TREBALL FINAL DE GRAU

Implementación de un prototipo basado en la ambientación sonora dinámica

Adrián Núñez Garrido
Tutor: Dr. Enric Sesa i Nogueras

Grau en Disseny i Producció de Videojocs

CURS 2020-21

TecnoCampus
Escola Superior Politècnica

Centre adscrit a la

upf. Universitat Pompeu Fabra *Barcelona*

Abstract

This project is focused on the creation of a game prototype using *Unity* focused on different mechanics based on a dynamic sound system, previously theoretically documented within the memory of this thesis.

Resumen

Este proyecto se basa en la creación de un prototipo de videojuego utilizando *Unity* centrado en diferentes mecánicas que giren en torno a un sistema de ambientación sonora dinámica, documentada previamente durante el trabajo.

Resum

Aquest projecte es basa en la creació d'un prototip de videojoc utilitzant *Unity* centrat en diferents mecàniques que girin entorn a un sistema d'ambientació sonora dinàmica, documentada prèviament durant el treball.

Índice

1. Introducción 1

2. Marco Teórico 3

2.1. La importancia de la obra de Karen Collins 3

2.2. Funciones del Apartado Sonoro de un Videojuego 4

2.3. La Ambientación Dinámica 8

2.3.1. Tipos de ambientación dinámica 9

2.3.2. Música adaptativa 11

2.4. Variabilidad del Sonido 13

2.4.1. Variación del Volumen 14

2.4.2. Variación del Tono 14

2.4.3. Variación del Timbre 15

2.4.4. Variación del Tempo 15

2.4.5. Variación de la Melodía 15

2.4.6. Variación del Ritmo 16

2.4.7. Variación de la Armonía 16

2.4.8. Variación de la Mezcla 17

2.4.9. Variación de la Forma 17

3. Análisis de Referentes 19

3.1. “Ape Out” de Gabe Cuzzillo 19

3.2. “140” de Carlsen Games 21

4. Herramientas de sonorización dinámica 25

4.1. Imuse (LucasArts) 25

4.2. Wwise (Audiokinetic) 25

4.3. Fmod (Firelight Technologies) 27

5. Objetivos 29

5.1. Objetivos Principales 29

5.2. Objetivos Secundarios 29

6. Diseño Metodológico y Cronograma 31

6.1. Metodología 31

6.2. Cronograma 32

7. Desarrollo del prototipo 33

7.1. Diseño conceptual 33

7.2 Implementación del prototipo 39

7.2.1 Organización del estudio 39

7.2.2 Arquitectura Editor-Estudio en C# 44

7.2.3 Producción de sonidos 66

8. Conclusiones 67

9. Glosario 71

10. Referencias 73

10.1. Bibliografía 73

10.2. Ludografía 76

10.3. Sistemas y componentes 77

Índice de figuras

Fig. 2.1. Ejemplo de una de una secuencia que utiliza el método de variación de forma en Fmod. 17

Fig. 3.1. Método utilizado para lanzar eventos en conjunto con la música en *140***.**.. 21

Fig. 3.2. Ejemplo de la latencia ocurrida por el motor de sonido en *140***.**..... 22

Fig. 4.1. Interfaz del programa Wwise..... 25

Fig. 4.2. Interfaz del programa Fmod. 26

Fig. 6.1. Cronograma. 31

Fig. 7.1. Ajustes del bucle melódico en Fmod..... 38

Fig. 7.2. Ventana de eventos de Fmod Studio. 39

Fig. 7.3. Ventana del evento en Fmod studio..... 40

Fig. 7.4. Parámetros utilizados en el prototipo. 40

Fig. 7.5. Parámetros del evento Reliquia Verde en Fmod Studio..... 41

Fig. 7.6. Ventana de bancos en Fmod Studio. 41

Fig. 7.7. Ventana emergente de Fmod Studio en *Unity*. 42

Fig. 7.8. Diagrama de clases del dominio sonido, *Reliquary.Sound*. 46

Fig. 7.9. Arquitectura del patrón de diseño MVC..... 49

Fig. 7.10. Diagrama de clases del dominio del jugador, *Reliquary.Player.* 51

Fig. 7.11. Apariencia y organización del monje en la escena de juego..... 53

Fig. 7.12. Diagrama de clases del dominio de la reliquia, *Reliquary.Relic.* 54

Fig. 7.13. Apariencia y organización del monasterio en el editor. 55

Fig. 7.14. Diagrama de clases del dominio del monasterio, *Reliquiary.Hub*. 56

Fig. 7.15. Apariencia del monasterio en el editor. 57

Fig. 7.16. Diagrama de clases del dominio del acechador, *Reliquary.Stalker*..... 58

Fig. 7.17. Apariencia del *Navigation Mesh* de la escena de juego. 60

Fig. 7.18. Apariencia y organización del acechador en el editor. 61

Fig. 7.19. Diagrama de clases del dominio del nivel, *Reliquary.Level.*..... 62

Fig. 7.20. Apariencia y organización del nivel en la escena de Unity. 63

Fig. 7.21. Ventana de edición de la sesión en *ProTools 12.* 65

Índice de tablas

Tab. 2.1. Ejemplos de sonidos dinámicos según la clasificación de Karen Collins…..10

Tab. 7.1. Clasificación de los elementos sonoros del prototipo.....38

1. Introducción

El sonido ha sido una parte crucial que ha formado parte de los videojuegos desde la comercialización de los primeros sistemas recreativos domésticos, mejorando en gran medida la experiencia e inmersión de los usuarios. La evolución tecnológica del medio ha permitido la creación de paisajes sonoros cada vez más complejos, partiendo de un número de efectos limitados y muy sencillos, hasta llegar a las majestuosas bandas sonoras actuales compuestas por decenas de piezas musicales interpretadas por cientos de instrumentos de forma simultánea.

Paralelamente, el videojuego ha sido popularizado globalmente, convirtiéndose para muchos en una de las principales fuentes de entretenimiento del día a día. Tal como menciona Collins (2007), esto ha traído consigo reconocimiento e interés por los autores y técnicas detrás de los diversos departamentos de desarrollo y, con ello, nuevas oportunidades formativas. Pero, aún y así, el terreno académico sigue siendo escaso, y ofrece pocos recursos rigurosos respecto a la teoría relacionada con el diseño de elementos sonoros. En concreto, aquellos que muestren las oportunidades creativas que puede traer el sonido a la hora de crear una experiencia interactiva.

Así mismo, el desarrollo de videojuegos trae retos de sonorización únicos y diferentes de otros medios, que implican la necesidad de poder adaptarse a las decisiones de los usuarios. Esto se debe mayoritariamente a la naturaleza interactiva del mismo, que ha llevado a los desarrolladores a la creación de un conjunto de procesos agrupados bajo el término “ambientación dinámica”. Estos procesos no se limitan solo a la producción de dichos elementos sonoros, sino que también incluyen en gran medida su implementación técnica dentro del proyecto a través de múltiples herramientas y sistemas.

El propósito de esta memoria es mostrar el proceso a seguir en la implementación técnica de la ambientación dinámica, dentro de un prototipo de videojuego basado en este concepto. Este prototipo contiene varias mecánicas que guardan estrecha relación con diversos elementos sonoros dinámicos, dando a conocer así un procedimiento técnico y teórico a seguir, y enseñar el potencial expresivo del sonido en el medio del videojuego. Para ello se utilizan *Unity* y *Fmod* como herramientas principales.

2. Marco Teórico

Esta sección consiste en la exposición de la terminología y tipología empleada para describir la ambientación sonora y sus funcionalidades en un espacio interactivo, clasificar los diferentes tipos de sonido y, finalmente, exponer los sonidos que forman parte de la ambientación dinámica y como diferenciarlos.

2.1. La importancia de la obra de Karen Collins

Antes de exponer el marco teórico, cabe mencionar la importancia de la autora Karen Collins y de su libro publicado en 2007 *"Game Sound: An introduction to the History, Theory and practice of Videogame Music and Sound Design"* (Collins, 2007).

Este ofrece una visión en profundidad del ámbito sonoro del mundo de los videojuegos, desde la historia desde sus inicios hasta la actualidad como sus diferentes aplicaciones en la jugabilidad. Durante la memoria, se utilizan fuentes, definiciones y clasificaciones empleadas por Collins, con el objetivo de poder asentar una base teórica que aclare términos que serán utilizados durante el desarrollo del prototipo.

Gran parte de la información se extrae de los últimos dos apartados del libro, *"Chapter 7: Gameplay, Genre, and the Functions of Game Audio"* y *"Chapter 8: Compositional Approaches to Dynamic Game Music*", pero también se hace referencia a otros apartados, como podrían ser el Capítulo 3 o la introducción.

2.2. Funciones del Apartado Sonoro de un Videojuego

Tal y como afirma Scott Genshin (Klein, Genshin, Bush, Boyd, Shah, 2007) “Una de las cosas que diferencia a los videojuegos de cualquier otro medio es la capacidad de otorgar satisfacción directa a los usuarios”.

El videojuego es un medio capaz de sumergir a los usuarios en la fantasía a la que están siendo expuestos gracias a la capacidad de interacción directa que este proporciona. Dentro de esta fantasía, la capacidad auditiva tiene el potencial de reforzar en los jugadores la experiencia que se les ofrece, ya que se trata de uno de los sentidos humanos esenciales de la percepción del entorno.

Incluso con todo su potencial, el sonido suele ser una de las partes más ignoradas a la hora de valorar un videojuego. Esto lo lleva a ser relegado normalmente en críticas como el eslabón menos importante y comentado. Tracy Bush (Klein et al., 2007) menciona “Si la música apesta, el consumidor apagará el sonido y punto”. En caso de que el sonido haga su trabajo, será ignorado, y en caso contrario, muchos jugadores se limitarán a ignorarlo o quitarlo.

A pesar de esto, profesionales como Tracy Bush (Klein et al., 2007) defienden la importancia del audio, con ejemplos como son un experimento donde una escena es expuesta con diferentes piezas musicales de fondo, las cuales cambian totalmente el contexto y sentimiento transmitido al público. “Algo que no contienen los apartados visuales es emoción. Si pones justo el sonido que necesita, mejorará completamente la experiencia final” (Klein et al., 2007). Además, estudios como el de Raymond Usher (Usher, 2012), el cual diseña un experimento donde doce jugadores distintos interactúan con escenas de varios juegos, primero sin y luego con audio, muestran como aquellos que juegan con sonido tienen un pulso y ritmo respiratorio más alto, lo cual demuestra que el sonido agregado al juego mejora la experiencia a nivel emocional.

Siguiendo esta filosofía de que el audio puede mejorar la experiencia de los jugadores, Karen Collins recopila una serie de funciones que el sonido puede emplear dentro de un videojuego para guiar o mejorar la experiencia de los jugadores:

- Creación de atmósfera, ritmo y contexto narrativo: Un uso muy común cuyo uso viene ya de otros medios como el cine, donde, ya sea con una pieza musical o con un efecto sonoro que añada impacto, se refuerza el mensaje global que el creador quiere transmitir a través del sonido, que ayuda a evocar la emoción deseada al jugador, ya sea frenetismo en una escena de acción o nostalgia en un lugar abandonado. Esto se puede ver en casos como el anteriormente mencionado por Tracy Bush (Klein et al., 2007), donde diferentes tipos de música cambiaban la sensación final que transmitía la escena, o en casos como el expuesto por Elliot Callighan (Callighan, 2019) que muestra la escena introductoria del juego *Bioshock* (2K Boston, 2007), primero sin ambientación musical y viceversa, exponiendo así como adquiere una ambientación siniestra y extraña que en completo silencio se pierde completamente.

- Inducir inmersión ambiental al jugador: Otro uso muy común fuera del medio de los videojuegos, pero muy necesario en juegos con navegación en tres dimensiones. Tal y como explica Elliot Callighan (Callighan, 2019), consiste en la definición del espacio que rodea al jugador mediante el sonido con el objetivo de hacerle creer que el espacio que le rodea es real.

- Percepción acústica: Consiste en la importancia de enfatizar puntos de atención que rodean al jugador con el objetivo de que éste dirija su atención hacia ellos, aunque su origen visual no quede claro. Tal y como lo describe (Chion, 1994), el sonido nos sirve para incitar al jugador a ir ahí y averiguar qué es lo que hay. Este uso se suele fomentar principalmente en juegos con espacios tridimensionales, donde la profundidad y navegación giran entorno todos los ángulos del usuario, y necesita la máxima información posible para hacer su experiencia justa y disfrutable.

- Guía espacial: Consiste en la decisión de utilizar el sonido como opción de diseño para guiar las decisiones del jugador en un espacio no lineal, por ejemplo, mediante el acuñado por Karen Collins como "*Boredom Switch*", que consiste en hacer desaparecer la música lentamente con tal de transmitir al jugador que tarda demasiado en finalizar una sección y debe avanzar, además de impedir que la música se haga demasiado repetitiva y rompa la inmersión de la escena. Tal como lo menciona Anabelle Cohen (Cohen, 1999, pp. 41, 68) "Una pausa o silencio en la música puede indicar un cambio de narrativa". Otras formas de utilizar el audio para guiar decisiones del jugador que Collins menciona sería el uso del diálogo para dar pistas a los jugadores o usar una canción para crear la transición entre espacios y mantener al jugador alerta.

- "*Leitmotivs*": Los llamados "*leitmotivs*" son temas, ya sean melodías o pequeñas piezas musicales, relacionados a personajes, espacios o conceptos que añadan una emoción a la escena que el espectador pueda reconocer fácilmente y construyan profundidad adicional al personaje. Citando a la Real Academia Española, un "*leitmotiv*" es "un tema musical dominante y recurrente en una composición" o "un motivo central o asunto que se repite". Un claro ejemplo sería el señalado por Jason Yu (Yu, 2016) sobre el juego *Undertale* (Toby Fox, 2015), el cual ejerce un amplio uso de este recurso en numerosos temas para caracterizar personajes y describir acciones del jugador.

- Suspensión de la incredulidad: Esta función acompaña a otras mencionadas anteriormente, como la inducción atmosférica o la creación de contexto narrativo. Consiste en la inducción psicológica del jugador en la fantasía en la que se encuentra a través del sonido reproducido, ya sea mediante sonidos que definen el espacio a nivel de profundidad o música integrada físicamente en el juego.

- Eliminar sonidos molestos: Una función menor normalmente ignorada por los usuarios, es la capacidad del sonido de eliminar ruidos del ambiente que saquen al jugador de la ilusión del videojuego. “En dispositivos domésticos, la música podría enmascarar distracciones del ventilador u otros sonidos emitidos por el ambiente circundante” (Cohen, 1999, p. 41).

- Crear estructuras de juego: Karen Collins (2007) menciona una rara función en la que el propio audio define la estructura del juego de forma directa, como por ejemplo el juego *Vib Ribbon* (NanaOn-Sha, 1999) o el juego de PC *AudioSurf* (Dylan Fitterer, 2008), donde el usuario puede elegir su propia música y el juego crea circuitos basándose en su ritmo, entre otros aspectos.

En conclusión, queda demostrado el papel y capacidad del audio en un videojuego a través de las múltiples opciones expuestas, y la variedad, riqueza y profundidad que estas ofrecen a usuarios y diseñadores por igual.

2.3. La Ambientación Dinámica

Una vez definida la importancia del sonido en el medio, hay que aclarar el significado detrás de la denominada como ambientación dinámica, que la caracteriza y cómo se identifica.

En primer lugar, ¿qué define a algo dinámico? La Real Academia Española contiene diferentes definiciones para este concepto, siendo la primera: “Perteneciente o relativo a la fuerza cuando produce movimiento”. Es decir, aquello relacionado con el movimiento de forma directa o indirecta. También cabe destacar la definición relacionada con el campo físico, “Rama de la mecánica que trata de las leyes del movimiento en relación con las fuerzas que lo producen”, pero por conveniencia se elegirá la definición anterior.

A partir de esta definición, el sonido dinámico, o ambientación dinámica, sería aquel que gira en torno al movimiento realizado en una escena. En un videojuego, pero, esta definición se ve alterada por la naturaleza de las acciones que lo componen, ya que este será compuesto por acciones causadas tanto por el jugador como acciones automáticas obligatorias. Karen Collins (2007) aclara esta diferencia proponiendo que el sonido dinámico sea aquel que es causado por cambios durante la interacción directa o indirecta del usuario con el juego.

Esta definición es la que se utiliza a lo largo de esta memoria. Además, a partir de esta definición, Karen Collins (2007) propone también diferenciar el sonido según el grado de interacción que ha tenido en su reproducción y el grado de diégesis, o grado de integración narrativa del sonido en el mundo con tal de conseguir una clasificación definida y ordenada con la que poder trabajar.

2.3.1. Tipos de ambientación dinámica

Siguiendo la definición de sonido dinámico expuesta por Karen Collins (2007), se divide el apartado sonoro en dos tipos, según el grado de interacción del jugador respecto a la acción que causa la reproducción de dicho sonido:

- **Sonidos adaptativos:** Aquellos efectos sonoros que responden activamente a la situación del jugador dentro del mundo y sistema de juego, y su evolución dentro de este.
- **Sonidos interactivos:** Aquellos efectos sonoros que responden a acciones directas realizadas por el jugador y su personaje, ya sea a través de la pulsación de un botón o el movimiento del personaje.

Al mismo tiempo. los sonidos quedarían divididos por la ya mencionada noción de diégesis, que divide elementos narrativos según su integración con el mundo presentado, en dos grupos:

- **Sonidos diegéticos:** Aquellos que están integrados como elementos narrativos y son reproducidos dentro del contexto de este, e incluso a veces son comentados por sus integrantes.
- **Sonidos no diegéticos:** Aquellos sonidos que, ya sean parte de una acción o se añadan por cuestiones de ritmo para esta, como podría ser una canción, no forman parte del contexto presentado a los jugadores y, por lo tanto, los personajes que integran dicho mundo no pueden escucharlos.

Con tal de ejemplificar el uso de esta clasificación, a continuación, se muestra una tabla que clasifica diferentes sonidos del *juego The Legend of Zelda: Ocarina of Time* (1998), extraídos del libro de Karen Collins (2007).

	Diegéticos	**No diegéticos**
Adaptativos	Sonidos de animales (como el aullido de un lobo al anochecer o el canto de un gallo al amanecer) que suenan cuando el juego cambia el estado de la hora del día.	Música ambiental que se reproduce mientras el jugador navega el mundo y cambia su ritmo cuando el juego cambia de día a noche y viceversa.
Interactivos	Efecto de sonido reproducido cuando Link golpea con la espada.	Sonido que el juego reproduce cuando el jugador lo pausa al abrir el menú.

Tab. 2.1. Ejemplos de sonidos dinámicos según la clasificación de Karen Collins.

Fuente: Elaboración propia

Karen Collins (2007) señala también un tipo de sonorización a acciones especiales acuñados con el término de Interacción Gestural Kinetica, basada en la producción de sonidos para acciones donde el jugador participa mimetizando el movimiento del personaje, obteniendo un nivel de interacción extra por parte del jugador. Un ejemplo sería el uso del controlador de la consola Nintendo Wii (Nintendo, 2006) para golpear con una espada.

2.3.2. Música adaptativa

Dentro del apartado sonoro dinámico de un videojuego, cabe destacar la importancia de la composición musical. Este suele ser el más complejo y señalado y, paralelamente, el más importante para los jugadores dentro del apartado sonoro. La música dinámica puede situarse también dentro de la clasificación descrita por Karen Collins (2007), normalmente dentro del apartado adaptativo, ya que responde al estado directo del jugador, pero no las acciones que realiza.

Cabe destacar las diferencias que tiene una composición para el escenario de un videojuego debido a que, tal y como menciona Karen Collins (2007), normalmente se trata con casos donde la música ha de ser compuesta y dispuesta de forma no lineal, con tal de poder adaptarse al estado del jugador, estando en bucle constante o cambiando cuando sea preciso. “La música dinámica se está volviendo cada vez una necesidad mayor, debido al aumento de los valores de producción y al agotamiento de los jugadores de la habitual música en bucle de juegos más antiguos” (Karen Colllins, 2007, p. 139).

Esto genera numerosas dificultades para tanto compositores como diseñadores, que deben crear y disponer de canciones que se repitan de forma cíclica sin fatigar mentalmente al jugador y lo mantengan en el ritmo deseado constantemente y le instiguen a seguir avanzando hacia sus objetivos.

La música dinámica, llamada también como música adaptativa, puede componerse de dos formas distintas.

- **Secuenciación Horizontal:** Se basa en la composición de una pieza con diferentes secciones, o piezas musicales totalmente distintas, entre las que el sistema cambiará según el escenario en el que se encuentra el jugador. Jake Butineau (Butineau, 2017) ejemplifica este caso en el juego *Furi* (The Game Bakers, 2016), donde el jugador combatirá contra un enemigo con diferentes fases y el sistema cambiará de una sección musical a otra cada vez que lo haga.

- **Orquestación Vertical:** Se basa en la composición de una pieza musical en diferentes capas instrumentales que se adaptan al estado del personaje y el jugador de formas distintas, apareciendo y desapareciendo según las acciones realizadas. Jake Butineau (Butineau, 2017) muestra su uso en el juego Portal 2 (Valve, 2011), donde una base musical estándar de un puzle tiene diferentes capas instrumentales relacionadas con elementos del nivel que aparecen y desaparecen según el jugador se mueve alrededor de dichos elementos, o realiza acciones como saltar largas distancias o lanzar un objeto.

2.4. Variabilidad del Sonido

En los apartados anteriores, se ha definido que es y que define al sonido para que este sea dinámico. También se ha podido establecer una clasificación que permita diferenciar los elementos según su interactividad y su cabida en el contexto de la obra. Pero, a la hora de producir e implementar la ambientación sonora en un proyecto, ¿qué atributos definen la flexibilidad de sus elementos sonoros y, en consecuencia, ¿cómo se pueden modificar para adaptarse dinámicamente a las acciones del jugador?

Karen Collins define diversas técnicas o acercamientos a la hora de añadir variabilidad a los elementos sonoros, las cuales se enfocan principalmente a piezas musicales. Esto se debe a que, como se ha comentado en apartados anteriores, son los elementos sonoros que requieren más riqueza debido a que los jugadores escuchan constantemente e influyen en gran medida la ambientación y contexto emocional de la escena. Aun así, también pueden aplicarse en los efectos sonoros para añadir variedad a las acciones del jugador, o para reutilizar efectos sonoros de forma sutil, entre otros usos.

En los siguientes apartados se definirá brevemente en primer lugar el concepto teórico sonoro con el que se trata y, a continuación, en qué consiste la técnica que se aplica sobre este con tal de obtener el efecto modificado.

2.4.1. Variación del Volumen

Según la Real Academia Española, volumen es el término comúnmente utilizado para referirse a la intensidad del sonido.

La variación de este es el recurso más comúnmente utilizado en de la lista elaborada por Karen Collins (2007), en concreto sobre la música. Algunos de los usos comunes de este recurso que se describen, es la de crear una transición sonora entre escenas, la desaparición de música en escena para indicar al jugador que ha de seguir progresando, o el empleo de múltiples intensidades en función de la tensión en escena creada por las acciones del jugador.

No obstante, también se menciona el uso de la variación de intensidad sobre efecto sonoros para indicar distancia, modificando el volumen de los elementos en función de la distancia entre el avatar y las fuentes de sonido.

2.4.2. Variación del Tono

Según la Real Academia Española, el tono es la cualidad sonora que describe la posición de un sonido en función de su frecuencia. Tal como dice Ebenezer Prout (Prout 1889, p. 24), el tono de una nota depende de la rapidez de su vibración, es decir su frecuencia, siendo las notas graves aquellas con vibraciones más lentas y viceversa.

Karen Collins (2007) menciona que, en juegos de dispositivos antiguos, como *Sonic The Hedgehog* o *Rally X*, se aplica una transposición de tono se utiliza con el objetivo de ahorrar memoria digital. Esto se realiza sobre diversas secuencias musicales, las cuales se guardan a una frecuencia inferior, obteniendo así un archivo reducido, y estiradas durante la ejecución mediante código. Al igual que la intensidad, Karen Collins (2007) menciona que también se puede variar el tono con el objetivo de intensificar una emoción en el jugador. Por ejemplo, durante la lucha contra el jefe final en *The Legend Of Zelda: Twilight Princess* (Nintendo, 2006) el tono se incrementa cada vez que este recibe un golpe, incrementando la tensión a medida que progresa la pelea.

2.4.3. Variación del Timbre

Según la Real Academia Española, el timbre es la cualidad del sonido dada por el efecto perceptivo de los oyentes, la cual permite distinguir entre diferentes fuentes sonoras.

Karen Collins (2007) se refiere al cambio de timbre como el uso de sistemas tipo *DSP* (*Digital Signal Processor*) sobre los archivos de audio con el objetivo de añadir efectos en tiempo real cómo, por ejemplo, la reverberación. Estos son ampliamente utilizados con el objetivo de mejorar la ambientación del escenario y se incluyen tanto sobre efectos sonoros como pistas musicales.

2.4.4. Variación del Tempo

Según la Real Academia Española el tempo es la celeridad de ejecución de una composición musical. Esta se mide en *bpm* o *beats per minute*.

Según Karen Collins (2007) los ajustes de velocidad han sido muy comunes, sobre todo a la hora de influir en la tensión de la escena, desde los inicios de la industria. Un claro ejemplo es el uso de la aceleración del tempo en *Space Invaders*, donde la música acelera a medida que la velocidad de bajada de los enemigos aumenta.

2.4.5. Variación de la Melodía

Ebenezer Prout (Prout, 1889, p. 13) describe la melodía como la sucesión de sonidos con diferente tono una uno tras otro. Ya se ha mencionado en apartados anteriores el uso de las melodías para crear relaciones con personajes o lugares, reforzando así su identidad hacia el jugador.

Karen Collins (2007) describe la variación de melodías como el uso de algoritmos para la creación de secuencias instrumentales que se adapten de forma activa a las acciones y estados del jugador. Un claro ejemplo de su uso se encuentra en el videojuego Ape Out (2019) donde la banda sonora es creada algorítmicamente mediante el uso de una batería digital separada por capas. Esta responde al estado de peligro en el que se encuentra el jugador, tocando una melodía con un kit de batería más suave cuando se encuentra oculto y viceversa, y a la posición de la pantalla donde transcurre la acción, tocando una serie de instrumentos concretos del kit de batería.

2.4.6. Variación del Ritmo

La Real Academia Española define el ritmo como el orden acompasado en la sucesión de las cosas. Es decir, en música el ritmo es la métrica o patrón que forma una melodía o varias de partes de esta.

Altman (Altman, 2004) menciona como este recurso puede utilizarse para enfatizar momentos de anticipación, ya sea de forma placentera, ansiosa o aprensiva. Karen Collins (2007) indica que el uso de instrumentos digitales o MIDI (Musical Instrument Digital Interface) puede facilitar la alteración del patrón rítmico en la que se reproduce una melodía, la cual se produce en tiempo real durante la ejecución del juego.

2.4.7. Variación de la Armonía

Karen Collins (2007) indica que la alteración de los elementos que componen la armonía, sus acordes y su tonalidad, puede alterar la ambientación de una pieza en tiempo real. Ebenezer Prout (Prout, 1889, 13-23) explica como un acorde es el resultado de escoger una serie de tonos, separados por un intervalo o distancia concreta y basándose en una tonalidad, o tono de referencia que todos los tonos escogidos comparten para así sonar de forma resolutiva. Ebenezer Prout explica como la disposición de estos tonos según los intervalos que los separen crean como resultado acordes mayores o menores.

Un ejemplo del uso de la variación armónica puede verse en el videojuego *Asheron's Call 2* (Turbine Entertainment Software 2002), donde Jason Booth (Booth, 2004, p. 480) describe como la tonalidad de una pieza musical cambia según el estado de juego para así avisar al jugador del peligro que le rodea.

2.4.8. Variación de la Mezcla

Karen Collins (2007) define la variación de la mezcla como el proceso de estructurar una secuencia musical por capas para luego ser importado para su modificación individual durante la ejecución del juego.

Un claro ejemplo del uso de esta técnica es en el videojuego *140* (Carlsen Games, 2013), donde cada instrumento es representado por un tipo de obstáculo del nivel y aumenta o disminuye su intensidad en mezcla en función del estado y la distancia del avatar.

2.4.9. Variación de la Forma

Karen Collins (2007) describe la forma de una pieza musical como la estructura mediante la cual se estructuran diferentes secciones de una pieza musical. Estas se separan por intensidad, melodía o instrumentos utilizados durante su ejecución. En otros medios, como el cine, Esta estructura tiene una duración y ambientación definida. En cambio, en videojuegos ha de poder adaptarse al progreso y estado de juego en tiempo real, independientemente de la duración de la sección, con tal de poder integrar al jugador y guiarlo en la fantasía que se encuentra viviendo. Esta técnica de variación está estrechamente con el método anteriormente expuesto como orquestación vertical.

Actualmente el método de variación secuencial o variación de forma, más utilizado es la denominada ramificación por parámetros. Esta se basa en la transición entre secciones de la secuencia controlada mediante una serie de *"triggers"* o puntos de decisión. Cuando la pista musical se ejecuta y su tiempo de reproducción llega donde se ha dispuesto uno de estos *"triggers"*, se analizarán una serie de atributos. Si se cumplen, se producirá una transición que llevará a la siguiente sección de la secuencia. En caso contrario, se seguirá reproduciendo la misma sección hasta que llegue al siguiente punto de decisión o tenga que volver a reproducirse la misma parte.

Un claro ejemplo de esto ocurre en el juego *Furi* (The Game Bakers, 2016) donde la música cambiará entre secciones dependiendo de la fase del combate en la que se encuentre el jugador.

En la siguiente imagen se muestra un ejemplo presentado por Jake Buttineau (Butineau, 2017), en el que aplica el método de variación de forma basada en ramificación por parámetros para demostrar el proceso de orquestación vertical. En esta puede observarse cómo se divide una pieza de *Furi* (The Game Bakers, 2016) en varias secuencias. Una vez dividida, cada secuencia es ajustada para poder reproducirse en bucle por separado. Además, en cada secuencia se ha dispuesto una serie de puntos de control, donde se comprobará el parámetro *"Stage"*, que responde al estado en el que se encuentra la pelea entre el jugador y un jefe final. Dependiendo del estado, una secuencia u otra serán reproducidas y debido a que los puntos de control se han controlado estratégicamente para coincidir con el ritmo de la canción, la transición parecerá natural para el jugador.

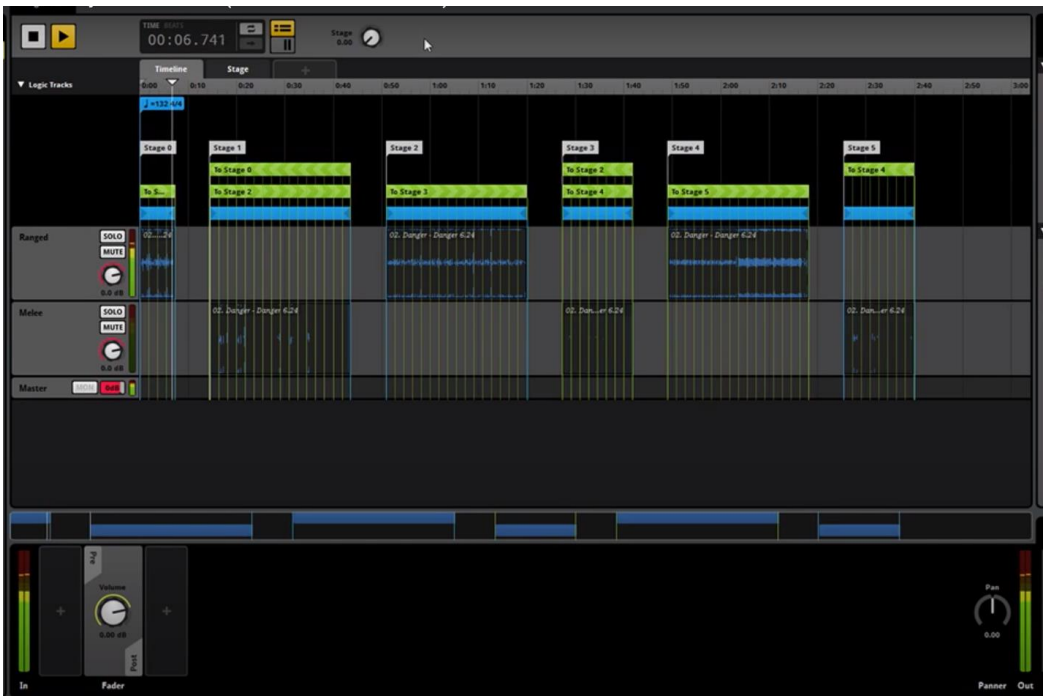

Fig. 2.1. Ejemplo de una de una secuencia que utiliza el método de variación de forma en Fmod.

Fuente: (Jake Butineau, "What is Vertical and Dynamic Music")

3. Análisis de Referentes

Dos obras resultan un gran referente para la inspiración del prototipo. Tanto *Ape Out* (Gabe Cuzillo, 2019) como *140* (Carlsen Games, 2013) hacen uso vital de la ambientación y música dinámica, en torno a los cuales giran tanto sus mecánicas como su temática, mientras mantienen una estética visual y entorno de juego simple y directo hacia el jugador. Estas obras también nos muestran diferentes retos y decisiones creativas que los desarrolladores toman para poder implementar apartados sonoros que requieren una experiencia de juego con interacción sonora.

3.1. “Ape Out” de Gabe Cuzzillo

Ape Out (Gabe Cuzzillo, 2019) resulta uno de los referentes actuales en el uso de la música dinámica. La obra ideada por el programador y diseñador Gabe Cuzzillo tiene un acercamiento sencillo al género *twin-stick shooter*. En este, el jugador controla un gorila que tendrá que recorrer varios niveles para poder escapar sin ser abatido por cazadores utilizando como única arma su habilidad para agarrar y pegar a sus enemigos y el entorno para zafarse de estos.

Gabe Cuzzillo diseña *Ape Out* en torno a la pieza de free-jazz *“You’ve Got to Have Freedom”* (Sanders, 1987) del saxofonista Pharoah Sanders, la cual le marca profundamente en un mal momento de su vida, durante sus estudios en la escuela de cine en Nueva York (Webster, 2019). Esto se refleja en decisiones como es la generación procedural de los niveles y enemigos, que evoca en el jugador la sensación de improvisación constante característica del género musical, pero esencialmente se percibe en el diseño sonoro del proyecto, realizado por el desarrollador y músico Matt Boch.

Tal como explica el propio Boch (Webster, 2019), el objetivo de la música en *Ape Out* es crear una experiencia interactiva a nivel sonoro, a través de un apartado sonoro que preste atención al jugador. Con este objetivo, Boch experimenta con diferentes acercamientos durante el desarrollo del proyecto con el objetivo de crear una banda sonora de percusión que suena natural y da control de la intensidad al jugador sin repetirse (Betts, 2019).

Tal como nos describe el propio Boch (Webster, 2019), en la versión inicial de *Ape Out*, el apartado musical consiste en tres solos de batería en bucle. Adicionalmente, un efecto sonoro de platillo suena cada vez que se produce una muerte. Con esto, Gabe Cuzzillo busca crear una banda sonora interactiva con una esencia de percusión.

En su primer acercamiento, Matt Boch trata de crear una banda sonora compuesta de una gran librería de bucles de percusión. Para crear la interactividad musical, un sistema de juego reactivo intercambia los bucles entre sí. Pero, debido al alto consumo de los archivos de sonido en el tamaño del juego y los tiempos de carga requeridos para crear una banda sonora que no suene repetitiva, Boch crea un sistema que genera la banda sonora de manera procedural.

Para ello Boch crea un *sampler* que simula un kit de batería digital. Este reproduce sonidos siguiendo una serie de patrones guardados utilizando el estándar *MIDI*, como si de un músico se tratara. El *sampler* cambia entre patrones utilizando un algoritmo de macro control, que varía entre estos en función de parámetros como son la intensidad, densidad y similaridad. Cada patrón conoce los patrones con parámetros más parecidos a él, lo que permite al *sampler* cambiar entre estos de manera reactiva y sin repetirse siguiendo el progreso del jugador.

Un problema afrontado constantemente a la hora de implementar un instrumento de percusión virtual es la presencia de sonidos repetitivos que hacen que la música suene artificial. Para evitar esto, Boch crea varios kits de batería con sonidos afines a cada nivel del juego. El *sampler* hace sonar patrones utilizando un kit u otro dependiendo del nivel en el que se encuentra el jugador. También añade variación a cada sonido reproducido mediante el uso de distintos niveles de velocidad.

Para obtener los patrones MIDI que el *sampler* interpreta, Matt Boch empieza componiendo cada pieza por separado. Pero, para obtener la cantidad de variación buscada, decide utilizar un sistema de aprendizaje automático apodado *Variational Autoencoder Neural network*. Este genera los patrones y sus parámetros utilizando los intentos del jugador y agiliza el proceso de implementación.

Además, el sonido que suena al aplastar un enemigo es integrado en el *sampler*. En vez del platillo inicial, el *sampler* reproduce un sonido u otro de la batería dependiendo de cómo se vea reflejada la acción del jugador en la pantalla respecto a la disposición de una batería acústica real.

Con estos ingredientes, Matt Boch consigue crear una banda sonora procedural en la que un baterista simulado improvisa al son de la experiencia del jugador. Además, a nivel técnico, el acercamiento procedural muestra ser más eficiente que el inicial y obtiene la enriquecida variación que busca Gabe Cuzzillo en su experiencia.

3.2. “140” de Carlsen Games

140 (Carlsen Games, 2013) es la primera obra creada por Carlsen Games, un pequeño estudio indie fundado por Jeppe Carlsen, el ex diseñador del reconocido estudio *Playdead*. Su objetivo era crear juegos que experimentaran con diferentes conceptos, entre ellos la creación de paisajes atmosféricos a través del sonido. 140 es un juego de plataformas en dos dimensiones donde el jugador deberá atravesar varios niveles repletos de obstáculos. Los niveles tienen distintas mecánicas estrechamente relacionadas con el ritmo de una pista musical, característica de cada nivel.

140 muestra un uso único del sonido, donde el nivel es un puzle musical que el jugador ha de resolver cada una de sus piezas instrumentales por separado (Drew, 2013). También es un ejemplo de cómo el apartado musical puede llegar a marcar la toma de decisiones del proyecto, incluyendo su arte. Tal como menciona Niels First, artista de Carlsen Games (Nutt, 2013), el apartado visual fue creado con la intención de realzar el papel del sonido en el juego, siendo este limpio y directo. Según el diseñador Jeppe Carlsen, la decisión de cambiar el ritmo de los temas del juego de 120 *beats* por minuto a 140 rompió todas sus decisiones de diseño anteriores, pero generó una sensación de juego más energética e incluso dio título al proyecto.

En su exposición para el Sonic College, Jakob Schmid (Schmid, 2018) describe los retos de diseño afrontados. Debido a la relación de las mecánicas con el apartado sonoro y musical, Carlsen Games implementa un controlador que utilice el tiempo del bucle musical para controlar los elementos del juego.

Para ello, Schmid explica cómo se utiliza el tiempo transcurrido y los *bpm* o pulsaciones por minuto de la canción para extraer la nota actual de la canción. En primer lugar, se calcula el número de notas totales en un minuto utilizando el número de notas por pulsación y las pulsaciones por minuto. Dividiendo este por el número de segundos por minuto se obtiene el número de segundos por nota.

El controlador contiene una serie de eventos y la nota en la cual cada evento ha de ser lanzado. A medida que el bucle avance el controlador extrae la nota actual comparando el tiempo de juego transcurrido con el número de segundos por nota. Al cambiar esta, el controlador envía una señal a los elementos de juego asignados a esa nota concreta.

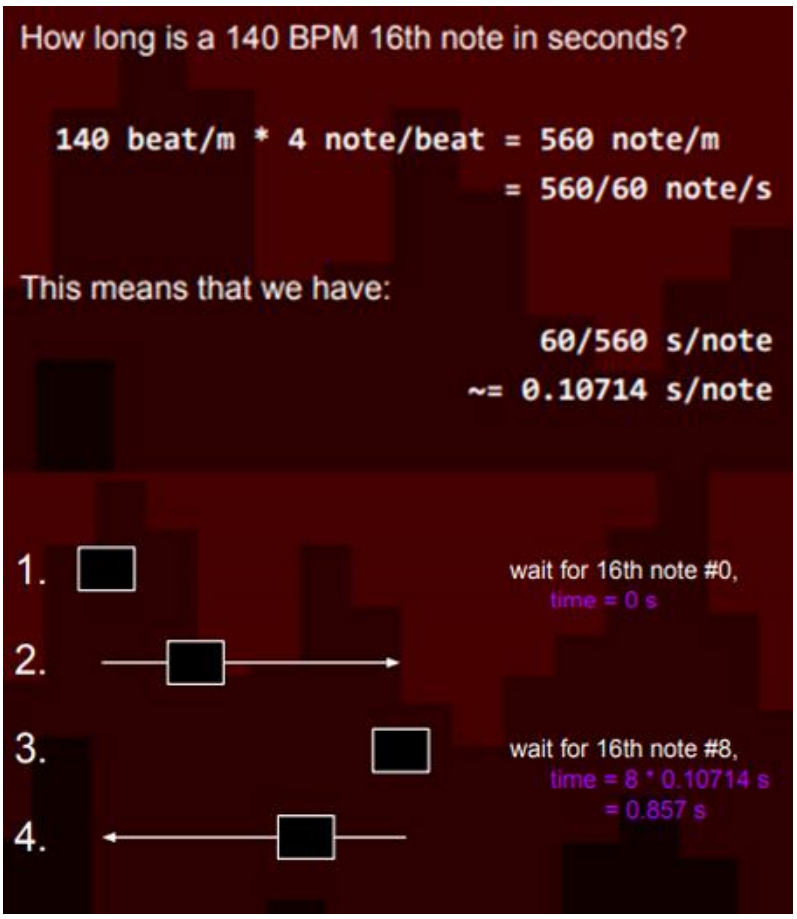

Fig 3.1: Método utilizado para lanzar eventos en conjunto con la música en *140*

Fuente: (Jakob Schmid, Sonic College, Haderslev: Game Audio - INSIDE and 140)

Schmid (Schmid, 2018) expone también como *Carlsen Games* implementa interactividad musical mediante el uso de la variación de mezcla en *140*. Por ejemplo, los elementos sonoros reaccionan a las acciones del jugador variando el volumen según su distancia en el nivel. Para ello, cada elemento jugable se implementa como un elemento sonoro separado y se coordina musicalmente siguiendo el ritmo de un bucle principal del juego. Los bucles han de ser reproducidos con 0.00002 segundos de diferencia como máximo para estar sincronizados rítmicamente.

Un problema afrontado durante la implementación es la sincronización rítmica de los bucles sonoros en el motor de *Unity*. Schmid explica que se debe a una latencia encontrada en todos los motores de sonido y es causada por el tiempo de muestreo, o *sample rate*. En el caso del motor de audio en *140*, esta latencia es de 21 milisegundos, y es el tiempo en el que el motor tarda en recibir, procesar y reproducir una señal de sonido. Además, los fotogramas de juego en *Unity* y el tiempo de procesador del motor de sonido van descompasados.

Por lo tanto, aunque el controlador reproduzca el bucle sonoro inmediatamente, este acumula el tiempo que ha tardado en recibir la señal y el tiempo que tarda el motor de sonido en procesar e iniciar la señal de sonido y, por ello, nunca va sincronizado.

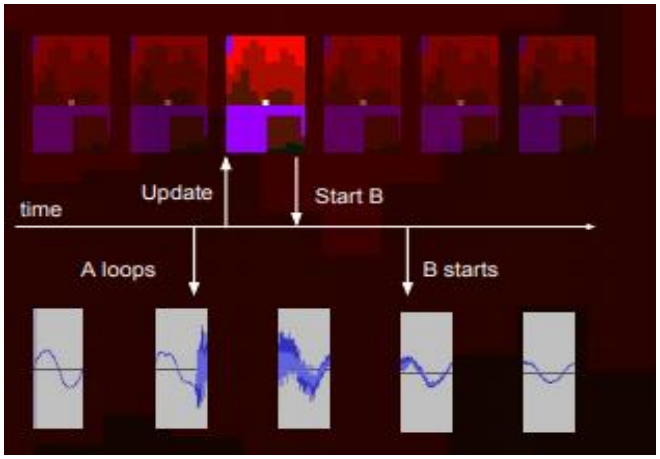

Fig 3.2: Ejemplo de la latencia ocurrida por el motor de sonido en *140*

Fuente: (Jakob Schmid, Sonic College, Haderslev: Game Audio - INSIDE and 140)

Para evitar la latencia, Schmid (Schmid, 2018) propone dos soluciones. En primer lugar, Schmid propone que se programen los bucles sonoros, calculando la latencia entre *Unity* y el motor de audio y adelantando el momento que deben ser reproducidos. Por lo tanto, mientras que el controlador de juego envía la señal de reproducir el bucle antes de tiempo, este empieza a sonar acorde con el ritmo del bucle principal.

La segunda solución propuesta por Schmid, es la sincronización de todos los bucles sonoros desde el inicio de la ejecución. De esa forma, todos los bucles quedan sincronizados desde el principio y se varia la mezcla de forma dinámica. Para ello, todos los bucles sonoros deben tener la misma longitud, o ser múltiple de esta. Los bucles empiezan silenciados y se varia su volumen durante la ejecución. Tal como explica Schmid, este es el método más sencillo de implementar y es el utilizado para la implementación de *140*.

4. Herramientas de sonorización dinámica

Una vez documentadas las definiciones que giran en torno al sonido dinámico, cabe destacar diferentes antecedentes y métodos implementados por técnicos de la industria. En primer lugar, se muestran herramientas utilizadas por grandes compañías de la industria, como *Imuse*, y posteriormente se muestran algunas herramientas accesibles para el público, como Wwise (Audiokinetic, 2000) y Fmod (Firelight Technologies, 1995).

4.1. Imuse (LucasArts)

"*Imuse* (*Interactive Music Streaming Engine*) fue desarrollado por los compositores Michael Land y Peter McConnell para permitirles crear música dinámica" (Collins 2007, p. 51). Fue desarrollado en 1991 y permitía manipular archivos *MIDI* (*Musical Instrument Digital Interface*) en tiempo real, facilitando el trabajo de los compositores al poder crear piezas con varios instrumentos y probar transiciones entre distintas capas, entre otras funcionalidades. *Imuse* fue reconocido como uno de los primeros motores complejos utilizados en varios títulos consecutivos de una misma compañía, *Monkey Island 2: LeChuck's Revenge* (LucasArts, 1991) y *Grim Fandango* (LucasArts, 1998).

4.2. Wwise (Audiokinetic)

Tal como explica Gary Hiebner (Hiebner, 2015), Wwise es un programa ejecutable externo desarrollado por *Audiokinetic* utilizado para uso en múltiples sistemas operativos diseñado para la creación de efectos sonoros y música para videojuegos. Se basa en la importación y creación de eventos para ser llamados a través del código del juego., en vez de ejecutar directamente los archivos a través del motor de juego. *Wwise* es uno de los motores más utilizados en la industria, con una amplia lista de títulos tanto de grandes cómo de pequeñas empresas. Puede ser integrado en motores como *Unity* (Unity Technologies 2005) y *Unreal Engine 4* (Epic Games, 2014).

Wwise incluye funciones como los *Random Containers*, que permite utilizar aleatoriamente varios sonidos para un sólo evento, parámetros, que permiten cambiar los sonidos de un evento con modificaciones en tiempo real, o el uso de ejecutables MIDI dentro de la propia aplicación.

En la siguiente imagen se muestra la interfaz utilizada por el usuario para la importación, creación y modificación de eventos de forma individual.

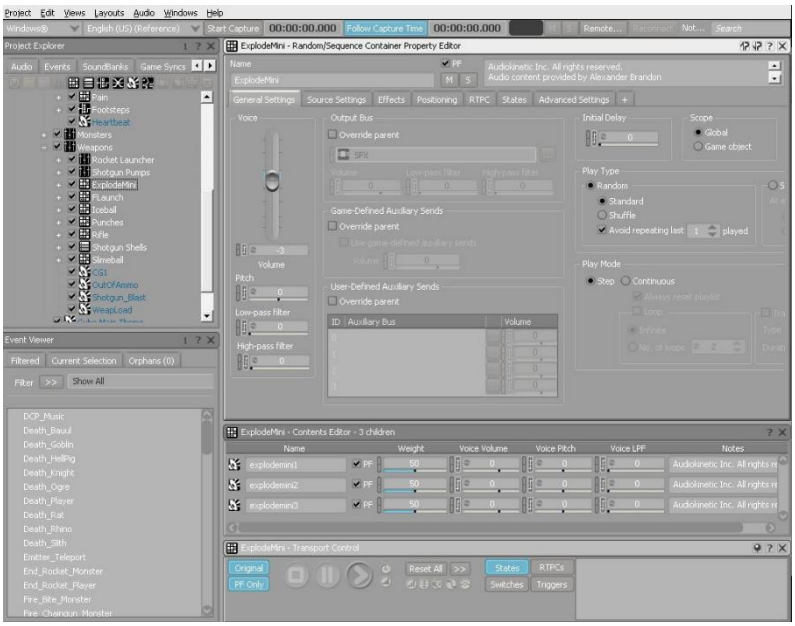

Fig. 4.1. Interfaz del programa *Wwise*.

Fuente: (Gary Hiebner, Review: Wwise For Game Audio)

4.3. Fmod (Firelight Technologies)

Similar a *Wwise* (Audiokinetic, 2000), *Fmod* es un motor desarrollado por *Firelight Technologies* para uso en múltiples sistemas operativos y, al igual que *Wwise* (Audiokinetic, 2000), puede ser integrado tanto en *Unity* (Unity Technologies, 2005) como Unreal Engine 4 (Epic games, 2014).

Fmod se presenta como una alternativa más sencilla a *Wwise* (Audiokinetic, 2000)., compartiendo muchas funcionalidades con una interfaz más limpia, pero más limitada en interacción para su usuario. Se basa también en la importación de audio, creación de eventos, y opciones como la creación de parámetros, mezcla de volúmenes, y exportación a múltiples plataformas.

En la siguiente imagen se muestra la interfaz utilizada por *Fmod* en la que un evento dividido en diferentes capas instrumentales, está siendo modificado para su posterior ejecución en el motor de juego.

Fig. 4.2. Interfaz del programa *Fmod*.

Fuente:(Firelight Technologies, Fmod Studios)

5. Objetivos

La meta de este proyecto es el desarrollo de un prototipo jugable que integre diversas mecánicas basadas en la ambientación dinámica. Con este propósito, se definen los siguientes objetivos.

5.1. Objetivos Principales

1. Analizar y categorizar los elementos sonoros que forman parte de los videojuegos, tanto su función como su implementación técnica.
2. Diseñar e implementar un prototipo jugable en *Unity* (Unity Technologies, 2005), basado en una serie de mecánicas estrechamente relacionadas con diversos elementos sonoros. Este ha de contar con un bucle de juego completo, es decir que cumpla las condiciones necesarias para la posible victoria del usuario.

5.2. Objetivos Secundarios

1. Crear un módulo que contenga una serie de funcionalidades que faciliten la implementación de interacciones entre elementos sonoros para futuros proyectos en *Unity* (Unity Technologies, 2005), englobadas en una librería o *add-on*. Este módulo ha de incluir aquellas interfaces que se hayan utilizado para desarrollar esta memoria, y podría utilizarse como base para elaborar prototipos futuros, además de servir como elemento complementario para este proyecto.

6. Diseño Metodológico y Cronograma

6.1. Metodología

La metodología del desarrollo de dicho prototipo se basará en tres fases principalmente, la planificación del prototipo, el proceso de implementación y, finalmente, la revisión del proyecto:

- En primer lugar, el proceso de planificación gira en torno a la creación de un esquema organizativo donde se muestran las mecánicas que deben implementarse.
- En segundo lugar, se realiza el proceso de implementación técnica, mediante el uso del motor *Unity* (Unity Technologies, 2005) y *Fmod* (Firelight Technologies, 1995). En este proceso se utiliza *Git* (Linus Torvalds. 2007) para mantener el control de una versión estable durante el proceso de creación.
- Finalmente, se realiza el proceso de revisión, donde, si el tiempo premia, se añadirán mecánicas adicionales que incrementen el valor del prototipo. Es a partir de esta fase donde se escribirá la memoria del prototipo, y redactarán los resultados pertinentes.

6.2. Cronograma

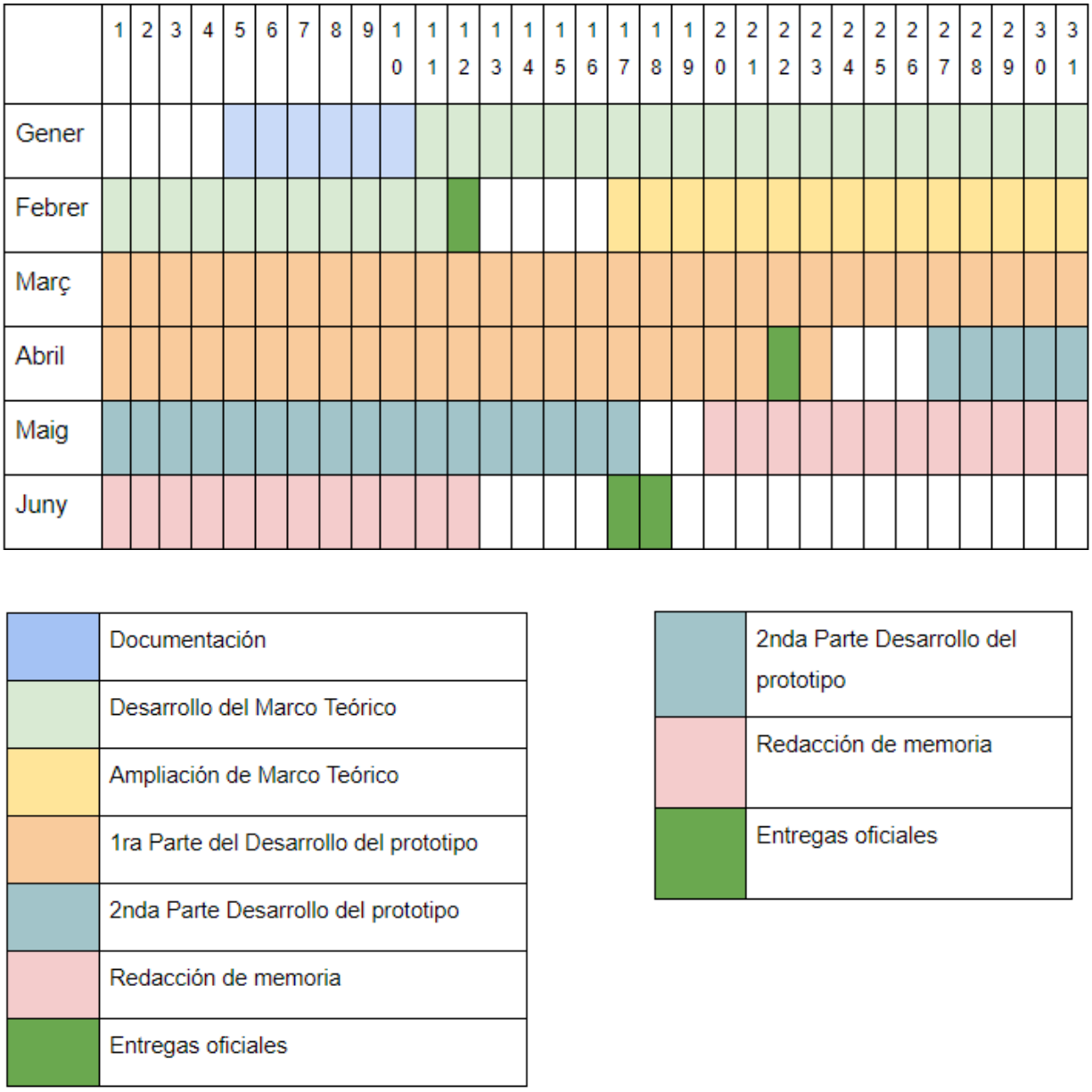

Fig. 6.1. Cronograma de elaboración de la memoria

Fuente: Elaboración propia

7. Desarrollo del prototipo

Una vez establecidas las bases teóricas relacionadas con la ambientación dinámica sonora y expuestos los objetivos entorno a los cuales girará el desarrollo del prototipo, se expone su proceso de desarrollo, comenzando por su diseño conceptual y seguido de su implementación técnica.

7.1. Diseño conceptual

Con el propósito de mostrar la variedad y potencial en la adaptabilidad del sonido en un videojuego, se presenta a continuación el diseño conceptual del prototipo.

Tal como establecen los referentes de la memoria, una base clave en el desarrollo del prototipo es la definición de un apartado visual sencillo, lo cual facilite el proceso de implementación y realce la importancia del apartado sonoro durante la experiencia. El mismo principio< se aplica también al apartado jugable, el cual consiste en una serie de mecánicas que, aunque sencillas, resultan fáciles de comprender y de interiorizar durante la corta duración del prototipo.

Cabe destacar que, pese a tener un impacto importante en la experiencia, las mecánicas de juego no giran en torno al apartado sonoro y, por lo tanto, la presencia del sonido no es esencial para completar el juego. Esto es debido a que el objetivo del prototipo no es crear una experiencia centrada en este apartado, si no en mostrar su capacidad de amplificar y mejorar la experiencia del usuario mediante diversas técnicas de adaptabilidad. Así mismo, esto permite observar y comparar la experiencia de usuario con y sin sonido.

Siguiendo estas reglas se describe el género, obstáculos y metas del prototipo y, posteriormente, se desglosan las diferentes mecánicas y elementos sonoros relacionados, los cuales se identifican siguiendo la clasificación expuesta anteriormente.

El prototipo, llamado *Reliquary*, consiste en un juego del género aventuras en perspectiva cenital, o *top-down*, en tercera persona. En este, el jugador protagoniza a un monje el cual debe recorrer un laberinto en busca de tres reliquias que ha de traer de vuelta al monasterio. Durante las travesías por el laberinto, debe evitar ser atrapado por acechadores, los cuales persiguen al monje en caso de escucharlo. Para evitarlo, el monje debe aprovecharse de las debilidades de los acechadores: el monasterio, que espanta a las criaturas en caso de acercarse, y su pobre sentido de la percepción, basado en el sonido.

Si el monje es atrapado por un acechador este reaparecerá en el monasterio y las reliquias que no hayan sido colocadas en este, volverán al laberinto. Una vez que el monje devuelve las tres reliquias al monasterio, el juego termina con la victoria del jugador.

Una vez explicado el concepto del prototipo, se explican los elementos de juego, sus mecánicas, y se desglosan los elementos sonoros relacionados pertinentes:

Monje: Se trata del avatar del usuario y su canal de interacción directa con el juego. Mediante este, el usuario puede moverse por el mapa en horizontal y en vertical. La cámara, con vista cenital, mantiene al monje siempre en el centro de la pantalla. El monje también puede coger y soltar objetos, pero solo de uno en uno. Mientras el monje esté cargando un objeto este se moverá más lentamente. El monje también puede moverse lentamente para evitar hacer ruido al caminar. En caso de llevar un objeto no puede moverse lentamente.

- Pasos: Un bucle de sonido activado cuando el monje está en movimiento. Sirve como contacto sonoro directo del movimiento del usuario. Debido a que el sonido responde directamente a la acción de movimiento del jugador, lo identificamos como un elemento sonoro de tipo interactivo. Además, este elemento puede clasificarse como un sonido diegético, ya que se trata de un sonido creado por el monje al moverse, dentro del contexto narrativo de juego. Por lo tanto, este elemento sonoro es de tipo interactivo diegético.

- Objeto recogido/soltado: Elementos sonoros emitidos al recoger un objeto. Estos se sincronizan al ritmo de la música principal y su propósito es proporcionar reacción y satisfacción directa hacia el usuario. Estos ocurren dentro del contexto narrativo y son activado por la acción directa del jugador. Por lo tanto, se clasifican como elementos interactivos diegéticos.

Reliquia: Se trata del objeto clave para la condición de victoria del jugador. Tres reliquias estarán repartidas por el laberinto. El usuario ha de recogerlas y traerlas de vuelta al monasterio de una en una. La reliquia emite una melodía que el jugador debe utilizar para encontrarla en el laberinto. Hay una melodía específica para cada reliquia.

- Bucle melódico: Elemento sonoro que el jugador utiliza para identificar y poder encontrar la reliquia en el laberinto. Para ello, el elemento sonoro se liga al objeto de juego y varia su volumen en intensidad y orientación de izquierda a derecha según la distancia y posición del monje. Cada reliquia contiene su melodía que se coordina con el bucle musical del monasterio en ritmo y armonía. Esto tiene como objetivo dar al jugador una sensación de similitud que le incita a plantearse encontrar el origen de este sonido como objetivo, sin explicárselo de forma directa. También tiene el propósito de actuar como *leitmotiv* para que el jugador pueda diferenciar las reliquias entre sí. Adicionalmente, el elemento sonoro varía su mezcla dependiendo de ser recogido o soltado. Debido a que el elemento sonoro no depende de ninguna interacción directa del jugador y de estar fuera del contexto narrativo, se trata de un elemento sonoro adaptativo no diegético.

Monasterio: Se trata de una ubicación ubicada en el centro del laberinto, y es la zona segura y punto de aparición del monje. El jugador debe traer las tres reliquias y debe colocarlas en sus respetivos relicarios para cumplir la condición de victoria.

- Bucle musical: Se trata del elemento sonoro que caracteriza la principal ambientación musical del juego. Este elemento varía su instrumentación, forma y mezcla según el estado del jugador y de la hora del día. Al estar en el monasterio, el bucle contiene un coro angelical que le da la bienvenida, pero al estar dentro del laberinto tan solo contiene un ritmo de batería. El bucle también se adecua al peligro. Si el jugador es perseguido por un acechador, este cambia a un ritmo frenético hasta que logra huir de él. Mediante estas variaciones de mezcla, este bucle sonoro busca crear una sensación de ritmo al mismo tiempo que poner en contexto al jugador de manera instintiva. Debido a que no es parte del contexto y forma parte de la ambientación musical, este se identifica como un elemento adaptativo no diegético.
- Reliquia colocada: Se trata de una melodía de campanas activado por el jugador al colocar una reliquia. Esta cambia según la melodía de la reliquia que se coloca. De esta forma el juego recompensa al jugador por haber conseguido parte de la condición final de victoria. El sonido forma parte de la ambientación musical y no forma parte del contexto de juego. Por lo tanto, se trata de un elemento interactivo no diegético.

Acechador: Se trata del antagonista principal del juego. Normalmente, se encuentra vagando por el laberinto lentamente. El acechador tiene un pobre sentido de la percepción, por lo que no detecta al monje si se encuentra lejos o camina sigilosamente. Este rango de percepción es indicado por un área circular a su alrededor. En caso de detectar al monje, el acechador cambia su velocidad y persigue al monje hasta tocarlo. Al tocarlo, el acechador mata instantáneamente al monje. Si el monje se aleja lo suficiente del acechador, este entra en estado de búsqueda. En este estado, el acechador va hasta la última posición donde ha detectado al jugador. Si, persiguiendo al monje, el acechador se acerca demasiado al monasterio, este entra en estado de huida, y se aleja en dirección de vuelta al laberinto. Puede haber más de un acechador en una zona del laberinto.

- Bucle sonoro: Se trata de un elemento sonoro utilizado para mostrar al usuario el estado actual del acechador. Este varía según el estado en el que se encuentra el acechador, siendo estos estados vagar, buscar y perseguir. La variación del bucle de perseguir se utiliza también para el estado de huida. Este bucle también se adecua a la posición de pantalla del acechador. Mediante esto, se busca poner en contexto al jugador del estado del acechador y de su situación en la pantalla y crear una sensación de peligro sin que tenga que estar mirándolo directamente. Este se encuentra dentro del contexto del juego, como un sonido que emite el propio acechador. Debido a que cambia para adaptarse al estado actual del enemigo, se trata de un elemento sonoro adaptativo diegético.

Una vez descritos los elementos de juego y los elementos sonoros relacionados con estos, se muestra el resultado de su clasificación en el prototipo en la tabla 7.1:

Elemento de juego	Elemento sonoro	Clasificación
Monje	Pasos	Interactivo, diegético
	Objeto recogido	Interactivo, diegético
	Objeto soltado	Interactivo, diegético
Reliquia	Bucle melódico	Adaptativo, no diegético
Monasterio	Bucle musical	Adaptativo, no diegético
	Reliquia colocada	Interactivo, no diegético
Acechador	Bucle sonoro	Adaptativo, diegético

Tab. 7.1. Clasificación de los elementos sonoros del prototipo

Fuente: Elaboración propia

7.2 Implementación del prototipo

Una vez descrito el concepto del prototipo y establecidas las herramientas, a continuación, se describe el proceso de implementación técnico, dividido en la organización del estudio de *Fmod*, la organización de código y arquitectura de clases en *C#* y la organización del proyecto en *Uniy 3D*.

7.2.1 Organización del estudio

Con el objetivo de agilizar la implementación del prototipo, el apartado sonoro se implementa mediante el uso de *Fmod Studio* (Firelight Technologies, 2020) que, tal como se ha expuesto previamente, incluye varias funcionalidades que facilitan la introducción de sonido adaptativo en juegos al actuar como interfaz de control para el desarrollador. El proceso de implementación en el estudio consiste en la creación de una sesión en *Fmod Studio*, para ser posteriormente exportado al motor de sonido, el cual está integrado en Unity3D, para poder llamar y variar los elementos sonoros en tiempo de ejecución. En este caso se usará la versión 2.01 del estudio.

La integración de un elemento sonoro en el editor empieza por la importación de los archivos de sonido que se desee utilizar. Estos son importados en el formato de compresión *.WAV*. Se pueden escoger diferentes ajustes a la hora de importar el archivo, como es el modo de codificación, el modo de carga, compresión y el muestreo. Estos ajustes afectan al tiempo de carga, calidad y llamada del elemento durante la ejecución, y variarán según el tipo de elemento y uso que se le dé durante el juego. En las figuras 7.1 se muestran los ajustes escogidos para el bucle melódico de una reliquia.

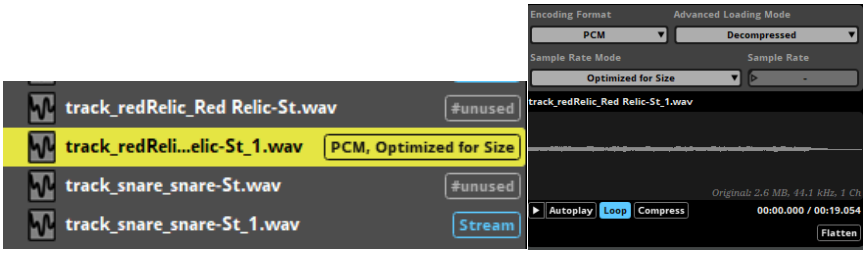

Fig. 7.1. Ajustes del bucle melódico en Fmod.

Fuente: (Firelight Technologies, Fmod Studio 2.01)

Una vez importados los archivos del elemento, estos se agrupan dentro de un nuevo evento de estudio. Tal como se describe en el Manual de Usuario de *Fmod 2.01* (Firelight Technologies, 2020) un evento es una unidad instanciable de sonido correspondiente a un elemento sonoro del juego. Dependiendo de la clase de la mecánica o elemento del juego relacionado con el evento, crearemos un evento de tipo 3D o 2D.

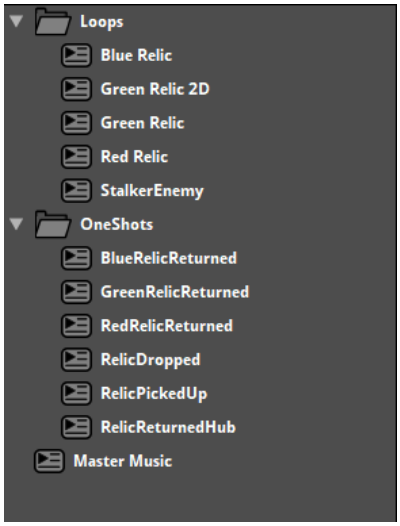

Fig. 7.2. Ventana de eventos de Fmod Studio.

Fuente: (Firelight Technologies, Fmod Studio 2.01)

Mediante el uso del estudio, podremos modificar los ajustes y comportamiento del evento para su posterior instanciación en el código del editor y su ejecución en tiempo real. Según el Manual de Usuario de *Fmod Studio 2.01* (Firelight Technologies, 2020), un evento se divide en tres funcionalidades:

- **Instrumentos**: Elementos que contienen el archivo de audio para poder introducirlo en una pista del evento.
- **Pistas**: Módulos del evento que pueden utilizarse para separar los instrumentos en diversas capas, con cada pista actuando como una capa de **mezcla** por separado. Por defecto todas las pistas son enrutadas a una pista general o *master track*.

- **Parámetros**: Propiedades que pueden ser utilizadas para modificar los instrumentos, lógica, estructura y mezcla del evento. Estos son los atributos que se utilizan para variar un elemento sonoro durante la ejecución.

En la figura 7.3 se muestra la vista del evento correspondiente al bucle melódico de la reliquia en *Fmod Studio*. En esta, se puede observar la pista principal del evento, que contiene el instrumento que contiene el archivo de audio. Mediante el uso de una región de bucle, el evento automáticamente reiniciará su ejecución al llegar al final de esta, pudiendo reproducirse así de forma infinita hasta que pare de reproducirse el evento. En la figura 7.4 se muestra la totalidad de parámetros utilizados en los diferentes eventos del prototipo

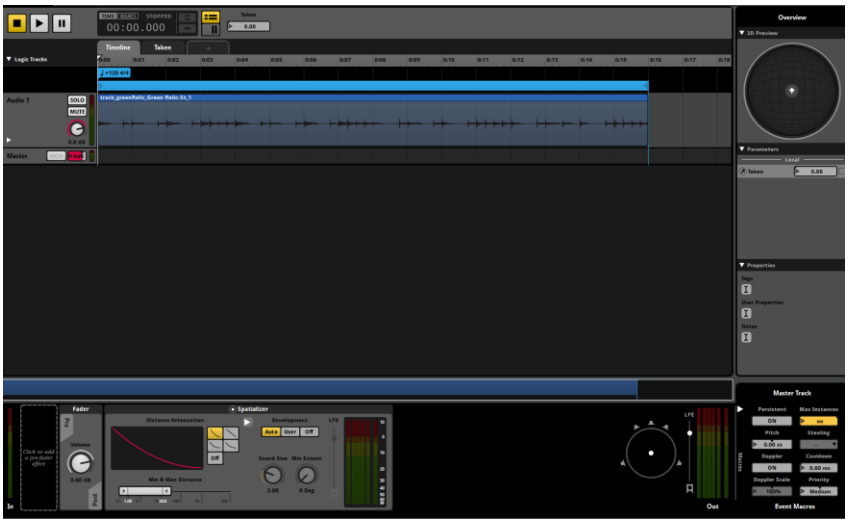

Fig. 7.3. Ventana del evento en Fmod studio.

Fuente: (Firelight Technologies, Fmod Studio 2.01)

Fig. 7.4. Parámetros utilizados en el prototipo.

Fuente: (Firelight Technologies, Fmod Studio 2.01)

Este evento también contiene un parámetro llamado “*taken*”, o cogido, el cual se modifica en el editor a medida que el usuario interactúe con el objeto en el juego, afectando así a la mezcla del evento.

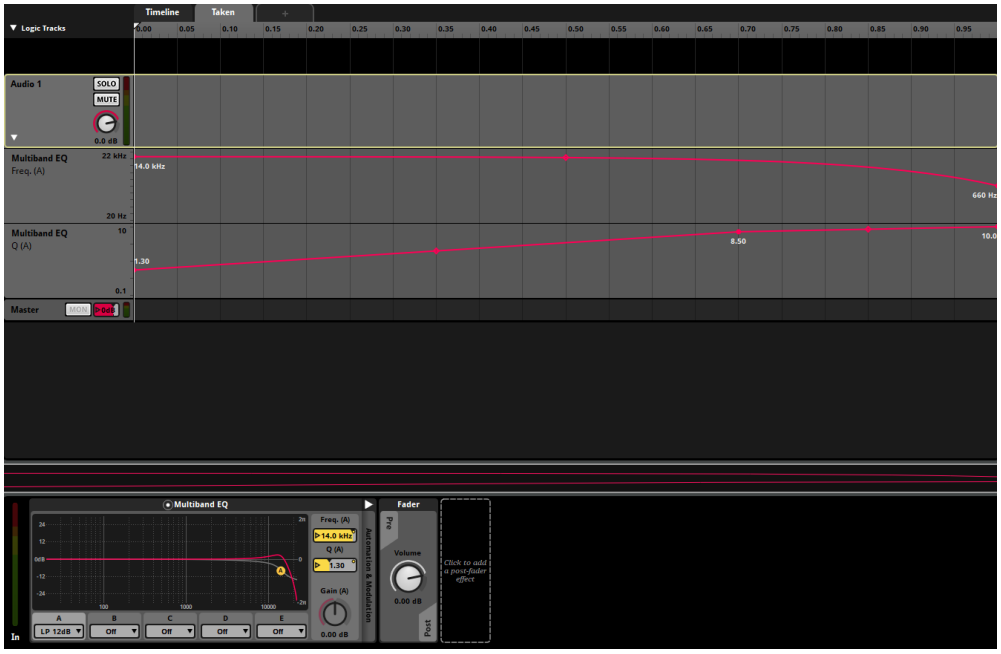

Fig. 7.5. Parámetros del evento Reliquia Verde en Fmod Studio.

Fuente: (Firelight Technologies, Fmod Studio 2.01)

Una vez se crean y editan los eventos con los elementos sonoros deseados, estos han de ser exportados al editor para poder ser llamados. Para ello, los eventos se agrupan y comprimen en bancos de datos, con el objetivo de ayudar al desarrollador a controlar la carga de datos del estudio durante la ejecución, cargando los eventos y descomprimiendo el contenido de los bancos según se necesite.

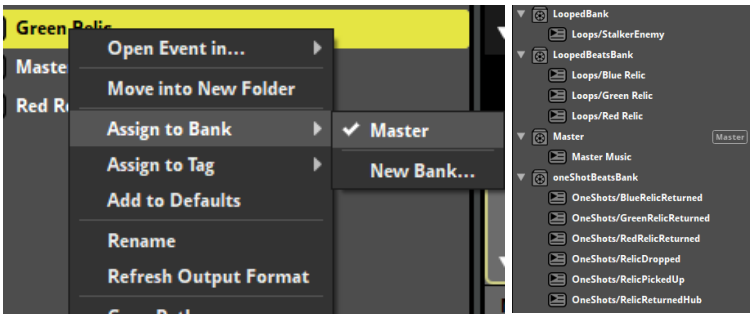

Fig. 7.6. Ventana de bancos en Fmod Studio.

Fuente: (Firelight Technologies, Fmod Studio 2.01)

Una vez se han asignado los eventos, se hace una ***build*** del banco o bancos creados y se exportan al proyecto. En caso de haber conectado la sesión del estudio al proyecto del editor, este se enrutará automáticamente, permitiendo así trabajar y modificar los eventos de la sesión continuamente sin tener que importar manualmente los bancos actualizados en el proyecto de *Unity*.

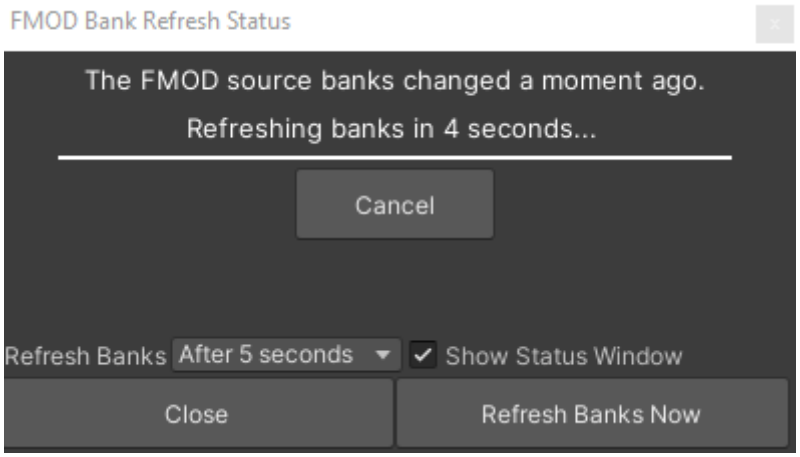

Fig. 7.7. Ventana emergente de Fmod Studio en *Unity 3D*.

Fuente: (Firelight Technologies, Fmod Studio 2.01)

Analizando la decisión de utilizar *Fmod Studio* (Firelight Technologies, 2020), este cumple con el propósito de trabajar con los elementos sonoros por separado una vez implementados en el motor de juego.

Esto se observa por ejemplo a la hora de trabajar en el bucle musical, el cual se ha podido separar y automatizar los instrumentos en varias pistas mediante el uso del estudio, manteniéndolo como una única entidad dentro del juego.

Fmod Studio (Firelight Technologies, 2020) también ha facilitado la implementación de variación en elementos sonoros con efectos ya integrados. Un ejemplo es la percepción acústica de las reliquias y acechadores a través del efecto de sonido espacial. Este permite transferir la posición del objeto respecto al usuario en la escena de Unity al evento en el estudio, donde este calcula automáticamente el volumen pertinente a la distancia.

7.2.2 Arquitectura Editor-Estudio en C#

Una vez expuesto el procedimiento de crear y exportar la sesión de *Fmod Studio* en *Unity*, se muestra a continuación el proceso de implementación del código para el prototipo, mostrando la lógica y arquitectura de clases utilizada en la implementación de las mecánicas de juego descritas anteriormente y la integración del motor de audio *Fmod*.

Tal como se menciona, la implementación de código se lleva a cabo utilizando *C#*, con *Rider* como *IDE* principal de desarrollo. La organización y estructura se llevan a cabo utilizando métodos "*clean"* siguiendo los cinco principios *SOLID*, y utilizando patrones de distintos tipos con el objetivo de obtener un diseño técnico del proyecto que facilite la legibilidad durante el proceso de creación y posibilite la reutilización de partes del código en futuros prototipos.

Adicionalmente, el código se separa también en *namespaces* o regiones, que permiten segmentar las clases de código en ámbitos, o *scope*, según el propósito que cumplen. Todas las regiones de código forman parte de una región general, que **encapsula** todo el código del proyecto con el nombre de este, en este caso siendo *Reliquary*.

Sistema de control de audio

Fmod proporciona su motor de audio para *Unity* en forma de *plugin* gratuito. En este, se encapsulan librerías que ofrecen opciones *scripting* en forma de clases y métodos y atributos. Estos permiten al desarrollador integrar e interactuar con los elementos definidos en la sesión de estudio, ya sea la carga de bancos, modificación de un parámetro o la instancia de un evento.

Con el objetivo de encapsular las funcionalidades del estudio que se necesiten en el prototipo y segregar las dependencias y responsabilidades de librerías de *Fmod* de ámbitos que no correspondan a la gestión de la sesión del estudio o sus elementos, se define a continuación la siguiente estructura de clases dentro de la región *Reliquary.Sound*

- ***SoundController:*** Es el controlador global del sonido del proyecto y el punto intermedio entre el estudio y las mecánicas. Este contiene un listado de bancos de eventos y se encarga de cargarlos en el proyecto según sea necesario en la ejecución. También contiene dos métodos estáticos, *PlayElement* y *StopElement*, en los cuales se gestionan las instancias de los eventos a través de la clase *ASoundEement.* También se encarga de gestionar la coordinación de elementos musicales utilizando un elemento general como referente mediante una instancia de la clase *StudioEventProxy*, a la cual asignará los elementos como observadores hasta que sea su torno de empezar a sonar.
- ***ASoundElement*:** Clase abstracta encargada de encapsular los métodos y atributos relacionados con la gestión de un evento del estudio. Esta contiene los ajustes de un evento en forma de componente que el desarrollador puede editar manualmente. Este componente se referencia en aquellas clases encargadas de gestionar la *vista* de una mecánica de juego, con el propósito de ejecutar elementos sonoros sin tener que gestionar las dependencias de un evento de *Fmod*. La clase contiene métodos abstractos para poder hacer sonar, parar y extraer una instancia del evento de estudio en la escena de juego.

- ***LoopedSoundElement:*** Clase hija de *ASoundElement* utilizada para elementos sonoros de tipo bucle, y que, por lo tanto, solo requieren de una instancia del evento.
- ***OneShootSoundElement:*** Clase hija de *ASoundElement* utilizada para elementos sonoros de tipo *one shoot*, es decir, elementos que tienen un principio y fin y, por lo tanto, pueden contener varias instancias del mismo evento simultáneamente.
- ***SoundParameter:*** Clase con la responsabilidad de actuar como interfaz de un parámetro del estudio. Este guarda los ajustes del parámetro, el elemento sonoro que afecta, y método que modificará el parámetro en tiempo real al ser llamado.
- ***StudioEventProxy*****:** Clase encargada de inicializar y encapsular métodos, *StudioBeatCallback* y *StudioMarkerCallback*, que observan un evento de estudio y guardan la información en tiempo real en una instancia de la clase *EventTimelineInfo.* Debido a que estos métodos tienen que comunicarse directamente con el código nativo en C del motor de audio, tienen que contener el atributo *[AOT.MonoPInvokeCallback]* y deben ser estáticos.
- ***EventTimelineInfo:*** Clase encargada de encapsular la información editada por los métodos que interoperan con el motor de audio. Por ello, esta clase contiene el atributo *[StructLayout(LayoutKind.Sequential)]*, que permitirá que sus atributos se modifiquen de forma secuencialmente a medida que cambie el código nativo del motor. Adicionalmente las propiedades de la clase son de tipo reactivo, un tipo de atributo proporcionado por la librería reactiva *UniRx*. lo cual permite que se suscriban bloques de código que respondan a un cambio en el valor del atributo. Por ejemplo, la propiedad reactiva de tipo entero *bar*, guarda el compás actual del evento. Al cambiar, se ejecutarán aquellos métodos suscritos a la propiedad, como por ejemplo *OnBarChanged* de la clase *SoundController*, que recibirán el nuevo valor como parámetro.

En la figura 7.7 se muestra un diagrama de las clases relacionadas el motor de audio *Fmod*, descritas anteriormente:

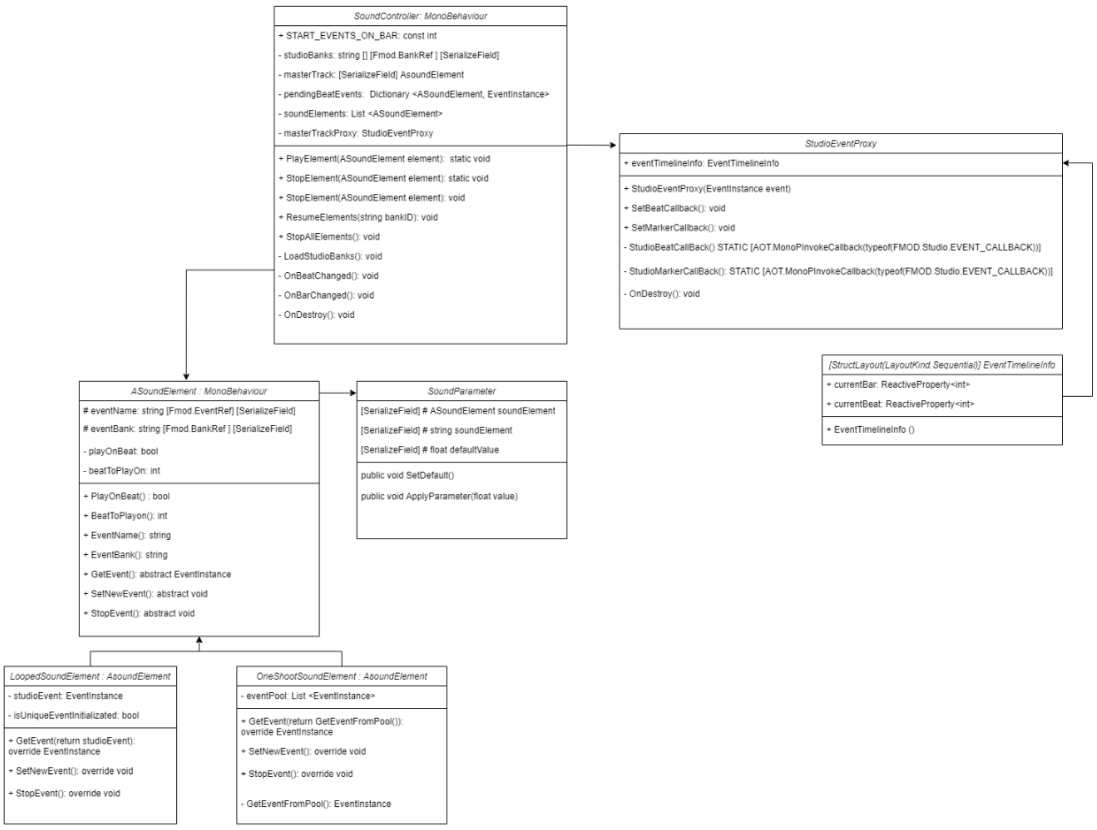

Fig. 7.8. Diagrama de clases del dominio sonido, *Reliquary.Sound*.

Fuente: Elaboración propia

Analizando la implementación técnica del apartado sonoro en *Unity*, se consigue crear una estructura de código separada del resto de la arquitectura del proyecto. Este módulo, aunque dependiente del motor de audio y estudio de *Fmod* (Firelight Technologies, 1995), permite encapsular las funcionalidades para gestionar los eventos, parámetros y bancos del estudio de manera separada al resto del código, limitándolo a referencias y llamadas desde las diferentes clases tipo vista del juego. No obstante, durante la implementación la implementación del controlador trae diversos problemas.

Uno de estos se trata de la música interactiva, sobre la que surge la necesidad de coordinar los diferentes eventos del estudio acorde al ritmo de la música principal. Siguiendo la toma de decisiones observada en el desarrollo de *140* (Carlsen Games, 2013) se decide crear una funcionalidad para anticipar la reproducción de los eventos al ritmo. Pero, debido a un retraso no documentado entre los eventos del estudio y el motor de juego, no se puede calcular el ritmo actual del evento musical desde el motor de juego con fidelidad.

Este problema se puede solucionar en caso de utilizar el motor de sonido integrado en *Unity*, que ofrece un control más directo sobre el sonido reproducido. En su lugar, se decide seguir con el uso de *Fmod* por las funcionalidades que integra el estudio. Una solución alternativa es combinar los dos motores de audio, pero la documentación de *Fmod* no lo recomienda debido al gasto de recursos que causa tener dos motores de audio activos.

Para solventarlo, se hace uso de los *callbacks* del estudio en la clase *StudioEventProxy*, que permiten reproducir los eventos al ritmo del evento musical principal. Para una reproducción inmediata sin retraso también se carga los eventos del estudio utilizando la opción *loadSampleData*, que carga los eventos al iniciar la ejecución y los prepara para su reproducción.

Esto conlleva otro inconveniente, que es un alto uso de memoria RAM desde el principio de la ejecución. En un proyecto que no requiera la coordinación musical, se administra la memoria de manera ordenada cargando y descargando los eventos en bancos separados según se necesite. Debido a que se trata de un prototipo sencillo sin muchos eventos sonoros en el estudio, el uso de la memoria se mantiene bajo, pero no es una práctica recomendada para proyectos más grandes.

Durante la implementación de los elementos de juego, se plantea la posibilidad de utilizar variación de la forma en el bucle musical para cambiar entre diferentes secciones del evento. Esto trae consigo dos problemas. Por un lado, se intenta utilizar un parámetro de tipo global que modifica el estado de los diferentes bucles internamente en el estudio. Sin embargo, el estudio no permite editar parámetros globales fuera de este.

En su lugar, se utiliza el mismo parámetro en el estudio en varios eventos, y la clase *SoundParameter* guarda referencia de todos los elementos sonoros que se vean afectados por este parámetro. A parte, al cambiar entre secciones en diferentes bucles coordinados musicalmente, se pierde el ritmo entre bucles.

Como consecuencia, se decide dejar de lado la variación de forma, y se implementa la orquestación vertical mediante el uso de las técnicas de variación del volumen y variación de la forma. Estas no requieren alterar los bucles, y, por lo tanto, se puede mantener el ritmo durante toda la secuencia.

Arquitectura de los elementos de juego

Una vez expuesta y analizada la lógica relacionada con el motor de sonido, se expone a continuación los procesos y métodos empleados en la implementación de las mecánicas de juego.

De igual forma que con el apartado anterior, cada mecánica constituye un dominio que opera de forma separada de los demás. Cada ámbito se implementa partiendo de la arquitectura *MVC*, o *model-view-controller* el cual divide el código de cada dominio en tres tipos de interfaces o clases:

- **Modelo (*Model*):** Interfaz encargada de encapsular los datos del dominio, almacenando la información de este.
- **Vista (*View*):** Interfaz encargada de encapsular los atributos y responsabilidades relacionados con la presentación del dominio de cara al usuario, como es la reproducción de sonido o la entrada de información por los periféricos del jugador. En el caso del prototipo, las clases de tipo vista son las únicas que heredan de la clase *MonoBehaviour*.
- **Controlador (*Controller*):** Interfaz encargada de actuar de intermediario y gestionar las operaciones entre la vista y el modelo del dominio. Este guarda la mayoría de las operaciones del ámbito

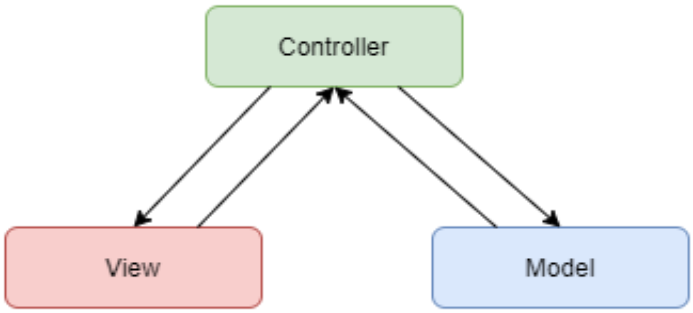

Fig. 7.9. Arquitectura del patrón de diseño MVC.

Fuente: Elaboración propia

El uso de este patrón de software permite al desarrollador separar los componentes de cada dominio de manera arbitraria por responsabilidades, obteniendo como resultado un código modular legible y extensible, tal como se establece al inicio. Cada dominio también hace uso de otros tres tipos de patrones fuera del patrón *MVC*:

- **Patrón Instalador *(Installer)*:** Interfaz encargada de inicializar el resto de clases del ámbito al que pertenece. Las clases de este tipo heredan de *MonoBehaviour*, guardan una referencia de la vista del dominio y tienen un único método Initialize o inicializar. En este se pasan las referencias del modelo y comandos del dominio y se instancia su controlador. El único instalador que rompe esta regla es el *MainInstaller*, o instalador general, el cual guarda referencia e inicializa el resto de instaladores del prototipo. Una vez ejecutadas, estas clases terminan su función y no vuelven a utilizarse durante la ejecución
- **Patrón comando *(Command)*:** Interfaz encargada de actuar como intermediaria en una operación concreta entre dos ámbitos distintos. Estas clases son bloques de código desechables destinadas únicamente a encapsular código entre dominios separados, de tal forma que estos puedan operar de manera independiente. Estos contienen las referencias de las clases implicadas en la operación, habitualmente de tipo vista o modelo, y un método llamado *"Execute"* el cual contiene el código de la operación.
- **Patrón configuración (*Configuration)*:** Tipo de interfaz encargada de tratar información compartida por las diferentes instancias de un dominio. Esta interfaz se compone de una clase serializable de tipo *ScriptableObject*, donde se encapsulan las propiedades que se quieran configurar. Mediante el uso de la clase *ScriptableObject*, se crea una instancia de la clase en el archivo, que el desarrollador puede configurar. Una vez editado, el objeto se referencia al instalador del dominio, el cual pasa la referencia a su clase modelo.

Una vez se han descrito los patrones de diseño de software utilizados durante el proyecto, a continuación, se expone la estructura de código de cada elemento de juego. Para ello, se utilizan diagramas de clase como apoyo visual.

En primer lugar, en la figura 7.10, se encuentra el dominio del jugador, *Reliquary.Player*, que agrupa las funcionalidades relacionadas con el monje, el avatar del jugador. Estas incluyen el movimiento del avatar, los efectos sonoros que emite y la gestión de objetos y estados del jugador.

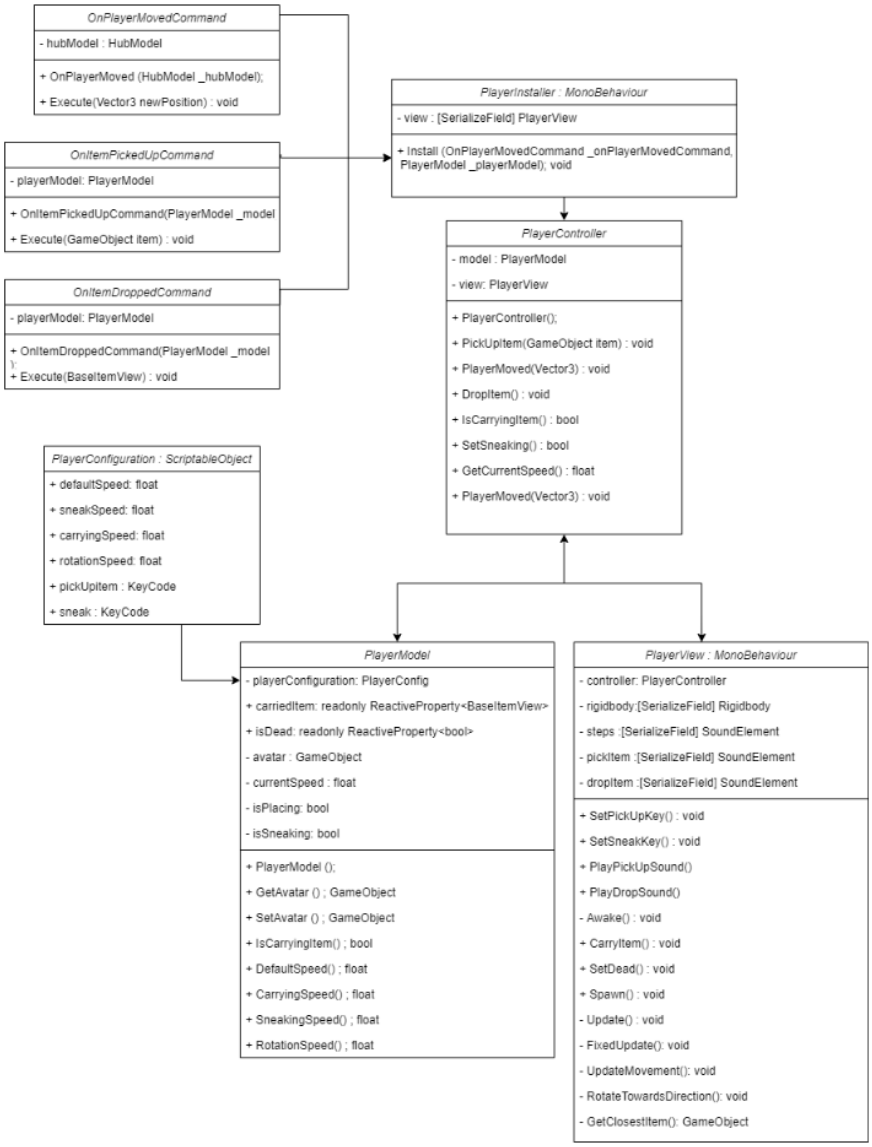

Fig. 7.10. Diagrama de clases del dominio del jugador, *Reliquary.Player*.

Fuente: Elaboración propia

Tal como se ve en el diagrama, el código se estructura en ocho interfaces. La estructura es inicializada por el instalador general, que crea una instancia del *PlayerModel* con su configuración y la pasa al *PlayerInstaller*. Dentro de este, se instancia el controlador referenciando a la vista, el modelo y los tres comandos *OnItemDropped*, *OnItemPickedUp* y *OnPlayerMovedCommand*. En el constructor del controlador, se observan los cambios de la propiedad reactiva *carriedItem*, que guarda el *GameObject* del objeto adquirido. Dentro de la lógica, se gestiona la posición del objeto y se une este a la vista del jugador para que le siga en el movimiento llamando al método *CarryItem*. También se observa la variable reactive *isDead*. Al cambiar, esta llama a los métodos *SetDead* y *Spawn* de la vista actualizando así el objeto del jugador. Finalmente, se configura en la vista las teclas de sigilo y coger/dejar objeto, referenciadas en *PlayerConfiguration*.

En el caso de la gestión del movimiento, el modelo guarda las diferentes velocidades del avatar, encapsuladas en la clase *PlayerConfiguration*. El controlador contiene la lógica utilizada para gestionar la velocidad actual del jugador dentro del método *GetCurrentSpeed*, el cual es llamado cada fotograma por la vista-. Esta graba el *input* del jugador y calcula la dirección en el método *UpdateMovement*. Esto se debe a que el tiempo de juego es gestionado por las clases de tipo *MonoBehaviour*, y las clases de tipo controlador no usan estas como base, por lo que no se puede gestionar el tiempo de juego dentro de la clase *PlayerController*. Se observa si se está apretando la tecla asignada al movimiento sigiloso y si se está llevando un objeto y, dependiendo de estas condiciones, se obtiene una velocidad u otra. Esta velocidad es devuelta a la vista, donde se aplica al componente *Rigidbody* del avatar. También se llama al comando *OnPlayerMoved*, dentro del cual se calcula y notifica la distancia del jugador al monasterio. En el método *UpdateMovement* se gestiona también el elemento sonoro de las pisadas del jugador.

Por el lado de la gestión de objetos, cuando el jugador presiona la tecla asignada a este apartado se observa en el controlador si el jugador está llevando un objeto o no, mediante el uso del método *IsCarryingItem*. En caso de no estar cargando con ninguno, la vista llama al método *GetClosestItem*, mediante el cual se guarda una referencia del objeto de tipo *ítem* que esté colisionando con un *trigger* externo.

Una vez hecho esto, se llama al método *PickUpItem* del controlador, que guarda el objeto en la variable *carriedItem* del modelo. También se llama al método de la vista *PlayPickUpSound*, que llama al controlador de sonido para poner en cola al elemento sonoro del mismo nombre con tal de que suene sincronizado al ritmo de la música del juego.

Este, también, llama al comando *OnItemPickedUp*, dentro del cual se notifica a la estructura del objeto que ha sido cogido por un jugador. En caso de tener un objeto, se llama al método *DropItem* de controlador. Este luego llama al comando *OnItemDropped*, que borra la referencia del objeto de la propiedad *carriedItem* y notifica a la vista del objeto del jugador de que ha dejado de estar cogido. De manera similar al caso opuesto, también llama al método de la vista *PlayDropSound*.

En la figura 7.11 se observa la apariencia del monje en la escena. Se observa también los componentes sonoros que utiliza.

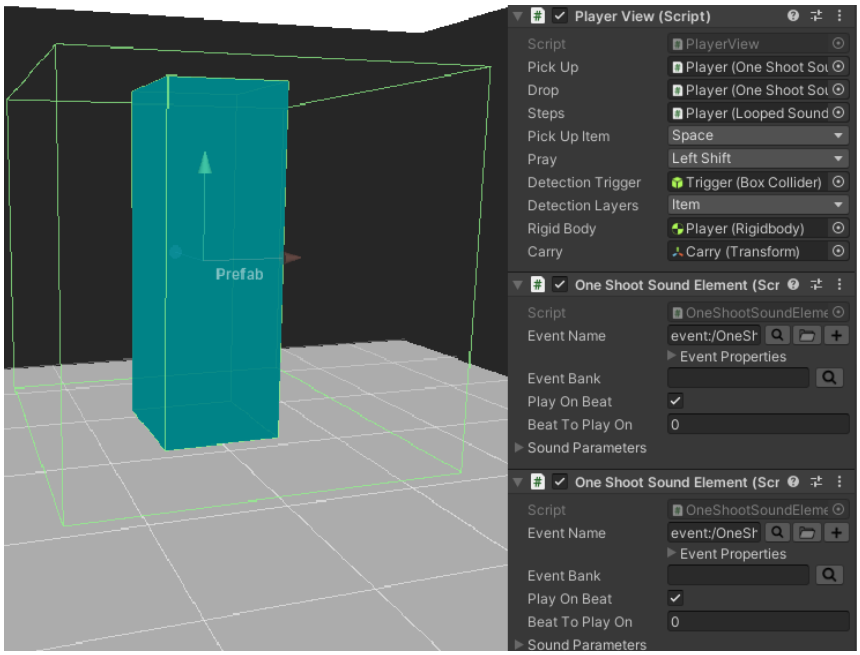

Fig. 7.11. Apariencia y organización del monje en la escena de juego

Fuente: Elaboración propia

A continuación, en la figura 7.12, se describe el dominio de la reliquia, *Reliquary.Relic*, que agrupa los atributos utilizados por el objeto interactivo del mismo nombre. Esta gestiona únicamente su propia lógica de ser colocada en su pilar del monasterio y su bucle sonoro.

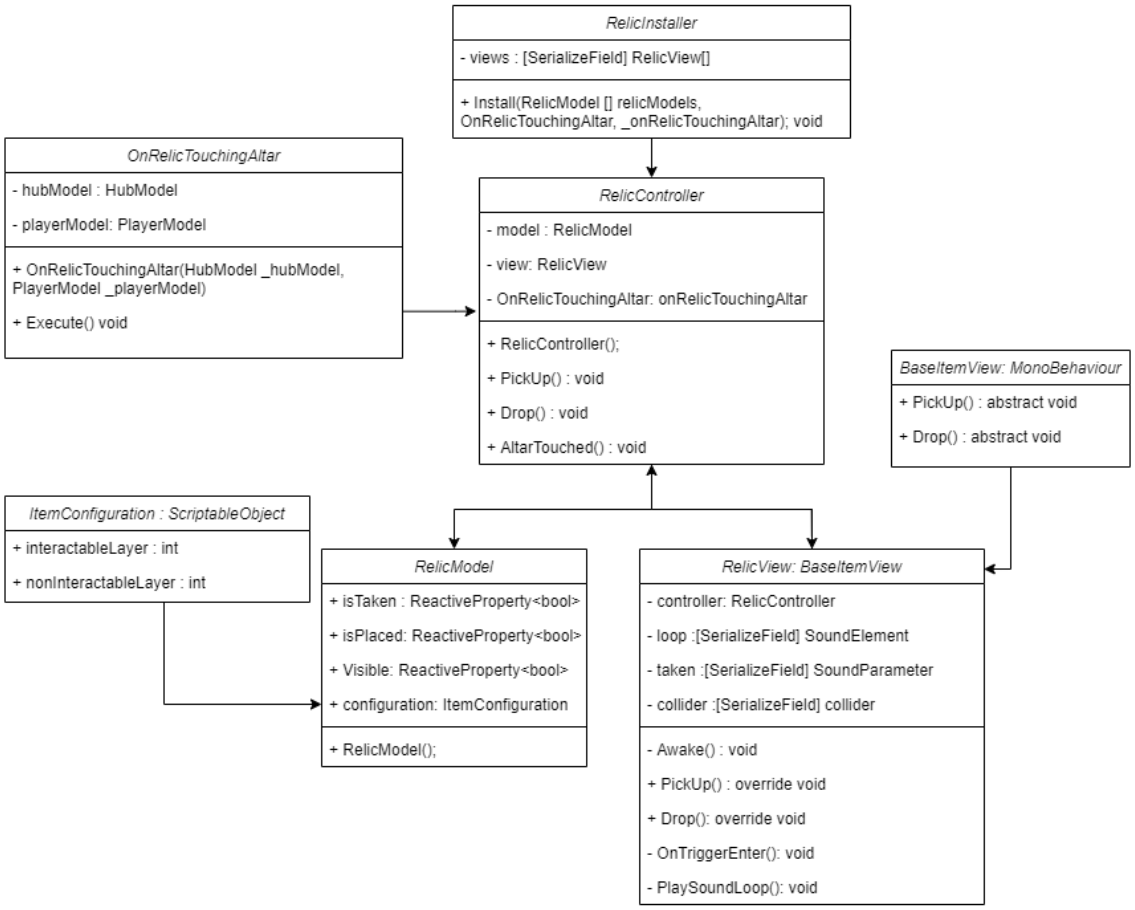

Fig. 7.12. Diagrama de clases del dominio de la reliquia, *Reliquary.Relic*

Fuente: Elaboración propia

A diferencia del dominio anterior, el juego contiene varias instancias de la estructura de la reliquia, las cuales ya están instanciadas y colocadas en la escena al ser cargadas. Por ello, el instalador guarda referencia de la clase vista de todas las reliquias del juego para iterar sobre el proceso de inicializar cada una de ellas.

La vista utiliza una clase base llamada *itemView* genérica, que encapsula métodos comunes de un objeto, *Drop y PickUp*, los cuales se llaman externamente cuando el jugador interactúa con un objeto. Estos, a través de una referencia del controlador en la vista, llaman a los métodos el mismo nombre. por un lado, el atributo *isTaken* se observa al instanciarse el controlador. Al modificarse el modelo, la lógica suscrita modifica el parámetro *taken* de la vista, para modificar el sonido. También se modifica la capa de la vista, para impedir que otras entidades interactúan con el objeto mientras ha sido cogido por el jugador. Además, en la vista, el bucle sonoro es puesto en la cola del controlador de sonido al empezar el bucle de juego desde el método *Awake*.

Finalmente, respecto a la gestión del pilar, la vista observa las colisiones y detecta si se trata de su destino mediante la llamada del método *IsAltarTouched*. Este compara los identificadores de los dos objetos y, en caso de coincidir, se llama al método *PlayPlacedSound* y se ejecuta el comando *OnRelicTouchingAltar*. Este gestiona el registro del objeto en el monasterio, modificando así la condición de victoria del jugador. Este comando también modifica el modelo de jugador para quitar la reliquia de su inventario. En la figura 7.13 se muestra la apariencia de la reliquia verde en la escena de juego junto con sus elementos sonoros acoplados como componentes referenciados.

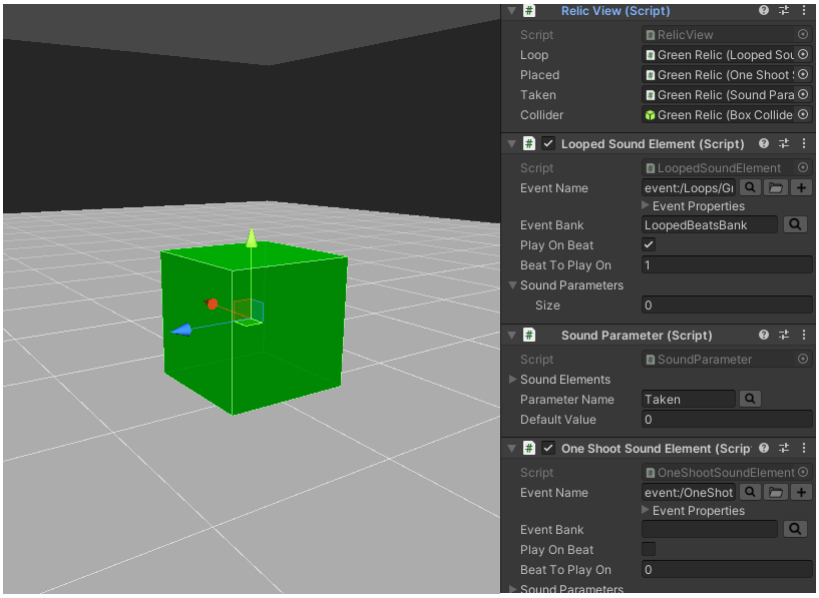

Fig. 7.13. Apariencia y organización del monasterio el editor.

Fuente: Elaboración propia

A continuación, en el diagrama 7.14, se describe la estructura relacionada con la zona segura o monasterio encapsulada en el dominio *Reliquary.Hub*.

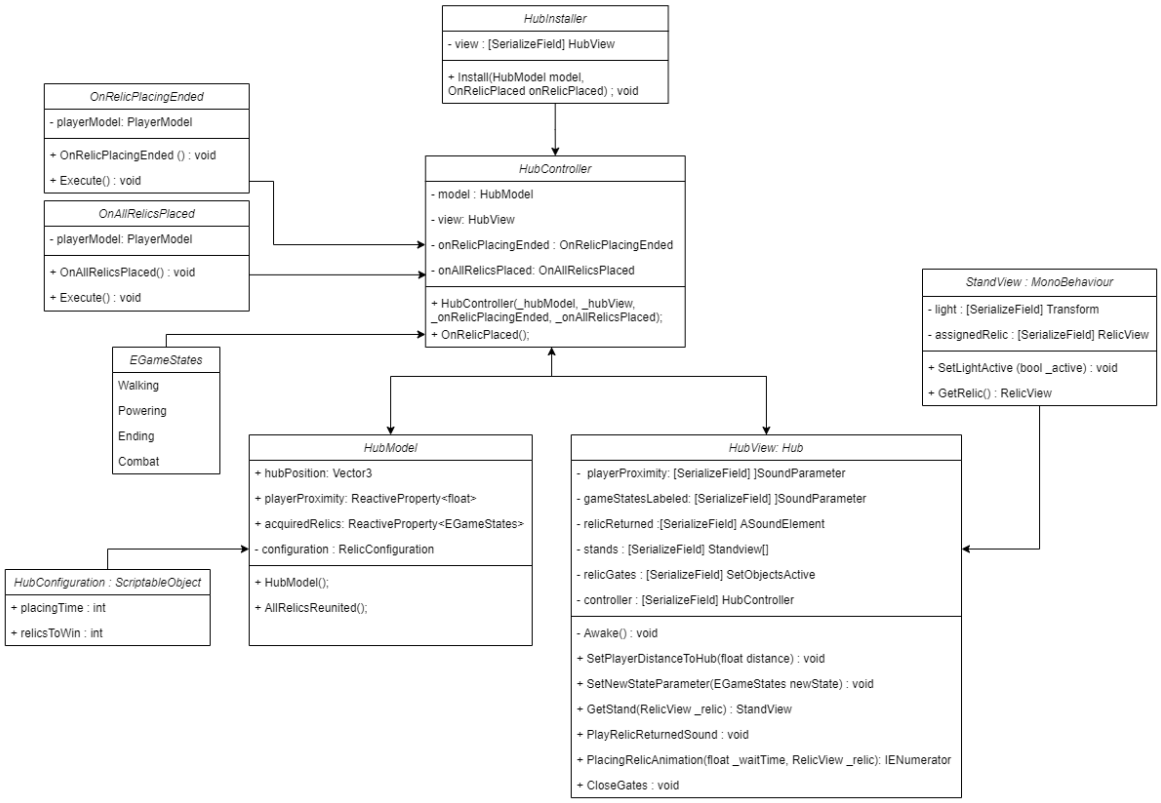

Fig. 7.14. Diagrama de clases del dominio del monasterio, *Reliquiary.Hub*.

Fuente: Elaboración propia

Al igual que la estructura del jugador, la estructura del monasterio se inicializa una única vez. Por un lado, la estructura guarda la distancia respecto al jugador y modifica el parámetro que varía la orquestración musical del juego mediante el uso del método *SetPlayerDistanceToHub*

Esta también gestiona las condiciones de victoria para el jugador, guardando el número de reliquias necesarias para ganar el juego, *relicsToWin*, y llevando cuenta de las ya conseguidas como propiedad reactiva, *acquiredRelics*. Esta se observa desde el controlador, el cual inicia la corrutina *PlacingRelicAnimation* y cambia el estado de juego a *powering*, lo cual varía el bucle musical del juego en el método de vista *SetNewStateParameter*.

Una vez terminado el tiempo en el que la nueva reliquia tarda en situarse en su pilar del monasterio, la corrutina llama de vuelta al método *OnRelicPlaced*. También modifica la vista del pilar para mostrar la reliquia adquirida correspondiente en esta.

Este vuelve a cambiar el estado del bucle a *walking*, la variación normal. En caso de que todas las reliquias hayan sido reunidas, el método modifica el bucle al estado *ending*.

En la figura 7.15, se observa la apariencia del monasterio en la escena. Este contiene cuatro salidas, una que conduce al punto donde aparece el monje por primera vez y una por cada laberinto. A la entrada de cada salida se encuentra el *stand* o pilar pertinente a la reliquia conseguida en el laberinto. El monasterio contiene también una puerta por cada salida, que cierra el paso del jugador una vez terminada la partida.

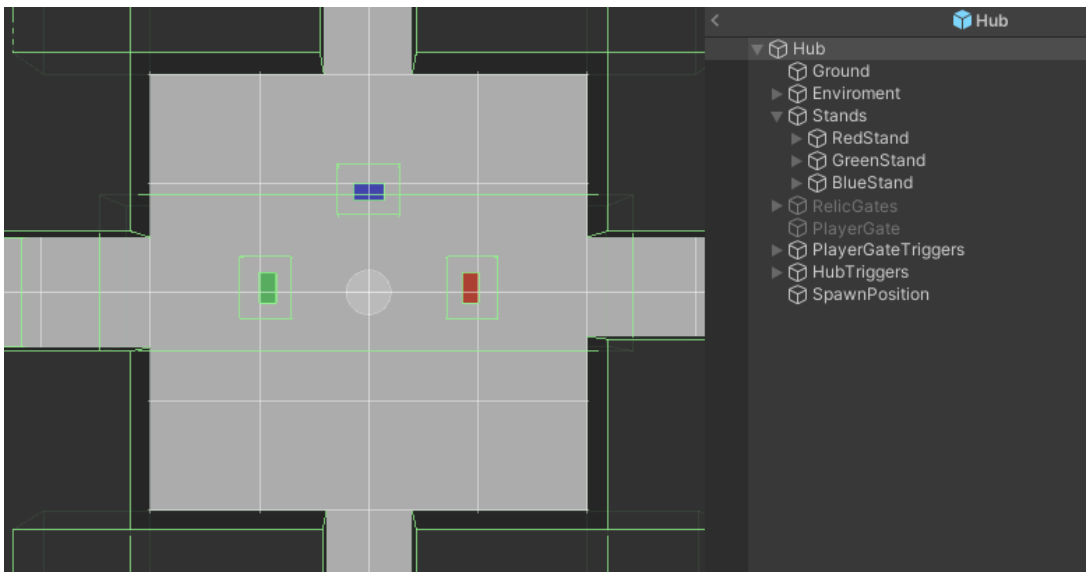

Fig. 7.15. Apariencia y organización del monasterio el editor.

Fuente: Elaboración propia

Seguidamente, se describe la estructura relacionada con el funcionamiento del enemigo o acechador, encapsulada en el dominio *Reliquary.Stalker*, descrita visualmente en el diagrama 7.16

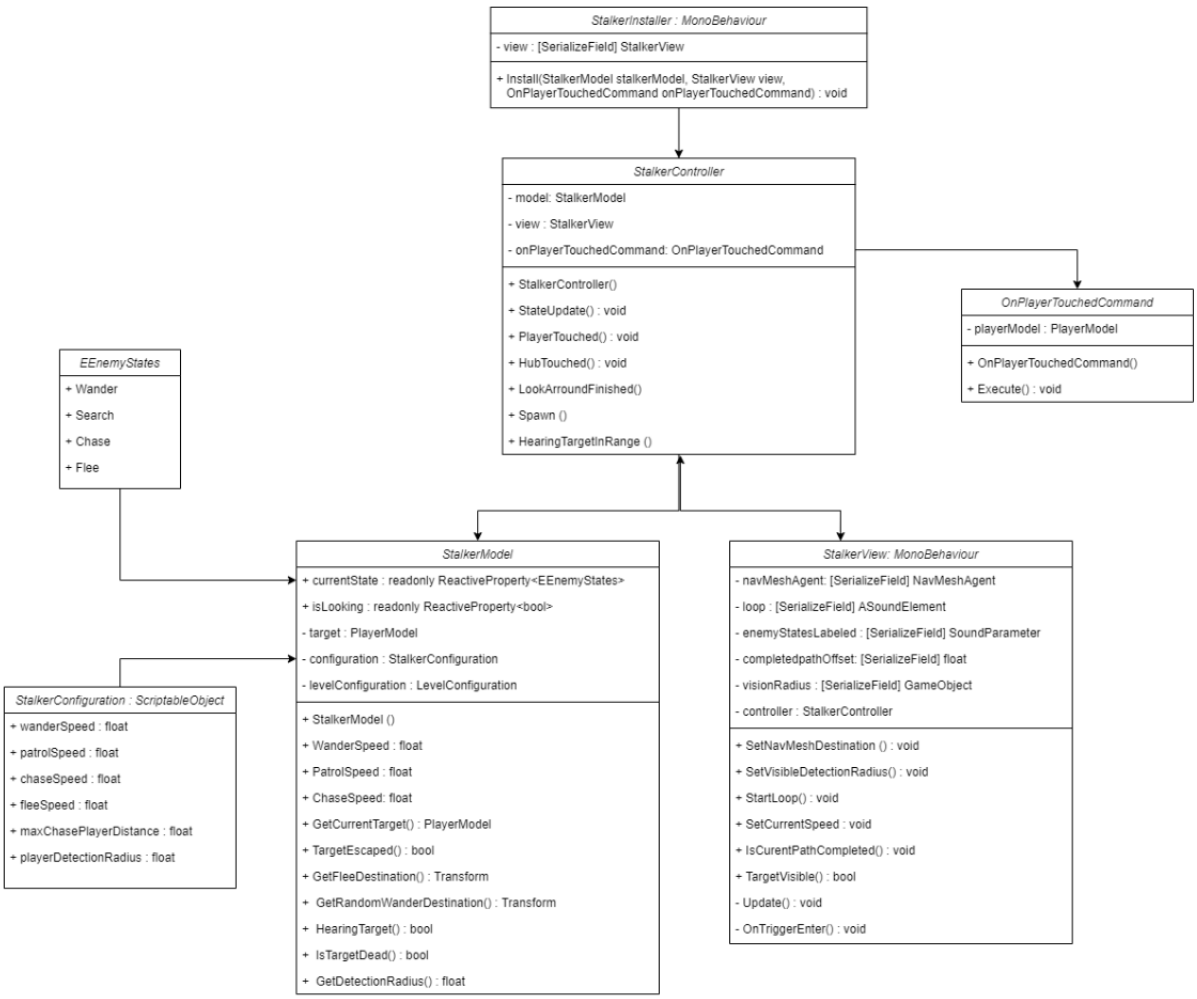

Fig. 7.16. Diagrama de clases del dominio del acechador, *Reliquary.Stalker*.

Fuente: Elaboración propia

Esta estructura gestiona la inteligencia artificial del enemigo, mediante el uso de una máquina de estados, que administra los diversos comportamientos, y el componente *NavMeshAgent*, que automatiza el movimiento y la búsqueda de ruta en la escena de juego. También gestiona las interacciones del enemigo con el jugador.

A diferencia de otros dominios que son iniciados por el instalador general de la escena, la estructura del enemigo se inicializa a través de la estructura del nivel.

Al iniciar el controlador, este observa activamente los cambios en la propiedad *currentState* del modelo. Este suscribe una máquina de estados que modifica el comportamiento del enemigo y se ejecuta tan solo al modificar la variable. Posteriormente, mediante el uso del mismo atributo reactivo, que contiene el estado actual, el controlador gestiona en cada fotograma el comportamiento a seguir el método *StateUpdate*. De manera similar a la gestión del movimiento del usuario, este método es llamado por la función *Update* de la vista, para funcionar acorde con el tiempo de la escena. Ambas máquinas de estados utilizan una expresión de tipo *Switch* para encapsular las condiciones y llamadas a propiedades de la vista y el modelo.

Tal como se ha expuesto, el movimiento del acechador se basa en rutas calculadas automáticamente utilizando el componente *NavMeshAgent*. Este componente, acoplado a la vista del enemigo, utiliza una malla de navegación creada en *Unity* que determina las áreas por las que los agentes pueden caminar en la escena de juego.

Dependiendo del estado del enemigo, el controlador suministra un destino al *NavMeshAgent* utilizando el método *SetNavMeshAgentDestination* de la vista. Una vez adquiere el destino, el *NavMeshAgent* calcula la ruta y mueve el objeto al destino. El controlador comprueba si el destino ha sido alcanzado mediante el uso del método de la vista *IsCurrentPathCompleted*, que comprueba si el componente tiene una ruta pendiente o si la distancia entre la vista y el destino es menor al atributo *completePathOffset*.

En la figura 7.17 se muestra la malla de navegación que los acechadores utilizan para moverse por el nivel. También se la configuración del agente que se utiliza para crear esta malla, Esta determina la libertad de movimiento por diferentes terrenos del agente. Las zonas navegables se muestran en azul claro. Por otro lado, las paredes y zonas por las que el acechador no puede navegar se muestran en azul oscuro.

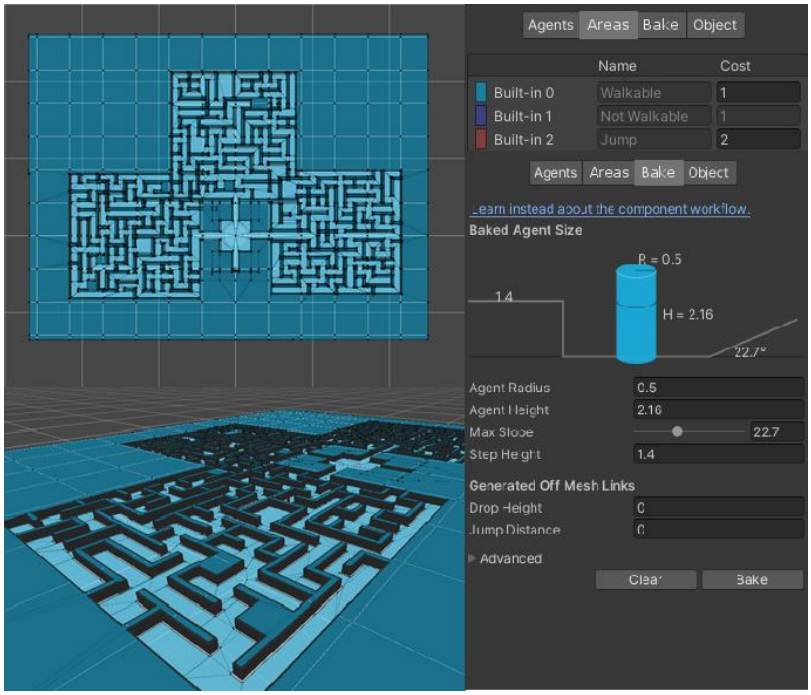

Fig. 7.17. Apariencia del *Navigation Mesh* de la escena completa de juego.

Fuente: Elaboración propia

El destino que sigue el acechador es determinado por el estado actual. Al ser activados, todos los acechadores comienzan en estado de patrulla o *wander.* En ese caso, el controlador adquiere un destino aleatorio del modelo utilizando el método *GetRandomWanderDestination* del modelo. En caso de huir del monasterio, estado *flee*, el controlador extrae una posición fija mediante el método *GetFleeDestination*, que hace que el acechador huya a un punto alejado del laberinto. Estas posiciones se declaran en el instalador del nivel, que las guarda en su configuración. Para los casos *chase y search* en los que se ha detectado al jugador, el modelo guarda una referencia de este, llamada *target*, en el modelo.

Esta referencia, extraída utilizando el método *GetAvatar* del modelo, se utiliza también para determinar si el jugador se encuentra en el rango de detección del enemigo. Esto ocurre en el método *TargetInRange* del controlador.

Al entrar en el nuevo estado, también se varía la velocidad del *NavMeshAgent*, mediante el método *SetCurrentSpeed* de la vista. Las diferentes velocidades se extraen del modelo y se declaran en la configuración, *StalkerConfiguration.*

Desde la vista, el acechador gestiona las colisiones con dos elementos del juego mediante el método *OnTriggerEnter* en la vista. En caso de colisionar con un *trigger* del monasterio, el acechador llama al método *HubTouched*, que cambia el estado del enemigo a huir. En caso de colisionar con el jugador, se llama al método del controlador *PlayerTouched*.

Finalmente, el acechador también gestiona su elemento sonoro. De manera similar al bucle sonoro principal, se trata de un bucle sonoro anidado que varía según el parámetro *enemyStatesLabeled*. Este se modifica al llamar al método *SetNewState*, y varía acorde con el estado actual del enemigo. En la figura 7.17 se observa como figura una instancia del objeto acechador en escena, así como la organización de sus componentes, incluyendo el agente de navegación, en el inspector.

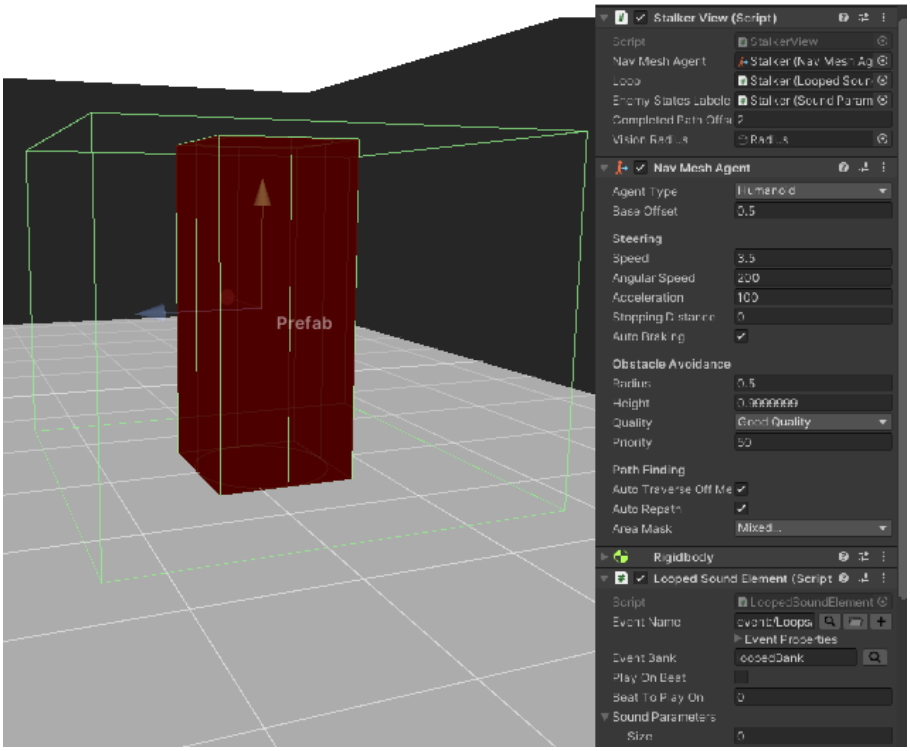

Fig. 7.18. Apariencia y organización del acechador en el editor.

Fuente: Elaboración propia

Finalmente, en la figura 7.13 se expone la estructura relacionada con la zona segura o monasterio encapsulada en el dominio *Reliquary.Level*, encargada de encapsular las propiedades relacionadas con uno de los niveles del juego.

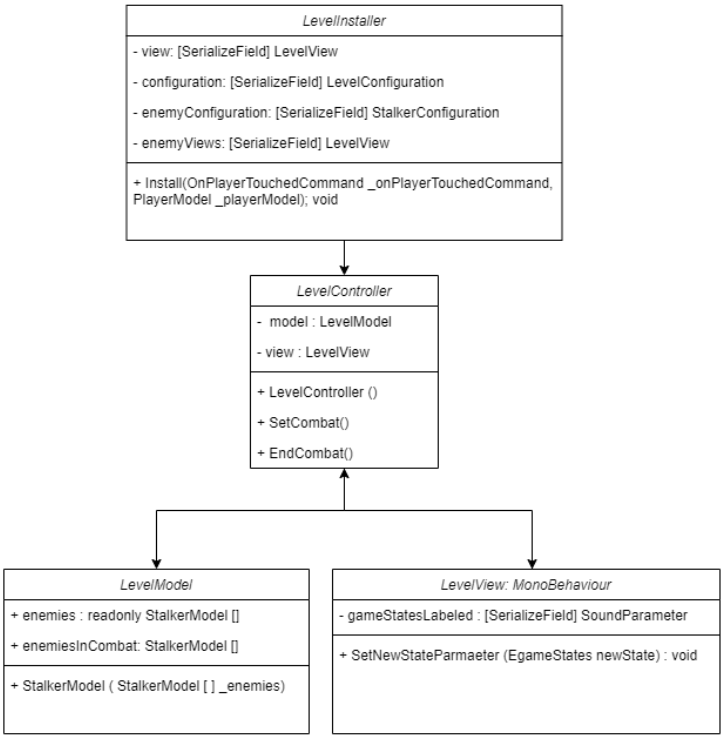

Fig. 7.19. Diagrama de clases del dominio del nivel, *Reliquary.Level*.

Fuente: Elaboración propia

Esta estructura es responsable de gestionar e inicializar los eventos que implican a los elementos de juego de un laberinto. Entre estos se encuentran los enemigos del laberinto y la reliquia correspondiente que se encuentran dentro del laberinto.

Por un lado, el instalador del nivel se encarga de inicializar cada uno de los enemigos del nivel, referenciados en el inspector, y de configurar sus modelos, referenciando el modelo del jugador y declarar las posiciones guardadas en *LevelConfiguration*.

El controlador observa la propiedad *currentState* de cada enemigo. Cuando el estado de alguno de los enemigos cambia a *chase*, se llama al método *SetCombat*. En este, el nivel accede al parámetro sonoro *gameStatesLabeled* y activa el modo *combat*, cambiando así la orquestación del bucle musical a instrumentos de acción.

Cuando un enemigo deja de perseguir al jugador, se llama al método *EndCombat*. Dentro de este, se comprueba si hay algún otro enemigo del nivel sigue persiguiendo al jugador. En caso contrario, se devuelve al bucle al estado normal.

En la figura 7.20 se presenta uno de los laberintos presentes en la escena de juego, la organización de sus elementos y de sus componentes en el inspector.

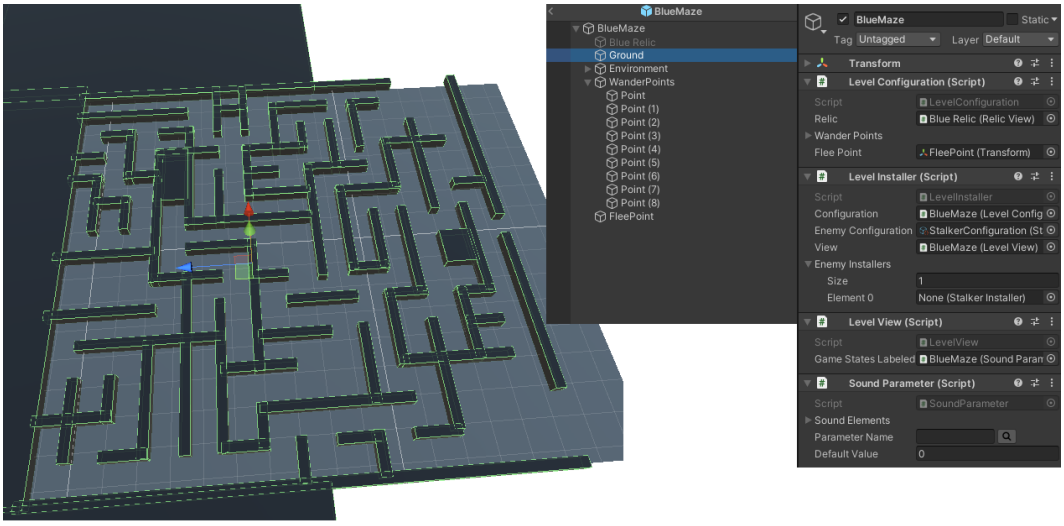

Fig. 7.20. Apariencia y organización del nivel en la escena de *Unity*.

Fuente: Elaboración propia

Analizando la implementación de las mecánicas de juego, el uso de una arquitectura MVC presenta beneficios e inconvenientes.

Por un lado, tal como se ha descrito, el uso de una arquitectura de este tipo resulta beneficiosa para generar interfaces que segreguen las responsabilidades y encapsulen el código de manera cómoda e intuitiva. La segregación de interfaces beneficia también en la organización de la escena. Esto facilita la iteración y extensión del código de forma modular. En la implementación de los elementos de juego, el patrón MVC ha presentado inconvenientes a la hora de agilizar el proceso de creación inicial de las mecánicas.

Esto demuestra que no es una arquitectura que se recomienda para implementar proyectos pequeños, ya que ralentiza el primer proceso de prueba de las mecánicas de juego. Esto ha causado que el número de mecánicas a implementar en cada elemento de juego haya sido limitado.

7.2.3 Producción de sonidos

Con el propósito de crear una serie de elementos sonoros musicales, se lleva a cabo un proceso de composición y producción digital de efectos sonoros. Por ello, cabe hacer una mención al proceso de creación y exportación de los efectos sonoros utilizados en el prototipo, realizado en *ProTools 12* (Avid Technology, 2015).

Por un lado, se encuentra el proceso de creación de los bucles musicales. Este consiste en el uso de instrumentos digitales *MIDI*, utilizando *plugins* como *Xpand2*, para obtener una serie de pistas separadas como si de una canción corta se tratara. Esto se lleva a cabo con el objetivo de obtener los instrumentos separados, pero coordinados en armonía ritmo y duración. Este proceso también se aplica a los elementos de tipo *oneshoot*, aunque solo han de coincidir en armonía.

Una vez producidos, los elementos sonoros son exportados por separado en formato *.WAV* para crear los eventos en *Fmod Studio* (Firelight Technologies 1995).

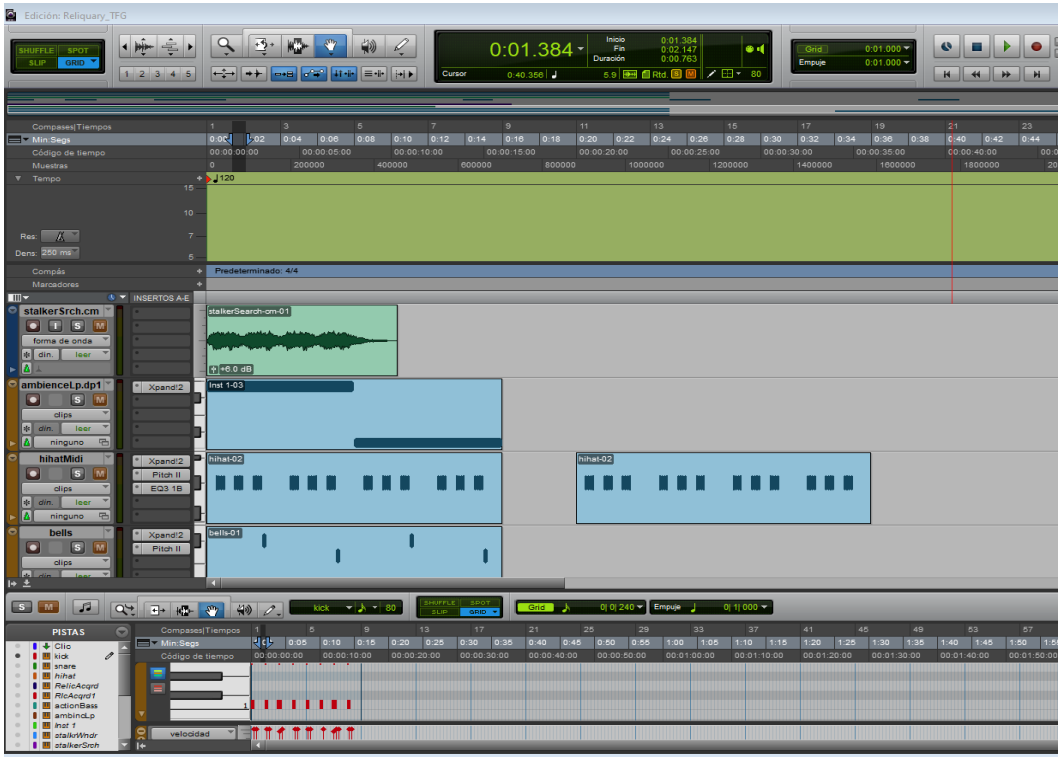

Fig. 7.21. Ventana de edición de la sesión en *ProTools 12*.

Fuente: Elaboración propia

8. Conclusiones

Este trabajo parte de la idea de señalar la importancia del apartado sonoro a la hora de sentar la ambientación en una experiencia interactiva.

El proceso de establecer el marco teórico tiene el objetivo de crear una base que formalice la naturaleza y propósito del sonido dinámico en la experiencia de usuario. Al apoyarse en el libro de Karen Collins (Collins, 2017), se logra establecer una base teórica que exponga la función de dicho apartado. Se logra establecer también una clasificación mediante la cual diferenciar los sonidos de un videojuego, según el propósito que cumplen y el contexto en los que se hallan. El marco teórico recalca la importancia del subapartado musical, responsable por determinar la sensibilidad de la escena en la que se encuentra.

Esto, y la influencia de los referentes del prototipo, contribuye posteriormente en la decisión de implementar diferentes elementos sonoros coordinados musicalmente para mostrar este papel. En el marco teórico también se establece el horizonte de posibles técnicas y herramientas mediante las cuales se puede traer el sonido adaptativo e interactivo al videojuego. Pese a esto, se presenta la ausencia de profundizar en teoría detrás del sonido, que aportaría perspectiva durante la fase de desarrollo.

Con todo, el marco teórico cumple con el objetivo de definir la naturaleza del sonido dinámico y categorizar los componentes de un apartado sonoro. Durante la primera fase de la implementación del prototipo, la categorización establecida sirve como base general para establecer el uso y naturaleza de los diferentes elementos del prototipo. Por lo tanto, se asume el primer objetivo principal como alcanzado.

Respecto a la implementación técnica del apartado sonoro en *Unity*, este ha presentado una serie de conflictos que han limitado la implementación de diferentes tipos de variación sonora.

No obstante, este módulo cumple el propósito de gestionar y operar el apartado sonoro de manera independiente al resto de la estructura del juego, además de permitir gestionar elementos musicales acordes al ritmo. Pese a facilitar la implementación de los elementos sonoros en este proyecto, el modulo tiene funciones limitadas y no se puede considerar el controlador de sonido como una herramienta operacional viable para proyectos más grandes. Por lo tanto, se determina que el objetivo secundario no se ha alcanzado en su totalidad.

Respecto a la implementación de los elementos de juego, durante el proceso de implementación del módulo de sonido se toma la decisión de encapsular el código utilizando una arquitectura de código que organice las clases por responsabilidades. Esta decisión lleva a utilizar la arquitectura MVC con la intención de estructurar las clases y formalizar su documentación, afianzando así la calidad del apartado de desarrollo. Tal como se describe en este, esta decisión afecta a la primera fase de iteración de las mecánicas. Pese a ello, se logra implementar los elementos de juego descritos correctamente, además de obtener una estructura separada por regiones abierta a extensión.

Asimismo, cada elemento de juego utiliza los elementos sonoros especificados en el diseño conceptual. También se implementa en cada apartado las diferentes técnicas de variación descritas, aplicando así diferentes métodos de ambientación dinámica sonora. Finalmente, se completa la implementación del bucle de juego, con la posible condición de victoria del jugador. Por lo tanto, se considera el segundo objetivo como alcanzado.

Más allá de la implementación técnica, cabe argumentar si la ambientación dinámica cumple con el propósito especificado. Por un lado, la música marca gran parte de la ambientación del juego. Esta, mediante el uso de las técnicas de variación, logra sentar el sentimiento adecuado a cada momento del juego. En esta se incluye los bucles musicales de elementos como los de las reliquias, que añaden capas extras satisfactorias para el usuario a medida que el usuario avanza en el juego y consigue devolverlas al monasterio. Pero, debido a no poder variar la forma, los bucles se vuelven repetitivos a medida que avanza el juego, rompiendo así la ambientación que se sienta en un principio.

Una forma sencilla de solventar esta repetición es variar los bucles musicales que se reproducen, o implementar más técnicas de variación que añadan capas de riqueza extra. Debido al tiempo dedicado a implementar las mecánicas durante el desarrollo, no se ha dedicado tiempo en añadir más variación. Aun así, el sistema sienta las bases para añadir detalle y mejorar la ambientación del juego.

En conclusión, se ha conseguido implementar un prototipo con un apartado sonoro dinámico de manera exitosa estableciendo también una base teórica. Pese a las dificultades en la implementación del subapartado musical, se ha conseguido encontrar una solución adecuada para el prototipo, utilizando tanto *Fmod* como *Unity* durante el proceso.

Esta implementación abre la posibilidad a una futura mejora y extensión del prototipo. Esto también se aplica al módulo de controlador de sonido para su uso en otros proyectos. De igual forma, se muestra la capacidad y complejidad del apartado sonoro dinámico y queda demostrada la importancia de su papel en el videojuego.

9. Glosario

.*WAV (Waveform Audio Format):* Formato de audio digital de alta calidad desarrollado por Microsoft.

Enrutar: Acción de dirigir una conexión de software entre dos dispositivos.

Interoperar: Procedimiento o conjunto de procedimientos mediante el cual un sistema posibilita el intercambio de datos con otros sistemas de información.

Modo de codificación: En sonido, tipo de proceso mediante el cual se almacenan y transmiten los datos de audio. Entre estos se encuentran la codificación numérica, la codificación alfabética y la codificación alfanumérica.

Muestreo: En sonido, proceso mediante el cual se digitaliza una señal analógica de sonido.

Scope: Atributo utilizado para describir el que pertenece una propiedad dentro de una interfaz de software.

SOLID: Acrónimo descrito por Robert C Martín utilizado para resumir los cinco principios básicos de la programación orientada a objetos (*Single responsibility, Open-closed, Liskov substitution, Interface segregation* y *Dependency inversion*).

Pattern: Plantilla de diseño técnico destinada a acciones o problemas concretos.

Proxy: Interfaz destinada a realizar de intermediaria en las peticiones de recursos entre dos aplicaciones

Plugin: Extensión de software destinada a añadir funciones extra a una aplicación

10. Referencias

10.1. Bibliografía

Altman, R. (2004). Silent Film Sound. New York: Columbia University Press.

Betts, W. (2019, 04 24). Algorithms, apes and improv: the new world of reactive game music. MusiTech. Retrieved 01 05, 2021, from https://www.musictech.net/features/interviews/ape-out-matt-boch-game-soundtrack/

Booth, J. (2004). "A DirectMusic Case Study for Ashenon's Call 2: The Fallen Kings" In DirectX 9 Audio Exposed: Interactive Audio Development. Todd M. Fay, Scott Selfon, Todor J. Fay.

Butineau, J. (2017, 09 14). What is Horizontal and Vertical Dynamic Music? (Game Music Discussion). Youtube. Retrieved 01 06, 2021, from https://www.youtube.com/watch?v=WLMBnoACpj8

Chion, M. (1994). AudioVision: Sound on Screen. New York: Columbia University Press.

Cohen, A. J. (1999). Functions of music in multimedia: a cognitive approach. S.W. Yi.

Drew, W. (2013, 02). 140 IS A BEAUTIFUL JIGSAW PUZZLE BUT BARELY A GAME. Kill Screen. Retrieved 01 05, 2021, from https://killscreen.com/previously/articles/140-beautiful-jigsaw-puzzle-barely-game/

Elliot Callighan. (2019, 10 09). Eight essential ways to use sound in video games. gamesindustry.biz. Retrieved 01 06, 2021, from https://www.gamesindustry.biz/articles/2019-10-08-eight-ways-to-use-sound-in-video-games

Firelight Technologies (2020). Fmod Studio User Manual, Versión 2.01.

Retrieved from: https://www.fmod.com/resources/documentation-studio?version=2.1&page=welcome-to-fmod-studio.html

Hiebner, G. (2015, 07 18). Review: Wwise For Game Audio. Ask.Audio. Retrieved 01 08, 2021, from https://ask.audio/articles/review-wwise-for-game-audio

Karen Colllins. (2007). Game Sound: An Introduction to the History, Theory and Practice of Videogame Music and Sound Design. The MIT Press.

Klein, M. H., Scott Genshin, Bush, T. W., Adam Boyd, & Sarju Shah. (2007). The Importance of Audio In Gaming: Investing in Next Generation Sound. GDC Vault. Retrieved January 06, 2021, from https://www.gdcvault.com/play/668/The-Importance-of-Audio-In

Nutt, C. (2013, 02 25). Road to the IGF: Jeppe Carlsen and team's 140. Gamasutra. Retrieved 01 04, 2021, from https://www.gamasutra.com/view/news/187261/Road_to_the_IGF_Jeppe_Carlsen_and_teams_140.php

Prout, E. (1889). Harmony: Its Theory and Practice. Retrieved from https://imslp.org/wiki/Harmony:_Its_Theory_and_Practice_(Prout%2C_Ebenezer)

Real Academia Española (2001). Diccionario de la lengua española, 23.ª ed., [versión 23.4 en línea]. Retrieved from: http://www.rae.es/rae.html

Raymond Usher. (2012, 04 18). How Does In-Game Audio Affect Players? Gamastura. Retrieved 06 January, 2021, from https://www.gamasutra.com/view/feature/168731/how_does_ingame_audio_affect_.php

Sanders, P. (11, 03 1987) You’ve got to Have Freedom. *Africa*. Timeless Records.

Webster, A. (2019, 03 08). Ape Out turns raging monkeys into improvisational jazz. The Verge. Retrieved 01 05, 2021, from https://www.theverge.com/2019/3/8/18255975/ape-out-game-pc-nintendo-switch

Yu, J. M. (2016, 06 31). An Examination of Leitmotifs and Their Use to Shape Narrative in UNDERTALE – Part 1 of 2. Retrieved 01 06, 2021, from http://jasonyu.me/undertale-part-1/

10.2. Ludografía

Gabe Cuzzillo. 2019. *Ape Out*. PC, MacOS, Nintendo Switch. (28 de Febrero de 2019).

Carlsen Games. 2013. *140*. PC, MacOS, Linux, PlayStation 4, Xbox One, Wii U, Nintendo Switch. (9 de Febrero de 2020).

2k Boston. 2007. *Bioshock*. PC, Xbox 360, PlayStation 3, MacOS, iOS, Nube, OnLive, Playstation 4, Xbox One, Nintendo Switch. (19 de Agosto de 2007).

Toby Fox. 2015. *Undertale*. PC, MacOS, Linux, PlayStation 4, PlayStation Vita, Nintendo Switch (15 de Septiembre de 2015).

NanaOn-Sha. 1999. *Vib Ribbon*. PlayStation. (9 de Diciembre de 1999).

Dylan Fitterer. 2008. *AudioSurf*. PC. (15 de Febrero de 2008).

Nintendo. 1998. *The Legend Of Zelda Ocarina Of Time*. Nintendo 64. (21 de Noviembre de 1998).

The Game Bakers. 2016. *Furi*. PC, PlayStation 4, Xbox One, Nintendo Switch. (5 de Julio de 2016).

Valve. 2011. *Portal 2*. PC, MacOS, Linux, Xbox360, PlayStation 3. (19 de Abril de 2011).

Nintendo. 2006. *The Legend Of Zelda: Twilight Princess*. GameCube, Wii, Nvidia Shield TV. (19 de Noviembre de 2006).

Turbine Entertainment Software. 2002. *Asheron's Call 2: Fallen Kings*. PC. (22 de Noviembre de 2002).

LucasArts. 1991. *Monkey Island 2: Le Chuck's Revenge*. PC, MacOS, FM Towns, DOS, AmigaOS. (1 de Diciembre de 1991).

LucasArts. 1998. *Grim Fandango*. PC, PlayStation 4, PlayStation Vita, Android, iOS, Nintendo Switch. (30 de Octubre de 1998).

10.3. Sistemas y componentes

Codigames. 2019. *Hotel Empire Tycoon.* Android, iOS. (12 de Diciembre de 2019).

Nintendo. 2006. *Nintendo Wii.* (19 de Noviembre de 2006).

Unity Technologies. 2005. *Unity*. PC, MacOS, Linux. (1 de Junio de 2005).

LucasArts. 1991. *Imuse*.(1 de Enero de 1991)

Firelight Technologies. *Fmod*. PC, MacOS, Linux, Android, iOS, HTML 5. (6 Marzo de 1995).

Epic Games. (2014). *Unreal Engine 4*. PC, MacOS, Linux. (19 de Marzo de 2014).

Jet Brains. 2017. *Jet Brains Rider*. PC, MacOS. (1 de Enero de 2017).

Audiokinetic. (2000). *Wwise*. (1 de Enero de 2000).

Linus Torvalds. 2007. *Git*. (19 de Octubre de 2007).

Avid Technology, 2015. *ProTools 12*. (23 de Marzo de 2015

TREBALL FINAL DE GRAU

Anexos

Implementación de un prototipo basado en la ambientación sonora dinámica

Adrián Núñez Garrido
Tutor: Dr Enric Sesa i Nogueras

Grau en Disseny i Producció de Videojocs

CURS 2020-21

TecnoCampus
Escola Superior Politècnica

Centre adscrit a la

upf. Universitat Pompeu Fabra *Barcelona*

Índice

1. Localización de los elementos 1

1.1 Prototipo desarrollado 1

1.2 Proyecto de Fmod Studio 2.00.07 1

1.3 Repositorio del proyecto de *Unity* 1

1.4 Código documentado del proyecto 1

1. Localización de los elementos

1.1 Prototipo desarrollado

Localización:

https://drive.google.com/drive/folders/1vloO30ooMxOFFjR0VxIPxpdORh6ILcdq

1.2 Proyecto de Fmod Studio 2.00.07

Localización:

https://drive.google.com/drive/folders/1_6Cr1AGi4eQcv5M4MgxVP3YkpwMdBqyb

1.3 Repositorio del proyecto de *Unity*

Localización:

https://github.com/anugaDev/TFG_Prototype_1.git

1.4 Código documentado del proyecto

Localización:

https://anugadev.github.io/TFG_Reliquary_Documentation/annotated.html