....	106
Ilustración 69- Máquina de estados de locomoción básica	108
Ilustración 70 - Blend Tree del movimiento horizontal.....	109
Ilustración 71- Blend Tree para el strafing	109
Ilustración 72- Submáquina del salto	110
Ilustración 73- Animator con root motion activada.....	111
Ilustración 74- Animación con Bake Into Pose desactivada para hacer uso de root motion.....	111
Ilustración 75- Curvas de animación que activan o desactivan el root motion	112
Ilustración 76- Rayos de detección de salientes	114
Ilustración 77- Utilización de un collider adicional (en verde) debido a la delgadez de la geometría para que los rayos detecten la plataforma	114
Ilustración 78 - Ajuste de posición a la izq. y movimiento de subido a la dcha., en rojo script motion y en azul root motion.....	115
Ilustración 79- Máquina de estados común.....	117
Ilustración 80- Componente CombateEntity.....	118
Ilustración 81- Componente Damage Handler.....	119
Ilustración 82- Componente HitboxRecolector	120
Ilustración 83- Componente HurtboxRecolector	121

Ilustración 84- Componente AnimationEventUtilities	122
Ilustración 85- Uso de Animation Events en una animación de ataque	122
Ilustración 86- Combo graph reducido del repertorio de ataques del personaje jugador, Juno.....	123
Ilustración 87- Ejemplo de AttackNode.....	124
Ilustración 88- Todos los tipos de Magnitud existentes.....	124
Ilustración 89 - Todos los tipos de Interrupción existentes	125
Ilustración 90- Ejemplo de nodo de animación con algunos StateMachineBehaviours: StartAttackAnimationDetector, SetAttackNode y ClearAniamtorCombatParameters.....	126
Ilustración 91- Disposición de Hitboxes (verde) y Hurtboxes (rojo) sobre el personaje jugador	127
Ilustración 92 - Componente Hutbox.....	127
Ilustración 93- Diferencia entre hitboxes sin trailing y con trailing	128
Ilustración 94- Eventos EnableHitbox (Fram 9) y DisableHitbox (Frame 13)	128
Ilustración 95- Componente Hitbox	129
Ilustración 96- Ejemplo de CustomCancel para cancelar la acción en un dash	131
Ilustración 97- Ventanas de cancelación y activación en una animación de ataque	132
Ilustración 98- Evento StartDisplacement.....	133
Ilustración 99- Knockbacks del personaje Juno dependiendo de la magnitud del ataque recibido...	133
Ilustración 100- Ejemplo de un Knockback Object.....	134
Ilustración 101 - Secuencia de animaciones cuando un personaje sale volando por los aires.....	134
Ilustración 102- Blend Tree de las animaciones normales de daño.....	135
Ilustración 103 - Attack Node para el combate aéreo	136
Ilustración 104 - Knockback Object para el combate aéreo.....	136
Ilustración 105 - Componente BlockHandler	137
Ilustración 106- Animaciones de bloqueo.....	138
Ilustración 107- Identificador de las acciones del modo concentración.....	139
Ilustración 108- Submáquina de las acciones del modo concentración del personaje Juno	140
Ilustración 109- Condiciones de las transiciones para reproducir las animaciones de consumir poción (izq) y curación (dcha)	140
Ilustración 110- UML FocusActions y Usables.....	140
Ilustración 111- ScriptableObjects de la Curación (izq.) y del Tajo Brutal (dcha). En rojo sus identificador y el Usable que utiliza la cura	141
Ilustración 112- ScriptableObject de la poción. En rojo su identificador y el Usable referenciado....	141
Ilustración 113- Evento de animación UseAct	142
Ilustración 114- Usables de los dos objetos disponibles en el juego	142
Ilustración 115- CharacterData del personaje Juno	144
Ilustración 116- AnimaData pertenecientes a la anima naranja (izq.) y anima roja (dcha.)	144
Ilustración 117- Componente CombatEffects	146
Ilustración 118- Componente CombatEffectsApplier	147
Ilustración 119- Ejemplo de CameraShake.....	148
Ilustración 120- Componente CinemachineImpulseSource.....	148
Ilustración 121- Tipos de gamepad rumble, de izq. a dcha.: Linear Rumble, Pulse Rumble y Constant Rumble	149
Ilustración 122- Hit pause, camera shake y gamepad rumble en diferentes AttackNodes	150
Ilustración 123- Componente VirtualCamerasHolder	151
Ilustración 124 - Prioridades de las cámara virtuales	151
Ilustración 125- Componente TargetableEntity	153
Ilustración 126 - Componente TargetSystem.....	154
Ilustración 127 - Modo debug del sistema de fijado.....	155
Ilustración 128- Objetivos ordenados de menor a mayor ángulo con respecto el jugador según los diferentes rangos	156



Ilustración 129- Respuestas sobre la sensación de movimiento	160
Ilustración 130- Respuestas sobre el comportamiento de la cámara	160
Ilustración 131- Respuestas sobre la sensación de movimiento en combate	161
Ilustración 132- Resultados sobre la sensación de combate	161
Ilustración 133- Respuestas sobre la sensación de movimiento en el alpha test	162
Ilustración 134- Respuestas sobre la sensación de combate en el alpha test	163



Índice de tablas

Tabla 1-- Comparativa Unity-Unreal	31
<i>Tabla 2- Organización de tareas entre febrero 2021 a julio 2021</i>	<i>33</i>
<i>Tabla 3- Organización de tareas entre agosto 2021 y enero 2022</i>	<i>33</i>
<i>Tabla 4- Organización de tareas de la Alpha entre febrero 2022 y junio 2022</i>	<i>34</i>
<i>Tabla 5- Organización de tareas de la Showcase Beta entre julio 2022 y octubre 2022</i>	<i>34</i>
Tabla 6- Organización de Ferias entre noviembre 2022 y diciembre 2022.....	35
Tabla 7- Organización de tareas de la Steam Beta entre enero de 2023 y marzo de 2023	35
Tabla 8- Organización de tareas de la Golden Candidate entre abril de 2023 y julio de 2023	35
Tabla 9- Integrantes y roles del equipo	37
Tabla 10- Naturalezas del ánimo	68

Índice de *snippets*

Snippet 1- Método Update del CharacterMovement, el vector se le suministra al CharacterController de Unity	97
Snippet 2- Método GetComputedMovement, división del movimiento en horizontal y vertical	97
Snippet 3- Método ComputeHorizontalMovement	97
Snippet 4- Método GetWorldMovementDirection	98
Snippet 5- Método Approach para la simulación de aceleraciones y fricciones	98
Snippet 6- Método GetAcceleratedHorMovement	98
Snippet 7- Método GetBrakedHorMovement	99
Snippet 8- Diferente aceleración y fricción cuando el personaje está en el suelo o en el aire	99
Snippet 9- Método ComputeVerticalMovement	99
Snippet 10- Frame Independent en el movimiento	100
Snippet 11- Frame Independent en las aceleraciones y fricciones	100
Snippet 12- Estructura MovementData	101
Snippet 13- Método StartDisplacement	102
Snippet 14- Método StartJump	106
Snippet 15- Rotación del personaje	107
Snippet 16- Métodos ForceDirectionTo	108
Snippet 17- Implementación de OnAnimatorMove para obtener la influencia del root motion	112
Snippet 18- Integración del root motion en el cómputo de movimiento	113
Snippet 19- Desactivación del movimiento proporcionado por el NavMeshAgent	116
Snippet 20- Dirección del movimiento marcada por la velocidad del agente	116
Snippet 21- Actualización manual de la posición del agente	116
Snippet 22- Incorporación de la cancelación para el salto y el dash	131
Snippet 23- Determinación de si la acción actual puede ser cancelada al recibir un ataque	135
Snippet 24- Cálculo del daño infligido según las fortalezas y debilidades del anima	145
Snippet 25- Aturdimiento del personaje	146
Snippet 26- Método ApplyCameraShake	149

Glosario

<i>Build</i>	Creación de una versión ejecutable de un software a partir de su código fuente.
<i>Buffer</i>	Área de memoria temporal utilizada para almacenar y transferir datos entre componentes de un sistema, normalmente utilizado para equilibrar diferencias de velocidad o capacidad y evitar retrasos o problemas de sincronización.
<i>Built-in</i>	Características, funciones o componentes que están incorporados o integrados de forma nativa en un sistema, sin necesidad de instalación o componentes externos adicionales.
<i>Callback</i>	Función que se pasa como argumento a otra función y que será ejecutada más adelante en respuesta a un evento o condición determinada. Generalmente utilizada para manejar eventos, implementar funcionalidad asíncrona o permitir una mayor flexibilidad en el flujo del programa.
<i>Debug</i>	Proceso de identificar y encontrar errores o defectos en el código de un programa, con el fin de asegurar su correcto funcionamiento produciendo los resultados deseados.
<i>Feedback</i>	Retroalimentación, comentarios o respuestas que se proporcionan a alguien o sobre algo con el propósito de mejorar o corregir errores.
<i>Gameplay</i>	Experiencia jugable en la que el jugador interactúa con las mecánicas, reglas, desafíos y sistemas del juego.
<i>Input</i>	Acciones o comandos que un jugador proporciona al sistema a través de dispositivos de entrada, como un teclado, un ratón, o un mando.
<i>Pathfinding</i>	Proceso de encontrar una ruta entre dos puntos en un espacio virtual, evitando obstáculos y teniendo en cuenta las limitaciones del entorno.
<i>Playtesting</i>	Proceso de someter un videojuego a pruebas y evaluaciones por parte de jugadores reales antes de su lanzamiento con el fin de obtener retroalimentación para identificar problemas y mejorar la experiencia de juego.
<i>RPG</i>	"Role-Playing Game" o juego de rol, se trata de un género de videojuegos en el que los jugadores asumen el papel de un personaje y participan en una narrativa interactiva, tomando decisiones que afectan la historia y el desarrollo del personaje.
<i>Snap</i>	Técnica que permite a los objetos o personajes ajustarse automáticamente a una posición específica o a una rejilla predefinida.

<i>Snippet</i>	Fragmento pequeño de código.
<i>Vertical Slice</i>	Versión reducida pero completa de un juego que muestra todos los aspectos clave con el objetivo de representar la experiencia final del jugador.
<i>VFX</i>	“Visual Effects”, se tratan de elementos o efectos visuales utilizados para mejorar la apariencia, la inmersión o la calidad estética. Algunos ejemplos son los sistemas de partículas o el postprocesado.

Introducción

En este capítulo se ahondará en los orígenes y características del género *Hack 'n' Slash*, así como en su evolución a lo largo de los años. Debido a la importancia que ostentan las animaciones en cualquier sistema de combate, se realizará un repaso de las técnicas de animación ampliamente utilizadas hasta llegar a la actual tecnología de vanguardia basada en redes neurales.

1.1 Introducción y orígenes del Hack 'n' Slash

El término *Hack 'n' Slash* hace referencia a un subgénero de los juegos *Beat'em Up*, los cuales se caracterizan por estar mayoritariamente centrados en librar **combates** contra oleadas de **enemigos** a puño limpio o haciendo uso de **artes marciales**. La principal **diferencia** que mantienen los *Hack 'n' Slash* con su género padre es la incorporación del **uso exclusivo** de **armas cuerpo a cuerpo** en sus combates [1], acompañadas ocasionalmente de armas a distancia como recurso secundario.

Actualmente, los *Hack 'n' Slash* no solo se diferencian de otro tipo de juegos basados en enfrentamientos por el uso exclusivo de armas, sino en la ejecución y desarrollo de sus combates. Estos se caracterizan por tener un **ritmo rápido y frenético**, en el que el personaje controlado por el jugador pueda ejecutar de forma fluida un gran número de ataques en poco tiempo. A su vez, los *Hack 'n' Slash* ofrecen un **gran repertorio** de **movimientos** para que cada **jugador** pueda **desarrollar** su propio **estilo de combate** [2].

Los **orígenes** del *Hack 'n' Slash* se remontan a los **juegos de mesa de rol**, en los que se utilizaba este término para definir un tipo de **campaña** en la que la **abundancia** e **intensidad** de los **enfrentamientos** eran las protagonistas [3]. Posteriormente, este término acabaría utilizándose para referirse a videojuegos que conservaran estos mismos pilares.

La **evolución** del *Hack 'n' Slash* empezó **a partir** de dos grandes géneros preexistentes en la década de los 80, los *RPGs* y los *Beat'em Up*.

Por un lado, los *RPGs*, juegos más parecidos a una campaña de **rol clásico**, fueron los primeros en **adaptar** ciertos **elementos** del *Hack 'n' Slash* de rol de mesa en su jugabilidad. Los **primeros indicios** se encuentran en el juego de arcade *"The Tower of Druaga"* (1984) [4], en el que un caballero llamado *Gilgamesh* debería de escalar los 60 pisos de una torre para rescatar a la doncella *Ki* del demonio *Druaga*, para ello el caballero tendría que

encontrar una llave en cada planta para poder avanzar mientras se enfrenta a enemigos con su espada. Este y tantos otros juegos como “*Hydlide*” (1984), “*Dragon Slayer*” (1984) o el mismísimo primer “*The Legend of Zelda*” (1986) [5], formaron los primeros pasos del *Hack ‘n’ Slash* en el medio.

Paralelamente, la línea de los *Beat‘em Up* fue la que más cambios experimentó. “*Golden Axe*” (1989), inspirado en gran medida por “*Double Dragon*” (1987), contaba con una **jugabilidad** muy parecida a la de cualquier *Beat‘em Up* contemporáneo, pero centrada enormemente en el uso de **armas cuerpo a cuerpo** para todos los personajes disponibles [4]. Debido a su alta popularidad, este se convertiría en el juego que popularizó el género *Hack ‘n’ Slash* de la época. Posteriormente le seguirían otros juegos como “*The King of Dragons*” (1991), “*Knights of the Round*” (1991) o “*Guardian Heroes*” (1996).

A **finales** la década de los **90**, los *RPGs* volverían a **evolucionar** el género *Hack ‘n’ Slash* con el lanzamiento de la primera entrega de la saga *Diablo* (1996) [6], en el cual el personaje **jugador** podía realizar una **gran variedad de acciones en tiempo real** durante los enfrentamientos, a **contraposición** de los *RPGs* de la época que se caracterizaban por un sistema de **combate por turnos** [7].

Años más tarde, saldría a la luz “*Dinasty Warriors 2*” (2000) el cual trasladó la jugabilidad de los *Hack ‘n’ Slash* de la época a **escenarios** completamente en **3D** con **libertad de movimiento**, deshaciéndose del característico desplazamiento lateral 2D que predominaba en el género [4]. La saga “*Dinasty Warrior*” dio lugar a su vez a un **nuevo subgénero** llamado **Musou** [8], caracterizado por incluir **grandes hordas de enemigos** a los que el jugador tiene que **derrotar** utilizando **poderosos ataques**. Algunos juegos que siguieron sus pasos fueron *Samurai Warriors* (2004), *Fist of the North Star: Ken’s Rage* (2010), *Hyrule Warriors* (2014) o *Persona 5: Strikers* (2021).



Ilustración 1- Dos juegos Hack ‘n’ Slash con 30 años de diferencia: Golden Axe (1989) a la izq. y Devil May Cry 5 (2019) a la dcha.

A principios de siglo, la aparición de “*Devil May Cry*” (2001) marcaría un **nuevo estándar** en el género *Hack ‘n’ Slash* y sentaría las **bases actuales** por las que se caracteriza el **género: combates rápidos y frenéticos**, gran repertorio de ataques y combos, diferentes armas, medidor de combate, etc. A “*Devil May Cry*” le siguieron sagas tan importantes en el medio como “*Ninja Gaiden*” (2004), “*God of War*” (2005) o “*Bayonetta*” (2009) [9]. A partir de estos juegos a los *Hack ‘n’ Slash* también se les empezó a conocer como **Chatacter Action Games**

[10] [11], debido a que la **jugabilidad** se **centraba** en las **acciones** que los **personajes** protagonistas podían llegar a realizar.

Finalmente, con la aparición de **“Demon Souls”** (2009) el género *Hack ‘n’ Slash* vería su fórmula alterada con **combates** mucho más **lentos**, un gran **aumento** de la **dificultad**, la adición de numerosos **elementos** tomados de los **RPGs** como un sistema de niveles, clases, inventario y estadísticas, y una gran **importancia** de la narrativa basada en el **worldbuilding** del juego [4]. La posterior saga **“Darks Souls”** (2011) fue tan **popular** que una cantidad considerable de **juegos adoptaron** algunas de las **mecánicas** más características a la vez que su **filosofía** de diseño, dando lugar a un **nuevo subgénero**, los **Souls-like**. Entre estos juegos se encuentran **“Salt and Sanctuary”** (2016), **“Nioh”** (2017) o **“Code Vein”** (2019) [12].

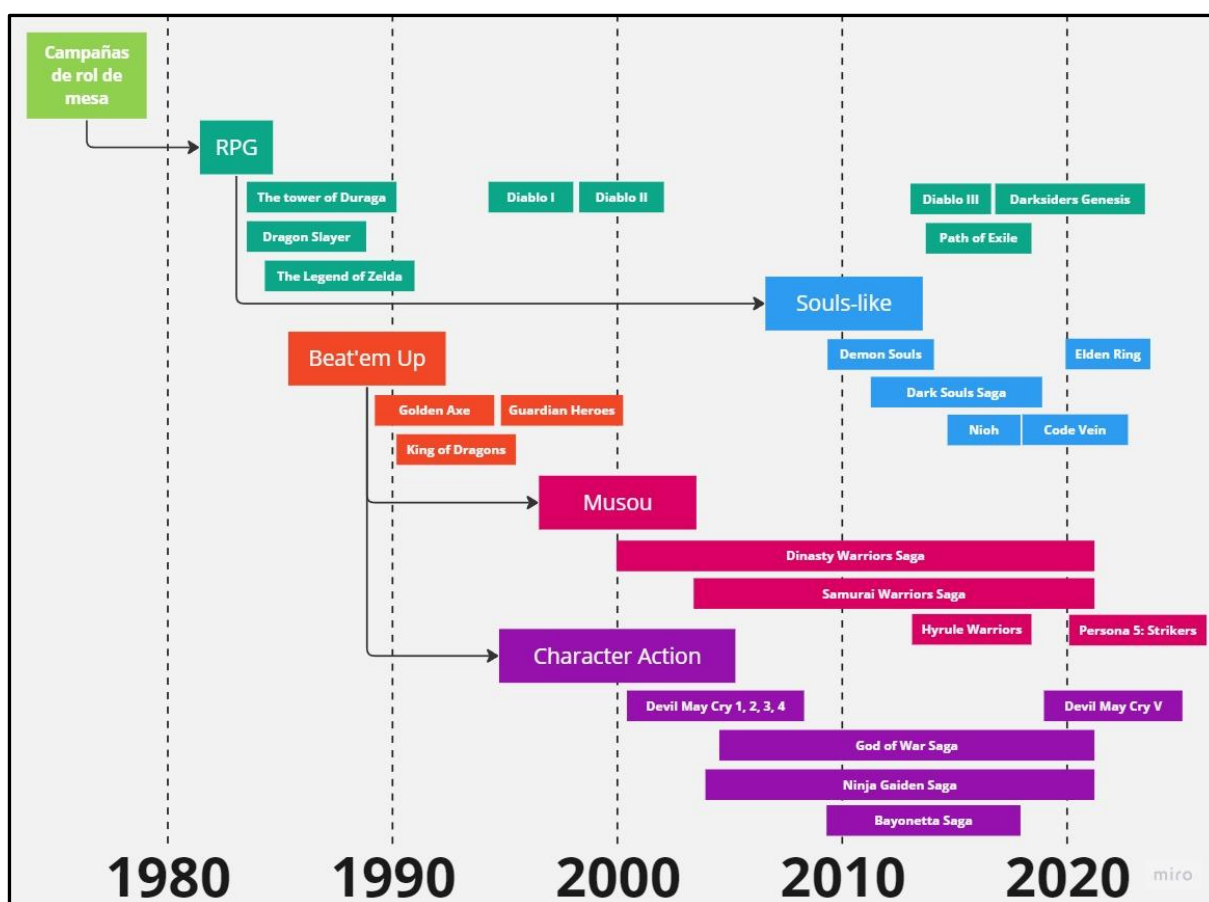


Ilustración 2 – Línea de tiempo de la evolución del género Hack ‘n’ Slash

1.2 Importancia de la animación en un sistema de combate

Como se verá a lo largo del desarrollo de esta memoria, **no es posible hablar** de un Hack ‘n’ Slash, o de cualquier otro **sistema de combate** actual, **sin hacer referencia** a las **animaciones** que se utilizan en el mismo. Estas son las **encargadas** de marcar el **ritmo** e **intensidad** de los **ataques** y, por tanto, del propio sistema en sí: animaciones largas y con gran anticipación dejan entrever ataques potentes que infligen mucho daño, mientras que animaciones cortas y concisas dan lugar a ataques rápidos que tienden a concatenarse.

A su vez, las **animaciones** son una parte **esencial** del **feedback** que se le brinda al jugador. Las **poses** iniciales para indicar un **cambio** de estado - el comienzo de un ataque o bien si el personaje ha sido dañado- o la **telegrafía** integrada en los ataques de los personajes **enemigos** para **leer** bien sus **movimientos** son partes muy **importantes** de un **combate satisfactorio**.

Adicionalmente, en las **animaciones** también se **codifica** y **gestiona** información del **ataque**, indicando, por ejemplo, los frames en los que la caja de colisión del ataque está activa o cuando el jugador puede ingresar un nuevo input para continuar con la cadena de golpes. Esto cobra gran **importancia** en los **juegos de lucha** orientados al **competitivo** como pueden ser la saga de “*Street Fighter*” o “*Super Smash Bros*”. Prueba de ello se puede encontrar en los denominados **Frame Data**, una base de datos que desgrena frame a frame todas las animaciones de ataque de cada uno de los personajes disponibles del juego.

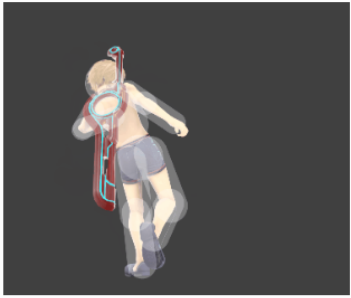
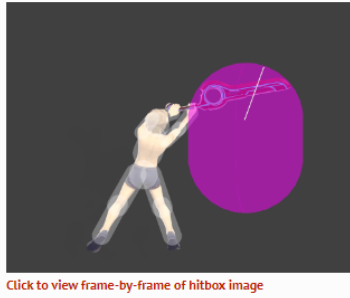
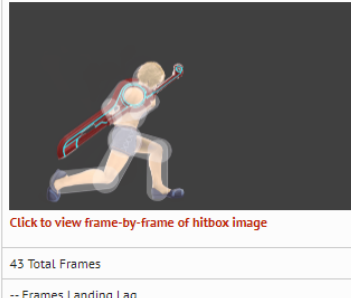
Jab 2	Jab 3	Forward Tilt
5 Frame Startup	6 Frame Startup	12 Frame Startup
Active on 5	Active on 6–7/8	Active on 12–13
28 Frames End Lag	36 Frames End Lag	30 Frames End Lag
-25 On Shield	-32 On Shield	-19/-20 On Shield
		
Click to view frame-by-frame of hitbox image	Click to view frame-by-frame of hitbox image	Click to view frame-by-frame of hitbox image
33 Total Frames	44 Total Frames	43 Total Frames
-- Frames Landing Lag	-- Frames Landing Lag	-- Frames Landing Lag
Note: Transitions to jab 3 as early as frame 12	Note: --	Note: --
1.5% Base Damage	5.0/4.2% Base Damage	13.5/12.0% Base Damage
4 Frames Shield Lag	12/11 Frames Shield Lag	9/9 Frames Shield Lag
3 Frames Shield Stun	6/5 Frames Shield Stun	12/11 Frames Shield Stun
--	Early/Late	Blade/Beam
(If the move has multiple hitboxes, which hitbox is it?)	(If the move has multiple hitboxes, which hitbox is it?)	(If the move has multiple hitboxes, which hitbox is it?)

Ilustración 3-- Frame Data de Shulk en Super Smash Bros Ultimate [13]

1.3 Técnicas de animación

Una vez vista la importancia que decae sobre la animación en un sistema de combate, se revisarán brevemente las técnicas de animación más utilizadas en el ámbito de los videojuegos.

Spritesheets: imitando a la **animación tradicional** en papel, cada uno de los **frames** que conforman una animación se organizan en una **cuadrícula** en forma de sprites dentro de una misma imagen. Una vez montada la spritesheet, dentro del motor de juego se pueden indicar qué sprites concretos de la imagen total corresponden a qué determinadas

animaciones, y así montar las animaciones finales que serán reproducidas a lo largo del juego.

La principal razón de esta técnica es el **ahorro de recursos** al tratar todos los sprites, y por tanto todas las animaciones, como una sola unidad en el hardware gráfico. Esto produce una reducción en las sobrecargas provocadas por operaciones de E/S de datos del disco y por los cambios de contexto al favorecer el principio de localidad de la memoria, además del ahorro de espacio que supone [14] [15].

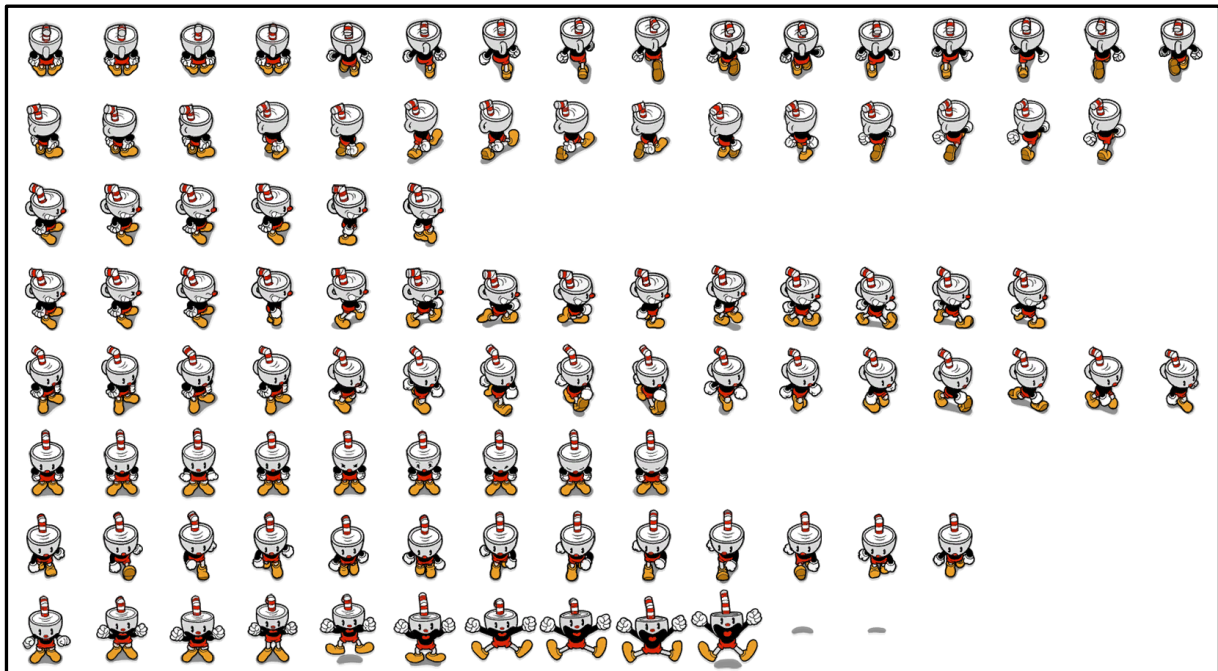


Ilustración 4 -- Spritesheet utilizada en el juego Cuphead cuando el jugador recorre el mapa del mundo

Animación esquelética: se trata de la **técnica estándar** en la industria para la **animación en 3D**, el objeto o **personaje** a animar se **divide** en dos partes principales, la propia **mall** o skin, que es la representación visual o estética, y el **esqueleto** (rig), que representa una jerarquía de partes interconectadas a las que se les denominan huesos. En el proceso de preparar un modelo para que este pueda ser animado (*skinning*), los **vértices** de la **mall** son **asociados** a los **huesos** del **esqueleto** virtual posibilitando que al **mover** un **hueso** este **mueva** en mayor o menor medida los **vértices** afectados, **deformando** así la **mall**.

La manera de crear animaciones, al igual que la animación tradicional, se basa en frames o **keyframes**. Estos, en vez de imágenes como tal, se identifican con la **deformación** resultante de la **mall** al **modificar** la transformada (posición, rotación y escalado) de los **huesos** que conforman el esqueleto. Las **animaciones** pueden ser creadas **“a mano”**, frame a frame con interpolación, mediante programas 3D como *Blender* o *Maya* o desde los propios motores de juegos o bien utilizar **captura de movimiento** para obtener animaciones sobre la marcha de una forma más realista y rápida.

La **animación esquelética** también es **compatible** con la animación **2D**, en vez de asociar vértices de la malla a los **huesos** del esqueleto creado, se **asocian sprites**. Estos *sprites* son partes, como recortables, de un *sprite* final más grande que conforma el modelo final del objeto que se anima [16]. Estos **sprites**, al igual que los vértices, son **deformados por los huesos** designados en el esqueleto. Preparar los sprites funciona de forma muy similar a cómo se haría de forma tradicional para las llamadas animaciones *cutout* [17].

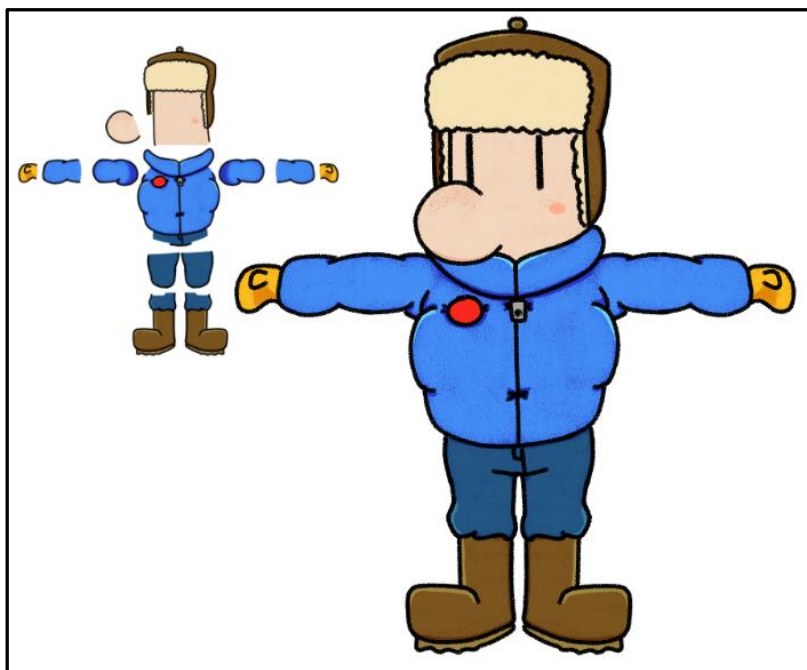


Ilustración 5 -- Ejemplo de personaje 2D preparado para realizar animación esquelética

Blendshapes: una *blendshape* o *morphtarget* se trata de una **versión deformada** de la malla del modelo, las animaciones de *blendshapes* están formadas por **keyframes generados** por la **interpolación** entre estas **deformaciones**. Esta **técnica** es utilizada mayoritariamente para crear **animaciones faciales**; cada expresión viene dada por una *blendshape* (deformación) y el cambio de estas expresiones, es decir, la animación facial, viene dada por la interpolación entre ellas. Aunque no precisa de esqueleto para aplicarse se utiliza en conjunto con la animación esquelética para dar vida y expresividad en la creación de personajes 3D.

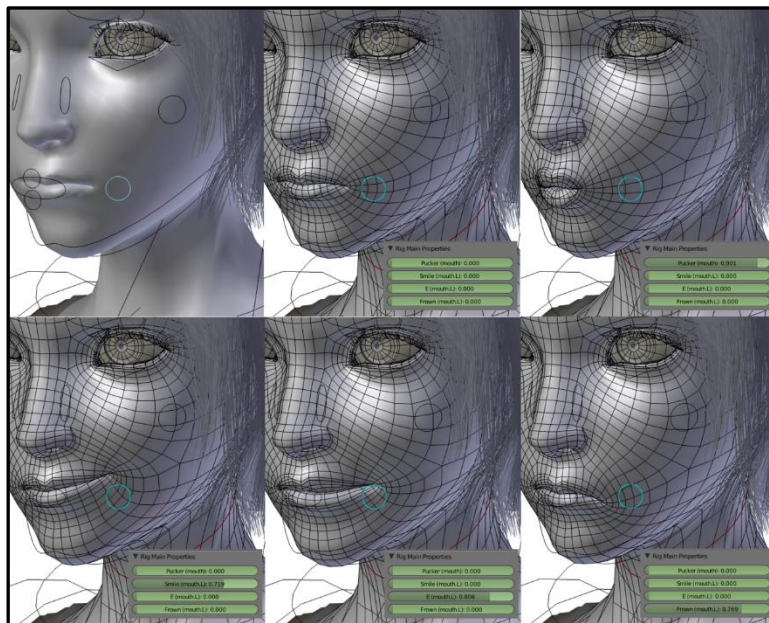


Ilustración 6 -- Ejemplo de diferentes blendshapes que definen deformaciones para la boca

Animaciones procedimentales: a diferencia del resto de técnicas que guardan las animaciones como información ya generada antes de que el juego corra, ya sea como sprites o como posiciones de un esqueleto virtual en forma de keyframes, las animaciones procedimentales son **generadas en tiempo real** por medio de **código** mientras el juego está en funcionamiento. Este tipo de animaciones aprovechan otras técnicas como la esquelética para mover los huesos del modelo desde la programación, pudiéndose ajustar así a distintos escenarios o diferentes situaciones a lo largo del juego [18].

Un ejemplo claro de animación procedural se puede encontrar en las **Inverse Kinematics (IK)** utilizadas para determinar el **movimiento** que necesita el fin de una **articulación** para alcanzar una **posición deseada**. Un caso práctico muy extendido es la **colocación correcta de los pies** de los personajes sobre un **terreno**.

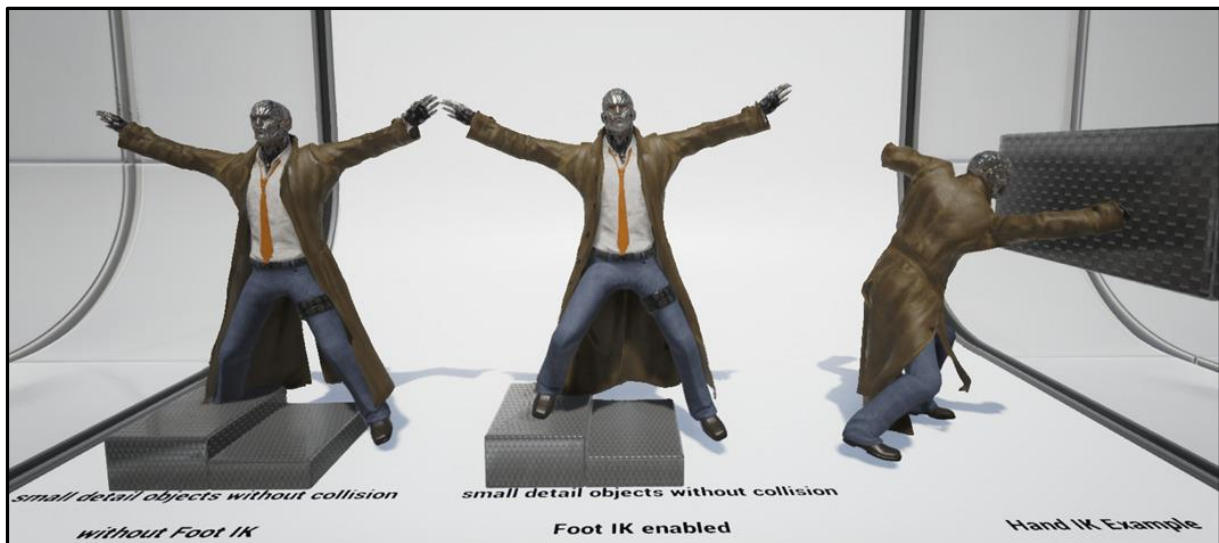


Ilustración 7 -- Comparación del uso de IK: a la izq. sin IK, en el medio con Foot IK, a la derecha con Hand IK [19]

Las animaciones basadas en físicas también forman parte de esta categoría, las llamadas **Ragdoll Physics** es el ejemplo más conocido [20]. Juegos como “GangBeast” o “Human Fall Flat” han hecho de este tipo de animación su seña de identidad.

Estas animaciones procedimentales **no necesitan** necesariamente de la existencia de un **esqueleto** para **deformar** su **malla**, es posible **animar modelos sencillos** utilizando la GPU **desplazando** los **vértices** del modelo por medio de un **shader**. Un gran ejemplo se encuentra en el juego de submarinismo “Abzu”, en el cual todos los peces presentados están animados utilizando este método [21].

1.4 Estado del arte

Hasta hace unos años, el **flujo de trabajo** en la animación de personajes en la industria **AAA** ha estado asentado durante mucho tiempo. El **pipeline** establecido consistía en la **captura de movimiento** de las animaciones necesarias para un personaje concreto, para después ser limpiadas y manipuladas por los animadores para obtener los clips de animación finales que se implementarían más tarde en el juego. Estas animaciones se gestionan a través de una **máquina de estados** en la que cada estado o nodo se encarga de reproducir o mezclar (blending) las animaciones pertinentes.

Las **limitaciones** de esta técnica empiezan a dejarse ver cuando es necesario implementar una **gran cantidad** de **animaciones** en un mismo personaje, requiriendo más nodos en la máquina de estados y, por tanto, aumentando de una manera **exponencial** en los peores casos el número de **transiciones** necesarias para ir de una animación a otra. Esto resulta en una mayor **complejidad** a la hora de **depurar** el grafo de animaciones y en el empeoramiento de la **escalabilidad** del propio sistema.

Aquí es cuando entra en escena el **Motion Matching**, esta técnica **propone** que **todas** las **animaciones** puedan **transicionar** hacia **cualquier frame** de **otra animación** en **cualquier momento** deseado **sin** la necesidad de una **máquina de estados** que defina las transiciones. Se trata de un **algoritmo de fuerza bruta** para la selección de animaciones, ya que en **cada actualización** del juego se procede a **consultar todas** las **animaciones** para **decidir a qué frame** de estas **saltar** [22].

Los **frames** se convierten en **puntos de salto** al que se les **asocia un coste**, el **siguiente frame** al que saltar será el de **menor coste según** unos **parámetros** determinados como pueden ser la **pose** y **velocidades** actuales del modelo y la **trayectoria** futura indicada por el **input** del **jugador**. Así pues, esta técnica se traduce en un **cambio de animaciones** constante, para que estos **cambios instantáneos** de frames **no sean perceptibles** y las animaciones se vean entrecortadas, se realiza una **mezcla** entre el **frame actual** y el **futuro**.

El **motion matching** se adapta **perfectamente** a **sistemas de animaciones** que trabajan en un **espacio**, como puede ser un sistema de **locomoción** y todas sus animaciones derivadas, pero a la hora de adaptar **animaciones** más concretas y **aisladas**, como puede una cadena de ataques, se **compatibilizan** con las **máquinas de estados** anteriormente mencionadas [23]. Estas proporcionan un control más específico y se encargarían de gestionar solo la parte lógica de la animación delegando o haciendo peticiones al sistema de *motion matching* para que este reproduzca la animación como tal.

Una de las claras **ventajas** de esta técnica se aprecia en el salto de **calidad** de las animaciones, ya que al **elegir siempre** el **mejor frame** al que **saltar** estas se reproducen en conjunto de una manera mucho más **realista**. A su vez, se **integra perfectamente** en el **flujo de trabajo** que venía manteniendo la **industria** a lo largo de los años, haciendo incluso más **sencillo** la **implementación** de las animaciones al poder **alimentar directamente** al **sistema** con la **captura de movimiento** previamente obtenida, **sin** la **necesidad** de crear una **infinidad de transiciones** en una máquina de estados. Esto último también afecta a la escalabilidad del sistema, ya que si se requieren más animaciones solo es necesario dárselas al algoritmo para que este las integre automáticamente.

En **contrapartida**, uno de los principales **inconvenientes** se encuentra en el **uso intensivo** de **memoria** que requiere, ya que, a fin de cuentas, el *motion matching* se basa en **buscar** en una **enorme base de datos** de animaciones para dar con el **frame** que **mejor** se **adapte** en un contexto dado, por lo que es necesario **mantener** esta **base de datos en memoria** para **todos** los **personajes** existentes. Este problema se traduce en que debe haber un **compromiso** existente **entre** la **calidad** y **expresividad** del **personaje** y el **presupuesto de memoria**, un asunto importante para la industria AAA que cada vez presenta personajes cada vez más vivos y, por tanto, complejos.

Como **solución** a este **problema** se presenta el **Learned Motion Matching**, cuya idea se basa en **reemplazar** la **base de datos** por una sucesión de **redes neuronales** que, una vez entrenadas, **generan** la **pose** (frame) a la que **transicionar** [24]. Las redes neuronales consiguen aplacar el problema de memoria del que sufría el Motion Matching sin perder calidad en las animaciones.

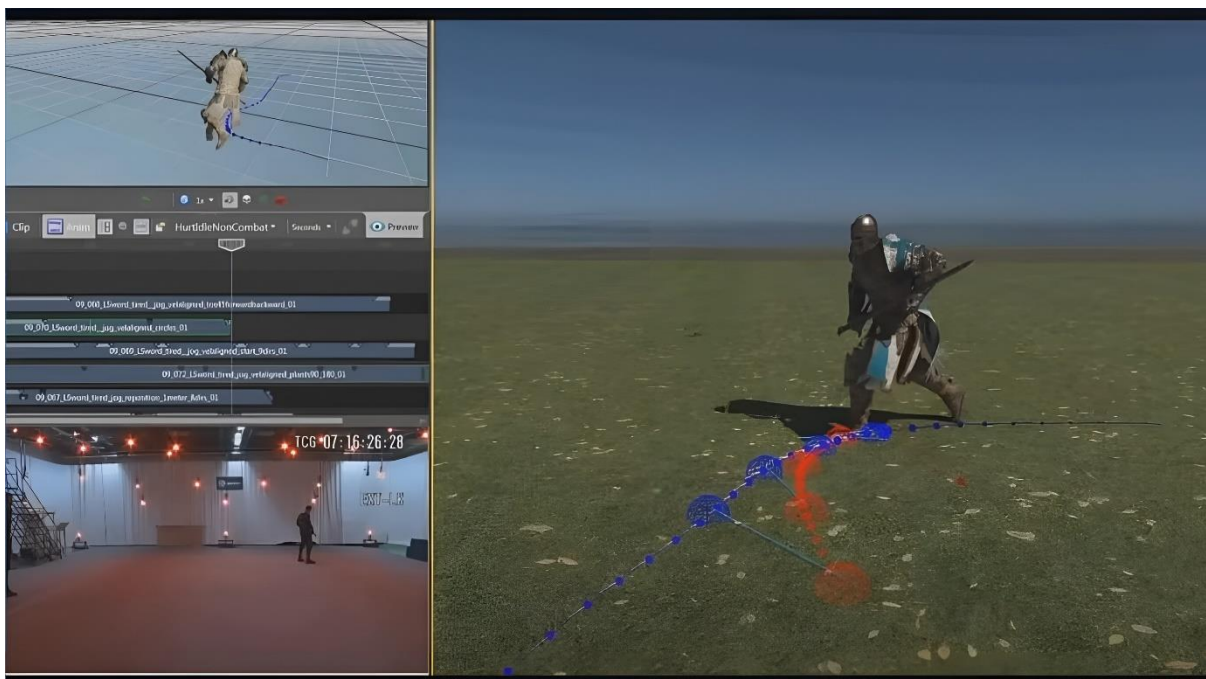


Ilustración 8 -- El videojuego de lucha medieval “For Honor” utiliza Learned Motion Matching en sus animaciones [25]

Con la **incorporación** de las redes neuronales, y más en concreto del **Deep Reinforcement Learning**, a la caja de herramientas se abre un nuevo abanico de posibilidades, el de **animaciones** de personajes **basadas en físicas** [26]. Conforme han pasado los años, los **escenarios** en los **videojuegos** se han vuelto cada vez **más y más complejos y exigentes**, las **físicas ayudan** a las animaciones, y, por tanto, al **personaje**, a **mimetizarse** con estos avanzados **escenarios orgánicos**. Asimismo, se **persigue** más **realismo**, más **fidelidad**, a la hora de que los **personajes reaccionen** a sea lo que sea que les esté sucediendo en su **entorno**. Las **físicas**, además, **aportan** un nuevo nivel de **interactividad** para el jugador con el mundo que rodea al personaje.

Con esto en mente, existen mayoritariamente **dos estrategias** para implementar un personaje basado en físicas: **mantener** el **agente cinemático** como **referencia** para el **agente físico** [27] o **construir** un **agente físico puro independiente** [28]. Ambos tienen la misma base, encontrar las fuerzas que, cuando son aplicadas a los agentes físicos, resultan en las poses deseadas.

En la primera, el **agente físico** se trata de una **copia simulada** del **agente** que está siendo **controlado** por **animaciones**, el **agente físico aprende** a **ajustar** estas **animaciones** según el comportamiento físico deseado [25]. Como resultado se obtiene un agente con un conjunto de animaciones físicas parecidas a las aprendidas que se adaptan al contexto físico de su entorno, esto es, si hay alteraciones en el terreno como pendientes o desniveles, colisiones o impactos. El **agente físico** se puede **entrenar** con un **agente basado** en **Motion Matching** como referencia [29] [30].

La segunda estrategia se basa en **entrenar** con una **animación de referencia**, pero esta, a diferencia del caso anterior, **no interfiere** a la hora de **simular** la **animación resultante**. El agente **aprende desde cero** a **crear** una **copia simulada físicamente independiente** de la animación original utilizada. El **resultado** se trata de un **agente físico especializado** en un **comportamiento** determinado [31].

Actualmente ambas corrientes mantienen caminos diferentes, la primera aboga por la búsqueda de un agente físico universal que, utilizando un agente cinemático, funcione para cualquier personaje y cualquier tarea, mientras que la segunda pretende encontrar un proceso de entrenamiento para un agente físico autónomo que permita combinar diferentes movimientos respondiendo al input del usuario, y así reemplazar completamente a los agentes cinemáticos.



Objetivos

En el presente capítulo se realiza un breve resumen sobre la motivación del trabajo, así como los objetivos propuestos para llevarlo a cabo. Se presenta, además, un estudio de alternativas entre los motores de videojuego más populares en el mercado y se expone en detalle tanto la metodología como la planificación llevada a cabo durante el desarrollo del proyecto. Para terminar, se atiende al contexto grupal del trabajo y al alcance de este.

2.1 Descripción del problema

“Arcanima: Mist of Oblivion” se trata de un videojuego *Hack ‘n’ Slash* 3D en **tercera persona** con **elementos de RPG** por **turnos**. El **jugador** tiene que ser capaz de **entablar enfrentamientos** contra enemigos de una **forma satisfactoria**, es decir, **no es suficiente** con construir un **combate funcional**, es **imperativo** lograr que el **combate** en sí **sienta bien** en las manos del jugador. El sistema tiene que **comunicar** adecuadamente los **golpes**, ser **fluido** y **totalmente reactivo** al **input** del jugador. A su vez, el **combate** es **bidireccional**, lo que significa que hay que **atender** con **detenimiento** tanto al **personaje jugador** encargado de ejecutar los ataques y movimientos, como también a los **enemigos** contra los que se combate, ya que forman parte de la **experiencia global**.

No solo el combate tiene que ser satisfactorio, sino también el movimiento del personaje. Tener una **locomoción fluida** y **ajustada** que permita al **jugador desplazarse** con **libertad** por el **escenario** y **durante el combate** será **clave** para lograr una **buena experiencia** de juego. Es preciso tener en cuenta que las **acciones más frecuentes** que va a estar haciendo el jugador son **desplazarse** por el mundo y **combatir** contra enemigos, por lo que hay que cuidar en gran medida estos dos sistemas que se relacionan entre sí.

Todo esto da como resultado la **necesidad** de un **gran número** de **animaciones** que hay que gestionar e implementar en los sistemas de locomoción y combate ya mencionados. Las **animaciones de locomoción** serán la **parte visual** de la **simulación de movimiento** que corra debajo del personaje y las **animaciones de combate** serán las **protagonistas** a la hora de **implementar** los **ataques** tanto del personaje jugable como de los enemigos.

Por último, estos **sistemas** tienen que poder ser **altamente modificables** debido a la propia naturaleza **iterativa** de ambos, el combate y la locomoción van a estar en **continua**



evolución durante el desarrollo del juego, atendiendo al *feedback* recibido en los *playtests* que se vayan sucediendo.

Por consiguiente, el proyecto plantea un **diseño** e **implementación** de un **sistema de combate Hack 'n' Slash satisfactorio** acompañado de un **sistema de locomoción** con la consecuente **implementación** y **gestión** de todas las **animaciones** pertinentes.

2.2 Objetivos

Los objetivos para el siguiente trabajo son:

- Diseñar e implementar un **sistema de locomoción**
- Diseñar e implementar un **sistema de combate**
- Investigar y aplicar principios y **técnicas de gamefeel**
- Gestionar e implementar las **animaciones** requeridas por los sistemas de combate y locomoción

2.3 Estudio de alternativas

Aunque **Unity** sea la opción **más popular** en **España** para el desarrollo de juegos [32], **Unreal Engine** ha ido ganando terreno [33] y se ha convertido en una alternativa no solo para los estudios de videojuegos AA o AAA, sino también para los estudios más pequeños, atrayendo a un base de usuarios interesados en crear **gráficos realistas** de forma **sencilla**.

A continuación, se presentan tanto los pros como los contras de estas dos opciones:

Unity

Ventajas:

- **Flexibilidad:** se trata de una de las grandes ventajas que ofrece Unity a sus usuarios gracias a su sistema de prefabs y al uso de componentes como forma de scripting, lo que permite implementar cualquier tipo de diseño o arquitectura de manera muy cómoda sin ataduras.
- **Lenguaje de programación:** Unity hace uso de **C#** en su sistema de scripting, el cual ofrece varias comodidades a los programadores, como una gestión automática de memoria gracias a su recolector de basura o una sintaxis amigable y fácil de usar.
- **Experiencia:** este trabajo forma parte de un proyecto en equipo, donde todos los miembros del grupo cuentan con una experiencia más que significativa en el uso de Unity.

Desventajas:



- **Todo por hacer:** en contrapartida a la flexibilidad se encuentra la carencia de un marco de trabajo definido desde el propio Unity, este solo en encarga de dotar al desarrollador de herramientas y componentes básicos para poder trabajar con los sistemas que ofrece
- **Mecanim como sistema de animaciones:** Peca de sencillo, haciendo que los propios desarrolladores tengan que implementar funcionalidad que debería ser *built-in*. Adicionalmente algunas características del sistema no funcionan como deben o son obtusas de utilizar, como el uso de *StateMachineBehaviours* en una submáquina de estado o la imposibilidad de reutilizar estas submáquinas [34].

Unreal Engine

Ventajas:

- **Facilidad de iteración:** en contraposición a *Unity*, *Unreal* ofrece desde el inicio plantillas para una amplia variedad de géneros de juegos, una gran cantidad de componentes específicos para funcionalidades recurrentes y herramientas integradas muy potentes y útiles para todas las disciplinas que aúnan los videojuegos.
- **Gameplay Framework:** *Unreal* proporciona un marco de trabajo en el que todos los elementos que incluye por defecto están estrechamente relacionados, lo que facilita la comunicación entre ellos mejorando la cohesión lógica del código [35].
- **Skeletal Mesh Animation System:** El sistema de animaciones de Unreal ofrece unas funcionalidades mucho más completas que su análogo en Unity. No solo se centra en las máquinas de estados como principal herramienta sino también en la composición de animaciones, montajes de animación, secuencias, poses... haciéndolo una opción mucho más potente [36].

Desventajas:

- **Lenguaje de programación:** Aunque la mayoría de la funcionalidad del motor de Unreal puede ser accedida a través del sistema de programación visual de *Blueprints*, en algunas ocasiones se requiere el uso de código en C++. Sin embargo, a diferencia de C# utilizado en Unity, C++ no es tan amigable y puede ralentizar el flujo de trabajo debido a los tiempos de compilación y a los problemas existentes con su *hot reload* [37].
- **Poca flexibilidad:** aunque se utilicen componentes, la forma predominante de ampliar funcionalidad sigue basándose en herencia. A diferencia de Unity, la creación de nuevos componentes y la composición por medio de anidar objetos dentro de otros no es tan común en Unreal [38].
- **Falta de experiencia:** como la mayoría de los miembros del equipo no tienen experiencia previa en el uso de este motor, sería necesario una fase de aprendizaje y

adaptación a la forma de trabajo en Unreal, lo que comprometería el proyecto en tiempo y calidad final.

	Unity	Unreal
Flexibilidad	★★★★★	★★★★☆
Lenguaje de programación	★★★★★	★★★☆☆
Facilidad de iteración	★★★☆☆	★★★★★
Sistema de animación	★★★☆☆	★★★★★
Experiencia	★★★★★	★☆☆☆☆

Tabla 1-- Comparativa Unity-Unreal

Aunque, debido a la índole de este trabajo, lo que más peso tuviera para decidir entre una de las alternativas fuese el sistema de animación, no hay que ignorar que este trabajo se enmarca en un **proyecto grupal**. En consecuencia, la notable diferencia de experiencia del grupo entre los motores propuestos hace que **Unity** sea la **opción indiscutible**.

2.4 Metodología empleada

En este proyecto, se sigue una metodología que combina **Kanban** con **Scrum**, adoptando un enfoque en **espiral** [39]. El desarrollo se ha dividido en **seis** hitos o **milestones**: prototipo de navegación, prototipo de combate, Alpha y *Showcase* Beta, *Steam* Beta y *Golden Candidate*.

Al inicio de cada **hito**, se **identifican** las **tareas** que deben ser realizadas y se considera el tiempo requerido para su cumplimiento en sesiones de **estimación**. Una vez que todas las tareas han sido estimadas, se registran en el *backlog*¹. La duración y fecha de finalización del hito se calcula sumando los tiempos calculados de todas las tareas.

Cada uno de los hitos se descompone en **sprints** que duran **dos semanas**. Al inicio de cada sprint, se establecen las tareas que deben completarse en un tablero Kanban y, a la terminación de este, se revisa el progreso realizado. Si existiesen tareas sin completar, estas se trasladan al siguiente sprint. Durante los *sprints*, tiene lugar al menos **una reunión por semana** para presentar el avance, discutir los problemas y coordinar el trabajo en equipo

Al **completar** un **hito**, se obtiene como **resultado** un **prototipo** que se somete a pruebas externas mediante **playtesting**. Después de este periodo de pruebas, se analizan los datos recopilados para identificar problemas y riesgos en el prototipo actual o en futuras

¹ En este proyecto se utiliza como *backlog* la aplicación web "*hacknplan.com*"

versiones. Posteriormente, se celebra una reunión para buscar soluciones a estos problemas y llevar a cabo iteraciones sobre el prototipo. Los cambios que se acuerden en esta reunión se deberán aplicar en el próximo hito.

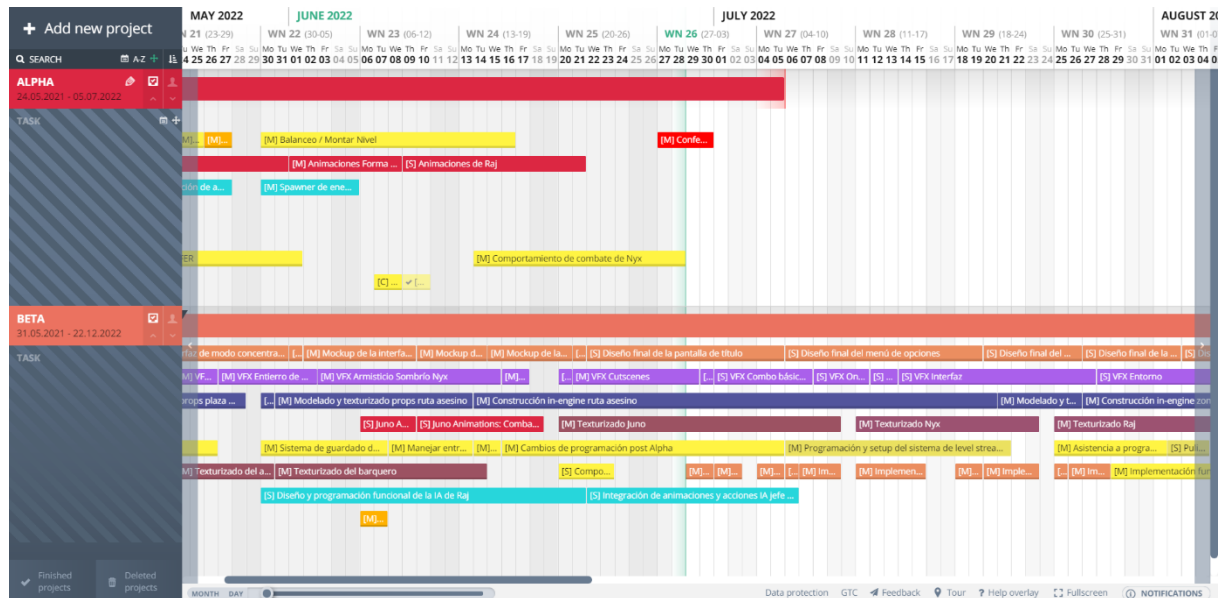


Ilustración 9-- Sección del diagrama de Gantt del proyecto

2.5 Planificación

A proyecto se inició a finales de febrero de 2020, en los siguientes cronogramas se presentan de forma sintetizada los hitos y algunas de las múltiples tareas que se llevaron a cabo:

Feb 2021	Mar 2021	Abr 2021	May 2021	Jun 2021	Jul 2021
Prototipo de Navegación					
Documentación Patrones de Diseño	Locomoción Personaje Principal	Implementar personaje principal	Interactuables	Plataformas móviles	Sistema de Hitboxes
Documentación Sistemas de Combate y Movimiento	Diseño de la Arquitectura	Player Input	Wall Climb	Sistemas de desplazamientos	Primeros ataques
		Input Buffer	Sprint	Implementar animaciones de una cadena de ataques	Object Pooling
		Máquina de estados jugador	Sistema de Eventos		

Tabla 2- Organización de tareas entre febrero 2021 a julio 2021

Ago 2021	Sep 2021	Oct 2021	Nov 2021	Dic 2021	Ene 2022
Prototipo de Navegación			Prototipo de Combate		
Preparar dummy para el combate	Implementar VFXs en el sistema de combate	Preparar Build prototipo de Navegación	Bugfixing prototipo de Navegación	Implementar GameFeel de combate	Preparar Build prototipo de Combate
Estado dañado	Cancelación animaciones	Playtesting prototipo de Navegación	Implementar Soldado raso	Modo Focus	Playtesting prototipo de Combate
Estado muerte	PlayerInfo	Combat Entity	Implementar ataques especiales	Sistema de ánimas	

Tabla 3- Organización de tareas entre agosto 2021 y enero 2022

Feb 2022	Mar 2022	Abr 2022	May 2022	Jun 2022
Alpha				
Mejorar combate	Implementar Autómata volador de ataque	Implementar Soldado Blindado	Mejorar combate aéreo	Preparar Build alpha
Bugfixing prototipo Combate	Consolidar la arquitectura personajes	Avoidance Volumes	Implementar Nyx	Playtesting alpha
	Habilidad Marcar y Acto Retorno			

Tabla 4- Organización de tareas de la Alpha entre febrero 2022 y junio 2022

Las *Showcase Beta* se corresponde con la versión de “Arcanima” utilizada para mostrar en ferias, eventos y concursos. A finales de octubre el juego hizo su primera presentación pública en el **Indie Dev Day**² de Barcelona, además se presentó al concurso de **Big Contest** a la categoría de mejor juego universitario.

Jul 2022	Ago 2022	Sep 2022	Oct 2022
Showcase Beta			
Bugfixing Alpha	Pulido desplazamientos	Level Streaming	Preparar Build Showcase Beta
Pulido del game feel	Pulido Nyx	Guardado de partida	Indie Dev Day
			BIG Contest (mejor juego universitario)

Tabla 5- Organización de tareas de la Showcase Beta entre julio 2022 y octubre 2022

Los meses de noviembre y diciembre se dedicaron a presentar el juego en las siguientes ferias: **BIG**³, **Guerrilla**⁴ y **Gamergy**⁵. Entre un evento y otro se llevaban cambios derivados del *feedback* recogido por los jugadores.

² <https://www.indiedevday.es/en/juegos-2022/arcanima-mist-of-oblivion/>

³ Bilbao International Games Conference (<http://www.bilbaogamesconference.com/>)

⁴ <https://guerrillagamefestival.es/>

⁵ <https://www.ifema.es/gamergy>

Nov 2022	Dic 2022
Ferias (Playtesting)	
Bilbao International Games Conference (BIG)	Guerrilla Game Festival
	Gamergy

Tabla 6- Organización de Ferias entre noviembre 2022 y diciembre 2022

La **Steam Beta** se trata de la versión del juego que se sube a la plataforma de juegos Steam para realizar un playtesting cerrado entre los jugadores que se han interesado por "Arcanima"

Ene 2023	Feb 2023	Mar 2023
Steam Beta		
Bugfixing & Feedback Showcase Beta	Implementar Raj	Estudio y Mejora del rendimiento
Level Streaming	Guardado de partida	Playtesting Steam Beta

Tabla 7- Organización de tareas de la Steam Beta entre enero de 2023 y marzo de 2023

Desde abril de 2023 el juego se encontraría acabado y solo se estaría trabajando en el pulido general necesario para conseguir una versión final comercializable, la **Golden Candidate**. De forma adicional, "Arcanima" se presentará al evento **Steam Next Fest** de Steam de octubre para todo el mundo que quiera probar el juego. La versión que se expondrá se tratará una versión pulida del mismo contenido que se dispuso en la *Steam Beta*.

Abril 2023	Mayo 2023	Junio 2023	Julio 2023
Golden Candidate			
Bugfixing & Feedback Steam Beta	Pulido del game feel	Implementar Steam Achivements	Preparar Build Gold Candidate
Pulido Raj	Foot IK		

Tabla 8- Organización de tareas de la Golden Candidate entre abril de 2023 y julio de 2023

Aunque según la planificación el juego ya lleva tiempo terminado en cuanto a contenido, la fecha del lanzamiento de “*Arcanima: Mist of Oblivion*” aun está por decidir.

2.6 Contexto, alcance e integrantes del proyecto

“Arcanima: Mist of Oblivion” se trata una **vertical slice** creada en colaboración con otros estudiantes del grado. Este proyecto sirve como base para desarrollar dos trabajos finales de grado:

- Diseño y desarrollo de un sistema de combate Hack and Slash: Gestión e Implementación de animaciones en “Arcanima: Mist of Oblivion”
- Patrones de diseño aplicados a videojuegos. Ejemplo Práctico: Arquitectura de personajes en “Arcanima: Mist of Oblivion”

Se tratan de trabajos complementarios, correspondientes respectivamente al grado de Diseño y Desarrollo de Videojuegos y al grado de Ingeniería de Computadores.

El **presente trabajo**, “Diseño y desarrollo de un sistema de combate...”, **abarca** el **diseño e implementación** del sistema de **combate** junto con el sistema de **locomoción**, pero **no** trata la **arquitectura** de **personajes** subyacente sobre la que se construyen estos sistemas. Por ejemplo, se atenderá a **cómo se construyen los ataques**, pero **no a cómo** se lleva a cabo la acción de **atacar** de los personajes, es decir, en este trabajo no importa la capacidad de atacar, sino la implementación de los diferentes ataques concretos que realiza un personaje. Así mismo, se profundizará en cómo funciona el sistema de locomoción, pero no en cómo se gestiona la capacidad de moverse de los personajes.

Toda la **arquitectura** de **personajes** que **posibilita** estas **capacidades** y muchas más se haya totalmente explicado en el correspondiente **TFG** “Patrones de diseño aplicados a videojuegos...” perteneciente al grado de **Ingeniería de Computadores**.

Para terminar este capítulo, se presentan en la siguiente tabla todos los integrantes que han llegado a conformar el equipo y los roles que han desarrollado:



Integrante	Rol
Alberto Alcázar	Programador de Audio Compositor Artista Foley
Alberto Romero	Artista personajes 3D Animador
Blanca de la Fuente	Artista personajes 3D
Enrique Sánchez	Diseñador de personajes Diseñador de interfaz
Flavio C. Jiménez	Programador de IA
Marcos Agudo	Lead Programmer Programador de gameplay Programador de animación
Mario Belén	Artista técnico Artista de VFX
Mireya Funke	Artista de <i>props</i> y escenarios Diseñadora de niveles
Mónica Sánchez-Pretilo	Community Manager
Víctor Sierra	Director creativo Programador Generalista Productor

Tabla 9- Integrantes y roles del equipo

Marco teórico

En este capítulo se atenderá a la teoría necesaria para llevar a cabo los objetivos propuestos, se definirá un vocabulario y unas mecánicas comunes a cualquier sistema de combate, se ahondará en la definición de gamefeel y sus técnicas más utilizadas y, por último, se expondrá los principios de animación utilizados universalmente seguido del marco de trabajo que ofrece Unity para gestionar e implementar animaciones.

3.1 Teoría de diseño de combate

Para crear un estilo de combate determinado es necesario diseñar un sistema que cumpla con los requisitos y necesidades que definan a ese estilo, si el combate es muy rápido y acrobático es necesario gran fluidez en los ataques y un gran repertorio de animaciones, si, por el contrario, el combate es lento pero preciso es indispensable cuidar en detalle la construcción de los ataques y sus colisiones. Atendiendo a la teoría, es posible saber qué conceptos y mecánicas aplicar para construir un estilo de combate que encaje con la experiencia objetivo del juego.

Anatomía de un ataque

La ejecución de un ataque puede ser dividido en tres fases principales [40]:

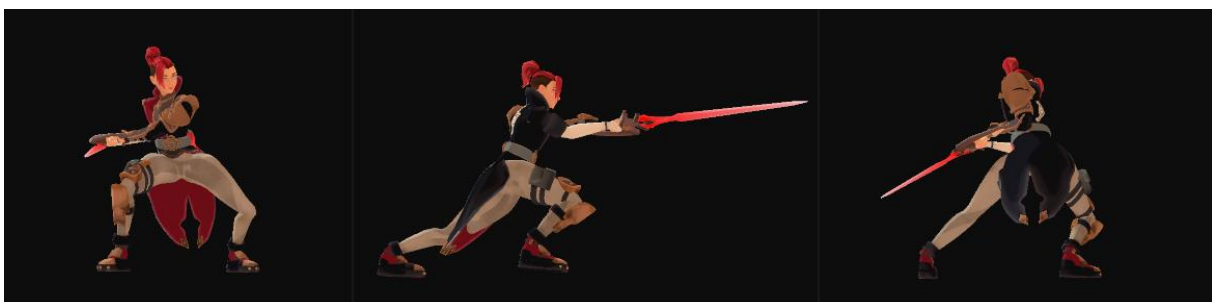


Ilustración 10 - Fases de un ataque: Wind-up, Attack, Recovery (de izq. a dcha.)

- **Wind-up:** se trata de una **anticipación**, con ella se consigue dar claridad a la animación y transmitir la intensidad del ataque. A mayor wind-up se espera una mayor fuerza en el ataque. La **pose** de la animación con la que se inicia esta anticipación es muy **importante**, debido a que es el **primer** atisbo de **feedback** que el jugador obtiene al haber pulsado el botón de ataque, una pose ambigua o poco

marcada puede hacer parecer al ataque como poco reactivo o que los controles tardan en responder.

- **Attack/Strike:** se trata del ataque en sí, se identifica con los **frames** de la animación en los que el personaje **inflige daño**, esto es, la **hitbox** del ataque se encuentra **activas** para registrar colisión y aplicar el impacto. El tiempo que estén activas la **hitbox** no tiene porqué coincidir con la parte visual de la animación, es posible activarlas de forma más temprana o desactivarlas con un poco de retraso a la terminación del golpeo para dar la sensación al jugador de un tiempo de respuesta más rápido.
- **Recovery:** última fase del ataque, en esta región el personaje se **recupera** de haber dado el golpe, es decir, vuelve a su posición inicial antes de haber realizado el ataque. Por ejemplo, si durante la animación ha avanzado un paso o ha adoptado una pose, aquí es donde el personaje deshace todos esos movimientos. Normalmente esta etapa sirve tanto para **equilibrar ataques** o dar **oportunidades** de **golpeo** como para **vender** la **fuerza** del **ataque**, a más intensidad se espera una mayor fase de recuperación por parte del personaje.

Vocabulario y mecánicas comunes

A continuación, se presentan términos y conceptos utilizados frecuentemente en el diseño de un combate para referirse a cierto tipo de situaciones o mecánicas recurrentes [41]:

Smasher: ataque en picado realizado desde el aire, el personaje se precipita hacia el suelo efectuando daño en área.

Launcher: ataque que eleva por los aires a los personajes que reciben daño, el personaje que realiza el ataque también puede verse elevado para seguir golpeando en el aire más fácilmente.

Ataque en carrera: ataque de embestida normalmente realizado al llegar a una cierta velocidad o al haber efectuado una esquivada.

Ataque aéreo: cualquier ataque ejecutado en el aire.

Ataque cargado: ataque potente que necesita un tiempo de canalización, normalmente manteniendo apretado el input apropiado del ataque.

Bloqueo: permite evitar el daño recibido por el atacante mientras se está estático adoptando una pose defensiva.

Parry/Contraataque: mientras que el bloqueo previene el daño recibido, el contraataque no solo para el ataque, sino que además permite devolver el golpe al enemigo.



Agarre/Lanzamiento: el personaje que realiza el agarre inmoviliza a su objetivo, las condiciones de poder agarrar a un objetivo varían dependiendo del juego. Una vez agarrado se puede lanzar al enemigo por los aires pudiendo resultar en una eliminación instantánea.

Asesinato sigiloso (*Stealth kill*): utilizado principalmente en juegos de sigilo, cuando el jugador no es detectado puede realizar una eliminación instantánea por la espalda. Existen otras variaciones como caer en picado desde una cierta altura hacia el objetivo a eliminar.

Ejecuciones: se trata de una eliminación instantánea del objetivo durante el combate, las condiciones para poder realizar una ejecución varían según el juego y el tipo de enemigos a los que el jugador se enfrenta. Algunos tropos comunes pueden ser efectuar ejecuciones sobre enemigos pequeños muy débiles o sobre enemigos comunes cuando ya se encuentran debilitados.

Acompañantes: personajes o agentes que siguen continuamente al personaje jugador y complementan el conjunto de acciones que este puede llegar a realizar. Algunos ejemplos son el POD en “Nier Automata” que ofrece ataques a distancia, las legiones en “Astral Chain” en la que se basa enteramente la mecánica principal del juego o el personaje de Lute en “Soulscite” que brinda protección al jugador.

Juggling: en el combate aéreo, referido a la acción de concatenar ataques sobre el enemigo sin que este caiga al suelo, como si el jugador estuviese haciendo “malabares” [42].

Stagger: estado en el que un personaje no puede realizar ninguna acción debido a que está siendo golpeado continuamente sin darle tiempo a recuperarse entre golpes.

Stun/Aturdido: estado en el que el personaje se encuentra inmovilizado durante un periodo de tiempo sin poder realizar ninguna acción dejándolo totalmente vulnerable a cualquier ataque.

Knockback: desplazamiento producido al recibir un golpe, durante el *knockback* el personaje puede llegar a perder el control, la distancia del desplazamiento o la duración de este depende de la magnitud del ataque recibido. La sucesión rápida de *knockbacks* pueden llegar a causar *stagger* sobre un personaje.

Airborne: *knockback* que lanza por los aires al personaje objetivo. El personaje tiene que esperar a llegar al suelo y levantarse para poder realizar acciones, normalmente cuando se le aplica al personaje jugador existen maneras de cancelarlo, como recomponerse del golpe mientras se está en el aire o cancelar la animación de levantarse del suelo.

Esquivas: desplazamientos en una dirección para evitar físicamente un ataque, además dotan al personaje de un periodo de invulnerabilidad y son de gran utilidad para reposicionarse durante el combate. Existen diferentes tipos de esquivas dependiendo de la animación y el movimiento que se ejecute, por ejemplo, mientras **Roll** hace una referencia a una voltereta, **Dash** se asocia con un movimiento rápido y normalmente corto.

Strafing: es el acto de moverse lateralmente, ya sea en relación con un objetivo o con la cámara del juego. Permite al jugador mantener la cámara centrada en un objetivo, como un enemigo, mientras se mueve en otra dirección. En términos de animación, el personaje no mira hacia dónde se mueve, sino hacia un objetivo, por lo que son necesarias diferentes animaciones que cubran las direcciones principales del movimiento (adelante, atrás, izquierda, derecha, diagonal izquierda hacia adelante, diagonal izquierda hacia atrás...).

Cadenas de ataques y combos

Una **cadena** de ataques consiste la **sucesión** de dos o más ataques. Estas cadenas permiten al jugador **cancelar** la etapa de **recovery** del ataque actual y pasar directamente a la fase de **wind-up** del siguiente, consiguiendo así una **concatenación fluida**. Normalmente, mientras el jugador se adentra más y más en la cadena de ataques este irá teniendo acceso a movimientos más fuertes y potentes.

Los ataques que comienzan una cadena son conocidos como **starters** y los que dan fin a una sin posibilidad de cancelar su etapa de **recovery** en otros ataques se denominan **finishers** [41].

La diferencia principal entre una cadena de ataques y un **combo** reside en que un combo es un caso particular de una **cadena**, la cual es muy **complicada** de **interrumpir** por el enemigo [43]. Si bien este término cobra mucha mayor relevancia mecánicamente hablando en juegos competitivos de lucha, en combates de un solo jugador no se atiende tan detalladamente a este matiz.

Las cadenas de ataques pueden ser representadas mediante una tabla, un grafo que indique su flujo (*combo graph*) o una lista de inputs a introducir en el caso de presentarlas al jugador dentro del juego.



Ilustración 11- Modo de práctica en Bayonetta 3, en rojo las cadenas de ataques disponibles

3.2 Gamefeel

El **gamefeel** o la sensación de juego es uno de los **pilares** fundamentales a la hora de construir un videojuego, es lo que realmente marca la diferencia entre un buen juego y un juego mediocre, se trata de la importancia de **sentir el videojuego**, sentir las acciones que se están ejerciendo dentro y sobre el mundo virtual. El gamefeel cobra aún más importancia si cabe cuando se trata de un sistema de combate el que los ataques y el movimiento tienen que fluir y responder satisfactoriamente a las órdenes del jugador, si el gamefeel falla, la experiencia de combate y, por tanto, la experiencia objetivo del juego también fallan.

¿Qué es el gamefeel?

No existe una definición estándar de *gamefeel*. Al tratar un juego se puede llegar a hablar de la sensación que transmite al jugarlo, definiéndolo como “suelto”, “apretado” o “torpe”, pero no se ha llegado a definir colectivamente y cada diseñador o jugador puede tener su propia interpretación del término. Por ello y para tener una idea concreta que sirva de utilidad cuando en este trabajo se haga referencia al *gamefeel*, se atenderá a la definición aportada por Steve Swink recogida en su libro “*Game Feel: a game designer’s guide to virtual sensation*” [44].

Para Swink el gamefeel se identifica con el **control en tiempo real** de **objetos virtuales** en un **espacio simulado**, con **interacciones enfatizadas** por el **pulido**.

A continuación, se procede a explicar cada uno de los conceptos involucrados en la definición:

- **Control en tiempo real:** habilita la **interactividad** de una forma aparentemente instantánea para el jugador, el input se percibe en el mismo momento en el que ha sido expresado lo que permite un control continuado y preciso del objeto virtual.
- **Objetos virtuales:** cualquier **entidad** dentro del **mundo virtual**, estas entidades pueden ser controladas por el jugador como el personaje avatar o ser reactivas a las acciones del jugador.
- **Espacio simulado:** hace referencia a las **interacciones** simuladas físicamente en un espacio virtual, así como al **espacio virtual** en sí. Las respuestas físicas y relaciones que se dan en el mundo virtual y sus objetos a través del avatar jugable permiten al jugador experimentar el espacio virtual de una forma táctil y física.
- **Interacciones:** **acciones** o verbos con el que el **jugador** puede **modificar** el **estado** y **atributos** del mundo y el de sus objetos a través del avatar jugable.
- **Pulido:** cualquier **efecto** que **potencie** la **interacción** sin cambiar su funcionalidad base. Esto puede ser desde la utilización de partículas, vibración de cámara para enfatizar o hasta respuestas hápticas como vibración del mando.



Así pues, se podría reformular la definición de **gamefeel** como el **conjunto de interacciones** continuadas e instantáneas con las **entidades**, por medio de **acciones enfatizadas** por efectos audiovisuales y hápticos, en un **espacio virtual** reactivo que se relaciona físicamente con el jugador.

Principios del gamefeel

A continuación, se presentan los principios generales expuestos en el libro de Swink orientados a crear juegos con un buen gamefeel:

- **Resultados predecibles:** cuando los **jugadores** realizan una acción deben de **obtener** la **respuesta** que ellos **esperan**. Esto no quiere decir que el juego deba de ser sencillo, no es una cuestión de habilidad, sino de **evitar** que haya **interferencias**, ambigüedades o problemas de comprensión entre la **intención** del jugador y el **resultado**. Por ejemplo, si el juego implementa una ayuda al escalar salientes y el jugador espera que al saltar hacia uno de ellos el personaje se agarre, pero este no lo hace debido a que la detección de salientes pueda fallar, el jugador se encontrará una interferencia entre su intención y el resultado esperable, perjudicando a la experiencia.
- **Respuesta inmediata:** esto no quiere decir que todas las acciones del jugador se tengan que realizar rápidamente de por sí, por ejemplo, hay juegos que implementan armas muy pesadas cuyos ataques pueden llegar a ser muy lentos. No importa tanto la velocidad de una **acción**, sino que se **transmita** de forma **inmediata** que se va a realizar una vez el jugador así lo ha expresado en el input; lo importante es la **existencia de cambios notables** que **comuniquen** al jugador que la **acción** se va a dar. En el caso del ataque pesado mencionado anteriormente, esto puede identificarse con el cambio de pose que genera la animación del personaje al indicar que se prepara para golpear.
- **Fácil pero profundo:** el gamefeel también se encuentra en el placer de **aprender, practicar** y **dominar** una **habilidad**. Los juegos que presentan al jugador unas habilidades básicas fáciles de aprender, pero siempre aportan nuevos niveles de maestría a los que aspirar, poseen una mejor sensación de juego debido a que esta va ligada al progreso del jugador.
- **Novedad:** para mantener el interés del jugador, el juego necesita **sentirse novedoso** e interesante después de horas de juego, incluso **acciones repetitivas** deberían sentirte **frescas** cada vez que se realizan. Por mucho que una mecánica sea divertida al principio, cuando pase suficiente tiempo esta pueda llegar a quemarse y aburrir al jugador.
- **Respuesta atractiva:** para **cualquier input** del jugador que reciba el sistema, el **resultado** debe resultar **convinciente** y **atrayente**. Estas respuestas deben de sentirse bien **sin necesidad** de que se efectúen dentro de un **contexto**. Un claro ejemplo es el

control de un personaje en un espacio vacío; la propia acción de moverse tiene que resultar divertida y agradable per se.

- **Movimiento orgánico:** un **movimiento fluido y orgánico** es muy importante a la hora de construir un buen gamefeel, no solo para el control de objetos virtuales sino también para animaciones de cualquier tipo ya sea para efectos, personajes o incluso la interfaz gráfica. El uso de *easing functions* o *tweening* ayuda a conseguir estos efectos.
- **Armonía:** cada elemento de la sensación de un juego apoya una sola percepción cohesiva de una única realidad física, esto es, una **realidad presentada** en el juego **debe actuar** como esa realidad en **todos** sus **aspectos**. Por ejemplo, si un objeto virtual se presenta como un coche, este objeto se deberá controlar como un coche, verse como un coche, escucharse como un coche (ruido de motor, ruedas contra el asfalto, etc), interactuar con el espacio virtual físico como un coche... en definitiva, sentirse como un coche.

Técnicas de gamefeel

Cada juego precisa de unas necesidades especiales a la hora de construir su propio gamefeel, dependiendo de su diseño, llegando incluso a tener que desarrollar nuevas estrategias para los objetivos o sensaciones que quiera transmitir. Aun así, existen técnicas reconocidas por la comunidad y ampliamente extendidas para conseguir resultados concretos que refuerzan los principios del gamefeel anteriormente expuestos:

ADSR: describe la **modulación** de un **parámetro** a lo largo del **tiempo** en cuatro fases: **Attack-Decay-Sustain-Release**. En la fase de *Attack* el parámetro empieza a incrementarse rápidamente hasta alcanzar su valor más alto, luego, la *Decay* describe una pérdida de fuerza disminuyendo el valor, para luego pasar a la etapa de *Sustain* en la que el parámetro se mantiene, por último, en la fase de *Release*, el parámetro vuelve a bajar hasta su valor mínimo. Es posible que en algunas aplicaciones se ignore la fase de *Decay*. Normalmente esta técnica es empleada para mapear el input a un parámetro que controla movimientos concretos, la velocidad de las animaciones a reproducir o las propias acciones que los personajes realicen.

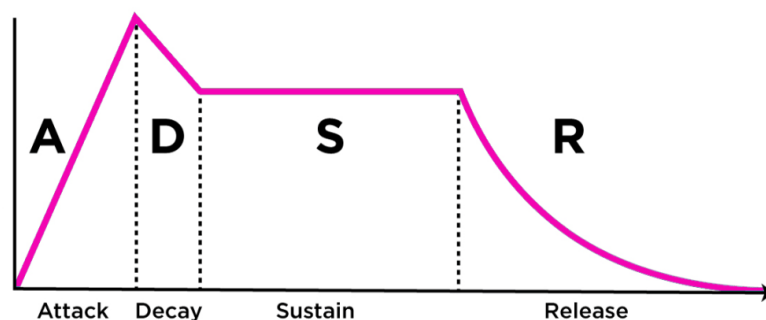


Ilustración 12- Ejemplo de una ADSR

Tweening: aunque en animación este término sea utilizado para referirse al proceso de generar frames intermedios entre dos poses (*in-betweening*) [45], aquí se utilizará su definición más genérica como la **interpolación** entre **dos valores** en un **tiempo** determinado. Gracias a esta técnica se pueden obtener movimientos orgánicos y atractivos con el uso de las **easing functions**, evitando así movimientos lineales y monótonos. Estas funciones son también utilizadas en las fases que componen las ADSR.

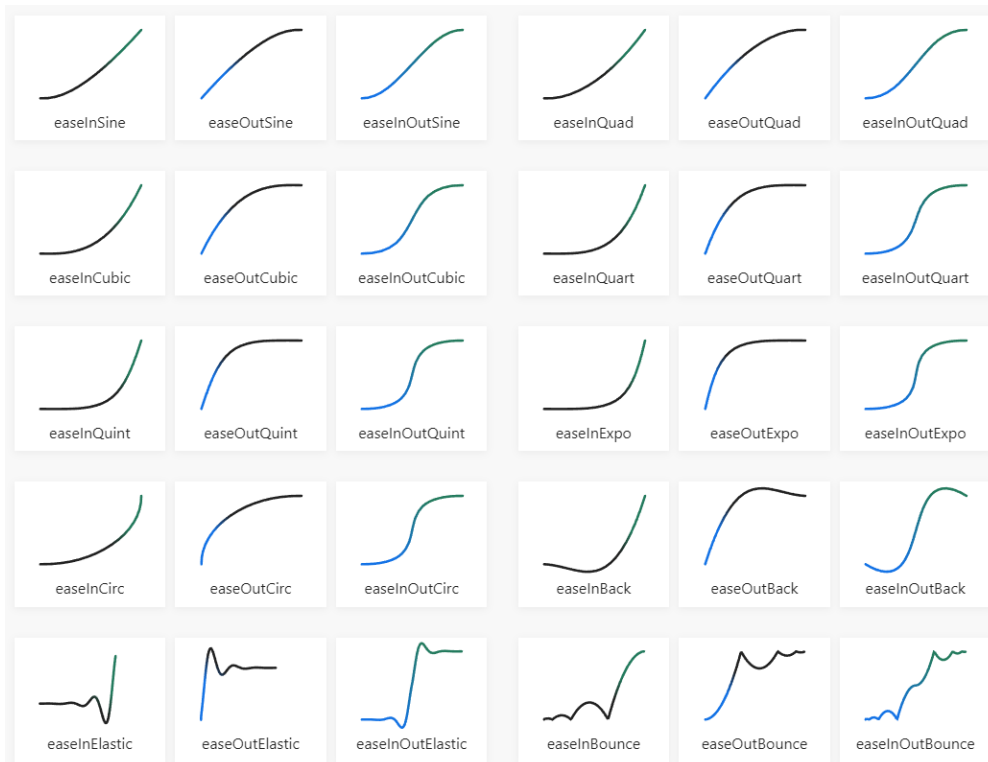


Ilustración 13- Ejemplos de Easing Functions [46]

Input buffering: técnica utilizada para **aportar** una **sensación** de **respuesta inmediata**, basada en **guardar** el **input** del **jugador** para ejecutarlo más tarde [47]. Mientras el personaje jugable está ocupado realizando una acción, todo input generado por el jugador será recordado por el personaje, para que luego este pueda ejecutar las acciones pertinentes en cuanto se encuentre disponible. Esto, sumado a la cancelación de animaciones, dotan al sistema de una gran fluidez y receptividad.

Coyote Time: utilizado sobre todo en juegos de plataformas, se basa en dar al jugador una ventana de **tiempo extra** para poder **saltar después** de haber **abandonado** una **plataforma** [47]. Es un claro ejemplo de un método que refuerza un resultado predecible, ya que, aunque el avatar sea detectado fuera de la plataforma por el juego de una manera milimétrica, el jugador no lo siente de esa manera y espera que el personaje pueda saltar. Así pues, al aplicar esta técnica los controles se sienten obedientes cooperando con la intención del jugador obteniendo el resultado esperado.

Hit pause/Hit stop: utilizado sobre todo en sistemas de combate para reforzar el impacto al golpear, cuando un **ataque** conecta la **animación** de ataque del **atacante** y la animación de

dañado del **atacado** se **pausan** simultáneamente durante unos frames para dar una sensación de **potencia** a la vez que una respuesta atractiva y satisfactoria [48].

Game slow: todo el **mundo virtual** procede a moverse a **cámara lenta**, en ciertas ocasiones el personaje jugador puede verse inafectado pudiéndose mover libremente a la velocidad normal, usualmente utilizado para dar una sensación de poder y superioridad en ciertos momentos [49]. A este efecto también se le conoce como “bullet time”. Un caso particular del *game slow* es el **game freeze** en el que el mundo virtual se queda completamente **congelado** [50].

Camera shake: pequeñas **variaciones** muy **rápidas** de **posición** y **rotación** de la **cámara** durante un período de tiempo, esta técnica es muy simple, pero a la vez muy eficaz para vender la acción que está ocurriendo en pantalla.

Cámara dinámica: se trata de la utilización de diversas **cámaras** que relevan a la cámara principal con el propósito de **resaltar acciones** concretas, mejorando así la experiencia al ofrecer nuevas perspectivas que aumentan la sensación de **dinamismo** y novedad.

Gamepad rumble: a diferencia del teclado y ratón el **mando** ofrece una capacidad háptica muy superior gracias a su **vibración**. Con ella es posible transmitir de una forma más clara las acciones que ocurren, así como dar una sensación táctil al jugador frente a diferentes situaciones o escenarios, como al golpear o al verse dañado [51]. Su uso también puede verse extendido a las cinemáticas del juego.

Adicionalmente, aunque cada juego sea distinto, el pulido representa un aspecto general presente en todo juego al que se le debe prestar gran atención para conseguir un buen gamefeel. En él entran todos los aspectos que un juego pueda llegar a abordar: mecánicas, animaciones, diseño de niveles, artes, efectos de sonido y música, penalizaciones y recompensas...

Casos de estudio

En este apartado se presentan diversos casos de estudio en referencia al movimiento y a efectos aplicados en el combate. En ellos se trata cómo diferentes títulos abordan aspectos comunes con enfoques concretos, para luego así poder tener un repertorio de diferentes soluciones en las que basarse. A fin de cuentas, y coloquialmente hablando, este apartado responde a la pregunta de “¿Cómo se hace ‘x’ cosa en este videojuego?”

Darksiders 3

Movimiento:

- El **esquive** se trata de un **dash**, se pueden realizar hasta **tres esquivas** seguidas, la última deja al jugador vulnerable durante un breve período de tiempo.



- Es posible **esprintar**, pulsando el joystick izquierdo, al hacerlo se efectúa un ligero **FOV kick** para dar sensación de velocidad. El esprint se pierde al realizar un dash, pero se mantiene después de saltar.
- Asistencia para **trepar bordes automáticamente**, el personaje no se queda colgado esperando al input del jugador. Aun así, si el jugador voluntariamente quiere dejarse caer por un borde el personaje se quedará colgado esperando el input para volver a subir o soltarse.

Combate:

- En el **contraataque** se aplica un efecto de **game slow** más **camera shake** para enfatizar que se ha realizado correctamente la acción.
- **Vibración de mando** al **recibir daño** y a la hora de **conectar un golpe**, no necesariamente en un enemigo ya que también se aplica a objetos rompibles.
- **Camera shake** a la hora de realizar algunos **ataques** sin necesidad de que estos conecten. También se aplica al recibir daño de ataques potentes del enemigo.
- **No** se percibe el uso de **hit pause**.

Astral Chain

Movimiento:

- **Sin doble salto**, adicionalmente tampoco existe ninguna ayuda automática para escalar bordes.
- Se pueden realizar hasta **dos esquivas**, una primera **corta** y una segunda más **larga** en forma de voltereta. Después del segundo hay un periodo en el que se deja expuesto al jugador.
- El **esprint** se hace pulsando el *joystick* izquierdo y se pierde después de realizar una esquivas o un ataque.

Combate:

- **Camera shake** en cada **impacto** que conecta con el enemigo, algunos ataques presentan también un poco sin necesidad de conectar.
- **Cámaras dinámicas** al realizar movimientos especiales con las Quimeras (acompañante).
- **Hit pause** claramente **apreciable** al conectar **ataques** con la Quimera, dando la sensación de que el acompañante realiza ataques más potentes que los del propio



personaje. Siendo un juego tan basado en el uso de acompañantes este efecto refuerza el tema principal.

Nier Automata

Movimiento:

- El **esprint** se realiza una vez realizado el dash y se mantiene después de saltar. Existen animaciones diferentes para cuando el personaje salta diferenciando si ha estado esprintando o no. Al mantener el esprint se puede apreciar un **aumento progresivo** en la **velocidad**.
- Hay un número de **esquivas ilimitado**, un dash puede interrumpir otro dash, con un período de vulnerabilidad muy pequeño entre ellos. También se puede ejecutar un **dash** en el **aire**.
- Existe una **asistencia** automática para **escalar bordes**. Si el jugador se deja caer voluntariamente esta asistencia intervendrá, el personaje nunca se queda colgando.
- El **acompañante** asiste en el movimiento aéreo del personaje habilitando al jugador **planear**.

Combate:

- Al **conectar** un **golpe** se producen **partículas** de chispas y un ligero **camera shake**, sin embargo, no se produce ninguna vibración en el mando. Al derrotar a un enemigo este explota y se realiza un efecto de *motion blur* en el contorno de la pantalla.
- Cuando el **jugador** es **golpeado**, se produce una **vibración** de **mando** dependiendo de la magnitud del golpe, además de un ligero **camera shake** acompañado de efectos de postprocesado (aberración cromática y desenfoque). Un sonido característico se reproduce cada vez que el jugador recibe daño.
- Al encontrarse con **poca vida** la imagen pasa a **escala** de **grises** y un **efecto** de **viñeta** rodea toda la pantalla. Un **sonido** de **alarma** se activa para indicar al jugador que se encuentra muy cerca de morir.
- En ciertos ataques se utilizan **cámaras dinámicas** para enfatizar la acción, por ejemplo, al realizar un **smasher** la cámara se mueve rápidamente para realiza un plano en picado.
- **No** se percibe el uso de **hit pause**.



Bayonetta

Movimiento:

- Aunque **no** haya **asistencia** a la hora de **trepar bordes**, es posible realizar un **salto adicional** en forma de golpe en la pared para subir más alto y alcanzar plataformas elevadas.
- **Sin esprint**, pero gracias a la buena velocidad por defecto del personaje y a que los escenarios son de un tamaño modesto esto no resulta un problema. Cuando se avanza en el juego se desbloquea una forma de animal (pantera) en la que el personaje puede transformarse para desplazarse más rápidamente.
- El jugador puede realizar hasta **cinco dashes** seguidos, el último tiene una animación de recuperación que deja expuesto al jugador durante un tiempo considerable.
- **Sin desplazamientos aéreos** de ningún tipo, aunque al progresar en el juego se desbloquea una forma de animal (cuervo) que permite al jugador planear por el escenario.

Combate:

- Al **conectar** un **golpe** se produce un leve **camera shake** junto con **vibración** del **mando**. Algunos ataques más potentes producen **motion blur** en la zona de impacto además de un notable **hit pause**
- Al ser **golpeado** se produce un efecto de **viñeta rosada**, acompañada de **camera shake** y **vibración** de **mando**, adicionalmente la interfaz gráfica donde se sitúa la vida también tiembla.

Como conclusiones se pueden extraer que tanto el **camera shake** como el **gamepad rumble** están **presentes** en **todos** los **títulos** estudiados, son técnicas universales, sobre todo a la hora de hablar de intercambio de golpes entre el jugador y los enemigos, ya que estos ayudan a comunicar claramente cuando el jugador inflige daño y cuando lo recibe. Otro ejemplo de esto son los **efectos** de **postprocesado**, **adaptados** a la **estética** del juego en concreto.

3.3 Principios de animación

La animación forma una parte clara del *gamefeel* y más aún si se enmarca en un sistema de combate. Por ello, los principios de animación marcados por Ollie Johnston y Frank Thomas en su libro “**The illusion of life: Disney Animation**” [52], serán de gran utilidad a la hora de modificar o editar las animaciones que utilizadas en el combate, para así poder perfilar el ritmo, la sensación y la comunicación de cada uno de los ataques involucrados en el sistema, hasta conseguir el resultado deseado.

A continuación, se presentan los doce principios de la animación:

Estirar y encoger: utilizado para dar una **sensación de peso y flexibilidad**, el **objeto** a dibujar se **alarga** y se **achata** siguiendo el movimiento que se quiera comunicar. Este principio también es aplicado a la hora de realizar expresiones faciales, exagerándolas para lograr más expresividad en los rostros de los personajes.



Ilustración 14- Dos ejemplos de "estirar y encoger" en las animaciones de los personajes del juego Overwatch: Zenyatta (arriba) y Cassidy (abajo)

Anticipación: sirve para **establecer el comienzo** de una **acción**, indicando a los espectadores que algo va a ocurrir a la vez que muestra de antemano la **intención del movimiento**. Por ejemplo, cuando un bateador gira su torso y lleva el bate detrás de su cabeza con la intención de batear. También la ausencia de anticipación o la no correlación entre anticipación e intención de la acción pueden utilizarse como recurso cómico o de sorpresa. Este principio se ve claramente en la fase de **wind-up** de la anatomía de un **ataque** descrita al principio de este capítulo.



Puesta en escena: se trata de **presentar** una **idea** (una acción, una expresión, un personaje...) de tal forma que esta sea total e **inequívocamente clara**. Para ello, es necesario **dirigir** la **atención** hacia lo más importante dentro de la escena **evitando detalles innecesarios**. Para conseguir esto se puede hacer uso de la composición dentro de propia escena, el posicionamiento de personajes, la utilización de luces y sombras o la posición y el ángulo de la cámara. El uso de **cámaras dinámicas** se sustenta en este principio.

Animación directa y pose a pose: se refiere a los dos enfoques principales para animar. La **animación directa** se basa en **dibujar en orden** los **frames** o poses que componen una **acción**, de principio a fin. Mientras, la **animación pose a pose primero** marca y dibuja unos **fotogramas clave** (keyframes) en el movimiento para **más tarde rellenar** los huecos entre estos con **poses intermedias**, esto es, el **tweening** del cual se hablaba en las técnicas de gamefeel. Cada enfoque tiene sus pros y contras y normalmente se utiliza una combinación de ambas, utilizando la pose a pose para estructurar el movimiento y la animación directa para rellenar el espacio entre los keyframes.

Acciones complementarias y superpuestas: utilizadas para conseguir animaciones más realistas. Las acciones complementarias se basan en que las **piezas** no pertenecientes al cuerpo que genera el movimiento, pero que sí se encuentran **vinculadas** a este, deben de **seguir el movimiento** con cierto **retardo**, como si se tratase de una acción en cadena. Un caso concreto y muy extendido de acciones complementarias son las **físicas de pelo o telas** encontradas en cualquier personaje actual. Por otro lado, las **acciones superpuestas** se refieren a cuando **partes** de un mismo **cuerpo** se mueven a **ritmos y velocidades diferentes**.

Arcos: para dar una mayor **naturalidad** a los **movimientos**, la mayoría de las acciones describen **arcos** en sus **trayectorias**, esto es aplicable sobre todo al rango de movimiento de las extremidades de un personaje, pero el mismo principio también es utilizado en los cambios de velocidad a la hora de describir un movimiento. Este principio coincide con la máxima de mantener y lograr un **movimiento orgánico** en el **gamefeel**.

Acelerar y desacelerar: los movimientos necesitan **tiempo** para **acelerar y desacelerar**, contribuyendo a obtener un movimiento orgánico. Para obtener este resultado se realizan un **mayor número** de **poses** en los momentos en los que un cuerpo aceleran o desacelerar otorgándoles así más tiempo en la animación.

Acción secundaria: se basa en añadir **acciones** que **enfaticen** y respalden la **acción principal** que se está llevando a cabo, por ejemplo, en un ciclo de caminar el balanceo de brazos como acción secundaria puede indicar si el personaje está enfadado o contento. Aun así, si la acción secundaria quita protagonismo o no acompaña adecuadamente a la acción principal es mejor prescindir de ella.

Sincronización: referido al **número** de poses o **frames** usados en cualquier movimiento, el cual determina la cantidad de **tiempo** que tomará en **total**. La cantidad de frames necesarios depende de lo que se quiera transmitir, si, por ejemplo, el movimiento consiste en un golpe enorme, existirán pocos o ningún frame adicional entre las dos poses más extremas, la de anticipación y la propia del golpe, mientras que si se quiere dar detalle a un movimiento



para hacerlo fluido el número de frames aumentará considerablemente entre estas dos poses de principio y fin de la acción.

Exageración: usada para **enfatar** y vender la **acción**, si un personaje tiene que estar triste hay que presentarlo aún más triste, si tiene que estar preocupado, hacerlo aún más preocupado... **No** se basa en **exagerar** el **dibujo** como tal, deformando las proporciones del personaje, **sino** en acentuar y **dramatizar** la **acción** que se está llevando a cabo.

Dibujo sólido: se refiere a la técnica a la hora de dibujar, al animador necesita ser un **dibujante diestro** capaz de aplicar las bases del dibujo en la animación: perspectiva, luces y sombras, anatomía, composición...

Atractivo: no se atiende al atractivo de una acción como en el caso del gamefeel sino al personaje en sí. Se identifica con el carisma, que el **personaje resulte real, convincente e interesante** para el espectador.

Si bien es cierto que **algunos principios** como el de “Dibujo sólido” o “Atractivo” **no pertenecen** al **dominio** de este **trabajo**, ya que son muy específicas de otros campos como diseño de personajes o arte 2D, el resto de estos van a ser de gran utilidad para realizar modificaciones que realcen la acción que se está llevando a cabo en la animación.

3.4 Animación en Unity

A continuación, se presentan las partes y características fundamentales de **Mecanim**, el sistema de animación que incorpora Unity.

Animation Clips

Las **animaciones** en Unity son **almacenadas** como **Animations Clips** (.anim), en estos clips se puede tanto visualizar como modificar todas los frames y propiedades de la animación que representan. Unity permite importar animaciones desde fuentes externas además de crear **Animation Clips** propios dentro del motor haciendo uso de la ventana de **Animation**.

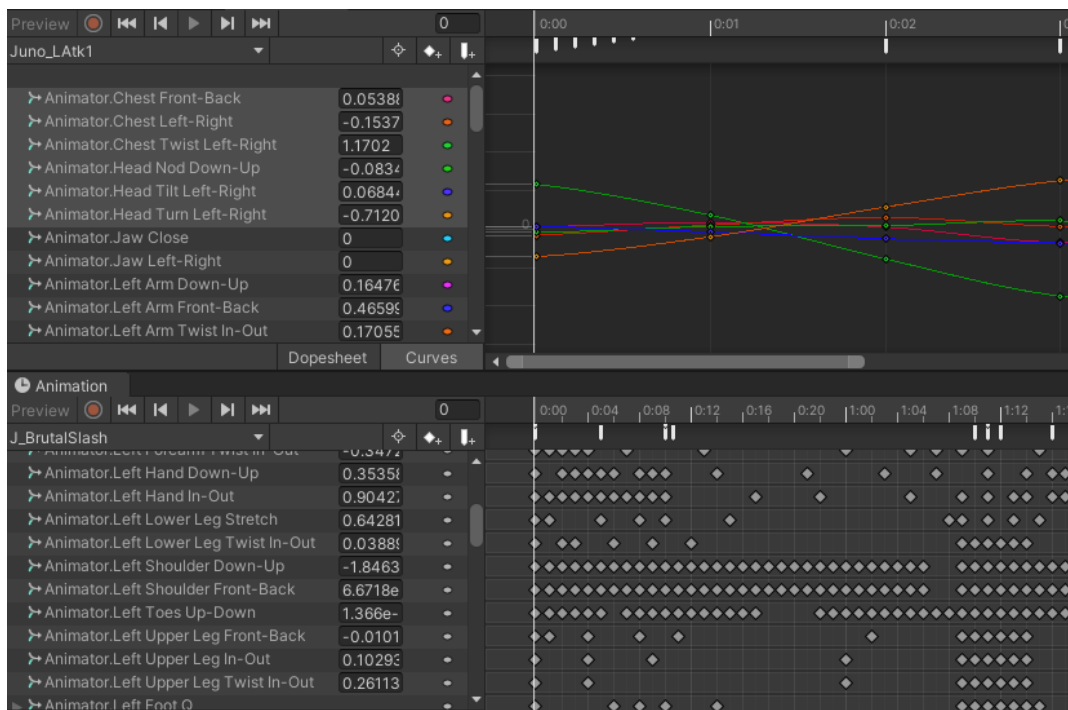


Ilustración 15 - Diferentes visualizaciones de una animación en la ventana Animation: con curvas (arriba) o con keyframes (abajo)

Existen diferentes tipos de configuraciones para un Animation Clip:

- **Legacy:** antes de *Mecanim*, Unity utilizaba un sistema de animación mucho más sencillo. Aunque este sistema sigue activo por **retrocompatibilidad**, se trata del método más rápido en cuestión de rendimiento, además de no necesitar apenas andamiaje para reproducir una animación. Las animaciones *legacy* son las únicas que no necesitan de *animation controllers* o avatares para realizar su función, ideal para evitar la sobrecarga que estos elementos provocan para simples objetos animados cosméticos que usan unas pocas animaciones.
- **Humanoid:** animaciones destinadas a **personajes humanoides**, los cuales **comparten** una **estructura de huesos común** gracias al sistema *Avatar*. Debido a esta configuración común del esqueleto es posible mapear animaciones de un personaje

humanoide a otro, lo que posibilita el **retargeting** y el uso genérico de **inverse kinematics (IK)** de una forma sencilla.

- **Generic:** utilizada para todas las demás animaciones que no sean de personajes humanoides. Se usan para objetos cuyo esqueleto difiere con la estructura común que define el Avatar Humanoide de Unity, como puede ser desde animales hasta objetos con ciertas partes mecánicas animadas.

En este trabajo se utilizarán extensamente animaciones *Humanoid* debido a la cantidad de personajes humanoides existentes en el proyecto. A la hora de importar una animación como *Humanoid*, aparecen las siguientes opciones para modificar la configuración del *Animation Clip* o añadir más información al mismo [53]:

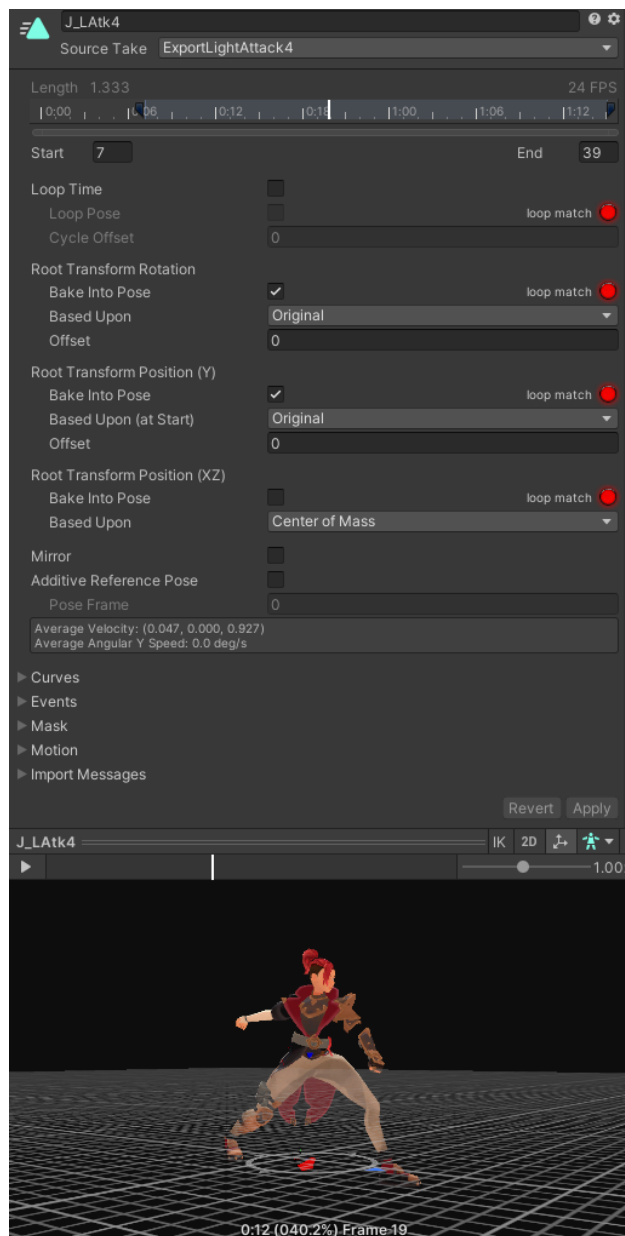


Ilustración 16 - Ventana de configuración de animaciones humanoides

- **Start/End:** utilizados para definir la longitud del clip indicando los frames de principio y fin.
- **Loop Time:** indica si la animación se reproduce en bucle.
- **Root Transform:** apartados pertenecientes al hueso raíz del esqueleto para diferentes ejes de traslación (*root transform position (Y/XZ)*) o para la rotación (*root transform rotation*):
 - **Bake into Pose:** indica si va a ignorar o no la información proporcionada por la *Root Motion* para mover al personaje por el espacio.
 - **Based Upon:** indica un punto de referencia sobre el que el personaje se mueve en el espacio. Existen diferentes opciones como *Original, Body Orientation, Center of Mass* o *Feet*.
 - **Offset:** define un desplazamiento sobre ese punto de referencia.
- **Mirror:** realiza la simetría de la animación.
- **Additive Reference Pose:** define el *frame* de referencia para animaciones aditivas. Las transformaciones del esqueleto que definen el resto de frames de la animación se tratan de forma relativa a la pose indicada.
- **Curves:** utilizadas para modificar parámetros a lo largo de la animación, estos parámetros podrán ser accedidos a través del *Animation Controller* o por medio de código.

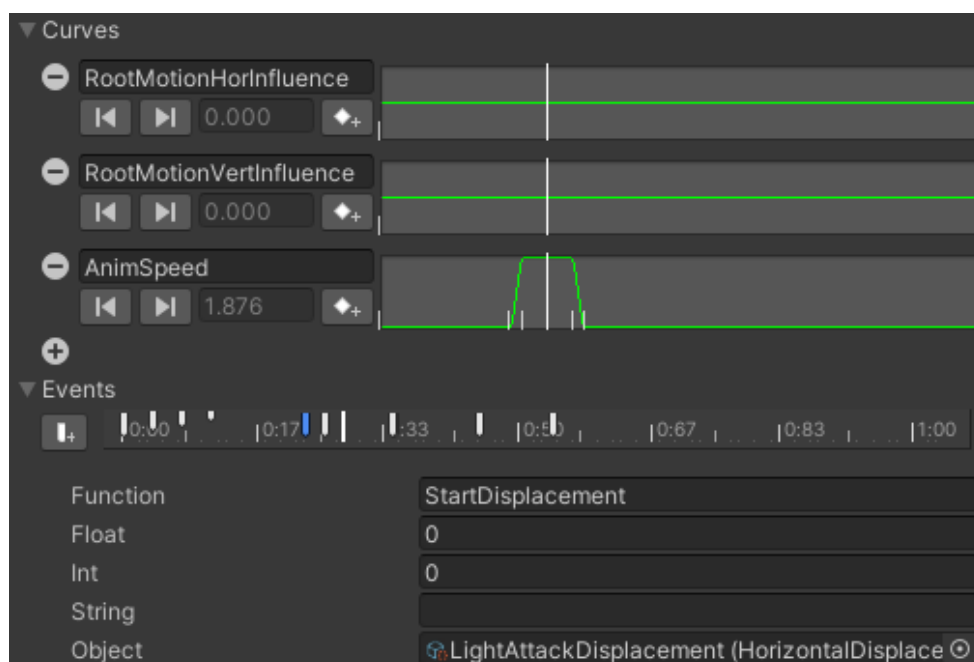


Ilustración 17 - Ejemplo de uso de curvas (arriba) y eventos (abajo) en una animación

- **Animation Events:** eventos de animación que activan la funcionalidad indicada explícitamente cuando la animación llega a un determinado frame. Son capaces de aceptar varios argumentos desde tipos primitivos hasta objetos.
- **Mask:** indica los huesos que serán tenidos en cuenta o a la hora de realizar la animación. En el caso de personajes humanoides con huesos extras a los definidos en la estructura común que aporta Unity, será aquí donde tengan que activarse para que sean animados.

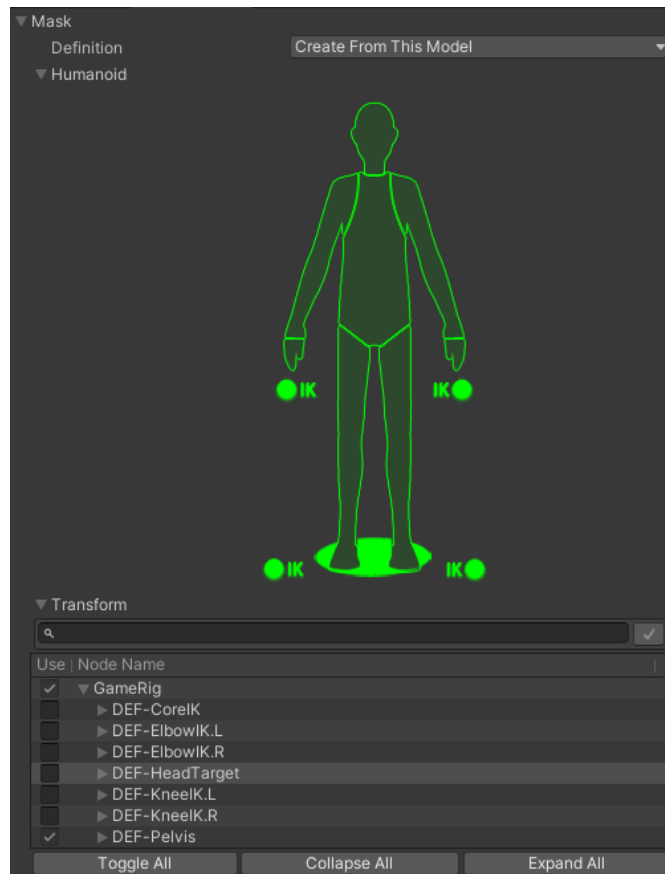


Ilustración 18 - Ventana de modificación de la máscara

Animator Controllers

Los *Animator Controllers* permiten **organizar** y **mantener** un conjunto de **animaciones** pertenecientes a un personaje u objeto. Estos contienen referencias a todos los *Animation Clips* necesarios y se encargan de **gestionar** los diversos **estados** de animación y las **transiciones** existentes entre ellos en una **máquina de estados**. Cualquier *Animator Controller* se encuentra formado por los siguientes elementos:

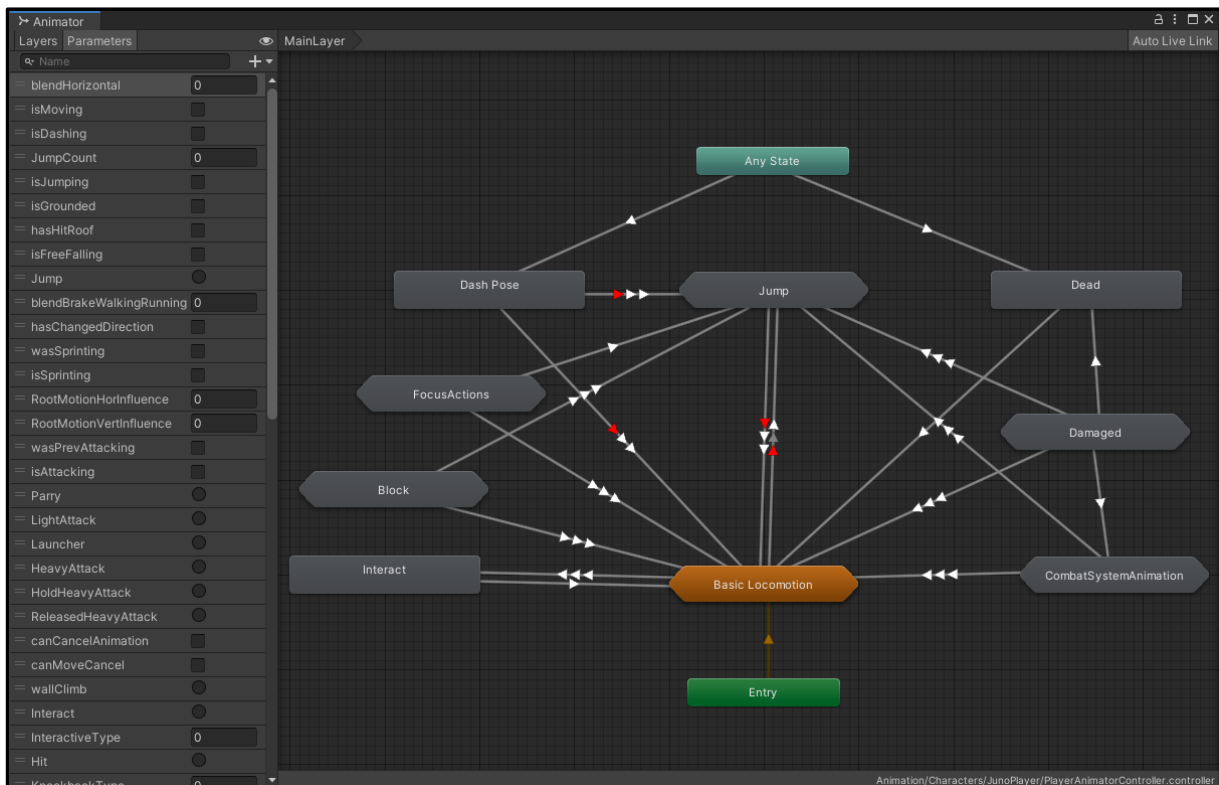


Ilustración 19 - Ejemplo de Animator Controller

Estados

Los estados forman los **bloques base** que forman el grafo de animación, estos mantienen una **referencia** a las **animaciones (Motion)** que deben ser reproducidas cuando el nodo se encuentra activo.

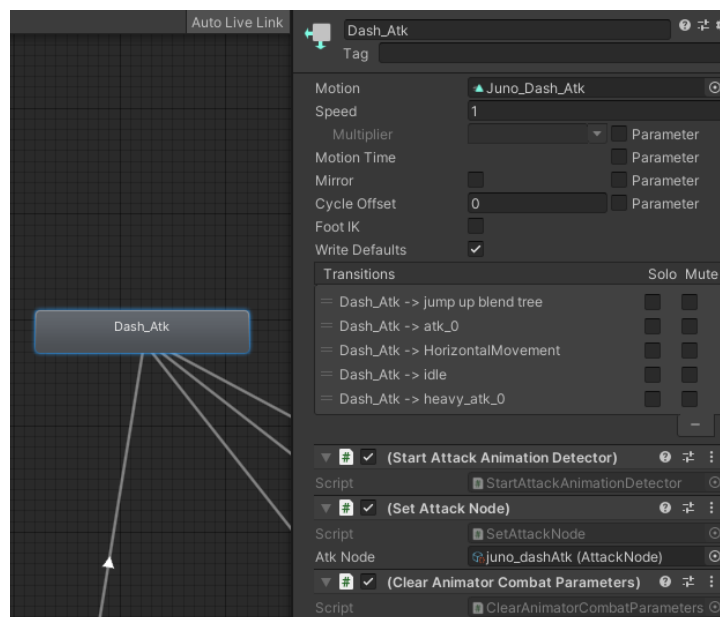


Ilustración 20- Ejemplo de un Estado

Una tarea común a la hora de implementar animaciones se trata de **mezclar** dos o más **movimientos** similares. Uno de los ejemplos más destacados se encuentra en las animaciones de locomoción, en las que se mezclan las animaciones de caminar, correr o esprintar de acuerdo con la velocidad y dirección del personaje. Para este tipo de funcionalidad se hace uso de unos estados especiales llamados **Blend Tree**, estos permiten mezclar de forma fluida múltiples animaciones incorporando parte de todas ellas en diferentes proporciones [54].

La **influencia** de cada uno de los **movimientos** suministrados al *Blend Tree* que contribuyen al efecto final es controlada por medio de **parámetros** modificables durante la ejecución del juego. A valores diferentes de estos parámetros, diferentes influencias en los movimientos y, por tanto, una mezcla resultante distinta.

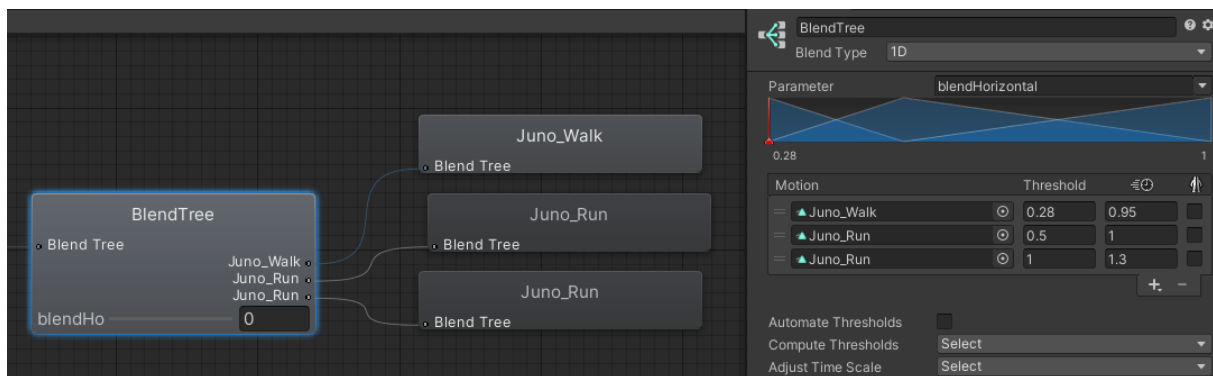


Ilustración 21- Ejemplo de Blend Tree

Los *Blend Tree* son utilizados sobre todo a la hora de **combinar animaciones** que dependen de unos **parámetros comunes**, como pueden ser animaciones faciales (expresiones) o de locomoción (velocidad y dirección). Existen diferentes tipos de mezcla (*Blend Type*) dependiendo del número de parámetros: *1D Blending* (1 parámetro), *2D Blending* (2 parámetros) [55], *Direct Blending* (>2 parámetros).

En diversas ocasiones, cuando un *Animation Controller* alcanza cierto grado de complejidad debido al gran número de animaciones que precisa, se recurre al uso de **Submáquinas de estados**. Su función consiste en **colapsar** un grupo de **nodos** en un solo elemento, optimizando el espacio en el grafo y mejorando la organización al juntar animaciones que mantienen una relación entre ellas (animaciones de locomoción, animaciones de combate, etc). Al acceder a este nodo, todos los estados vuelven a desplegarse en una nueva ventana mejorando así la visualización.

Transiciones

Las transiciones definen **cómo** y **cuándo** se produce la mezcla o el **paso** de un **nodo** de animación **a otro**. En ellas no solo se define cuánto tiempo debe durar la transición en sí, sino también qué condiciones son necesarios para activarse. Estas **condiciones** se establecen en relación con los parámetros definidos en el *Animator Controller*.

Existen diferentes propiedades que definen la configuración de la transición como la duración de esta (*Transition Duration*), si debe activarse automáticamente llegado a cierto punto de la animación (*Has Exit Time* y *Exit Time*) o si existe algún desplazamiento de tiempo para comenzar la reproducción del estado de destino (*Transition Offset*).

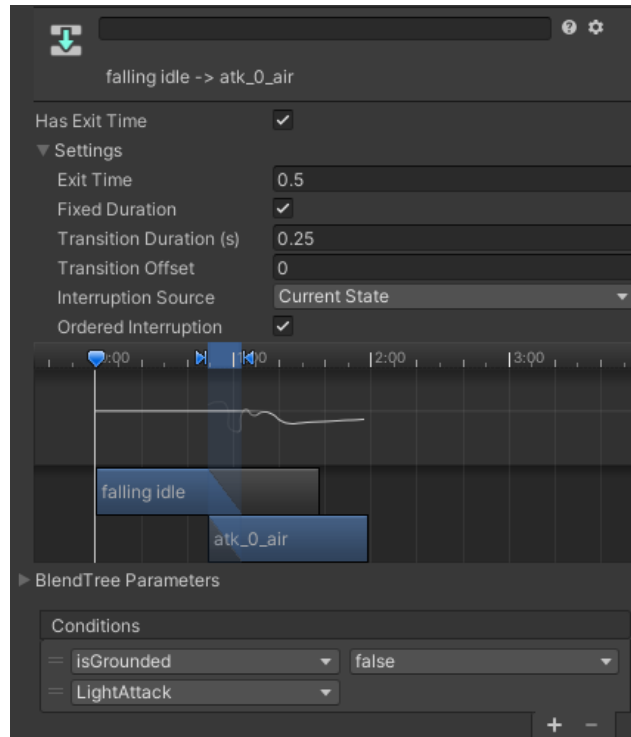


Ilustración 22- Ejemplo de transición

A la hora de gestionar diversas transiciones, caben destacar los campos de *Interruption Source* y *Ordered Interruption*. Estos parámetros definen si una **transición** que se encuentre **en curso** puede ser **cancelada** a favor de otra transición más **prioritaria** en el momento en que esta último cumple las condiciones para activarse:

- **Interruption Source:** dicta si una transición puede ser interrumpida por otra, adicionalmente marca el orden de prioridad según la opción elegida. Este orden se da en bloques formados por las transiciones que mantienen los estados involucrados en la transición.

Desde las opciones de *Interruption Source* no se puede modificar la prioridad dentro de estos bloques, solo el orden de estos, para establecer prioridades dentro de un bloque es necesario acudir al orden en el que las transiciones están dispuestas dentro de un estado (de arriba a abajo, de mayor a menos prioridad)

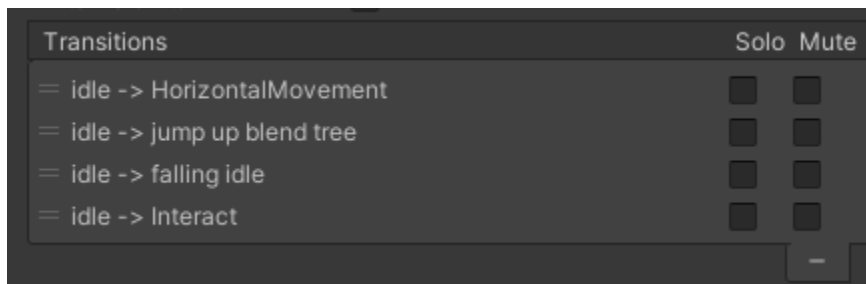


Ilustración 23- Orden de prioridad de las transiciones establecido dentro de un estado

Es importante destacar que el bloque de Transiciones perteneciente al estado de **Any State** es siempre el más prioritario de todos y se colocará siempre por delante del resto de bloques independientemente de la *Interruption Source* elegida.

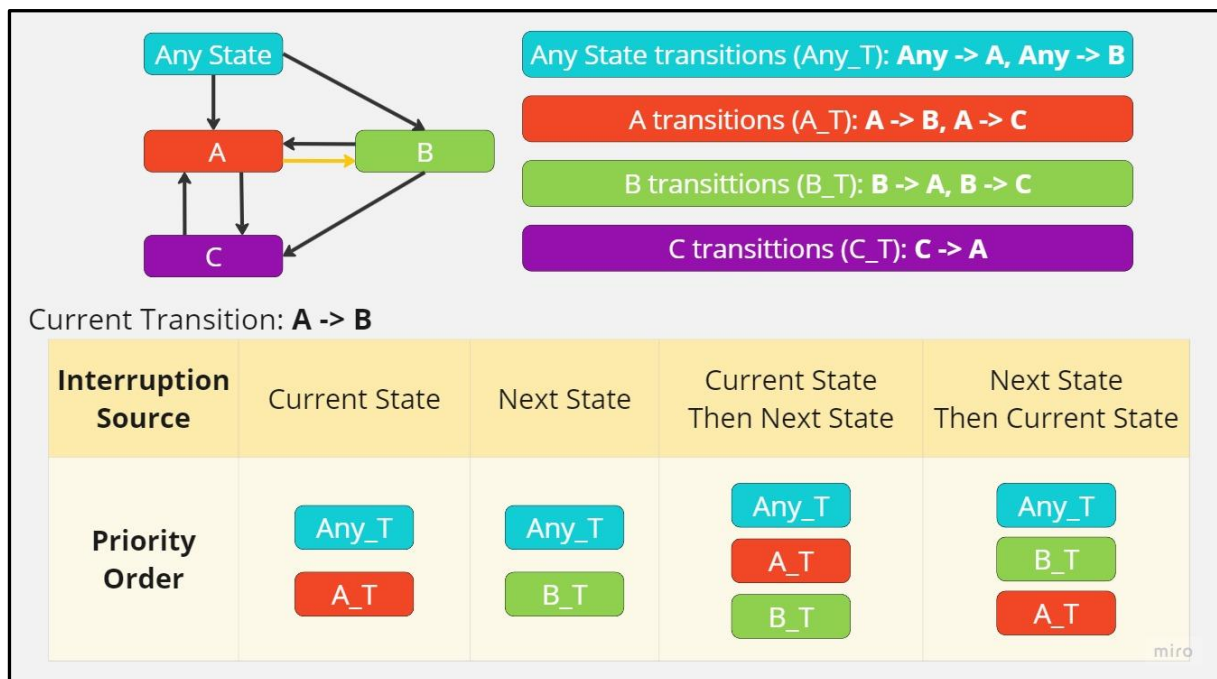


Ilustración 24- Orden de prioridad según la Interruption Source

- **Ordered Interruption:** simplemente marca si debe atenderse a las prioridades establecidas o si por el contrario cualquier transición que pertenezca a los bloques indicados por la *Interruption Source*, o al *Any State*, puede interrumpir a la transición actual, independientemente del orden.

En última instancia se encuentran las **condiciones**, en ellas se definen los **requisitos** necesarios en base a los **parámetros** para que la **transición** se **active**. Todas las condiciones declaradas en una transición deben cumplirse para que esta sea accionada.

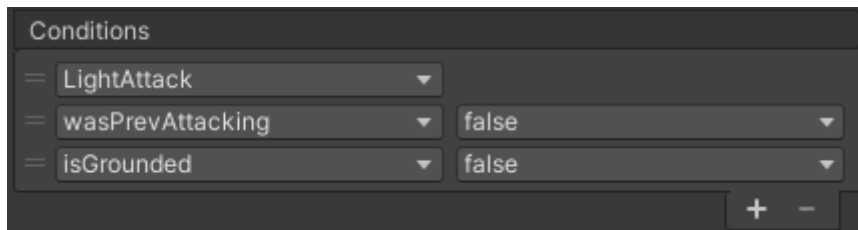


Ilustración 25- Ejemplo de condiciones

Parámetros

Simplemente son **variables** definidas dentro de un *Animator Controller*, estas pueden ser accedidas y modificadas desde código o a través de la propia animación. Estas variables son utilizadas tanto a la hora de declarar condiciones para las transiciones como para definir la mezcla en un Blend Tree.

Existen cuatro tipos de parámetros, los correspondientes a los tipos primitivos de datos de **Integer**, **Float** y **Bool**, y uno adicional llamado **Trigger**. Este último funciona como un parámetro booleano que es restablecido por el propio *Animator Controller* una vez es utilizado (consumido) en una transición.

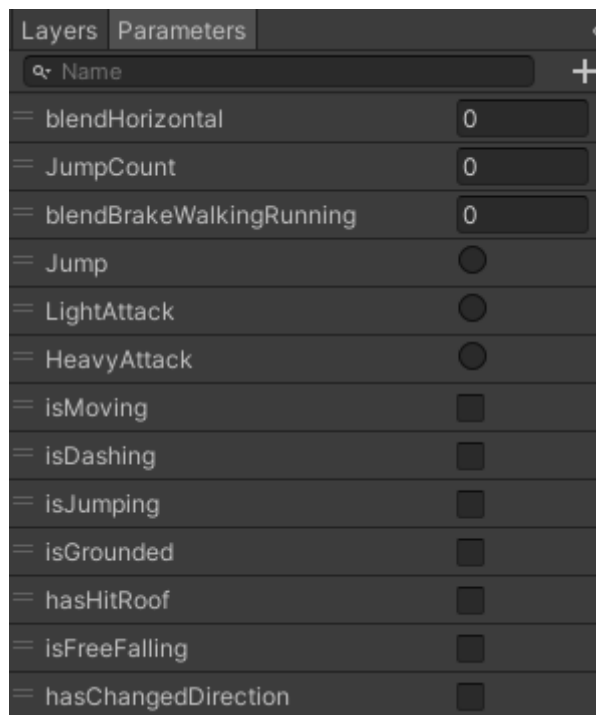


Ilustración 26- Ejemplo de algunos parámetros

Capas

Las capas de animación son utilizadas para **gestionar** diferentes **partes** del cuerpo o **esqueleto** del objeto animado [56], aunque también pueden ser utilizadas para **cambiar** un conjunto de **animaciones** que **conservan** la misma **lógica**, como por ejemplo el cambio de animaciones de locomoción normal a locomoción en combate. Las condiciones, transiciones y estados de este subconjunto de la máquina siguen siendo el mismo, lo único que varía son

las animaciones en sí. Las capas funcionan como una nueva máquina de estados que corren en paralelo, evaluando transiciones y teniendo un estado activo.

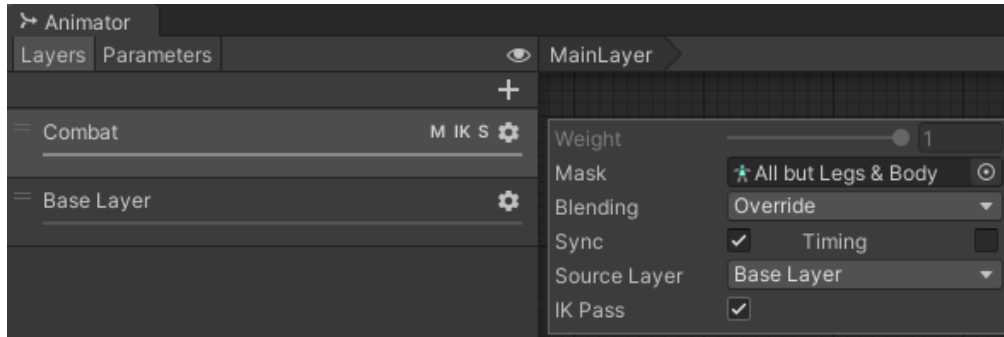


Ilustración 27 - Opciones de configuración de una capa

A la hora de crear una capa se pueden definir los siguientes parámetros:

- **Weight:** define la influencia de la capa, esto es, el grado en el que participa el nodo activo de la capa en la mezcla final junto con las otras capas activas. Este parámetro puede ser modificado a través de código.
- **Mask:** definen los huesos del esqueleto considerados a la hora de animar.
- **Blending:** especifica cómo se mezclan las animaciones de la capa con respecto al resto. Existen dos opciones disponibles:
 - o *Override:* reemplaza por completo las animaciones de capas inferiores.
 - o *Additive:* utiliza animación aditiva con respecto a las capas inferiores. Las deformaciones del esqueleto de la animación de la capa actual se añaden a las animaciones de las capas inferiores.
- **Sync:** crea una copia de la máquina de estados de animación perteneciente a la capa indicada en *Source Layer*. Esta opción resulta ideal para implementar conjuntos de estados que poseen la misma lógica, pero diferentes animaciones (locomoción normal, locomoción de combate, locomoción de herido).
- **Timing:** debido a que las animaciones de una capa que está sincronizada con otra con otra pueden diferir en tiempo, esta opción está destinada a ajustar la longitud de las animaciones de la capa sincronizada para que duren lo mismo que las animaciones de la capa referenciada.
- **IK Pass:** define si la capa va a hacer uso de Inverse Kinematics en tiempo de ejecución.

Componente Animator

Para hacer uso del *Animation Controller*, y finalmente animar un objeto o personaje cualquiera, es necesario añadir un componente *Animator* al *GameObject* pertinente. En este componente se especifica el *Animation Controller* encargado de gestionar las animaciones y su lógica, además del avatar en caso de que las animaciones lo precisaran.

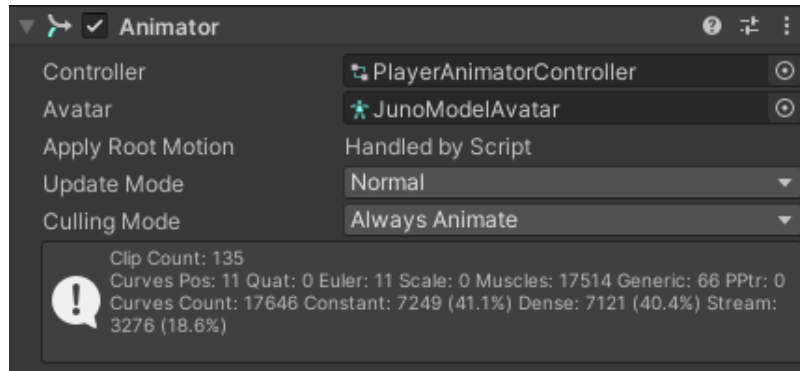


Ilustración 28 - Componente Animator

Con este último elemento se cierra el sistema de animación de Unity, el cual puede verse resumido en el siguiente diagrama:

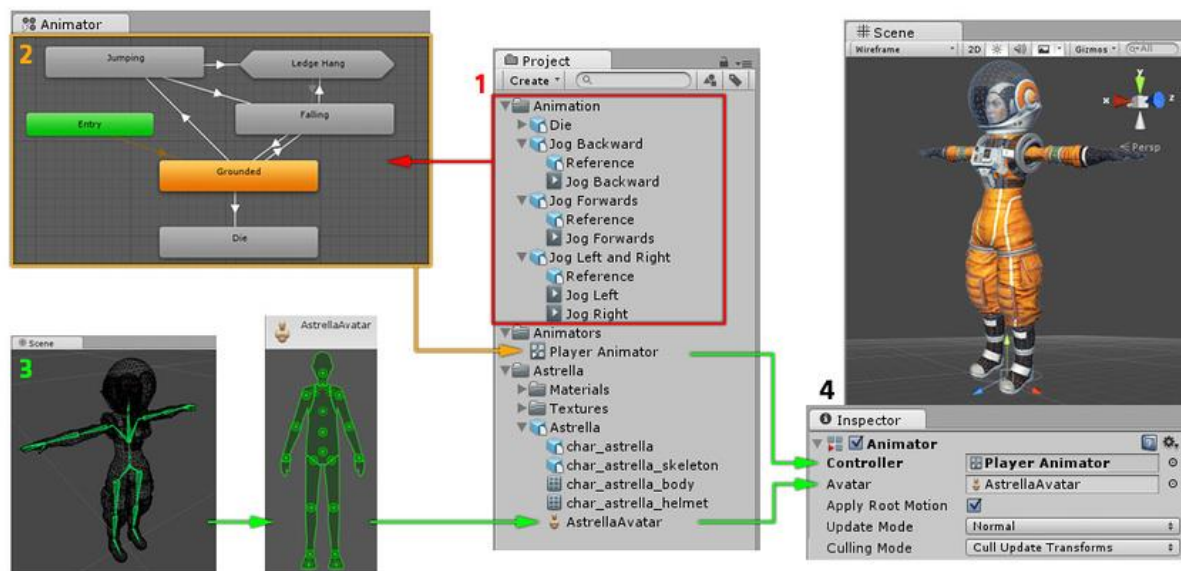


Ilustración 29- Diagrama del marco de trabajo de animación en Unity [57]

Retargeting de animaciones

En muchas ocasiones es necesario **reutilizar animaciones** ya hechas a mano o incluso adquirir packs externos de animaciones para poder trabajar más rápidamente y de manera más eficiente. Aunque las animaciones hayan sido hechas para un personaje en concreto es posible hacer uso de ellas en **otros modelos** gracias al **Animation Retargeting** [58].

Esta técnica se implementa a través del **sistema Avatar** [59] de *Mecanim* el cual representa una interfaz común, que define una estructura del esqueleto compartida entre diferentes personajes. Al tener la misma organización de huesos es posible aplicar las mismas transformaciones a estos huesos indicados a través del Avatar, logrando así que las animaciones puedan verse reutilizadas de nuevo en otros modelos. Aun así, por este mismo motivo, solo es posible realizar el *retargeting* de animaciones en personajes que sigan el esquema de esqueleto humanoide definido por Unity.

Otros usos del sistema avatar se extienden a las mascarás de animación (*Avatar Mask*) utilizadas en las capas o en las mismas animaciones para evitar animar ciertas partes concretas del esqueleto.



Ilustración 30- Diferentes modelos usando la misma animación mediante Animation Retargeting [60]

3.5 Root Motion y Script Motion

A la hora de mover a un personaje animado por el espacio virtual existen dos enfoques diferentes: *Root Motion* y *Script Motion*.

En la técnica de **Root Motion** el **movimiento** está **construido dentro** de la propia **animación**, y es esta la que se encarga de determinar cómo se mueve el personaje, de esta forma es posible desplazar un modelo de formas más complejas y variadas, sumado a que el **movimiento** está perfectamente **sincronizado** con la **animación**, evitando el problema del *feet-sliding*. Unity dota a los desarrolladores de una implementación de *Root Motion* integrada junto con el sistema de animaciones [61] al mismo tiempo que permite sobrescribirla para construir una solución personalizada [62].

Por otro lado, se encuentra el **Script Motion**, en el que el **movimiento** del personaje se **controla** por medio de **código** a través de una simulación, la **animación solo** conforma la parte **visual** del movimiento. Este método no es tan exacto como el *Root Motion*, pero intercambia precisión por **flexibilidad**, ya que los desplazamientos pueden ser ajustados sin

tener que realizar modificaciones en la animación en sí, lo que optimiza en gran medida la carga de trabajo para movimientos que precisan de un volumen de **iteración** considerable. Adicionalmente, el *Script Motion* aporta al movimiento de una **mayor capacidad de respuesta** debido a que no se rige por un movimiento predefinido por la animación, lo que mejora el control del personaje por parte del jugador.

En Unity existen varias formas de mover un objeto por el mundo virtual mediante código:

- **Transform:** se basa en modificar la *Transform* del *GameObject* en cuestión, ya sea por medio de los métodos *Translate* o *Rotate* o asignando valores directamente a la posición y rotación del objeto. Este método resulta ideal, debido a su simplicidad, para elementos que no necesitan tener en cuenta ningún tipo de colisión o físicas.
- **Character Controller:** se trata de un componente de Unity pensado expresamente para el movimiento de personajes y aporta una gran cantidad de facilidades al desarrollador como la detección de si el personaje se encuentra en el suelo, colisiones predefinidas o la subida automática de rampas y escalones [63].



Ilustración 31- Componente Character Controller

- **Rigidbody:** aunque su principal función sea añadir físicas realistas a un objeto, también es utilizado para crear movimientos controlados para personajes de una manera parecida a como lo hace el *Character Controller*. Aun sin tener las facilidades que aporta este componente, *rigidbody* es ideal para movimientos que precisan ser integrados físicamente en el entorno virtual [64].

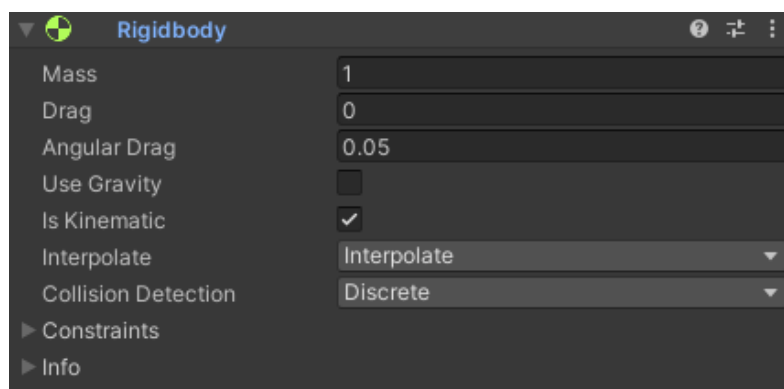


Ilustración 32- Componente Rigidbody

Aunque tengan diferencias notables, estos dos **métodos** son **complementarios** y pueden ser utilizado al mismo tiempo haciendo uso de sus cualidades en las situaciones que mejor se adapten y del resultado que se quiera obtener. Si se precisan animaciones de corte realista o animaciones que implementen movimientos complejos en los que el jugador no vaya a tener mucho control, el *Root Motion* será lo más indicado. Por el contrario, si se requiere perfilar continuamente el movimiento del personaje o que este siga de la manera más fluida los inputs del jugador, el *Script Motion* será la mejor opción.



Diseño de juego

En este apartado se verá todo el diseño de juego perteneciente a “Arcanima: Mist of Oblivion”, con gran hincapié al sistema de combate y a las mecánicas incluidas en este. También se realizará un repaso por todos los personajes existentes desde el punto de vista de la jugabilidad, atendiendo a cada uno de sus repertorios de acciones y ataques disponibles.

4.1 Sistema de combate

El sistema de combate de “**Arcanima: Mist of Oblivion**” se caracteriza por **combinar** el **combate en tiempo real** propio de un *Hack ‘n’ Slash*, con el **combate a tiempo parado** por turnos típico de los *JRPG*. En esta hibridación el jugador tendrá que ir alternando los diferentes modos de combate para enfrentarse a las diferentes situaciones que se le plantean.

El combate en **tiempo real** tiene como mayor referencia el diseño de “**Nier: Automata**”, en este se basa uno de los pilares fundamentales como es la **cancelación de acciones** en todo momento, lo que dota al sistema de una gran fluidez, haciendo el juego más permisivo y obediente, ya que casi cualquier ataque puede ser cancelado para realizar una esquiva, un bloqueo u otro ataque. Este sistema pone a disposición del jugador diferentes cadenas de ataques de distinta magnitud y cadencia, además de la posibilidad de realizar combos en el aire.

Mientras, el **combate a tiempo parado** tiene lugar por medio del **modo concentración**. En este modo el tiempo del **juego se ralentiza** y el jugador tiene a su disposición nuevas herramientas como **habilidades** especiales, **magias** y **objetos** de los que hacer uso en cualquier momento del combate. La principal influencia de este sistema viene del videojuego “**Final Fantasy VII: Remake**”, con la diferencia de que el modo concentración aquí implementado posee un **tiempo límite de uso** en el que el jugador puede estar dentro de esto para no romper el flujo de combate.

Concentración y voluntad

Durante los enfrentamientos se hace uso de dos recursos fundamentales que forman tanto parte del combate a tiempo real como del combate a tiempo parado:

- **Concentración:** utilizada como **limitador** del **tiempo** que el jugador puede permanecer dentro del **modo concentración**. Se genera pasivamente mientras el jugador está combatiendo en tiempo real y se gasta mientras se encuentra dentro del modo concentración. Si se agota antes de haber realizado una acción el jugador se verá expulsado de este modo.
- **Voluntad:** se trata de la moneda de cambio para **realizar técnicas y actos** dentro del modo concentración, su generación se basa en la realización de acciones positivas dentro del combate a tiempo real como ataques, bloqueos o esquivas exitosas. Al ir encadenando estas acciones se va ganando cada vez más voluntad adicional. En contraposición, la cadena se rompe y el bonus de voluntad se pierde si el jugador es golpeado o pasa mucho tiempo si realizar ninguna acción.

Sistema Ánima

En “Arcanima” cada **personaje** tiene asociado consigo un **ánima**, esta ánima se identifica con la naturaleza del individuo y se caracteriza por un **color**. Las personas que son capaces de manifestar su ánima en forma física son conocidos como “usuarios de ánima”. Existen ocho tipos diferentes de ánimas según su función psíquica dominante y su dirección:

Naturaleza		Dirección	
		Introspectivo	Extrospectivo
Función psíquica dominante	Pensamiento	ROJO	BLANCO
	Sentimiento	CIAN	AZUL
	Sensación	VERDE	VIOLETA
	Intuición	NEGRO	NARANJA

Tabla 10- Naturalezas del ánima

Esta idea, mecánicamente, se traduce en un **sistema de fortalezas y debilidades** donde unas ánimas son más fuertes o débiles contra otras. Los individuos que reciban un ataque elemental de un ánima a la que es débil se verá incapacitado temporalmente a la vez que recibe daño extra.

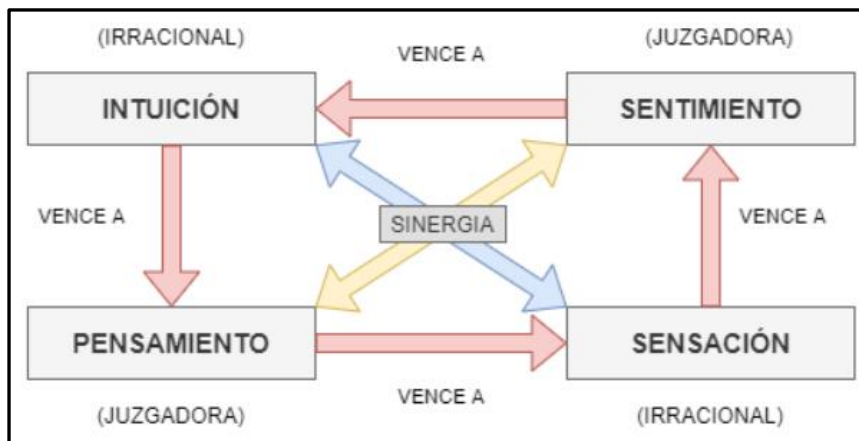


Ilustración 33- Fortalezas y debilidades de las ánimas

Las **ánimas**, además, definen el **repertorio** de **habilidades** y **actos** que posee un personaje. El personaje jugador al poder cambiar de ánimas tendrá a su disposición un abanico diferente de opciones dependiendo de la forma en la que se encuentre. Del mismo modo, dependiendo del ánimo, sus fortalezas y debilidades cambian. Actualmente, en la demo del juego, el jugador solo tiene disponibles la forma roja y naranja, esta última obteniéndola al derrotar al primer jefe.

Flujo de combate

En el flujo de combate se define la **experiencia** objetivo **deseada** durante el desarrollo de los **enfrentamientos**. La idea principal se basa en **intercalar** el combate a **tiempo real** con el combate a **tiempo parado** de la forma más fluida posible.

Al iniciar un enfrentamiento, el jugador empieza sin recursos para poder realizar ninguna acción del modo concentración. Mientras se combate en **tiempo real**, la concentración va creciendo pasivamente y se va **obteniendo voluntad** conforme se van sucediendo las **acciones exitosas**, permitiendo al jugador hacer **uso** de las **técnicas** o **actos** dentro del **modo concentración**. La realización de estas acciones especiales (habilidades, actos y objetos) actúa sobre el combate a tiempo real generando oportunidades beneficiosas que el jugador puede aprovechar para volver a conseguir la voluntad gastada y recargar su concentración, repitiendo así el ciclo.

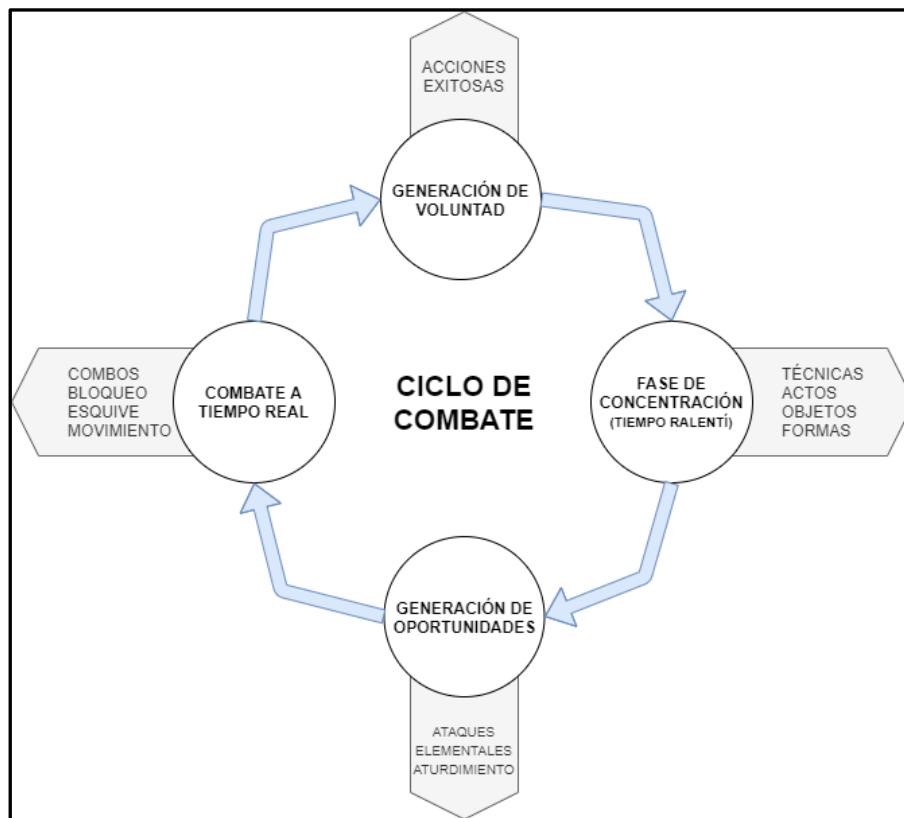


Ilustración 34- Flujo de combate

Sistema de fijado

Una de las características más importantes que acompañan siempre al combate es su sistema de fijado, este es el encargado de **indicar** al jugador el **enemigo** al que irán destinados los ataques. Existen dos modos de fijado disponibles:

- **Automático:** el **sistema selecciona** al **enemigo** más susceptible de ser atacado en base a la distancia respecto al personaje jugador y la orientación hacia la que se encuentre mirando. Esto es, el sistema prioriza a los enemigos más cercanos y afrontados al avatar del jugador.
- **Manual:** la **selección** del **enemigo** está completamente **controlada** por el **jugador** pudiendo cambiar de objetivo en cualquier momento que le sea necesario. Durante este modo de fijado, la cámara se orienta de tal manera que al avatar jugable y el enemigo seleccionado queden siempre enmarcados en pantalla.

Cualquiera de los modos de **fijado** expuestos **redirige** al **jugador** hacia el enemigo objetivo a la hora de **atacar**, independientemente de su orientación o cercanía. Esto se traduce en que por muy lejos que esté un enemigo o quede a las espaldas del personaje, si está fijado como objetivo, el jugador se girará y se desplazará en dirección a este para conectar el ataque.



4.2 Locomoción

Debido a que “*Arcanima*” presenta enfrentamientos rápidos y espacios moderadamente extensos la locomoción no puede quedarse atrás, el movimiento debe adecuarse al combate y a los escenarios.

Al tratarse de un juego *Hack ‘n’ Slash*, el **movimiento** del personaje debe ser **preciso** y **rápido**, en el que cualquier cambio de dirección ingresado por el jugador se refleje de la manera más instantánea posible, produciendo así una sensación de **control «apretado»**

Teniendo en cuenta la extensión de los espacios por los que se desplazará el jugador, el personaje debe tener una velocidad media óptima para que estos recorridos no se alarguen en demasía. Adicionalmente, se proporcionan diversas mecánicas como el **dash** y el **esprint** que favorecen la adaptación a las **zonas** más **extensas**. Al realizar un *dash* el personaje entrará automáticamente en un estado de esprint el cual irá aumentando su velocidad hasta un límite preestablecido.

De igual forma, es importante atender a las **secciones** de **plataformas** que se presentan en el juego. La locomoción también contempla **desplazamientos** en el **aire** como pueden ser los **saltos** o incluso realizar un **dash** en el **aire** para abarcar las distancias más largas entre plataformas. El personaje cuenta con la habilidad de realizar un doble salto como es tradicional en los juegos de plataformas. Adicionalmente y para **mejorar** la **navegación** en la verticalidad del escenario, el personaje tiene la capacidad de poder **trepar salientes automáticamente** una vez este se encuentre lo suficientemente cerca de estos.

Por último, y a modo de recompensa para los jugadores más expertos, el sistema de **combate aéreo** también se presta al jugador como una herramienta para **desplazarse** de una manera **alternativa** por el **aire**. Debido a que al realizar ataques el personaje se desplaza, esto puede ser utilizado a favor y de forma creativa para llegar a localizaciones donde, con el repertorio normal de acciones de locomoción, no es posible acceder a priori. Ejemplos claros de estas decisiones de diseño se pueden observar en la saga Nier [65].

4.3 Mecánicas

Existen diferentes mecánicas involucradas en el sistema de combate, estas se identifican con los verbos del juego, esto es, con las **acciones** que el **jugador** puede hacer.

Atacar

La principal fuente de daño que posee el jugador para derrotar a los enemigos, los ataques pueden concatenarse para formar cadenas de ataques.

- **Ataques ligeros:** golpes rápidos de daño leve a moderado. Utilizados tanto en tierra como en el combate aéreo.
- **Ataques pesados:** aunque más lentos que los ligeros estos aportan más daño y abarcan más área. No pueden utilizarse en el combate aéreo.



- **Cadenas de ataques:** los ataques ligeros pueden concatenarse hasta cuatro veces y los ataques pesados hasta tres. Aun así, golpes ligeros y pesados pueden combinarse entre ellos para dar como resultado cadenas más largas y diversas.

Adicionalmente, el jugador tiene a su disposición otros ataques más específicos como el **launcher**, el **smasher** o el **ataque en carrera** ya definidos en el apartado teórico.

A la hora de **conectar** cualquier tipo de **ataque**, independientemente de si el objetivo es un enemigo o el propio jugador, el **personaje atacado** puede experimentar un **desplazamiento** en forma de **knockback** o **airborne**, dependiendo de la naturaleza del ataque. La longitud y el tiempo de estos desplazamientos depende de la magnitud del ataque recibido o de si el personaje se ve siquiera afectado por ellos, como en el caso de los jefes del nivel, en los que la mayoría de knockbacks no tienen efecto. “*Arcanima*” presenta tres tipos diferentes de desplazamientos al ser atacado:

- **Knockback horizontal:** el personaje retrocede en la dirección en la que ha recibido el ataque, es el desplazamiento con más variabilidad en cuanto al tiempo de incapacitación que aplica.
- **Airborne vertical:** el personaje sube en vertical y se queda un breve periodo de tiempo suspendido en el aire hasta que vuelve a bajar, obteniendo el control al tocar el suelo.
- **Airborne parabólico:** el personaje describe una parábola mientras vuela por los aires, al llegar al suelo tiene que recuperar poniéndose nuevamente de pie. Se trata del desplazamiento más incapacitante.

Dash

Aunque desempeñe un papel importante en el movimiento como se ha visto en el apartado de locomoción, el dash se integra en el sistema de combate como una **esquiva** para eludir los ataques enemigos. Además, funciona como una herramienta de reposicionamiento durante el enfrentamiento ya sea de manera ofensiva o defensiva.

Al igual que el bloqueo, el jugador puede cancelar cualquier acción para efectuar un dash. Para evitar el uso abusivo de esta mecánica, y que así existan ventanas de vulnerabilidad en las que el jugador pueda recibir daño, se incorpora un leve **tiempo de espera** entre *dashes* para evitar un uso abusivo.

Bloquear y Contraatacar

El jugador podrá cancelar la mayoría de las acciones para realizar un bloqueo. Este bloqueo podrá **detener** cualquier **ataque** independientemente de su dirección, es decir, aunque el jugador este de espaldas al atacante, ese se girará automáticamente para detener el golpe. Aun así, existen **ataques no bloqueables** en los que la única opción es esquivarlos. Mientras se ejecuta el bloqueo el personaje no podrá moverse ni cancelarlo, queda en una pose de



guardia hasta que se acabe el tiempo de bloqueo o hasta que bloquee exitosamente un ataque.

Al **parar un golpe** se prolonga brevemente la duración efectiva del bloqueo, para así poder seguir deteniendo ataques sucesivos. Además, se le proporciona al jugador una pequeña ventana de reacción en la que puede realizar un **contraataque**. Este contraataque puede ser interrumpido si el jugador es golpeado durante su ejecución.

Concentrarse

Para poder **acceder al modo concentración** el jugador deberá concentrarse, esta mecánica es la que marca el límite entre el combate a tiempo real y el combate a tiempo parado. Mientras el jugador está concentrado el tiempo del juego se ralentiza drásticamente y se ponen a disposición **acciones especiales** solo accesibles desde el modo concentración:

- **Técnicas:** ataques especiales puramente **físicos**, es decir, no son elementales y no se aplican en el sistema de fortalezas y debilidades. Estas además de utilizar voluntad para su uso pueden requerir cierta cantidad de la vida del personaje para ejecutarse.
- **Actos:** análogo a las **magias** en un JRPG clásico, a diferencia de las habilidades, son ataques elementales que participan en el sistema de fortalezas y debilidades pudiendo ser más o menos efectivos dependiendo de la naturaleza del ánima enemiga.
- **Objetos: consumibles** de un solo uso que mejoran el estado del jugador al usarse. En la demo solo existen dos tipos de objetos:
 - o **Poción:** restaura una cantidad fija de la vitalidad del personaje.
 - o **Potenciador de voluntad:** durante un periodo de tiempo se aumenta enormemente la obtención de voluntad a través de acciones positivas en combate.

El uso de los objetos nunca es inmediato, siempre hay un tiempo de preparación sincronizado con una animación antes de que lleguen a tener efecto.

- **Cambio de forma:** mientras el jugador esté concentrado podrá **cambiar de ánima** de manera instantánea. Al cambiar de ánima también se **cambia** el repertorio de **técnicas y actos**.



Ilustración 35 - Interfaz modo concentración

Todas las **acciones** del **modo concentración** no son cancelables durante su ejecución, y la mayoría, a excepción del cambio de forma, tienen un **tiempo de preparación** para poder realizarla al completo. Estos tiempos de espera varían según la acción especial que se haya seleccionado.

Todas las acciones, excepto el uso de objetos y el cambio de forma, se anuncian por su nombre en forma de un cartel sobre la cabeza de los personajes.



Ilustración 36- Cartel anunciando la técnica del soldado raso, “Carga de Valor”

Marcar

Cuando el jugador consigue el **ánima naranja** al derrotar al primer jefe desbloquea una habilidad de dicha ánima que puede ejecutar en tiempo real sin necesidad de pasar por el modo concentración, esta habilidad se trata de “Marcar”.



Ilustración 37- Soldado blindado marcado

Al utilizar “Marcar” el personaje realiza un ataque físico a modo de empujón que deja una marca sobre el enemigo golpeado o sobre el suelo si no alcanza a ningún objetivo. Esta mecánica trabaja junto con el **acto** llamado “**Retorno**” del **ánima naranja** accesible a través del modo concentración. “**Retorno**” **teletransporta** al **personaje encima** de la **marca** y realiza un *smasher* de daño elemental naranja.

4.4 Personajes

Juno

Protagonista de la historia de “*Arcanima*” y **avatar del jugador**. Es el personaje con la **mayor variedad de acciones** en el combate a tiempo real, pudiendo realizar combinaciones de ataques ligeros y pesados, efectuar ataques especiales como *smashers* o *launchers* y hacer uso del modo concentración para utilizar técnicas, actos y objetos a su favor. Cuando obtiene el ánima crepuscular según avanza la historia desbloquea nuevas mecánicas y habilidades, además de la capacidad de cambiar de ánima cuando sea necesario.

Ánima: roja y naranja.

Técnicas:

- **Tajo Brutal:** después de imbuir la espada con energía, realiza una voltereta en el aire para luego caer en picado sobre el suelo produciendo un gran daño en área. Al aterrizar, emergen del suelo una cadena de pinchos en dirección a donde el personaje esté mirando.



Ilustración 38- Personaje Juno

Actos:

- **Curación:** el personaje recupera una cantidad de salud considerable después de un tiempo de preparación.
- **Retorno:** transporta instantáneamente al personaje encima del objetivo designado por la mecánica de “Marcar” para caer sobre él produciendo daño en área. Este acto se consigue al derrotar al primer jefe del juego, Raj. Se trata de un ataque elemental de naturaleza naranja, afectando a los personajes débiles a esta ánima.

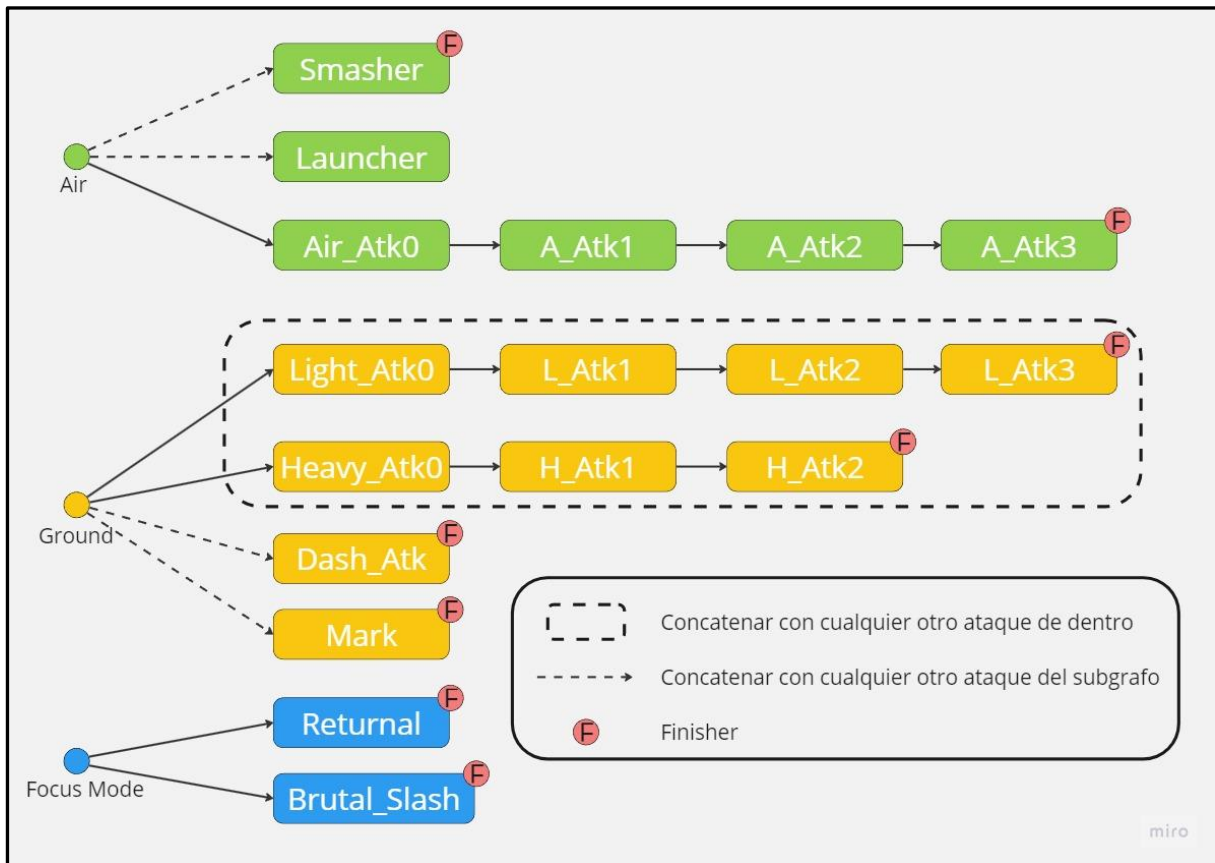


Ilustración 39- Combo graph Juno

Soldado raso

Enemigo más común del juego, compensa su **bajo** nivel de **dificultad** presentándose en **grupo** en los enfrentamientos. Estos soldados se organizan para **rodear** al **jugador** y **atacar** por diferentes **flancos** realizando tajos dobles o muy de vez en cuando una carga de valor para sorprender al jugador.

Ánima: verde.

Técnicas:

- Carga de valor: Después de realizar un grito de guerra, el soldado se dirige a gran velocidad hacia el jugador, una vez se encuentre cerca de este realiza una rápida carga hacia él para propinarle un tajo amplio horizontal.

Actos: ninguno.



Ilustración 40- Personaje Soldado Raso

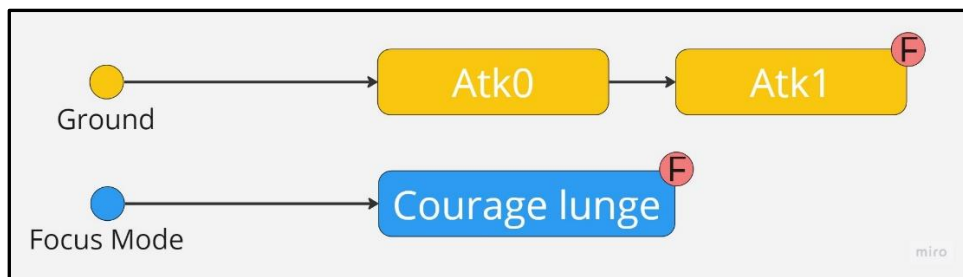


Ilustración 41- Combo graph Soldado Raso

Soldado blindado

Utiliza su gran **escudo** para **repeler** los **ataques** del jugador protegiéndolo de la mayor parte del daño recibido. El escudo presenta una barra de resistencia que es necesario destruir para poder romper su guardia y poder dañar por completo al blindado, aunque solo los ataques pesados afectan al escudo. **Colabora** con los **soldados rasos** aportando un frente fuerte mientras estos intentan buscar oportunidades para atacar al jugador.

Ánima: roja.

Técnicas:

- Carga de escudo: se trata de una técnica idéntica a la Carga de valor del soldado raso, solo cambia su tempo y el daño que inflige.
- Golpe aturridor: realiza un rápido revés con su escudo, aturdiendo al jugador.



Ilustración 42- Personaje Soldado Blindado

Actos: ninguno.

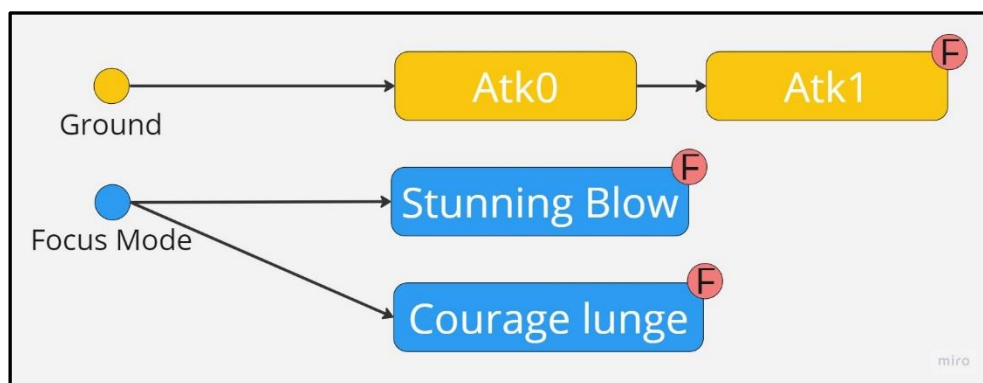


Ilustración 43- Combo graph Soldado Blindado

Autómata volador de ataque

Ataca al jugador **a distancia** disparando ráfagas de **proyectiles**, manteniéndose lejos del jugador cuando este se le acerca. Su estrategia se basa en **sobrecalentarse** para **aumentar** su **cadencia** de tiro; si es ignorado por mucho tiempo es capaz de realizar grandes cantidades de daño de forma rápida. Cabe la posibilidad de que se autodestruya cuando le queda poca vitalidad. Debido a su capacidad de volar el jugador deberá saltar hacia él y realizar ataques aéreos para poder infligirle daño.

Ánima: violeta.

Técnicas:

- Kamikaze: se precipita sobre el jugador para explotar a su lado infligiendo una gran cantidad de daño



Ilustración 44- Personaje Autómata Volador de Ataque

Actos:

- Overclocking: dedica unos momentos para sobrecalentarse y así aumentar su cadencia de tiro, incorporando más proyectiles en la ráfaga. Puede realizar *overclock* hasta un máximo de cinco veces.

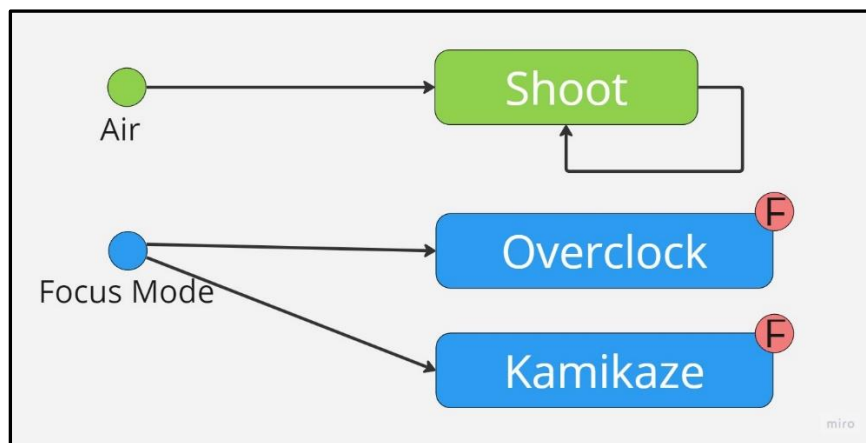


Ilustración 45- Combo graph Automata Volador de Ataque

Raj

Se trata del **primer jefe** al que se enfrentará el jugador. Raj es un usuario de **ánima crepuscular** (naranja). Se desplaza lentamente, pero sus ataques son realmente potentes. Es capaz de realizar dos combos diferentes además de las habilidades de “Marcar” y “Retorno” descritas anteriormente, las cuales pertenecerán a Juno una vez lo haya derrotado.

Ánima: naranja.

Técnicas:

- Marcar: funciona de manera idéntica a la mecánica de “Marcar” del jugador.
- Rondo del comandante: realiza un tajo circular completo de gran alcance que hace volar al personaje por los aires.



Ilustración 46- Personaje Raj

Actos:

- Retorno: Al igual que el Retorno de Juno, se trata de un ataque elemental naranja. Como Juno solo posee el ánima roja hasta el momento del encuentro, el jugador deberá tener especial cuidado con esta habilidad, ya que siempre recibirá daño extra.

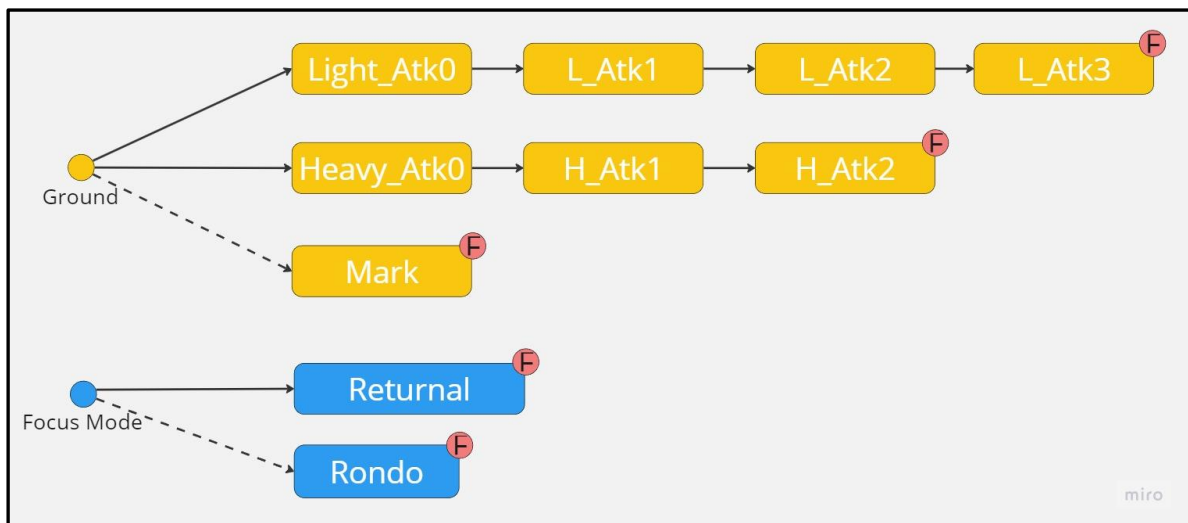


Ilustración 47- Combo graph Raj

Nyx

Jefe final de la demo, se trata de un usuario de ánima oscura con la capacidad de **adueñarse** de **ánimas ajenas**. Su estrategia se basa en **teletransportarse** continuamente durante el combate para confundir al jugador y generar ventanas de oportunidades en las que atacar. Además de su propia ánima, puede hacer **uso** de las **ánimas robadas** como el ánima **crepuscular** de Raj (glaive) o el ánima **añil** perteneciente a un personaje aún desconocido (ballesta).

Ánima: negra, pero puede utilizar otras ánimas como naranja y azul.

Técnicas:

- **Armisticio:** Nyx se teletransporta fuera del alcance del jugador y empieza a invocar diferentes tipos de ánimas al alrededor de la arena de combate, las cuales se moverán a gran velocidad hacia el centro del escenario atravesando todo el campo de batalla.



Ilustración 48- Personaje Nyx

Actos:

- **Entierro:** se trata de una modificación de Retorno, con la diferencia de que no es necesario mantener un objetivo marcado para poder realizarla.
- **Ánima Esclava:** Nyx solo puede mantener una de las dos ánimas robadas de forma activa y dependiendo de esta podrá realizar un ataque u otro. La habilidad siempre viene precedida de un tiempo de canalización y aplican el elemento del ánima al que pertenecen.
 - **Bala del diablo – Ánima Añil:** dispara un proyectil que sigue al jugador hasta que lo golpea o es bloqueado.
 - **Rondo del comandante – Ánima Crepuscular:** idéntica a la técnica de Raj, con la salvedad de que ahora es un ataque elemental en vez de solo físico.



Ilustración 49- Combo graph Nyx



Descripción informática

Este capítulo se dedicará al análisis, diseño e implementación de los sistemas y características del juego expuestos en el apartado anterior. Se extraerán los requisitos necesarios para cumplir con toda la funcionalidad requerida y se procederá a realizar una etapa de diseño para planificar la estructura del proyecto. Por último, se ahondará en cómo se ha llevado a cabo la integración de cada uno de los sistemas de locomoción, combate, gamefeel y fijado y la consiguiente gestión de las animaciones necesarias.

5.1 Análisis

En el siguiente apartado se exponen los requisitos extraídos a partir del diseño de juego

Extracción de Requisitos

Requisitos del sistema

- RS1. Cada uno de los personajes posee un repertorio de ataques diferentes definido por un **combo graph**.
- RS2. Todos los personajes albergan la información necesaria que define su movimiento sobre cómo desplazarse por el mundo virtual en forma de **velocidades máximas, aceleraciones y fricciones**.

Requisitos funcionales

Locomoción:

- RF1. El movimiento del personaje jugador es relativo a la cámara. Si la cámara cambia de orientación, la dirección de movimiento se verá alterada en consecuencia.
- RF2. El personaje jugador puede llegar a saltar hasta dos veces antes de tocar el suelo.
- RF3. El personaje jugador puede escalar salientes si este se encuentra cerca del borde de una plataforma o estructura.
- RF4. El personaje jugador puede realizar un dash.



- RF5. Después de realizar un dash, el personaje jugador entra en modo sprint, lo que aumenta su velocidad de desplazamiento máximo.
- RF6. Todos los personajes pueden llegar a realizar desplazamientos horizontales o verticales predefinidos en distancia y tiempo en una dirección concreta e inalterable durante el mismo.
- RF7. Las animaciones de locomoción deben ir acorde a la velocidad y dirección del movimiento. Estas deben reaccionar a las diferencias entre correr, andar y esprintar, así como al cambio de dirección cuando los personajes realizan *strafing*.
- RF8. El sistema de movimiento debe contemplar el sistema de *pathfinding* de Unity para los agentes controlados por la IA.
- RF9. El sistema de movimiento debe tener la capacidad de alternar entre *scripted motion* y *root motion*.

Combate a tiempo real:

- RF10. Cualquier personaje debe tener la capacidad de concatenar ataques.
- RF11. Cualquier ataque debe tener la posibilidad de cancelar su propia animación con el fin de poder realizar otra acción como bloquear, moverse, saltar o atacar nuevamente.
- RF12. Cuando un ataque conecta se inflige la cantidad de daño correspondiente, atendiendo al sistema de fortalezas y debilidades.
- RF13. Al realizar un bloqueo con éxito se evita todo el daño recibido.
- RF14. Al bloquear, el personaje jugador dispone de una ventana de tiempo para realizar un contraataque.
- RF15. El personaje jugador puede realizar ataques aéreos, suspendiéndose en el aire mientras los ejecuta.
- RF16. Al conectar un ataque contra un personaje este puede experimentar un knockback que interrumpa su acción actual, incapacitándolo durante un periodo de tiempo predefinido. El knockback a aplicar depende de la magnitud del ataque y del personaje involucrado.
- RF17. Existen acciones que no pueden ser interrumpidas, durante las cuales no se les puede aplicar ningún tipo de knockback a los personajes que las están realizando.
- RF18. Al recibir un ataque elemental, si el ánimo del personaje es débil contra el ánimo del atacante este se verá aturdido durante un periodo de tiempo.



RF19. Durante la realización de un dash, el personaje jugador es completamente invulnerable a cualquier ataque.

Gamefeel:

RF20. Al conectar un golpe se debe poder aplicar un efecto de hit pause, la duración de este debe ser personalizable para cada ataque.

RF21. Se debe poder realizar en cualquier momento a lo largo del juego cada uno de los siguientes efectos de gamefeel:

- *Game Slow*
- *Gamepad rumble*
- *Camera Shake*

Combate a tiempo parado:

RF22. Al seleccionar una acción en el modo concentración se debe reproducir inmediatamente la animación correcta.

RF23. Al ejecutar un acto o consumir un objeto el jugador debe esperar un periodo de tiempo dictado por la animación hasta que se aplique el efecto en sí de la acción.

RF24. La acción de usar objetos debe hacer uso de la misma animación, aunque los objetos sean diferentes.

Sistema de fijado

RF25. El modo de fijado automático debe seleccionar al objetivo más adecuado en base a la distancia y orientación con respecto al personaje avatar.

RF26. El jugador debe poder cambiar de objetivo durante el modo manual, seleccionando al enemigo previo o posterior al objetivo actual según un orden basado en la orientación de estos con respecto al personaje jugable.

Requisitos no funcionales

RNF1. Los sistemas y sus elementos (valores de movimiento, propiedades de los ataques, efectos de gamefeel...) deben ser fácilmente modificables, al mismo tiempo que se mantienen usables y accesibles para los integrantes no programadores del equipo.

5.2 Diseño

A continuación, se exhiben una serie de gráficos⁶ con el propósito de mostrar los diversos componentes implicados en el sistema y las conexiones existentes entre ellos. También se describen los distintos procedimientos necesarios que garantizan el adecuado funcionamiento de la aplicación.

Modelo de Dominio

En el siguiente **diagrama** se presentan de **forma conceptual** las **principales entidades** en el **contexto** del **proyecto**, junto con sus **relaciones** y **características fundamentales**. Este modelo también establece el **vocabulario** y los **conceptos** empleados en todo el ámbito de la aplicación.

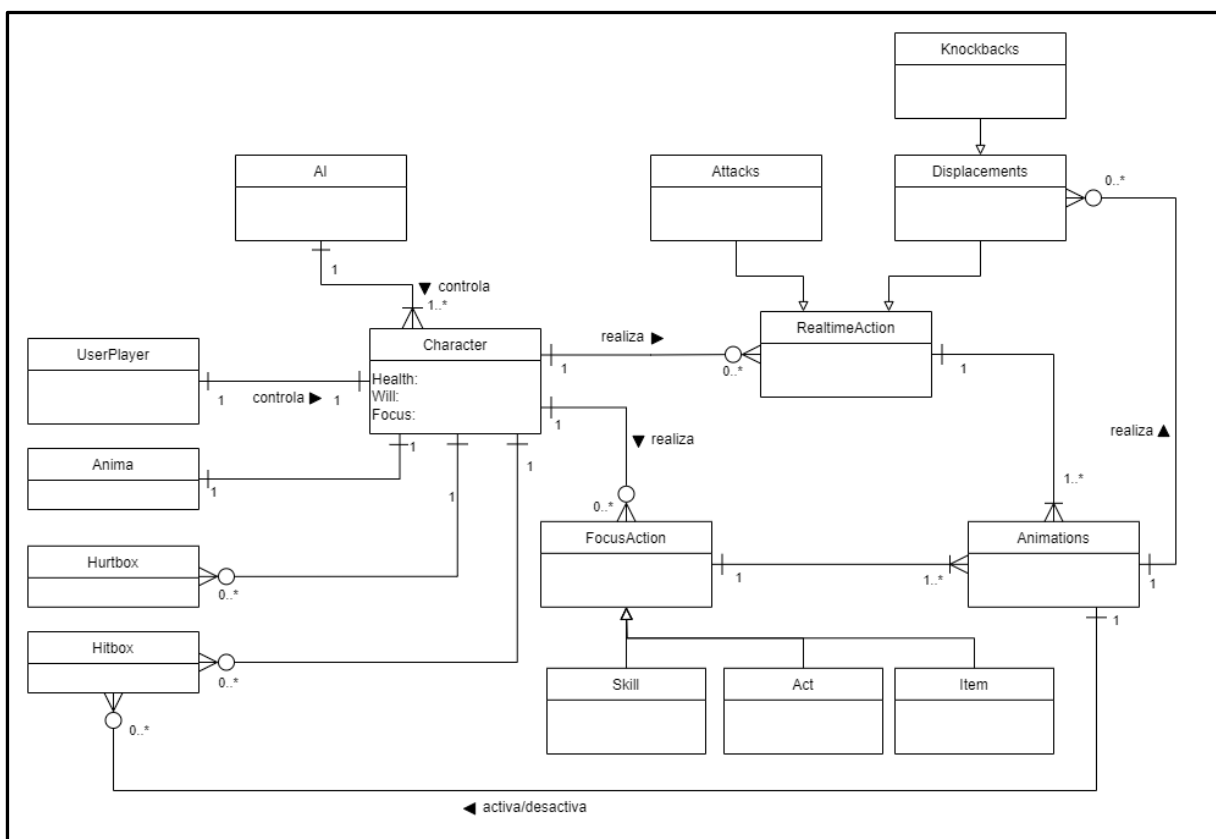


Ilustración 50- Modelo de Dominio

Casos de uso

En este apartado se presentan los principales casos de uso que pueden darse durante la ejecución del programa.

UC1: Un personaje realiza un ataque.

⁶ Estos diagramas se basan en los ejemplos presentados en los libros de “UML y Patrones” [87] e “Ingeniería del Software: Un enfoque práctico” [88]



Actor Principal: *Character*.

Precondiciones: El personaje no puede estar en el estado de “dañado” o, si está realizando otra acción, esta debe poder ser cancelable.

Garantías de éxito: El personaje ataca. La animación correspondiente del ataque se reproduce. El input de movimiento se bloquea en una dirección fija.

Escenario principal de éxito:

1. El *UserPlayer* o la AI genera el *input* del ataque.
2. El personaje comienza la animación de ataque.
3. Se activan las *hitboxes* correspondiente.
4. Se producen los desplazamientos indicados en la animación.

Extensiones:

- 2a. El personaje es dañado durante la animación:
 1. La animación del ataque se ve interrumpida.
 2. El personaje pasa al estado “dañado”.
 3. El personaje realiza un *knockback*.

UC2: El personaje jugador se pone en guardia para bloquear.

Actor Principal: *Character*.

Precondiciones: El personaje no puede estar en el estado de “dañado” o, si está realizando otra acción, esta debe poder ser cancelable.

Garantías de éxito: La animación del bloqueo se reproduce. El personaje bloquea los próximos ataques.

Escenario principal de éxito:

1. El *UserPlayer* acciona el input de bloquear.
2. El personaje entra en el estado de “bloqueo”.
3. Se reproduce la animación de bloqueo.
4. Se desactiva la *hurtbox* principal y se activa la *hurtbox* de *bloqueo*.
- 5a. El personaje no recibe ningún ataque:
 1. Se reproduce la animación de recuperación de bloqueo.
 2. Se activa *hurtbox* principal y se desactiva la *hurtbox* de bloqueo.
 3. Se sale del estado de “bloqueo”.
- 5b. El personaje recibe un ataque:
 1. Se reproduce la animación de ataque bloqueado.
 2. El personaje ignora toda el daño del recibido.
 3. Se extiende el tiempo efectivo de bloqueo.



4. Se abre la ventana de tiempo para contraatacar.
-

UC3: El personaje jugador realiza un dash.

Actor Principal: *Character*.

Precondiciones: El personaje no puede estar en el estado de “dañado” o, si está realizando otra acción, esta debe poder ser cancelable.

Garantías de éxito: El personaje efectúa un dash. El personaje es invulnerable.

Escenario principal de éxito:

1. El *UserPlayer* acciona el input de *dash*.
 2. Se reproduce la animación de *dash*.
 3. El personaje entra en el estado de “dash”.
 4. El personaje efectúa un desplazamiento horizontal.
 5. El personaje se vuelve invulnerable.
-

UC4: El personaje jugador escala un saliente.

Actor Principal: *Character*.

Precondiciones: El personaje debe de haber detectado un saliente previamente al mismo tiempo que se encuentra en el aire.

Garantías de éxito: El personaje realiza la animación de escalado. El personaje sube automáticamente.

Escenario principal de éxito:

1. El personaje entra en el estado de “escalado”.
 2. El *UserPlayer* pierde el control del personaje.
 3. Se ajusta el personaje al saliente.
 4. Se reproduce la animación de escalado.
 5. Se efectúa un pequeño desplazamiento vertical.
 6. El root motion posiciona al personaje encima del saliente.
 7. El *UserPlayer* toma el control de nuevo.
-

UC5: Un personaje conecta un golpe.

Actor Principal: *Character*.



Precondiciones: Una *hitbox* debe de haber colisionado con la *hurtbox* de otro personaje.

Garantías de éxito: El personaje conecta el ataque contra otro.

Escenario principal de éxito:

- 1a. El personaje atacado es invulnerable:
 1. Se detecta el ataque como no registrado.
 - 1b. El personaje atacado está bloqueando:
 1. Se detecta el ataque como bloqueado.
 - 1c. El personaje atacado no es invulnerable ni está bloqueado:
 1. Se detecta el ataque como registrado.
 2. Se daña al personaje atacado.
 3. Se aplica el *hit pause*, *camera shake* y *gamepad rumble* correspondiente.
-

UC6: Un personaje es dañado

Actor Principal: *Character*.

Precondiciones: El personaje no debe ser invulnerable.

Garantías de éxito: El personaje ve reducida su vitalidad.

Escenario principal de éxito:

1. Se calcula el daño según el sistema de fortalezas y debilidades.
2. Se baja la vitalidad del personaje en su interfaz.
3. El personaje pasa al estado “dañado”.
4. El personaje realiza un knockback.

Extensiones:

- 2a. El personaje se muere al no tener más vitalidad:
 1. Se reproduce la animación de muerte.
 2. El personaje pasa al estado de “muerte”.
 - 4a. El alma del personaje es débil al alma del personaje atacante:
 1. El personaje se ve aturdido.
-

UC7: El personaje jugador cancela un ataque en otra acción.

Actor Principal: *Character*.

Precondiciones: El personaje debe estar atacando.

Garantías de éxito: El personaje interrumpe el ataque y pasa a realizar la nueva acción.

Escenario principal de éxito:

1. La animación de ataque activa las ventanas de cancelación correspondientes a las acciones disponibles.
- 2a. El *UserPlayer* introduce el input de salto:
 1. Se sale del estado “*attack*” y se entra en el estado “*idle*”.
 2. Se reproduce la animación de salto.
 3. El personaje efectúa un desplazamiento vertical.
- 2b. El *UserPlayer* introduce el input de bloqueo:
 1. Se sale del estado “*attack*”.
 2. El personaje se pone en guardia para bloquear.
- 2c. El *UserPlayer* introduce el input de *dash*.
 1. Se sale del estado “*attack*”.
 2. El personaje realiza un *dash*.
- 2d. El *UserPlayer* introduce el input de ataque.
 1. El personaje realiza el siguiente ataque de la cadena.
 3. Se reinician las cancelaciones.

Diagramas de caso de uso

En el siguiente diagrama se sintetiza los casos de uso desarrollados.

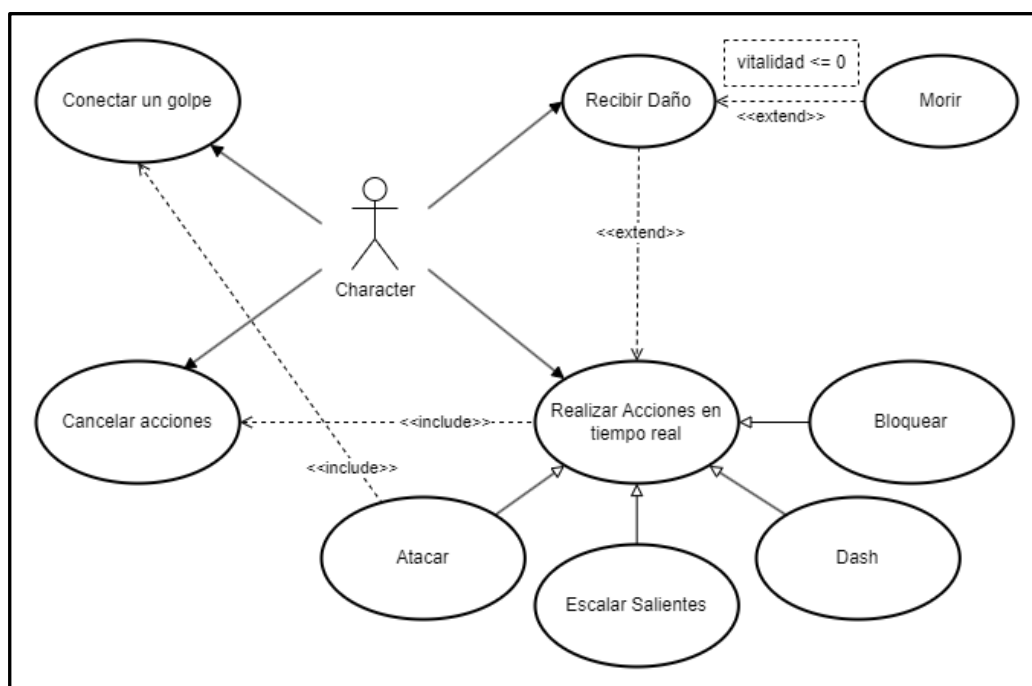


Ilustración 51- Diagrama de casos de uso

Diagramas de secuencia

A continuación, se presentan los diagramas de secuencia de algunas acciones en tiempo real como la **realización del dash** o la de **bloquear**, así como cuando un **personaje** resulta **dañado**.

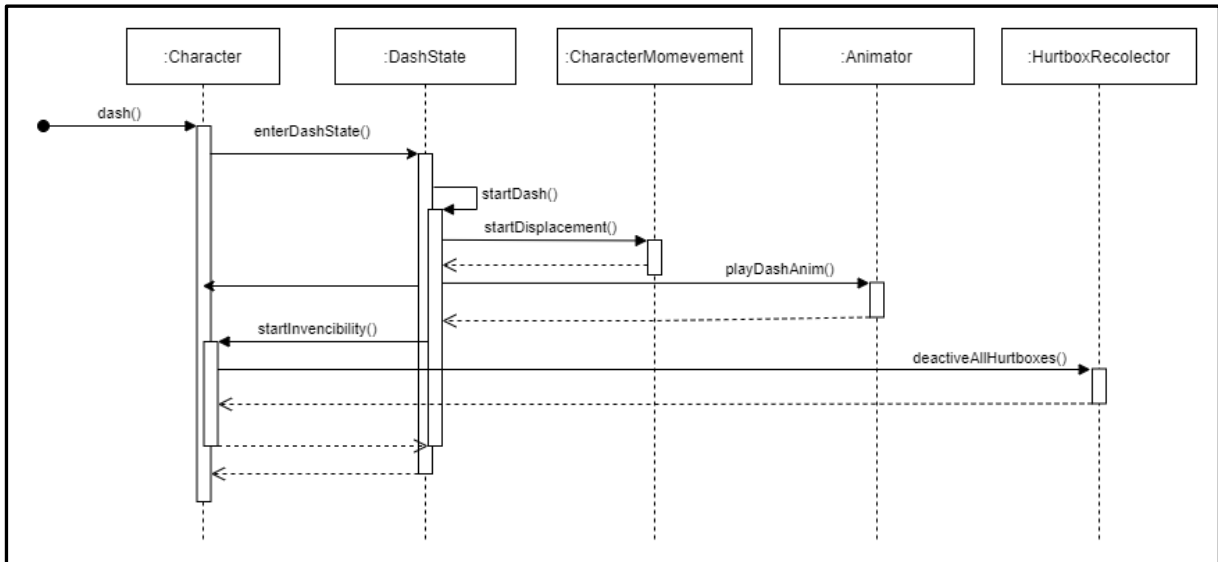


Ilustración 52- Diagrama de secuencia de realizar un dash

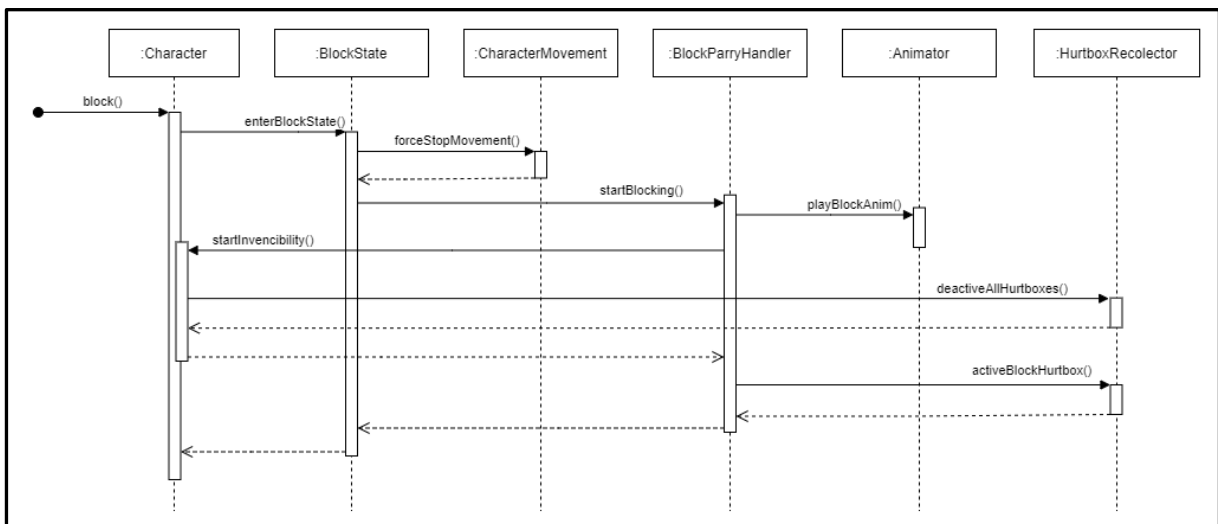


Ilustración 53- Diagrama de secuencia de empezar a bloquear

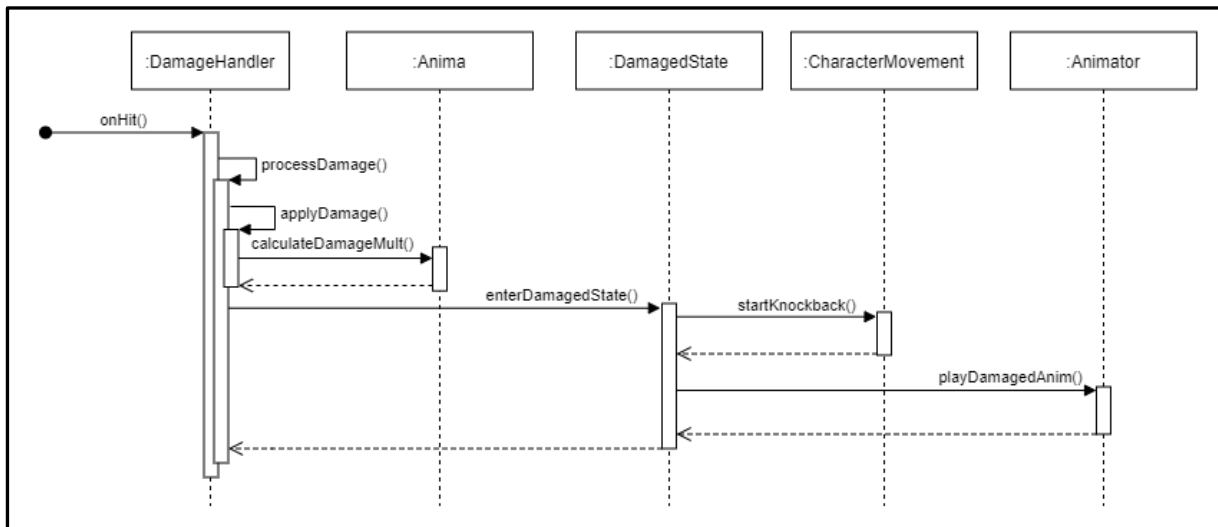


Ilustración 54- Diagrama de secuencia de personaje dañado

Diagramas de colaboración

Para observar detenidamente las relación presentes entre las distintas entidades a la hora de realizar algunas acciones, se atiende a los siguientes diagramas de colaboración.

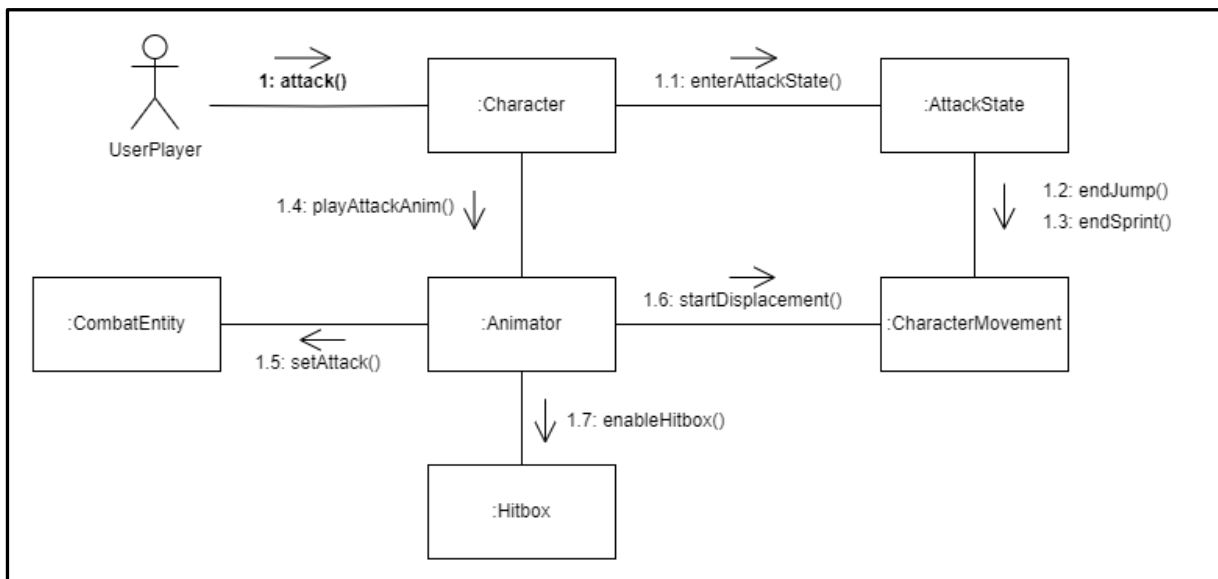


Ilustración 55- Diagrama de colaboración de realizar un ataque

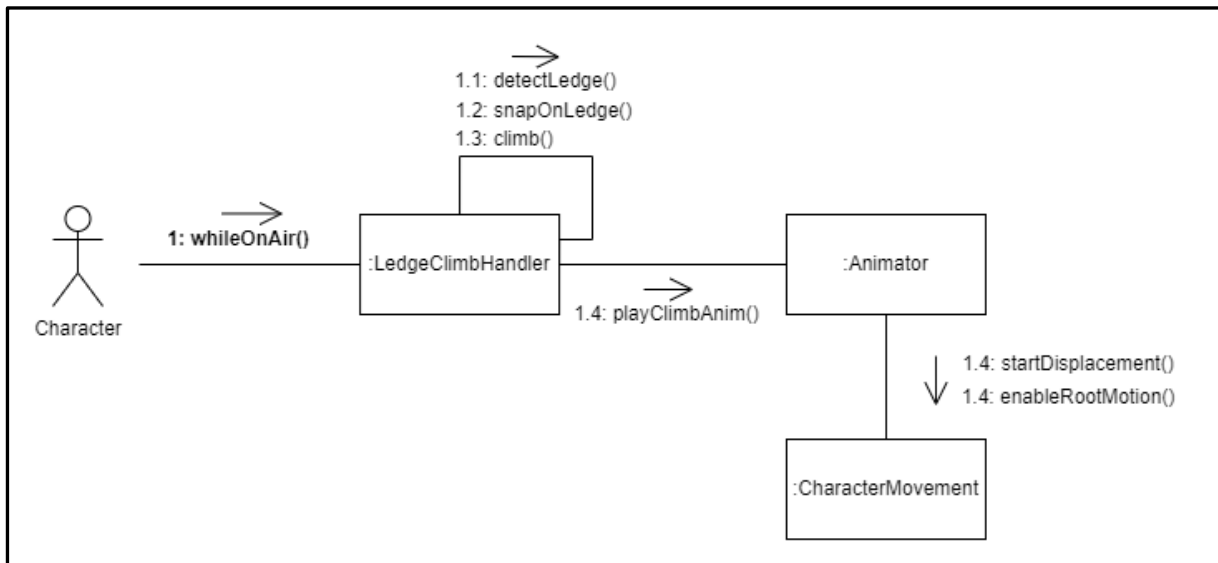


Ilustración 56- Diagrama de colaboración de escalar salientes

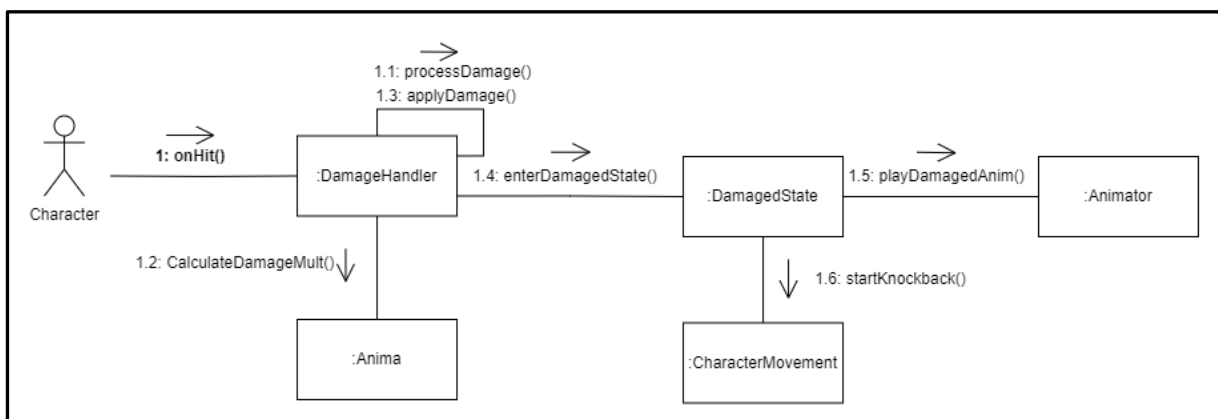


Ilustración 57- Diagrama de colaboración de personaje dañado

Diagrama de clases

En el diagrama de clases se exhiben las principales entidades junto con sus métodos y propiedades más relevantes. Con el fin de mejorar su legibilidad, se ha optado por adjuntar los diagramas de clase como anexo debido a su tamaño ([Anexo I Diagrama de clases](#)).

5.3 Implementación

Sistema de locomoción

El sistema de locomoción incluye tanto la **simulación** del **movimiento** utilizado para desplazar al personaje por el escenario virtual como la **implementación** de las **animaciones** necesarias. Aunque estos aspectos estén altamente relacionados, las animaciones conforman la capa visual y estética que recubre la simulación, y se trabajará de tal modo que la simulación sea completamente ajena a la implementación específica de estas animaciones. De tal modo, se obtendrá un movimiento unificado y compartido por todos los personajes que no dependa de las animaciones concretas de cada uno de ellos.

Preparativos

Primeramente, se crea un **objeto de pruebas** que **simula** los **modelos 3D finales** que se utilizarán para cada personaje. Este objeto resulta de mucha utilidad, debido a que solo contiene la capa visual mínima requerida para evaluar el correcto funcionamiento de la simulación, con lo que se puede apreciar el movimiento tal y como es mientras se implementa.

Como buena práctica, en este objeto ya se introduce la **división** entre la **capa lógica** y la **capa visual** del personaje como pueden ser el modelo o sus animaciones. La primera reside en el objeto raíz, donde se dispondrán todos los scripts necesarios para el movimiento y el combate, mientras que la segunda capa se alojará en objetos hijos para un mayor control y organización.

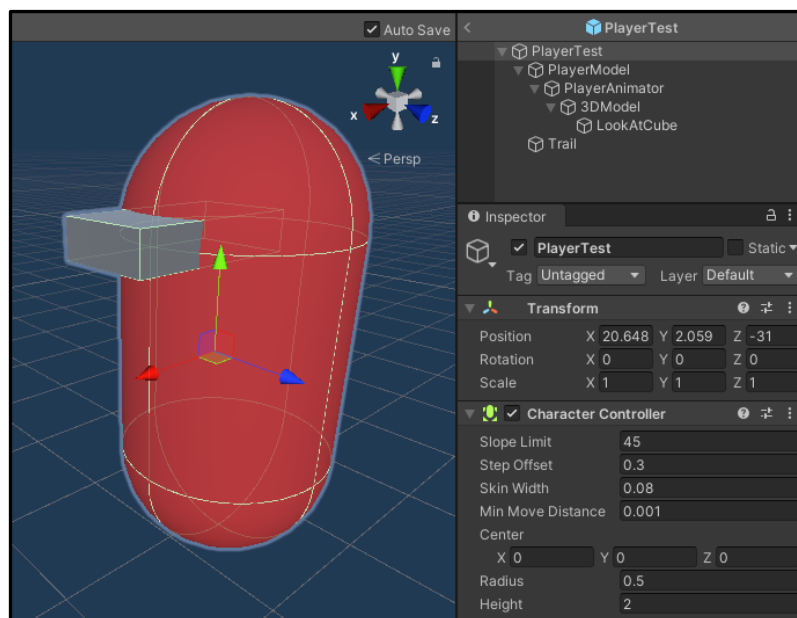


Ilustración 58- Objeto de pruebas que simula un personaje

Adicionalmente, se le añade un componente de *TrailRenderer* [66] que irá dibujando la trayectoria del personaje como ayuda visual para encontrar ciertos artefactos o desperfectos en el movimiento [67].

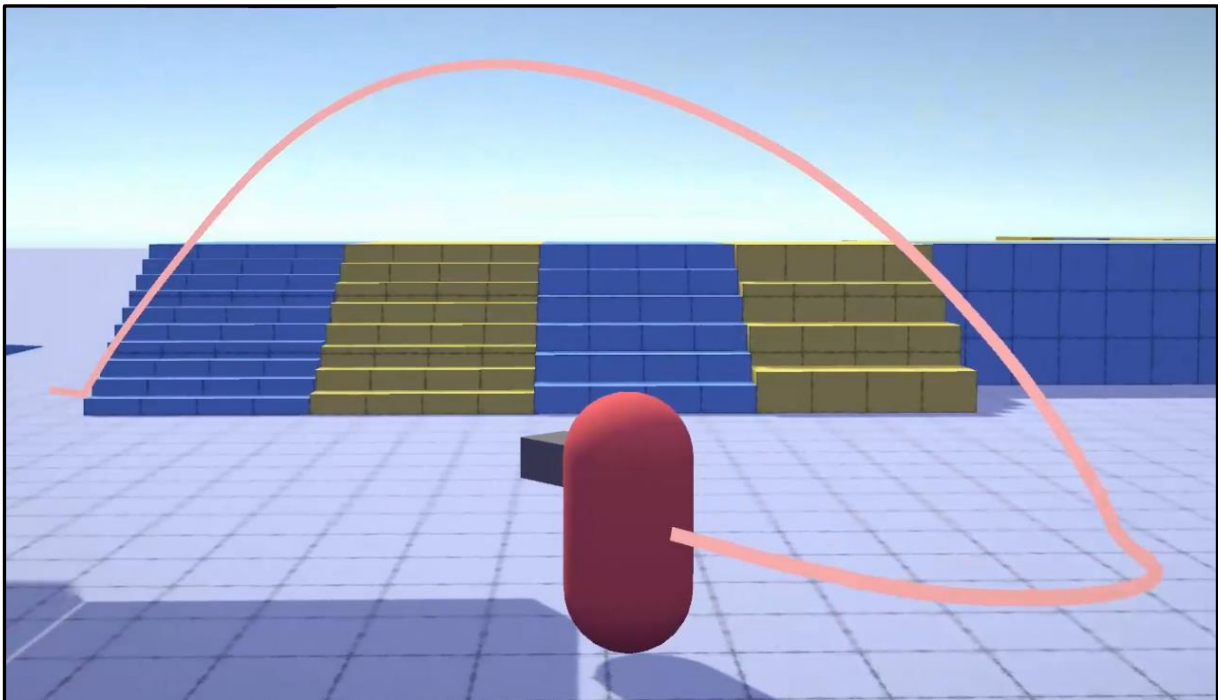


Ilustración 59- Ejemplo de uso del Trail Renderer, el cual dibuja la trayectoria del movimiento

Para terminar, se construye una escena a modo de “gimnasio” en la cual se disponen diferentes superficies y plataformas, que simulan una gran variedad de casos reales, en las que poder probar el correcto funcionamiento del movimiento.

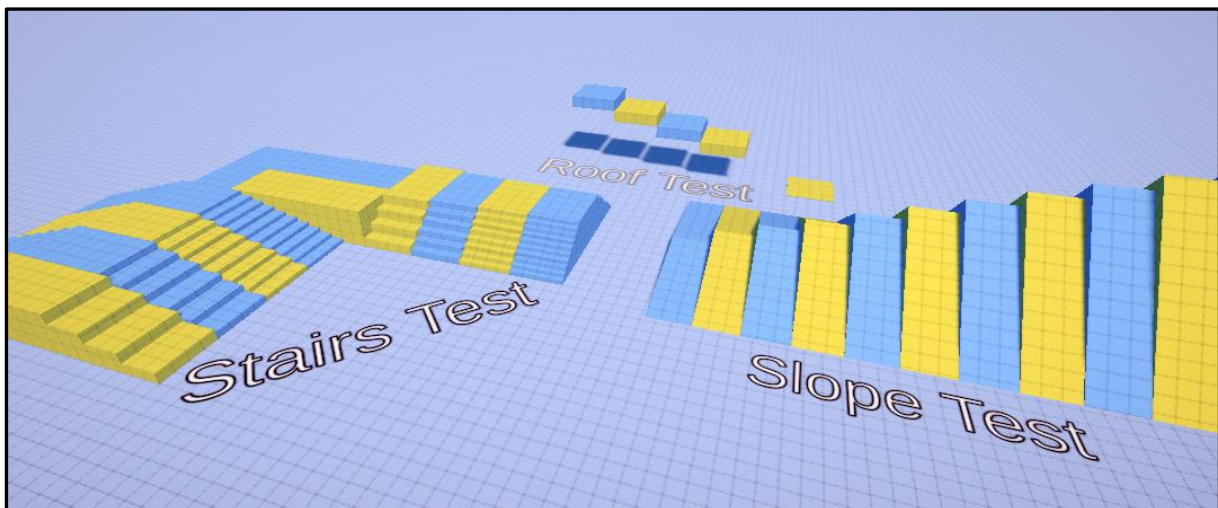


Ilustración 60- Level Gym utilizado para probar el movimiento

Movimiento básico

El movimiento de todos los personajes se basa en una **simulación** definida por el siguiente conjunto de parámetros:

- **Velocidades máximas:** MaxHorSpeed y MaxVertSpeed

- **Aceleraciones:** GroundAcc y AirAcc
- **Fricciones:** GroundFric y AirFric

A grandes rasgos, esta **simulación** del movimiento consiste en el **cálculo** de un **vector** que se le suministra al componente CharacterController de Unity para que este mueva al objeto en sí. La dificultad reside en cómo calcular este vector para todas las diferentes situaciones que es necesario que el sistema contemple. Todos estos cálculos son llevados a cabo en el script CharacterMovement, del cual disponen todos los personajes para realizar su movimiento.

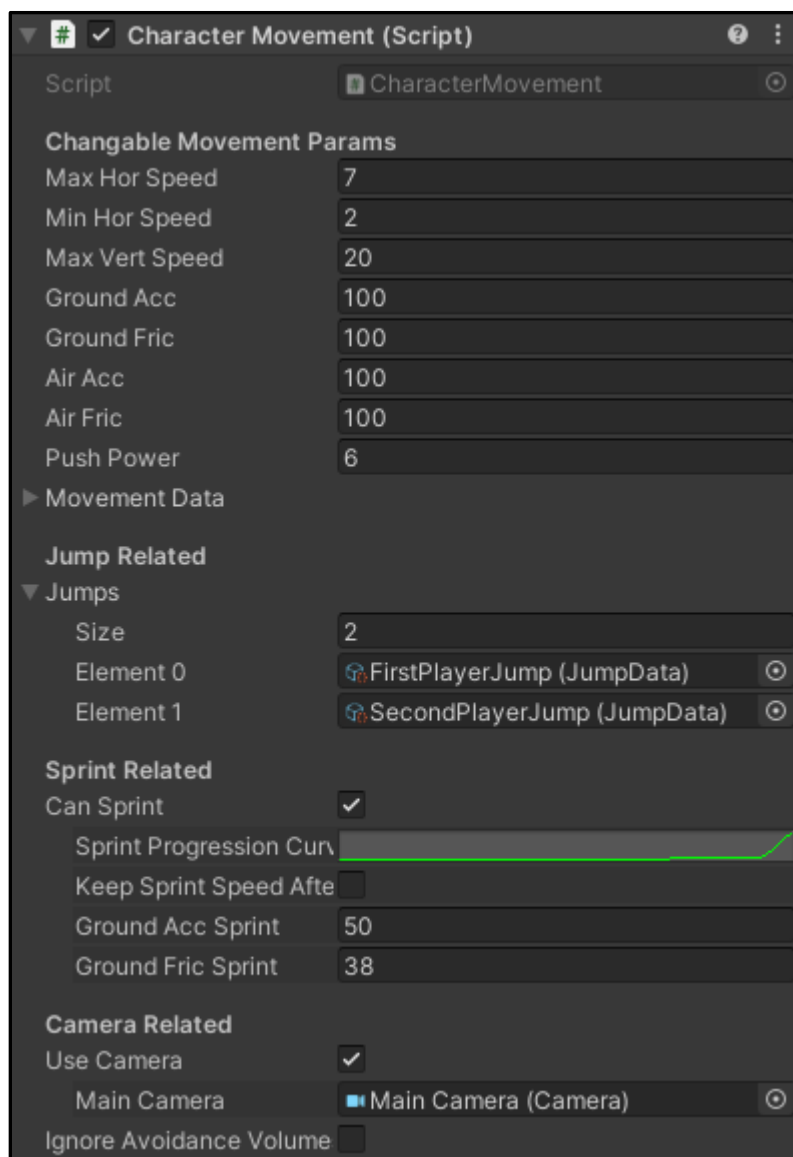


Ilustración 61- Componente CharacterMovement


```
private void Update()
{
    Vector3 moveVector = GetComputedMovement();
    controller.Move(moveVector);
}
```

Snippet 1- Método Update del CharacterMovement, el vector se le suministra al CharacterController de Unity

Dos aspectos troncales que se aplican en cualquier situación a la hora de **calcular** este **vector** son la **dirección** en la que se da el movimiento y el uso de **aceleraciones** y **fricciones** para simular una ASDR.

En cuanto a la **dirección** y debido a la naturaleza del juego, la simulación se **divide** en el cálculo del movimiento **horizontal** y el cálculo del movimiento **vertical**. Aunque el jugador posea total libertad para moverse sobre el plano horizontal, este se ve restringido por la gravedad en su eje vertical, requiriendo un tratamiento diferente.

```
public Vector3 GetComputedMovement()
{
    CurrentHorSpeed = ComputeHorizontalMovement();
    CurrentVertSpeed = ComputeVerticalMovement();

    Vector3 horMovement = CurrentHorSpeed.x0y();
    Vector3 vertMovement = Vector3.up * CurrentVertSpeed;

    return horMovement + vertMovement;
}
```

Snippet 2- Método GetComputedMovement, división del movimiento en horizontal y vertical

El **movimiento horizontal** se calcula en función de la **dirección** proporcionada por el **input** del **usuario**, mediante el joystick de un mando o las teclas de un teclado, o por el *pathfinding* del agente controlado por la IA. Adicionalmente, se atiende a la **magnitud** de esta **dirección** para saber si es necesario **acelerar** o **frenar** el movimiento. Comprobar que no sea cero es lo mismo que saber si se le está ordenando al personaje que se mueve o no.

```
private Vector2 ComputeHorizontalMovement()
{
    Vector2 inputedMoveDir = GetMovementDirection();
    if(inputedMoveDir.magnitude != 0)
        //Acelerar
        return GetAcceleratedHorMovement(inputedMoveDir);
    else
        //Frenar
        return GetBrakedHorMovement();
}
```

Snippet 3- Método ComputeHorizontalMovement

A diferencia de los agentes controlados por la IA, que trabajan desde el primero momento en coordenadas del mundo, la **dirección** del **jugador** necesita ser trasladada desde el sistema de coordenadas del dispositivo de entrada (mando o teclado) hasta el **sistema de**

coordenadas del mundo virtual teniendo en cuenta la **orientación de la cámara**. Con el cambio de base oportuno, el personaje jugador obtiene el vector dirección sobre el que moverse en el plano horizontal teniendo en cuenta la cámara y el input del usuario.

```
public Vector3 GetWorldMovementDirection (Vector2 inputedMoveDir)
{
    if (useCamera)
    {
        Vector3 inputedDir = inputedMoveDir.x0y();
        Vector3 worldMoveDir = mainCamera.transform.TransformVector(inputedDir);
        return worldMoveDir.normalized;
    }
    else
    {
        return inputedMoveDir.x0y().normalized;
    }
}
```

Snippet 4- Método GetWorldMovementDirection

RF1 El movimiento del personaje jugador es relativo a la cámara. Si la cámara cambia de orientación, la dirección de movimiento se verá alterada en consecuencia. ✓

Ahora solo queda definir cómo se realizan las aceleraciones y fricciones, para ello se crea un método el cuál calcula variaciones incrementando o decrementando un valor actual (*current*) según la cantidad estipulada (*step*) para aproximarse a un valor objetivo indicado (*goal*).

```
private float Approach (float current, float goal, float step)
{
    if (current < goal)
        return Mathf.Min(current + step, goal);

    return Mathf.Max(current - step, goal);
}
```

Snippet 5- Método Approach para la simulación de aceleraciones y fricciones

Este método se aplica a cada una de las componentes del vector de movimiento de la siguiente manera:

```
private Vector2 GetAcceleratedHorMovement (Vector2 inputedDir)
{
    Vector3 worldDir = GetWorldMovementDirection(inputedDir);
    Vector3 maxHorSpeedWorldDir = worldDir * maxHorSpeed * inputedDir.magnitude;

    float xSpeed = Approach(CurrentHorSpeed.x, maxHorSpeedWorldDir.x, groundAcc);
    float zSpeed = Approach(CurrentHorSpeed.y, maxHorSpeedWorldDir.z, groundAcc);

    return new Vector2(xSpeed, zSpeed);
}
```

Snippet 6- Método GetAcceleratedHorMovement

```
Vector2 GetBrakedHorMovement()  
{  
    float xSpeed = Approach(CurrentHorSpeed.x, 0, groundFricc);  
    float zSpeed = Approach(CurrentHorSpeed.y, 0, groundFricc);  
  
    return new Vector2(xSpeed, zSpeed);  
}
```

Snippet 7- Método GetBrakedHorMovement

Con esta estructura, se consigue que al **cambiar** las **aceleraciones** y **fricciones** se **cambien** a su vez las fases de *Attack* y *Release* de la **ADSR** que define el **movimiento**, obteniendo en consecuencia nuevas **variaciones** de este. Para obtener un movimiento diferente cuando el personaje se encuentra en el aire, simplemente es necesario modificar la aceleración y fricción suministradas al método Approach para que dé resultado.

```
float tempAcc = characterController.isGrounded ? groundAcc : airAcc;  
float tempFric = characterController.isGrounded ? groundFric : airFric;
```

Snippet 8- Diferente aceleración y fricción cuando el personaje está en el suelo o en el aire

El **movimiento vertical** se calcula de forma análoga, **desacelerando** cuando el personaje se desplaza verticalmente **hacia arriba** y **acelerando** cuando se encuentra en **caída libre**. En este caso la aceleración y fricción se identifican con el valor de la **gravedad**.

```
float ComputeVerticalMovement()  
{  
    float ySpeed = 0.0f;  
  
    if (CurrentVertSpeed > 0)  
    {  
        ySpeed = Approach(CurrentVertSpeed, 0, gravity);  
    }  
    else  
    {  
        if (!controller.isGrounded)  
            ySpeed = Approach(CurrentVertSpeed, -maxVertSpeed, gravity);  
        else  
            ySpeed = gravity;  
    }  
  
    return ySpeed;  
}
```

Snippet 9- Método ComputeVerticalMovement

Movimiento Frame Independent

Hasta ahora se han estado realizando cálculos sin tener en cuenta la posible **variación** de **frames** que pueda haber durante el juego. Esta variación puede ser causada por una diferencia en la **capacidad** de **cómputo** de diferentes **máquinas** -las más potentes serán capaces de ejecutar más iteraciones que máquinas con menos potencia en el mismo período de tiempo- o bien por **picos de carga** o **descarga** en diferentes momentos del **juego** en los que se necesitan más o menos **recursos**.

Si la **lógica de movimiento** se encuentra **atada al número de frames por segundo** esto **repercute** directamente en el **cómputo** y la **sensación del movimiento**; a **mayor** número de **frames, mayor** número de veces que se ejecuta el método **Update** en un lapso de tiempo determinado, de igual forma, a menor tasa de frames menor número de iteraciones. Por tanto, si el personaje se mueve 0.15 unidades por frame, en un segundo se habrá movido diferentes distancias dependiendo de la variación de frames, resultando en un **movimiento más lento o rápido** y produciendo **inconsistencias** en la **experiencia** que debería ser lo más **universal** posible.

Para solucionar este problema, el **movimiento se debe basar en el tiempo** y **no** en el **framerate**, esto es, en vez de que sopesar la **velocidad** del personaje en unidades por frame, es necesario adaptarla a **unidades por segundo**. Esta conversión se calcula de forma muy sencilla **multiplicando** la **velocidad** por el tiempo que ha transcurrido entre el fotograma anterior y el actual, llamado normalmente *'delta time'*. De esta forma, si la cantidad de frames por segundos es menor esto repercutirá en un *'delta time'* mayor dando como resultado una mayor velocidad, así pues, aunque haya un menor número de frames, el personaje se moverá más distancia por cada iteración para compensar y corresponderse con el movimiento objetivo.

De forma inversa, si el framerate es alto, el *'delta time'* será más pequeño y, a pesar de un mayor número de frames, el personaje para compensar se moverá menos distancia en cada iteración debido a una menor velocidad.

```
public Vector3 GetComputedMovement()  
{  
    [...]  
    return (horMovement + vertMovement) * Time.deltaTime;  
}
```

Snippet 10- Frame Independent en el movimiento

Este problema no ocurre solo con la velocidad, sino con toda **variable** que **represente** una **variación** en el **tiempo**, como la aceleración -la velocidad es una variación de la posición, mientras que la aceleración es una variación de la velocidad-. Por tanto, para evitar que el personaje acelere o decelere más o menos rápido dependiendo del framerate, también es necesario multiplicar las aceleraciones y fricciones por el *'delta time'*.

```
private float Approach(float current, float goal, float step)  
{  
    if (current < goal)  
        return Mathf.Min(current + step * Time.deltaTime, goal);  
  
    return Mathf.Max(current - step * Time.deltaTime, goal);  
}
```

Snippet 11- Frame Independent en las aceleraciones y fricciones

Sistema de desplazamientos y saltos

Los **desplazamientos** y los **saltos** se apoyan en el sistema de **movimientos básico** creado, con tan solo **modificar** las variables de **aceleración, fricción y velocidades máximas** se pueden obtener una gran variedad de movimientos nuevos. De este modo, la

implementación se basa simplemente en **calcular nuevos valores** para las variables utilizadas en la simulación **a partir** de los **parámetros** que **definen** a los **desplazamientos**.

Para trabajar de una manera más cómoda se crea una estructura de datos que guarda todas las variables utilizadas en la simulación y que servirá para almacenar los nuevos valores de las aceleraciones y velocidades a la hora de calcular un desplazamiento.

```
public struct MovementData
{
    public Vector2 currentHorSpeed;
    public float currentVertSpeed;

    public float currentMaxHorSpeed;
    public float currentMinHorSpeed;

    public float currentGroundAcc;
    public float currentGroundFric;

    public float currentAirAcc;
    public float currentAirFric;

    public float gravityUp;
    public float gravityDown;
}
```

Snippet 12- Estructura MovementData

A partir de este momento, se accederá a la estructura de MovementData para saber los valores más recientes a utilizar en la simulación. Con esto se consigue que modificar esta estructura sea igual a modificar el movimiento como tal.

RS2 Todos los personajes albergan la información necesaria que define su movimiento sobre cómo desplazarse por el mundo virtual en forma de velocidades máximas, aceleraciones y fricciones. ✓

Todos los **desplazamientos** son creados a modo de **ScriptableObjects** e implementan una misma interfaz para poder tratar a cualquier tipo de desplazamiento de la misma forma haciendo uso del polimorfismo.

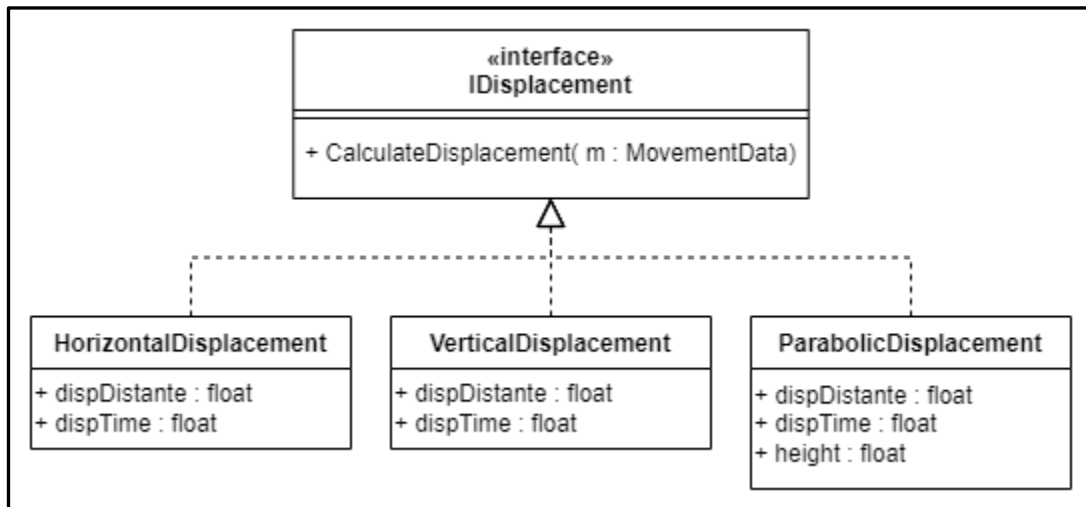


Ilustración 62 - Diagrama de clases de los desplazamientos

Cada **desplazamiento** posee unos **parámetros** propios que lo definen, como la **distancia** o la **duración** de este. El cálculo del desplazamiento se basa en modificar las variables almacenadas en el *struct* de MovementData de tal manera que la simulación cumpla con los parámetros indicados en cada uno de ellos.

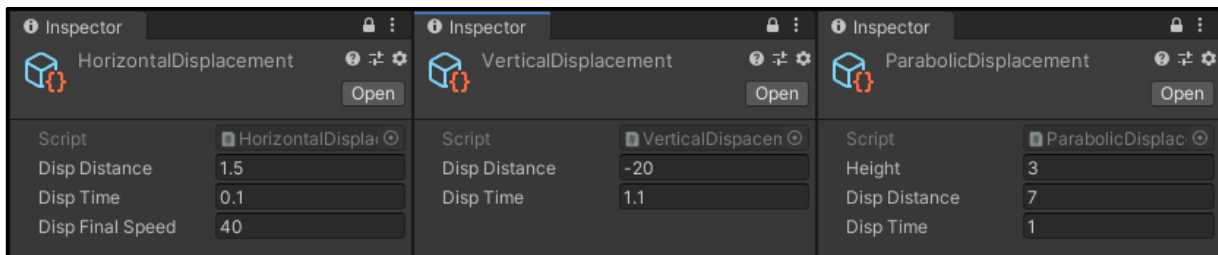


Ilustración 63- ScriptableObjects de los diferentes tipos de desplazamientos: horizontal, vertical y parabólico (de izq. a dcha.)

Para efectuar un desplazamiento se hace uso del método StartDisplacement, en el *script* de CharacterMovement:

```

public void StartDisplacement(IDisplacement displacement, Vector2 worldDirection)
{
    EndDisplacement();
    ForceMoveDir(worldDirection);
    isInHorDisplacement = displacement.isHorizontal;
    isInVertDisplacement = displacement.isVertical;
    displacement.ComputeDisplacement(movementData, worldDirection, c.isGrounded);
    currentDisplacementDuration = displacement.DispTime;
    timeTravelledOfCurrentDisplacement = 0.0f;
}
  
```

Snippet 13- Método StartDisplacement

El **sistema** solo admite la posibilidad de realizar **un desplazamiento a la vez**, de tal manera que al comenzar uno nuevo siempre se detiene el que se estuviese ejecutando con anterioridad en caso de haberlo. Una particularidad de los **desplazamientos** es que **bloquean** el **input** de **movimiento** del personaje para evitar que su trayectoria pueda ser

modificada, para ello, se fuerza a que el vector de dirección del movimiento sea el suministrado al método (*worldDirection*) durante la totalidad de la duración del desplazamiento.

Al **terminar** un **desplazamiento**, todas las **variables** de la **simulación** utilizadas **vuelven** a sus **valores originales** y se libera el input para que el personaje pueda volver a tomar el control sobre su movimiento.

Con lo expuesto hasta ahora, implementar el **dash** y el **sprint** se convierte en algo trivial. Por un lado, el dash no es sino otra cosa que un **desplazamiento horizontal** que se ejecuta cuando el jugador entra en el estado de DashState al accionar el input correspondiente. Por otro lado, el sprint se basa en un **incremento progresivo** de la **velocidad máxima horizontal** a lo largo del tiempo haciendo que el jugador se vaya volviendo más veloz hasta un tope.

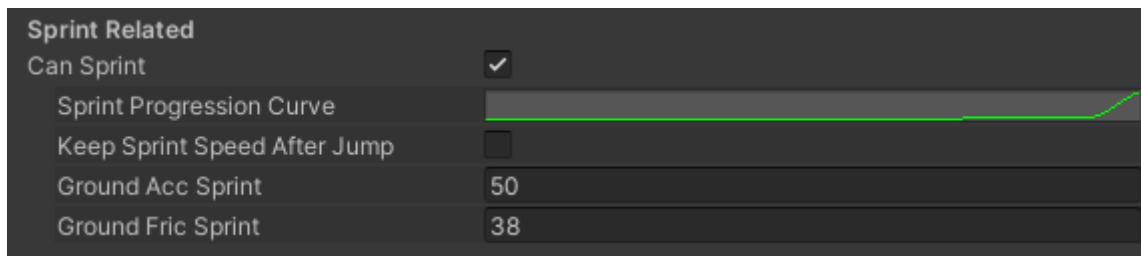


Ilustración 64- Parámetros de configuración del sprint

RF4 El personaje jugador puede realizar un dash. ✓

RF5 Después de realizar un dash, el personaje jugador entra en modo sprint, lo que aumenta su velocidad de desplazamiento máximo. ✓

RF6 Todos los personajes pueden llegar a realizar desplazamientos horizontales o verticales predefinidos en distancia y tiempo en una dirección concreta e inalterable durante el mismo. ✓

Inputed Movement Modifiers

Para **lograr** ciertos **comportamientos** en el **movimiento** es necesario **modificar** el **input** suministrado por el propio jugador (o IA) antes de que sea procesado por la simulación, un ejemplo de ello se tiene en bloquear el input en una dirección forzada determinada como en el caso de los desplazamientos, para ello se utilizan los **inputed movement modifier**.

Estos no son otra cosa que un enumerador que designa la modificación deseada con la que afectar el input:

- **NONE**: no se aplica ningún modificador
- **ZERO**: devuelve un vector cero, de gran utilidad para evitar que el jugador puede moverse.
- **NORMALIZED**: normalmente la velocidad máxima del jugador depende de cómo de desplazados se encuentre los joysticks del mando de su centro, es decir, con un mando se puede conseguir un vector de dirección cuya magnitud sea menor a uno produciendo que la velocidad máxima se reduzca, obligando al personaje ir más lento. Esto es un comportamiento deseado la mayor parte del tiempo, pero cuando el personaje entra en modo sprint, se requiere evitar este efecto normalizando siempre la dirección suministrada por el joystick.
- **FORCED**: devuelve la dirección suministrada previamente al método ForceMoveDir(), evita poder alterar la dirección del movimiento.

Estos modificadores se gestionan en el método GetMovementDirection(), utilizado al principio de esta sección en el método ComputeHorizontalMovement():

```
private Vector2 GetMovementDirection()
{
    switch (CurrentInputModifier)
    {
        case InputModifier.NONE:
            return input.GetMovementDirection();
        case InputModifier.ZERO:
            return Vector2.zero;
        case InputModifier.NORMALIZED:
            return input.GetMovementDirection().normalized;
        case InputModifier.FORCED:
            return forcedInputtedMoveDir;
        default:
            return input.GetMovementDirection();
    }
}
```

Ilustración 65 - Método GetMovementDirection con Inputted Movement Modifiers

Saltos

Aunque los **saltos** puedan entenderse como desplazamientos su **tratamiento** necesita ser bastante **diferente**, en tanto que los saltos **no bloquean** el **input** de movimiento del jugador y están conformados por **dos partes**: la **subida** y la bajada o **caída libre**. La lógica de los saltos se lleva a cabo en un nuevo script, en el JumpFreeFallingHandler.



Ilustración 66- Componente JumpFreeFallingHandler

Para el **cálculo** de los **saltos** es necesario definir **dos gravedades** diferentes como ya se contempla en la estructura de MovementData, estas dos gravedades ayudarán a conseguir un movimiento distinto cuando el personaje suba o caiga al realizar un salto. Con este método es posible lograr una bajada más rápida que la subida dando una sensación de peso en las manos del jugador [68].

Al mismo tiempo, los **saltos** son una pieza muy importante del gameplay y necesitan estar **perfectamente acotados** en cuanto a la **altura** que alcanzan, la **longitud** que puedan llegar a abarcar o incluso el **tiempo** que pueda estar el jugador en el aire. Por ello, los saltos se definen en relación a estos parámetros:

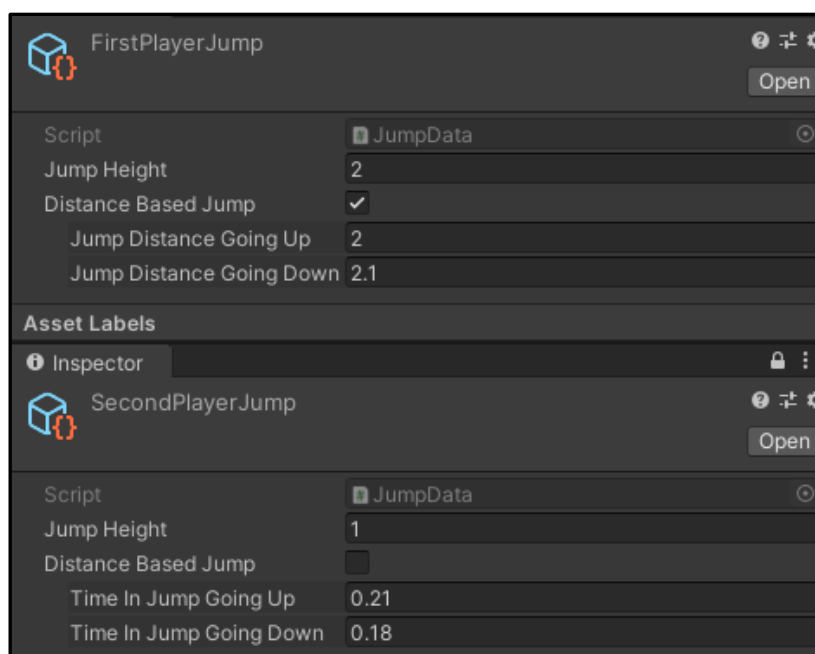


Ilustración 67- ScriptableObjects del primer y segundo salto del jugador. El primero basado en distancia (arriba) y el segundo en tiempo (abajo)

En cuanto al **cálculo** del salto en sí, solo es necesario obtener la **velocidad inicial**, utilizada para establecer la CurrentVertSpeed, que es la que origina el salto como tal, y las

gravidades de subida y bajada que van a ser utilizadas como fricción y aceleración, respectivamente, en el cómputo de la velocidad vertical.

```
public void StartJump(int jumpIndex)
{
    isJumping = true;
    currentJump = jumps[jumpIndex];
    CurrentVertSpeed = currentJump.initVertSpeed;
    movementData.gravityUp = currentJump.gravityGoingUp;
    movementData.gravityDown = currentJump.gravityGoingDown;
}
```

Snippet 14- Método StartJump

Atendiendo a la posibilidad de realizar más de un salto, su implementación es tan simple como mantener una **lista** con los **saltos disponibles** y un **índice** que indica cuál salto ejecutar. Mas atención requiere la gestión de **impedir** que diferentes **saltos** puedan **solaparse**, debido a que la altura total resultante dependería en gran medida del tiempo en el que el jugador acciona el input de salto. Se podría dar el caso en el que nada más empezar el primer salto este se ve interrumpido en favor de ejecutar el segundo, obteniendo una altura completamente distinta a la suma de ambos y distinto a lo que el jugador espera, dando una sensación de inconsistencia que afectaría a la navegación y a la experiencia.

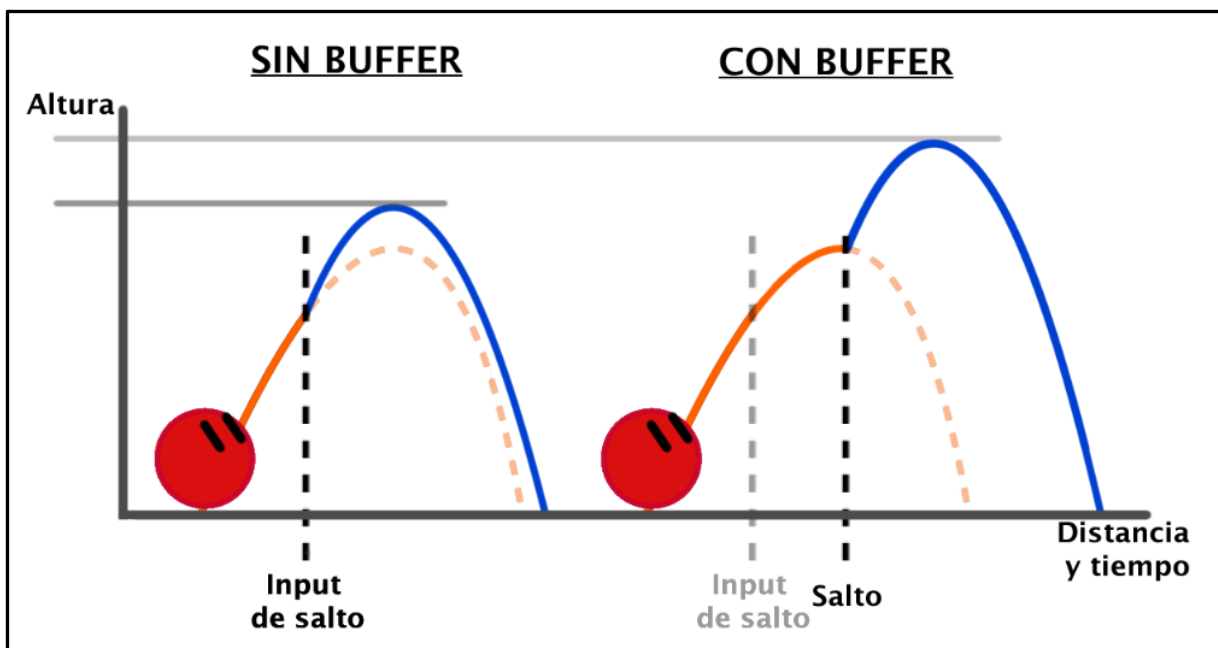


Ilustración 68 - Doble salto sin y con buffer

Para solucionar esta cuestión se emplea un pequeño **buffer** durante la **subida** del **salto**, si el jugador acciona de nuevo el input este se guardará y no se accionará hasta que el salto haya llegado a su punto de máxima altura, momento en el que realizará el siguiente salto. De este modo el solapamiento se convierte en una **concatenación** con un resultado esperable y **consistente**.

RF2 El personaje jugador puede llegar a saltar hasta dos veces antes de tocar el suelo. ✓

Rotar el personaje

Por norma general, un **personaje siempre mira hacia la dirección en la que se mueve**. En esta ocasión se trabaja directamente sobre la *transform* del personaje, aplicando una interpolación para suavizar la rotación.

```
void Update()
{
    Quaternion destination = transform.rotation;
    Vector3 horMoveDir = characterMovement.HorizontalWorldMoveDirection;

    switch (LookAtMode)
    {
        case LookAtMode.AUTO:
            if (horMoveDir.magnitude != 0.0f)
                destination = Quaternion.LookRotation(horMoveDir, Vector3.up);
            break;

        case LookAtMode.FORCED:
            if(forcedDirection.magnitude != 0.0f)
                destination = Quaternion.LookRotation(forcedDir, Vector3.up);
            break;

        default:
            destination = transform.rotation;
            break;
    }

    Quaternion smoothRotation = Quaternion.Slerp(transform.rotation, destination,
rotSpeed * Time.deltaTime);

    transform.rotation = smoothRotation;
}
```

Snippet 15- Rotación del personaje

De forma análoga a cómo se trataba el movimiento, la rotación también posee un modo de **forzar** la **orientación** del personaje en casos de que sea necesario ignorar la dirección del movimiento, como ocurre con el *strafing* o en algunos desplazamientos concretos.

```
public void ForceDirectionTo(Vector2 forced)
{
    forcedDirection = forced.x0y().normalized;
}

public void ForceDirectionTo(GameObject target)
{
    Vector3 dirNoY = (target.transform.position - transform.position).x0z();
    forcedDirection = dirNoY.normalized;
}
```

Snippet 16- Métodos ForceDirectionTo

Implementación de animaciones de locomoción

Una vez obtenida la simulación por la que el personaje puede desplazarse, se procede a sustituir la cápsula por el modelo del personaje 3D y a implementar las animaciones de locomoción.

Para ello, primero se añade el componente Animator en el objeto padre del modelo configurándolo con un AnimationController en el que gestionar todas las animaciones. Debido a que todos los personajes comparten una lógica de movimiento muy similar, se parte de un **AnimationController general** del que derivarán el resto de los controladores de animación. Por tanto, el AnimationController se divide en las siguientes partes:

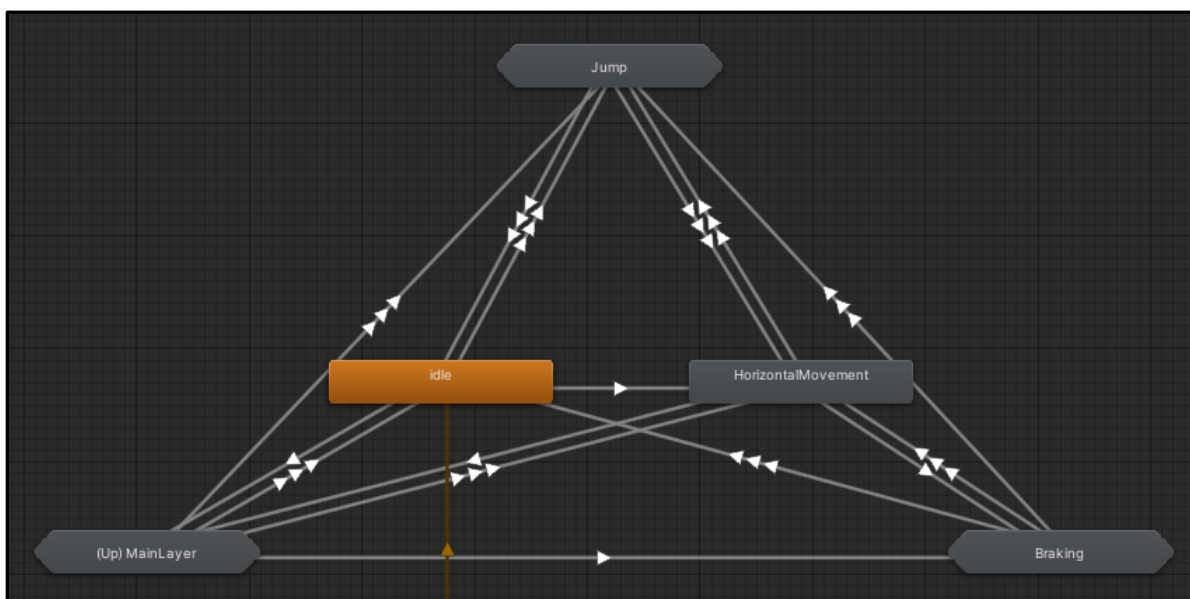


Ilustración 69- Máquina de estados de locomoción básica

Idle: se trata de la animación de **reposo** del personaje, cuando el personaje no recibe ningún tipo de input, ya sea para moverse o realizar alguna acción, se vuelve automáticamente a este estado.

Horizontal Movement: se trata de un **blend tree de una dimensión** que utiliza como parámetro la velocidad horizontal del personaje para mezclar las diferentes animaciones de

andar y correr. Como distintos personajes pueden tener diferentes velocidades máximas se utiliza una **velocidad normalizada** entre cero y uno ($\text{currentHorSpeed} / \text{maxHorSpeed}$)

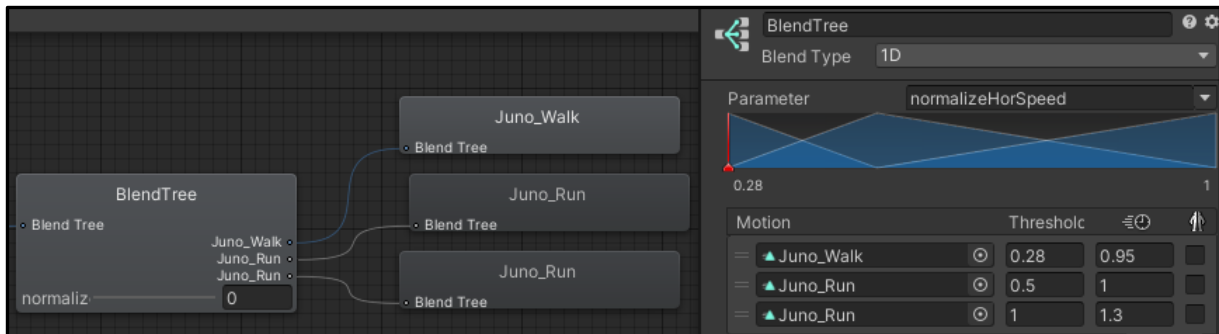


Ilustración 70 - Blend Tree del movimiento horizontal

En el caso de que el personaje pueda realizar **strafing** se utiliza un **blend tree de dos dimensiones** el que se le indica tanto la versión normalizada de la **velocidad** como su **dirección** para indicar la mezcla correcta que realizar.

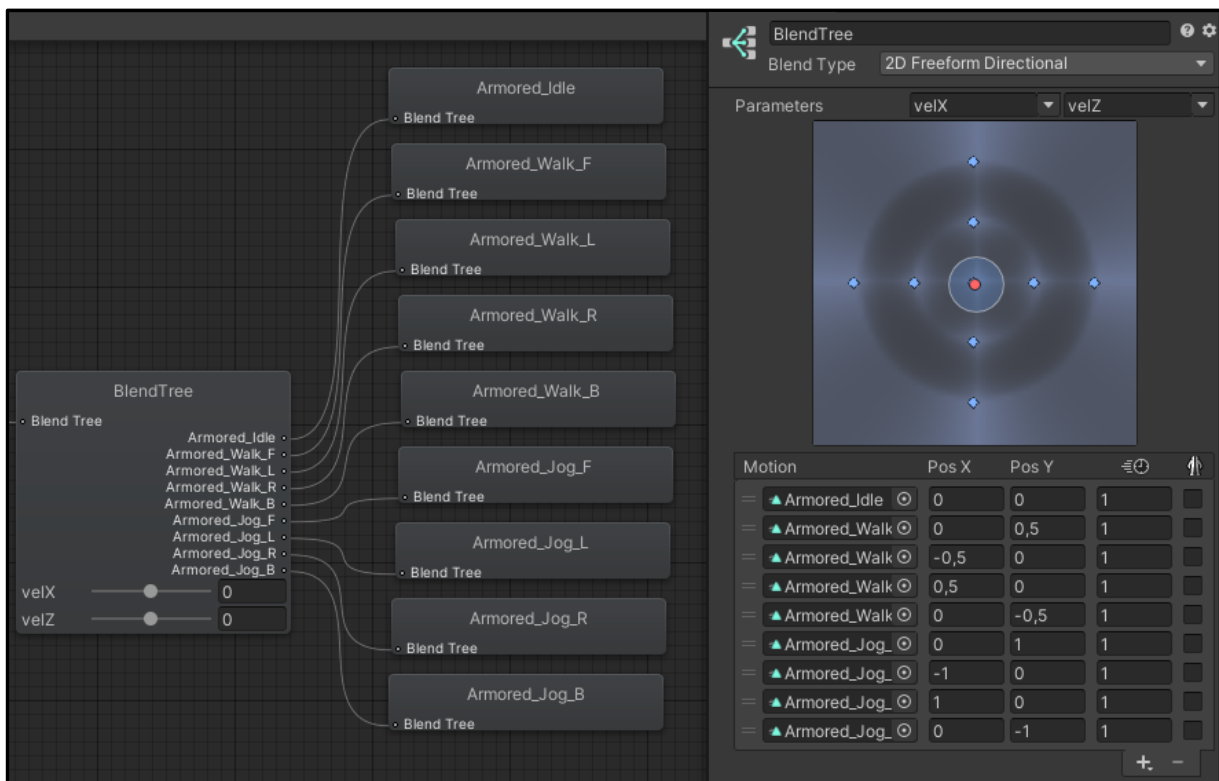


Ilustración 71- Blend Tree para el strafing

Braking: aunque no todos los personajes necesitan estas animaciones debido a su nivel de importancia o detalle en el juego, cuando un personaje deja de moverse, en vez de ir directamente a la animación de repaso, antes pasa por el **blend tree de frenado**. En este nodo se selecciona una animación de frenado u otra dependiendo de la velocidad con la que el personaje se desplazase justo antes de dejar de ingresar algún input de movimiento; a mayor velocidad, mayor exageración en la animación de frenado.

Jump: se trata de una submáquina en el que se gestionan los saltos, la caída y el aterrizaje del personaje. El nodo **“Jump Start”** es un **blend tree**, pero a diferencia de otros, no se utiliza para mezclar animaciones, sino para **acceder** directamente a **animaciones** concretas por medio de un **índice** [69], como en este caso, acceder a la animación de primer o segundo salto. Esta técnica convierte a los **blend tree** en una especie de **array** en el que **guardar** y **seleccionar animaciones** de una forma sencilla y ordenada sin tener que crear un gran número de parámetros y transiciones que compartan las mismas condiciones.

Al igual que en el frenado el nodo **“Land End”** se trata de un **blend tree** que selecciona una animación u otra dependiendo de la velocidad del salto. Como el salto se puede realizar en cualquier momento es más cómodo definir directamente una transición desde el nodo **“Any State”** y manejar sus condiciones, que crear un gran número de transiciones desde el resto de las animaciones en las que se puedan interrumpir por un salto.

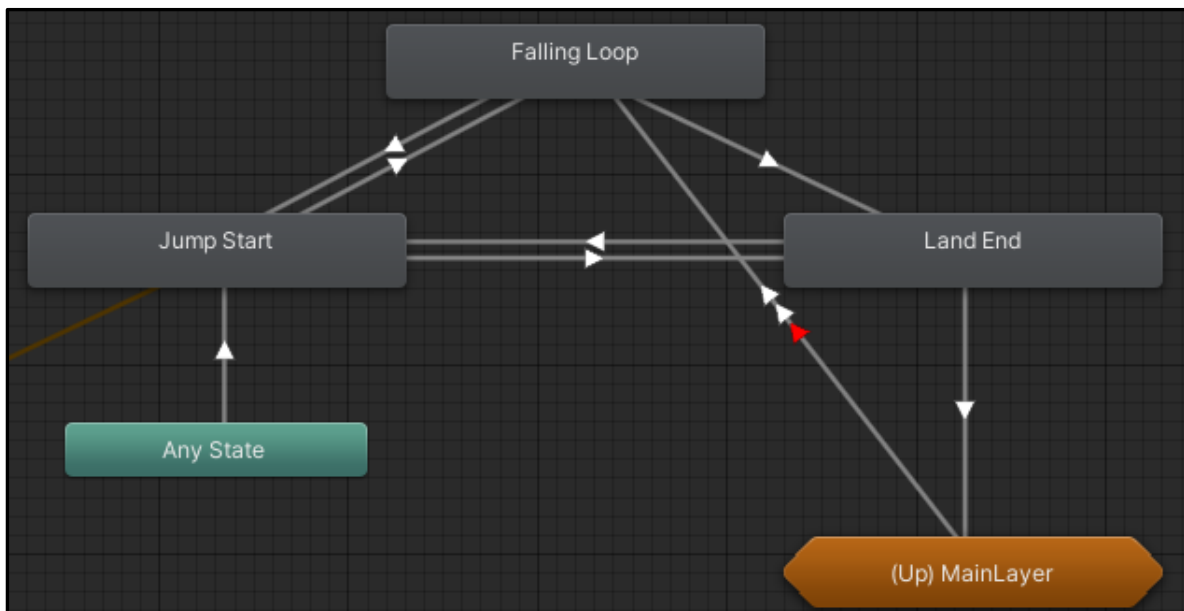


Ilustración 72- Submáquina del salto

El **patrón en triángulo** [69] que refleja la submáquina de estados de salto va a ser bastante recurrente a lo largo del proyecto a la hora de **implementar** una **sucesión** de **animaciones** cuya **duración** es **indeterminada** y que presentan un **inicio** y **final definidos**. En este caso en concreto, el inicio es el salto en sí y el final el aterrizaje, mientras que la sección de duración desconocida es la caída, cuya animación se reproduce en bucle.

A partir de este AnimationController genérico se pueden crear controladores más complejos, con más animaciones, como la utilizada para el jugador, que, debido a su importancia, posee un nivel de detalle mayor. Como la base del controlador y toda la lógica utilizada es compartida por el conjunto de los personajes, se consigue un ahorro de tiempo y esfuerzo considerable a la hora de implementar las animaciones de locomoción del resto de agentes.

RF7 Las animaciones de locomoción deben ir acorde a la velocidad y dirección del movimiento. Estas deben reaccionar a las diferencias entre correr, andar y esprintar, así como al cambio de dirección cuando los personajes realizan *strafing*. ✓

Root Motion

Para hacer uso del **root motion** en *Unity* basta con simplemente activar la casilla de “**Apply Root Motion**” en el **Animator** del personaje y dejar desactivadas las opciones de “**Bake Into Pose**” en las secciones de “**Root Transform**” en las **animaciones** deseadas.

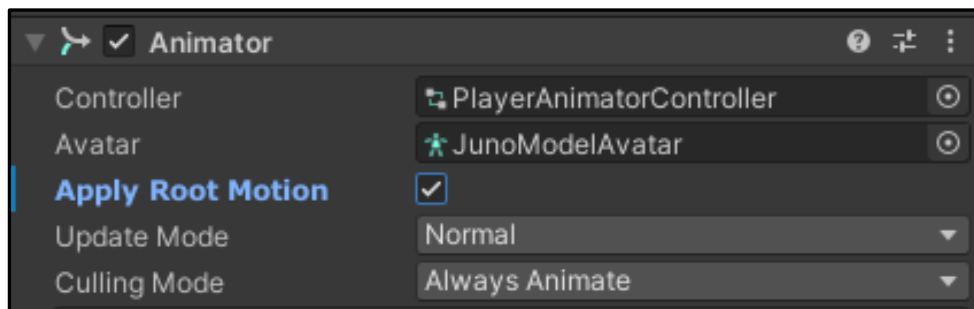


Ilustración 73- Animator con root motion activada

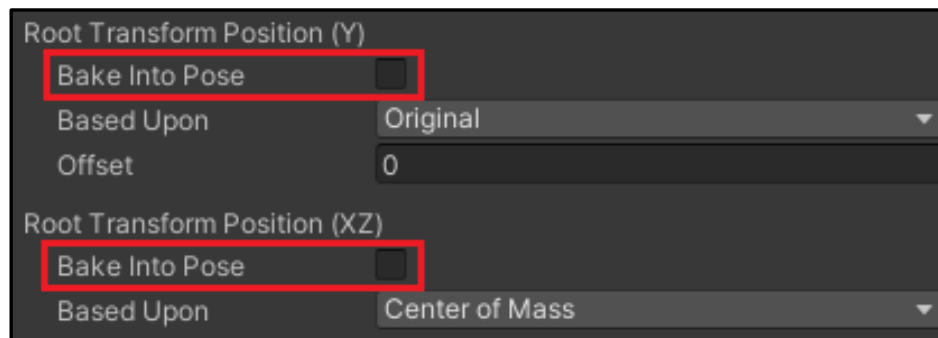


Ilustración 74- Animación con Bake Into Pose desactivada para hacer uso de root motion

Con esto se logra una implementación directa y sencilla, pero este método plantea varios **problemas** como la **imposibilidad** de poder **compatibilizar** el movimiento proveniente del *root motion* con la simulación de movimiento, lo que provoca que solo se pueda utilizar **root motion** o **script motion** en la totalidad de la animación, convirtiendo a estos dos métodos en excluyentes entre sí.

Por este motivo, se procede a realizar una **implementación propia** del *root motion*. Para ello, se crea un script, “*RootMotion.cs*”, en el que se registrará la **velocidad** del **desplazamiento** procedente de la **animación**, aportada por el componente *Animator*, y la **influencia** que tiene el **root motion** en cada momento de la animación.

La **influencia** se **define** en cada una de las animaciones como **curvas de animación** adicionales, en estas curvas se marcan los momentos en los que el *root motion* tendrá lugar. De forma simplificada, **actúan como interruptores** que activan o desactivan el *root motion* a

lo largo de la animación. Al igual que como se hizo con el movimiento básico, el *root motion* también se divide en sus componentes horizontales y verticales.



Ilustración 75- Curvas de animación que activan o desactivan el root motion

Una vez preparada la información, se procede a **extraer** los **datos** del **root motion** para poder realizar la integración con en el sistema de movimiento. En este caso y a diferencia del resto de componentes, el `RootMotion` script necesita situarse en el mismo objeto en el que el componente `Animator` se encuentre, debido a que el método `OnAnimatorMove` [70], que se invoca en cada frame después de que las animaciones hayan sido evaluadas, así lo precisa.

```
private void OnAnimatorMove()
{
    Vector3 localVel = transform.InverseTransformDirection(animator.velocity);
    velocity = transform.TransformDirection(localVel);

    float speed = animator.velocity.magnitude;

    if (speed != 0.0f)
    {
        horInfluence = animator.GetFloat("RootMotionHorInfluence");
        vertInfluence = animator.GetFloat("RootMotionVertInfluence");
    }
    else
    {
        horInfluence = 0.0f;
        vertInfluence = 0.0f;
    }
}
```

Snippet 17- Implementación de `OnAnimatorMove` para obtener la influencia del root motion

Con la **información** de root motion **extraída** solo es necesario **contemplar** estos datos en el **cómputo** del **movimiento** total del `CharacterMovement` como se muestra a continuación:


```
public Vector3 GetComputedMovement()
{
    CurrentHorSpeed = ComputeHorizontalMovement();
    CurrentVertSpeed = ComputeVerticalMovement();

    Vector3 horMovement = CurrentHorSpeed.x0y();
    Vector3 vertMovement = Vector3.up * CurrentVertSpeed;

    Vector3 horRootMotionMovement = rootMotion.velocity.x0z();
    Vector3 vertRootMotionMovement = Vector3.up * rootMotion.velocity.y;

    Vector3 totalHorMovement = horMovement * (1 - rootMotion.horInfluence) +
    horRootMotionMovement * rootMotion.horInfluence;
    Vector3 totalVertMovement = vertMovement * (1 - rootMotion.vertInfluence) +
    vertRootMotionMovement * rootMotion.vertInfluence;

    return totalHorMovement + totalVertMovement;
}
```

Snippet 18- Integración del root motion en el cómputo de movimiento

Como resultado, se obtiene un **sistema de movimiento** que no solo **integra** los dos enfoques de movimiento vistos hasta ahora: **script motion y root motion**, sino que también es **capaz** de **permutar entre ellos** en tiempo de ejecución de una manera fácil y rápida gracias al uso de curvas de animación.

RF9 El sistema de movimiento debe tener la capacidad de alternar entre *scripted motion* y *root motion*. ✓

Ledge Climb

La escalada de salientes se divide en tres fases:

Detección del saliente: para detectar un saliente se realizan varios **Raycast** [71] desde **diferentes localizaciones** entorno al jugador. En total se utilizan tres rayos, uno **horizontal** desde la **cadera** del jugador en su dirección *forward* y otros **dos verticales** hacia abajo que tienen su inicio por encima de la cabeza del jugador y desplazados del centro de este, con esto se quiere conseguir **detectar salientes** que estén por **encima** y un poco por **delante** del jugador.

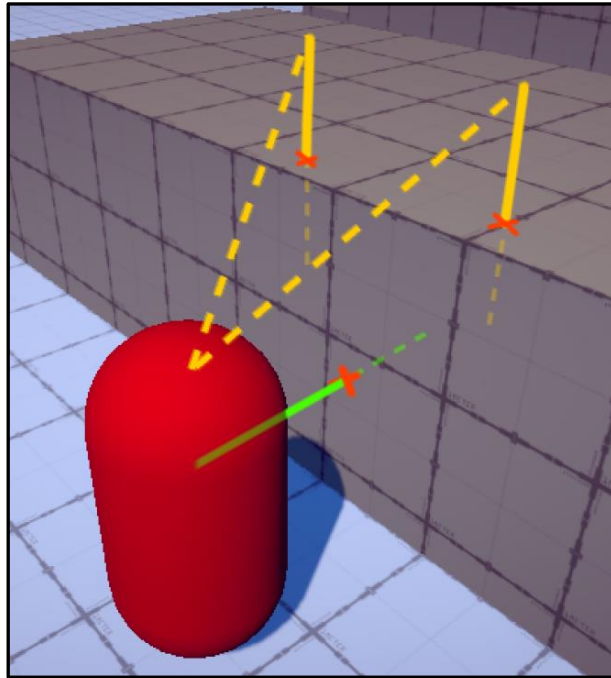


Ilustración 76- Rayos de detección de salientes

Los **rayos** solo comprueban **colisiones** con objetos situados en la **capa física** [72] “**Ledges**” para optimizar el proceso de detección. Adicionalmente, se utilizan objetos invisibles con un simple `BoxCollider` para poder explicitar las zonas a las que son posible agarrarse, en vez de depender de la propia geometría del objeto en sí.

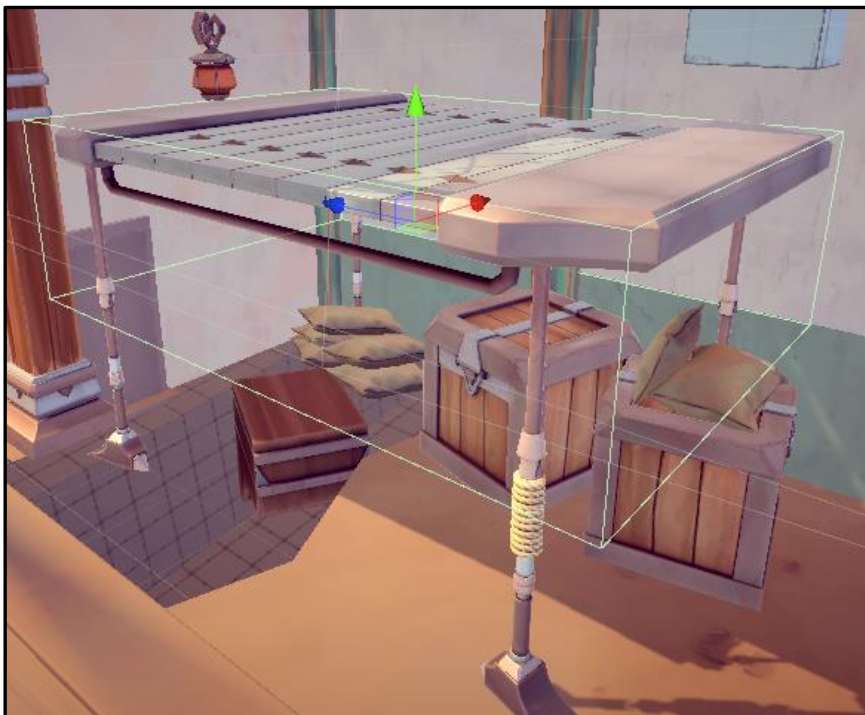


Ilustración 77- Utilización de un collider adicional (en verde) debido a la delgadez de la geometría para que los rayos detecten la plataforma

Ajuste de posición: cuando se detecta un saliente y, al mismo tiempo, el jugador transmite su intención de escalarlo por medio del input de movimiento, el personaje avatar se **ajusta automáticamente** en **posición y rotación** de forma inmediata (snapping) al **saliente**. De esta forma se **evita** que la **animación** de subida y la disposición del **saliente** se **descuadren** independientemente de a qué altura se haya empezado a escalar el saliente.

Movimiento de subida: el movimiento en “L” invertida que sucede en la escalada del saliente es un claro ejemplo de **root motion** y **script motion** trabajando **juntos**. La **subida** de la animación sucede por medio de un **desplazamiento** puramente **vertical**, ya que es necesario tener control sobre la cantidad de altura que puede llegar a subir, pero la **reincorporación**, que es la parte de la animación en la cual el personaje avanza ligeramente en el plano horizontal para situarse encima del saliente, se realiza por **root motion** debido a la exactitud necesaria para hacerlo.

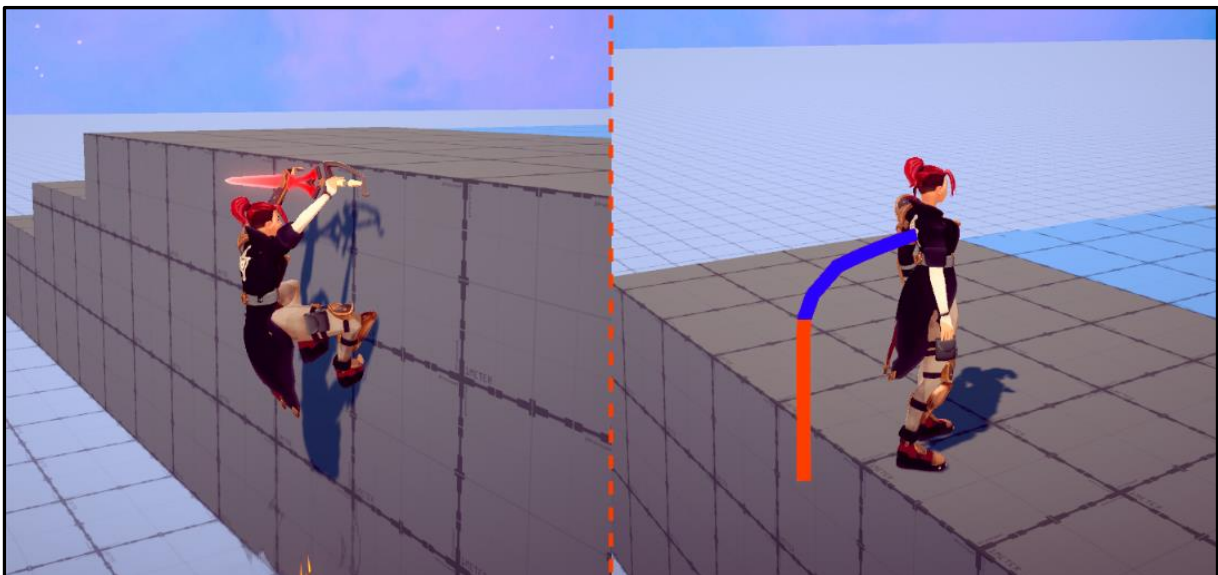


Ilustración 78 - Ajuste de posición a la izq. y movimiento de subida a la dcha., en rojo script motion y en azul root motion

RF3 El personaje jugador puede escalar salientes si este se encuentra cerca del borde de una plataforma o estructura. ✓

Soportar agente de IA

Aunque el componente **NavMeshAgent** [73] de Unity encargado del *pathfinding* proporcione al mismo tiempo de la **capacidad de mover al agente** por el **entorno** virtual, es necesario **desactivar** esta **funcionalidad** para no perder los **beneficios** que aporta el **sistema de locomoción** construido, como la gestión de animaciones de forma genérica o el sistema de desplazamientos.

Siguiendo lo expuesto, es necesario **desactivar** este **movimiento automático** que afecta tanto a la **posición** como a la **rotación** del agente:

```
void Start()
{
    agent.updatePosition = false;
    agent.updateRotation = false;
}
```

Snippet 19- Desactivación del movimiento proporcionado por el NavMeshAgent

Ahora existen **dos simulaciones** llevándose a cabo, la **simulación del movimiento** y la **simulación del pathfinding** para encontrar la siguiente posición a la que mover el agente. Como se ha visto con anterioridad, el **movimiento** solo **precisa** de un input a modo de **dirección** para funcionar, este input viene marcado por la dirección de la **velocidad del agente calculada por el pathfinding**.

```
public Vector2 GetMovementDirection()
{
    return agent.velocity.xz().normalized;
}
```

Snippet 20- Dirección del movimiento marcada por la velocidad del agente

Por último, como se ha **desactivado** la **actualización automática** de la **posición** para evitar que el agente sea desplazado por el sistema de pathfinding, es necesario **suministrar manualmente** la **posición actual** del agente, ya que sino la simulación del pathfinding estaría haciendo sus cálculos de trayectoria a partir de una posición antigua que no representa el estado actual del personaje.

```
void Update()
{
    agent.nextPosition = transform.position;
}
```

Snippet 21- Actualización manual de la posición del agente

RF8 El sistema de movimiento debe contemplar el sistema de *pathfinding* de Unity para los agentes controlados por la IA. ✓

Sistema de combate

El sistema de combate se basa principalmente en la **gestión de animaciones** y la relación que se establecen entre ellas. Aunque exista bastante **funcionalidad fuera** de las **animaciones** para que el sistema funcione, la **mayoría** de esta **lógica** es llamada **a través de Animation Events** en los propios clips de animación, lo que **aporta** al sistema no solo una gran **facilidad de ampliar** nueva **funcionalidad** sino también una amplia capacidad de **iterar** de forma **rápida** sobre las propias **animaciones** de ataques que conforman el sistema.

Máquina de estados y Animator Controller

Lo primero de todo es establecer una **división** clara entre la **máquina de estados lógica** que implementan los **personajes** y los **AnimatorController** usados para gestionar las **animaciones**.

Al igual que se expuso al principio de este capítulo de implementación, existe una **separación** entre el nivel **lógico** y el nivel **visual** dentro de un **personaje**. Aunque el `AnimatorController` pueda parecer una opción válida para implementar a su vez una máquina de estados lógica que maneje el comportamiento de los personajes, este solo es utilizado para administrar las animaciones necesarias y nada más. **Los nodos del `AnimatorController` no representa ningún estado lógico del personaje, solo la animación que designan.**

De tal modo, los **estados lógicos** presentes en la máquina de estados de los personajes **abarcaban una o más animaciones**. Por ejemplo, el estado *“Idle”* comprende todas las animaciones de movimiento existentes y el estado *“Attack”* todas las animaciones que se utilizan para este fin.

Esta **máquina de estados**, implementada por medio del patrón *State* [74], es **común** a todos los **personajes**, y aunque pueda cambiarse un estado en concreto para cumplir las necesidades específicas de cada personaje, siempre presenta la misma estructura y transiciones.

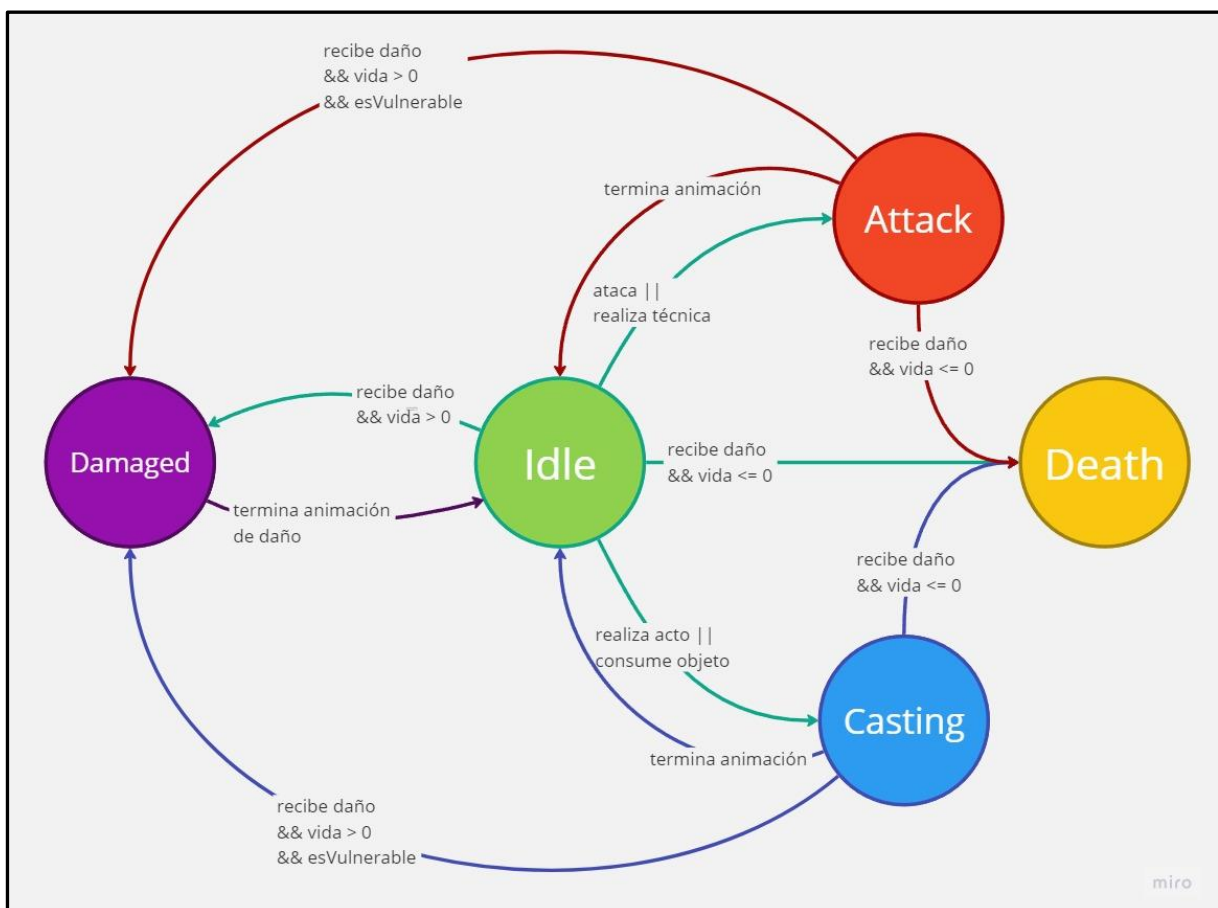


Ilustración 79- Máquina de estados común

Idle: representa tanto el estado de movimiento de los personajes, contando el salto, como el estado de reposo. Este es el único estado que entre personajes cambia debido a que cada uno precisa de su propio estado `Idle` (`SoldierIdleState`, `JunoidleState`, `ArmoredIdleState`, etc.)

Attack: maneja la mayor parte del comportamiento del combate, mientras el personaje esté realizando alguna animación de ataque el personaje se mantendrá en este estado.

Casting: sirve de un modo parecido al estado de “Attack” con la única diferencia que contempla que el efecto de la acción no sea inmediato, se utiliza para la consumición de objetos y la ejecución de actos en el modo concentración.

Damaged: gestiona el momento en el que un personaje resulta dañado por un ataque, en este estado el personaje no puede moverse ni realizar ningún tipo de acción durante un periodo de tiempo determinado.

Death: cuando los puntos de vida de un personaje se reducen a cero se pasa a este estado automáticamente, dependiendo del personaje la muerte se gestiona de formas diferentes.

Existen otros estados como el de “LedgeClimb”, el de “Block” o el de “Lunge” que solo tienen unos personajes en concreto, pero estos se implementan sobre la máquina de estados presentada sin alterar su estructura común.

Componentes

A nivel lógico, el sistema de combate necesita los siguientes componentes para poder operar:

Combat Entity: se encarga de **gestionar y almacenar información** sobre el **ataque actual** que se esté ejecutando: administra las cancelaciones, guarda acciones para realizar más tarde en la propia animación y restablece todos los cambios que se hayan podido aplicar para dar paso al siguiente ataque.

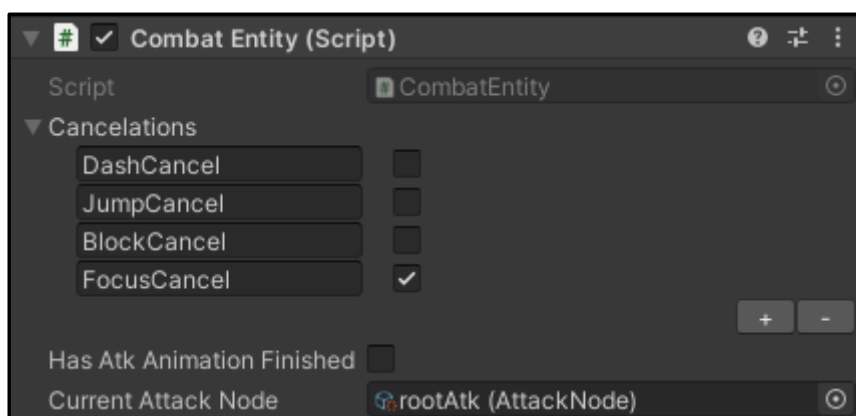


Ilustración 80- Componente CombateEntity

Damage Handler: gestiona el **daño recibido** y la **muerte** del personaje. Adicionalmente, expone eventos a los que poder suscribirse cuando un personaje recibe daño o muere, así como un pequeño modo debug con el que poder revivir, hacer invulnerable o matar directamente al personaje.

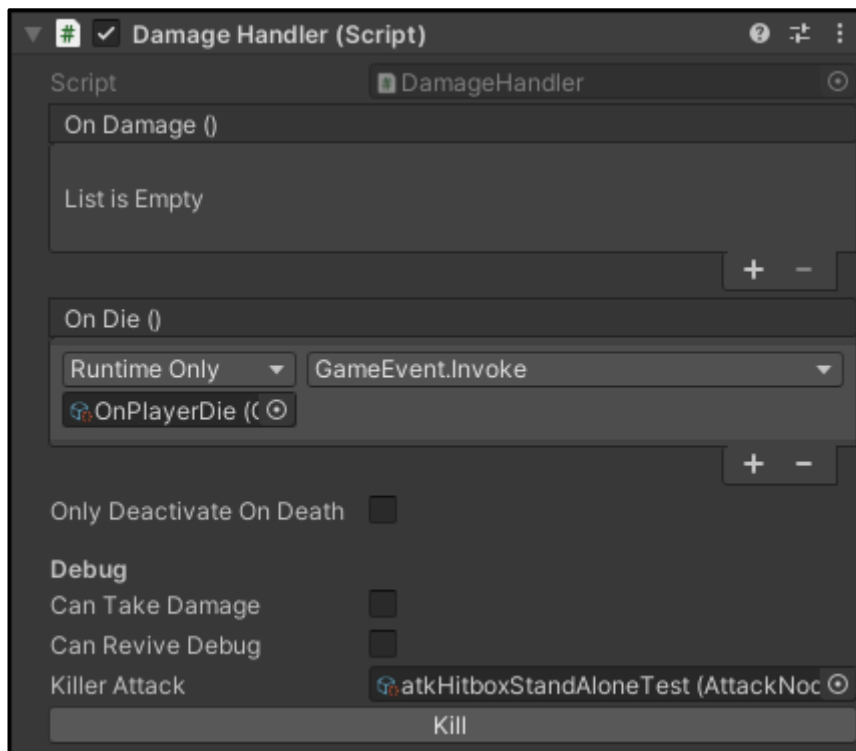


Ilustración 81- Componente Damage Handler

Hitbox Recolector: se encargar de **mantener** una lista con todas las **hitboxes** que el personaje tiene a su disposición, para más tarde poder referenciarlas de manera cómoda desde la CombatEntity durante la ejecución del ataque. Adicionalmente, el HitboxRecolector es una forma sencilla de aplicar cambios a la totalidad de las hitboxes como una amplificación de daño (*damage multiplier*).

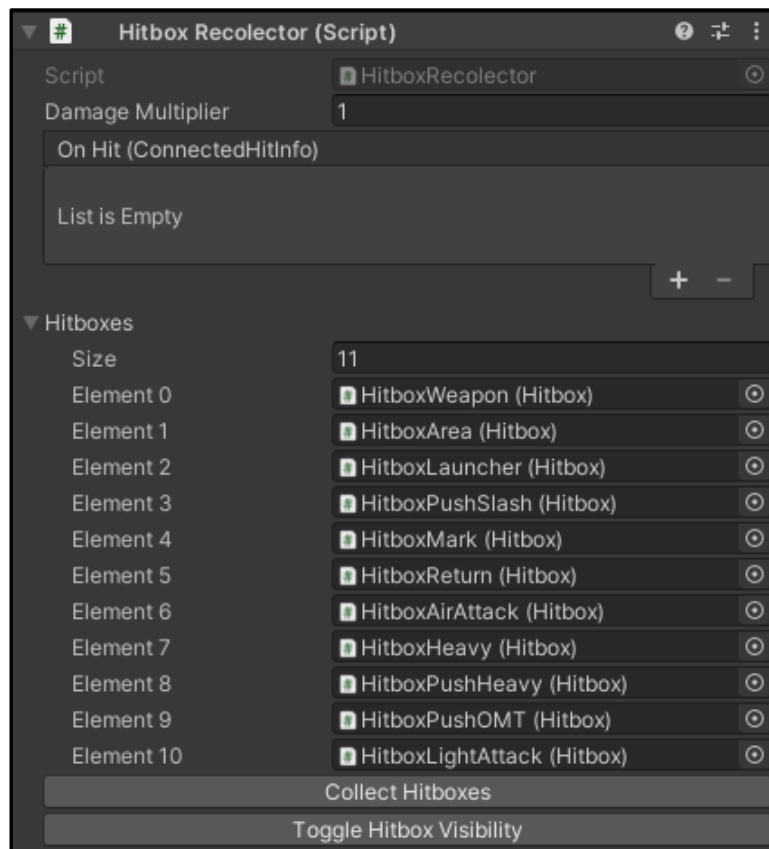


Ilustración 82- Componente HitboxRecolector

Hurtbox Recolector: funciona de forma análoga al HitboxRecolector, aunque no solo mantiene las *hurtboxes* en una colección, sino que también las gestiona activándolas y desactivándolas cuando sea pertinente, como en el caso del dash, en el que todas las hitboxes se desactivan para que el personaje jugador no reciba ningún daño mientras lo efectúa. Además, presenta un modo debug en el que probar diferentes ataques sobre un personaje concreto sin la necesidad de que otro le esté atacando.

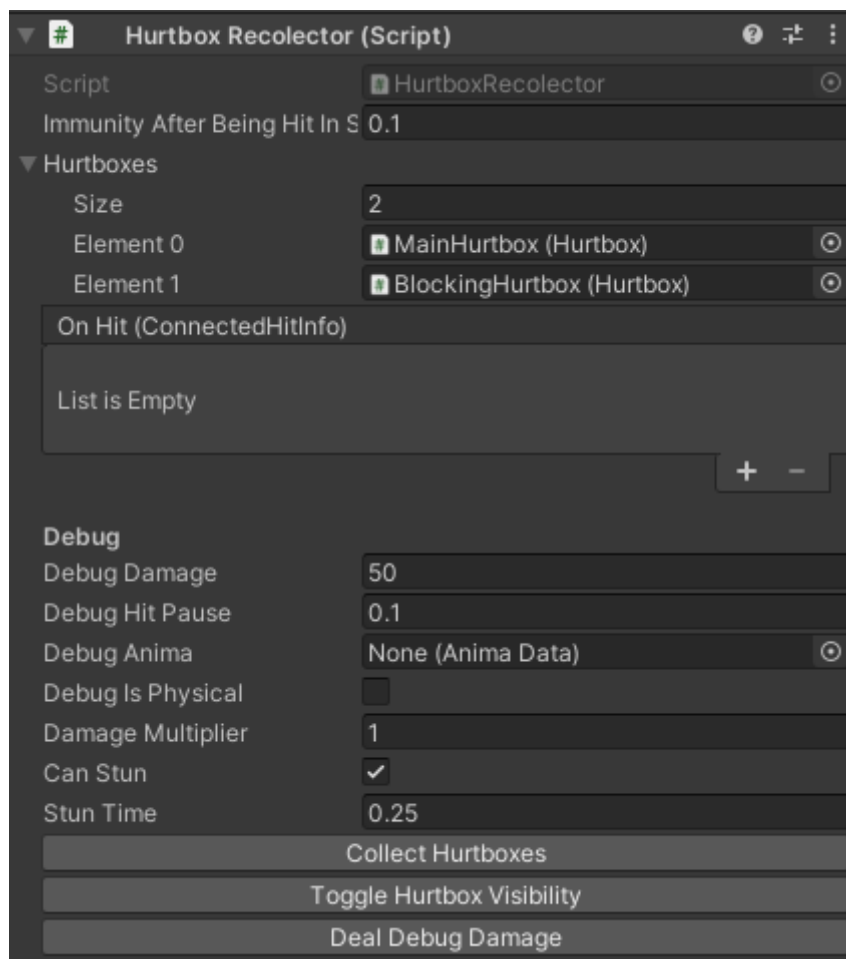


Ilustración 83- Componente HurtboxRecolector

RF19 Durante la realización de un dash, el personaje jugador es completamente invulnerable a cualquier ataque. ✓

Animation Event Utilities: este componente se encuentra situado en el mismo objeto donde se dispone el componente Animator, este actúa de **punto** entre los **scripts** mencionados hasta el momento y las **animaciones**. En él se **guardan** todos los **métodos utilizados por los Animation Events** en las animaciones de ataque, métodos que llaman a la funcionalidad creada en el resto de los componentes: CombatEntity para habilitar cancelaciones y hitboxes, CharacterMovement para aplicar desplazamientos, VFXLocater para generar VFXs... La lista completa de eventos de animación se puede observar en "[Anexo II Animation Events](#)"

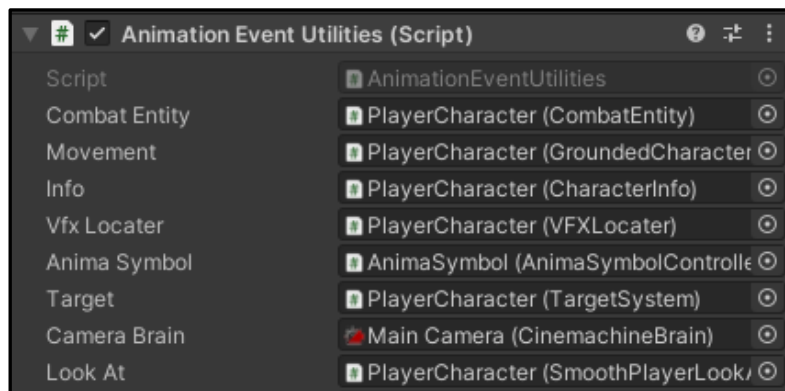


Ilustración 84- Componente AnimationEventUtilities

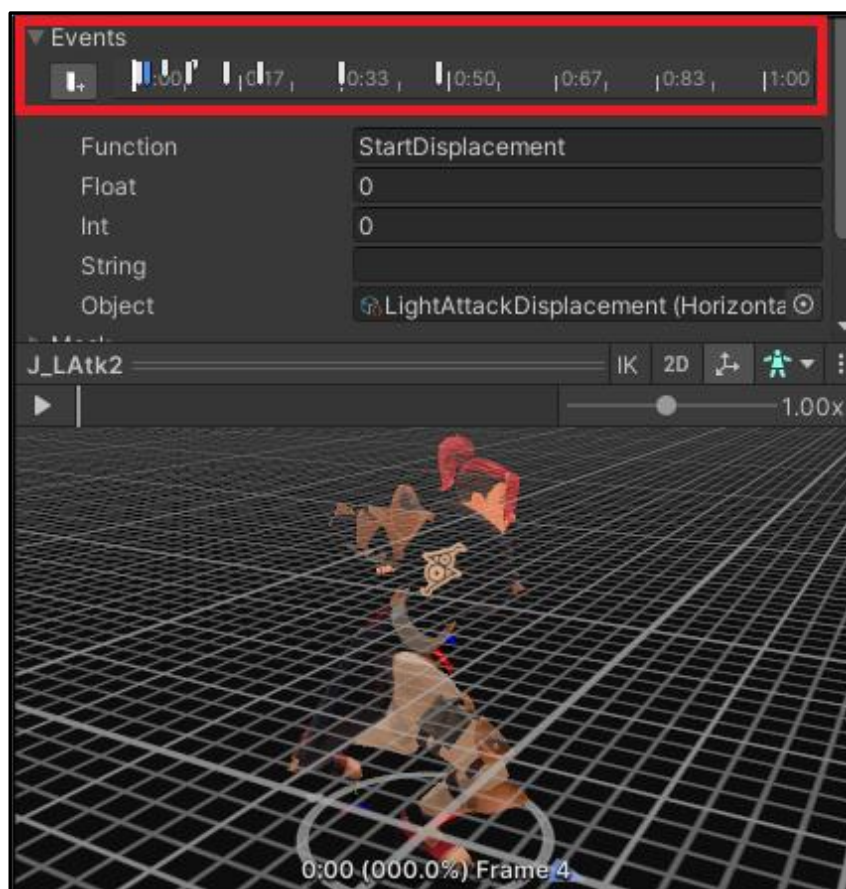


Ilustración 85- Uso de Animation Events en una animación de ataque

Combo Graph

El **combo graph** representa el repertorio completo de **ataques** de un personaje y **define** todas las **cadenas** que pueden llegar a formarse. En vez de programar desde cero una estructura de datos de grafo que haga de **combo graph**, se ha utilizado el **AnimatorController** como uno, en él, cada **ataque** se **representa** por **uno o más nodos de animación** y las **cadenas** se forman a partir de las **transiciones** creadas entre estos nodos.

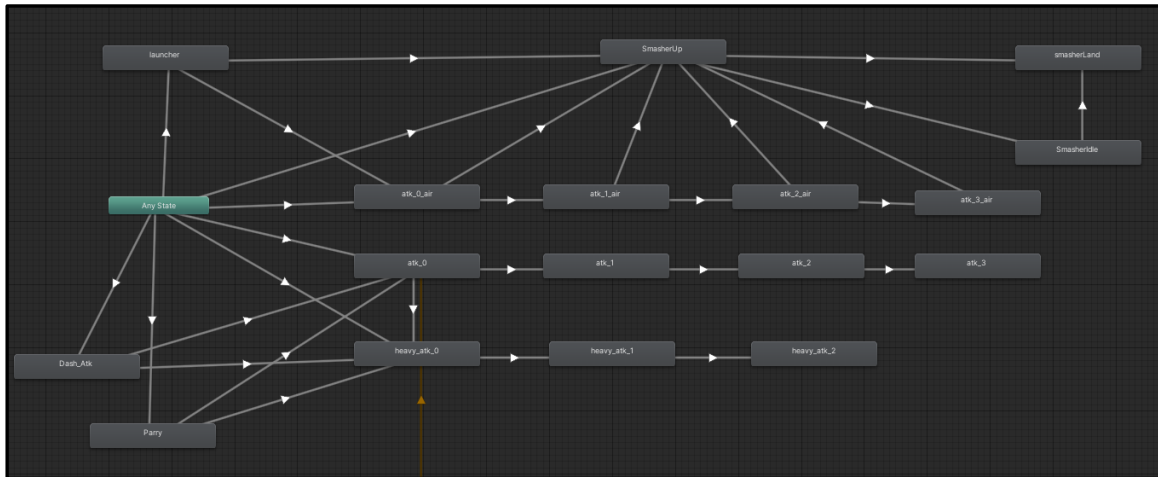


Ilustración 86- Combo graph reducido del repertorio de ataques del personaje jugador, Juno

De este modo, en cada AnimatorController se encuentra una **submáquina** de estados que **representa** el **combo graph** de cada **personaje**.

RS1 Cada uno de los personajes posee un repertorio de ataques diferentes definido por un **combo graph**. ✓

Ataque individual

Todo **ataque** de forma individual se **representa** por medio de un **AttackNode**, se trata de una **estructura** de datos en forma de **ScriptableObject** utilizada para **mantener** toda la **información necesaria** a la hora de que un **golpe conecte con éxito**.

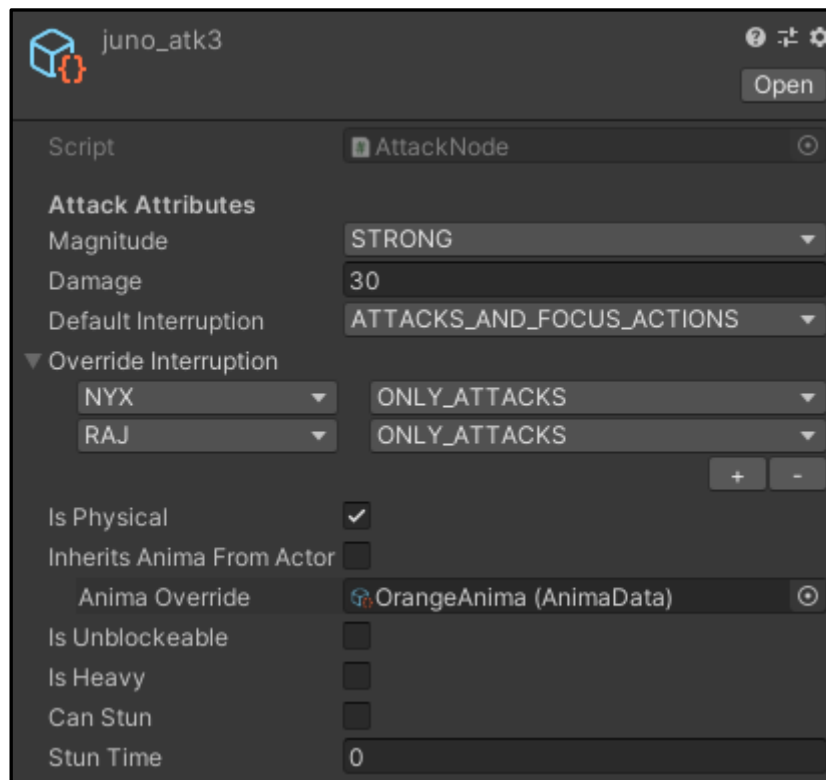


Ilustración 87- Ejemplo de AttackNode

Magnitude: define la fuerza del golpe, existen diferentes tipos de magnitud y dependiendo de ella se aplicará un knockback u otro cuando el ataque conecte. Esta propiedad tiene como único objetivo **indicar el knockback a aplicar** y no modifica en ninguna medida el daño que pueda infligir el ataque.

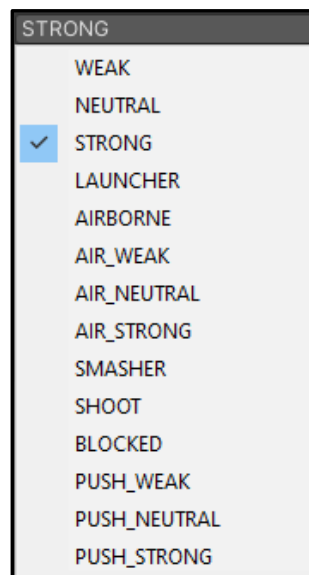


Ilustración 88- Todos los tipos de Magnitud existentes

Damage: cantidad de daño a aplicar cuando el golpe conecte

Default Interruption: define el rango de acciones que un ataque puede interrumpir al personaje que recibe el daño. Es decir, si el personaje atacado recibe un impacto durante una acción, esta se verá interrumpida si el ataque recibido así lo indica.

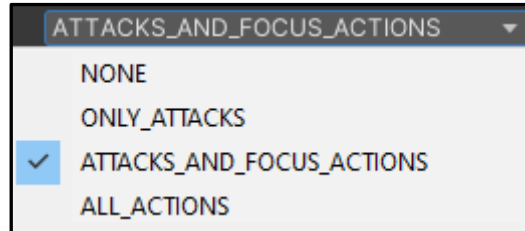


Ilustración 89 - Todos los tipos de Interrupción existentes

Override Interruption: sobrescribe la interrupción por defecto dependiendo del personaje al que se ataca.

Is Physical: si el daño infligido es solo físico, a contraposición del daño elemental.

Inherits Anima From Actor: define si el ánima del ataque es el mismo que el ánima del personaje que lo ejecuta, ya que no siempre el ataque es del mismo ánima que el personaje como en el caso de Nyx y sus ánimas esclavas.

Anima Override: en caso de que el ataque no herede el ánima, se le especifica una en concreto que debe tener en cuenta para el sistema elemental de fortalezas y debilidades.

Is Unblockeable: indica si el ataque no es bloqueable.

Is Heavy: define si un ataque es pesado, utilizada para indicar cuando un impacto puede dañar escudos como el del soldado blindado.

Can Stun: si el ataque puede aturdir independientemente del ánima que utilice, ignorando el sistema de fortalezas y debilidades.

Stun Time: la duración del aturdimiento que aplica al personaje golpeado.

Estos **AttackNodes** se **asignan** en cada uno de los **nodos** de animación del **AnimatorController** que forman el combo graph por medio de los **StateMachineBehaviours**⁷, de esta manera, cuando una *hitbox* detecta colisión contra

⁷ La lista completa de los **StateMachineBehaviours** utilizados a lo largo del proyecto se puede observar en "[Anexo III](#)"

una *hurtbox* solo necesita mirar el **AttackNode** actual **establecido** al **comienzo** de la **animación** para saber la cantidad de daño a aplicar, el anima, la magnitud, etc.

Además de configurar el nuevo AttackNode, en los **nodos** de **animación** también se llevan a cabo otras tareas como indicar el **inicio** de una **nueva animación de ataque** o **restablecer** los valores de los **parámetros** del AnimatorController que hayan sido modificados por animaciones anteriores.

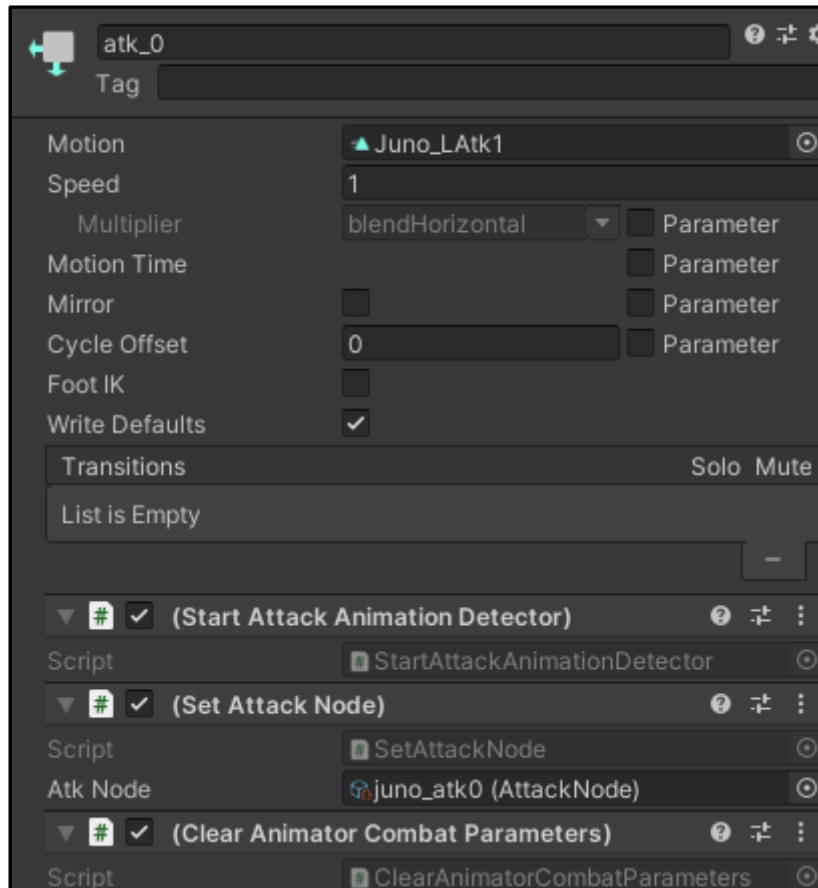


Ilustración 90- Ejemplo de nodo de animación con algunos StateMachineBehaviours: StartAttackAnimationDetector, SetAttackNode y ClearAniamtorCombatParameters

Hitboxes y Hurtboxes

Las **hitboxes** y **hurtboxes** **no** son simples **componentes** de **colisión**, sino que son **GameObjects propios** situados dentro de los *prefabs* de los personajes cuyo objetivo es determinar si un ataque ha impactado con éxito.

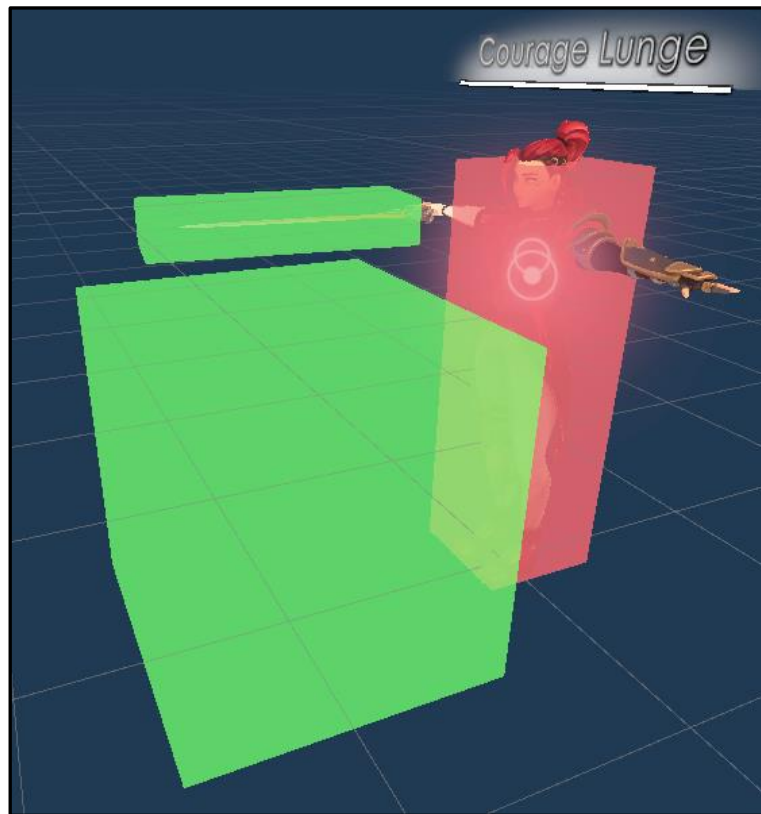


Ilustración 91- Disposición de Hitboxes (verde) y Hurtboxes (rojo) sobre el personaje jugador

Aunque una *hurtbox* es muy sencilla, ya que solo tiene que esperar a que colisionen contra ella y dar una respuesta de si el golpe ha sido satisfactorio o no, la *hitboxes* mantienen casi toda la lógica en cuanto a la hora de detectar el impacto.

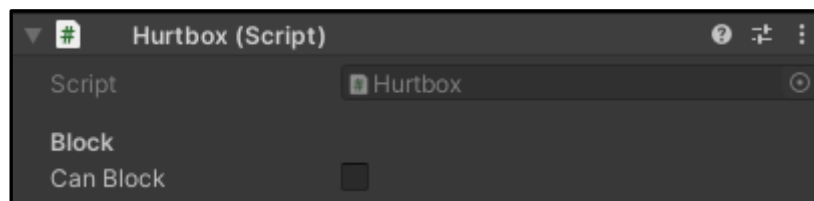


Ilustración 92 - Componente Hutbox

Las **hitboxes** no utilizan los **callbacks** de `OnTriggerEnter/OnTriggerExit` [75] propios de las **colisiones básicas** de Unity, sino que realizan **overlaps** [76] en cada *frame* para detectar las hurtboxes. Este método permite **detectar** de forma robusta **colisiones** de **hitboxes** que se **mueven**, ya que dejan un **rastro** (*trail*), una estela, que sigue generando **colisión** para evitar problemas de detección.

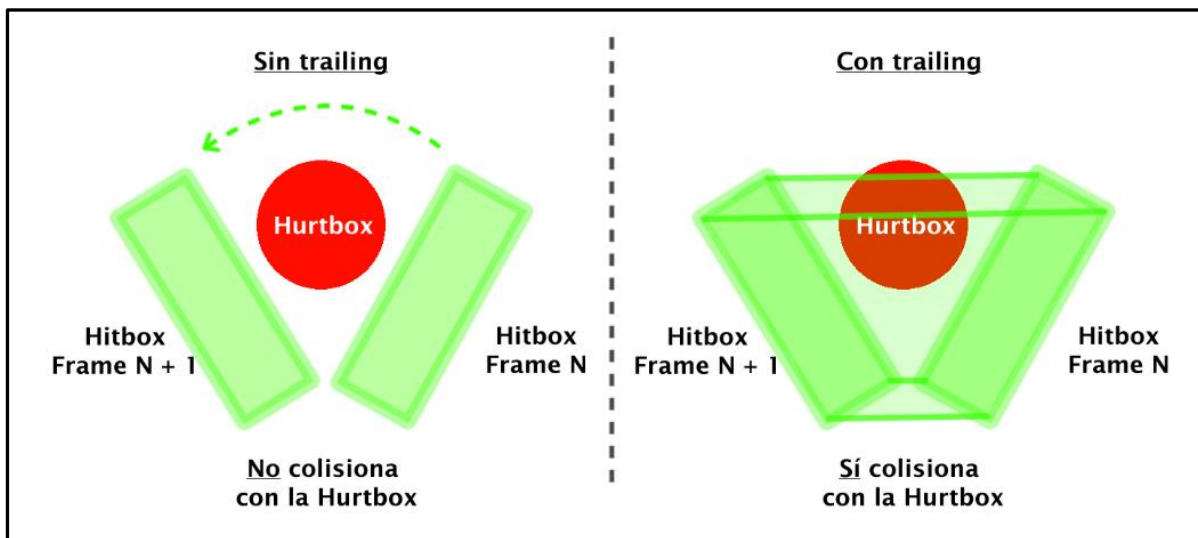


Ilustración 93- Diferencia entre hitboxes sin trailing y con trailing

Las **hitboxes**, a diferencia de las hurtboxes, **no están activas en todo momento**, solo se habilitan durante **secciones específicas** en la **animación** de ataque por medio de AnimationEvents, en estos eventos se especifica la hitbox determinada que se quiere activar o desactivar. Cuando una hitbox se activa se le asigna automáticamente el AttackNode correspondiente del ataque a realizar.

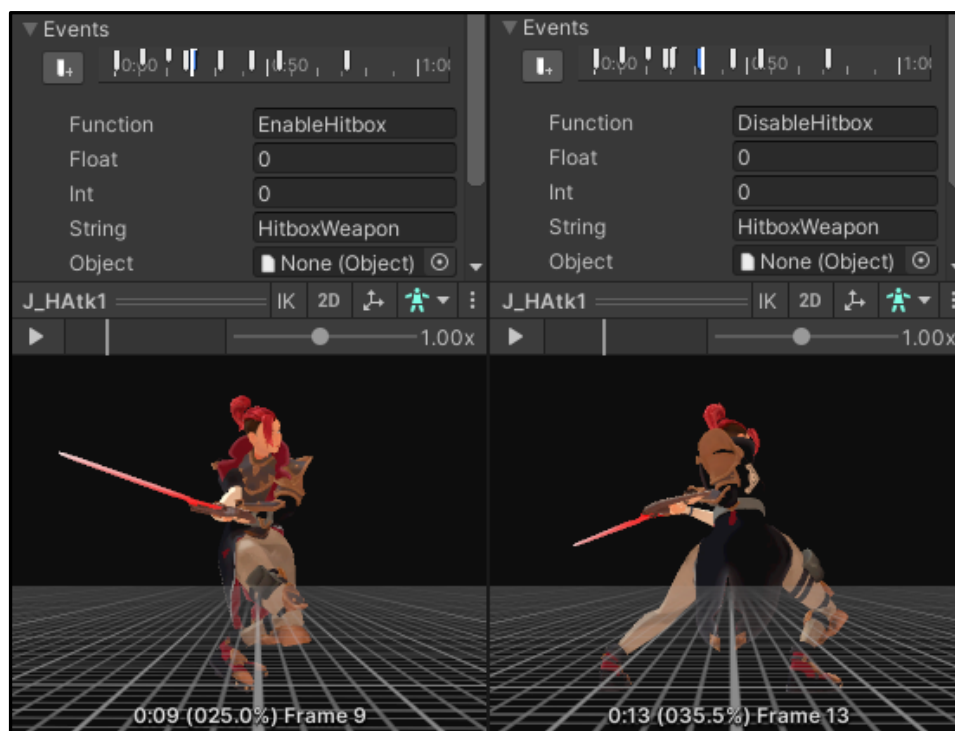


Ilustración 94- Eventos EnableHitbox (Fram 9) y DisableHitbox (Frame 13)

Las hitboxes puede tener comportamientos diferentes dependiendo de lo que se les indique en su script:

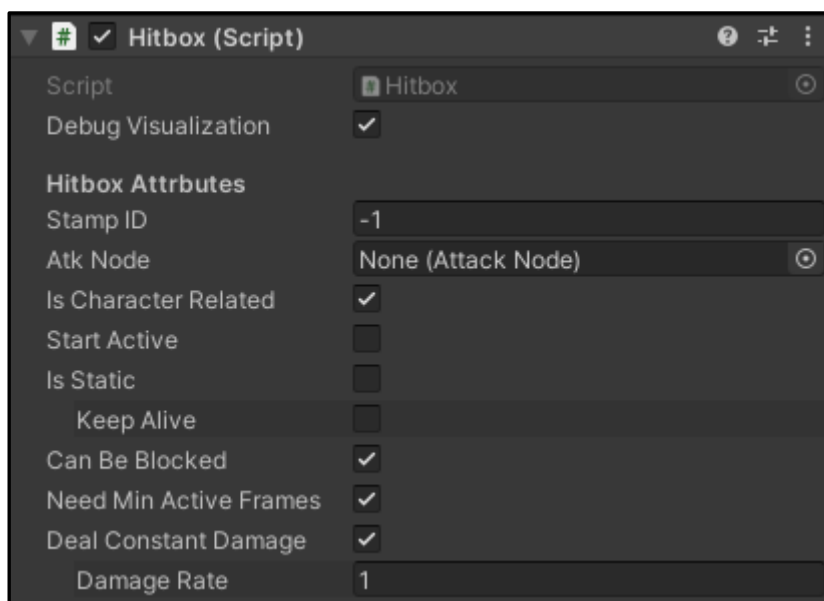


Ilustración 95- Componente Hitbox

StampID: marca de tiempo utilizada para registrar la *hitbox* en el *HurtboxRecollector* y evitar que inflija daño más de una vez si permanece más de un frame colisionando con una *hurtbox*.

Atk Node: el *AttackNode* asignado a la *hitbox*.

Is Character Related: indica si la *hitbox* se mantiene emparentada al personaje cuando esta se activa o bien se desprende de él quedándose en el sitio, aunque este se mueva.

Start Active: indica si la *hitbox* empieza activada sin necesidad de eventos de animación, muy útil para proyectiles, ya que carecen de animaciones.

Is Static: indica si la *hitbox* se mueve mientras se encuentra activa o si se mantiene estática. Normalmente una *hitbox* estática se desactiva una vez haya golpeado a uno o varios objetivos en un instante de tiempo concreto, pero debido a que las dinámicas tardan un tiempo en realizar el recorrido completo, durante el cual pueden golpear a diferentes enemigos, estas se deben mantener activas para detectar todas los posibles impactos.

Has trailing: si la *hitbox* es dinámica esta variable define si se quiere generar la estela de colisión para una mejor detección.

Keep Alive: en caso de que se quiera mantener activa la *hitbox* estática después de detectar un impacto.

Deal Constant Damage: si la *hitbox* inflige daño constante mientras se mantiene activa, ignorando el *StampID*.

Damage Rate: indica cada cuánto tiempo en segundos la *hitbox* inflige daño.



A continuación, se expone la secuencia de acciones que tiene lugar cuando una *hitbox* colisiona contra una *hurtbox* para infligir daño:

1. Se **comprueba** que la **hurtbox** y **hitbox** **pertenezcan** a un tipo de **personaje diferente**, para evitar que un personaje pueda infligirse daño a sí mismo o a sus compañeros, en el caso de los enemigos.
2. Se **consulta** a la **hurtbox** la **respuesta al impacto**, esta recurre al `HurtboxCollector` para indicar si el impacto ha sido exitoso, bloqueado o si no se ha registrado - en el caso en el que personaje sea invulnerable o la hitbox ya haya colisionado con anterioridad-.
3. Si el **impacto** ha resultado **exitoso** o **bloqueado**, la **hitbox** se encarga de **crear** una estructura de datos llamada **ConnectedHitInfo** que **incluye** toda la **información** pertinente al **golpe** en sí, incluido el `AttackNode`.
4. Se **llama** a un **evento OnHit** [74], tanto en el personaje **atacante** como en el personaje **atacado**, cuyo único argumento es la recién creada **ConnectedHitInfo** para poder **distribuir** la **información** del **golpe** a todo el que se suscriba a este evento.
5. El **DamageHandler**, suscrito al evento de **OnHit**, **procesa** la información guardada en la **ConnectedHitInfo**. Si la **respuesta** dada con anterioridad por el **HurtboxRecollector** ha sido **exitosa** entonces **aplica** el **daño** y pasa al personaje al **estado** de **"Damaged"**, en el que se gestionará el **knockback**, o al **estado** de **"Death"**.

Cadenas de ataques y cancelaciones

Las **cadenas de ataques** se generan por medio de **cancelaciones** entre las **diferentes animaciones involucradas**, normalmente la etapa de **recovery** se **corta** para **dar paso** al **wind-up** o directamente a la fase de **attack** de la **siguiente**. Para **evitar** que el jugador pueda ejecutar una **cadena** de ataque en bucle de manera **ilimitada** se usan **finishers** al final de la sucesión, los cuales deben reproducir la animación entera sin la posibilidad de poder interrumpirse para realizar otra acción. Algunos **finishers** son: el último ataque del combo ligero y del combo pesado o el *smasher*.

Las cancelaciones no solo sirven para generar cadenas, también son utilizadas para **interrumpir** cualquier **ataque** y **dar paso a una acción diferente** como puede ser **bloquear**, realizar un **dash** o un **salto** o, simplemente, **moverse**. Debido a la filosofía de diseño en *"Arcanima"*, la mayoría de los ataques son cancelables en otras acciones, lo que aporta al jugador una gran sensación de respuesta inmediata que beneficia a la experiencia objetiva deseada.

Las **cancelaciones** se definen en cada una de las animaciones y están formadas por una **dupla** de **eventos de animación** que **delimitan** el **principio** y **fin** de una **ventana** de tiempo en la que el ataque actual puede verse **interrumpido** por la **acción especificada**, estos

eventos cambian el valor de los booleanos que representan la cancelaciones determinada en el componente `CombatEntity` visto con anterioridad:

- ***Enable/DisableAnimationCancel***: definen si el ataque puede cancelarse en el siguiente ataque de la cadena.
- ***Enable/DisableDamageCancel***: delimitan secciones en la que el personaje puede ver su ataque interrumpido cuando este recibe daño.
- ***Enable/DisableMoveCancel***: indican si el ataque puede ser interrumpido simplemente por moverse. Estos eventos se suelen disponer al final de la animación, cuando las hitboxes han sido desactivadas y el ataque ya ha tenido lugar. Este tipo de cancelación evita que el jugador tenga que esperar a que la animación se reproduzca en su totalidad si está ingresando algún input de movimiento durante el *recovery* del ataque [77]. Con esto, nuevamente, se consigue una buena sensación de respuesta.
- ***Enable/DisableCustomCancel***: a diferencia de las otras cancelaciones vistas, de las que pueden hacer uso cualquier personaje, estas cancelaciones son propias de cada uno. Por ejemplo, al contrario que los enemigos, el personaje jugador puede cancelar un ataque para realizar un salto, un dash o un bloqueo. A estos eventos se les indica la cancelación deseada a través de los parámetros de los `AnimationEvents`.

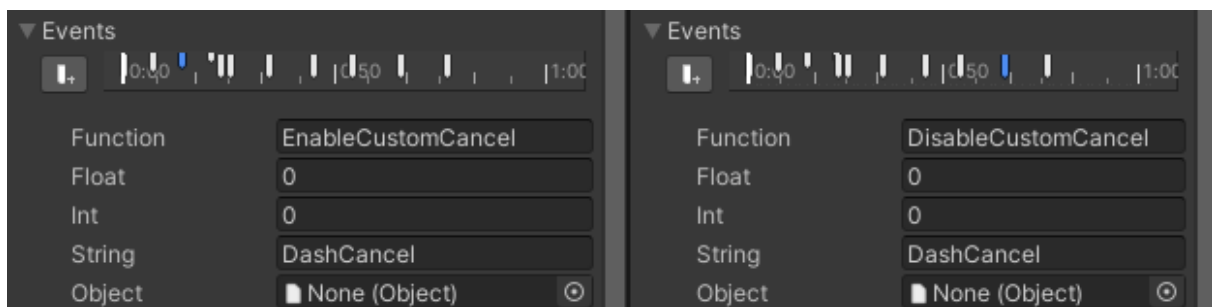


Ilustración 96- Ejemplo de CustomCancel para cancelar la acción en un dash

Para saber si una acción puede interrumpir a la actual se consulta el valor de estas cancelaciones a través del `CombatEntity` en los componentes que implementan la funcionalidad de la acción a ejecutar.

```
public bool IsAbleToDash()
{
    return !isDashing && combat.CanCancelOn("DashCancel");
}

public bool IsAbleToJump(int jumpCount)
{
    return jumpCount < maxNumOfJumps && combat.CanCancelOn("JumpCancel");
}
```

Snippet 22- Incorporación de la cancelación para el salto y el dash

Normalmente las cancelaciones vienen precedidas por un período de escucha durante el cual todo input introducido se almacena en un buffer. Cuando se activa una ventana de

cancelación, se **atiende inmediatamente** al **buffer** para **ejecutar** cuanto antes las **acciones** previamente **ingresadas**. Esto aporta al jugador una gran comodidad para cancelar ataques, ya que no es necesario que ingrese el input justo cuando la ventana de cancelación se encuentre activa, sino que el sistema recuerda sus intenciones. Estas secciones de escucha también vienen delimitadas por eventos en cada animación (Enable/DisableInputListening)

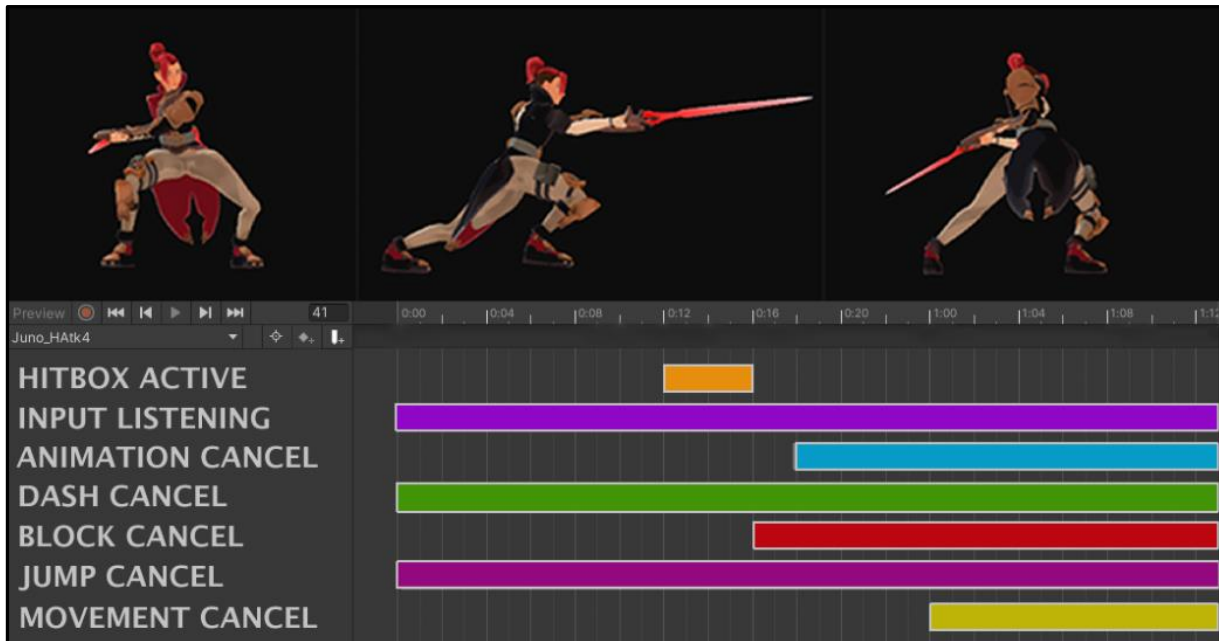


Ilustración 97- Ventanas de cancelación y activación en una animación de ataque

RF10 Cualquier personaje debe tener la capacidad de concatenar ataques. ✓

RF11 Cualquier ataque debe tener la posibilidad de cancelar su propia animación con el fin de poder realizar otra acción como bloquear, moverse, saltar o atacar nuevamente. ✓

Desplazamientos y Knockbacks

Gracias al sistema de movimiento se pueden utilizar los desplazamientos a lo largo del sistema de combate. En las **animaciones** de ataque se puede **ejecutar** cualquier tipo de **desplazamiento** con el **evento "StartDisplacement"**, ya sea horizontal, vertical o parabólico, al cual se le pasa el ScriptableObject correspondiente como parámetro.

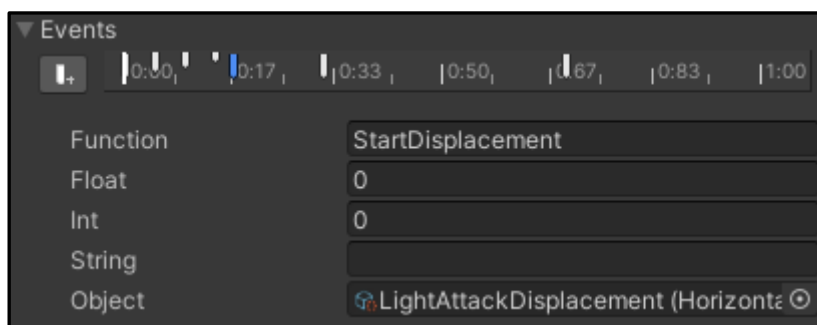


Ilustración 98- Evento StartDisplacement

Adicionalmente, a través de los **eventos** se pueden **modificar** ciertas **características** del **movimiento** para adaptarse a las necesidades del propio ataque, como **forzar** que el personaje mire hacia una **dirección**, desactivar y activar el **strafing** o incluso **deshabilitar** el **movimiento vertical** por completo.

Por otro lado, los **knockbacks** se tratan como **desplazamientos** que se ejecutan cuando un **personaje recibe daño** y ve **interrumpidas** sus **acciones**. Estos desplazamientos son diferentes **dependiendo** de la **magnitud** del **ataque** recibido, con esto se consigue un efecto de acción-reacción que aporte mayor fuerza al impacto en el que cuanto mayor sea el golpe mayor será el knockback.

Cada **personaje mantiene** una **colección** que **relaciona** cada **magnitud** posible de un ataque con un ScriptableObject que define el tipo de **knockback** a aplicar.

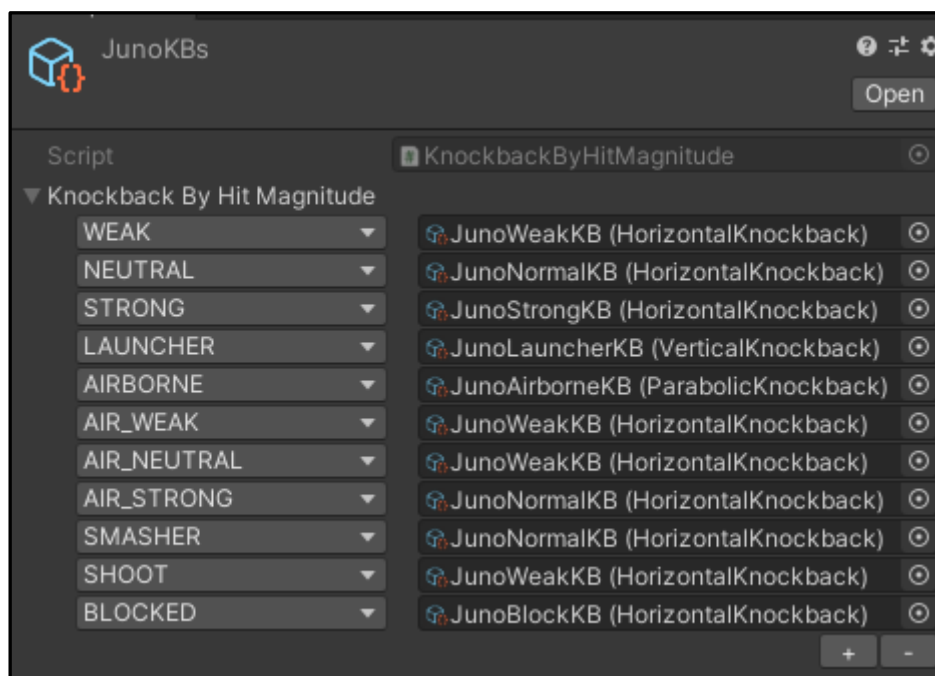


Ilustración 99- Knockbacks del personaje Juno dependiendo de la magnitud del ataque recibido

Los **knockback objects** especifican el **tiempo de incapacitación**, determinado por un valor específico o una animación, el **desplazamiento** a ejecutar y unos parámetros concretos utilizados para el combate aéreo que modifican el movimiento vertical.

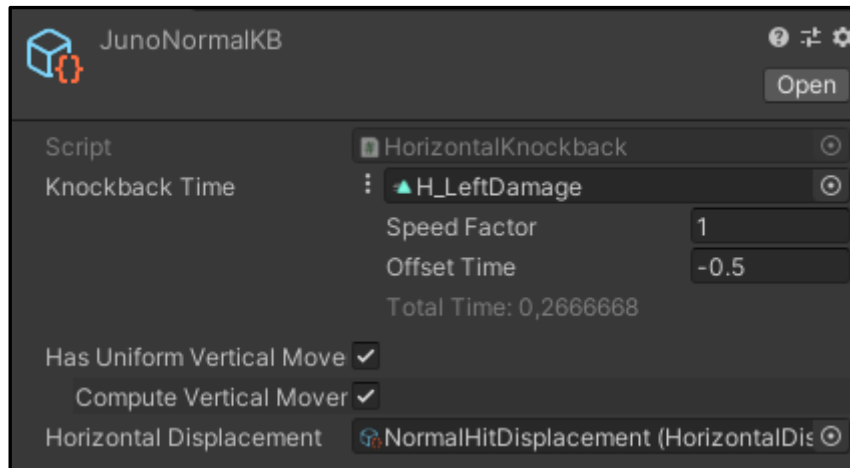


Ilustración 100- Ejemplo de un Knockback Object

La **animación de daño** a reproducir viene dada por la **magnitud** del propio **ataque**, si el personaje sale volando por los aires como en los ataques de magnitud “**AIRBORNE**” se reproduce la siguiente secuencia de animaciones con el patrón triangular visto anteriormente:

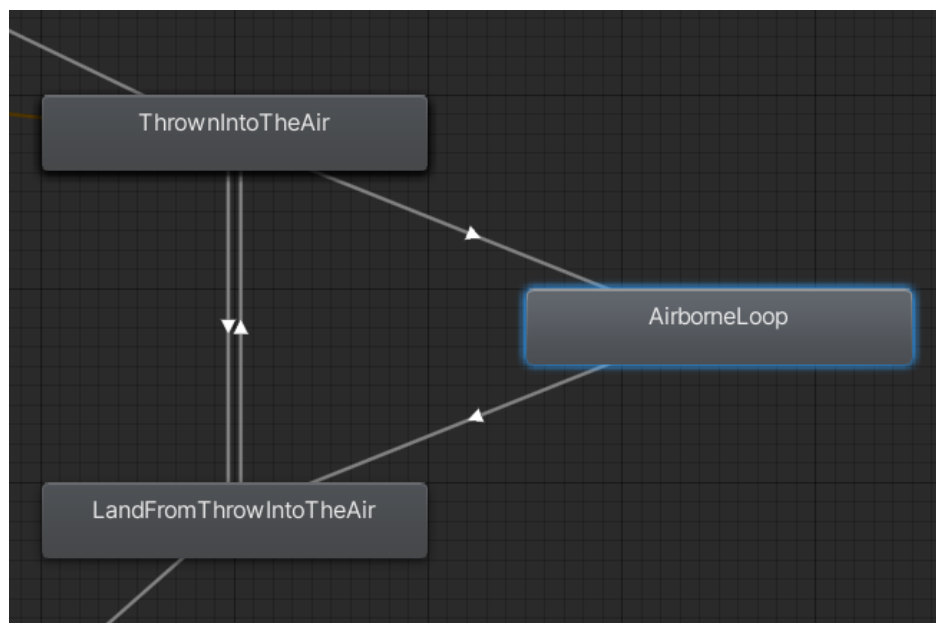


Ilustración 101 - Secuencia de animaciones cuando un personaje sale volando por los aires

Mientras tanto, las **animaciones normales** de daño se mantienen en un **blend tree** que proporciona cierta **aleatoriedad** para dar diversidad visual, además, se contempla el **lado** desde el que el **personaje** ha **recibido** el **golpe**:

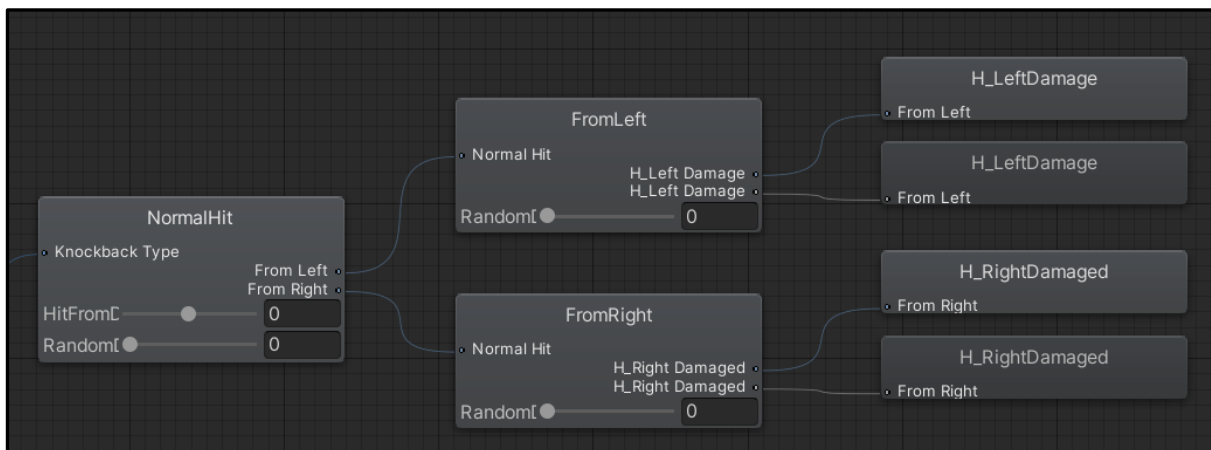


Ilustración 102- Blend Tree de las animaciones normales de daño

Por último, **no siempre que un personaje recibe daño se produce un knockback**. Como ya se expuso anteriormente, **mientras un personaje esté realizando una acción esta puede verse cancelada o no según la capacidad de interrupción del ataque que recibe**. Solo en el caso en el que el ataque corte la acción, se produce el knockback.

```

Interruptio atkInterrupt = hit.hitInfo.interruptEnemy;

bool isAttackInterrupted = info.IsAttacking && !combat.isInFocusAction && atkInterrupt == Interruption.ONLY_ATTACKS;
bool isFocusActionInterrupted = atkInterrupt == Interruption.ATTACKS_AND_FOCUS_ACTIONS;
bool isVulnerable = !info.IsAttacking && !info.IsCasting || combat.CanCancelOn("DamageCancel");

reactToAttack = (isAttackInterrupted || isFocusActionInterrupted || isVulnerable);

playKnockback = reactToAttack || atkInterrupt == Interruption.MAX_PRIORITY;
  
```

Snippet 23- Determinación de si la acción actual puede ser cancelada al recibir un ataque

RF16 Al conectar un ataque contra un personaje este puede experimentar un knockback que interrumpa su acción actual, incapacitándolo durante un periodo de tiempo predefinido. El knockback a aplicar depende de la magnitud del ataque y del personaje involucrado. ✓

RF17 Existen acciones que no pueden ser interrumpidas, durante las cuales no se les puede aplicar ningún tipo de knockback a los personajes que las están realizando. ✓

Combate aéreo

El **combate aéreo** presenta unas cuantas **peculiaridades** con respecto al combate normal, debido a que el juego tiene que ser capaz de proporcionar al jugador la capacidad de realizar **juggling** con los enemigos.

Una de las principales diferencias es que la **gravedad no afecta al movimiento**, dejando al **jugador suspendido en el aire mientras este ejecuta ataques**, logrando así un **mayor control y facilidad** a la hora de **conectar golpes**. Esta funcionalidad se contempla en el propio AttackNode en el que se le indica si el ataque se ve afectado por la velocidad vertical (Affected By Vertical Speed)

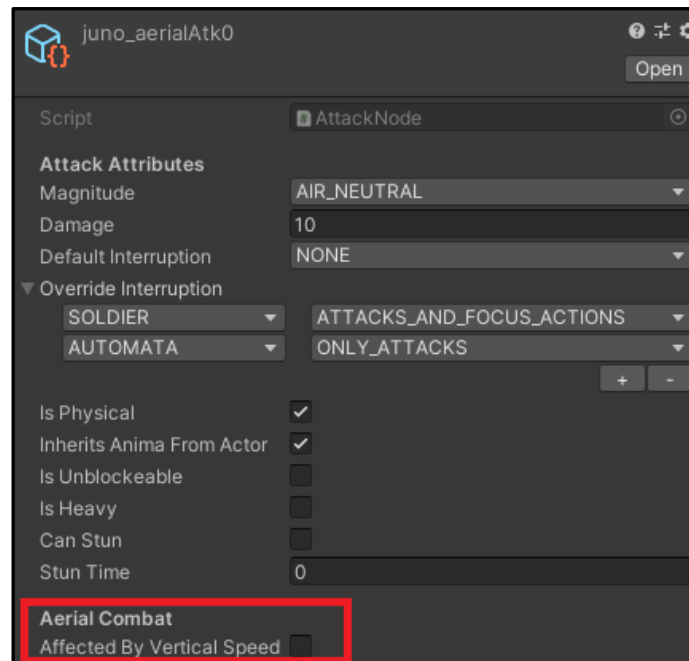


Ilustración 103 - Attack Node para el combate aéreo

Adicionalmente, los **enemigos** también ven su **movimiento vertical alterado** cuando son **golpeados** en el **aire**. Los **knockbacks objects** tienen en cuenta **cambios** en la **gravedad** y en la **velocidad** de **caída** para que el **enemigo** se mantenga **suspendido** en el **aire** durante una parte del tiempo de la incapacitación y no caigan inmediatamente después de ser golpeados, facilitando al jugador la concatenación de combos.

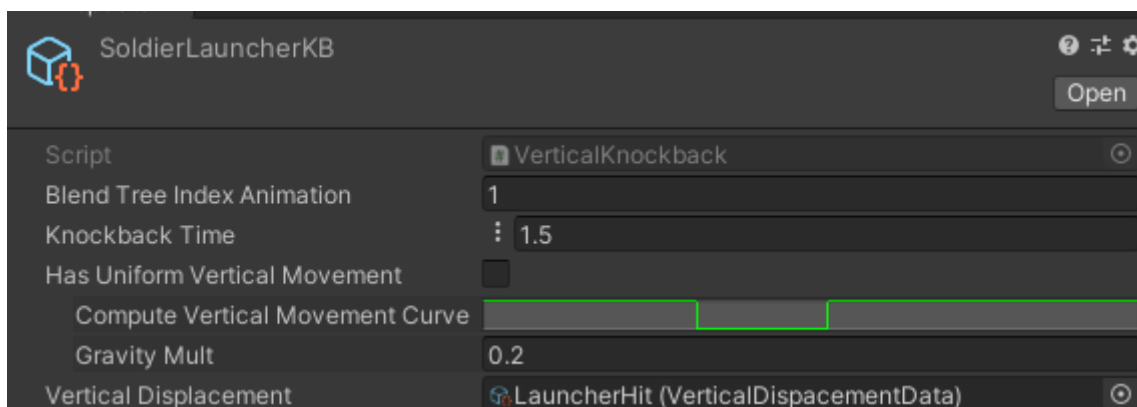


Ilustración 104 - Knockback Object para el combate aéreo

El campo "Has Uniform Vertical Movement" indica que durante el **knockback** se van a producir alteraciones en el cálculo del movimiento vertical designados por la curva de animación definida. En esta curva los valles que presenta definen los momentos en los que la

velocidad vertical se ignora, es decir, cuando los enemigos se quedan flotando en el aire. Además, también es posible modificar la gravedad a la hora de que el enemigo caiga para facilitar el *juggling* al jugador.

RF15 El personaje jugador puede realizar ataques aéreos, suspendiéndose en el aire mientras los ejecuta. ✓

Bloqueo y contraataque

Cuando el jugador ingresa el input para bloquear, el **personaje** jugable **entra** en el estado de “**BlockState**”, en él, la **hurtbox principal** se **desactiva** y se **habilita** otra (*BlockingHurtbox*) de un **mayor tamaño** que siempre **da** como **respuesta** que el **golpe** ha sido **bloqueado**, a no ser que el ataque recibido sea no bloqueable.

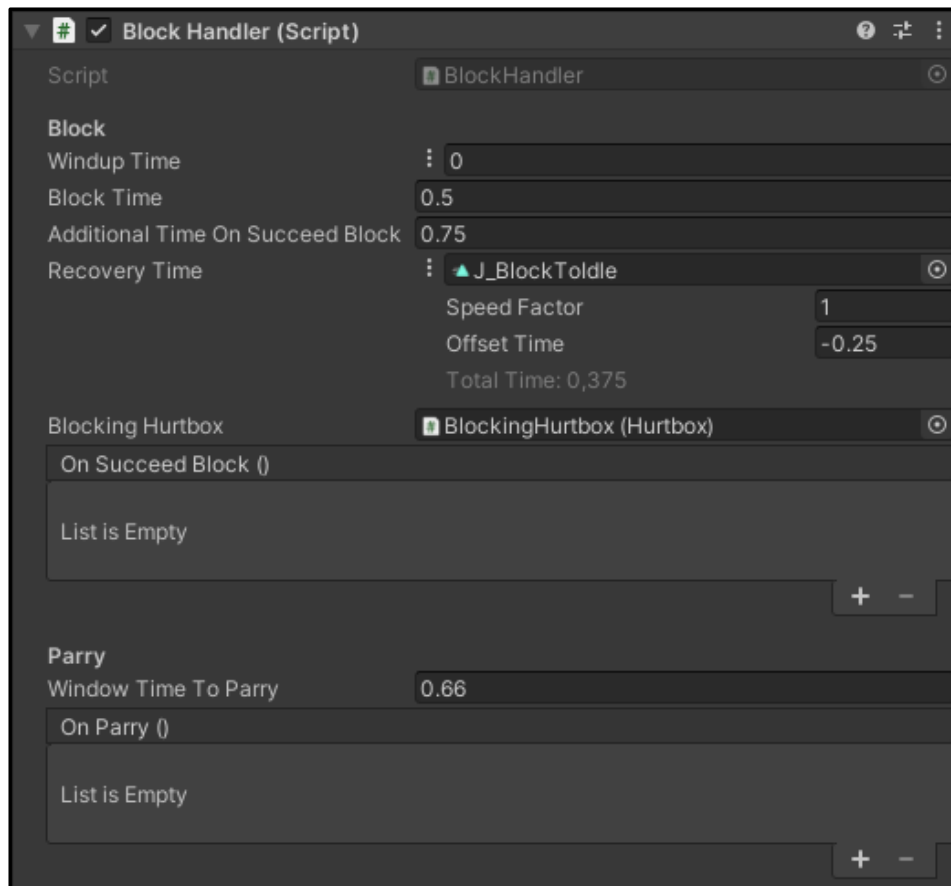


Ilustración 105 - Componente BlockHandler

Mientras el **personaje** se encuentra **en guardia** (*Block Time*) para prevenir cualquier ataque, el jugador **no** puede **realizar** otras **acciones** hasta que el tiempo del bloqueo termine. Al recibir y **parar** un **golpe** con éxito se abre una **ventana** de tiempo en el que si el jugador ingresa cualquier input de ataque el personaje realizara el **contraataque** o *parry* (*Window Time To Parry*). Esta ventana se reinicia y el periodo de bloqueo efectivo se amplía cada vez que el personaje bloquea un golpe con éxito (*Additional Time On Succeed Block*) hasta un número de veces indefinido.

En contrapartida, cuando el **personaje falla el bloqueo**, esto es, cuando el jugador no predice bien las intenciones del enemigo y bloquea en vano sin recibir ningún ataque, se le **aplica una penalización de tiempo** (*Recovery Time*) durante la cual **no puede moverse** ni realizar ninguna **acción**, dejándolo totalmente vulnerable.

La gestión de las **animaciones de bloqueo** es un tanto similar al **patrón triangular** visto anteriormente, existe una animación de *wind-up* (**BlockWindUp**) y de *recovery* (**BlockRecovery**) que actúan como **principio** y **fin**, pero esta vez la parte que se repite en **bucle** la conforman **dos animaciones**: la de estar en guardia (**BlockLoop**) y la de bloquear un golpe recibido (**AttackBlocked**). Ambas animaciones irán transicionando entre ellas dependiendo de los ataques que el jugador reciba hasta que el tiempo del bloqueo efectivo se agote.

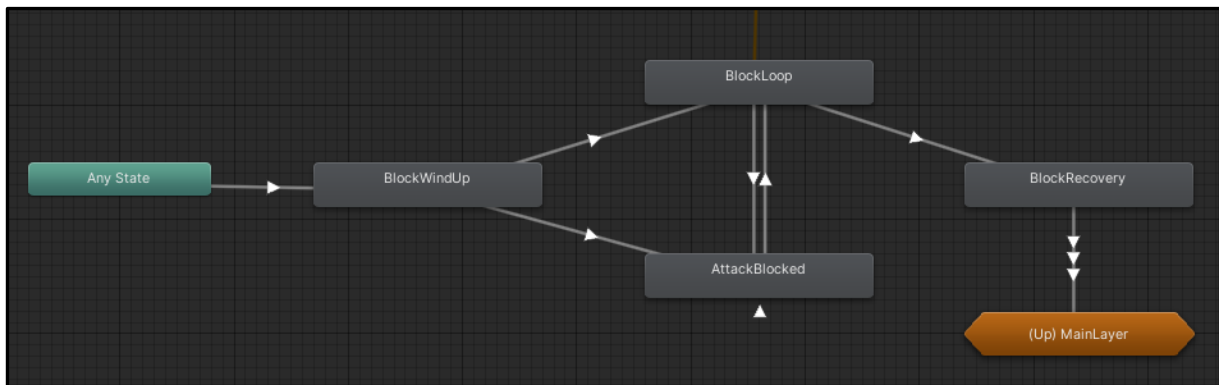


Ilustración 106- Animaciones de bloqueo

Para terminar, el **contraataque a nivel lógico** se trata simplemente como un **ataque normal** y corriente: el **personaje abandona el "BlockState"** para pasar al **"AttackState"**, se asigna el **AttackNode** correspondiente y se reproduce la animación pertinente en la que se activarán los eventos de animación que se han estado exponiendo a lo largo del trabajo.

RF13 Al realizar un bloqueo con éxito se evita todo el daño recibido. ✓

RF14 Al bloquear, el personaje jugador dispone de una ventana de tiempo para realizar un contraataque. ✓

Modo Concentración

Con el sistema de combate ya construido, incluir las **técnicas**, los **actos** y la consumición de **objetos** se convierte en una **tarea** un tanto **trivial**, ya que la **gestión** de sus **animaciones** es **tratada** como cualquier **ataque** visto hasta ahora. Cada **acción** en el **modo concentración** de cada uno de los personajes tiene **asociado** un **identificador** que sirve para **reproducir** la **animación correspondiente** en el Animator.

```
public enum FocusActionID
{
    JUNO_COMET_SWORD = 0,
    JUNO_BRUTAL_SLASH = 1,
    JUNO_PERFECT_GUARD = 2,
    JUNO_IGNITION = 3,
    JUNO_RETURN = 4,
    JUNO_HEALING = 5,

    SOLDIER_COURAGE_LUNGE = 6,

    ARMORED_SHIELD_LUNGE = 7,
    ARMORED_STUNNING_HIT = 8,

    AUTOMATA_OVERCLOCK = 9,
    AUTOMATA_KAMIKAZE = 10,

    RAJ_RETURN = 11,
    RAJ_SHOCKWAVE = 12,
    RAJ_PERFECT_GUARD = 13,

    NYX_SHADY_STEP = 14,
    NYX_RONDO = 15,
    NYX_PERFECT_GUARD = 16,
    NYX_RETURN = 17,
    NYX_BURIAL = 18,
    NYX_SLAVE_SWORD = 19,
    NYX_DIABOLIC_BULLET = 20,
    NYX_ARMISTICE = 21,
    CONSUMABLE_POTION = 22,
    CONSUMABLE_BOTTLED_DETERMINATION = 22,
}
```

Ilustración 107- Identificador de las acciones del modo concentración

Cada **animación** que conforman las **técnicas**, **actos** y **objetos** se mantiene en una **submáquina** dentro del AnimatorController. En ella, las animaciones, como en la submáquina de ataque, parten del “AnyState” en el que cada **transición** se activa con el parámetro **trigger “FocusAction”**, accionado siempre que se realiza una acción en el modo concentración, seguido del **identificador** correspondiente.

RF22 Al seleccionar una acción en el modo concentración se debe reproducir inmediatamente la animación correcta. ✓

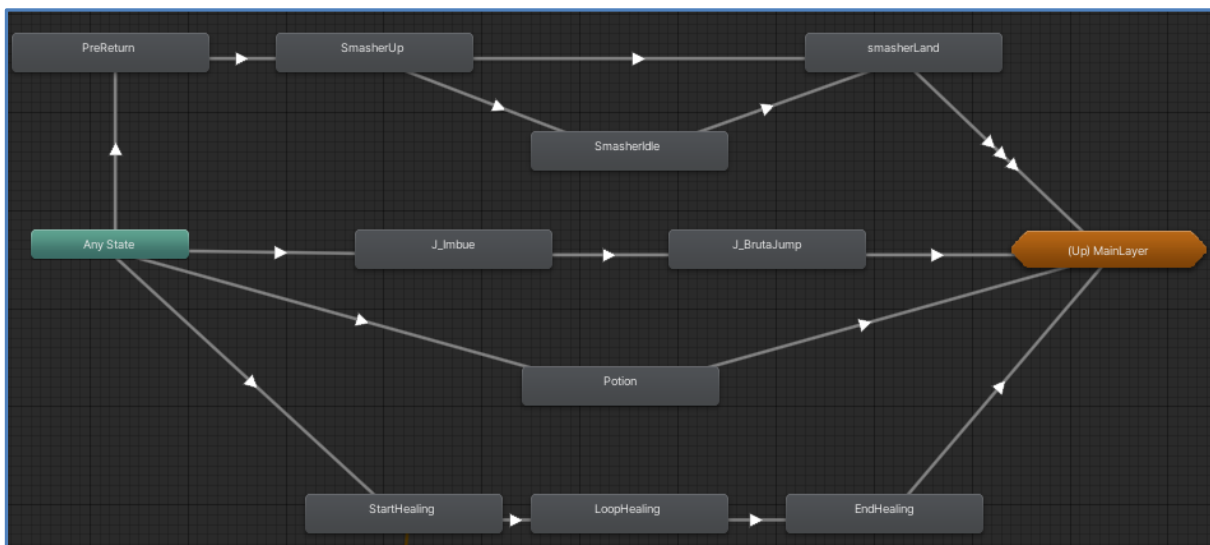


Ilustración 108- Submáquina de las acciones del modo concentración del personaje Juno

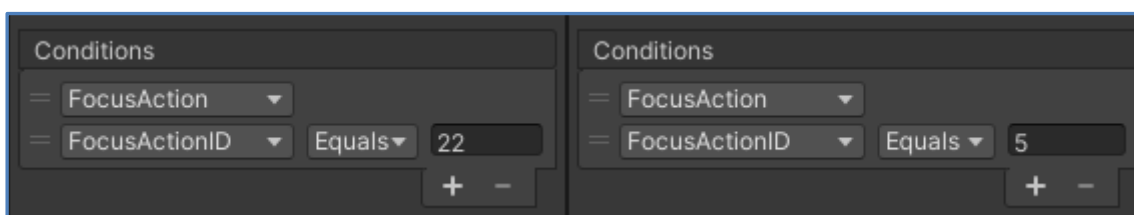


Ilustración 109- Condiciones de las transiciones para reproducir las animaciones de consumir poción (izq) y curación (dcha)

A diferencia de las **técnicas** que se gestionan en el “**AttackState**”, la ejecución de **actos** y la consumición de **objetos** se manejan desde el “**CastState**” y no tienen un **AttackNode** asignado ya que no son ataques como tal. Estas **acciones**, que precisan de un **tiempo de espera** para que tomen efecto, **almacenan** su **funcionalidad** real en un **objeto** de tipo **Usable** al que hacer referencia más tarde durante la animación, por ejemplo, la curación y la poción mantienen un **Usable** de tipo **UsableHeal** que hará que el personaje restablezca sus puntos de vidas cuando este se accione.

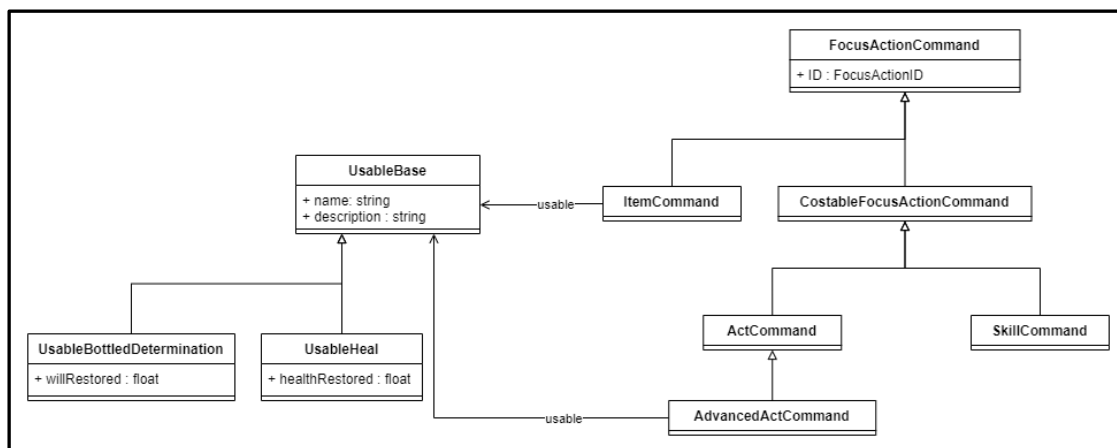


Ilustración 110- UML FocusActions y Usables

Todas las **acciones** del **modo concentración** se definen en **ScriptableObjects** (SkillCommand, ActCommand, ItemCommand) en los que se mantienen toda la **información necesaria** para representarlas en la **interfaz gráfica** de usuario y para su correcto **funcionamiento** como pueden ser los **costos de voluntad** o **vida**, las **condiciones** necesarias para ejecutarlas o, lo que más atañe a este trabajo, su **identificador** y los **Usables** que guardan su funcionalidad adicional.

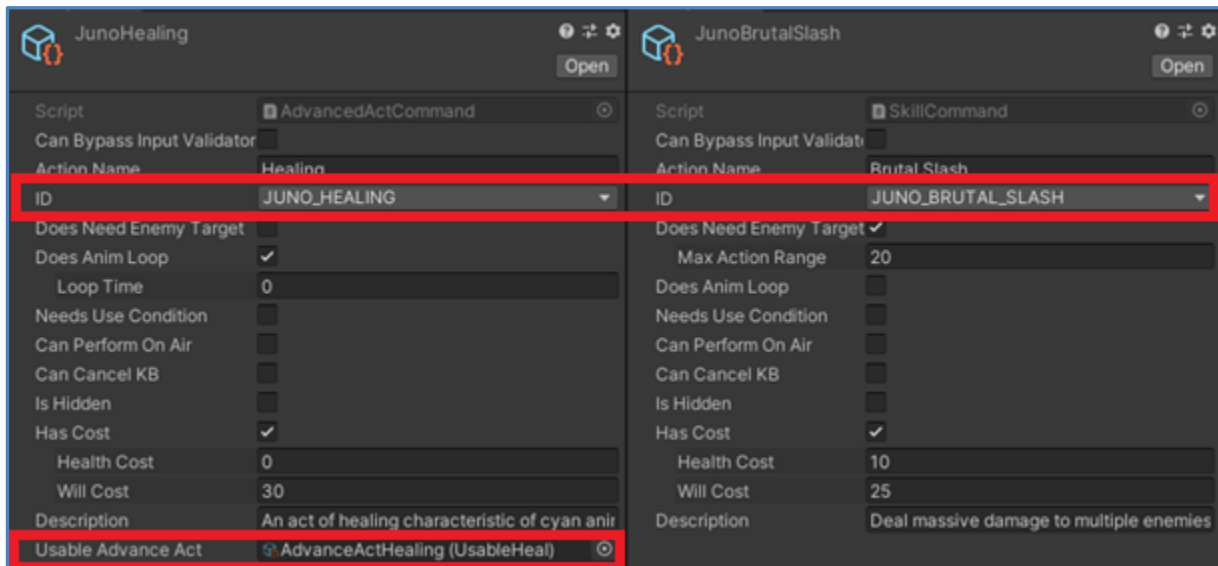


Ilustración 111- ScriptableObjects de la Curación (izq.) y del Tajo Brutal (dcha). En rojo sus identificador y el Usable que utiliza la cura

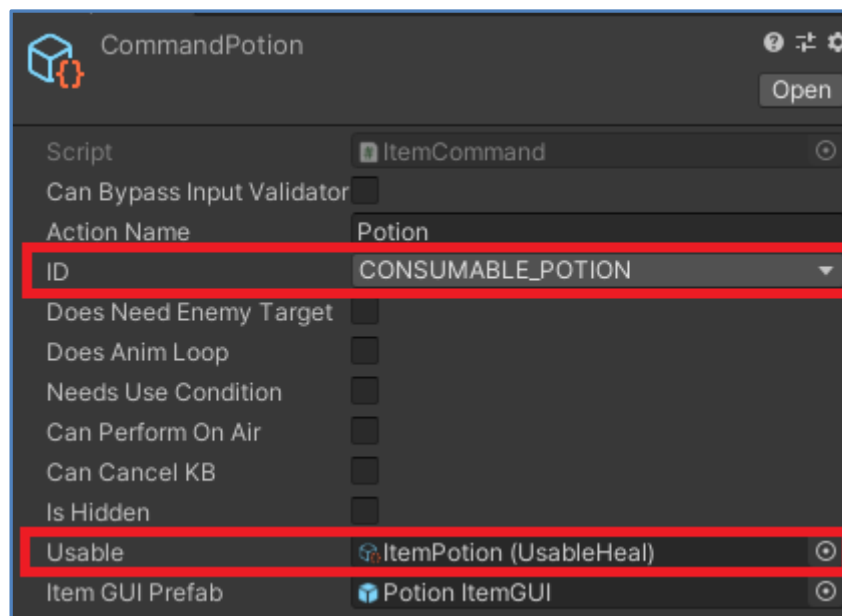


Ilustración 112- ScriptableObject de la poción. En rojo su identificador y el Usable referenciado

Los **Usable** se **guardan** temporalmente en el componente **CombatEntity** cuando una de estas acciones se empieza a ejecutar. Para **marcar** el **momento** exacto en el que

desencadenar el **efecto** del acto o del objeto se utilizan los **eventos** de animación “**UseAct**” y “**UseConsumable**”, respectivamente. De esta forma, como la funcionalidad queda encapsulada en estos eventos, se puede reutilizar la misma animación como en el caso de consumición de objetos ya que el efecto a aplicar dependerá del objeto Usable referenciado con anterioridad.

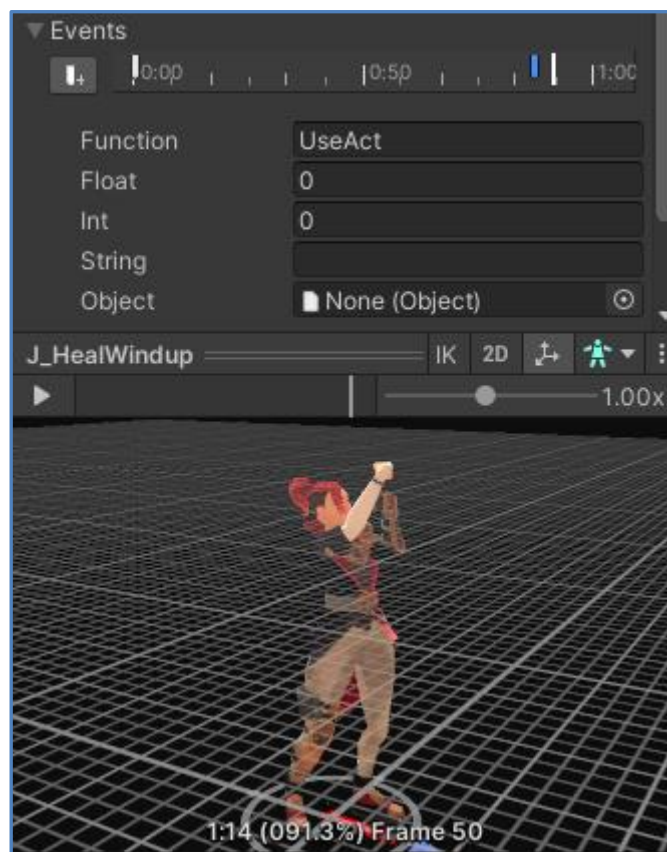


Ilustración 113- Evento de animación UseAct

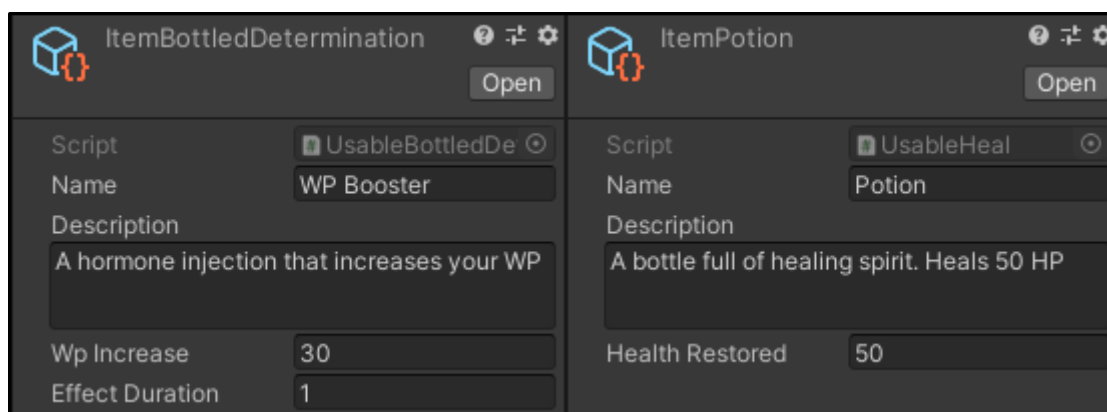


Ilustración 114- Usables de los dos objetos disponibles en el juego



RF23 Al ejecutar un acto o consumir un objeto el jugador debe esperar un periodo de tiempo dictado por la animación hasta que se aplique el efecto en sí de la acción. ✓

RF24 La acción de usar objetos debe hacer uso de la misma animación, aunque los objetos sean diferentes. ✓

Fortalezas y debilidades

Cada **personaje** tiene un objeto **CharacterData** que indica su **vida** y **voluntad** máximas, su conjunto de **knockbacks** y también su **anima** y los **multiplicadores** de daño del **sistema de fortalezas y debilidades**. A su vez, el anima, representada por su AnimaData, define contra qué otras animas es fuerte o débil.

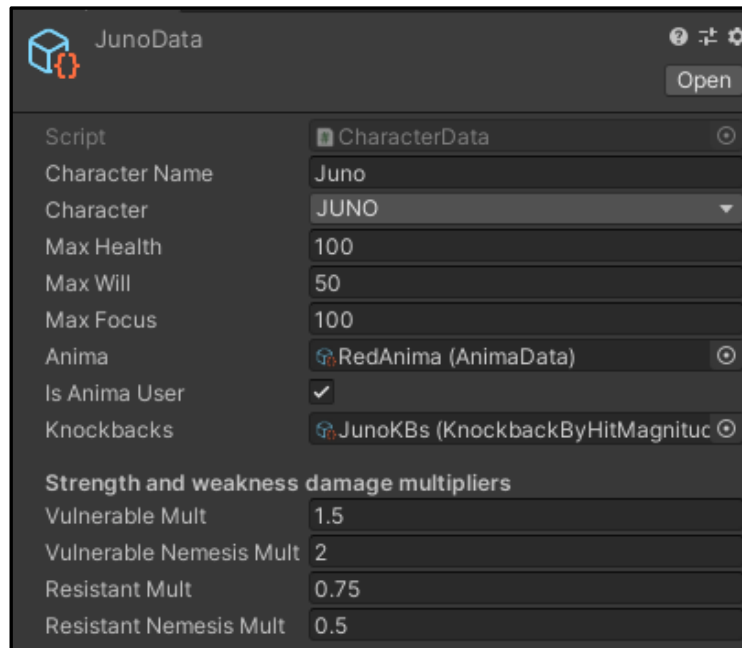


Ilustración 115- CharacterData del personaje Juno

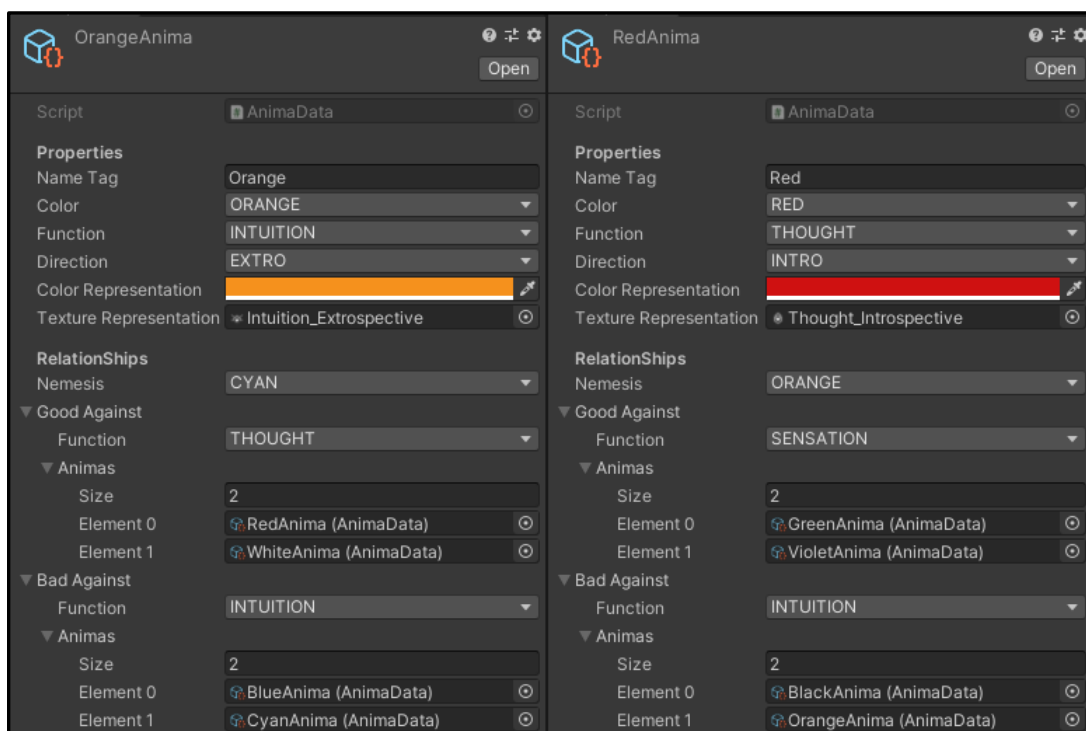


Ilustración 116- AnimaData pertenecientes a la anima naranja (izq.) y anima roja (dcha.)

Al recibir cualquier **ataque elemental** de anima, simplemente se **comprueba** la **relación** que mantiene el **anima** del **atacante** con el **anima** del **atacado** para saber qué multiplicador es necesario aplicar en el cálculo del daño.

```
public void ApplyDamage(ConnectedHitInfo hit)
{
    if (canTakeDamage)
    {
        float damageMultiplier = GetAnimaBasedDamageMultiplier(hit.hitInfo);
        info.CurrentHealth -= hit.hitInfo.damage * damageMultiplier;
    }
}

private float GetAnimaBasedDamageMultiplier(HitInfo hitInfo)
{
    if (hitInfo.isPhysical)
        return 1.0f;

    AnimaData enemyAnima = hitInfo.anima;
    bool resistant = info.Anima.function == enemyAnima.badAgainst.function;
    bool vulnerable = info.Anima.function == enemyAnima.goodAgainst.function;
    bool isResistantNemesis = info.Anima.color == enemyAnima.nemesis;
    bool isVulnerableNemesis = info.Anima.nemesis == enemyAnima.color;

    if (isVulnerableNemesis)
        return info.VulnerableNemesisMult;

    if (isResistantNemesis)
        return info.ResistantNemesisMult;

    if (vulnerable)
        return info.VulnerableMult;

    if (resistant)
        return info.ResistantMult;

    return 1.0f;
}
```

Snippet 24- Cálculo del daño infligido según las fortalezas y debilidades del anima

De forma análoga, se confirma si el **personaje** queda **aturdido** y durante cuánto tiempo. El aturdimiento se trata del mismo modo que cualquier **knockback**, pasando por el “*DamagedState*”, tan solo **cambia** la **animación** y el tiempo del *knockback object* se sobrescribe por el **tiempo** de la **incapacitación** definida en el AttackNode (*Stun Time*).

```
float incapacityTime = currentKnockback.knockbackTime;

bool isAnimaVulnerable = !hitInfo.isPhysical && info.Anima.function ==
hitInfo.anima.goodAgainst.function;

if ((hitInfo.canStun || isAnimaVulnerable) && hitInfo.stunTime > 0.0f)
{
    info.IsStunned = true;
    incapacityTime = hitInfo.stunTime;
    info animator.SetTrigger(stun);
    return;
}
```

Snippet 25- Aturdimiento del personaje

RF12 Cuando un ataque conecta se inflige la cantidad de daño correspondiente, atendiendo al sistema de fortalezas y debilidades. ✓

RF18 Al recibir un ataque elemental, si el ánima del personaje es débil contra el ánima del atacante este se verá aturdido durante un periodo de tiempo. ✓

Gamefeel

En cuanto a las **técnicas** de **gamefeel** expuestas con anterioridad, su implementación corre a cargo de los componentes **CombatEffects** y **CombatEffectsApplier**.

Por un lado, **CombatEffects** se trata de un *singleton* utilizado para **generar efectos globales**, este script mantiene la funcionalidad necesaria para poder generar **game slows/ game freezes, camera shakes y vibraciones de mando** desde cualquier parte del código y en cualquier situación, ya sea durante el combate o en cinemáticas.

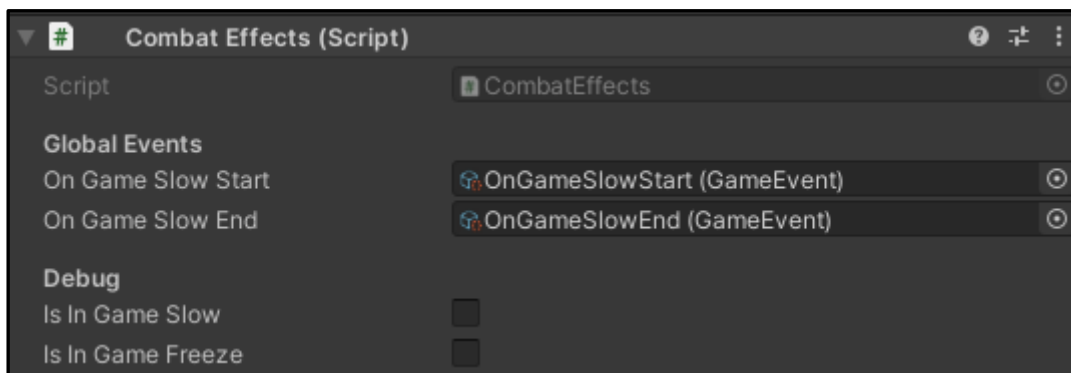


Ilustración 117- Componente **CombatEffects**

Por otro lado, **CombatEffectsApplier** es un **componente** dispuesto **en cada** uno de los **personajes** involucrados en el sistema de combate. Este script, aunque hace uso de la

funcionalidad implementada en `CombatEffects`, también se encarga de aportar otros efectos que son solo posible a nivel de personajes, como el **hit pause** o, por la parte de *VFX*, resaltar al personaje de un color específico cuando ha recibido daño.

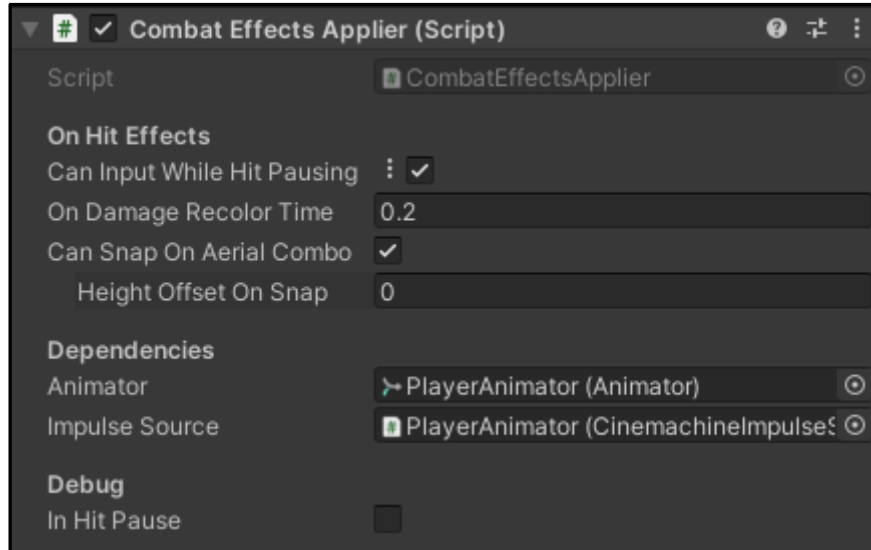


Ilustración 118- Componente `CombatEffectsApplier`

Además, controla algunos parámetros relacionados para **mejorar** el **juggling** del combate aéreo como el “*Snap On Aerial Combo*”, que permite al jugador no dejar caer al enemigo ajustándolo automáticamente a la altura de su personaje.

Como viene siendo habitual en este trabajo, toda esta funcionalidad viene expuesta en forma de `Animation Events` para poder hacer uso de todas estas técnicas en el sistema de combate de una forma rápida y sencilla.

Camera Shake

Cada **camera shake** se representa por medio de un `ScriptableObject` en el cual se define el **comportamiento** del temblor o **sacudida** que debe experimentar la **cámara**. El movimiento se obtiene a partir de una velocidad base, una capa de ruido y una ASDR que modifica la magnitud durante la ejecución del efecto.

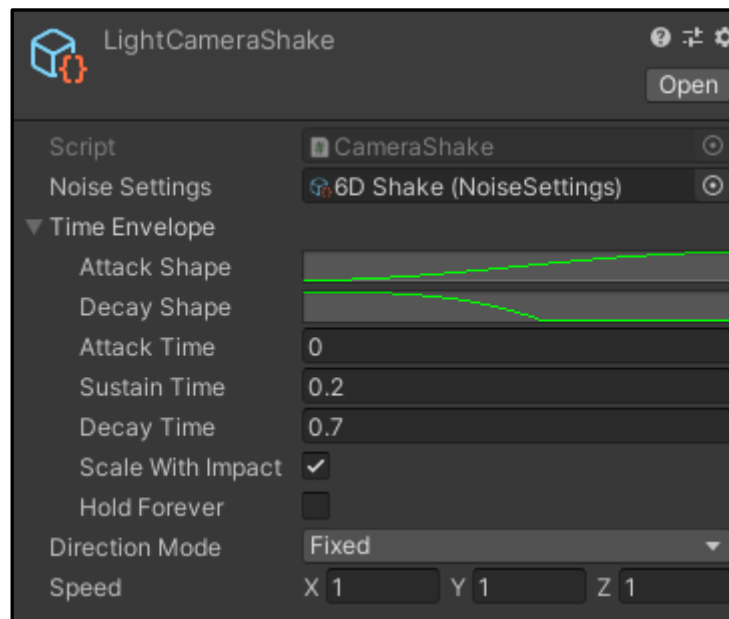


Ilustración 119- Ejemplo de CameraShake

Debido a que en este proyecto se usa el paquete de **Cinemachine** [78] para la **gestión de cámaras virtuales**, es necesario que el personaje tenga a su disposición un **componente** de tipo **CinemachineImpulseSource** [79] para **aplicar** el **camera shake**, el cual se define como un impulso dentro de este paquete.

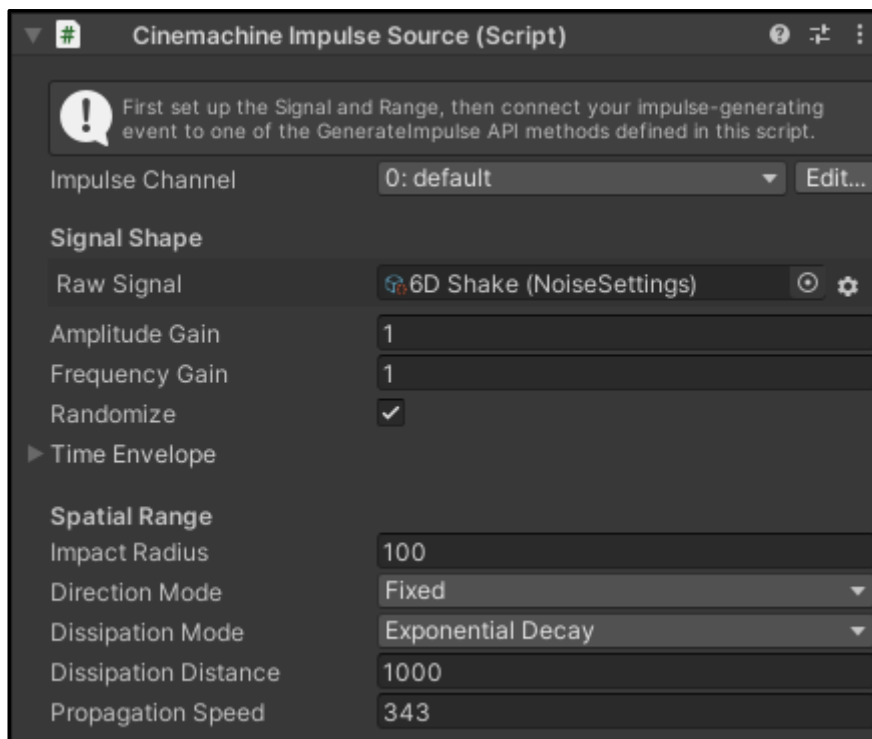


Ilustración 120- Componente CinemachineImpulseSource

Así pues, la forma de aplicar un *camera shake* se basa en configurar las propiedades de este nuevo componente para que coincidan con las definidas en el `ScriptableObject` y generar el impulso construido.

```
public void ApplyCameraShake(CinemachineImpulseSource impulseSource, CameraShake shake)
{
    impulseSource.m_ImpulseDefinition.m_RawSignal = shake.noiseSettings;
    impulseSource.m_ImpulseDefinition.m_TimeEnvelope = shake.timeEnvelope;
    impulseSource.m_ImpulseDefinition.m_TimeEnvelope = shake.timeEnvelope;
    impulseSource.GenerateImpulse(shake.speed);
}
```

Snippet 26- Método `ApplyCameraShake`

Vibración de mando

Al igual que el *camera shake*, la vibración de mando, o **gamepad rumble**, se define en **ScriptableObjects**. En el proyecto existen tres tipos de vibración diferentes que indican cómo han de comportarse los motores dentro del mando para generar el efecto deseado.

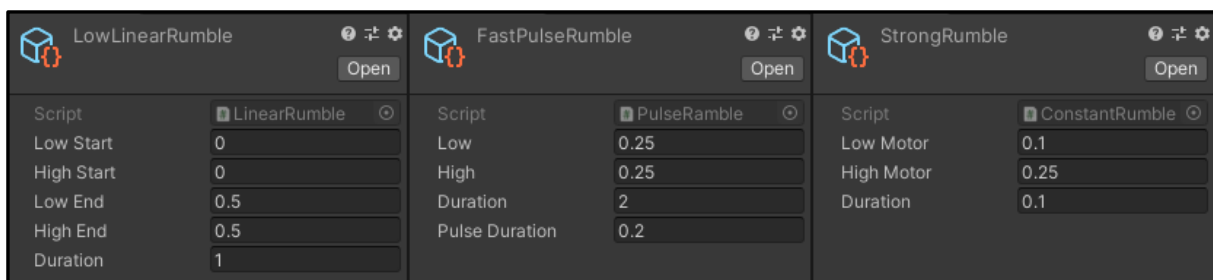


Ilustración 121- Tipos de gamepad rumble, de izq. a dcha.: Linear Rumble, Pulse Rumble y Constant Rumble

- **Linear Rumble:** genera una vibración que aumenta o disminuye de forma lineal a lo largo del tiempo.
- **Pulse Rumble:** produce un conjunto de pulsos de una cierta intensidad y duración mientras dure el efecto.
- **Constant Rumble:** genera una vibración constante durante el tiempo especificado

La implementación de cada *gamepad rumble* simplemente consiste en establecer las velocidades de los motores de alta y baja frecuencia a lo largo del tiempo por medio del método `Gamepad.SetMotorSpeeds()` [80] proporcionado por Unity.

Hit Pause

Junto con el *camera shake* y el *gamepad rumble*, el hit pause forma la triada por excelencia a la hora de proporcionar una sensación satisfactoria cuando el jugador consigue conectar un ataque detrás de otro durante el combate.

El único **parámetro** que define el **hit pause** es el **tiempo** en el que los **personajes** involucrados **permanecen inmóviles** cuando un ataque impacta. Estos tiempos se marcan en la **sección de gamefeel** del mismo **AttackNode** que define el ataque, junto con el *camera shake* y la vibración de mando que deben efectuarse cuando el jugador recibe o da un golpe.

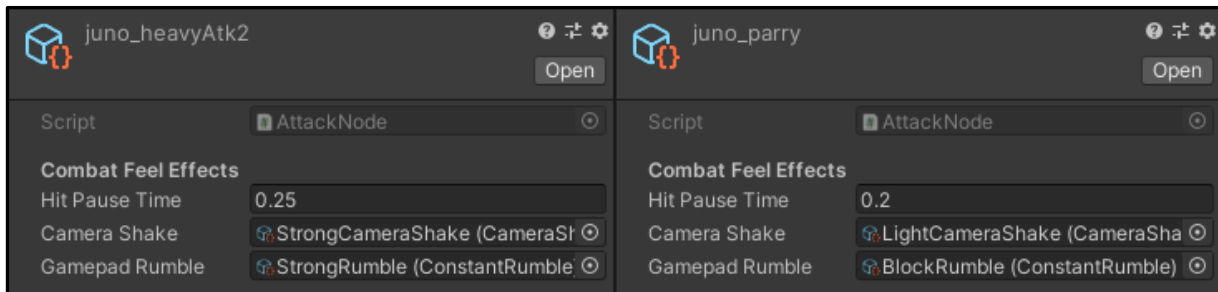


Ilustración 122- Hit pause, camera shake y gamepad rumble en diferentes AttackNodes

De esta manera, la **sensación** que pueda **transmitir** cada **ataque** puede ser **personalizada** de una forma **fácil** y **sencilla**. Si el ataque inflige bastante daño se le puede asignar un hit pause moderado, con una camera shake y una vibración acorde a la fuerza, mientras que, si se trata de un impacto ligero, se puede suprimir el hitpause y el camera shake y solo aplicar una leve vibración de mando.

La implementación del hit pause como tal es bastante sencilla gracias al uso de las *coroutines* proporcionadas por Unity. Tan solo es necesario pausar el cálculo del movimiento, los VFX involucrados y el Animator [81], para luego reanudarlos después de que haya transcurrido el tiempo indicado por el hit pause.

RF20 Al conectar un golpe se debe poder aplicar un efecto de hit pause, la duración de este debe ser personalizable para cada ataque. ✓

Cámaras dinámicas

El uso de **diferentes cámaras** durante el combate sirve para **enriquecer** la **experiencia** al **enfatar** la **acción** o al proporcionar **nuevos puntos** de **vista** que estimulan el **dinamismo**. En las propias animaciones de ataque es posible, por medio de eventos de animación, cambiar de una cámara a otro con tan solo indicar su nombre. Estas cámaras dinámicas se almacenan y gestionan en el componente **VirtualCamerasHolder** utilizando el paquete de Cinemachine.

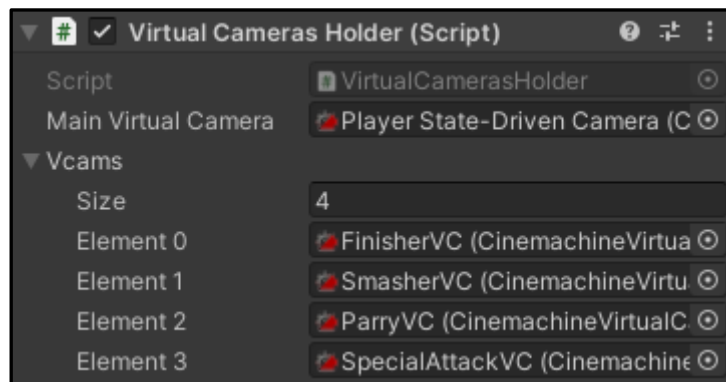


Ilustración 123- Componente VirtualCamerasHolder

Una cuestión importante a la hora de tratar con cámaras es tener en cuenta su **prioridad**, ya que no existe solo una cámara virtual para el combate y más aún si se contemplan las usadas en las secciones cinemáticas del juego. Por ello, se mantiene un enumerador que ordena por prioridad las diferentes cámaras en caso de que hubiera conflicto al encontrarse activas más de una al mismo tiempo.

```
public enum CameraPriority
{
    UNUSED = 0,
    LOCOMOTION = 100,
    DYNAMIC_1 = 200,
    DYNAMIC_2 = 600,
    COMBAT = 700,
    VIEW = 900,
    LOCK_ON = 1000,
    COMBAT_TEMP = 1010,
    CINEMATIC = 1100,
}
```

Ilustración 124 - Prioridades de las cámara virtuales

Para **realizar un cambio de cámara** tan solo se debe **asignar la prioridad adecuada**, y, una vez esta sea dispensable, establecer el valor de prioridad por defecto de UNUSED.

Game Slow y Game Freeze

El **game slow** se trata de la funcionalidad más fácil de implementar debido a que solo es necesario modificar la variable de **Time.timeScale** proporcionada por Unity para lograr el efecto deseado [82]. El **game slow** se encuentra parametrizado por una **variable** de reproducción que **define la velocidad** a la que debe ir el **juego**. De igual modo, el **game freeze** es un **caso concreto** de **game slow** en el que el factor de **velocidad** es **cero**. Como ejemplo claro, el modo concentración aplica un **game slow** cada vez que el jugador entra en él.

Aunque sea un efecto que pueda ser considerado global, existen momentos en el que es necesario que no afecte al personaje jugador en particular; mientras todo se para o se reproduce a cámara lenta, el avatar sigue a su velocidad normal. Para lograr esto, se recurre a las variables **Time.unscaledDeltaTime** [83] en el CharacterMovement y al **UnscaledTime** en el campo Update Mode del Animator [84]. De esta manera, el personaje ignora el



valor que se haya indicado en el *time scale* haciendo que su movimiento y animaciones permanezcan con el ritmo inalterado.

RF21 Se debe poder realizar en cualquier momento a lo largo del juego cada uno de los siguientes efectos de gamefeel: game slow, gamepad rumble y camera shake ✓

Sistema de fijado

El sistema de fijado consta de tres elementos bien diferenciados: la **targeting zone**, los **targetable entities** y el **target system**.

Targeting Zone

Se trata simplemente de un **collider** situado **alrededor** del **personaje jugador** con el único fin de **añadir** a los **enemigos** con lo que **colisione** a la **lista** de **objetivos** del **sistema** de fijado. Al mismo tiempo, actúa como el rango máximo del sistema, si algún objetivo sale de este **collider** se elimina inmediatamente de la lista.

Todos los enemigos situados dentro de la *targeting zone* se mantienen en un ScriptableObject llamado PlayerTargetHolder.

Targetable Entity

El **sistema** de fijado **solo contempla** a **personajes** que posean un **componente** de tipo **TargetableEntity**, en vez de almacenar el **GameObject** completo, en el PlayerTargetHolder solo se hace referencia a este script.

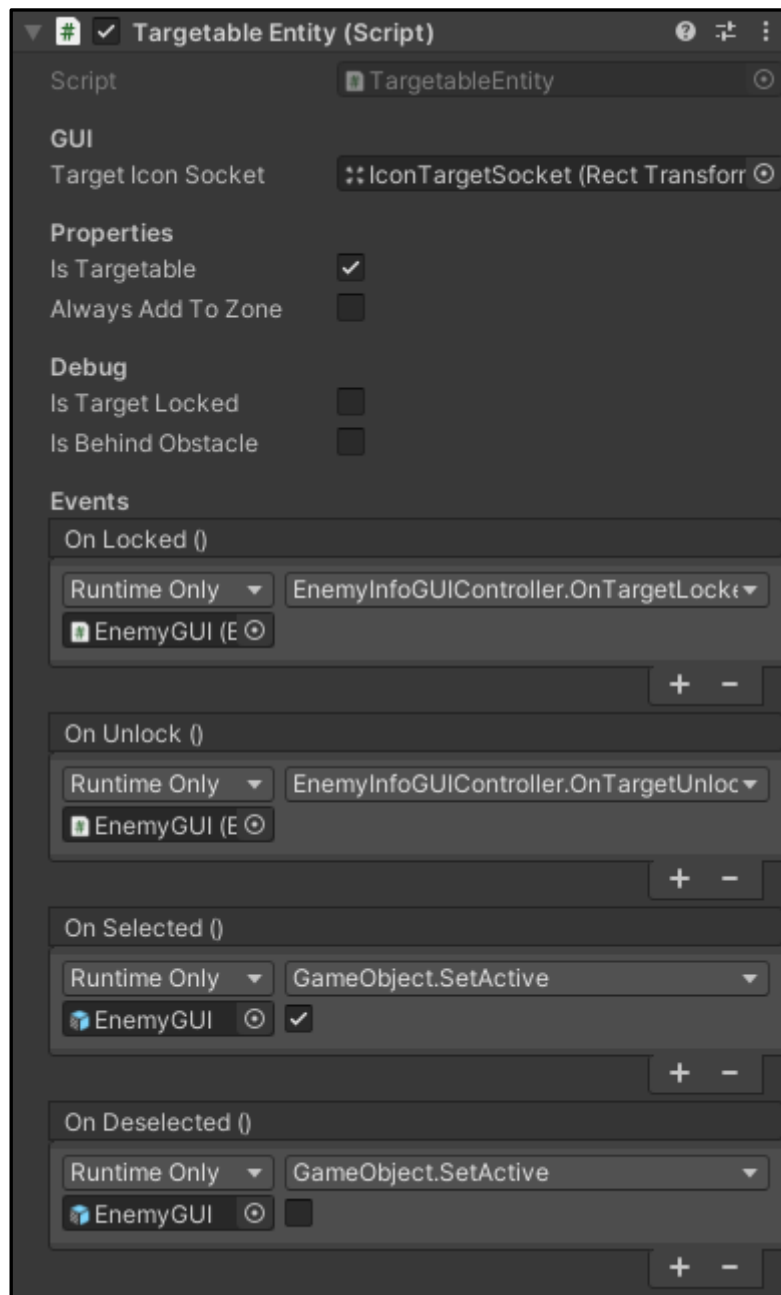


Ilustración 125- Componente TargetableEntity

En este **componente** se **gestiona** la **elegibilidad** del **enemigo** como objetivo principal (*isTargetable*) así como las **respuestas** a cuando un enemigo es **seleccionado** de **forma automática** o fijado de manera **manual** gracias a los eventos expuestos.

Target System

El **sistema de fijado** representa una parte muy importante del combate debido a que **designa** el **enemigo** al que los **ataques** irán **dirigidos**, **facilitando** que los **golpes conecten** y **mejorando** la **experiencia** en su conjunto. Cuando el objetivo es fijado por el jugador de forma manual, no existe ninguna interferencia a la hora de cumplir la intención del jugador, el problema aparece cuando el **sistema de fijado** se encuentra en **modo automático**, ya que

este debe ser capaz de **ofrecer** al **jugador** el **mejor candidato** que se **adapte** a sus **intenciones** de golpeo.

El sistema de fijado dispone de dos comportamientos bien marcados dependiendo del modo en el que esté operando:

- **Automático:** elige el mejor candidato (TargetableEntity) según cercanía y orientación con respecto al personaje jugador.
- **Manual:** ordena los enemigos según la orientación con respecto al personaje jugador.

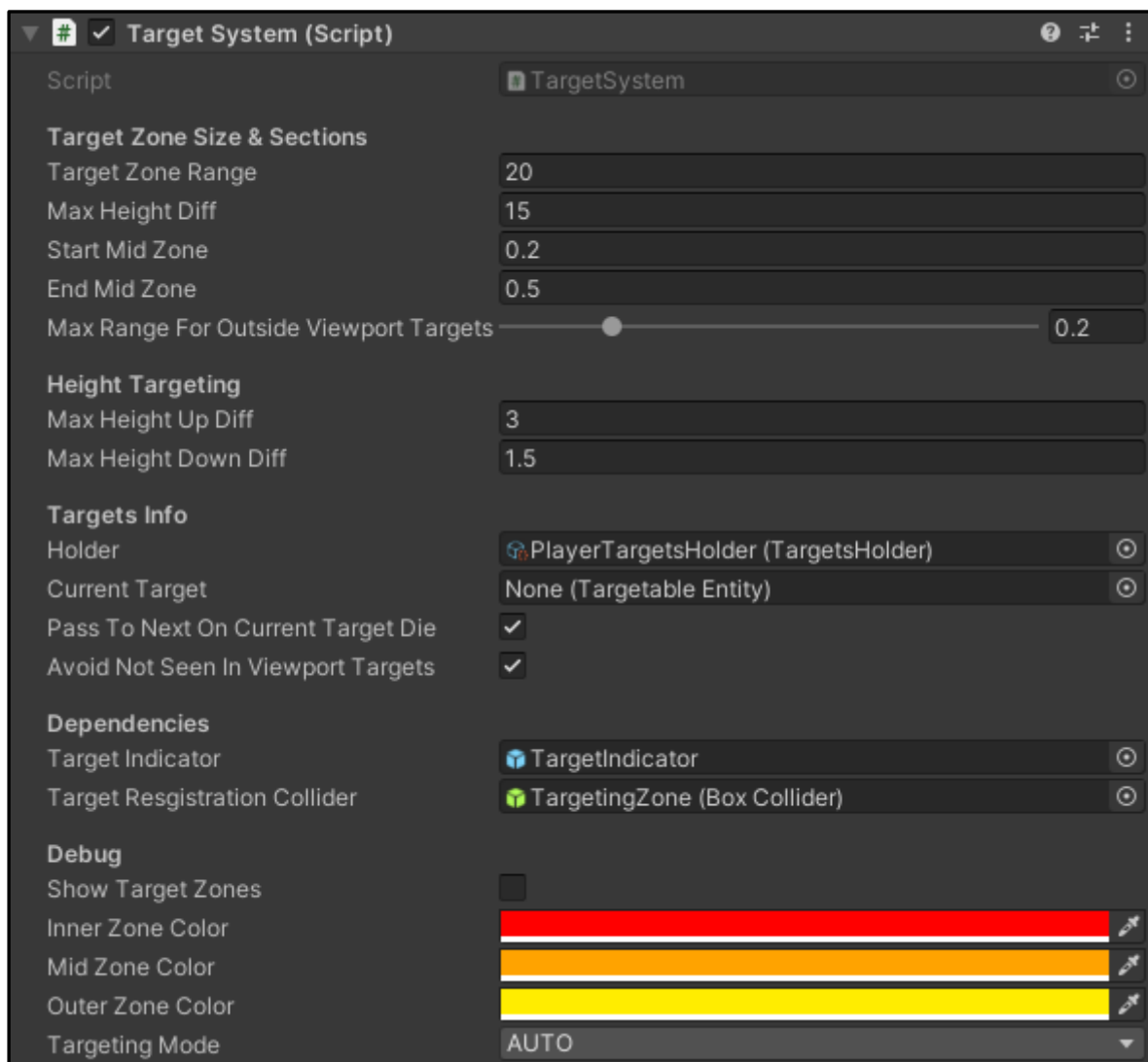


Ilustración 126 - Componente TargetSystem

La **implementación** del **modo automático** se basa en **definir zonas** que **representan** diferentes **rangos** de **distancia**, las **zonas interiores** presentan una **mayor prioridad** frente a las **zonas** más **externas**, y así mismo, **dentro de ellas**, se **prioriza** al **objetivo** que más **alineado** esté con el vector forward del **personaje jugador**. Debido a que no es posible

definir que factor es más importante en todos los casos, si cercanía u orientación, se opta por esta **solución híbrida**. Adicionalmente, en modo automático, el **sistema ignora los objetivos** que estén **fuera de pantalla** o **detrás** de un **objeto** por muy cerca o alineados que estén con el jugador.

En “Arcanima” se delimitan tres zonas que pueden ser modificadas con los parámetros presentados en la sección de “*Target Zone Size & Sections*”, adicionalmente se pone a disposición un modo debug en el que estas zonas se dibujan en tiempo real con diferentes colores.



Ilustración 127 - Modo debug del sistema de fijado

En contraposición, en el **modo manual** solo hay que preocuparse por la **ordenación** de los **objetivos** a la hora de que el jugador desee **cambiar de objetivo**, ya que el objetivo actual ya ha venido dado desde el modo automático.

Para **gestionar** el **cambio de objetivo** se mantiene una **lista circular doblemente enlazada ordenada** según el **ángulo** existente **entre** el **vector forward** del jugador y el **vector dirección** hacia el **enemigo**. El **rango de grados** se **convierte** de **0-180**, utilizado por defecto, a **-180-180**, donde el **signo** indica la **lateralidad** con respecto al personaje, y así poder ordenar correctamente todos los objetivos de menor a mayor ángulo (de izquierda a derecha). De esta manera, cuando el jugador cambia de enemigo, se establece como objetivo fijado el anterior o el siguiente que aparezca en la lista según el input introducido.

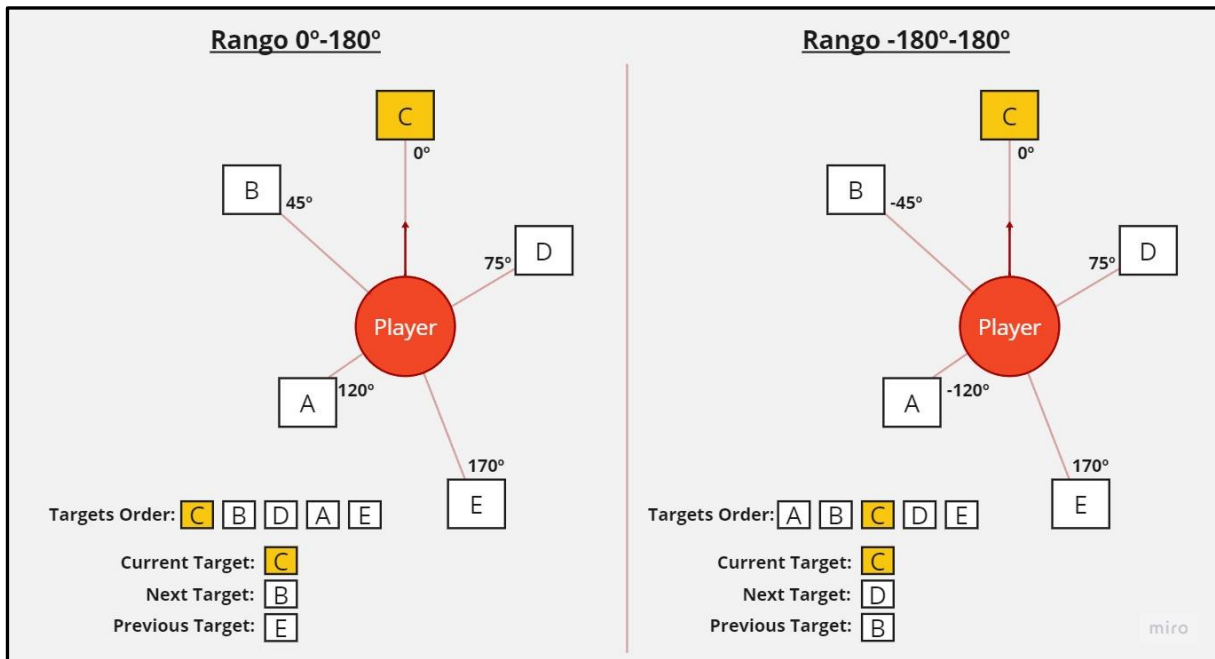


Ilustración 128- Objetivos ordenados de menor a mayor ángulo con respecto el jugador según los diferentes rangos

A diferencia del modo automático, los enemigos situados fuera de pantalla sí son incluidos en el proceso de ordenación.

RF25 El modo de fijado automático debe seleccionar al objetivo más adecuado en base a la distancia y orientación con respecto al personaje avatar. ✓

RF26 El jugador debe poder cambiar de objetivo durante el modo manual, seleccionando al enemigo previo o posterior al objetivo actual según un orden basado en la orientación de estos con respecto al personaje jugable. ✓

5.4 Workflow

Para terminar el apartado de desarrollo se expone el **flujo de trabajo** adoptado a la hora de **implementar** e **iterar** sobre el conjunto de las **animaciones** utilizadas en el sistema de combate.

1. **Ritmo:** se trabaja sobre **cómo se percibe visualmente la animación**, lanzando los ataques al aire sin golpear a ningún objetivo. Se ajustan los tiempos de *wind-up*, *attack* y *recovery*, así mismo como las transiciones al resto de animaciones de la cadena. En el caso de los **enemigos** la **preparación** y la **recuperación** del ataque se **acentúan**, mientras que para el personaje **jugador** tienden a **acortarse** lo máximo posible. Cuando se trata de una **secuencia de ataques**, se contempla el **ritmo**

generado en conjunto, normalmente optando por retrasar la activación de la hitbox conforme la cadena progresa para obtener una menor cadencia final [85].

2. **Desplazamientos:** en esta fase se **integran** en la animación los **desplazamientos** que se ejecutarán para mover al personaje mientras ataca. Adicionalmente, se decide en que **secciones** de la animación se utilizará **root motion** o no, siendo generalmente las fases de wind-up y recovery donde se usa con más frecuencia, debido a los pequeños ajustes que incorpora la propia animación como la recolocación de pies después de realizar un ataque.
3. **Hitboxes:** se define la ventana de tiempo en que la **hitbox** se mantendrá **activa**. Lo más común es que **coincida** con la fase *attack* de la animación, pero puede extenderse o anticiparse a esta si el ataque lo requiere. Además, se **determina** el **tamaño** de la **hitbox** a utilizar. A partir de este punto, el ataque se encuentra totalmente operativo y puede probarse contra otros personajes.
4. **Knockbacks:** se **determinan** los **knockbacks** que se ejecutan cuando un ataque impacta en un enemigo, esto puede hacerse mismamente desde la propiedad de **magnitud** del propio **ataque**, pero siempre pueden retocarse los desplazamientos generados para un personaje en concreto. En ocasiones es necesario reemplazar el conjunto entero de knockbacks para ajustarse a la sensación que se quiere transmitir, como en el caso de la pelea de jefe de Juno contra Raj, en el que todos los desplazamientos que experimenta el jugador se ven acentuados para vender el impacto de los ataques de Raj.
5. **Cancelaciones:** en la fase de ritmo ya se contemplaron las cancelaciones de animación para generar una transición fluida entre ataques de la misma cadena, en esta sección se definen las **cancelaciones** que tienen que ver con otras **acciones** como esquivar, bloquear, saltar o, sencillamente, moverse. Para los enemigos no es necesario atender minuciosamente a este apartado, pero es importante para el personaje jugador, ya que repercutirá en la capacidad de respuesta percibida.
6. **Gamefeel:** en este último paso se define el **camera shake**, el **gamepad rumble** y el **hit pause** que el ataque aplicará a la hora de impactar. Debido a que el **hit pause** añade una breve pausa es necesario revisar el ritmo de la secuencia de ataques por si genera alteraciones no deseadas en el resultado.

Aunque el flujo de trabajo esté ordenado esto no significa que los pasos sean dependientes unos de otros, solo se trata de una guía que ha ido adaptándose a lo largo del desarrollo según qué resultaba más cómodo y rápido a la hora de implementar las animaciones. Así pues, cada fase puede ser tratada de forma aislada, sin entorpecer el trabajo en las iteraciones futuras a la hora de pulir o modificar aspectos concretos como el ritmo o las cancelaciones.

En este apartado se examinarán los resultados obtenidos a partir de la realización de este trabajo, representados en los sistemas y características que conforman la jugabilidad de “Arcánima: Mist of Oblivion”. Adicionalmente, se expondrán las evaluaciones a los que estos sistemas se han visto sometidos por medio de varios playtests en forma de cuestionarios, realizados durante las diferentes fases del desarrollo, así como los problemas encontrados y las posibles soluciones aplicadas.

6.1 Resultado final

El proyecto concluye obteniendo como resultado el **sistema de combate y locomoción** que conforman el núcleo duro de la **jugabilidad** y, por tanto, de la experiencia de “**Arcánima: Mist of Oblivion**”. El juego se encuentra actualmente en su versión **gold candidate**, con todas las funcionalidades implementadas, siendo el pulido y la resolución de *bugs* las principales tareas.

El producto final de este trabajo se materializa en los siguientes sistemas y características:

- **Sistema de locomoción:** responsable del **movimiento** de **todos los personajes**, posibilitando la navegación por el mapa para el jugador de una manera efectiva y satisfactoria. A través del **subsistema de desplazamientos**, la locomoción trabaja junto con el sistema de combate para llevar a cabo los **knockbacks** y las translaciones efectuadas durante los ataques.
- **Sistema de combate:** vertebra el **gameplay** del juego. El repertorio de **ataques y habilidades** de todos los personajes son gestionados a partir de **animaciones** junto con los **eventos** de animación que proporcionan la funcionalidad adecuada. Además, el sistema integra dinámicas de **debilidades y fortalezas** y se apoya en el sistema de locomoción para crear los desplazamientos oportunos durante el combate.
- **Enfoque orientado al gamefeel:** desde el primer momento se han tenido en cuenta **técnicas de gamefeel** para integrarlas completamente en el sistema de combate, así como los **principios fundamentales** que afectan al juego en su totalidad. Esto ha dado como resultado un conjunto de funcionalidades- usadas tanto dentro del combate como fuera de él- que se materializan tanto en aspectos generales como

proporcionar una **alta capacidad de respuesta** o **cumplir la intención del jugador** como en elementos concretos con la utilización de **camera shake**, **gamepad rumble** o **hit pause**.

- **Un total de seis personajes: un personaje jugable, dos jefes y tres enemigos** forman el elenco completo de “Arcanima”, cada uno con sus propias peculiaridades en su repertorio de ataques, desde combate aéreo (Juno) o ataques a distancia (Automata) hasta poder teletransportarse libremente por el escenario (Nyx)

6.2 Evaluación de sistemas

Al terminar cada una de las fases marcadas en el desarrollo se obtuvo como resultado un prototipo sobre el que realizar **playtesting**. Los datos aquí presentados fueron recabados a partir de **formularios** basados en **preguntas** con **escala de puntuación** dando la posibilidad de que el encuestado desarrolle su opinión.

Al tratarse de preguntas con puntuación se tratarán como respuestas positivas todas aquellas que estén por encima de la mitad de la escala, esto es, en una escala del ‘1’ a ‘5’, las puntuaciones de ‘4’ y ‘5’ serán las deseables. Al atender a estas puntuaciones a lo largo de los diferentes playtests se obtendrá una **visión objetiva** sobre la posible **mejoría** de los aspectos evaluados. Lo ideal sería obtener la mayor cantidad de puntuaciones altas posibles atendiendo al feedback recibido para garantizar que los **sistemas** funcionen correctamente y sean satisfactorios para los jugadores.

Aunque los formularios empleados llegan a tratar cuestiones sobre diversas disciplinas, en esta sección solo se expondrán los datos y resultados relacionados con la materia de este trabajo. En total se realizaron tres test, cada uno con el propósito de evaluar ciertos aspectos concretos del juego.

Playtest de navegación

Este **playtest** se realizó con el objetivo de **evaluar el diseño del nivel** y examinar cómo el jugador se desenvuelve dentro del mismo. El principal interés con respecto a este trabajo se trata del resultado obtenido sobre la **sensación de movimiento**, es decir, si el jugador ha encontrado satisfactorio o no el desplazarse por el entorno virtual.

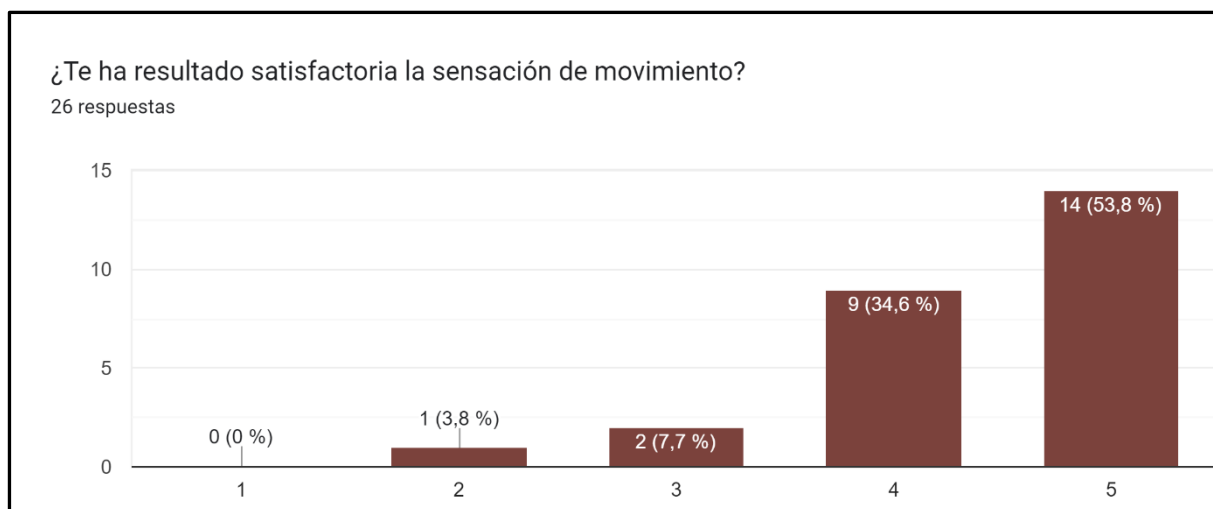


Ilustración 129- Respuestas sobre la sensación de movimiento

De los **26** participantes encuestados, el **88,4%** considera que la sensación de movimiento es **bastante o muy satisfactoria**. Los principales problemas encontrados en los comentarios de los *testers* son respecto al comportamiento de la cámara que, aunque hubiese una pregunta dedicada a ella en específico, ha influido también en la experiencia general de movimiento.

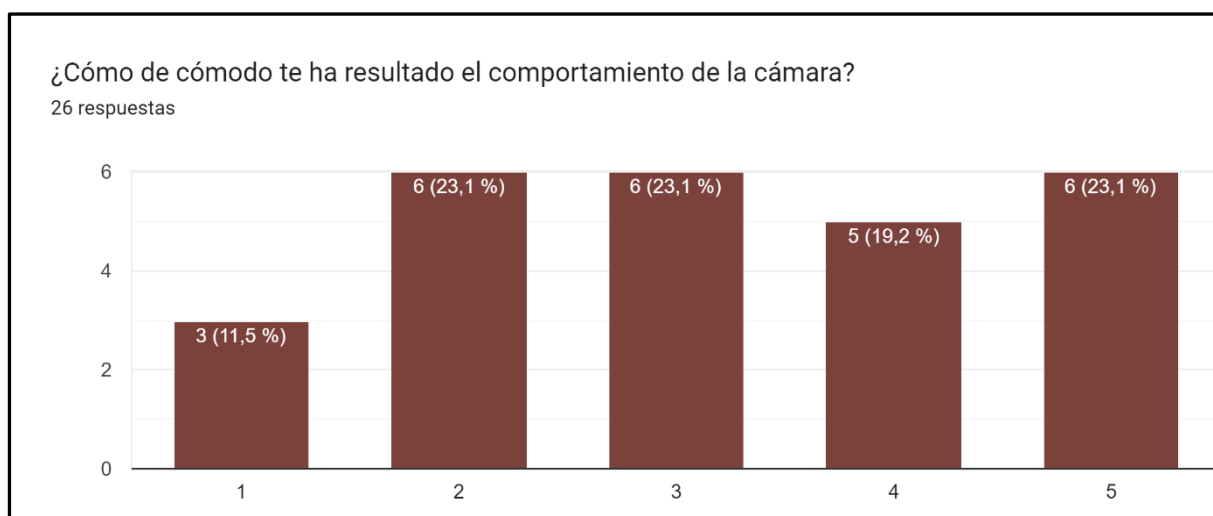


Ilustración 130- Respuestas sobre el comportamiento de la cámara

Aunque la cámara quede fuera del alcance del proyecto, se puede observar la gran variedad en los resultados, siendo un poco menos de la mitad de los encuestados, el **42.3%**, los que consideran que la **cámara ha sido cómoda de manejar**. En contraposición con los resultados de la sensación de movimiento, este es un claro ejemplo de distribución en la puntuación que revela de forma muy clara un aspecto a mejorar.

Comparando los resultados de ambas preguntas se puede observar que, aunque la cámara necesite bastante pulido y algunos encuestados la hayan identificado como un problema para la locomoción, esta no afecta en gran medida a la propia sensación de movimiento.

Playtest de combate

Este cuestionario tuvo como objetivo la evaluación de la **sensación** y las **mecánicas de combate**. El test empezaba con unos pequeños **tutoriales** para que los jugadores aprendiesen los **controles**, para más tarde ponerlos a prueba en **enfrentamientos de diversas dificultades**. Una vez superadas las primeras pruebas, se realizaba una **arena de batalla** compuesta por un total de **diez oleadas** en las que aparecían diferentes configuraciones de enemigos. Aunque **solo existía un enemigo implementado** (soldado raso) en el momento de la realización de este playtest, se crearon diferentes versiones más poderosas del mismo para emular a los enemigos pendientes de implementar.

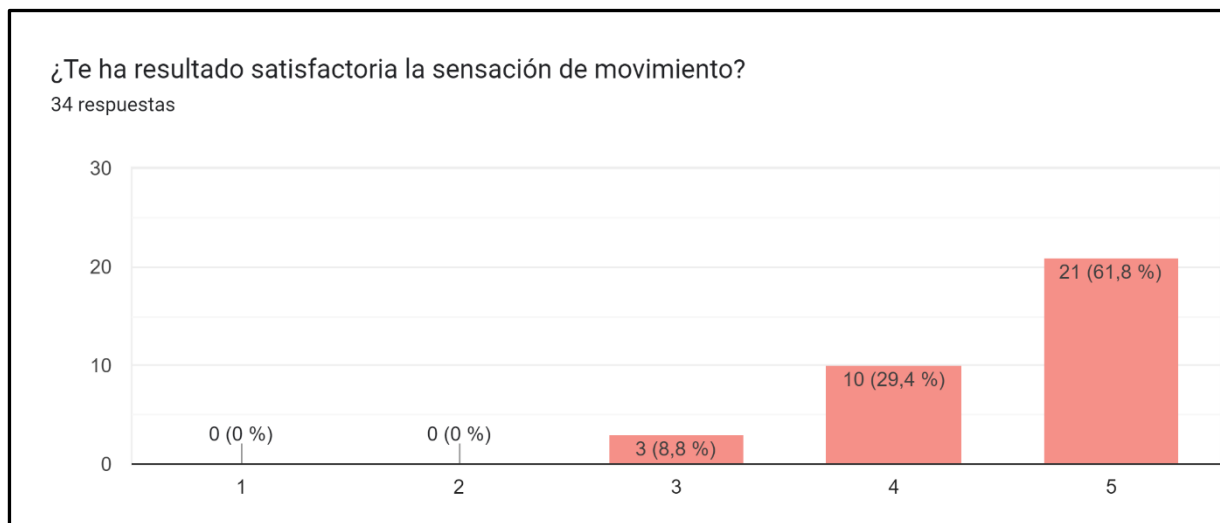


Ilustración 131- Respuestas sobre la sensación de movimiento en combate

De nuevo se evalúa la sensación de movimiento, pero esta vez enmarcada en situaciones de combate, muy diferente al carácter de navegación presentado en el primer test. De los **34** participantes, el **91,2%**, considera que la sensación de movimiento es **bastante o muy satisfactoria**, siendo una **mejora** de casi **tres puntos** con respecto al test anterior.

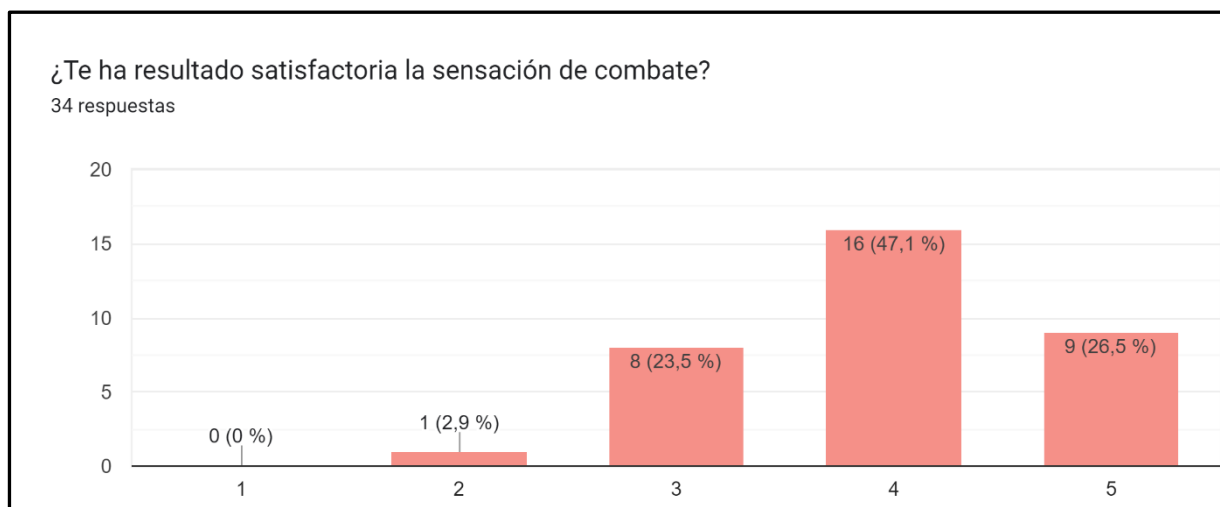


Ilustración 132- Resultados sobre la sensación de combate

En cuanto a la sensación de combate, se obtiene de los resultados que el **73,6%** de los participantes consideran que es **bastante o muy satisfactoria**. El **mayor problema** indicado por los encuestados señalaba a la **fluidez entre ataques** y a la **poca rapidez** de estos, así como a la **escasa anticipación** que presentan los **enemigos** en sus ataques. También se dejó entrever que algunas mecánicas no funcionaban correctamente debido a errores de programación que, naturalmente, afectaron a la experiencia.

De una forma más general, este test dio a luz la **estrategia dominante** utilizada durante los combates. Esta se basaba en realizar en bucle los **dos primeros ataques del combo ligero** y después **huir** utilizando el **dash** sin utilizar apenas las habilidades del modo concentración, lo que se alejaba en gran medida de la experiencia de combate objetivo deseada.

Las causas de este resultado se podrían explicar atendiendo a la **gran cantidad de enemigos** que se llegaron a disponer en ciertos momentos del test, a la **poca anticipación** que estos enemigos presentaban a la hora de atacar y lo **implacables** que resultaban **persiguiendo** al jugador, llegando a atosigarle la mayor parte del tiempo. Esto, unido a la relativa **lentitud** de la mayoría de los **ataques**, incrementaba la sensación de vulnerabilidad del jugador que desarrollaba como mejor estrategia el atacar y esquivar continuamente, evitando la confrontación prolongada para no resultar acorralado.

Alpha Playtest

En el test de la versión *alpha* se incluyó todo el contenido del juego, esto es, los **nuevos enemigos** que faltaban por implementar (soldado blindado y automata) y la **pelea final** contra **Nyx**. Sin embargo, por causas organizativas y de prioridad, el combate contra el primer jefe, Raj, no se llegó a introducir.

Esta prueba tenía como objetivo **obtener** una **visión** a grandes rasgos sobre la **experiencia global del juego**, de principio a fin, con el fin de observar si los jugadores comprendían las mecánicas con los tutoriales implementados y si los cambios tanto en la navegación del nivel como en el combate mejoraban los problemas encontrados en los playtests anteriores.

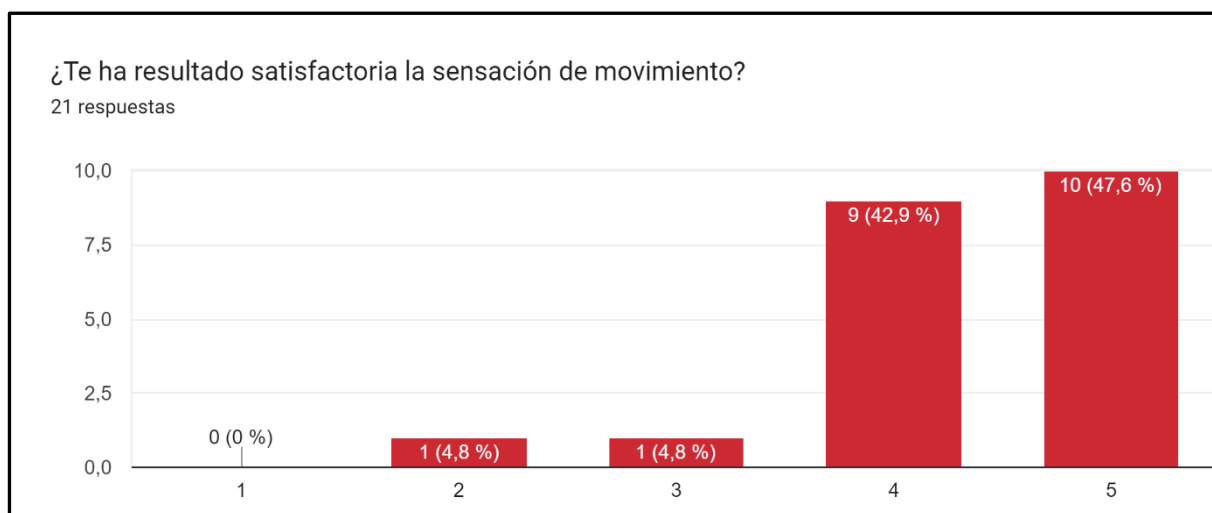


Ilustración 133- Respuestas sobre la sensación de movimiento en el alpha test

La percepción sobre el movimiento se mantiene estable, de los **21** participantes, el **90,5%** sigue pensando que el movimiento es bastante o muy satisfactorio.

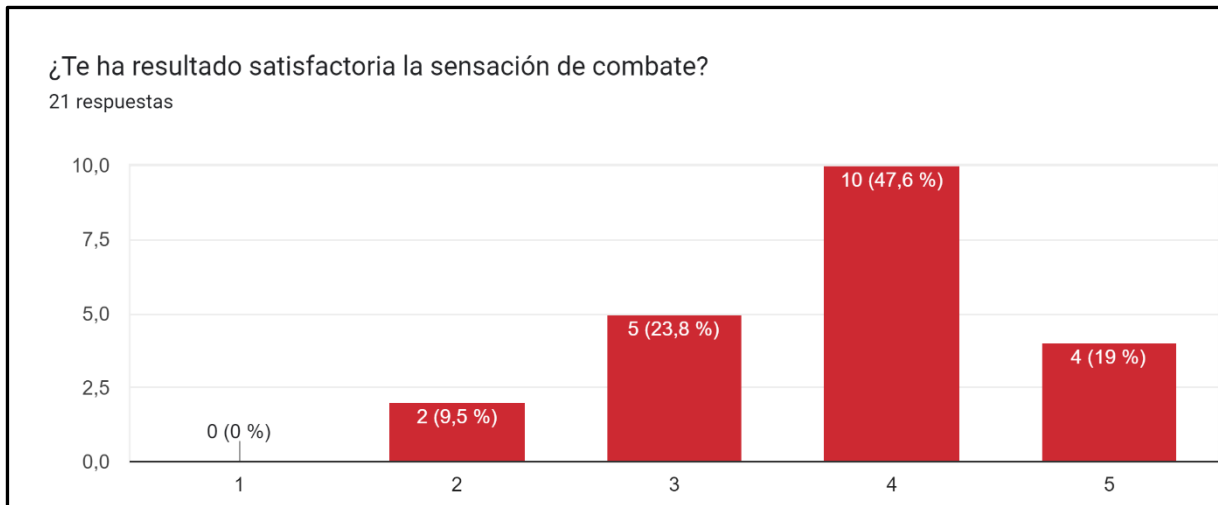


Ilustración 134- Respuestas sobre la sensación de combate en el alpha test

Con respecto a la sensación de combate se ha experimentado una **bajada notable de siete puntos**, siendo el **66.6%** de los encuestados los que consideran que el combate es bastante o muy satisfactorio.

La posible explicación a estos resultados se puede encontrar en los propios comentarios de los jugadores donde marcan como causa a los **nuevos enemigos implementados**. Debido a las limitaciones de solo poder utilizar al soldado raso como único enemigo en el test de combate, los problemas ahora resaltan en los nuevos personajes con cuestiones similares: la falta de anticipación, la gran cantidad de daño que infligen o la implacabilidad con la que estos persiguen al jugador sin darle un respiro.

Con la implementación de algunos arreglos al combate como el aumento de la velocidad de los ataques del jugador, se consiguió **eliminar en mayor medida la estrategia dominante** de golpear y esquivar en bucle. Aun así, esta seguía **apareciendo en momentos** cuando el **jugador** se encontraba mayormente **abrumado** y sin capacidad de desenvolverse cómodamente, como al enfrentarse a los soldados blindados o en la batalla final contra Nyx. Como resultado, **la aparición de esta estrategia se volvió un indicador** para detectar que algo no andaba bien y necesitaba una revisión, como bien coincidía con los comentarios de los encuestados.

Playtests en ferias

Una versión temprana de la Beta se presentó en ferias del sector como el **Indie Dev Day**, la **Bilbao International Game Conference (BIG)**, la **Guerrilla Game Festival** y la **Gamergy**. En ella se incluía todo el contenido del juego a excepción de la batalla contra Raj y un escenario posterior a la misma, la pelea final contra Nyx seguía siendo accesible y todo el entorno estaba completamente texturizado.

Aunque no se llegó a realizar ningún cuestionario por descanso del equipo, sí que se estuvieron observando atentamente todos los *playtest* realizados. Con la **incorporación de soluciones** a los problemas surgidos en el anterior *playtest*, como una mayor **ventana de anticipación** por parte de los **enemigos**, un **comportamiento más pasivo** por parte de estos y algunos **ajustes** en el **daño** que infligían en el jugador, **la estrategia de golpear y esquivar desapareció por completo de la experiencia**. El resultado no fue un juego más sencillo, sino un juego que se dejaba jugar y disfrutar manteniendo un nivel de desafío asequible.

Conclusiones

En este último capítulo se efectuará una breve evaluación sobre el cumplimiento de cada uno de los objetivos propuestos al principio de este trabajo. Seguidamente, se presentará un análisis de las lecciones aprendidas que más relevancia han tenido a lo largo del desarrollo de "Arcánima" y, finalmente, se concluirá con el posible futuro del juego.

7.1 Logros alcanzados

A continuación, se procede a realizar una valoración del grado de cumplimiento de los objetivos planteados para este trabajo:

- **Diseñar e implementar un sistema de locomoción.**

En el apartado de desarrollo se ha expuesto la implementación de los requisitos requeridos para lograr el movimiento de un personaje que posibilite la navegación por un entorno virtual 3D. El sistema se basa en una **simulación** que contempla **aceleraciones y fricciones, desplazamientos, saltos y agarres**, con la gran ventaja de poder ser compartido entre diferentes personajes independientemente de si estos son controlados por el jugador o no.

Adicionalmente, el sistema posee la flexibilidad suficiente como para modificar los valores que definen la simulación, pudiendo así crear diferentes tipos de movimientos personalizados fácilmente, incluso en tiempo de ejecución, de una manera rápida y accesible para los integrantes no programadores del equipo.

Con todo lo expuesto, se considera que el grado de cumplimiento de este objetivo ha sido **totalmente satisfactorio**.

- **Diseñar e implementar un sistema de combate.**

Todas las **tablas de cadenas de ataques** expuestas en las fichas de personajes del apartado de diseño de juego han sido **implementadas** gracias al **sistema de combate** desarrollado en base a los requisitos extraídos de estas mismas tablas.



Como se ha expuesto en la sección de desarrollo, este sistema se rige en gran parte por las **animaciones** y los **eventos de animación** que conforman el conjunto de ataques. Las animaciones marcan el ritmo y la fluidez de la cadena mientras que los eventos aportan la funcionalidad que vertebran la lógica necesaria para que el ataque funcione como es debido: activar/desactivar *hitboxes*, establecer ventanas de cancelación, habilitar desplazamientos, etc.

Por otro lado, en este sistema se han implementado funcionalidades más complejas aparte de llevar a cabo una sucesión de ataques, como puede la capacidad de realizar **bloqueos y contraataques**, efectuar **habilidades especiales** en tiempo parado (modo concentración), incorporar **fortalezas y debilidades** o la implementación del **combate aéreo**.

A todo esto, se le suma la **gran flexibilidad del sistema** de combate para poder modificar e iterar sobre cualquier aspecto de un ataque concreto gracias a que se construye sobre sistemas ya instauradas en el propio motor como *Mecanim* o los *Animation Events*.

Por ende, este objetivo se califica como **completamente satisfactorio**.

- **Investigar y aplicar principios y técnicas de *gamefeel*.**

Como se puede observar en el apartado teórico de este trabajo, se ha realizado un **estudio** de los **principios** y **técnicas** generales que logran obtener una mejor sensación de juego para el tipo de juego a desarrollar.

Desde los sistemas aquí contemplados, se han tenido en cuenta desde el primer momento, tanto es su diseño como implementación, estos principios y técnicas, materializándose en pautas o guías generales a la hora de trabajar o en funcionalidades concretas como pueden ser la cancelación de animaciones, el *hitpause* o el *camera shake*.

Aun así, como se ha podido observar en las conclusiones sacadas en las encuestas realizadas en el apartado de validación, aunque estos principios y técnicas **son necesarios** para conseguir una experiencia global óptima, **no son suficientes** por sí mismos.

Debido a lo expuesto, el grado de cumplimiento de este objetivo se considera **satisfactorio, aunque no ideal**.

- **Gestionar e implementar las animaciones requeridas por los sistemas de combate y locomoción.**

Durante todo el desarrollo se ha llevado a cabo un uso extensivo y completo de *Mecanim* para poder implementar las animaciones requeridas por el juego, llegando

incluso a integrar ciertas funcionalidades consideradas básicas para este trabajo que no vienen incluidas en el motor.

Aunque en el sistema de locomoción pueda dividirse perfectamente la lógica (simulación) del apartado visual (animaciones), el sistema de combate se construye en su mayor parte sobre la gestión que se realiza de las animaciones involucradas. La implementación de estas animaciones se materializa en el flujo de trabajo descrito en el desarrollo, en el que se utilizan todas las herramientas y funcionalidades construidas para este cometido.

Atendiendo a lo expuesto, el grado de cumplimiento de este objetivo se considera **satisfactorio**.

7.2 Lecciones aprendidas

Durante el proceso de iteración del sistema de combate, y gracias a los playtesting realizados, se llegó a la conclusión de que **no es posible tratar de igual manera a todos los personajes** desde un punto de vista jugable, ya que estos aportan y suplen diferentes necesidades para el jugador.

Atendiendo al **avatar jugable**, este necesita ser **altamente reactivo** y con **gran capacidad de respuesta** hacia los comandos que lance el jugador, dando la sensación de inmediatez. Independientemente de lo rápida o lenta que sea la acción a realizar por el personaje jugable, el jugador necesita cualquier tipo de indicativo reconocible que implique que su input ha sido respondido.

En cuanto a los **enemigos**, estos necesitan **proporcionar información** al jugador, necesitan transmitir señales claras que anticipen sus intenciones. Estas señales pueden ser alteradas para modificar la dificultad del personaje, pero nunca obviadas debido a que se estaría privando de información al jugador, dando como resultado una mala sensación de juego.

Atendiendo a la **manera de trabajar**, una lección aplicada ha sido enfocar el **flujo de trabajo** de **macro a micro**, por ejemplo, primero atendiendo a los rasgos generales como el ritmo de una secuencia de ataques o sus transiciones, y luego centrarse en los detalles de cada ataque individualmente. Esto produce un acercamiento de lo general a lo concreto que facilita la organización y resulta más abarcable.

Como **lección de diseño**, se ha estudiado que el **gamefeel** forma una **parte fundamental** de las mecánicas; no es suficiente con que una mecánica funcione bien, sino que también debe de sentirse bien. El gamefeel debe ser considerado una **pieza más de la jugabilidad** y no ser tratado como una tarea de pulido en una fase tardía del desarrollo como cabría pensar, sino al contrario, el **gamefeel** debe **existir desde el principio** y ser pulido como cualquier otro elemento del juego.

En cuanto al proyecto en sí, una de las principales lecciones ha sido **no subestimar la cantidad** de carga de **trabajo** que puede llegar a generar una idea concreta. *“Arcanima: Mist*

of Oblivion” se planteó como un **proyecto** de un **año de duración** el cual se ha sobrepasado enormemente siendo actualmente un desarrollo de casi dos años y medio. La inexperiencia a la hora de enfrentarse a un proyecto tan grande ha sido la principal causa de ello, ya sea por no estimar correctamente las tareas, pecar de ambiciosos o no tener suficiente formación para lograrlo. Se podría decir que este tipo de experiencias son necesarias para ser consciente del trabajo que conlleva desarrollar un juego de este calibre, aunque lo ideal hubiese sido haberse podido equivocar y experimentar con un proyecto mucho más reducido, ya que no es necesario llegar a esta conclusión con un proyecto de la talla de *“Arcanima”*.

7.3 Líneas futuras

Como futuro más inmediato, todos los esfuerzos del equipo estarán puestos en realizar controles de calidad de sus respectivos campos y pulir los aspectos del juego que más lo necesiten para su próxima publicación en *Steam*. Dependiendo de la acogida que *“Arcanima”* tenga en el mercado se podría llegar a añadir contenido que resultó descartado debido a las limitaciones de tiempo:

- **Mayor variedad de enemigos:** al inicio del desarrollo se llegaron a plantear tres enemigos adicionales que finalmente fueron excluidos. Estos se trataban de un autómatas de apoyo no ofensivo que serviría de ayuda a los enemigos otorgándoles mejoras en el tiempo, un fusilero que se mantendría lejos y en lugares altos para atacar al jugador a grandes distancias y de un especialista (*rogue*) que acosaría al jugador con sus rápidos ataques y gran movilidad.
- **Mayor repertorio de acciones por parte de los jefes:** las versiones actuales de los jefes existentes en el juego están capadas con respecto a cómo fueron planteadas inicialmente en el GDD. Raj en sus inicios tenía la capacidad de realizar un ataque a distancia y de poder bloquear y contraatacar los golpes del jugador, al igual que Nyx el cual tenía más cadenas de ataques y más fases de batalla.
- **Modo “práctica de combate”:** se trata de una simple arena en la que el jugador puede practicar combos y habilidades, pudiendo colocar a cualquier número y tipo de enemigos. Además, siguiendo el ejemplo de *“Nier: Automata”* [86], se proporcionarían *cheatcodes* para ofrecer un espacio seguro en el que combatir y probar nuevas estrategias, a la vez que, de forma interna, actuarían como herramientas para agilizar el desarrollo e iterar más rápidamente en la implementación de nuevos personajes.



Bibliografía

- [1] G. B. V., «Euston96,» 2019. [En línea]. Available: <https://www.euston96.com/en/hack-and-slash-en/>. [Último acceso: 3 5 2023].
- [2] C. Hovermale, «Destructoid,» 10 3 2019. [En línea]. Available: <https://www.destructoid.com/how-devil-may-cry-arcade-inspirations-shaped-character-action-games/>. [Último acceso: 29 5 2023].
- [3] J. W. & K. Mohan, «Women want equality -and why not?,» *Dragon Magazine*, nº 39, 1980.
- [4] C. Fellrin, «Youtube,» 5 3 2021. [En línea]. Available: <https://www.youtube.com/watch?v=Me4LLrlpMMA>. [Último acceso: 29 5 2023].
- [5] «Nintendo Software,» *Computer Entertainer*, vol. 6, nº 5, 1987.
- [6] C. Khan, «IGN Souteast Asia,» 9 1 2013. [En línea]. Available: <https://sea.ign.com/conan-the-dark-axe/60097/blog/evolution-of-hack-and-slash-games>. [Último acceso: 29 5 2023].
- [7] Bitmap Books, *A Guide to Japanese Role-Playing Games*, Bitmap Books, 2021.
- [8] «SteamDB,» [En línea]. Available: <https://steamdb.info/charts/?tagid=323922>. [Último acceso: 29 5 2023].
- [9] Gamerzlounge, «Gamerzlounge,» 14 8 2019. [En línea]. Available: <https://gamerzlounge.me/blog/Evolution-hack-n-slash>. [Último acceso: 29 5 2023].
- [10] B. Stegner, «MakeUseOf,» 12 9 2021. [En línea]. Available: <https://www.makeuseof.com/what-are-hack-and-slash-video-games/>. [Último acceso: 29 5 2023].
- [11] «SteamDB,» [En línea]. Available: <https://steamdb.info/charts/?tagid=3955>. [Último acceso: 29 5 2023].
- [12] «SteamDB,» [En línea]. Available: <https://steamdb.info/charts/?tagid=29482>. [Último acceso: 29 5 2023].
- [13] MetalMusicMan, «Ultimate Frame Data,» [En línea]. Available: <https://ultimateframedata.com/>. [Último acceso: 31 5 2023].
- [14] S. Masala, «Code Captain,» 27 10 2019. [En línea]. Available: <https://www.codecaptain.io/blog/game-development/improving-performance-in-your->



- games-using-spritesheets. [Último acceso: 31 5 2023].
- [15] A. Löw, «Code and Web,» [En línea]. Available: <https://www.codeandweb.com/what-is-a-sprite-sheet-performance>. [Último acceso: 31 5 2023].
- [16] A. Nuñez, «Bringing 2D characters to life with sprite rigging,» 2019 10 14. [En línea]. Available: <https://www.youtube.com/watch?v=vap04-Py9QM>.
- [17] «Animation Explainers,» 4 2 2022. [En línea]. Available: <https://animationexplainers.com/what-is-paper-cut-out-animation/>. [Último acceso: 1 6 2023].
- [18] D. Rosen, «Animation Bootcamp: An Indie Approach to Procedural Animation,» 2014. [En línea]. Available: <https://www.youtube.com/watch?v=LNidsMesxSE>.
- [19] Epic Games, «Unreal Engine Docs - IK Setups,» [En línea]. Available: [https://docs.unrealengine.com/4.27/en-US/AnimatingObjects/SkeletalMeshAnimation/IKSetups/#::~:~:text=Inverse%20Kinematics%20\(IK\)%20provide%20a,location%20as%20best%20it%20can..](https://docs.unrealengine.com/4.27/en-US/AnimatingObjects/SkeletalMeshAnimation/IKSetups/#::~:~:text=Inverse%20Kinematics%20(IK)%20provide%20a,location%20as%20best%20it%20can..) [Último acceso: 1 6 2023].
- [20] M. Mach, «Physics Animation in Uncharted 4: A Thief's End,» 2017. [En línea]. Available: https://youtu.be/7S-_vuoKgR4.
- [21] M. Nava, «Creating the Art of ABZU,» 2017. [En línea]. Available: <https://youtu.be/I9NX06mvp2E>.
- [22] K. Zadziuk, «Motion Matching, The Future of Games Animation... Today,» 2016. [En línea]. Available: <https://youtu.be/KSTn3ePdt50>.
- [23] H. Z. T. K. J. S. Sebastian Starke, «Neural State Machine for Character-Scene Interactions,» 2019.
- [24] O. K. M. P. T. P. Daniel Holden, «Learned Motion Matching,» 2020.
- [25] S. Clavet, «Machine Learning Summit: Ragdoll Motion Matching,» 2020. [En línea]. Available: <https://youtu.be/JZKaQKcAnw>.
- [26] P. A. S. L. M. V. d. P. Xue Bin Peng, «DeepMimic: Example-Guided Deep Reinforcement Learning of Physics-Based Character Skills,» 2018.
- [27] D. G. J. H. Jungdam Won, «A Scalable Approach to Control Diverse Behaviors for Physically Simulated Characters,» 2020.
- [28] L. H. e. a. Josh Merel, «Neural probabilistic motor primitives for humanoid control,» 2019.



- [29] S. C. D. H. J. R. F. Kevin Bergamin, «DReCon: Data-Driven Responsive Control of Physics-Based Characters,» 2019.
- [30] Y. Z. T. N. K. Z. Sebastian Starke, «Local Motion Phases for Learning Multi-Contact Character Movements,» 2020.
- [31] Y. G. M. S. S. F. Tingwu Wang, «UniCon: Universal Neural Controller For Physics-based Character Motion,» 2020.
- [32] DEV, «Libro Blanco del Desarrollo Español de Videojuegos,» 2022.
- [33] Epic Games, «Resumen de la tecnología en tiempo real: el estado del 3D interactivo,» 14 2 2023. [En línea]. Available: <https://www.unrealengine.com/es-ES/blog/real-time-round-up-the-state-of-interactive-3d>. [Último acceso: 5 6 2023].
- [34] L. F. Josh Camas, «Unity Forums: Reusing Sub-State Machines,» 13 12 2017. [En línea]. Available: <https://forum.unity.com/threads/reusing-sub-state-machines.508645/>. [Último acceso: 5 6 2023].
- [35] Epic Games, «Gameplay Framework,» [En línea]. Available: <https://docs.unrealengine.com/5.0/en-US/gameplay-framework-in-unreal-engine/>. [Último acceso: 5 6 2023].
- [36] Epic Games, «Skeletal Mesh Animation System,» [En línea]. Available: <https://docs.unrealengine.com/4.27/en-US/AnimatingObjects/SkeletalMeshAnimation/>. [Último acceso: 5 6 2023].
- [37] Epic Games, «Hot Reload and Live Coding,» [En línea]. Available: <https://unrealcommunity.wiki/live-compiling-in-unreal-projects-tp14jcgs>. [Último acceso: 5 6 2023].
- [38] S. D. Jong, «Blueprints In-depth - Part 1 | Unreal Fest Europe 2019 | Unreal Engine,» 2019. [En línea]. Available: <https://www.youtube.com/watch?v=j6mskTgL7kU>. [Último acceso: 5 6 2023].
- [39] J. Schell, The Art of Game Design, CRC Press, 2020.
- [40] N. Gladstein, «Frame-specific attacks in Unity,» 3 9 2029. [En línea]. Available: <https://www.gamedeveloper.com/design/frame-specific-attacks-in-unity>. [Último acceso: 6 6 2023].
- [41] S. Rogers, Level Up! The Guide to Great Video Game Design, Wiley, 2014.
- [42] P. García, «Eurogames,» 24 10 2022. [En línea]. Available: <https://www.eurogamer.es/como-empezar-con-el-genero-hack-slash-sagas-y-titulos-impresdindibles>. [Último acceso: 6 6 23].



- [43] Just Add Monsters, «Kung Fu Story Design Documentation Combat Mechanics,» 20003.
- [44] S. Swink, Game Feel: A Game Designer's Guide To Virtual Sesation, Morgan Kaufmann, 2009.
- [45] Adobe, «Adobe Tweening,» [En línea]. Available: <https://www.adobe.com/creativecloud/video/discover/tweening.html>. [Último acceso: 6 6 2023].
- [46] I. s. Andrey Sitnik, «Easings,» [En línea]. Available: <https://easings.net/>. [Último acceso: 6 6 2023].
- [47] Catalyst, «Catalyst - Timing Windows,» 26 5 2021. [En línea]. Available: <https://gamedev.catalystsoftworks.com/hidden-mechanics/timing-windows/>. [Último acceso: 7 6 2023].
- [48] J. d. Heras, «Jason de Heras - Hit Pause,» 22 4 2021. [En línea]. Available: <https://twitter.com/jasondeheras/status/1385104141383475200>. [Último acceso: 7 6 2023].
- [49] J. d. Heras, «Jason de Heras - Game Slowdown,» 22 4 2021. [En línea]. Available: <https://twitter.com/jasondeheras/status/1385104111192797186>. [Último acceso: 7 6 2023].
- [50] J. d. Heras, «Jason de Heras - Game Freeze,» 22 4 2021. [En línea]. Available: <https://twitter.com/jasondeheras/status/1385103977033854986>. [Último acceso: 7 6 2023].
- [51] V. Napoli, «Combat Recall,» 12 5 2018. [En línea]. Available: <http://combatrecall.blogspot.com/2018/05/controller-rumble-controller-rumble-is.html>. [Último acceso: 7 6 2023].
- [52] F. T. Ollie Johnston, The Illusion of Life: Disney Animation, 1995.
- [53] Unity Technologies, «Unity Documentation - Animation Tab,» [En línea]. Available: <https://docs.unity3d.com/Manual/class-AnimationClip.html>. [Último acceso: 7 6 2023].
- [54] Unity Technologies, «Unity Docuemntation - Blend Trees,» [En línea]. Available: <https://docs.unity3d.com/Manual/class-BlendTree.html>. [Último acceso: 8 6 2023].
- [55] Unity Technologies, «Unity Documentation - 2D Blending,» [En línea]. Available: <https://docs.unity3d.com/Manual/BlendTree-2DBlending.html>. [Último acceso: 8 6 2023].
- [56] Unity Technologies, «Unity Documentation - Animation Layers,» [En línea]. Available: <https://docs.unity3d.com/Manual/AnimationLayers.html>. [Último acceso: 8 6 2023].
- [57] Unity Technologies, «Unity Documentation - Animation System Overview,» [En línea]. Available: <https://docs.unity3d.com/Manual/AnimationOverview.html>. [Último acceso: 8 6 2023].



- [58] Unity Technologies, «Unity Documentation - Retargeting of Humanoid animations,» [En línea]. Available: <https://docs.unity3d.com/Manual/Retargeting.html>. [Último acceso: 8 6 2023].
- [59] Unity Technologies, «Unity Documentation - Humanoid Avatars,» [En línea]. Available: <https://docs.unity3d.com/Manual/AvatarCreationandSetup.html>. [Último acceso: 8 6 2023].
- [60] K. Iglesias, «Humanoid Animation Retargeting Tutorial - Unity 3D,» 23 5 2022. [En línea]. Available: <https://www.youtube.com/watch?v=kaURB6jHYOQ>. [Último acceso: 8 6 2023].
- [61] Unity Technologies, «Unity Documentation - Root Motion,» [En línea]. Available: <https://docs.unity3d.com/Manual/RootMotion.html>. [Último acceso: 8 6 2023].
- [62] Unity Technologies, «Unity Documentation - Scripting Root Motion for "in-place" humanoid animations,» [En línea]. Available: <https://docs.unity3d.com/Manual/ScriptingRootMotion.html>. [Último acceso: 8 6 2023].
- [63] Unity Technologies, «Unity Documentation - Character Controller,» [En línea]. Available: <https://docs.unity3d.com/Manual/class-CharacterController.html>. [Último acceso: 8 6 2023].
- [64] Unity Technologies, «Unity Documentation - Rigidbody,» [En línea]. Available: <https://docs.unity3d.com/es/2019.4/Manual/class-Rigidbody.html>. [Último acceso: 8 6 2023].
- [65] Yotsumaru, «Nier Replicant - Animation Cancel: shortcuts,» 9 5 2021. [En línea]. Available: <https://www.youtube.com/watch?v=F8QMp7b00DA>. [Último acceso: 9 6 2023].
- [66] Unity Technologies, «Unity Documentation - Trail Renderer Component,» [En línea]. Available: <https://docs.unity3d.com/es/2018.4/Manual/class-TrailRenderer.html>. [Último acceso: 13 6 2023].
- [67] J. Flick, «Cats like coding - Sliding a Sphere,» 25 10 2019. [En línea]. Available: <https://catlikecoding.com/unity/tutorials/movement/sliding-a-sphere/>. [Último acceso: 13 6 2023].
- [68] K. Pittman, «Math for Game Programmers: Building a Better Jump,» 12 12 2016. [En línea]. Available: <https://youtu.be/hG9SzQxaCm8>. [Último acceso: 14 6 2023].
- [69] W. Armstrong, «Five tips for keeping animator controllers nice n' tidy,» 8 2021. [En línea]. Available: <https://unity.com/es/how-to/build-animator-controllers#reuse-patterns>. [Último acceso: 13 6 2023].
- [70] Unity Technologies, «Unity Documentation - On Animator Move,» [En línea]. Available: <https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnAnimatorMove.html>. [Último acceso: 14 6 2023].
- [71] Unity Technologies, «Unity Documentation - Physics.Raycast,» [En línea]. Available:



- <https://docs.unity3d.com/ScriptReference/Physics.Raycast.html>. [Último acceso: 15 6 2023].
- [72] Unity Technologies, «Unity Documentation - Uses of layers in Unity,» [En línea]. Available: <https://docs.unity3d.com/Manual/use-layers.html>. [Último acceso: 15 6 2023].
- [73] Unity Technologies, «Unity Learn - Working with NavMesh Agents,» [En línea]. Available: <https://learn.unity.com/tutorial/working-with-navmesh-agents#>. [Último acceso: 15 6 2023].
- [74] M. Agudo, Patrones de diseño aplicados a videojuegos. Ejemplo práctico: arquitectura de personajes en "Arcanima: Mist of Oblivion", 2022.
- [75] Unity Technologies, «Unity Documentation - Collider.OnTriggerEnter,» [En línea]. Available: <https://docs.unity3d.com/ScriptReference/Collider.OnTriggerEnter.html>. [Último acceso: 15 6 2023].
- [76] Unity Technologies, «Unity Documentation - Physics.OverlapBox,» [En línea]. Available: <https://docs.unity3d.com/ScriptReference/Physics.OverlapBox.html>. [Último acceso: 15 6 2023].
- [77] B. Ruiz, «The Amalur Problem,» 24 5 2012. [En línea]. Available: <https://aztez.com/blog/2012/05/24/the-amalur-problem/>. [Último acceso: 15 6 2023].
- [78] Unity Technologies, «Unity Manual - Cinemachine,» [En línea]. Available: <https://docs.unity3d.com/Packages/com.unity.cinemachine@2.9/manual/index.html>. [Último acceso: 16 6 2023].
- [79] Unity Technologies, «Cinemachine Documentation - Cinemachine Impulse Source,» [En línea]. Available: <https://docs.unity3d.com/Packages/com.unity.cinemachine@2.3/manual/CinemachineImpulseSourceOverview.html#:~:text=An%20Impulse%20Source%20is%20a,and%20the%20source%20generates%20impulses..> [Último acceso: 16 6 2023].
- [80] Unity Technologies, «Input System Documentation - Interface IDualMotorRumble,» [En línea]. Available: https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/api/UnityEngine.InputSystem.Haptics.IDualMotorRumble.html#UnityEngine_InputSystem_Haptics_IDualMotorRumble_SetMotorSpeeds_System_Single_System_Single_. [Último acceso: 16 6 2023].
- [81] Unity Technologies, «Unity Documentation - Animator.speed,» [En línea]. Available: <https://docs.unity3d.com/ScriptReference/Animator-speed.html>. [Último acceso: 16 6 2016].
- [82] Unity Technologies, «Unity Documentation - Time.timeScale,» [En línea]. Available: <https://docs.unity3d.com/ScriptReference/Time-timeScale.html>. [Último acceso: 16 6 2016].
- [83] Unity Technologies, «Unity Documentation - Time.unscaledDeltaTime,» [En línea]. Available: <https://docs.unity3d.com/ScriptReference/Time-unscaledDeltaTime.html>. [Último acceso: 16



6 2023].

- [84] «Unity Documentation - AnimatorUpdateMode,» [En línea]. Available: <https://docs.unity3d.com/ScriptReference/AnimatorUpdateMode.html>. [Último acceso: 16 6 2023].
- [85] B. Ruiz, «Aztez - Frame by Frame: The Sword's Basics,» 8 9 2014. [En línea]. Available: <https://aztez.com/blog/2014/09/08/frame-by-frame-the-swords-basics/>. [Último acceso: 21 6 2023].
- [86] «Nier Wiki - Debug,» [En línea]. Available: <https://nier.fandom.com/es/wiki/Debug>. [Último acceso: 12 6 2023].
- [87] J. Schreier, *Blood, Sweat and Pixels*, HarperCollins, 2017.
- [88] M. Barton, *Honoring the Code*, CRC Press, 2016.
- [89] C. Larman, *UML y Patrones*, 2003.
- [90] R. S. Pressman, *Ingeniería del software. Un enfoque práctico*, 2010.

Ludografía

Abzu, Giant Squid Studios, 505 Games

Astral Chain, Platinum Games, Nintendo, 2019

Bayonetta, Platinum Games, Sega, 2009

Bayonetta 3, Platinum Games, Nintendo, 2022

Code Vein, Bandai Namco Studios, Bandai Namco Entertainment, 2019

Cuphead, Studio MDHR, Studio MDHR, 2017

Darksiders 3, Gunfire games, THQ Nordic, 2018

Dark Souls, From Software, Bandai Namco, 2011

Demon Souls, From Software, Sony & Atlus & Bandai Namco, 2009

Devil May Cry, Capcom, Capcom, 2001

Devil May Cry 5, Capcom, Capcom, 2019

Diablo, Blizzard North, Blizzard Entertainment, 1996

Dragon Slayer, Nihon Falcom, Nihon Falcom, 1984

Dynasty Warrior 2, Omega Force, Koei, 2000

For Honor, Ubisoft Montreal, Ubisoft, 2017

Fist of the North Star: Ken's Rage, Tecmo Koei, Tecmo Koei, 2010

GangBeast, Boneloaf, Coatsink, 2017

God of War, Santa Monica Studio, Sony & Capcom, 2004

Golden Axe, Sega, Sega, 1989

Guardian Heroes, Treasure, Sega, 1996

Hyrule Warriors, Omega Force, Koei Tecmo, 2014

Hydride, T&E Soft, T&E Soft, 1984

Human Fall Flat, No Brakes Games, Curve Games, 2016

Knights of the Round, Capcom, Capcom, 1991

Nier Automata, Platinum Games, Nintendo, 2017

Ninja Gaiden, Team Ninja, Tecmo, 2004

Nioh, Team Ninja, Sony & Koei Tecmo, 2017

Overwatch, Blizzard Entertainment, Blizzard Entertainment, 2016

Persona 5: Strikers, Omega Force, Sega, 2020

Salt and Sanctuary, Ska Studios, Ska Studios, 2016

Samurai Warriors Saga, Omega Force, Koei & Electronic Arts, 2004

Soulstice, Reply Games Studios, Modus Games, 2022

Street Fighter (saga), Capcom, Capcom, 1987 - 2023

Super Smash Bros (saga), HAL Laboratory | Bandai Namco, Nintendo, 1999 - 2018

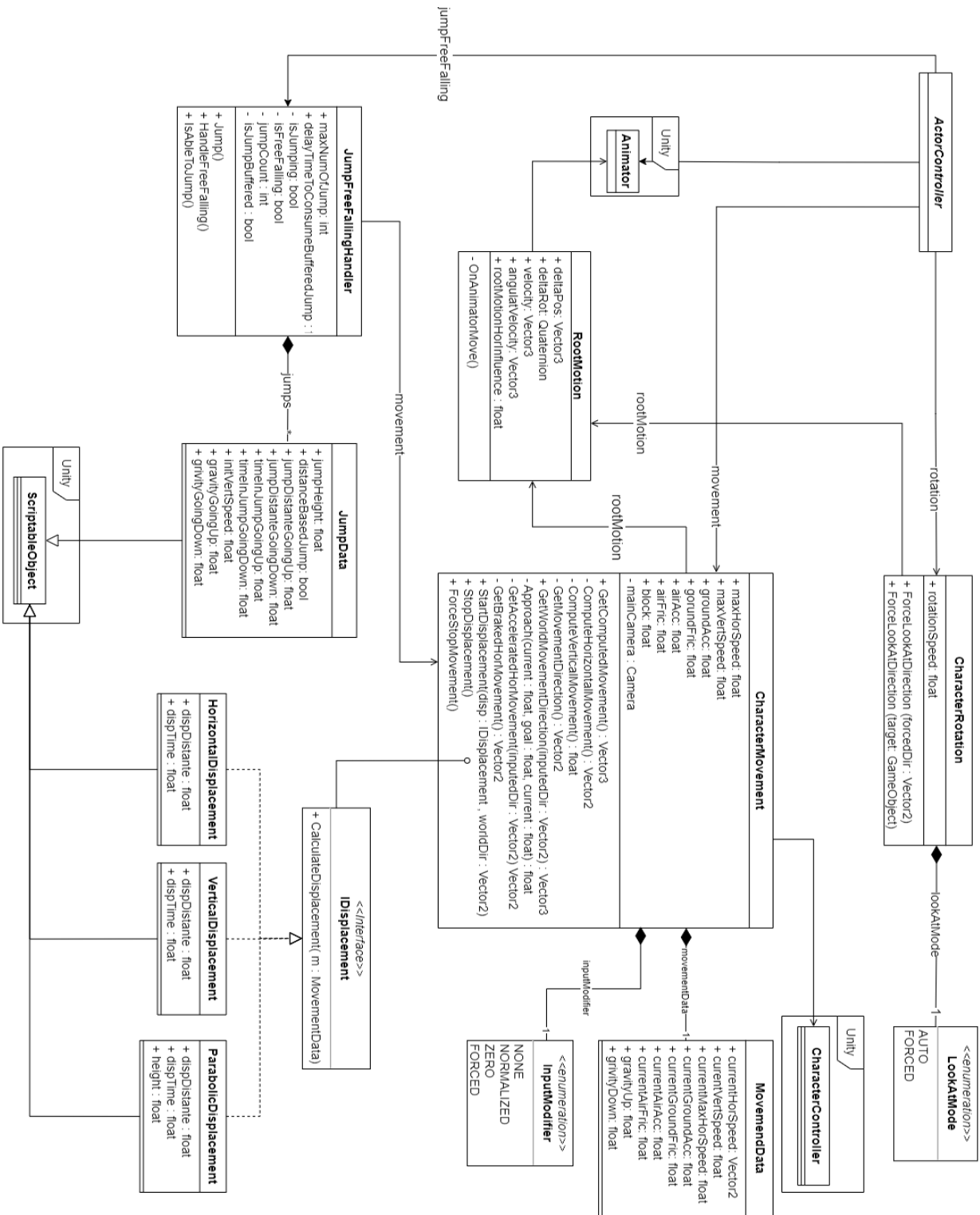
The King of Dragons, Capcom, Capcom, 1991

The Legend of Zelda, Nintendo, Nintendo, 1986

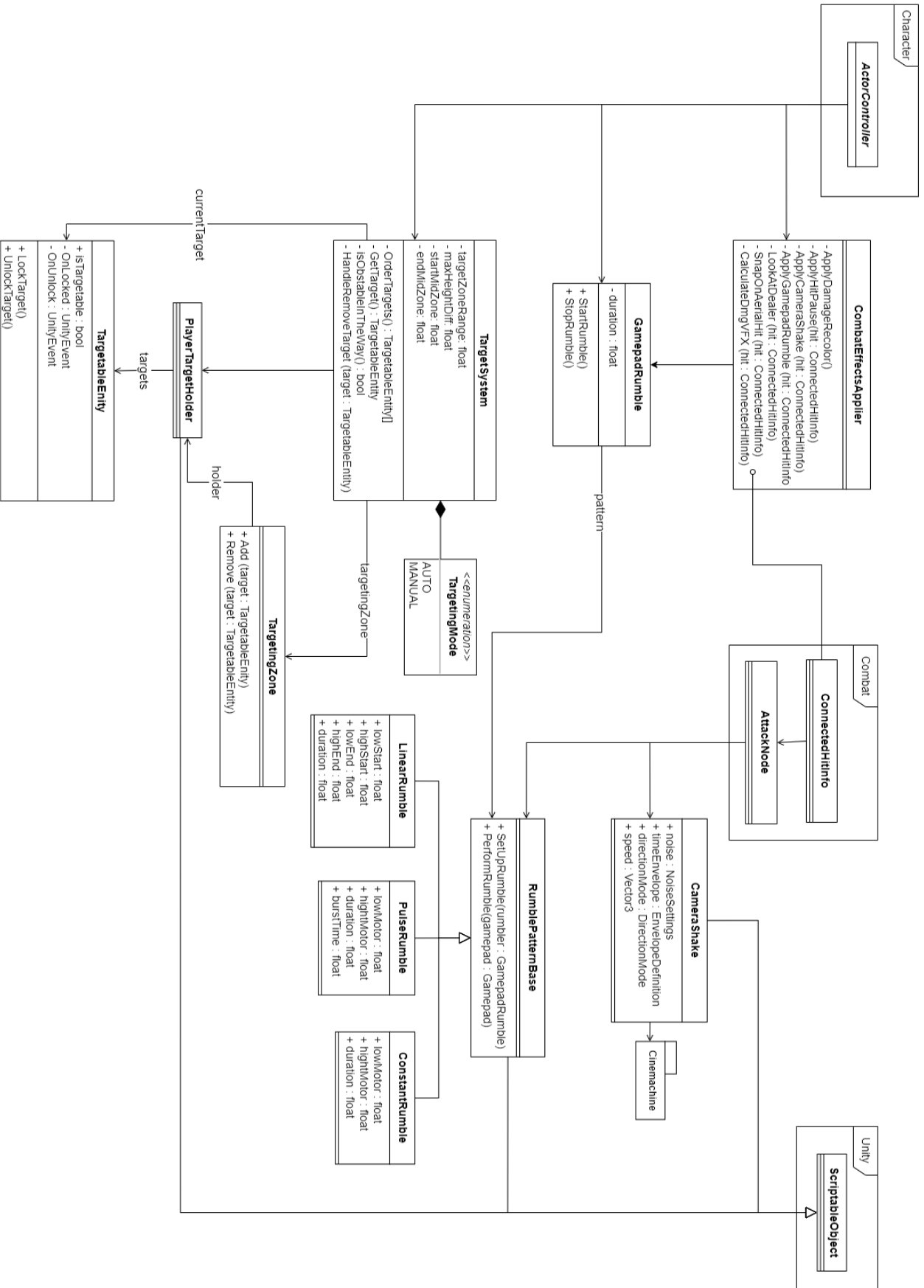
The Tower of Druaga, Namco, Namco, 1984

Diagrama de clases

Movimiento



Sistema de fijado y Gamefeel



Animation Events

EnableAnimationCancel()
Habilita la ventana de cancelación para la siguiente animación de la cadena de ataques
DisableAnimationCancel()
Deshabilita la ventana de cancelación para la siguiente animación de la cadena de ataques
EnableSecondaryAnimationCancel()
En el caso de Juno, en el que existen dos clases de combos, ligero y pesado, este evento habilita la ventana de cancelación para cambiar de un ataque ligero a uno pesado y viceversa.
DisableSecondaryAnimationCancel()
Deshabilita la ventana de cancelación para cambiar de un ataque ligero a uno pesado y viceversa.
EnableInputListening()
Habilita el <i>input buffering</i> .
DisableInputListening()
Deshabilita el <i>input buffering</i> .
EnableMoveCancel()
Habilita la ventana de cancelación para moverse.
DisableMoveCancel()
Deshabilita la ventana de cancelación para moverse.
EnableCustomCancel(cancel : string)
Habilita la ventana de cancelación para la acción indicado por el parámetro "cancel"
DisableCustomCancel(cancel : string)



Deshabilita la ventana de cancelación para la acción indicado por el parámetro “cancel”

EnablePreventDamageCancel()

Habilita que el personaje prevenga cancelar sus animaciones cuando recibe daño

DisablePreventDamageCancel()

Deshabilita que el personaje prevenga cancelar sus animaciones cuando recibe daño

EnableCanForceLookAt()

Cambia el modo de rotación del personaje a FORCED.

DisableCanForceLookAt()

Cambia el modo de rotación del personaje a AUTO.

EnableStrafing()

Habilita el *strafe*.

DisableStrafing()

Deshabilita el *strafe*.

ForceLookAtTarget()

Cambia el modo de rotación del personaje a FORCED y se asigna un objetivo al que mirar.

SnapLookAtTarget()

Cambia el modo de rotación del personaje a FORCED, se asigna un objetivo al que mirar abruptamente, sin la interpolación por defecto que lleva consigo la rotación.

StartDisplacement(disp : IDisplacement)

Se ejecuta el desplazamiento indicado por el parámetro “disp.”

StartFollowDisplacement()

Utilizado para el comportamiento de ataque de carga de los enemigos (“Carga de Valor” del soldado raso y “Carga de Escudo” del soldado blindado).

EndDisplacement()

Para la ejecución de cualquier desplazamiento que se esté llevando a cabo.



EnableComputeVerticalMovement()

Habilita el computo del movimiento vertical. Útil para el combate aéreo.

DisableComputeVerticalMovement()

Deshabilita el computo del movimiento vertical.

EnableHitbox(hbName : string)

Activa la *hitbox* indicada por el parámetro “hbName”

DisableHitbox(hbName : string)

Desactiva la *hitbox* indicada por el parámetro “hbName”

ApplyCameraShake (shake : CameraShake)

Aplica el *camera shake* indicado por el parámetro “shake”.

ApplyGamepadRumble (rumble : RumblePatternBase)

Aplica la vibración de mando indicado por el parámetro “rumble”.

PlayFVX(vfxName : string)

Reproduce el VFX indicado por el parámetro “vfxName”

PlayRotatedVFX (vfxName : string, rotation : RotationObject)

Reproduce el VFX indicado por el parámetro “vfxName” con una rotación especificada por el parámetro “rotation”. Utilizado a la hora de realizar los efectos de estela al atacar de todas las armas de los personajes.

PlayVFXOnce (vfxName : string)

Reproduce el VFX indicado por el parámetro “vfxName” solo una vez en la animación, utilizado para evitar reproducciones no deseables en animaciones que trabajan en bucle.

ApplyGameFreeze (freezeDuration : float)

Se aplica *game freeze* durante un período de tiempo indicado por el parámetro “freezeDuration”.

EnableGameFreeze()

El juego entra en *game freeze*.



DisableGameFreeze()

El juego sale del *game freeze*.

UseItem()

Ejecuta la funcionalidad del objeto seleccionado en el modo concentración.

UseAct()

Ejecuta la funcionalidad del acto seleccionado en el modo concentración.

NyxTeleport()

Ejecuta la teletransportación de Nyx.

NyxCyanShoot()

Marca cuando el alma esclava cian de Nyx debe disparar.

NyxThrowKnife()

Marca cuando Nyx lanza su cuchillo para paralizar al jugador.

NyxParentToWeaponSockets()

Marca cuando las armas esclavas de Nyx deben volver a su posición original.

RajParentWeaponToLeftHand()

Emparenta el arma de Raj a su mano izquierda.

RajParentWeaponToRightHand()

Emparenta el arma de Raj a su mano derecha.

State Machine Behaviours

SetLightAttackIndex
Al entrar en el nodo de animación, marca el índice del ataque en el combo ligero de Juno, para poder intercalar adecuadamente ataques entre los dos combos.
SetHeavyAttackIndex
Al entrar en el nodo de animación, marca el índice del ataque en el combo pesado de Juno, para poder intercalar adecuadamente ataques entre los dos combos.
StartAttackAnimationDetector
Marca el comienzo de una animación de ataque.
EndAttackAnimationDetector
Marca el final de una animación de ataque. Utilizado para detectar que el personaje ha terminado de atacar.
StartDamagedAnimationDetector
Marca el comienzo de una animación de daño.
EndDamagedAnimationDetector
Marca el final de una animación de daño. Utilizado para detectar que el personaje ha terminado de estar dañado.
StartLedgeClimbAnimationDetector
Marca el comienzo de la animación de escalar salientes.
EndLedgeClimbAnimationDetector
Marca el fin de la animación de escalar salientes. Utilizado para detectar que el jugador ha terminado de atacar.
SetAttackNode
Marca el AttackNode correspondiente al entrar en la animación.
ClearAnimatorCombatParameters



Resetea al entrar en la animación todos los parámetros del *AnimatorController* utilizados para el combate, para mantener un estado limpio para el siguiente ataque.

EnableNyxWantsToTeleport

Habilita a Nyx la posibilidad de realizar una teletransportación al entrar en la animación.

DisableNyxWantsToTeleport

Deshabilita a Nyx la posibilidad de realizar una teletransportación al salir de la animación

ImmunityAnimation

Marca al jugador como inmune al daño durante toda la animación.

NyxParentDeparentWeapons

Utilizado para desemparentar las armas esclavas de Nyx al entrar en la animación y volverlas a emparentar al salir de la misma.

PreventDamageCancelAnimation

Evita que el personaje cancele su acción al recibir daño durante toda la animación.

StopDisplacement

Para por completo cualquier desplazamiento que se esté llevando a cabo al entrar en el nodo de animación.

EnableStrafingOnExit

Habilita el *strafe* al salir del nodo de animación.

ResetRajWeaponHand

Coloca el arma de Raj en su posición original al entrar en el nodo de animación.