



ugr | Universidad
de **Granada**

TRABAJO FIN DE GRADO
INGENIERÍA INFORMÁTICA

Generación procedimental de sonido en videojuegos

Autor

Alberto Pérez Morales

Director

Antonio Bautista Bailón Morillas



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación

—

Granada, septiembre de 2023

Generación procedural de sonido en videojuegos

Alberto Pérez Morales

Palabras clave: videojuegos, generación procedural, música, Unity, SuperCollider.

Resumen

La generación procedural es una práctica que permite a los desarrolladores de videojuegos crear gran cantidad de tipos de contenido distintos para su producto sin necesidad de recurrir a la contratación de artistas, ahorrando así tanto en tiempo como en recursos económicos. También consigue que el contenido generado sea más dinámico y permite que se adapte a lo que ocurre en las partidas, mejorando en gran medida la experiencia del jugador y fomentando la rejugabilidad. El impacto de la generación procedural en el mundo de los videojuegos ha sido tan fuerte que incluso ha provocado la aparición de géneros nuevos que no podrían existir sin ella, como, por ejemplo, el *roguelike*.

De entre todos los tipos de contenido posibles, uno de los menos explorados, y en el cual vamos a centrar este trabajo, es la generación procedural de música. A diferencia de otros, la música es un elemento abstracto en el que emitir juicios en cuanto a la calidad, adecuación o adaptabilidad se complican de forma notable. Además, la música cumple un objetivo concreto dentro del juego, que es servir de acompañamiento para el apartado gráfico y la narrativa, induciendo emociones al jugador, lo cual es muy difícil de lograr de forma efectiva usando música generada de manera procedural.

Para la experimentación práctica de este trabajo, presentamos el proceso seguido para implementar un generador de música para un videojuego, usando para ello el entorno y el lenguaje proporcionados por el software de código abierto SuperCollider y el motor de videojuegos Unity. Para desarrollar dicho generador, en primer lugar, se ha implementado un grafo dirigido de transiciones entre acordes, gracias al cual es posible generar secuencias de acordes con sentido haciendo recorridos aleatorios. Después, se han definido sintetizadores y patrones de SuperCollider para la percusión, armonía y melodías y, finalmente, se han parametrizado ciertas variables del juego, como el número de enemigos o los puntos de salud del jugador, para conseguir una adaptación dinámica al estado de la partida.

Procedural generation of music in video games

Alberto Pérez Morales

Keywords: video games, procedural generation, music, Unity, SuperCollider.

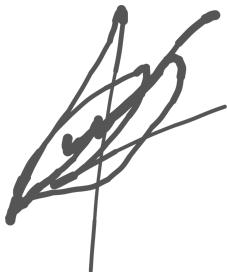
Abstract

Procedural generation is a practice that allows video game developers to create many different types of content for their product without the need to hire artists, thus saving both time and money. It also makes the generated content more dynamic and allows it to adapt to what happens in the games, greatly enhancing the player experience and encouraging replayability. The impact of procedural generation in the video games world has been so strong that it has even led to the emergence of new genres that could not exist without it, such as *roguelike*, for example.

Among all the possible types of content, one of the least explored, and the one on which we will focus this work, is the procedural generation of music. Unlike others, music is an abstract element in which making judgments as to quality, appropriateness or adaptability are notably complicated. In addition, music serves a specific purpose within the game, which is to serve as an accompaniment to the graphics and narrative, inducing emotions to the player, which is very difficult to achieve effectively using procedurally generated music.

For the practical experimentation of this work, we present the process followed to implement a music generator for a video game, using the environment and language provided by the open source software SuperCollider and the video game engine Unity. To develop this generator, first of all, a directed graph of transitions between chords has been implemented, thanks to which it is possible to generate meaningful chord sequences by following random paths. Then, SuperCollider synthesizers and patterns have been defined for percussion, harmony and melodies and, finally, certain game variables have been parameterized, such as the number of enemies or the player's health points, in order to achieve a dynamic adaptation to the game state.

Yo, **Alberto Pérez Morales**, alumno del Grado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicaciones de la Universidad de Granada**, con DNI 26519334N, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.



Fdo: Alberto Pérez Morales

Granada, 7 de septiembre de 2023

D. Antonio Bautista Bailón Morillas, profesor del Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada.

Informa:

Que el presente trabajo, titulado **Generación procedural de sonido en videojuegos**, ha sido realizado bajo su supervisión por **Alberto Pérez Morales**, y autorizo la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expide y firma el presente informe en Granada a 7 de septiembre de 2023.

El director:

Antonio Bautista Bailón Morillas

Agradecimientos

En primer lugar, dar las gracias a mis padres, que, incluso habiéndoles fallado en varias ocasiones, nunca han dejado de creer en mí y apoyarme a su manera. También a mi tía Merche, porque la considero mi segunda madre; con su simpatía y cariño incondicional ha conseguido tranquilizarme en momentos complicados y reconducir mi camino en cada desvío. Y a mis abuelos, especialmente, a Millán, que vivió tiempos no tan buenos y siempre nos recordaba a mí y a mis primos la importancia de estudiar para labrarse un futuro mejor.

Quiero mencionar de forma especial a Laura, mi Poquet, porque desde que la conocí en primero de carrera mi vida ha cambiado para siempre. Ella consiguió lo que nadie antes, poner orden en mi vida y centrarme en las cosas realmente importantes y, por ello, le estaré eternamente agradecido.

Por supuesto, a todo mi grupo de amigos, Los invencibles Peces Gaming, porque con ellos cualquier momento se pasa volando y son el mejor equipo de waterpolo que alguien puede desechar. Y, en especial, a mis hermanos, aunque no de sangre, Antonio, Josan y Luis, porque el tiempo pasa y la gente va y viene, pero ellos siempre han estado ahí.

Por último, agradecer a mi tutor, Antonio, por haber fomentado que dedique este trabajo a un tema que me apasiona, como es la música, y a todos los profesores que me han impartido clase durante el grado. Aunque unos me hayan gustado más que otros, todos me han hecho madurar, tanto personal como profesionalmente.

1. Introducción	9
2. Música y emociones	11
2.1. La música como transmisor de emociones	11
2.1.1. Psicobiología y psicofisiología	11
2.1.2. Inteligencia emocional y conducta musical	13
2.2. Evolución histórica de la música en videojuegos	14
3. Marco teórico sobre generación procedural	19
3.1. Definiciones e ideas generales	19
3.2. Propiedades de las soluciones que usan PCG	21
3.3. Tipos de PCG	22
3.4. Generación procedural de sonido y adaptación a la experiencia del jugador	23
3.4.1. Revisión histórica y acercamientos destacados	25
3.4.2. Estado actual	26
4. Acercamiento práctico	28
4.1. Objetivos	28
4.2. Herramientas utilizadas	28
4.2.1. Unity Engine	29
4.2.2. SuperCollider	33
4.2.3. UnityOSC	41
4.2.4. ChordGenerator.sc	42
4.3. Detalles de implementación	45
4.3.1. GeneradorDeSecuencias.sc	45
4.3.2. ChordGenerator.sc (implementación propia)	51
4.3.3. ManejadorMensajes.sc y adaptación dinámica de la música	53
5. Conclusiones y trabajo futuro	59
5.1. Conclusiones	59
5.2. Trabajo futuro	60
Índice de figuras	61
Sección 2	61
Sección 3	61
Sección 4	61
Bibliografía	63

1. Introducción

La música es una parte integral de la cultura humana, y lo lleva siendo desde antes incluso de los primeros registros históricos. Se han encontrado restos arqueológicos, como, por ejemplo, huesos de mamut perforados a modo de primigenias flautas, sonajeros hechos con huesos o cuernos, que se piensa que podrían haber sido utilizados para emitir sonidos. A través del tiempo, la música ha acompañado al ser humano en muchos ámbitos distintos, desde el simple ocio hasta los conflictos bélicos. Algunas veces usada con fines concretos, como servir de base rítmica para una marcha militar, y otras como elemento relajante o, simplemente, como acompañamiento en tareas de la vida cotidiana.

Dentro del ocio, la música ha sido utilizada desde los inicios del cine mudo para, de alguna forma, dotar a las imágenes de la capacidad de transmitir sentimientos. Desde ese momento han ido apareciendo otros medios audiovisuales, como la televisión o plataformas de vídeo en internet (YouTube, TikTok, etc.). Aunque, de entre todos ellos, uno de los más recientes y exitosos son, sin duda, los videojuegos.

Los videojuegos nacen en la década de 1950, en laboratorios y para un hardware muy específico. Es en la década de los 70, con la aparición de los primeros sistemas domésticos, cuando los videojuegos empiezan a popularizarse, hasta llegar a la actualidad, donde la industria se ha situado a la cabeza de todos los medios audiovisuales y se estima que, durante el año 2021, a nivel global, generó más de 189000 millones de dólares. El problema que tiene la música en el ámbito del videojuego es que, al ser un medio interactivo, se espera que, de alguna forma, se adapte a las decisiones que toma el jugador, dificultando en gran medida la tarea del compositor. A lo largo de los años se han buscado posibles soluciones a este problema, siendo una de las más recientes e interesantes la que estudiaremos en este trabajo: la generación procedural.

La generación procedural es una técnica que permite crear contenido para videojuegos mediante el uso de algoritmos. Cuando hablamos de contenido, hablamos, realmente, de cualquier elemento existente dentro de un videojuego, como niveles, mapas, reglas, texturas, historias, objetos, misiones, o música. El uso de esta técnica tiene gran cantidad de beneficios, como pueden ser reducir la carga de trabajo de los desarrolladores humanos (las máquinas son más rápidas y menos costosas a nivel económico) o crear contenido dinámico que se ajuste en tiempo real a lo que pasa en el videojuego.

Este trabajo se centra en la generación procedural de música y, concretamente, en la implementación de un generador de música para un videojuego, así como la adaptación de esta a la situación de la partida.

Para conseguirlo se utiliza Unity como motor de juego, junto con el entorno de desarrollo y lenguaje de programación SuperCollider. En este último se crean los sonidos mediante la definición de sintetizadores para después ser utilizados en un tipo de funciones especiales de SuperCollider, llamadas patrones, que permiten generar secuencias de sonidos de forma sencilla. Para conseguir una música con sentido, también se ha implementado un generador de secuencias de acordes basado en un grafo dirigido, en el que los nodos representan acordes y las aristas, transiciones posibles. Finalmente, se

decidieron algunas variables del juego, como la salud del jugador o el número de enemigos, para realizar una adaptación de la música generada al estado de las mismas.

La estructura del trabajo es la siguiente. En el capítulo 2, realizamos un estudio de la importancia de la música en general, haciendo hincapié en su capacidad para influir en las emociones humanas. Después, repasamos la historia de los videojuegos, centrándonos en la importancia de la música en su evolución y mencionando algunos de los temas más importantes que fueron apareciendo en la industria.

El capítulo 3 trata la generación procedural desde un punto de vista general. Primero, presentamos algunas definiciones y conceptos básicos, para continuar, después, explicando sus propiedades y taxonomía. Para concluir dicha sección, dedicamos un apartado completo a la generación procedural de música, repasando su evolución histórica, el estado en que se encuentra su uso en la actualidad y argumentando la necesidad de adaptar el resultado a la experiencia del jugador.

En el capítulo 4 está desarrollada toda la explicación del proceso seguido para implementar la parte práctica del trabajo. Comienza puntuizando los objetivos de la misma y analizando las herramientas que se han utilizado para lograrlos. Continúa haciendo un repaso de todas las funcionalidades implementadas en los distintos archivos y, finalmente, se explica el proceso seguido para adaptar la música a la experiencia del jugador.

Por último, en la quinta sección, destacamos las conclusiones extraídas tras la realización del trabajo y proponemos una serie de ideas que se podrían aplicar en el futuro para mejorar los resultados obtenidos.

2. Música y emociones

En este primer apartado, nos centramos en entender por qué es tan importante la música. Intentaremos dar una explicación, desde el ámbito más psicológico, a la capacidad que tiene la música para transmitir emociones. Después, haremos un breve repaso a la evolución histórica de la música en los videojuegos.

2.1. La música como transmisor de emociones

Prácticamente todo el mundo está de acuerdo en que la música es capaz de expresar y transmitir emociones. Lo que casi nadie se plantea es cómo lo hace y qué tipo de emociones es capaz de transmitirnos. Para responder esta pregunta es indispensable intentar comprender cómo funciona la mente humana en este contexto, lo que nos lleva, indudablemente, a adentrarnos en el complejo mundo de la Psicología de la Música.

Como comenta Lacárcel Moreno en [1], la conducta humana es tan compleja y la música está presente en contextos tan distintos que no se puede analizar este problema desde una única perspectiva. Vamos a comentar los dos principales campos de investigación que existen, teniendo en cuenta que no son excluyentes, ya que cada uno aporta resultados interesantes a su forma.

2.1.1. Psicobiología y psicofisiología

Este primer acercamiento busca dar una explicación científica al problema que estamos abordando. Para ello, se centra en el estudio de la evolución del cerebro, ya que es el órgano responsable de recibir la información del mundo exterior (en nuestro caso, el sonido) y procesarla. Desde esta perspectiva, el sonido es una vibración que se transmite en forma de ondas hasta el cerebro y este, al procesarlas, es capaz de distinguir las distintas naturalezas que puede tener (agradable, triste, excitante...).

Este campo también se encarga de estudiar las distintas zonas cerebrales y cómo estas trabajan y se complementan para extraer toda la información de la vibración original. De esta cooperación se pueden obtener ciertas características psicológicas de la música:

- La actividad sensorial de la música se sitúa principalmente en la zona bulbar, donde se encuentra el centro de las reacciones físicas. Aquí juega un papel crucial el ritmo, ya que afecta a la vida fisiológica y con él se tiende a la acción.
- La afectividad se puede localizar en el diencéfalo, ya que es la zona encargada de controlar las emociones. Este es capaz de recibir motivos y melodías y dotarlos de significación, despertando sentimientos y emociones.
- En la zona cortical se sitúa la actividad intelectual, y es aquí donde se procesan las armonías más complejas, ya que necesitan una actividad psíquica y mental más evolucionada y estructurada.

En la imagen siguiente, aparecen representadas las zonas mencionadas. La parte más externa, en un tono rosáceo, se corresponde con la zona cortical (*cerebrum*); el diencéfalo se encuentra en el centro y aparece coloreado en verde (*diencephalon*) y, por último, la zona bulbar está compuesta por el cerebelo y el tronco encefálico, en la parte inferior (*brainstem* y *cerebellum*).

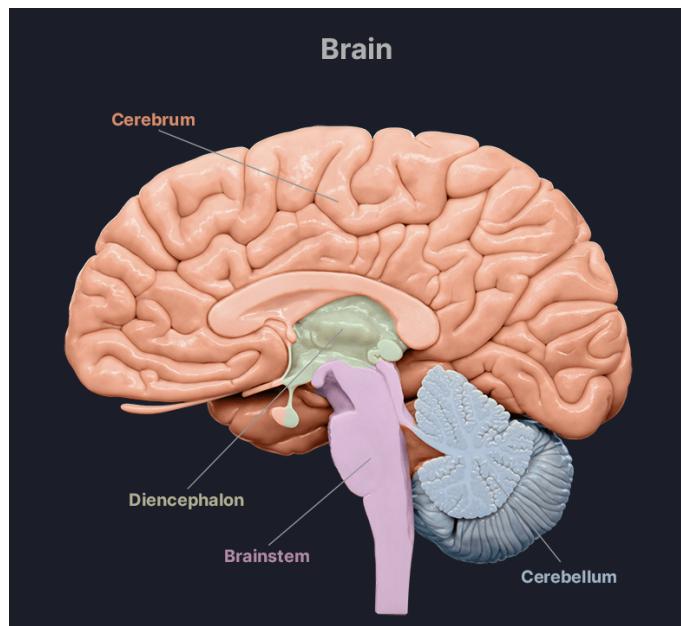


Figura 2.1. Anatomía del cerebro humano. Imagen extraída y adaptada de [2]

Aunque el cerebro actúa como un todo, existen funciones que están situadas en centros, zonas e, incluso, hemisferios completos del cerebro. Existen diversos estudios que han sido capaces de localizar dónde se sitúan algunas funciones cerebrales de la música. Por ejemplo, en la siguiente tabla, podemos ver qué funciones cerebrales se relacionan con cada uno de los hemisferios del cerebro.

Hemisferio izquierdo	Hemisferio derecho
Predominancia de análisis Ideas Lenguaje Matemáticas Preponderancia rítmica (base de los aprendizajes instrumentales) Elaboración de secuencias Mecanismos de ejecución musical Pronunciación de palabras para el canto Representaciones verbales	Predominancia de síntesis Percepción del espacio Percepción de las formas Percepción de la música Emisión melódica no verbal (intervalos, intensidad, duración, etc.) Discriminación del timbre Función vídeo-espacial Intuición musical Imaginación musical

Figura 2.2. Separación de funciones cerebrales de la música en hemisferios. Información extraída y adaptada de [1, p. 216-217]

Vemos que en el hemisferio izquierdo se concentran las funciones musicales más analíticas, mientras que el derecho reúne aquellas más emocionales, visuales y creativas, lo cual concuerda con las competencias que Roger Sperry atribuyó a cada lado del cerebro en [3] y [4].

2.1.2. Inteligencia emocional y conducta musical

Cuando hablamos de inteligencia emocional, nos referimos a un conjunto de habilidades entre las que podemos destacar el control de los impulsos, el entusiasmo, la empatía o la perseverancia. En general, hace referencia a actuar en el mundo teniendo en cuenta las emociones y sentimientos. Las últimas tendencias en psicología meditan sobre si son las emociones la verdadera base de la inteligencia humana. Existen numerosos ejemplos de personas con gran inteligencia que fracasan en su vida por gestionar incorrectamente sus emociones. En cambio, otras personas con una inteligencia promedio son capaces de cosechar mayores éxitos gracias al desarrollo de la inteligencia emocional.

A modo de enlace entre estas ideas y el tema que nos ocupa, Lacárcel Moreno [1, p. 223] explica que “la música, ya sea mediante el comportamiento de interpretación, de escucha o de composición, si ésta es adecuada, nos conduce a una rearmonización del estado de ánimo y de los sentimientos”. Esta capacidad de la música para influir en el ánimo y las emociones se presenta también en [5].

Aunque existen muchos, podemos destacar dos elementos dentro de la música que tienen capacidad para influir en las emociones: el ritmo y la armonía. Según Almansa Martínez [6, p. 4], “Los estímulos rítmicos provocan respuestas psicofisiológicas sobre el ritmo cardíaco, la respiración y el encefalograma, aumentando el rendimiento corporal y el riego sanguíneo cerebral”. Podemos entender, así, que ritmos rápidos nos conducirán hacia sentimientos de agitación o aventura, mientras que aquellos más lentos nos inducirán tranquilidad, tristeza, melancolía, etc.

En cuanto a la armonía, sucede algo parecido: las tonalidades mayores las percibimos en mayor medida como alegres, mientras que las menores son percibidas como tristes. La teoría musical puede explicar este hecho en base a las distancias interválicas de los acordes o conjuntos de notas, es decir, las distancias que hay entre las distintas notas que los conforman. En música, estas distancias se miden en tonos y semitonos y juegan un papel crucial en la sonoridad del acorde. En las escalas menores, sucede que la tercera nota, conocida como mediante, es más cercana a la primera, llamada tónica, que a la quinta, llamada dominante. Por el contrario, en los acordes mayores, la mediante está más cerca de la dominante. Hay que tener en cuenta que la tónica es la nota principal y que más capta nuestra atención, por lo que su menor distancia con la mediante en los acordes menores podría ser la causante de que estos transmitan más tensión.

Todos estos razonamientos nos han hecho entender que no es casualidad que la música sea capaz de influir en nuestras emociones, sino que existen argumentos desde distintos puntos de vista, como la psicología, la fisiología, o, incluso, la teoría musical, que son capaces de explicar este hecho.

2.2. Evolución histórica de la música en videojuegos

Como hemos visto en el apartado anterior, la música tiene la capacidad de transmitir emociones e influir en nuestro estado de ánimo. Además, cuando se aplica con éxito a medios como el cine, la televisión o los videojuegos, somos capaces de almacenar los temas en la memoria. En este apartado, hacemos un repaso por la reciente historia de los videojuegos, centrándonos en la música y los leitmotivs más importantes.

Es muy complicado decidir cuál fue el primer videojuego, pero hay bastante consenso en considerar a *Bertie, the Brain* como tal. Nos remontamos a 1950, cuando los ordenadores eran de gran tamaño y los programas, normalmente, se creaban para un hardware específico [7, 8, 9]. Es en este escenario donde Josef Kates diseñó y construyó *Bertie, the Brain*, una computadora de unos cuatro metros de alto que incorporaba una primitiva IA capaz de jugar al tres en raya. En la misma línea, en 1952, Alexander Sandy Douglas creó OXO para la computadora EDSAC, otro videojuego de tres en raya. En las figuras siguientes se pueden ver dichos juegos y las respectivas computadoras.



Figura 2.3. *Bertie, the Brain*. Imagen extraída de [10]

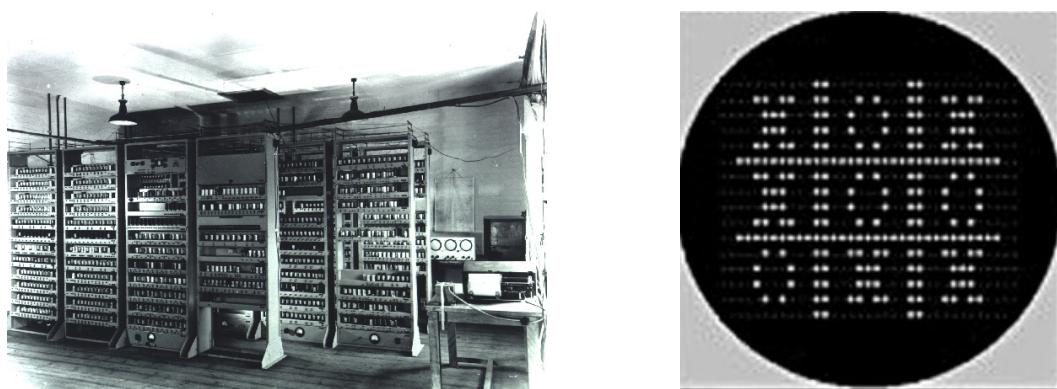


Figura 2.4. (Izda) Computadora EDSAC. (Dcha) OXO. Imágenes extraídas de [10]

En 1958, el físico Willy Higinbotham creó un juego de tenis de mesa interactivo, el cual se mostraba en un osciloscopio (Figura 2.5.). Tres años después, en 1961, Steve Russell, un estudiante del MIT, creó el primer juego de ordenador interactivo para el computador PDP 1, un simulador de combates espaciales. En este punto, la música y los videojuegos empiezan a conectarse [7], gracias a que PDP 1 disponía de un programa

reproductor de música capaz de reproducir armonías de hasta cuatro voces, pero incapaz de crear dinámicas o imitar instrumentos. Aun así, todavía se trataba de una conexión artificial, ya que esta música no formaba parte del juego, sino que el jugador podía elegir alguna pista para escuchar de acompañamiento mientras jugaba.



Figura 2.5. *Tennis for Two*. Imagen extraída de [10]

Para encontrar la verdadera conexión entre sonido y videojuegos debemos avanzar a los años setenta. En esta época, los principales medios de almacenamiento de audio eran cassetes y discos de vinilo, los cuales tenían un elevado coste y fragilidad. Así, los primeros juegos con sonido se basaban en el uso de microprocesadores específicos para la generación de ondas analógicas mediante códigos digitales y el envío de impulsos eléctricos a un altavoz. Este primitivo sistema permitió crear efectos sonoros sencillos o pequeñas melodías monofónicas, típicamente usadas al comienzo o final de la partida. Uno de los primeros juegos que podemos destacar en este contexto es *Pong*, de Atari, en 1972, con sus sonidos para el rebote y la salida de la bola [7, 11]. En 1975, el juego *Gunfight*, de Taito, incorporaba sonidos para los disparos y una conocida melodía al final de los duelos. Aunque podríamos hacer mención a muchos otros, hay que destacar, en 1980, la aparición de *Pac-Man*, el juego de recreativas más popular de la historia, cuya música de inicio de partida y sonidos se han convertido en melodías icónicas de la historia de los videojuegos.

Los microprocesadores fueron mejorando y bajando de coste, lo que permitió la aparición de las primeras consolas domésticas, como Magnavox Odyssey o Atari 2600 (Figura 2.6.a. y 2.6.b.). Estas primeras consolas y las recreativas contemporáneas comenzaron a incorporar chips específicos para la generación de sonido, conocidos como generadores de sonido programables (PSGs) [13]. Estos se encargaban de generar sonido activando los osciladores en función de las órdenes del usuario (usualmente codificadas en ensamblador).

En 1983, Nintendo lanza en Japón la consola de cartuchos Famicom, llamada posteriormente en occidente NES (Nintendo Entertainment System) (Figura 2.6.c.), siendo una de sus principales características un chip de sonido con capacidad para tres canales de música y otro para efectos sonoros. Nintendo se había dado cuenta de que, en una época en la que los juegos eran muy parecidos entre sí a nivel de mecánicas y gráficos, debido a las limitaciones del hardware, las recreativas que más éxito habían tenido eran las que incorporaban mejor apartado sonoro (*Pac-Man*, *Space Invaders*, *Gunfight*, etc). Por este motivo, Nintendo contrató como compositor para el juego *Super Mario Bros* (1985) a Koji

Kondo, en el que fue el primer contrato específico como compositor de música para videojuegos de la historia [13].



Figura 2.6. (a) Consola Magnavox Odyssey. (b) Consola Atari 2600. (c) Consola NES. (d) Consola Mega Drive. (e) Consola SNES. Imágenes extraídas de [12]

No tardaron en aparecer competidores para la NES, pero, entre todos ellos, destacó SEGA, con su consola Mega Drive (Figura 2.6.d.). Una vez más, el factor sonoro tuvo gran relevancia en el éxito, ya que incorporaba dos chips de sonido, uno de cuatro canales y otro de seis, que le permitían reproducir audio en ocho bits. Sin duda, hay que destacar la música de *Sonic the Hedgehog* (1991) como una de las más importantes de este periodo.

La tecnología continuó desarrollándose y, a medida que la memoria iba bajando de precio y evolucionando, el sonido en videojuego fue encaminado hacia el muestreo (*sampling*). Además, la irrupción de programas específicos y el formato MOD facilitaron en gran medida la creación de música a partir de muestras digitalizadas. Un ejemplo de esta evolución lo tenemos en la arcade *Street Fighter II* (1991), con su mítica banda sonora compuesta por Yoko Shimomura.

La continua mejora de los chips de sonido y las memorias también llegó al mundo de las consolas domésticas. En este ámbito, Nintendo volvió a acumular gran éxito con el lanzamiento de SNES (Figura 2.6.e.), en 1991, que contaba con un chip de ocho canales de sonido muestreado con resolución de hasta 16 bits. Algunos títulos destacables de esta época pueden ser *Super The Legend of Zelda: A Link to the Past* (1991), *Mario Kart* (1992) y *Donkey Kong Country* (1994).

Hacia finales de los noventa, la fuerte irrupción de los medios ópticos de almacenamiento, como el CD, junto al inexorable avance en el hardware supusieron una absoluta revolución [7, 13]. En 1994, PlayStation de Sony contaba con una unidad CD-ROM y un chip de sonido con 24 canales y hasta 44.1 kHz, el cual proporcionaba una calidad de sonido idéntica a los reproductores de CD. Existen compositores de gran importancia en la llamada “quinta generación de consolas”, pero, de entre todos ellos, hay que destacar a

Nobuo Uematsu, compositor de la banda sonora de *Final Fantasy VII* (1997), una de las más recordadas y queridas de todos los tiempos. Aunque esta generación fue liderada por Sony, hay que destacar también la Sega Saturn y la Nintendo 64 (Figura 2.7.). A diferencia de otras generaciones pasadas, Nintendo 64, que todavía usaba cartuchos sólidos, no consiguió igualar la calidad de las consolas de la competencia en cuanto al apartado sonoro. Aún así, la calidad de las composiciones de títulos como *Super Mario 64* (1996), *The Legend of Zelda: Ocarina of Time* (1998) y *The Legend of Zelda: majora's mask* (2000), compensó la diferencia técnica y se convirtieron en juegos muy exitosos.



Figura 2.7. (a) Consola Sega Saturn. (b) Consola PlayStation. (c) Consola Nintendo 64. Imágenes extraídas de [12]

Llegamos a la sexta generación de consolas, en la que se encuentra la consola más vendida de la historia, la PlayStation 2. En este punto, los avances tecnológicos hicieron posible que la música en los videojuegos igualara en calidad a otros medios, como el cine. PlayStation 2, por ejemplo, montaba un chip de audio con 48 canales de sonido envolvente 3D y hasta 48KHz. Pero no solo evolucionó el hardware en el apartado de sonido; las consolas de la nueva generación permitieron la creación de juegos 3D mucho más ambiciosos, con mundos cada vez más grandes y abiertos para explorar. Esto hizo que el género de rol y RPG viviera un gran auge y los compositores se vieron obligados a adaptar su trabajo. Cada nueva ubicación, descubrimiento o combate necesitaba una música que acompañara al momento. Son muchos los títulos que podemos nombrar en este período, pero algunos de los más importantes son, sin duda, *Kingdom Hearts I* (2002), *Grand Theft Auto: San Andreas* (2004), *Dragon Quest VIII: El periplo del Rey Maldito* (2004) y *Shadow of the Colossus* (2005).

Una vez las limitaciones de hardware dejaron de ser un problema, las compañías y los compositores empezaron a centrarse en nuevas técnicas que pudieran mejorar la experiencia de los jugadores. Con juegos cada vez más complejos y llenos de posibilidades, componer música que sirviera para todas las situaciones empezaba a ser un gran problema. Es este el comienzo de la exploración de nuevas técnicas, como son la música adaptativa y la música generativa [13].

Se denomina música adaptativa a aquella generada dinámicamente, mediante el uso de distintos mecanismos para, modificando ciertas características de una composición (como pueden ser el ritmo, la armonía o la densidad), adaptarse a situaciones que son impredecibles por los desarrolladores. Por otro lado, la música generativa pretende ir un paso más allá y, en vez de adaptar músicas precompuestas a la situación, busca generar piezas específicamente compuestas para una determinada situación del juego, haciendo uso de herramientas hardware o algoritmos software.

3. Marco teórico sobre generación procedural

En este capítulo, empezaremos explicando qué es la generación procedural de contenido en videojuegos de manera general. A continuación, estudiaremos qué propiedades pueden tener las soluciones que usan PCG y qué tipos podemos distinguir. Seguidamente, entraremos en detalle en la generación procedural de música, dando una definición y algunas características importantes. Justificaremos la necesidad de adaptar la música a lo que ocurre dinámicamente en una partida y haremos una revisión histórica de algunas de las técnicas más usadas y, por último, analizaremos en qué estado se encuentra en la actualidad.

3.1. Definiciones e ideas generales

Según Shaker et al., en [14, p. 1], “la generación procedural de contenido, abreviada como PCG, es la generación mediante el uso de algoritmos de contenido para juegos, con limitado o indirecto aporte del usuario. En definitiva, PCG hace referencia a un software capaz de crear contenido para juegos por sí mismo, o en combinación uno o varios usuarios o diseñadores”. Respecto a lo que se entiende por contenido, Shaker y sus colaboradores indican que este término engloba prácticamente todo lo que encontramos en un juego, desde los niveles, las reglas, las texturas, las historias, las misiones o la música hasta las armas y los vehículos que aparecen. De acuerdo con el autor, a excepción del propio motor de juego y la inteligencia artificial de los NPC, cualquier cosa puede ser considerada contenido.

Aunque en el capítulo anterior ya se mencionaron algunas ideas por las que el uso de generación procedural de contenido está en auge, vamos a indagar un poco más en su importancia. Uno de los primeros motivos que encontramos es que, si una máquina es capaz de generar todas las piezas necesarias para componer un juego, la necesidad de contratar artistas (tanto gráficos como músicos) desaparece. Desde la aparición de los primeros videojuegos, en los que una persona o un pequeño grupo de ellas se repartían todas las tareas, el número de personas necesarias para el desarrollo de un juego realmente exitoso a nivel comercial se ha incrementado mucho. También ha crecido en gran medida el tiempo necesario para llevarlo a cabo, aumentando en gran medida los costes de desarrollo [15, 16, 17, 18]. En la [Figura 3.1](#). puede verse el incremento en el tamaño de los equipos, tiempo de desarrollo y costes de producción de los videojuegos en función de las consolas; podemos ver que aquellas más nuevas, para las que se producen juegos con mayor complejidad, tienen cifras considerablemente más altas. Todo esto puede provocar que disminuya la rentabilidad de los videojuegos, así como el número de desarrolladores. Por tanto, en este contexto, el hecho de poder reemplazar parte de este equipo por algoritmos automáticos y más rápidos puede derivar en menor tiempo de desarrollo y menor coste económico, obteniendo juegos en los que se mantenga la calidad [17]. Esto puede resultar especialmente beneficioso para pequeños equipos de desarrollo que no disponen de los recursos necesarios para afrontar proyectos ambiciosos.

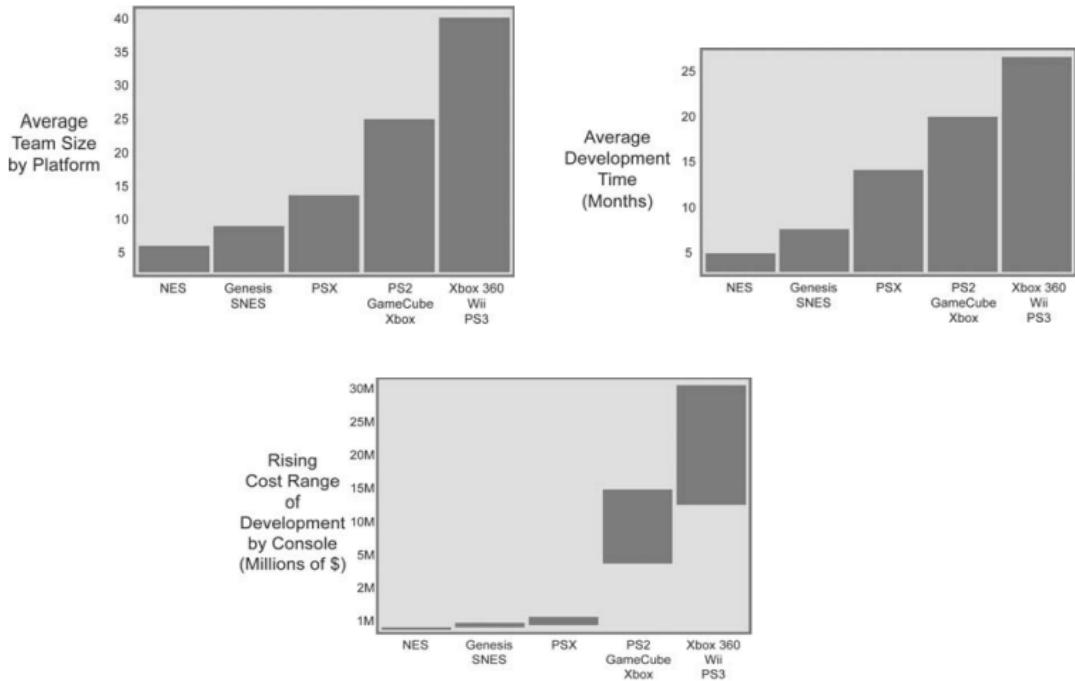


Figura 3.1. (a) Tamaño de los equipos de desarrollo, en función de la consola. (b) Tiempo medio de desarrollo en meses, en función de la consola. (c) Incremento de costes de producción en millones de dólares, en función de la consola. Imágenes extraídas de [15]

Pero no solo está el argumento económico para justificar el uso de PCG, también abre la puerta al diseño de juegos totalmente nuevos que no serían posibles sin el uso de estas herramientas. Juegos como *Spelunky* (2008), *Minecraft* (2009) o *The binding of Isaac* (2011) generan de forma procedural el mapa por el que el jugador debe avanzar en cada partida, dotándolos de un potencial de rejugabilidad infinita. Otros, como *Galactic Arms Race* (2010), basan su mecánica en generar proceduralmente las armas, dependiendo del comportamiento del jugador.

Estas son solo algunas razones para justificar el uso de PCG, pero, como se puede apreciar, el potencial es prácticamente infinito. En la tabla siguiente encontramos un resumen de las ventajas e inconvenientes principales del uso de PCG:

Ventajas	Inconvenientes
<ul style="list-style-type: none"> Permite crear contenido dinámico Puede ahorrar memoria Permite reducir tiempo y costes de desarrollo Da lugar a gran cantidad de opciones Aumenta la posibilidad de jugar de nuevo (rejugabilidad) 	<ul style="list-style-type: none"> Hay cierta falta de control Puede demandar mucho a nivel hardware Puede generar contenido repetitivo Es difícil tener en cuenta determinados eventos del juego Puede generar contenido no útil

Figura 3.2. Ventajas e inconvenientes del uso de la generación procedural de contenido en videojuegos. Información extraída y adaptada de [19]

3.2. Propiedades de las soluciones que usan PCG

Un punto importante a tener en consideración es que la implementación de métodos de generación procedural de contenido no dejan de ser soluciones a un problema, que, en este caso, es el de la creación de las distintas piezas que componen un videojuego. En el ámbito del desarrollo de videojuegos, así como en la mayoría de ámbitos, el encontrar una solución suele implicar tomar decisiones que mejoren alguna propiedad para satisfacer algún requisito, aunque ello conlleve que otra propiedad se vea afectada negativamente. Por ejemplo, si necesitamos minimizar el tiempo de ejecución, normalmente, la calidad del resultado obtenido se suele ver afectada. Teniendo esto en cuenta, Shaker et al. [14, p. 6] distinguen las siguientes propiedades, que consideran “deseables” en cualquier solución PCG:

- **Rapidez.** Hace referencia al tiempo que tarda una solución en generarse. La viabilidad de esta característica dependerá, principalmente, de si la generación se produce durante el desarrollo del juego o de forma dinámica durante la partida.
- **Fiabilidad.** Este término se refiere a la capacidad que tiene un generador para crear contenido que se ajuste a un criterio de calidad, el cual dependerá, en gran medida, del contenido que se esté generando. Por ejemplo, una pieza de música que suene algo extraña puede ser tolerable, pero una mazmorra sin salida resultaría en un juego inacabable..
- **Controlabilidad.** En muchos casos es necesario que el generador de contenido sea manipulable. Por ejemplo, si queremos que la música generada se adapte a lo que está pasando en la partida, el generador debe tomar ciertos datos de la partida y usarlos en el proceso de creación musical.
- **Expresividad y diversidad.** Normalmente, esperamos de un generador que sea capaz de generar contenido variado que diferencie unas creaciones de otras. De esta forma, por ejemplo, no sería tolerable un generador de enemigos en el que no se diferencien unos de otros.
- **Creatividad y credibilidad.** Uno de los objetivos que tienen las técnicas de PCG es que el usuario no perciba, o lo haga de forma mínima, que se están usando. En otras palabras, debemos conseguir que el contenido generado se asemeje al que un desarrollador humano podría crear.

3.3. Tipos de PCG

La gran variedad de contenidos distintos que se pueden generar proceduralmente, junto con los distintos métodos que existen para resolver el problema, hacen que sea realmente complejo establecer una clasificación de PCG. Aun así, vamos a tomar como referencia la taxonomía presentada por Togelius et al. [20]. Es necesario especificar que no se trata de una clasificación rígida, sino que presenta los límites entre los que puede encontrarse una solución PCG.

- **Online versus offline.** Es posible distinguir entre soluciones en las que la generación se produce de forma dinámica, en tiempo de ejecución del juego (online), o durante el desarrollo (offline). Si tomamos como ejemplo la generación de modelos 3D para enemigos, un ejemplo online sería la creación de estos mientras el jugador explora el mapa, de forma que cada enemigo nuevo encontrado fuera distinto. En cambio, un ejemplo offline sería generar distintos modelos durante el desarrollo, que más tarde serán pulidos por artistas 3D, ahorrando así parte del trabajo de modelado. En medio de ambos, podemos encontrar una opción híbrida, que consiste en la creación de modelos al cargar partida.
- **Contenido necesario versus opcional.** El primer tipo de contenido hace referencia a aquél que el jugador necesita para seguir progresando en el juego, mientras que el segundo no ha de jugarse obligatoriamente para continuar. Por ejemplo, el contenido creado por un generador de misiones para la campaña principal de un juego, es contenido necesario. En cambio, las misiones secundarias son contenido opcional. Esta distinción es importante, debido a que el contenido necesario generado siempre debe ser correcto, ya que, en el caso anterior, una misión de la campaña principal imposible de completar bloquearía la partida. Sin embargo, si hablamos de contenido opcional, un error es más tolerable. Por ejemplo, si un jugador encuentra un error en una misión secundaria puede, simplemente, abandonarla y continuar con su aventura.
- **Semillas aleatorias versus vectores de parámetros.** Un punto importante dentro de cualquier algoritmo de PCG es hasta qué punto es parametrizable. En un extremo tenemos algoritmos que generan un número aleatorio y lo utilizan como semilla para generar el contenido. En el otro, tenemos algoritmos que reciben como entrada vectores multidimensionales con valores reales que especifican ciertas propiedades necesarias en la creación del contenido. Por ejemplo, en *Minecraft* todo el mapa está generado con una semilla aleatoria, de forma que si al crear el mapa introduces una semilla conocida puedes obtener un mapa en específico. En cambio, un generador de mazmorras puede tomar como entradas el número de habitaciones, el número de pasillos, el factor de ramificación, etc.
- **Generación estocástica versus determinística.** Hablamos de generación determinista cuando podemos volver a crear un contenido obtenido anteriormente dando los mismos parámetros de entrada. En cambio, en la estocástica, es muy difícil o imposible generar un contenido anterior. Es necesario puntualizar que la semilla del generador de números aleatorios no se cuenta entre los parámetros de entrada, ya que si fuera así, todos los algoritmos serían deterministas.

- **Constructiva versus generar-y-probar.** El primer tipo se refiere a algoritmos que generan el contenido una sola vez para implementarlo directamente en el juego. Normalmente, es necesario incorporar algunas operaciones para comprobar que el contenido generado cumple una cota de calidad mínima. En contraparte, en el segundo tipo, el contenido generado se somete a una prueba de calidad. En caso de no pasarla, el contenido es descartado y regenerado, repitiendo el proceso hasta que se considera lo suficientemente bueno.

Aunque los autores de [20] terminan aquí su taxonomía de PCG, Shaker et al. [14, p. 7] realizan una revisión sobre la misma añadiendo dos clasificaciones nuevas:

- **Genérica versus adaptativa.** Hablamos de PCG genérica si no se tiene en cuenta el comportamiento del jugador a la hora de crear el contenido. En cambio, la generación procedural adaptativa trata de crear un contenido personalizado para cada jugador, teniendo en cuenta la forma en la que cada uno se comporta e interactúa con el juego.
- **Generación automática versus autoría mixta.** Esta es la clasificación más novedosa. Tal como aparece en la columna de inconvenientes de la Figura 3.2., es normal que los diseñadores tengan poca participación o control dentro del algoritmo de generación de contenido, pudiendo únicamente modificar ciertos parámetros de entrada. Sin embargo, en los últimos años se han estado popularizando técnicas mixtas, en las que un jugador o diseñador crea una parte del contenido para luego completarlo usando PCG. Por ejemplo, un diseñador podría dibujar un fragmento del mapa de una escena, para luego dejar que el resto sea completado usando generación procedimental, o viceversa [21, p. 4].

3.4. Generación procedimental de sonido y adaptación a la experiencia del jugador

Como hemos visto en apartados anteriores, prácticamente todos los elementos que componen un juego son susceptibles de ser generados proceduralmente y, como no podía ser de otro modo, la música está entre ellos. Aunque hayamos dado una definición general de PCG, cuando nos centramos en elementos concretos, es importante analizar sus particularidades.

A diferencia de otros elementos, como pueden ser modelados, escenarios, mapas o texturas, la música es un elemento abstracto en el que se complica la toma de decisiones sobre calidad, adecuación o adaptabilidad. Según Collins [22], podemos definir la música procedural como “una composición que evoluciona en tiempo real de acuerdo con un conjunto específico de reglas o lógicas de control”. Aquí encontramos la primera diferencia con respecto al resto de elementos, y es que, ya sea online u offline, por lo general, la mayoría de elementos no se modifican una vez se han terminado de generar. En cambio, en la música, hablamos de una generación continua y dinámica que evoluciona con el avance de la partida.

En [22], Collins distingue entre dos tipos de audio dinámico, el interactivo y el adaptativo.

- **Audio interactivo.** Hace referencia a sonidos activados directamente por el jugador mediante el uso de controles. En este caso, normalmente, no hablamos de música, sino de efectos de sonido tales como pasos, disparos o golpes. De esta forma, no se trata de una composición dinámica como tal, pero el jugador sí interviene de primera mano en la creación de un paisaje sonoro personalizado.
- **Audio adaptativo.** Se refiere a sonidos que no se ven afectados de manera directa por los controles del jugador. En cambio, sí están influenciados indirectamente, ya que están controlados por parámetros del motor del juego, los cuales varían con el transcurso de la partida, dependiendo de las acciones del jugador. Así, el audio puede adaptarse a situaciones y factores como cambios de localización, niveles de dificultad, puntos de salud, etc.

Otro punto importante, ya tratado en capítulos previos, es el tamaño creciente de los videojuegos con el paso del tiempo [23]. Así, si, por ejemplo, tenemos un juego de mundo abierto de gran tamaño, pero en una zona repetimos la misma pieza en bucle, el jugador acabará aburrido y frustrado en un breve período de tiempo. Por este motivo, uno de los principales problemas de la industria del videojuego en la actualidad es, sin duda, la necesidad de que la música reaccione y se adapte de forma dinámica a la experiencia del jugador. Este problema ha llevado a los compositores a tener que crear diversas versiones de la misma pieza (usando distintas intensidades o instrumentaciones, por ejemplo), para que los desarrolladores puedan usar unas u otras, en función de la experiencia del jugador.

Existen muchos motivos que justifican el uso de generación procedural para obtener música, pero, de entre todos, destaca la impredecibilidad. Con esto hacemos referencia a que, al final, el desarrollo de una partida depende del jugador y cada persona puede afrontar un mismo evento de forma distinta. Por ejemplo, para un mismo combate, un jugador podría necesitar tres golpes de espada, otro, quizás, bloquear algún golpe y habrá otro que acabe muriendo. En este contexto, por muchas versiones distintas que tengamos de una pieza, resulta imposible cubrir todos los posibles desenlaces de un evento. Esto, sumado a la experiencia personalizada que comentábamos anteriormente, hace que la música generada de manera procedural sea una opción bastante atractiva [21, p. 10].

Como contraparte, la generación de música introduce un problema en el ámbito de los videojuegos: trabajar junto con la parte visual para crear una narrativa. Esto quiere decir que el sonido en videojuegos no solo enriquece la experiencia, sino que cumple una serie de objetivos, como pueden ser alertar de eventos próximos, notificar recompensas, crear emociones en el jugador (como exploramos en uno de los primeros apartados del trabajo) o servir de leitmotiv. Este tipo de objetivos son mucho más difíciles de cumplir por una música generada proceduralmente que por una creada por un compositor [21, p. 7].

3.4.1. Revisión histórica y acercamientos destacados

A lo largo de los años se han ido explorando nuevos métodos para el diseño de sonido y la composición musical, pero, antes de mencionar algunos de los más importantes, presentamos la división que hacen Wooller et al. [24] sobre algoritmos musicales. Encontramos los tres siguientes:

- **Analíticos.** Se trata de algoritmos que tienden a reducir el tamaño potencial de datos, extrayendo características específicas. Por ejemplo, un algoritmo que toma como entrada un conjunto de acordes y extrae una secuencia con sentido armónico puede considerarse algoritmo analítico.
- **Transformacional.** Este tipo de algoritmos tiende a no causar un gran impacto en el tamaño de los datos o la predisposición musical de la representación, pero sí que puede alterar la información. La predisposición musical, según Wooller et al., mide cómo de bien describe una representación musical las ideas musicales y cuánto facilita el proceso de composición. Por ejemplo, un algoritmo que recibe una melodía y realiza una transposición puede encuadrarse dentro de esta clase.
- **Generativos.** Los algoritmos generativos son aquellos que suelen producir como resultado una representación de datos con mayor predisposición musical que la recibida como entrada. Como consecuencia de esto, el tamaño de los datos aumenta. Por ejemplo, un algoritmo que usa una semilla para producir una secuencia de notas aleatoria se puede considerar generativo.

De acuerdo con Collins [22], debido a determinadas dificultades en el proceso de composición procedural de música, la mayoría de los algoritmos que controlan la música en los videojuegos son transformacionales, en lugar de totalmente generativos. Un ejemplo muy conocido de algoritmo transformacional es lo que observamos, por ejemplo, en la saga de videojuegos iniciada con *Super Mario Bros* (1985), en la que, cuando el jugador se está quedando sin tiempo en un nivel, el tempo de la música aumenta. Otro de los algoritmos transformacionales más utilizados es la música recombinatoria o de forma abierta. En ella, el orden de las distintas frases que componen una composición musical es elegido por el “intérprete”, mientras que otros elementos, como el tempo, se generan de manera algorítmica, teniendo en cuenta, de alguna forma, la interacción del jugador. Ya que en videojuegos no existe el concepto de “intérprete”, el orden mencionado es programado y elegido por el motor del juego. Un ejemplo de este tipo de implementación es la música de la zona “Hyrule field” del juego *The Legend of Zelda: Ocarina of Time* (1998).

Otro algoritmo, también transformacional, consiste en la inclusión de condicionales, bucles, saltos, puntos de entrada y salida y otras sentencias de control dentro de la música. El objetivo de esto es poder hacer uso de estas estructuras durante la ejecución del juego, para cambiar la música en función de lo que esté ocurriendo. Es decir, se colocan “puntos de control” en el código para que, en dichos puntos, se comprueben determinadas condiciones y, dependiendo del resultado, la música continúe de una forma u otra. Por ejemplo, el juego *No One Lives Forever* (2000) se caracteriza por introducir una matriz de transiciones para hacer cambios dinámicamente en la banda sonora en función de parámetros como el número de enemigos [22].

Como se ha comentado anteriormente, la mayoría de algoritmos utilizados en la generación de música son transformacionales, aunque también encontramos algunos ejemplos de tipo generativo. Uno de ellos lo encontramos en el videojuego *Ballblazer* (1984), en el que los pesos de las distintas opciones de los parámetros que generaban la música eran decididos por la máquina en tiempo real. En este caso, aunque el resultado obtenido puede considerarse música, no puede afirmarse que sea una música “interesante”, dado que ciertos patrones tenderán a repetirse [22, p. 11].

Otro de los acercamientos más destacados fue introducido por el tercer juego de la saga *Creatures* (*Creatures 3* (1999)). En él, la música fue grabada en pequeñas partes, conocidas como *samples*, formados por notas o acordes, y estos fueron usados por un algoritmo compositivo controlado por determinados parámetros del juego. Cada criatura tenía su propio conjunto de *samples*, que lo caracterizaban, y la música dependía de las criaturas que estaban interactuando y la localización. Esta misma técnica fue empleada más tarde para la creación de *Spore* (2008).

Por último, con la expansión de los videojuegos multijugador online, se han abierto otras posibilidades de generación procedural de música, ya que cada jugador puede representar un determinado sonido y su mera presencia o ausencia en escena sirve para introducir variaciones en la música de manera dinámica, así como las interacciones entre ellos [22].

3.4.2. Estado actual

En este punto podríamos suponer que el uso de generación procedimental para obtener música es una técnica muy explorada y usada en el diseño de videojuegos dadas todas sus ventajas, pero el escenario real dista bastante de ello. Plans y Morelli [23] comentan que la complejidad de la lógica necesaria para conseguir un buen resultado musical usando PCG entra en conflicto con el presupuesto que las desarrolladoras destinan al apartado musical. Por tanto, a pesar de que una de las ventajas principales de la generación procedural es la reducción de costes, en el caso de la música resulta más barato contratar a un compositor para crear una banda sonora que contratar a programadores de audio para usar PCG.

Además, como comentamos anteriormente, también está el hecho de que la música cumple con determinados objetivos concretos. Por ejemplo, debe ser capaz de transmitir sentimientos y ayudar a la inmersión del jugador en el juego, objetivos que son difícilmente alcanzables cuando se usa música generada proceduralmente.

Asimismo, Collins [22] comenta que otro de los motivos por los que estos métodos no terminan de extenderse es el miedo al retorno a la música MIDI (música generada mediante la conexión de instrumentos musicales electrónicos, ordenadores y controladores). La realidad es que, en la actualidad, el público está acostumbrado a música y sonidos de gran calidad, y hasta encontramos música orquestal dentro de los videojuegos. Por este motivo, aún cuando el formato MIDI ha ganado mucho en calidad desde sus inicios, la mayoría de usuarios todavía sigue teniendo una percepción negativa o de baja calidad de la música generada de esta manera. Esto, sumado al hecho de que los algoritmos PCG requieren de mayor uso de CPU (en un contexto en el que los gráficos se

llevan la mayoría de los recursos), ha llevado a las grandes desarrolladoras a no querer apostar por el uso de estas técnicas.

4. Acercamiento práctico

En este capítulo exponemos el caso práctico realizado para este trabajo, el cual consiste en el desarrollo de un generador procedimental de música para videojuegos, su conexión con un juego y la implementación de un sistema para adaptar de forma dinámica la música generada al estado de la partida. Es importante mencionar que, para la correcta realización y comprensión del trabajo, son necesarios conocimientos generales de teoría musical, los cuales se presentan con más detalle en apartados posteriores. En primer lugar, presentamos los objetivos del trabajo y, a continuación, pasamos a comentar el software utilizado, así como los detalles de implementación.

4.1. Objetivos

Como acabamos de comentar, el objetivo final de este trabajo es la creación de un generador de música procedimental para videojuegos, que sea capaz de trabajar junto a un motor de juego, como, por ejemplo, Unity, para generar y adaptar de forma dinámica la música al estado de una partida. Para lograrlo, se ha desglosado este objetivo principal en los siguientes objetivos parciales:

- Creación de un generador de secuencias de acordes con sentido armónico, es decir, conjuntos de notas que, al escucharlos seguidos, no resulten extraños o desagradables al oído.
- Implementación de un generador de música procedimental que utilice el generador de secuencias. Dicho generador lo dividimos en 3 capas:
 - Generador de percusión.
 - Generador de armonía.
 - Generador de melodías.
- Adaptación del código de un pequeño videojuego desarrollado usando el motor Unity para el envío de mensajes al servidor de SuperCollider.
- Implementación de funciones para adaptar de forma dinámica la música generada.
- Conexión de ciertas variables del juego con las funciones de adaptación.

4.2. Herramientas utilizadas

En este punto explicamos todas las herramientas software empleadas para el desarrollo de este trabajo. Primero, comenzamos con una breve presentación del motor de videojuegos Unity¹, analizando algunas de sus características más relevantes y el motivo de su elección. A continuación, nos centramos en SuperCollider², el lenguaje y entorno utilizado para la parte puramente musical del trabajo. Después, presentamos la API

¹ <https://unity.com/es>

² <https://superollider.github.io/>

empleada para la conexión entre Unity y SuperCollider y, por último, explicaremos un pequeño script de SuperCollider, utilizado para la construcción de múltiples acordes.

4.2.1. Unity Engine

Unity es un motor de videojuegos creado por Unity Technologies, lanzado al mercado en 2005. Entre sus mayores ventajas destacan la capacidad de desarrollar proyectos para gran cantidad de plataformas (Android, Windows, PS4 o Xbox One, entre otros.), su precio y su sencillez. Destaca su precio porque ofrece una licencia totalmente gratuita, con la condición de incorporar al inicio de los juegos una pantalla indicando que el desarrollo se realizó en Unity. Esta licencia es utilizable siempre que el juego no supere los 100000 dólares de beneficio; a partir de ese límite es necesario pasar a licencia Unity Pro, con un coste aproximado de 180 dólares mensuales.

Este motor ofrece todas las herramientas necesarias para el desarrollo de un videojuego, pero, además, dispone de compatibilidad con muchas de las herramientas más importantes de la actualidad para la creación de contenido para juegos. Algunos ejemplos son Blender o 3ds Max, para el modelado y animación 3D, FMOD o Wwise, para la gestión de sonido, o Adobe Photoshop, para gráficos 2D.

La elección de este motor para el desarrollo de este trabajo se debe a varios factores que vamos a destacar a continuación. En primer lugar, se trata del motor empleado desde que comencé en el mundo de la creación de videojuegos y, por tanto, con el que más experiencia he adquirido. Otro factor importante a destacar, es su arquitectura basada en componentes. En otros lenguajes, el tratar de dotar a un elemento ya definido de un comportamiento concreto, dependiendo del caso, puede ser una tarea compleja. En Unity, sin embargo, todos los elementos están compuestos por componentes que los dotan de características y comportamientos. Así, por ejemplo, si se busca que un elemento del juego sea afectado por las físicas, solo es necesario añadirle el componente `Rigidbody 2D` (`Rigidbody`, para juegos 3D). En la siguiente figura podemos ver el ejemplo concreto de adición de este componente a un elemento:

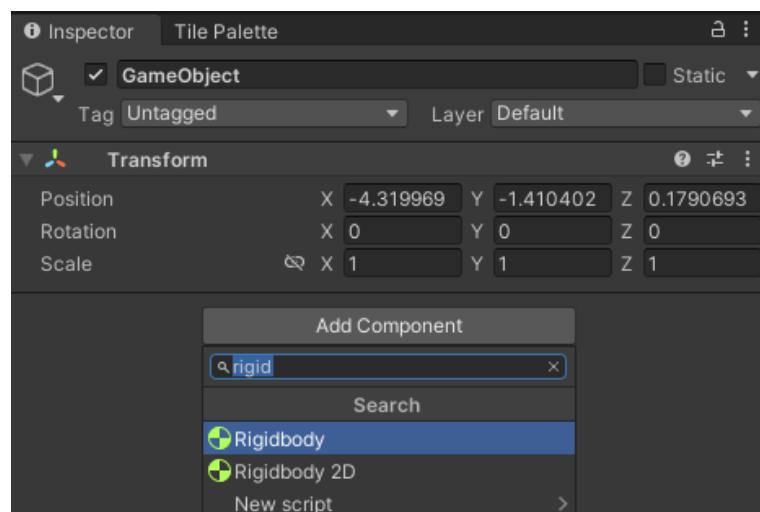


Figura 4.1. Ejemplo de cómo se añade un componente a un elemento en Unity

Una vez añadido, por ejemplo, el componente mencionado (`Rigidbody`), Unity proporciona gran cantidad de opciones para conseguir el comportamiento deseado y, si no existe la opción, siempre es posible programar un script que accede al componente para conseguirlo. En la Figura 4.2., por ejemplo, vemos las opciones `Drag`, que representa la resistencia al aire (en ese caso, el elemento no tendría), `Use Gravity`, que determina si un objeto se ve afectado por la gravedad o no, o `Is Kinematic`, que indica si un elemento debe estar controlado o no por el motor de física.

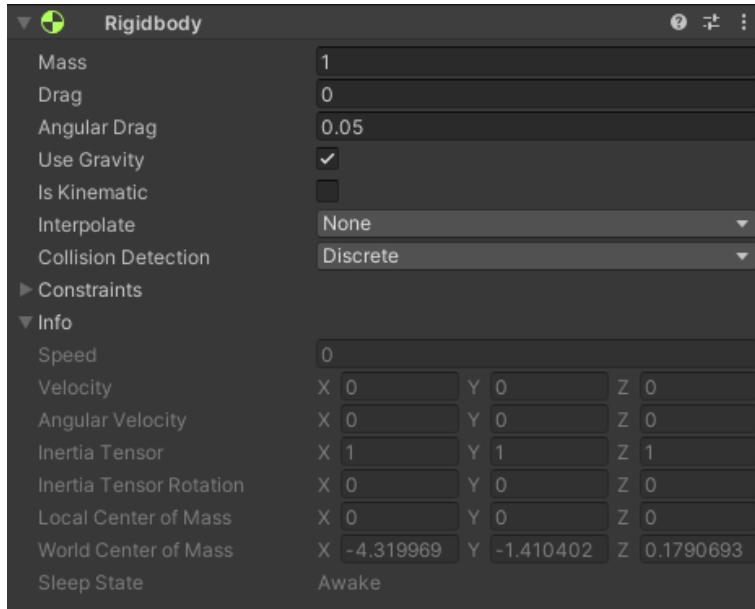


Figura 4.2. Ejemplo de opciones del componente `Rigidbody` en Unity

Los componentes de Unity, a bajo nivel, son scripts de C# que heredan de una clase padre llamada `MonoBehaviour`. Gracias a esta estructura, cualquier componente tiene una serie de atributos y funciones ya definidas, que simplifican en gran medida la tarea del programador. Algunas de las más importantes a destacar son las siguientes [25]:

- **Awake**. Esta función se llama una única vez durante la vida del componente, cuando una instancia de un script habilitado está siendo cargada.
- **Start**. Esta función es llamada, al igual que `Awake`, una sola vez en el ciclo de vida del componente. En cambio, se activa justo en el frame en el que el script se habilita, justo antes de comenzar las llamadas a `Update`.
- **Update**. Función llamada en cada frame, si el componente está activo.

Hemos recalcado estas funciones, aunque existen otras muchas que se pueden utilizar y resultan de gran utilidad a la hora de resolver problemas frecuentes en el desarrollo de un videojuego.

En las Figuras 4.3., 4.4. y 4.5. se puede observar el orden de ejecución de los eventos predefinidos en la clase `Monobehaviour` (la imagen original de la que se han

extraído las figuras puede consultarse en [26]). En la primera (4.3.), vemos las funciones reservadas para la inicialización de variables, entre las que encontramos las mencionadas `Awake` y `Start`, seguidas del ciclo del juego especializado en la programación de componentes físicos. A continuación (4.4.), vemos el ciclo del juego general, más orientado a elementos que no intervienen en cálculos físicos, y es en este punto donde entra en juego la función `Update`. Por último, en la Figura 4.5. aparecen representadas las funciones reservadas para la renderización de gráficos y la creación de la interfaz de usuario, así como para la finalización.

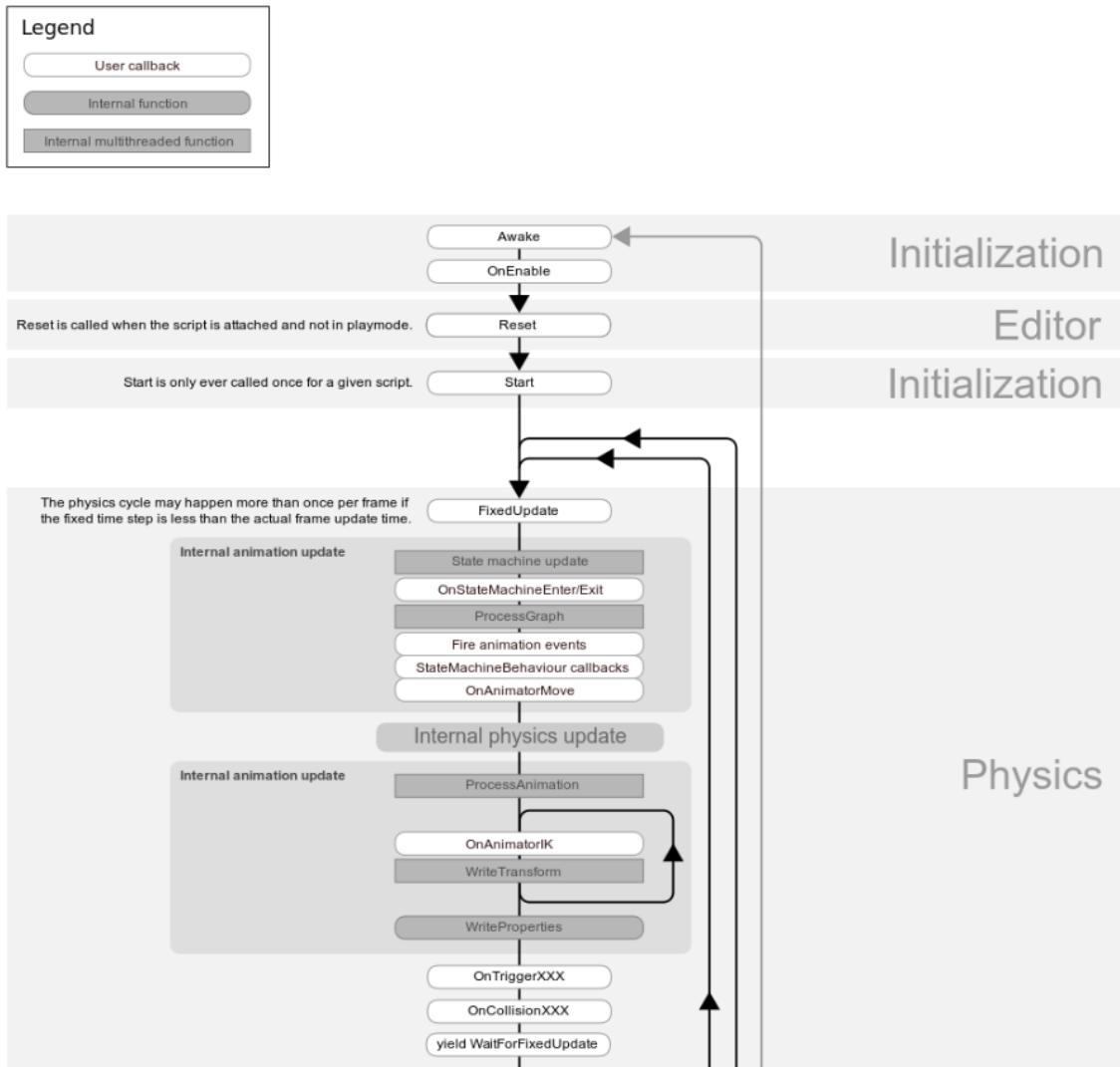


Figura 4.3. Diagrama del ciclo de ejecución de un script de Unity: inicialización y físicas.
Imagen extraída y adaptada de [26]

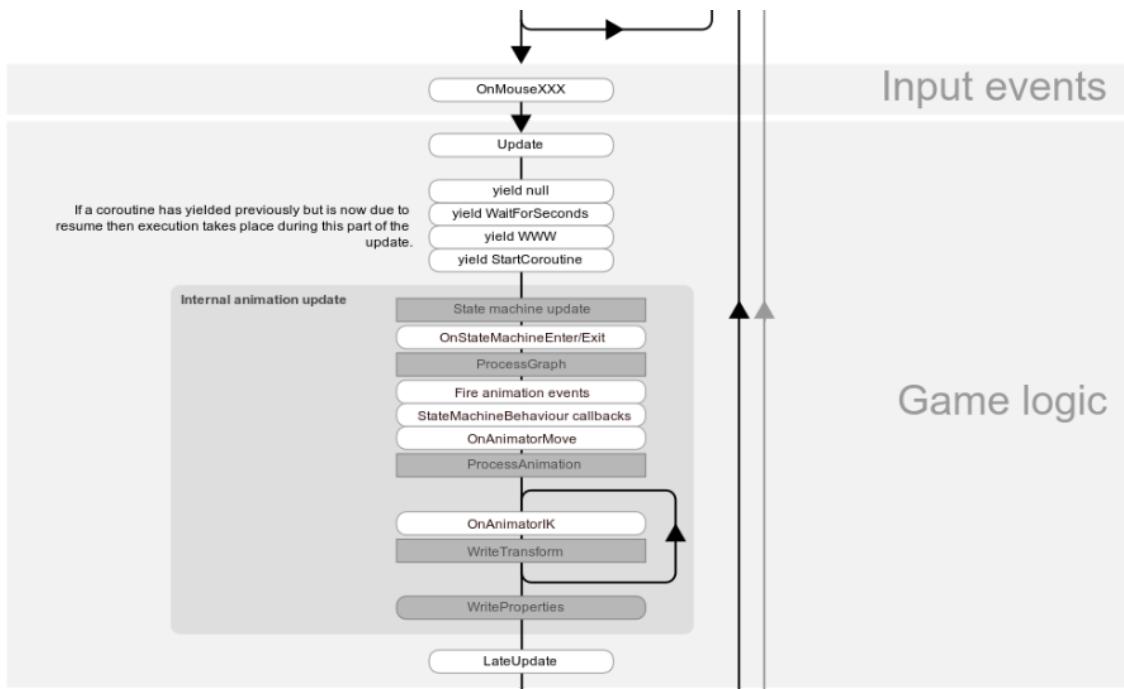


Figura 4.4. Diagrama del ciclo de ejecución de un script de Unity: entradas y lógica del juego. Imagen extraída y adaptada de [26]

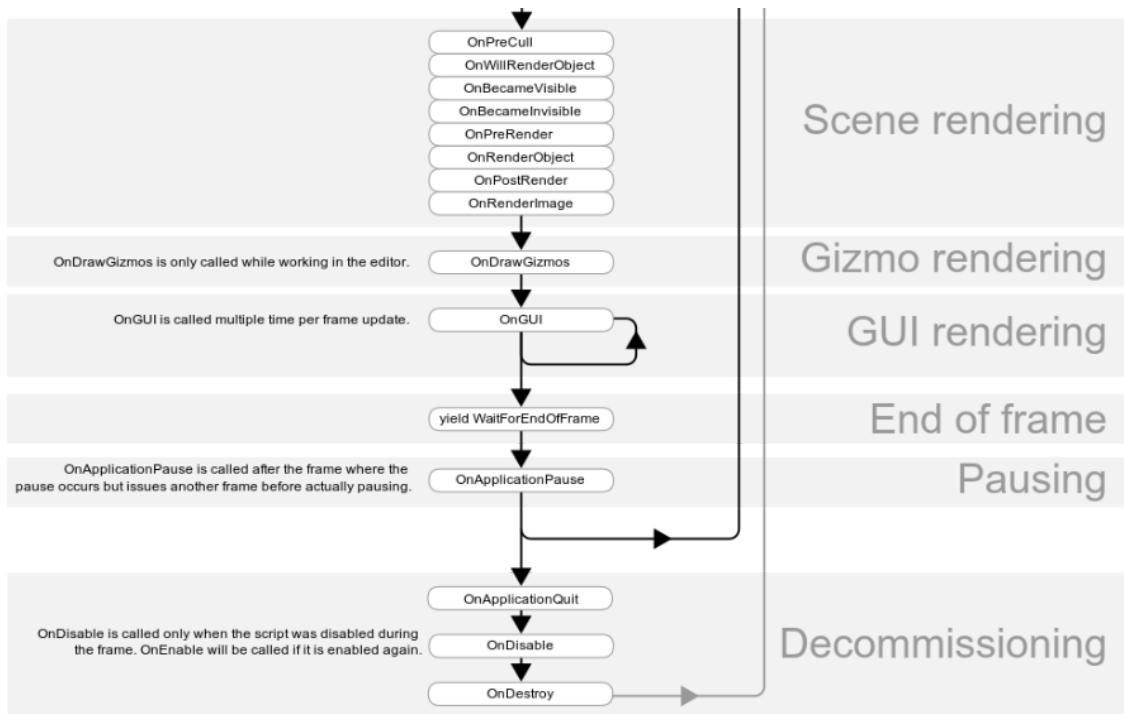


Figura 4.5. Diagrama del ciclo de ejecución de un script de Unity: renderización, interfaz de usuario y salida. Imagen extraída y adaptada de [26]

4.2.2. SuperCollider

SuperCollider es un entorno de trabajo y lenguaje de programación de código abierto, destinado a la síntesis de audio y la composición algorítmica. Desde su creación, en 1996, por James McCartney, y su posterior salida como código abierto en 2002, ha ido creciendo en popularidad hasta convertirse en la actualidad en uno de los lenguajes orientados a música más importantes para artistas, investigadores y programadores.

Está compuesto por cuatro elementos [27]:

- **scsynth.** Servidor de audio en tiempo real que incorpora cientos de generadores unitarios (llamados “Ugens”), destinados al análisis, síntesis y procesamiento de audio.
- **supernova.** Servidor alternativo destinado al soporte de procesamiento de señales digitales (DSP) en paralelo para procesadores con varios núcleos.
- **sclang.** Lenguaje de programación interpretado que controla los servidores.
- **scide.** Entorno de desarrollo para sclang, con sistema de ayuda integrado.

Además, SuperCollider dispone de un gestor de paquetes propio y, al estar desarrollado usando C++, existe la posibilidad de incorporar plugins tanto a scsynth como a supernova mediante APIs escritas en C o C++.

Una vez presentado qué es y de qué elementos está compuesto SuperCollider, pasamos a profundizar en los motivos que han llevado a su elección como herramienta para el proyecto. En primer lugar, está su facilidad para comunicarse con Unity y trabajar juntos para crear y adaptar la música en tiempo real. Esto se consigue gracias al servidor `scsynth`, en el cual se produce la síntesis del sonido, y a un plugin de Unity, que revisaremos en el apartado siguiente, que sirve como interfaz de paso de mensajes OSC (*Open Sound Control*) entre ambos.

Por otra parte, hay que mencionar el lenguaje de programación `sclang`, el cual tiene muchas similitudes con Python. Es interpretado, con variables de tipo dinámico y totalmente orientado a objetos. Gracias a esto, y estando previamente familiarizado con la programación en Python, el proceso de desarrollo se ha simplificado en gran medida y ha sido posible encapsular en clases ciertas funciones necesarias para generación de secuencias de acordes y su posterior uso para generar sonido.

Por último, cabe destacar que SuperCollider es un entorno específicamente diseñado para la creación de sonido y, como tal, pone a nuestra disposición gran variedad de clases, funciones y patrones que facilitan nuestra tarea. Los patrones son una de las herramientas más poderosas que nos proporciona SuperCollider. Estos son, básicamente, rutinas (funciones asíncronas) ya programadas con una funcionalidad concreta y encapsuladas para usarlas de forma rápida. Aparte de la documentación oficial de SuperCollider [28], el aprendizaje sobre este entorno se ha basado, mayormente, en el visionado de los videotutoriales creados por Eli Fieldsteel, una de las personas más importantes y relevantes en la divulgación de este entorno y lenguaje, disponibles en [29].

A continuación, vamos a explicar las funciones y patrones más importantes que se han utilizado para la realización de este trabajo. Primero, veremos algunas funciones simples que generan señales que se pueden utilizar para construir sonidos más complejos. Cada una de estas funciones se acompaña de una representación gráfica, para facilitar la comprensión de su funcionamiento.

- **SinOsc.** Genera una onda sinusoidal usando un oscilador que emplea interpolación lineal sobre una tabla para obtener los valores. Recibe como argumento la frecuencia (en Hercios) y la fase (en radianes).

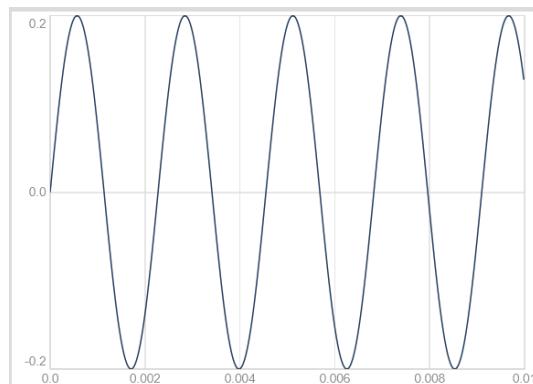


Figura 4.6. SinOsc con frecuencia 440 Hz y fase 0.2 radianes

- **LFSaw**. Genera una onda de sierra usando un oscilador tipo dientes de sierra. Recibe como argumento la frecuencia (en Hercios) y la fase inicial (en radianes).

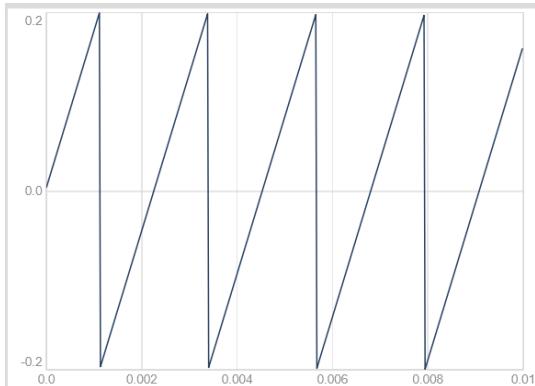


Figura 4.7. LFSaw con frecuencia 440 Hz y fase inicial 0.2

- **WhiteNoise**. Genera ruido aleatorio cuyo espectro tiene igual potencia en cualquier frecuencia. Recibe como argumento el factor por el que se debe multiplicar la salida.

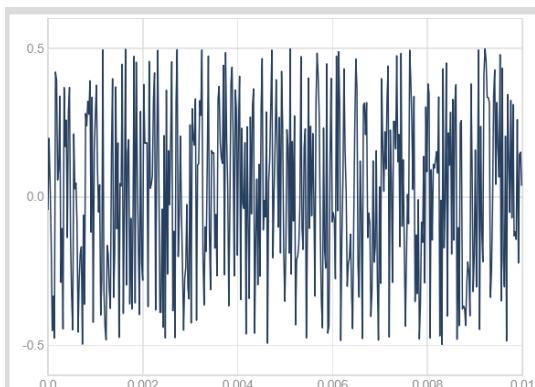


Figura 4.8. WhiteNoise con factor multiplicador 0.5

- **LFNoise1**. Genera valores aleatorios interpolados linealmente con una frecuencia dada por el número entero más cercano al resultado de dividir la frecuencia de muestreo entre un valor de frecuencia recibido como argumento (en Hercios).

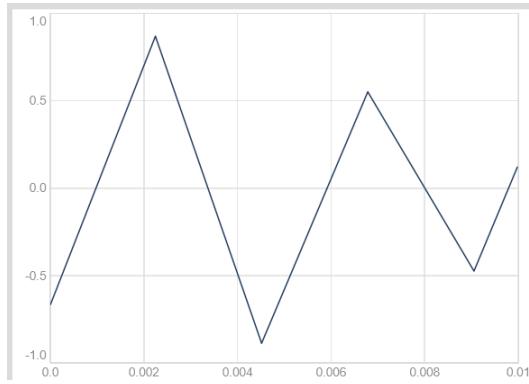


Figura 4.9. `LFNoise1` con frecuencia 440 Hz

- **Pulse.** Genera ondas de pulso con banda limitada. Recibe como argumentos la frecuencia (en Hercios) y la anchura del pulso (valores entre 0 y 1).

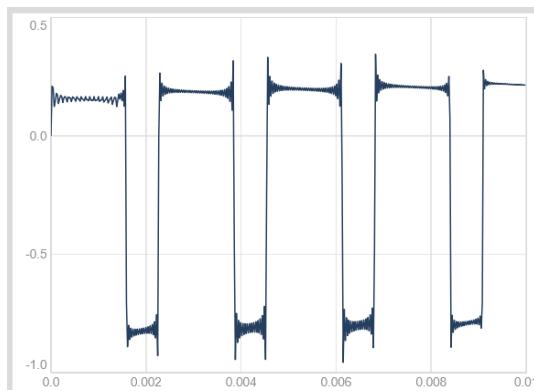


Figura 4.10. `Pulse` con frecuencia 440 Hz y anchura 0.3

- **RLPF.** Filtro de resonancia de paso bajo. Recibe como argumentos la señal a la que aplicar el filtro, la frecuencia de corte (en Hercios) y el recíproco de Q (ancho de banda entre frecuencia de corte).

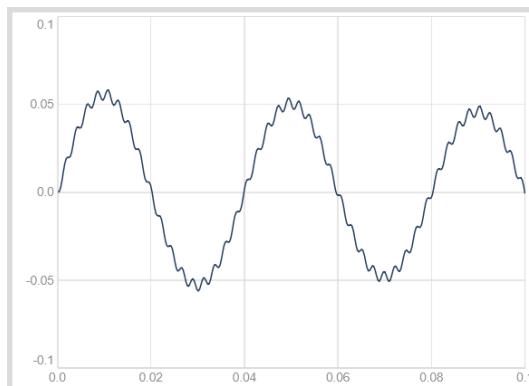


Figura 4.11. `RLPF` sobre `SinOsc(440, 0.2)` con frecuencia 25 Hz y rq 0.03

- **BPF.** Filtro de paso de banda secundario. Recibe como argumentos la señal a la que aplicar el filtro, la frecuencia (en Hercios) y el recíproco de Q.

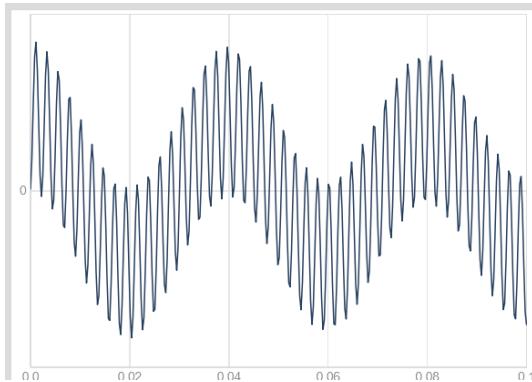


Figura 4.12. BPF sobre SinOsc (440, 0.2) con frecuencia 25 Hz y rq 0.03

- **Resonz.** Filtro de resonancia. Recibe como argumentos la señal a la que aplicar el filtro, la frecuencia (en Hercios) y el valor de ratio de ancho de banda.

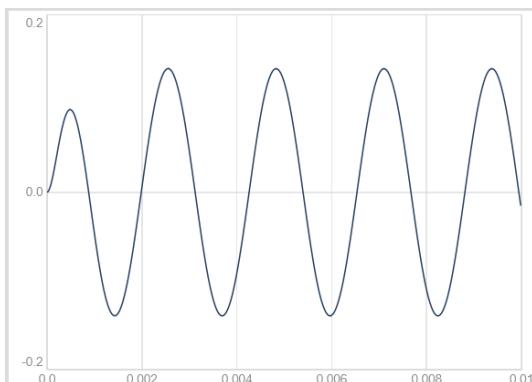


Figura 4.13. Resonz sobre SinOsc (440, 0.2) con frecuencia 440 Hz y bwr 3.0

Aunque existen algunos ejemplos más, se han explicado, exclusivamente, los que han sido utilizados en este proyecto. También es necesario explicar, aunque no son generadores de sonido de forma directa, un tipo de objetos especiales conocidos como "envelopes". Estos se pueden definir como una función que especifica cómo un sonido evoluciona con respecto al tiempo. Sin ellos, los sonidos generados por los sintetizadores serían totalmente planos en el tiempo, sin ningún cambio en la intensidad del sonido. Los envelopes están formados por nodos, cada uno de los cuales representa una intensidad de sonido en un determinado instante de tiempo.

- **Env.** Especificación de un envelope segmentado. Para su definición se necesitan los siguientes argumentos:
 - Array de niveles de intensidad. Su tamaño es el número de nodos y el valor de cada posición indica el nivel de intensidad en un nodo concreto.

- Array de tiempos. Tiene el tamaño del array de niveles menos 1, ya que el primer nodo se supone en el instante inicial, cuyo valor en la posición i representa el instante de tiempo en el que se llega al nivel de intensidad que encontramos en la posición $i + 1$ del array de niveles.
- Array de curvaturas. Esta estructura es similar a la de tiempos, pero, en este caso, cada posición expresa la curvatura de un segmento. La dirección de la curvatura depende del signo de los valores y el valor 0 indica que el segmento es lineal. Si se omite este parámetro, se suponen todos los segmentos lineales.

En la siguiente figura se muestra un ejemplo de definición de un envelope, junto con la gráfica que representa el envelope resultante:

```
//Envelope
(
~envelope = Env.new(
  [0,0.5, 1, 0.6, 0], //niveles
  [0.5, 1, 1, 2], //tiempos
  [0, 3, -5, 0] //Curvaturas
);
)
```

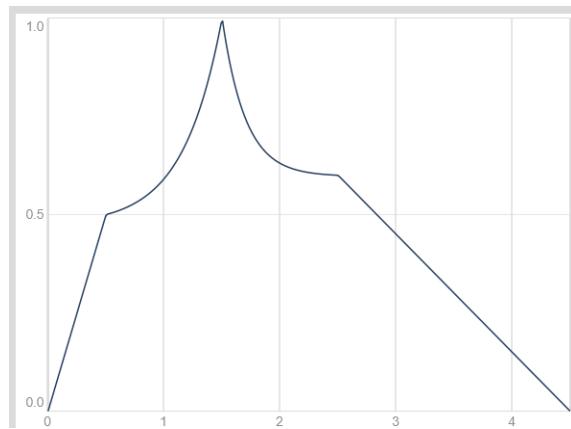


Figura 4.14. (Izda) Definición de un envelope con 5 niveles de intensidad. (Dcha)
Representación gráfica del envelope resultante

- **EnvGen.** Generador de envelopes, los cuales son instancias de la clase `Env`. En la práctica, aplican la función de un objeto `Env` a un sonido. Para su uso pueden emplearse múltiples parámetros, pero, en este proyecto, emplearemos solo dos de ellos:
 - El objeto `Env`, definido previamente, que contiene la función.
 - Un número entero entre 0 y 15, ambos incluidos, llamado `doneAction`, que indica qué debe ocurrir con el sintetizador cuando termine la ejecución del envelope. Por defecto, el valor es 0, lo que indica que no se hace nada. Esto es un problema, puesto que el sintetizador queda en memoria ocupando recursos sin reproducir ningún sonido. Para solucionarlo, haremos uso del valor 2, que indica que han de liberarse los recursos al finalizar.

Una vez introducidas las funciones básicas, podemos entender la definición de sintetizadores, que permiten manipular las señales anteriores para obtener sonidos más interesantes.

- **SynthDef.** Es la representación, a nivel de cliente, de la definición de un sintetizador. Un sintetizador (Synth) es la representación de una unidad que produce un sonido. En la práctica, sirven para crear instrumentos que, más tarde, pueden ser usados en funciones más complejas para crear música. En la Figura 4.15. observamos cómo se puede definir un SynthDef. En primer lugar, debemos definir los argumentos que puede recibir una llamada al sintetizador, incorporando valores por defecto para poder usar solo los necesarios en cada llamada. Justo después, encontramos la definición de variables y su inicialización, que, en la mayoría de casos en este proyecto, suelen ser dos: una para la señal y otra para el envelope. Un poco más abajo encontramos la zona donde se calcula el sonido deseado usando el envelope y algunas funciones simples, que hemos visto previamente. Por último, se pueden aplicar filtros a la señal calculada y se indica por qué canal o bus debe reproducirse el sonido.

```
SynthDef(\retro, {
    arg freq = 440, dur = 0.25, amp = 0.2, filterQ = 0.001, resonanceFreq = 440;

    //Variable para almacenar la señal
    var sig;

    //Definición del envelope
    var env = EnvGen.kr(Env([0, 1, 0], [0.01, 0.4]), doneAction: 2);

    // Calculo del sonido
    sig = LFSaw.ar(freq) * amp; //Sonido básico
    sig = (sig * 16).round / 16; //redondeo para crear sonido parecido a 8 bit

    //aplicación del envelope
    sig = sig * env;

    //Filtro de resonancia
    Resonz.ar(sig, freq * resonanceFreq, filterQ);
    Out.ar(~bus, sig);
}) .add;
```

Figura 4.15. Ejemplo de definición de un sintetizador con SynthDef

Para finalizar esta sección, vamos a explicar algunos patrones que han sido necesarios para la generación de la música. Recordamos que los patrones son funciones asíncronas que ya han sido programadas con una determinada funcionalidad y encapsuladas, de manera que puedan usarse de forma rápida. Es importante destacar que los patrones tienen un conjunto de parámetros ya definidos que permiten controlar su funcionamiento. Además, los patrones que usan sintetizadores para generar sonidos pueden emplear los parámetros propios del patrón y los definidos en el sintetizador. A todos estos parámetros también se los conoce como “claves”, y se escriben usando lo que SuperCollider conoce como “símbolos”, que son cadenas de texto entre comillas simples o con una barra invertida al comienzo. En la Figura 4.15. podemos ver un ejemplo de símbolo (\retro) al comienzo. En la mayoría de casos, SuperCollider utiliza los símbolos para identificar elementos. Los patrones usados en este trabajo son los siguientes:

- **Pseq.** Patrón que itera por una lista de variables. Recibe como argumentos la lista por la que se debe iterar y el número de repeticiones, pudiendo emplear `inf` para infinito.
- **Prand.** Patrón que, en cada ejecución, elige un ítem de una lista de forma aleatoria. Recibe como argumentos la lista en la que elegir el ítem y el número de repeticiones, pudiendo emplear `inf` para infinito.
- **Pwhite.** Patrón que, en cada ejecución, genera un valor aleatorio siguiendo una distribución uniforme. Recibe como argumentos el extremo inferior del intervalo, el extremo superior y el número de valores a generar, pudiendo, como en los casos anteriores, usar `inf` para infinito.
- **Pdup.** Patrón para repetir un patrón un número de veces concreto. Recibe como argumento el número de veces a repetir y el patrón que usar.
- **Pbind.** Patrón que combina varios flujos de patrones en uno conjunto. Cada valor que tiene cualquiera de los flujos es asignado a una o varias claves del flujo resultante, que, a su vez, están determinadas por el comportamiento definido en la clase `Event` y los argumentos definidos en el sintetizador que se va a utilizar. A continuación, repasamos brevemente las claves que han sido utilizadas en este proyecto:
 - `\instrument`. Para indicar el sintetizador a utilizar.
 - `\freq`. Frecuencia del sonido.
 - `\amp`. Amplitud del sonido
 - `\midinote`. Forma alternativa de indicar la frecuencia, esta vez usando notas MIDI.
 - `\degree`. Forma alternativa para indicar la frecuencia, usando grados.
 - `\dur`. Duración del patrón.
 - `\atk`. Tiempo desde el inicio del patrón hasta que se emite sonido.
 - `\rel`. Tiempo que tarda el sonido en detenerse.
 - `\legato`. Grado de legato entre las notas. En teoría musical, decimos que una secuencia de notas están ligadas si se tocan seguidas sin producir ningún silencio entre ellas. De esta forma, este valor indica la separación entre los sonidos.

- **Pbinddef.** Patrón que mantiene una referencia a un Pbind en el que las claves pueden ser reemplazadas mediante el uso de un proxy. Esto posibilita cambios en el sonido de forma dinámica, mediante la alteración del valor de alguna clave. Gracias a este patrón, más tarde podremos adaptar la música en función del estado de la partida. En la Figura 4.16. podemos apreciar cómo se declara un patrón Pbinddef, haciendo uso de las claves comentadas anteriormente.

```
~patronTom = Pbinddef(\patronTom,
    \instrument, \tom,
    \freq, 50,
    \amp, 0.05,
    \dur, Pseq([Pseq([3/24],3), Pseq([1/16],12), 1], inf) * 4,
    \atk, Pwhite(0.001, 0.1, inf),
    \rel, Pwhite(0.01, 0.1, inf),
    \out, 0
);
```

Figura 4.16. Definición de un patrón Pbinddef, con las claves explicadas previamente

4.2.3. UnityOSC

Open Sound Control, o, por sus siglas, OSC, es un protocolo de comunicación entre ordenadores, sintetizadores de sonido y otros dispositivos multimedia, optimizado para las tecnología de redes actual. Su objetivo, en el contexto que nos encontramos, es servir de enlace entre el servidor de SuperCollider y Unity.

Para alcanzar este objetivo, se ha empleado una API de código abierto desarrollada por Jorge García Martín, llamada UnityOSC [30]. Esta consiste en un conjunto de archivos que deben ser introducidos en la carpeta *Assets* del proyecto de Unity y que aportan todas las funcionalidades necesarias para el envío de mensajes OSC desde Unity al servidor scsynth. Concretamente, se han utilizado los archivos OSCHandler.cs, OSCHelper.cs, OSCBundle.cs, OSCClient.cs, OSCMessage.cs, OSCPacket.cs, OSCReceiver.cs y OSCServer.cs. Aunque todos los archivos son necesarios para el correcto funcionamiento del software, vamos a explicar brevemente la utilidad de los archivos más relevantes.

- **oscclient.cs.** Contiene la definición de la clase que alberga la información del cliente de la conexión. Esta información se compone de dirección IP y puerto y se utiliza para realizar una conexión UDP y enviar los mensajes.
- **oscpacket.cs.** Alberga los datos y los métodos para transformar dichos datos en secuencias de bytes que pueden ser enviadas al servidor. También dispone de métodos para realizar el proceso inverso y recibir datos.
- **oschandler.cs.** Este archivo es el encargado de usar el resto de archivos para controlar la creación de clientes y el envío de mensajes. Para ello, implementa un patrón *singleton* que permite acceder a la instancia de forma global desde cualquier script del juego. Dentro de este archivo se ha implementado la creación de cliente

para el proyecto; es necesario pasar como argumentos el nombre del cliente, la dirección IP y el puerto.

- **osccall.cs.** Pequeño archivo cuyo único objetivo es realizar la llamada al método `Init` de `OSCHandler.cs`, accediendo a la instancia del *singleton* para crear los clientes. Esta llamada está dentro de la función `Start`, heredada de la clase `MonoBehaviour`, como vimos en secciones anteriores, con el objetivo de que sea activada en cuanto se active el script.

Una vez programada la creación del cliente, solo falta añadir como componentes los scripts `OSCHandler.cs` y `OSCCall.cs` a un elemento de Unity que vaya a estar en todo momento activo en el juego. En este trabajo, se ha decidido que este elemento sea `CameraFollow`, una cámara que sigue al jugador constantemente durante la partida. Puede verse cómo los scripts mencionados están añadidos a dicho elemento en la siguiente figura:

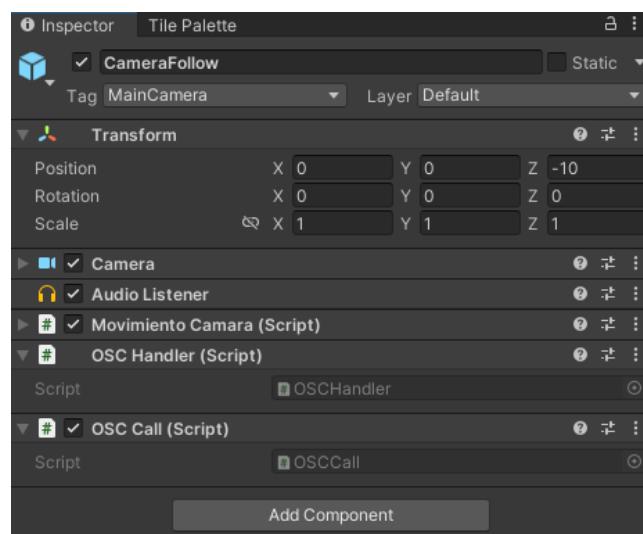


Figura 4.17. Scripts añadidos al elemento `CameraFollow`

4.2.4. ChordGenerator.sc

El objetivo de este trabajo, que es generar música, requiere, antes de afrontarlo, una pausa para analizar qué es realmente la música. Entender el concepto sin ninguna base sobre teoría musical puede ser realmente difícil, pero, dado que en este apartado y en los siguientes va a ser totalmente necesario mencionar muchos aspectos de este ámbito, a continuación, intentamos exponerlos de la forma más accesible y sencilla posible.

Para poder producir algo, primero necesitamos entender qué partes lo componen y cómo se estructuran esas partes. Así, por ejemplo, si nuestro objetivo es cocinar una receta, debemos conseguir los ingredientes, dividirlos en las proporciones necesarias y añadirlos siguiendo un orden en el tiempo. La música funciona de la misma forma. Tiene unos “ingredientes”, como notas, acordes o silencios, que siguen una estructura, dividida en capas con sus propias normas, como son la armonía o la melodía, y todos estos elementos respetan un orden temporal, que en música se conoce como *tempo*.

Una vez presentados, a grandes rasgos, qué elementos componen la música, debemos hacer una pausa en este apartado para hablar de la armonía. Se podría definir la armonía como un conjunto de normas que establecen cómo los sonidos individuales se pueden unir para formar acordes y cómo estos acordes pueden encadenarse para crear una pieza musical. Aunque entender estas normas es una tarea muy compleja, que lleva años de aprendizaje, para este trabajo no será necesario profundizar tanto. Sin embargo, sí hay que destacar que diseñar un software capaz de aplicar dichas normas de forma correcta sale del ámbito de un trabajo de este tipo. Por este motivo, para este proyecto se ha usado un archivo, disponible en [31], para resolver el problema de la construcción de acordes, llamado `ChordGenerator.sc`.

Este archivo cumple una única función: recibir un símbolo de SuperCollider, que representa un acorde, y transformarlo en un array de notas MIDI. De esta forma, por ejemplo, podemos construir un acorde de “Do mayor” pasando como argumento a la clase el símbolo `\CM` o uno de “Sol con séptima y novena aumentada” usando `\G7#9`³. Todo su funcionamiento se basa en el procesamiento de cadenas de texto, extrayendo de cada símbolo la letra inicial, que representa la nota base, y el resto como estructura. La nota inicial y la estructura son comparadas con una cadena de texto, situada al comienzo del archivo, que contiene gran cantidad de acordes distintos, junto con sus respectivos intervalos (un intervalo es la distancia que hay entre dos notas de un acorde o conjunto de notas). En la Figura 4.18. podemos ver la cadena de texto mencionada, en la que cada línea representa los acordes y sus intervalos.

³ Internacionalmente, las notas musicales se expresan con letras del alfabeto. Para la escala “Do, Re, Mi, Fa, Sol, La, Si”, la equivalencia es “C, D, E, F, G, A, B”. Por este motivo, los símbolos que se utilizan en SuperCollider para representar las notas son dichas letras.

```

*initClass {
    var t ;
    decay = 0.5;
    release = 0.5 ;
    symbols = () ;
    t = "CM M Cmaj {0,4,7}
Cm C- Cmin {0,3,7}
C+ Caug CM#5 CM+5 {0, 4, 8}
C° Cdim Cmb5 Cm°5 {0, 3, 6}
C7 Cdom7 {0, 4, 7, 10}
CM7 CMa7 Cj7 Cmaj7 {0, 4, 7, 11}
CmM7 Cm#7 C-M7 Cminmaj7 {0, 3, 7, 11}
Cm7 C-7 Cmin7 {0, 3, 7, 10}
C+M7 Caugmaj7 CM7#5 CM7+5 {0, 4, 8, 11}
C+7 Caug7 C7#5 C7+5 {0, 4, 8, 10}
CØ CØ7 Cø Cø7 Cmin7dim5 Cm7b5 Cm7°5 C-7b5 C-7°5 {0, 3, 6, 10}
Co7 C°7 Cdim7 {0, 3, 6, 9}
C7b5 Cdom7dim5 {0, 4, 6, 10}
CM9 Cmaj9 {0, 4, 7, 11, 14}
C9 Cdom9 {0, 4, 7, 10, 14}
CmM9 C-M9 Cminmaj9 {0, 3, 7, 11, 14}
Cm9 C-9 Cmin9 {0, 3, 7, 10, 14}
C+M9 Caugmaj9 {0, 4, 8, 11, 14}
C+9 C9#5 Caug9 {0, 4, 8, 10, 14}
CØ9 {0, 3, 6, 10, 14}
CØ9b9 {0, 3, 6, 10, 13}
C°9 Cdim9 {0, 3, 6, 9, 14}
C°b9 Cdimb9 {0, 3, 6, 9, 13}
C11 Cdom11 {0, 4, 7, 10, 14, 17}
CM11 Cmaj11 {0, 4, 7, 11, 14, 17}
CmM11 C-M11 Cminmaj11 {0, 3, 7, 11, 14, 17}
Cm11 C-11 Cmin11 {0, 3, 7, 10, 14, 17}
C+M11 Caugmaj11 {0, 4, 8, 11, 14, 17}
C+11 C11#5 Caug11 {0, 4, 8, 10, 14, 17}
CØ11 {0, 3, 6, 10, 13, 17}
C°11 {0, 3, 6, 9, 13, 16}
CM13 Cmaj13 {0, 4, 7, 11, 14, 17, 21}
C13 Cdom13 {0, 4, 7, 10, 14, 17, 21}
CmM13 C-M13 Cminmaj13 {0, 3, 7, 11, 14, 17, 21}
Cm13 C-13 Cmin13 {0, 3, 7, 10, 14, 17, 21}
C+M13 Caugmaj13 {0, 4, 8, 11, 14, 17, 21}
C+13 C13#5 Caug13 {0, 4, 8, 10, 14, 17, 21}
CØ13 {0, 3, 6, 10, 14, 17, 21}
C6 CM6 {0,4,7,6}
Cm6 Cminmaj6 {0,3,7,6}
C7#9 {0, 4, 7, 10, 15}
C7b9 {0, 4, 7, 10, 13}
C7#11 {0, 4, 7, 10, 19}
C7b11 {0, 4, 7, 10, 17}"
```

Figura 4.18. Representación de los acordes y sus respectivos intervalos como cadena de texto en ChordGenerator.sc

El código de este archivo se ha ampliado con algunas funcionalidades que han sido necesarias para la generación de música, que serán tratadas en el siguiente apartado, en el que se expone la implementación realizada para este trabajo.

4.3. Detalles de implementación

Habiendo explicado las herramientas software que han sido necesarias, en esta sección se presenta toda la implementación realizada en el contexto de este trabajo. Para ello, analizaremos con detalle los archivos desarrollados, explicando todas las funcionalidades. El código desarrollado puede consultarse en la dirección de GitHub siguiente: <https://github.com/albertopm97/TFG-Alberto-Perez-Morales>.

4.3.1. GeneradorDeSecuencias.sc

Partiendo desde un punto inicial en el que tenemos un software capaz de construir acordes de notas MIDI, lo primero que debemos preguntarnos es cómo va a ser la música que queremos generar. Dada mi formación en teoría musical, se tomó la decisión de producir una música sencilla, pero efectiva. Esta consiste en tres capas bien diferenciadas: una percusión para mantener el ritmo, una armonía basada en arpegios y una melodía.

En este primer apartado, vamos a centrarnos en la capa armónica, y debemos comenzar comprendiendo el concepto de arpegio. Un arpegio es una técnica para tocar acordes que consiste en hacer sonar sus notas en una sucesión, normalmente ascendente, en lugar de todas a la vez. Se ha optado por emplear arpegios porque son uno de los elementos más sencillos, pero que más enriquecen la música. Ahora bien, no basta con tocar cualquier secuencia de acordes aleatoria para conseguir una pieza musical con sentido. Como se comentó previamente, para que dos acordes suenen bien al tocarlos seguidos deben seguir una serie de normas de armonía. Por tanto, el primer paso es implementar un software capaz de generar secuencias de acordes que tengan sentido.

Una forma de implementar esto, tal como exponen Scirea et al. [32], es a través de un grafo dirigido cuyos nodos representan acordes y sus aristas las posibles transiciones. Un detalle importante, si recordamos, es que nuestro generador de acordes recibe símbolos que identifican un acorde concreto, por ejemplo “Do mayor” (\CM) o “Fa menor” (\Fm). Esto es un problema, porque nuestro objetivo es que el generador de secuencias sea genérico, es decir, que permita crear secuencias de acordes en cualquier escala. Para ello, en teoría musical se utiliza una notación distinta a los nombres de notas, llamada grados. Los grados se indican con números romanos del uno al siete y representan la posición de una nota con respecto a la tónica de la escala, siendo la tónica la primera nota de esta.

Por ejemplo, en una escala de “Do”, el grado V es la nota “Sol” y el VII es la nota “Si”. Gracias a esta notación, se ha podido usar un grafo genérico de transiciones entre acordes totalmente independiente de la escala. El grafo implementado deriva de uno ideado por Steve Mugglin [33]⁴; concretamente, el que encontramos en la Figura 4.19. [34].

⁴ Se han implementado todos los nodos del grafo, a excepción de aquellos que tienen una estructura del tipo X/Y, que indica que se debe tocar el acorde X con las notas de la escala Y, lo cual dificulta enormemente la implementación, ya que resulta incompatible con el generador de acordes.

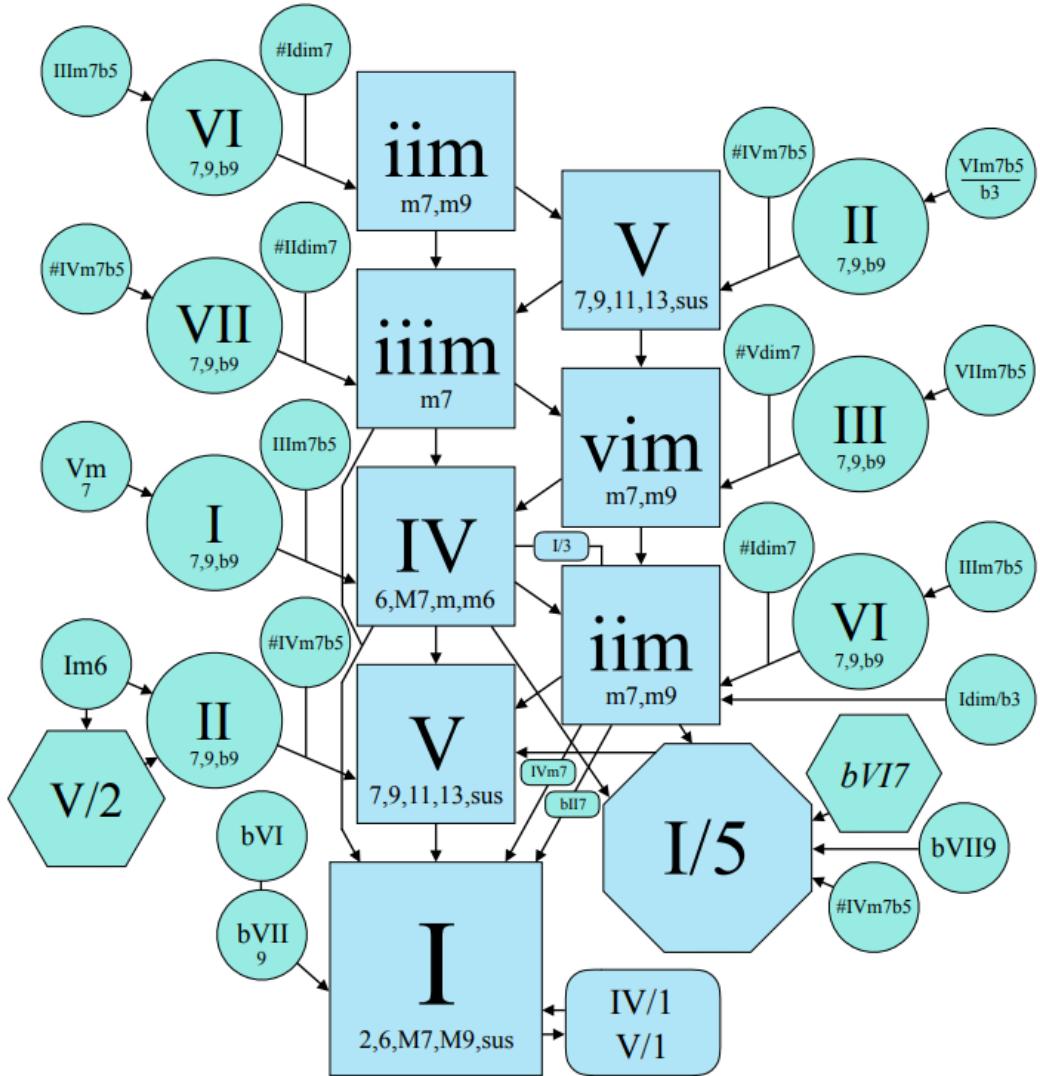


Figura 4.19. Grafo genérico de grados de transiciones de acordes. Imagen disponible en [\[34\]](#)

Para traducir este grafo a código y usarlo para la generación de secuencias de acordes, se han implementado dos clases en SuperCollider:

- **Nodo.** Esta clase mantiene toda la información de un nodo del grafo de transiciones. Contiene tres variables, el constructor y dos métodos. Primero, repasamos las variables utilizadas:
 - **nombres.** Array de símbolos que contiene todos los acordes que forman parte de un nodo. Por ejemplo, el primer nodo representado como un cuadrado azul en la parte superior izquierda sería: [\IIm, \VIm7, \IIm9].
 - **acordesPosibles.** Array de símbolos que contiene todos los acordes a los que un nodo puede transicionar. Su forma es similar al **nombres**.

- `esInicial`. Valor booleano que indica si el nodo puede iniciar o no una secuencia. Tomaremos como nodos iniciales aquellos que están en los extremos del grafo.

Los constructores únicamente inicializan `nombres` y `acordesPosibles` con arrays vacíos, por lo que vamos directamente a comentar los dos métodos. Son los siguientes:

- `esAcordePosible`. Recibe como argumento un acorde en forma de símbolo y busca dentro de `acordesPosibles` para comprobar si es una transición válida.
- `comprobarPertenencia`. Recibe como argumento un acorde y comprueba si pertenece al array de nombres del nodo.
- **GeneradorSecuencias**. Esta es la clase que encapsula el grafo de transiciones e inicializa los nodos. También incorpora todos los métodos necesarios para generar una secuencia de acordes genérica y transformarla en una de una escala en concreto. Comenzamos viendo las variables:
 - `grafoTransiciones`. Array de objetos de la clase `Nodo`, que alberga los nodos del grafo. En la práctica, representa el grafo en sí.
 - `secuenciaGenerada`. Array de símbolos que alberga la secuencia de acordes generada usando el grafo. Originalmente, guarda una secuencia genérica, usando grados, hasta que esta se transforma a acordes de una escala específica, usando el método `transformarSecuencia`, descrito en líneas posteriores.
 - `escalas`. Se trata de un diccionario en el que las claves son las notas iniciales de las siete escalas (“C”, “D”, “E”, “F”, “G”, “A”, “B”) y los valores, un array con las notas de cada escala. Para entender los arrays de notas de la escala, es necesario introducir el concepto de armadura musical. Como hemos comentado anteriormente, la distancia entre notas se llama intervalo. Estos intervalos se miden usando tonos y semitonos y, para que una escala suene bien, el intervalo de cada par de notas debe tener una medida concreta. Esta estructura consiste en dos tonos, un semitono, tres tonos y un semitono. Para respetar esta estructura, es necesario alterar ciertas notas de la escala, en función de la tónica. Al conjunto de alteraciones propias de la escala se le conoce como armadura.

En la Figura 4.20. pueden apreciarse las distintas armaduras existentes en función de la escala. Por ejemplo, si nos fijamos en la porción situada más a la derecha, con el título “LA M”, podemos apreciar que, para mantener la estructura, se deben alterar con sostenidos las notas “Fa”, “Do” y “Sol” (“F”, “C” y “G”). En la Figura 4.21. puede verse el contenido de este diccionario.

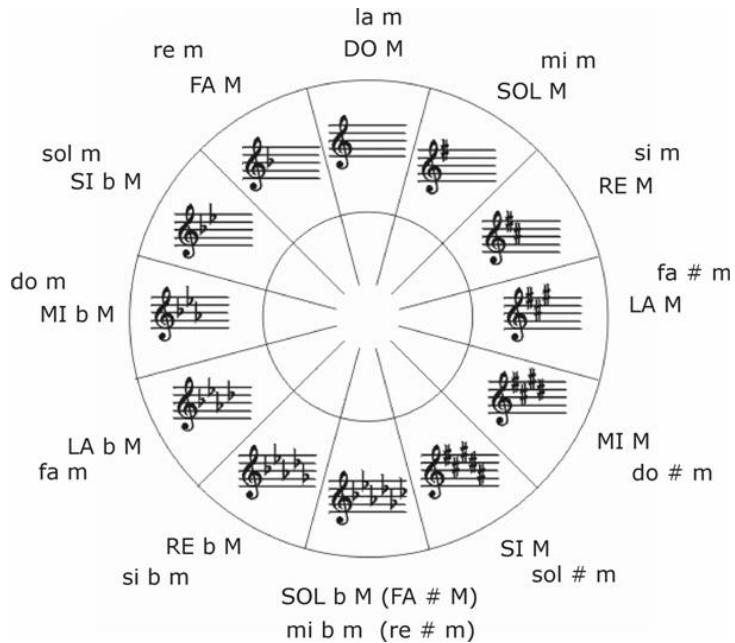


Figura 4.20. Cuadro resumen sobre armaduras. Imagen extraída de [35]

- equivalencias. Diccionario en el que las claves son los grados del I al VII y los valores, la posición que representa cada grado en los arrays de notas del diccionario escalas. En la Figura 4.21. puede verse el contenido de este diccionario.

Aunque, a priori, sean algo difíciles de entender, gracias a ambos diccionarios, realizar una transformación desde un acorde genérico se convierte en una tarea sencilla, que analizaremos más adelante. Continuando con la revisión de la clase, como ocurría con la clase `Nodo`, lo siguiente que encontramos es un constructor, que, en este caso, inicializa `grafoTransiciones` y `secuenciaGenerada` con arrays vacíos y escalas y equivalencias, con diccionarios vacíos. Pasamos ahora a examinar los métodos de la clase:

- `inicializarDiccionarios`. Método encargado de introducir en los diccionarios los pares clave-valor necesarios para las transformaciones de secuencias de acordes genéricos. En la siguiente figura se puede apreciar cómo se ha hecho:

```

inicializarDiccionarios{

    //Inicialización del diccionario de escalas
    escalas.put('C', ["C", "D", "E", "F", "G", "A", "B"]);
    escalas.put('D', ["D", "E", "F#", "G", "A", "B", "C#"]);
    escalas.put('E', ["E", "F#", "G#", "A", "B", "C#", "D#"]);
    escalas.put('F', ["F", "G", "A", "B", "C", "D", "E"]);
    escalas.put('G', ["G", "A", "B", "C", "D", "E", "F#"]);
    escalas.put('A', ["A", "B", "C#", "D", "E", "F#", "G#"]);
    escalas.put('B', ["B", "C#", "D#", "E", "F#", "G#", "A#"]);

    equivalencias.put('I', 0);
    equivalencias.put('II', 1);
    equivalencias.put('III', 2);
    equivalencias.put('IV', 3);
    equivalencias.put('V', 4);
    equivalencias.put('VI', 5);
    equivalencias.put('VII', 6);
}

```

Figura 4.21. Inicialización de los diccionarios escalas y equivalencias

- o `inicializarGrafoTransiciones`. Método encargado de introducir todos los nodos del grafo en el array `grafoTransiciones`. En la siguiente figura se puede observar un ejemplo del código necesario para añadir un nodo. En total, el grafo contiene 21 nodos.

```

~nodo7 = Nodo(['VM', 'V7', 'V9', 'V11', 'V13'], []);

~nodo7.acordesPosibles = ~nodo7.acordesPosibles.add('IIIIm');
~nodo7.acordesPosibles = ~nodo7.acordesPosibles.add('IIIIm7');
~nodo7.acordesPosibles = ~nodo7.acordesPosibles.add('VIM');
~nodo7.acordesPosibles = ~nodo7.acordesPosibles.add('VIM7');
~nodo7.acordesPosibles = ~nodo7.acordesPosibles.add('VIM9');
~nodo7.acordesPosibles = ~nodo7.acordesPosibles.add('IM');
~nodo7.acordesPosibles = ~nodo7.acordesPosibles.add('I6');
~nodo7.acordesPosibles = ~nodo7.acordesPosibles.add('IM7');
~nodo7.acordesPosibles = ~nodo7.acordesPosibles.add('IM9');
grafoTransiciones.add(~nodo7);

```

Figura 4.22. Ejemplo de código para crear un nodo. En la primera línea se establece el contenido del array `nombres` y en las líneas posteriores se añade el contenido del array `acordesPosibles`

- o `generarSecuencia`. Método que recibe como argumento el número de acordes que debe tener la secuencia y se encarga de crearla, usando el grafo de transiciones. En primer lugar, elige nodos aleatorios del grafo hasta encontrar uno que es inicial y, una vez lo encuentra, toma un acorde aleatorio de este y lo añade como acorde inicial de la secuencia. Seguidamente, se elige aleatoriamente un acorde del array `acordesPosibles` y se vuelve a buscar, de forma aleatoria, un nodo del array que contiene el grafo hasta encontrar uno que tenga el acorde elegido dentro del array `nombres`, en cuyo caso, se añade el acorde a la secuencia. Este proceso se repite hasta obtener el número de acordes indicado en el argumento.

- `transformarSecuencia`. Método que recibe como argumento el símbolo que representa una escala (por ejemplo, `\C`) y transforma la secuencia de acordes genérica generada a su correspondiente en la escala dada. Para ello, en primer lugar, recorremos la secuencia generada convirtiendo cada elemento en `string`. Recordemos que, originalmente, los elementos son símbolos, por lo que podemos hacer uso del método `asString`, propio de la clase, para convertirlos a `string`. Una vez hecho esto, iteramos nuevamente por el array, empleando el método `replace`, junto con los diccionarios inicializados previamente para reemplazar cada grado por su nota equivalente en la escala del argumento. La figura a continuación muestra en detalle el proceso descrito:

```

secuenciaGenerada.do{
    arg item, i;

    //Reemplazamos los grados por su nota correspondiente de la escala usando los diccionarios
    secuenciaGenerada[i] = secuenciaGenerada[i].replace("VII", escalas.at(escala)[equivalencias.at("\VII")]);
    secuenciaGenerada[i] = secuenciaGenerada[i].replace("VI", escalas.at(escala)[equivalencias.at("\VI")]);
    secuenciaGenerada[i] = secuenciaGenerada[i].replace("IV", escalas.at(escala)[equivalencias.at("\IV")]);
    secuenciaGenerada[i] = secuenciaGenerada[i].replace("V", escalas.at(escala)[equivalencias.at("\V")]);
    secuenciaGenerada[i] = secuenciaGenerada[i].replace("III", escalas.at(escala)[equivalencias.at("\III")]);
    secuenciaGenerada[i] = secuenciaGenerada[i].replace("II", escalas.at(escala)[equivalencias.at("\II")]);
    secuenciaGenerada[i] = secuenciaGenerada[i].replace("I", escalas.at(escala)[equivalencias.at("\I")]);

    //Si hemos metido un # extra lo quitamos
    secuenciaGenerada[i] = secuenciaGenerada[i].replace("##", "#");

    //Si en el proceso hemos metido seguidos # y b debemos quitar el #
    secuenciaGenerada[i] = secuenciaGenerada[i].replace("#b", "b");
};


```

Figura 4.23. Proceso de transformación de una secuencia genérica a su equivalente en la escala especificada

- `cambiarModoSecuencia`. Método que recibe como argumento un símbolo (`\M` para mayor o `\m` para menor) y transforma todos los acordes de la secuencia a ese modo. La implementación de este método es similar a `transformarSecuencia`, reemplazando únicamente los modos.

4.3.2. ChordGenerator.sc (implementación propia)

En el capítulo anterior, en la sección de herramientas, ya indagamos en este archivo, pero nos limitamos únicamente a las funcionalidades originales, sacadas de [31]. La necesidad de modificar este archivo viene dada porque encapsula la información necesaria para hacer sonar los acordes, pero solo implementa un único patrón de reproducción: tocar las notas del acorde al mismo tiempo. También comentamos previamente que el objetivo de la armonía iba a ser tocar la secuencia de acordes generada en forma de arpegios, por lo que fue necesario incorporar algunos métodos a la clase para conseguirlo. Asimismo, ya que este archivo contenía lo necesario para hacer sonar los acordes, se optó por definir aquí también tanto los sintetizadores como los patrones necesarios para generar la música. De esta forma, si `GeneradorDeSecuencias.sc` implementa el generador de secuencias de acordes, `ChordGenerator.sc` podría considerarse el generador de música en sí.

A continuación, comentamos los métodos de implementación propia para conseguir este objetivo:

- `inicializarSynths`. Método encargado de definir los sintetizadores que se utilizan para generar la música. La música de un juego debe acompañar al apartado gráfico y ambos deben trabajar juntos para crear una experiencia. Puesto que el juego que va a servir de base para las pruebas tiene un estilo clásico que usa *Pixel Art*, se ha intentado que los sintetizadores creen sonidos con un estilo retro, parecido al 8-bit. Concretamente, se han utilizado cinco sintetizadores, que vamos a comentar a continuación.
 - `melodia`. Emplea un envelope tipo percusión para obtener un sonido breve y punzante y calcula su sonido como combinación de una onda sinusoidal y una de pulso, para conseguir un sonido estilo retro, pero sin ruido. Su objetivo es que el sonido sobresalga ligeramente sobre los arpegios y la percusión.
 - `arpegios`. Emplea un envelope simple de tres nodos, en el que el máximo se alcanza muy rápido, en 0.01 segundos, y, a partir de ahí, el sonido decrece hasta desaparecer en 0.4 segundos. Para calcular el sonido utiliza una onda de sierra multiplicada por un factor, redondeada y dividida entre el mismo factor para generar distorsión. Además, puede recibir un parámetro `filterQ`, que utiliza como valor de `bwr` para activar o no un filtro de resonancia. Esta utilidad se explicará con más detalle en el siguiente apartado.
 - `reverb`. Este es el único sintetizador que no tiene como objetivo producir sonido de forma directa, sino dejarse en segundo plano en un bus para aplicar su efecto a cualquier sonido que se reproduzca en él. Para ello, usa una función capaz de leer señales y aplica sobre ella un filtro ya implementado por defecto en supercollider llamado `FreeVerb2`. Este produce un efecto de reverberación, es decir, hace que el sonido se mantenga de forma más suave una vez que la fuente lo ha emitido.

- `bombo`. Este sintetizador es algo más complejo que los anteriores. Puede recibir como parámetro los valores de un envelope para cambiar la forma de la función del sonido y calcula el sonido usando una onda sinusoidal junto a dos envelopes. El sonido generado intenta imitar la percusión de un bombo y se usa en la percusión.
 - `caja`. Sintetizador que pretende imitar el sonido de la caja de una batería. Emplea un envelope que alcanza su máximo en 0.01 segundos y decae en 0.1 segundos. Usa la combinación entre ruido blanco y una onda sinusoidal a la que se le aplica un filtro de paso bajo para calcular su sonido.
- `incluirAcorde`. Método auxiliar para la creación de arpegios. Para tocar un arpegio de la manera deseada para este trabajo, se necesitan todas las notas del acorde en sentido ascendente y las mismas notas en sentido descendente, menos la primera y la última. Este método parte de un acorde, introduciendo todas sus notas en orden ascendente en un array vacío. Después, calcula el inverso de dicho array y elimina su primer y su último elemento. Finalmente, une ambos arrays en uno e introduce todos sus elementos en el array que contiene las notas de todos los arpegios.
- `inicializarArpegios`. Método que recibe como argumento la secuencia de acordes generada en la clase `GeneradorDeSecuencias` y transformada a una escala concreta. Crea un array vacío para almacenar las notas de todos los arpegios y otro array, también vacío, para almacenar duraciones. Este es necesario para el patrón de reproducción y debe ser del mismo tamaño que el array de notas, y los valores indican la duración de la nota que está en la misma posición. A continuación, recorre la secuencia generada y, para cada acorde, llama a la función `incluirAcorde`, que introduce las notas de su arpegio en el array de notas. Por último, inicializa el array de duraciones creando uno del mismo tamaño que el de notas, en el que cada posición tiene el valor 1, ya que queremos que todas las notas de los arpegios tengan la misma duración.
- `tocarArpegio`. Método que inicializa en una variable global el patrón de reproducción `arpegios`. Este consiste en una secuencia infinita de las notas del array de arpegios, generada mediante el uso de `Pseq`, con las duraciones de cada nota iguales, calculadas normalizando el array `duraciones`. Usa el sintetizador `armonia` y puede recibir como parámetro un valor de `filterQ` para activar o no el filtro de resonancia.
- `tocarPercusion`. Este método se encarga de inicializar en dos variables globales los patrones de reproducción pertenecientes a la percusión. Dichos patrones son los siguientes:

- `patronBombo`. Usa el sintetizador `bombo` junto a valores aleatorios en el envelope y la amplitud para generar sonidos infinitos a tempo (ritmo constante), pero con distintos matices.
- `patronCaja`. Utiliza el sintetizador `caja` junto a valores aleatorios de `amp`, `atk` y `rel` (explicados en el apartado 4.2.2.), para generar infinitos sonidos a contratiempo.
- `tocarMelodia`. En primer lugar, inicializa en un array posibles frases melódicas, que en la práctica, son valores de notas MIDI. Después, crea en una variable global el patrón `melodia`, el cual emplea el sintetizador `melodia`, junto a una concatenación de patrones, como `Pdup` o `Prand`, en la clave `degree`, para generar frases melódicas distintas. También aplica valores aleatorios de `dur`, `amp` y `legato` para conseguir diferentes tipos de sonoridad y conexiones entre notas.

4.3.3. ManejadorMensajes.sc y adaptación dinámica de la música

Con los dos archivos anteriores ya tenemos todas las herramientas necesarias para generar la música, por lo que resta decidir qué variables del juego van a afectar de forma dinámica a la música e implementar las funciones manejadoras de los mensajes recibidos desde Unity. En primer lugar, vamos a explicar qué variables del juego intervienen en la adaptación de la música generada y por qué se han elegido.

Lo primero que debemos preguntarnos es qué queremos conseguir con esta adaptación. Si recordamos, en el apartado 2.1. de este trabajo ya se exploró, en términos fisiológicos y psicológicos, la capacidad que tiene la música para influir en las emociones y sentimientos humanos. Con esto en mente, nuestro objetivo debe ser que la música acompañe a lo que ocurre en partida y ayude al juego a transmitir al jugador las emociones que los desarrolladores querían expresar cuando lo crearon. Así, debemos hacer una pausa en la explicación del trabajo para explicar el juego que sirve como base en el proyecto.

Se trata de un juego en 2D, de género *roguelite*, en el que el jugador debe enfrentarse a oleadas de enemigos, recolectando experiencia para subir de nivel y así obtener mejoras para hacerse más fuerte e ir avanzando. En este tipo de juegos, cada partida depende de las elecciones que toma el jugador para mejorar al personaje y, aunque no en todos los juegos del género es así, en este concreto el objetivo es la supervivencia el mayor tiempo posible. En la figura siguiente aparece una captura de pantalla en un determinado instante en la ejecución del juego.



Figura 4.24. Captura de un instante de ejecución del juego

Partiendo de esta base, nos preguntamos nuevamente qué emociones queremos transmitir con la música. En un estado normal del juego, en el que el jugador tiene bajo control a los enemigos, la música debería inducir tranquilidad. En cambio, si, en un momento dado, una oleada sobrepasa al jugador y lo pone en una situación difícil, la música debería transmitir inquietud o desasosiego. Otro factor diferencial es el hecho de que, en el juego, el personaje va evolucionando sus poderes conforme sube de nivel, por lo que sería deseable que la música, de alguna forma, potenciara esta sensación de avance. De igual modo, en el juego existen dos tipos de enemigos, los normales y los mini jefes, que son más poderosos y suponen un reto mayor. La música también podría verse afectada de algún modo cuando apareciera algún mini jefe en partida, para alertar al jugador. Incluso, podríamos tomar los puntos de salud del jugador, como elemento crucial para la supervivencia que son, y usarlos para generar una sensación de agobio cuando caigan por debajo de un cierto umbral.

Con todas estas ideas en mente, las variables que se ha decidido parametrizar son las siguientes:

- **Ritmo de la música en función del número de enemigos activos.** En el apartado 2.1.2., ya vimos cómo el ritmo de la música puede afectar a nivel fisiológico al organismo para generar emociones estimulantes. Aplicando esto podemos conseguir que, si en un momento dado, el jugador se ve sobrepasado por la cantidad de enemigos, la música acompañe el momento creando agitación. De la misma manera, si el jugador lo tiene todo bajo control, la música se adaptará creando sensaciones de calma y tranquilidad.

- **Modo de los acordes en función de los puntos de salud restantes.** También en el apartado 2.1.2. pudimos entender que el modo de un acorde (mayor o menor) está estrechamente relacionado con la sensación que transmite. Podemos aprovechar esto para transformar la secuencia de acordes a modo menor en momentos en los que el jugador tenga su vida por debajo de un umbral, para generar sensación y emociones de tensión.
- **Nuevas melodías con cada subida de nivel.** Una de las características más importantes de este género de juegos es la sensación de evolución en la partida. El jugador debe sentir una clara mejora en el poder de su personaje con cada nivel adquirido. Para contribuir a esta sensación, debemos conseguir que la música no solo se adapte, sino que evolucione en complejidad con los niveles. Para ello, existe un concepto musical, conocido como densidad, que se adecúa bastante bien a esta idea. Podemos definir la densidad como la cantidad de sonidos simultáneos que suenan en un momento concreto en una obra musical. Este concepto interviene en muchos aspectos de la música. Por ejemplo, es posible crear un crescendo aumentando la densidad sin tener necesariamente que tocar los instrumentos más fuerte. Teniendo esto en cuenta, y haciendo uso del patrón que genera melodías, vamos a ir añadiendo capas melódicas con cada subida de nivel para generar un aumento de la complejidad musical y, en última instancia, fomentar la sensación de evolución en el juego.
- **Uso de un filtro de resonancia en los arpegios si hay mini jefes activos.** Comentan Plans y Morelli [23] que se puede hacer uso de un filtro de resonancia para crear excitación, y es, precisamente, esto lo que usamos para alertar al jugador cada vez que aparezca un mini jefe en la partida.

Ahora que hemos aclarado cómo vamos a adaptar la música generada, es el momento de introducir dentro de Unity las llamadas para enviar mensajes a SuperCollider, usando la función `SendMessageToClient`, proporcionada por la instancia del `OSCHandler`. Al usarla debemos pasar como argumentos el nombre del cliente, que ya creamos previamente, el identificador del mensaje (string con el símbolo que representa el mensaje) y la variable que queremos enviar. Esta función debe ser usada en cada script del juego que realice un cambio en una de las variables que hemos usado para la adaptación de la música. En los puntos siguientes se explica en qué script han sido colocadas las llamadas a la función `SendMessageToClient`, en función del efecto deseado en la música.

- **Inicio de la música.** Este mensaje puede ser enviado en la función `Start` o `Awake` de cualquier script que se active al inicio de la partida, por ejemplo, el que controla las estadísticas del jugador, llamado `EstadisticasJugador`. El mensaje enviado tiene clave `iniciarMusica` y, puesto que es necesario enviar una variable, se ha creado una llamada `iniciarMusica`, con valor 1.0.

- **Fin de la música.** Este mensaje se envía desde el mismo script que el anterior, ya que una posible forma de saber cuándo finaliza la partida es cuando se activa la función `Morir` que incorpora. El mensaje lleva clave `finalizarMusica` y valor 1.0, igual que en el inicio de la música.
- **Cambios de ritmo.** En el script encargado de hacer aparecer los enemigos, existe una variable llamada `enemigosActivos`, que se incrementa con cada nuevo enemigo y disminuye cuando son derrotados. Cada vez que esta variable es modificada se envía un mensaje a SuperCollider, con clave `cambioEnemigosActivos` y el valor de la variable.
- **Activación del filtro de resonancia.** Similar al caso anterior, pero la variable se llama `miniBossActivos`, controlando el número de mini jefes activos, y la clave del mensaje enviado es `cambioMinibosses`.
- **Cambios de modo en la secuencia de acordes.** Todas las variables y métodos necesarios para gestionar las estadísticas del jugador se encuentran en un script llamado `EstadisticasJugador`. Entre todas las variables se encuentra `vidaActual`, encargada de controlar los puntos de salud del jugador. Esta disminuye con los golpes enemigos recibidos a través de la función `RecibirAtaque`, y aumenta con la curación pasiva y la recolección de pociones, por medio de las funciones `curar` y `curaPasiva`. Es en cada uno de estos puntos donde se manda un mensaje a SuperCollider, con clave `cambioVidaActual` y el valor de la variable, `vidaActual`.
- **Añadir capas melódicas.** También en el mismo script descrito en el punto anterior se controla el nivel del jugador mediante la variable `nivel`. Esta variable varía con cada subida de nivel dentro de la función `comprobarLvUp`, y es aquí donde, si se comprueba la subida de nivel, la variable se actualiza y se lanza el mensaje a SuperCollider, con clave `lvUp` y el valor de la variable, `nivel`.

En este punto, es el momento de implementar los manejadores para los mensajes enviados desde Unity. Para ello se ha creado un archivo llamado `ManejadorMensajes.sc`, que los contenga todos. Un manejador de mensajes se implementa en SuperCollider creando un objeto de la clase `OSCdef`. En la Figura 4.25. puede verse un ejemplo de creación de manejador. Al crearlo, es necesario pasarle los siguientes argumentos:

- **key.** Clave en la que almacenar el objeto dentro de la colección global. Normalmente, se utiliza un símbolo.
- **func.** Función a utilizar cuando se recibe el mensaje.
- **path.** Símbolo que indica la dirección OSC del objeto.

- **srcID**. Dirección IP del remitente, que puede usarse para responder, exclusivamente, mensajes de esa fuente.
- **recvPort**. Puerto en el que se debe recibir el mensaje.

```
/*
 * Manejador para la llamada de fin de musica al finalizar el juego
 */
OSCDef.new(
    \finalizarMusica,           //Simbolo para identificar el manejador
    {arg msg;                  //Funcion que indica que hacer al recibir el mensaje|
     [msg].postln;

    //En este caso, Para parar la música simplemente interrumpimos la ejecución de los patrones
    ~patronBombo.stop;
    ~patronCaja.stop;
    ~arpegio.stop;

    ~melodias.do{
        arg item;

        item.stop;
    },
    '/finalizarMusica', //Mensaje a leer
    nil, //Opcional: Dirección IP del emisor del mensaje.
    57120 //Puerto para escuchar los mensajes
);
```

Figura 4.25. Ejemplo de creación de un manejador

A continuación, repasamos todos los manejadores implementados:

- **iniciarMusica**. Actúa ante el mensaje con clave iniciarMusica. Comienza creando un objeto de la clase GeneradorSecuencias y usándolo para inicializar los diccionarios y el grafo de transiciones, así como para generar una secuencia de acordes genérica de tamaño aleatorio entre 20 y 50. Después, se crean variables necesarias para ajuste dinámico del tempo, modo y resonancia y se elige una escala aleatoria para transformar la secuencia de acordes genérica. Seguidamente, se limpia la memoria de buses y se crea uno para el reverb y, a continuación, se inicializan los sintetizadores y los arpegios y se crean una variable para controlar el tempo de la música y dos relojes, uno para los arpegios y la percusión y otro para la melodía.

Los relojes son un tipo de objeto de SuperCollider, que permite recibir como argumento un ritmo, medido en beats por segundo, y utilizarlo para sincronizar varios patrones a ese ritmo. Además, facilitan el cambio de ritmo dinámico ya que, al cambiar el tempo del reloj, todos los patrones que lo usan se ven afectados. Tras esto se inicializan los patrones de percusión, arpegios y melodía, se inicia la ejecución del reverb en el bus y se crea una array para alojar los reproductores de las distintas melodías. Finalmente, se ejecuta una rutina auxiliar que inicia gradualmente la ejecución de los patrones de percusión y arpegios.

- **finalizarMusica.** Actúa ante el mensaje con clave `finalizarMusica`, interrumpiendo la ejecución de todos los patrones de la música.
- **cambioEnemigosActivos.** Actúa ante el mensaje con clave `cambioEnemigosActivos`, guardando en una variable el valor del número de enemigos activos, que es pasada como argumento a una función auxiliar. Esta función establece 4 rangos en función de la cantidad de enemigos activa: entre 0 y 5, entre 5 y 10, entre 10 y 15 y entre 15 y 20. Según el rango en el que se encuentra la variable pasada como argumento, adecúa el ritmo de la música.
- **cambioVidaActual.** Actúa ante el mensaje con clave `cambioVidaActual`, guardando en una variable el valor de la vida del jugador, y ejecuta una función auxiliar pasando como argumento este valor. La función auxiliar se encarga de comprobar si la vida del jugador ha bajado del 40%, cambiando la secuencia de acordes a modo menor en caso afirmativo. Si, más adelante, el jugador consigue recuperar salud, se recupera la secuencia original.
- **lvUp.** Actúa ante el mensaje con clave `lvUp`, creando una nueva melodía y añadiendo su reproductor al array de melodías.
- **cambioMinibosses.** Actúa ante el mensaje con clave `cambioMinibosses`, guardando en una variable el valor el número de mini jefes activos en la partida, y ejecuta una función auxiliar pasando como argumento este número. Esta función comprueba si el número de mini jefes es mayor que cero y, si el filtro de resonancia está desactivado, lo activa, usando el número de mini jefes como valor de `filterQ` al ejecutar el patrón de los arpegios para activar el filtro de resonancia. En caso de ser cero y estar el filtro activo, se desactiva el filtro ejecutando el patrón sin pasar valor de `filterQ`.

5. Conclusiones y trabajo futuro

5.1. Conclusiones

En este trabajo se ha hecho un estudio sobre la generación procedural de contenido en videojuegos en general, haciendo énfasis, principalmente, en el ámbito de la música.

Sobre la generación procedural en videojuegos, como concepto general, hemos aprendido que tiene posibilidades prácticamente infinitas. No solo se puede generar contenido de casi cualquier tipo, ahorrando tiempo y recursos económicos en el proceso, sino que la aparición de estas técnicas ha dado lugar a géneros totalmente nuevos, que han revolucionado la industria.

Sobre la generación de música, en concreto, hemos aprendido, entre otras cosas, que, aunque tiene grandes ventajas, como la capacidad de adaptación dinámica o el ahorro de espacio de almacenamiento, también hay ciertos inconvenientes, como el miedo al retorno al formato MIDI, la gran dificultad que tiene desarrollar algoritmos que generen música para transmitir emociones de manera eficaz o el costo de estos a nivel computacional, los cuales han provocado que este campo no haya sido muy explorado por las grandes desarrolladoras.

En cuanto a los resultados obtenidos, debemos hablar de luces y sombras. La música generada “suena bien” y cumple su función, pero, en una partida larga, puede volverse bastante repetitiva y, por consiguiente, tiende a cansar al jugador. Si bien es cierto que, con más tiempo de investigación y pruebas, se podría conseguir un mejor resultado, también es cierto que los patrones de SuperCollider basan gran parte de su capacidad para generar música en secuencias y números aleatorios, lo cual acota de forma severa el rango de resultados posibles. Otro factor negativo es que, una vez hemos definido y ejecutado un patrón, los cambios que se pueden hacer de forma dinámica son muy limitados. Además, el tener que crear un sintetizador para cada tipo de sonido que queremos usar limita en gran medida la riqueza tímbrica del resultado final. Todo esto, en conjunto, hace que sea realmente complejo conseguir una música con muchas variaciones en el tiempo.

En cuanto a SuperCollider, hay que señalar que es una herramienta muy poderosa para la síntesis de sonido, pero, quizás, su uso para el desarrollo de videojuegos no sea la mejor elección. Este software evolucionó de la mano de programadores y artistas dedicados al “live coding”, el cual consiste en generar música en directo mediante el uso de código. En este contexto, el artista que usa SuperCollider puede ejecutar unos patrones, parar otros y cambiar variables para crear música variada. En cambio, al usarlo para el desarrollo de videojuegos es necesario dejar todos los patrones y manejadores preprogramados, lo cual supone una gran limitación para el resultado final.

Por otra parte, en cuanto a la adaptación de la música generada al estado de la partida, debemos decir que el resultado ha sido sorprendente. Con solo cuatro tipos de adaptaciones distintas, el producto final supone una clara mejora en la experiencia del

jugador, lo cual nos lleva a plantearnos qué efecto tendría el uso de muchas más variables del juego para llevar a cabo la adaptación.

5.2. Trabajo futuro

Respecto a mejoras o ampliaciones futuras del trabajo presentado, tal como acabamos de comentar, una primera idea implicaría incluir más variables del juego en la lógica de adaptación de la música o introducir cambios adicionales utilizando aquellas que ya se han explicado y utilizado. Posibles nuevas variables a tener en consideración podrían ser el nivel de mejora de un determinado poder, el daño de ataque del jugador o, incluso, su velocidad de movimiento.

Otra posible ampliación interesante de este trabajo sería realizar un estudio, sobre una muestra suficientemente grande de personas, acerca del resultado final del proyecto realizado. Este estudio estaría dividido en dos partes. En primer lugar, el sujeto debería probar el juego las suficientes veces para adquirir una opinión bien formada sobre la música y su adaptación al estado de la partida. Después, debería someterse a un cuestionario en el que pueda expresar su opinión en cuanto a varios apartados, como pueden ser las variables usadas en la adaptación, las capas que forman la música o los sonidos que la componen. Con los resultados de este estudio podríamos adquirir una idea más clara del trabajo realizado y sus posibles mejoras futuras.

Yendo más allá, en el apartado anterior hemos expuesto por qué SuperCollider puede no haber sido la mejor herramienta para la tarea que nos ocupa, por lo que el siguiente paso lógico sería encontrar una herramienta que sí esté específicamente diseñada para nuestro objetivo. En este sentido, los dos programas más utilizados para la gestión de sonido en Unity en la actualidad son Audiokinetic Wwise y FMOD.

Por último, podríamos mejorar enormemente el resultado final si empleamos técnicas de inteligencia artificial. Por ejemplo, Scirea et al. [32] proponen el uso del algoritmo evolutivo FI-2POP para el desarrollo de melodías más complejas. Otra posible idea es el entrenamiento de una red neuronal para generar patrones armónicos más ricos que los arpegios empleados en nuestro acercamiento práctico.

Índice de figuras

Sección 2

- 2.1. Anatomía del cerebro humano
- 2.2. Separación de funciones cerebrales de la música en hemisferios
- 2.3. *Bertie, the Brain*
- 2.4. (Izda) Computadora EDSAC. (Dcha) OXO
- 2.5. *Tennis for Two*
- 2.6. (a) Consola Magnavox Odyssey. (b) Consola Atari 2600. (c) Consola NES. (d) Consola Mega Drive. (e) Consola SNES
- 2.7. (a) Consola Sega Saturn. (b) Consola PlayStation. (c) Consola Nintendo 64

Sección 3

- 3.1. (a) Tamaño de los equipos de desarrollo, en función de la consola. (b) Tiempo medio de desarrollo en meses, en función de la consola. (c) Incremento de costes de producción en millones de dólares, en función de la consola
- 3.2. Ventajas e inconvenientes del uso de la generación procedural de contenido en videojuegos

Sección 4

- 4.1. Ejemplo de cómo se añade un componente a un elemento en Unity
- 4.2. Ejemplo de opciones del componente `Rigidbody` en Unity
- 4.3. Diagrama del ciclo de ejecución de un script de Unity: inicialización y físicas
- 4.4. Diagrama del ciclo de ejecución de un script de Unity: entradas y lógica del juego
- 4.5. Diagrama del ciclo de ejecución de un script de Unity: renderización, interfaz de usuario y salida
- 4.6. `SinOsc` con frecuencia 440 Hz y fase 0.2 radianes
- 4.7. `LFSaw` con frecuencia 440 Hz y fase inicial 0.2
- 4.8. `WhiteNoise` con factor multiplicador 0.5
- 4.9. `LFNoise1` con frecuencia 440 Hz
- 4.10. `Pulse` con frecuencia 440 Hz y anchura 0.3
- 4.11. `RLPF` sobre `SinOsc(440, 0.2)` con frecuencia 25 Hz y rq 0.03
- 4.12. `BPF` sobre `SinOsc(440, 0.2)` con frecuencia 25 Hz y rq 0.03

- 4.13. Resonz sobre SinOsc(440, 0.2) con frecuencia 440 Hz y bwr 3.0
- 4.14. (Izda) Definición de un envelope con 5 niveles de intensidad. (Dcha) Representación gráfica del envelope resultante
- 4.15. Ejemplo de definición de un sintetizador con SynthDef
- 4.16. Definición de un patrón Pbinddef, con las claves explicadas previamente
- 4.17. Scripts añadidos al elemento CameraFollow
- 4.18. Representación de los acordes y sus respectivos intervalos como cadena de texto en ChordGenerator.sc
- 4.19. Grafo genérico de grados de transiciones de acordes
- 4.20. Cuadro resumen sobre armaduras
- 4.21. Inicialización de los diccionarios escalas y equivalencias
- 4.22. Ejemplo de código para crear un nodo. En la primera línea se establece el contenido del array nombres y en las líneas posteriores se añade el contenido del array acordesPosibles
- 4.23. Proceso de transformación de una secuencia genérica a su equivalente en la escala especificada
- 4.24. Captura de un instante de ejecución del juego
- 4.25. Ejemplo de creación de un manejador

Bibliografía

- [1] J. Lacárcel Moreno, "Psicología de la música y emoción musical," *Educatio siglo XXI*, vol. 20, pp. 213-226, 2003. Recuperado a partir de <https://revistas.um.es/educatio/article/view/138>
- [2] <https://anatomy.app/encyclopedia/brain>
- [3] R. W. Sperry, "Cerebral Organization and Behavior: The split brain behaves in many respects like two separate brains, providing new research possibilities," *Science*, vol. 133, no. 3466, pp. 1749-1757, 1961. DOI: [10.1126/science.133.3466.1749](https://doi.org/10.1126/science.133.3466.1749)
- [4] R. W. Sperry, "Hemisphere disconnection and unity in conscious awareness," *The American psychologist*, vol. 23, no. 10, pp. 723-733, 1968. DOI: [10.1037/h0026839](https://doi.org/10.1037/h0026839)
- [5] K. R. Scherer, "Which emotions can be induced by music? What are the underlying mechanisms? And how can we measure them?," *Journal of new music research*, vol. 33, no. 3, pp. 239-251, Septiembre 2004. DOI: [10.1080/0929821042000317822](https://doi.org/10.1080/0929821042000317822)
- [6] P. Almansa Martínez, "LA TERAPIA MUSICAL COMO INTERVENCIÓN ENFERMERA," *Enfermería Global*, vol. 2, no. 1, Mayo 2003. DOI: <https://doi.org/10.6018/eglobal.2.1.665>
- [7] J. Pendergrass, "The Rise of Reactive and Interactive Video Game Audio," 2015. Recuperado a partir de https://digitalcommons.csumb.edu/caps_thes/484/
- [8] J. K. Saucier, "The video game age: A brief history," *IEEE Potentials*, vol. 41, no. 2, pp. 7-16, Marzo 2022. DOI: [10.1109/MPOT.2021.3104638](https://doi.org/10.1109/MPOT.2021.3104638)
- [9] K. Hoffin y E. DeVos, "A Chronology of Video Game Deviance," *Video Games Crime and Next-Gen Deviance*, Emerald Publishing Limited, Bingley, pp. 49-72, Julio 2020. DOI: <https://doi.org/10.1108/978-1-83867-447-220201004>
- [10] <https://www.microsiervos.com/archivo/juegos-y-diversion/videojuegos-mas-antiguos.html>
- [11] S. L. Kent. *La gran historia de los videojuegos*. B DE BOOKS, 2016.

- [12] <https://histinf.blogs.upv.es/2012/12/14/historia-de-las-consolas/>
- [13] K. Collins. *Game sound: an introduction to the history, theory, and practice of video game music and sound design*. Mit Press, 2008.
- [14] N. Shaker, J. Togelius y M. J. Nelson. *Procedural content generation in games*. Ed. 1. Springer Cham, 2016. DOI: <https://doi.org/10.1007/978-3-319-42716-4>
- [15] T. Fullerton. *Game design workshop: a playcentric approach to creating innovative games*. Ed. 2. CRC press, 2014.
- [16]
- <https://www.ign.com/articles/major-publishers-report-aaa-franchises-can-cost-over-a-billion-to-make>
- [17] <https://www.raphkoster.com/2018/01/17/the-cost-of-games/>
- [18] <https://www.raphkoster.com/2017/11/27/some-current-game-economics/>
- [19] D. Green. *Procedural content generation for C++ game development*. Packt Publishing Ltd, 2016.
- [20] J. Togelius, G. N. Yannakakis, K. O. Stanley y C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172-186, Abril 2011. DOI: [10.1109/TCIAIG.2011.2148116](https://doi.org/10.1109/TCIAIG.2011.2148116)
- [21] T. Short y T. Adams. *Procedural generation in game design*. Ed. 1. CRC Press, 2017. DOI: <https://doi.org/10.1201/9781315156378>
- [22] K. Collins, "An introduction to procedural music in video games," *Contemporary Music Review*, vol. 28, no. 1, pp. 5-15, Febrero 2009. DOI: <https://doi.org/10.1080/07494460802663983>

- [23] D. Plans y D. Morelli, "Experience-driven procedural music generation for games," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 3, pp. 192-198, Agosto 2012. DOI: [10.1109/TCIAIG.2012.2212899](https://doi.org/10.1109/TCIAIG.2012.2212899)
- [24] R. Wooller, A. R. Brown, E. Miranda, R. Berry y J. Diederich, "A framework for comparison of process in algorithmic music systems," *Generative arts practice*, pp. 109-124, 2005. Recuperado a partir de <https://www.semanticscholar.org/paper/A-framework-for-comparison-of-process-in-music-Wooller-Brown/525e8d26fc2c11df39b4249b2111915fc025234d>
- [25] <https://docs.unity3d.com/ScriptReference/MonoBehaviour.html>
- [26] <https://docs.unity3d.com/Manual/ExecutionOrder.html>
- [27] <https://github.com/supercollider/supercollider>
- [28] <https://doc.sccode.org/>
- [29]
https://www.youtube.com/watch?v=sCWzcxEgxyo&list=PLPYzvS8A_rTblgN0NTMBPXjmdyNvID0cf&ab_channel=EliFieldsteel
- [30] <https://github.com/jorgegarcia/UnityOSC>
- [31] <https://gist.github.com/vanderaalle/4a638991d313b20fc638f172e3b29add>
- [32] M. Scirea, J. Togelius, P. Eklund y S. Risi, "Metacompose: A compositional evolutionary music composer," *Evolutionary and Biologically Inspired Music, Sound, Art and Design: 5th International Conference, EvoMUSART 2016, Porto, Portugal, March 30--April 1, 2016, Proceedings* 5, pp. 202-217, 2016. DOI: [10.1007/978-3-319-31008-4_14](https://doi.org/10.1007/978-3-319-31008-4_14)
- [33] <https://www.mugglinworks.com/chordmaps/chartmaps.htm>
- [34] <https://mugglinworks.com/chordmaps/GenericMap.pdf>
- [35]
<http://www.elblogdelenguajemusical.com/2018/05/juegos-para-repasar-las-armaduras.html>