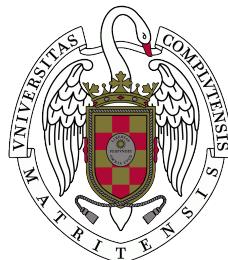

Herramienta de composición de música para videojuegos basada en IA

AI-based music composition tool for video games



Trabajo de Fin de Grado
Curso 2023–2024

Autores

Javier Callejo Herrero

Víctor Manuel Estremera Herranz

Miguel González Pérez

Rodrigo Sánchez Torres

Directores

Miguel Gómez-Zamalloa Gil

Jaime Sánchez Hernández

Grado en Desarrollo de Videojuegos
Facultad de Informática
Universidad Complutense de Madrid

Herramienta de composición de música para videojuegos basada en IA

AI-based music composition tool for video games

Trabajo de Fin de Grado en Desarrollo de Videojuegos

Autores

Javier Callejo Herrero

Víctor Manuel Estremera Herranz

Miguel González Pérez

Rodrigo Sánchez Torres

Directores

Miguel Gómez-Zamalloa Gil

Jaime Sánchez Hernández

Convocatoria: *Junio 2024*

Grado en Desarrollo de Videojuegos

Facultad de Informática

Universidad Complutense de Madrid

27 de mayo de 2024

Dedicatoria

*A Jaime y Miky, por descubrirnos este campo
y por su implicación en el trabajo*

Agradecimientos

A Ismael, por su consejo a la hora de diseñar las redes neuronales.

De Víctor a Marcos, quien fue mi profesor de armonía, de quien he aprendido todo lo que sé. Sin sus enseñanzas, mi trabajo no habría sido posible.

De Víctor para Elisa, quien lo inició todo y gracias a quien estoy donde estoy. Desde mi infancia, como directora de la coral La Petite Esperanza, hasta más adelante como la mejor profesora de piano que he tenido. Sin su influencia, no estaría firmando este proyecto ahora mismo.

A Paula y Marina, por escuchar las canciones generadas durante todos estos meses.

A Ludofonía, por su serie de vídeos sobre música temática.

Resumen

Herramienta de composición de música para videojuegos basada en IA

Este trabajo de fin de grado tiene como propósito crear una aplicación de escritorio que permita a los usuarios generar música temática para sus videojuegos de forma sencilla, sin requerir conocimientos musicales previos. El nombre que le hemos dado a la aplicación es Vanguard Music.

Mediante el uso y estudio de campos como el de la inteligencia artificial y el aprendizaje automático impulsamos la generación de temas musicales que cumplan las expectativas de la aplicación. Vanguard Music conecta con Reaper, un software de edición de audio que nos permite crear y editar pistas de audio de manera profesional, añadiendo los instrumentos y filtros necesarios para sonorizar la música generada por la IA.

Palabras clave

Música para videojuegos, Inteligencia Artificial, Redes Neuronales, Modelos de Markov, Composición automática de música.

Abstract

AI-based music composition tool for video games

This undergrad thesis aims to create a desktop app called Vanguard Music, enabling users to effortlessly craft thematic music for their video games, sans any prior musical knowledge. Through the use and study of fields such as artificial intelligence and machine learning, we will drive the generation of musical themes meeting the app's expectations. Vanguard Music interfaces with *Reaper*, a third-party digital audio work-station, empowering professional audio track creation and editing, by adding the necessary instruments and filters to add sound to the AI-generated music.

Keywords

Music for videogames, Artificial Intelligence, Neural Networks, Markov Models, Automatic music composition.

Índice

| | |
|---|-----------|
| 1. Introducción | 1 |
| 1.1. Motivación | 1 |
| 1.2. Objetivos | 3 |
| 1.3. Plan de trabajo | 4 |
| 1.4. Estructura de la memoria | 4 |
| 2. Preliminares | 7 |
| 2.1. Estado del arte | 7 |
| 2.1.1. Música adaptativa para videojuegos | 7 |
| 2.1.2. Música generativa | 8 |
| 2.1.3. Generación de música como audio | 8 |
| 2.1.4. Generación de música simbólica | 9 |
| 2.2. Pequeña introducción a la música | 10 |
| 2.3. Pequeña introducción a la armonía | 10 |
| 2.3.1. Intervalos | 11 |
| 2.3.2. Escalas | 12 |
| 2.3.3. Acordes | 13 |
| 2.3.4. Modos | 17 |
| 2.4. MIDI | 18 |
| 2.5. Software musical | 19 |
| 2.5.1. ¿Qué es una DAW? | 19 |
| 2.5.2. REAPER | 19 |
| 2.5.3. Instrumentos | 20 |
| 3. Guía de Uso | 23 |

| | | |
|-----------|---|-----------|
| 3.1. | Dependencias de la aplicación | 23 |
| 3.1.1. | Dependencias de Instalación | 23 |
| 3.1.2. | Interfaz gráfica con <i>TKInter</i> | 24 |
| 3.1.3. | Reaper | 24 |
| 3.2. | Preparación del entorno | 25 |
| 3.2.1. | Entorno virtual | 25 |
| 3.2.2. | Ejecución de scripts con Reaper | 25 |
| 3.2.3. | Iniciar la Aplicación | 26 |
| 3.2.4. | Instalación automatizada | 26 |
| 3.3. | Generación de audio | 27 |
| 3.4. | Selección de Temática y Sonorización | 28 |
| 3.4.1. | Temáticas | 28 |
| 3.4.2. | Entorno | 29 |
| 3.4.3. | Efectos | 29 |
| 3.4.4. | Semillas | 30 |
| 3.4.5. | Presets | 30 |
| 3.5. | <i>Renderizado</i> : Descarga de los resultados | 31 |
| 3.6. | Modo avanzado | 31 |
| 3.6.1. | Alternar Generador | 32 |
| 3.6.2. | Temperatura | 32 |
| 3.6.3. | Mezclar temáticas | 32 |
| 3.6.4. | Variar semitonos | 33 |
| 3.7. | Configuración | 34 |
| 3.8. | Ayuda y Tooltips | 34 |
| 4. | Generación de Melodías con <i>Machine Learning</i> | 35 |
| 4.1. | Cómo representamos la música | 35 |
| 4.1.1. | Representación " <i>pitch_duration</i> " | 35 |
| 4.1.2. | Magenta <i>NoteSequence</i> | 36 |
| 4.2. | Dataset utilizado y procesamiento de datos | 36 |
| 4.3. | Cadenas de Markov | 37 |
| 4.3.1. | Entrenamiento de las Cadenas de Markov | 38 |
| 4.3.2. | Generar melodías con Cadenas de Markov | 39 |
| 4.3.3. | Puntos fuertes y débiles de la generación de melodías con Cadenas de Markov | 39 |

| | | |
|-----------|---|-----------|
| 4.4. | Redes Neuronales Recurrentes | 40 |
| 4.4.1. | Introducción a las Redes Neuronales Recurrentes | 40 |
| 4.4.2. | Diseño de nuestra RNR | 43 |
| 4.4.3. | Resultados | 44 |
| 4.5. | Generación con Magenta | 44 |
| 4.5.1. | Paquete de Magenta para Python | 45 |
| 4.5.2. | Paquete de Magenta para JavaScript | 45 |
| 4.5.3. | Magenta en nuestro proyecto | 45 |
| 4.5.4. | Ventajas y desventajas de la generación con Magenta | 45 |
| 5. | Armonización | 47 |
| 5.1. | ¿Qué es armonizar? | 47 |
| 5.2. | Armonización por ventanas | 47 |
| 5.2.1. | Heurística | 50 |
| 5.2.2. | Conclusiones | 54 |
| 5.3. | Armonización por ventanas + | 56 |
| 5.3.1. | Armonización por ventanas de diferentes tamaños | 56 |
| 5.3.2. | Armonización por desplazamiento de ventana | 57 |
| 5.3.3. | Conclusiones | 57 |
| 5.4. | Relacionando acordes | 58 |
| 5.4.1. | Matriz de correspondencia | 59 |
| 5.4.2. | Modelo de predicción de acordes | 60 |
| 5.5. | Reconocimiento de progresiones de acordes | 61 |
| 5.5.1. | Política de elección de progresiones de acordes | 61 |
| 5.5.2. | Secuencia de progresiones sin temor al corte | 62 |
| 5.5.3. | Secuencia de progresiones que <i>loopea</i> | 62 |
| 5.5.4. | Conclusiones | 63 |
| 6. | Búsqueda de nuevos cromatismos musicales | 65 |
| 6.1. | Tratamiento de la melodía original | 65 |
| 6.2. | Modos griegos | 66 |
| 6.2.1. | Nota de color | 66 |
| 6.2.2. | Armonía modal | 67 |
| 7. | Generar Percusión por Ordenador | 69 |

| | | |
|------------|--|-----------|
| 7.1. | Generación de percusión | 69 |
| 7.1.1. | Generación de percusión con Magenta | 69 |
| 7.1.2. | Generación de percusión propia | 70 |
| 7.1.3. | ¿Cómo enfocamos la percusión? | 70 |
| 7.1.4. | <i>Drum Fills</i> | 71 |
| 8. | Sonorización del Proyecto | 75 |
| 8.1. | Presets | 75 |
| 8.2. | Plugins utilizados | 75 |
| 8.2.1. | Instrumentos virtuales | 76 |
| 8.2.2. | Plugins auxiliares | 77 |
| 8.2.3. | <i>BlueAmp</i> | 78 |
| 8.2.4. | Humanizador | 79 |
| 8.2.5. | <i>Delay</i> | 80 |
| 8.2.6. | <i>Reverb</i> | 80 |
| 8.2.7. | Mezcla | 81 |
| 8.2.8. | Efectos en la mezcla | 81 |
| 8.3. | ReaScript | 82 |
| 9. | Cómo arreglar una canción | 83 |
| 9.1. | Temáticas | 83 |
| 9.2. | Las partes fundamentales de una producción | 85 |
| 9.3. | Secciones | 85 |
| 9.4. | Arreglo | 86 |
| 9.5. | Pistas | 86 |
| 9.6. | Melodías | 87 |
| 9.7. | Armonía | 88 |
| 9.8. | Línea de bajo | 88 |
| 9.9. | Arreglos de batería | 90 |
| 9.10. | <i>Ear candy</i> | 90 |
| 10. | Conclusiones y Trabajo Futuro | 93 |
| 10.1. | Generación de melodías | 93 |
| 10.2. | Armonización | 93 |
| 10.3. | Arreglos e instrumentalización | 94 |

| | |
|--|------------|
| 10.4. Aplicación | 95 |
| 10.5. Conclusiones generales | 95 |
| 10.6. Trabajo futuro | 95 |
| Introduction | 99 |
| Contribuciones Personales | 107 |
| Bibliografía | 117 |
| A. Dependencias de Instalación | 119 |
| A.1. Dependencias fundamentales | 119 |
| A.2. Instrumentos virtuales recomendados | 120 |
| B. Representaciones de una canción | 123 |

Índice de figuras

| | | |
|-------|---|----|
| 2.1. | Notas en el piano | 11 |
| 2.2. | Acorde de C en su estado fundamental | 15 |
| 2.3. | Acorde de C en su 1 ^a inversión | 16 |
| 2.4. | Acorde de C en su 2 ^a inversión | 16 |
| 2.5. | Acorde de C en su estado fundamental con sus notas en diferentes octavas | 16 |
| 2.6. | Acorde de C invertido con sus notas en diferentes octavas | 16 |
| 2.7. | Acorde de C representado por múltiples notas | 16 |
| 2.8. | Captura de pantalla de Reaper | 19 |
| 2.9. | Notas MIDI en el rodillo de piano de Reaper. | 20 |
| 2.10. | Un instrumento virtual de guitarra eléctrica, el DVS Guitar. | 20 |
| 2.11. | Plugin de ecualización por defecto de Reaper, ReaEQ | 21 |
| 3.1. | Pestaña <i>Generación</i> | 27 |
| 3.2. | Pestaña Musicalización | 28 |
| 3.3. | Icono de guardado | 31 |
| 3.4. | Pestaña avanzada | 32 |
| 3.5. | Mezclar temáticas | 33 |
| 3.6. | Aleatorización de temáticas | 33 |
| 4.1. | Ejemplo de una cadena de Markov | 37 |
| 4.2. | Representación visual de la cadena obtenida a partir de la matriz de transición | 39 |
| 4.3. | Red neuronal simple, con 1 capa oculta | 41 |
| 4.4. | Despliegue temporal de una RNN | 43 |
| 5.1. | Partitura en 4/4 | 48 |

| | | |
|------|--|----|
| 5.2. | Partitura en 6/8 | 48 |
| 5.3. | Ventana o compás a armonizar | 50 |
| 5.4. | Ventana o compás a armonizar con los ticks clave marcados | 52 |
| 5.5. | Armonización por ventanas de diferentes tamaños | 57 |
| 5.6. | Armonización por desplazamiento de ventana | 58 |
| 7.1. | Ritmo básico de percusión. | 71 |
| 7.2. | Ritmo básico de percusión en el rodillo de piano. | 71 |
| 7.3. | Ejemplo del patrón de percusión resultante en el ejemplo. | 71 |
| 7.4. | Ítem MIDI auxiliar usado para generar <i>drum fills</i> | 72 |
| 7.5. | Arpegiador y resultado final de un <i>drum fill</i> | 72 |
| 7.6. | A la izquierda, ritmo estándar básico de percusión, a la derecha, ritmo estándar de jazz | 73 |
| 7.7. | Ritmo resultante completo | 74 |
| 8.1. | Efectos utilizados en Reaper | 76 |
| 8.2. | Interfaz del plugin Clap Machine de 99Sounds | 77 |
| 8.3. | Interfaz del plugin Zither Renaissance de Sample Science | 77 |
| 8.4. | Interfaz de <i>BlueArp</i> haciendo un arpegio | 78 |
| 8.5. | Acorde de C en MIDI arpegiado | 79 |
| 8.6. | Acorde de C en MIDI arpegiado humanizado | 80 |
| 8.7. | Interfaz del plugin de reverberación Oril River | 81 |
| 9.1. | Arreglo generado en Reaper | 86 |
| 9.2. | Ítems MIDI en complejidad estándar | 87 |
| 9.3. | Ítems MIDI en complejidad repetitiva | 87 |
| 9.4. | Ítems MIDI en complejidad súper repetitiva | 88 |
| 9.5. | Progresión de acordes sin inversión | 89 |
| 9.6. | Ejemplo de configuración de <i>BlueArp</i> y línea de bajo resultante | 89 |
| 9.7. | Otra configuración de <i>BlueArp</i> y línea de bajo resultante | 89 |

Índice de tablas

| | | |
|-------|---|----|
| 2.1. | Cifrado americano | 11 |
| 2.2. | Tabla de intervalos | 11 |
| 2.3. | Escalas mayor y menor | 12 |
| 2.4. | Tonalidades mayores | 12 |
| 2.5. | Tonalidades menores | 13 |
| 2.6. | Otras escalas | 13 |
| 2.7. | Tríadas | 13 |
| 2.8. | Tríadas en C | 13 |
| 2.9. | Escala mayor en grados y en C | 14 |
| 2.10. | Cuatriadas | 14 |
| 2.11. | Comparativa entre escalas (mayor y menor) y tónicas (C y D) | 15 |
| 2.12. | Modos griegos | 17 |
| 2.13. | Ejemplos de modos con distintas tónicas | 18 |
| 4.1. | Ejemplo de una matriz de transición | 38 |
| 4.2. | Ejemplo de matriz de ocurrencia | 38 |
| 4.3. | Ejemplo de matriz de transición calculada a partir de la matriz de ocurrencia | 38 |
| 5.1. | Estado inicial de los pesos de la ventana | 50 |
| 5.2. | Peso por posición de la nota | 51 |
| 5.3. | Pesos tras la valoración de la posición de la nota | 51 |
| 5.4. | Multiplicadores por posición del tick clave | 52 |
| 5.5. | Pesos tras tener en cuenta el multiplicador por tick clave | 53 |
| 5.6. | Pesos tras sumar las valoraciones por tick clave penalizado | 53 |
| 5.7. | Pesos finales | 54 |

| | |
|---|-----|
| 5.8. Tabla de progresiones | 61 |
| 6.1. Modos mayores | 66 |
| 6.2. Modos mayores | 66 |
| 6.3. Armonía del modo dórico | 67 |
| A.1. Programas externos fundamentales | 119 |
| A.2. Plugins de efectos fundamentales | 120 |
| A.3. Instrumentos virtuales recomendados | 121 |
| A.4. Instrumentos virtuales recomendados | 122 |
| B.1. Relación entre nota y su <i>pitch</i> en la primera octava | 123 |

Capítulo 1

Introducción

“¡Qué bonito, qué chulo!”
— Jaime Sánchez

1.1. Motivación

La música ha sido desde siempre un poderoso medio de expresión, capaz de transmitir una amplia gama de emociones. En campos como el cine y los videojuegos resulta crucial, pues es capaz de provocar fuertes sentimientos al público, sumergiéndolo en mundos ficticios. Cualquier desarrollador de videojuegos que quiera cautivar al jugador necesita apoyarse en la música y en los efectos de sonido.

En la actualidad hay muchas formas de abordar la composición y producción musical, pero la más común es el uso de una DAW (*Digital Audio Workstation*). Una DAW es una herramienta software que permite grabar, editar y mezclar pistas de audio, así como secuenciar música. Secuenciar música consiste en programar y reproducir eventos musicales, siendo el formato más extendido para dichos eventos el MIDI. MIDI es un estándar que define eventos básicos de inicio y fin de nota, así como otros que contienen información más específica sobre la pieza musical. Este tipo de eventos pueden ser interpretados por plugins. Específicamente los plugins que interpretan el MIDI son los llamados instrumentos virtuales. Los instrumentos virtuales pueden emular el timbre de instrumentos reales, ya sea utilizando muestras grabadas o síntesis de audio digital, para poder generar sonido en tiempo real.

La gran ventaja que tiene el MIDI es la modificación y edición de una canción de forma accesible, sin tener que volver a grabar la interpretación de la pieza. Esto se debe a que la notación musical está desligada de los instrumentos. Un archivo MIDI representa una partitura digital. Es, en definitiva, una notación musical simbólica de la música.

La otra alternativa de trabajo, donde no hay separación entre la notación musical y los instrumentos, consiste en trabajar directamente con audio digital. En este enfoque, los músicos graban las interpretaciones en vivo de los instrumentos y voces, capturando las ondas sonoras generadas por estos. Las grabaciones se editan y mez-

clan para crear la pista final. Este método permite capturar la interpretación y el timbre único de cada músico e instrumento, proporcionando un carácter auténtico y natural a la música, pero es muy limitado a la hora de realizar cambios en la composición una vez que las grabaciones están hechas.

En el campo de los videojuegos, la forma de composición de música tradicional no termina de encajar, pues una canción tradicionalmente está compuesta para ser escuchada de principio a fin y ser una experiencia completa. En los videojuegos, no sabemos a priori cuánto tiempo va a estar un jugador en una zona escuchando una canción, por lo que es crucial que esta no resulte repetitiva y cicle adecuadamente. Pero los problemas no terminan ahí, en los videojuegos, a diferencia del cine, no podemos ajustar qué canción va a sonar en cada minuto, ya que es un medio con mayor grado de libertad. Por estos motivos surge la música adaptativa. La música adaptativa es un estilo de música que se compone específicamente para que pueda ir cambiando en función de una serie de parámetros.

Al componer música adaptativa se puede trabajar con composición horizontal o vertical. La composición horizontal consiste en utilizar saltos temporales en una canción ya dividida en secciones, que se pueden ciclar. Estos saltos se definen para secciones que tengan distinta intensidad o instrumentos, para adaptarse mejor a una zona o momento específico. En la composición vertical se crean varias pistas o capas que se activan o desactivan para generar diferentes secciones. Aunque esto depende en gran medida del estilo musical, es habitual que podamos encontrar las siguientes capas: melodía, armonía, bajo y base rítmica entre otros. La melodía es el elemento protagonista y más distintivo, normalmente es una voz o instrumento con patrones definidos que lleva el liderazgo de la canción. La armonía acompaña y complementa a la melodía, y aparece normalmente en forma de acordes tocados por uno o varios instrumentos secundarios a la melodía. El bajo se encarga de crear una base sobre la que destacarán la melodía y armonía, generando las frecuencias más graves de la canción. La base rítmica finalmente, como su nombre indica, define el ritmo de la canción que seguirán el resto de elementos de esta. Combinando todas estas capas conseguimos una composición musical completa y, en la música adaptativa, las activamos o desactivamos para poder generar variedad y hacer que la música se ajuste a lo que está sucediendo en el videojuego, introduciendo más o menos intensidad por ejemplo. Esto es solo un ejemplo típico, la música adaptativa puede componerse de muchas más capas y diferentes elementos que permiten toda gama de combinaciones. En definitiva, es un campo muy rico y abierto a la imaginación.

Este trabajo busca desarrollar una herramienta que permita generar de manera automática música de distintas temáticas para poder usar en videojuegos.

En los últimos años, ha habido muchos avances en la generación automática de música con técnicas de IA y *machine learning*. Hay 2 maneras principales de generar música: la generación de audio y la generación a nivel simbólico.

La generación de audio produce la propia pieza musical en formatos como MP3 o WAV, normalmente utilizando técnicas de aprendizaje profundo o atención. Este tipo de generación no se explora en este trabajo, pero es una metodología válida que cada año no hace sino demostrar su potencial.

A nivel simbólico, la generación produce las notas que componen la canción en formato MIDI o similares, de forma que esta se puede *renderizar* utilizando instrumentos virtuales y una DAW, como se ha mencionado anteriormente. Como ya hemos comentado, la generación simbólica, a diferencia de la generación de audio, permite poder realizar ajustes sobre la canción y variar partes o instrumentos de esta. Este es el enfoque que hemos seleccionado para nuestro trabajo.

1.2. Objetivos

Como se ha mencionado, el objetivo principal de este trabajo es crear una herramienta que permita generar automáticamente música para videojuegos. La herramienta va a tener diferentes módulos que se encargan de generar distintas capas musicales, siendo estas melodía, armonía, bajo y base rítmica. Esta generación se hace a nivel simbólico y el resultado se utilizará para componer un arreglo en una DAW, utilizando una serie de timbres preseleccionados que interpretarán dichas generaciones. Este enfoque permite tener mucha flexibilidad a la hora de poder ser editado por el usuario tanto en el sentido musical, pudiendo editar las notas, como a nivel de timbres.

En concreto se plantean los siguientes objetivos específicos:

1. Conseguir generar melodías a través de distintas vías. Por un lado se investigará el uso de herramientas existentes, como por ejemplo Magenta de Google. Por otro lado se explorará el desarrollo de modelos propios de *machine learning* con este objetivo.
2. Conseguir armonizar la melodía generada de forma que resulte agradable al oído humano, ya sea a través del uso de inteligencia artificial o heurísticas propias.
3. Partiendo de las melodías y armonías generadas, realizar el arreglo completo de una canción, siguiendo una estructura que facilite su uso como música para videojuegos, ya sea de forma adaptativa o lineal.
4. Sonorizar las notas musicales generadas haciendo una selección apropiada de instrumentos virtuales en base a unas temáticas predefinidas.
5. Crear una aplicación de escritorio para Windows que integre las posibilidades mencionadas anteriormente de una forma amigable al usuario.
6. Conectar la aplicación desarrollada con Reaper, agregando los instrumentos necesarios y produciendo la mezcla a partir de la temática y modificadores seleccionados en nuestra aplicación.

1.3. Plan de trabajo

El tener objetivos claramente separados en cuanto a funcionalidad, nos permite separar el trabajo de manera más eficiente en módulos independientes, como una cadena de producción separada en etapas que se une en la aplicación final.

1. Investigación y estudio del campo de la inteligencia artificial aplicada a la generación musical. Durante esta fase, revisaremos la literatura existente, investigaremos los enfoques y algoritmos más relevantes y nos familiarizaremos con las herramientas y recursos disponibles.
2. Desarrollo del algoritmo de generación de melodías. Utilizaremos técnicas de aprendizaje automático y modelos generativos para crear sistemas capaces de componer melodías originales y variadas. Además, se explorará la herramienta Magenta ya existente.
3. Implementación de un algoritmo que armonice las melodías generadas. En esta etapa, diseñaremos y desarrollaremos heurísticas basadas en la teoría musical con el propósito de generar los acordes que acompañen a la melodía.
4. Estudiar, comprender y aplicar la API de ReaScript para programar en Reaper. Usaremos ReaScript con Python para cargar todos los archivos MIDI generados en Reaper, ordenarlos según un arreglo para crear una canción completa y cargar los plugins que harán sonar esos archivos MIDI.
5. Selección y configuración de instrumentos. Investigaremos y seleccionaremos una variedad de instrumentos musicales virtuales que se ajusten a las temáticas predefinidas de los videojuegos. Configuraremos estos instrumentos para que reproduzcan las melodías generadas de manera convincente al oído humano.
6. Desarrollo de la aplicación de escritorio. Utilizaremos las tecnologías adecuadas para desarrollar una interfaz de usuario intuitiva y amigable que permita a los usuarios generar, armonizar y sonorizar sus propias composiciones musicales para videojuegos. En concreto, usaremos TkInter, la biblioteca por defecto de Python para la creación de interfaces gráficas de usuario (GUI).

1.4. Estructura de la memoria

En el Capítulo 2 se presentarán los conceptos preliminares que hay que conocer para profundizar posteriormente en otras secciones. Este amplía algunos conceptos presentados en esta introducción.

El Capítulo 3 consiste en una guía de uso de la herramienta, explicando todos los aspectos desde su instalación y configuración hasta su uso para generar canciones.

El Capítulo 4 describe los modelos utilizados para generar melodías, comparando sus resultados y detallando su diseño e implementación. También se incluye la integración de Magenta a nuestra aplicación.

En el Capítulo 5 se profundiza en la armonización de una melodía, aplicando conceptos de teoría musical en algoritmos propios.

En el Capítulo 6 se modifican las melodías y se utilizan diversas técnicas de armonización con el objetivo de obtener nuevos colores a partir de una misma generación.

El Capítulo 7 detalla la generación de la base de percusión que se utiliza en la composición de canciones.

En el Capítulo 8 se explica cómo se eligen los sonidos e instrumentos virtuales que se aplicarán a las distintas partes generadas.

En el Capítulo 9 se profundiza sobre el arreglo de canciones para las distintas temáticas, describiendo cada parte del proceso de producción de una canción.

Finalmente, el Capítulo 10 contiene unas breves conclusiones sobre los resultados obtenidos con el trabajo así como una serie de mejoras y trabajo a realizar a futuro.

Capítulo 2

Preliminares

2.1. Estado del arte

Cada día se producen grandes avances en muchos campos con el uso de inteligencia artificial, sin embargo, en el campo de la composición musical no está tan desarrollada como en otras áreas. A pesar de esto sí existen algunas herramientas que podemos usar como referencias para el desarrollo de nuestro trabajo.

2.1.1. Música adaptativa para videojuegos

Debido a que en un videojuego no podemos saber cuánto tiempo permanecerá el jugador en una zona, ni en qué instante de tiempo realizará acciones que podríamos destacar usando la música, la forma tradicional de hacer música no funciona en los videojuegos. Es aquí donde entra la música adaptativa.

La música adaptativa consiste en componer música de forma que esta se acomode a lo que esté sucediendo en el videojuego y se pueda escuchar de forma continua. Para esto se utilizan distintas técnicas que se diferencian de la composición tradicional. Hay 2 maneras principales de componer música adaptativa:

- **Composición horizontal:** Consiste en componer una canción formada por varias secciones y realizar saltos temporales entre dichas secciones. Este tipo de composición se acerca más a la tradicional, teniendo como peculiaridad que las secciones deben de funcionar por separado y poder ciclarse, para que la música pueda tener una duración ilimitada.
- **Composición vertical:** En esta, se utilizan diferentes capas que se activan o desactivan para crear diferentes secciones. Estas capas pueden incluir: melodía, armonía, línea de bajo, base rítmica, ambientes, etc. En este trabajo se generarán diferentes capas para utilizar este tipo de composición.

2.1.1.1. FMOD

FMOD es un motor de efectos de sonido para videojuegos y aplicaciones desarrollado por Firelight Technologies, el cual consiste principalmente en una API de código. Se utiliza para sonorizar videojuegos y además, puede usarse para crear música adaptativa.

Posee una aplicación de escritorio llamada FMOD Studio, la cual permite tener una interfaz gráfica para realizar la mayor parte de las acciones que se pueden realizar utilizando la API de código.

Con FMOD podemos crear variables configurables desde el propio juego, para controlar aspectos como qué parte de la canción suena, volumen, intensidad de la música, etc. Además, presenta la posibilidad de omitir el uso de *timeline* para crear composiciones *no lineales*, es decir, música que no tiene una estructura temporal fija, sino que depende de la configuración de las variables en un momento dado.

2.1.2. Música generativa

La música generativa es un tipo de música la cual se va creando y evolucionando a medida que se va reproduciendo, de forma que cambia cada vez que suena. Uno de los mayores exponentes de esta corriente musical es Brian Eno, un artista enfocado principalmente a la composición de música *ambient*. En el trabajo de Eno podemos apreciar composiciones que utilizaban software de sintetizadores para introducir partes que se reproducen o no de manera aleatoria y así crear una canción única cada vez que se reproducen.

Este tipo de música se puede aplicar a videojuegos y, aunque no representa el grueso de nuestro trabajo, es una inspiración a la hora de plantear la composición de las distintas partes de nuestras canciones.

2.1.3. Generación de música como audio

Como se mencionó en la introducción, se puede trabajar con la música mediante grabaciones de interpretaciones de artistas, sin tener notas ni representaciones simbólicas. La generación de audio es un campo con muchos avances en los años recientes. Específicamente, desde el inicio del trabajo, ha sufrido una mejora tan grande que ha pasado a ser la opción predilecta para los usuarios. Normalmente, este tipo de modelos se basan en introducir *prompts* para generar una canción del estilo que se quiera.

Esta generación tiene como ventaja el gran acceso a información disponible, ya que hay más *datasets* de música ya *renderizada* que de música simbólica. Por este motivo, últimamente se ha visto su capacidad de generar audio que puede pasar por canciones compuestas por una persona.

Sin embargo, este método no proporciona mucha flexibilidad a la hora de poder variar secciones de la composición una vez generadas. No entra en los objetivos de

nuestro trabajo, aunque podría ser una opción más a explorar para generar secciones específicas de música adaptativa.

2.1.4. Generación de música simbólica

Además de como audio, se puede trabajar con la música a nivel simbólico. Como se mencionó en la introducción, el uso de música simbólica permite tener la flexibilidad de editar las notas y timbres de las canciones, por lo que se prefiere en este trabajo sobre la generación de audio. Podemos destacar 2 proyectos de investigación que han sido cruciales para el desarrollo de nuestros modelos.

2.1.4.1. Generación con Cadenas de Markov

En el paper de Ilana Shapiro (2021), se detalla cómo utilizar cadenas de Markov para la generación de fragmentos musicales con el estilo de un compositor específico.

En este, se centran en los aspectos matemáticos del modelo de Markov, no tanto en la calidad musical de la generación, que resulta ser más una prueba de concepto que una composición seria. Sin embargo, el tratamiento simbólico de la música en dicho paper resulta interesante y se ha utilizado como apoyo para desarrollar nuestros modelos de generación.

2.1.4.2. Magenta

Magenta es un proyecto de investigación propiedad de Google, compuesto por varios modelos de *machine learning*. Estos modelos están entrenados para generar tanto música como dibujos.

Particularmente, los modelos entrenados con música pueden realizar varias funcionalidades. Existen modelos que generan melodías, modelos que continúan una melodía dada, modelos que generan baterías, *autoencoders* que permiten humanizar baterías, armonización de una melodía dada... Podemos encontrar estos modelos y más en el repositorio de Magenta (Magenta, 2023) o en su web (Magenta, s.f.).

Magenta contiene además un plugin para la DAW Ableton Live llamado Magenta Studio (Magenta, 2022b). Dicho plugin fue además portado a aplicación de escritorio para Windows. En este plugin encontramos varios programas que desmuestran la funcionalidad de Magenta:

- **Generate:** Genera melodías de 4 compases.
- **Continue:** Continúa una melodía de entrada un número N de compases.
- **Drumify:** Crea una base de batería dado un archivo MIDI de input.
- **Interpolate:** Crea una melodía o base de batería combinando 2 de entrada.
- **Groove:** Humaniza un archivo MIDI de batería para que suene como si lo hubiera grabado una persona.

2.2. Pequeña introducción a la música

Nos centraremos en la notación musical occidental, utilizada prácticamente en todas las canciones que escuchamos diariamente, en la que los sonidos se dividen en conjuntos de 12 notas musicales, cada una con un nombre diferente, definiendo nota musical como un símbolo que representa una frecuencia determinada. Para entender de dónde viene esta división utilizaremos como base la nota A4 (La4), con una frecuencia de 440Hz y duplicaremos su frecuencia a 880Hz, obteniendo así A5 (La5). Estas dos notas comparten nombre, ya que si las escucháramos nos sonaría muy familiares entre sí. Esto tiene una explicación física: al reproducir esa frecuencia se generan también sus armónicos, que son múltiplos enteros de la frecuencia fundamental. Se llama octava a la distancia que separa una frecuencia de su duplicación, esto explica ahora los números escritos al lado de cada nota musical: A3 = 220Hz, A4 = 440Hz, A5 = 880Hz. Sabiendo esto se decidió entonces dividir la octava en 12 notas musicales. Sacamos como conclusión que el conjunto de sonidos utilizados está comprendido por las 12 notas musicales, cada una pudiéndose encontrar en una octava distinta.

Entrando en otra capa de profundidad, en la música se pueden encontrar diversos elementos si nos fijamos en las notas que conforman una canción, como por ejemplo el ritmo, sin embargo, en los dos en los que se quiere hacer inciso ahora son:

- **Melodía:** evolución horizontal de las notas a lo largo de una canción. Es la parte más reconocible de esta. En una composición musical es la parte que generalmente se canta o se toca de manera prominente. Según Dahlhaus (1978), “la melodía es una sucesión de sonidos musicales que se percibe como una sola entidad”.
- **Armonía:** asociaciones verticales de las notas. La armonía proporciona un acompañamiento sonoro a la melodía principal y contribuye a enriquecer la textura musical. Como explica Schoenberg (1911), “la armonía es la combinación de sonidos simultáneos y el estudio de su conexión” .

2.3. Pequeña introducción a la armonía

Como se acaba de explicar, la armonía se refiere a la combinación simultánea de notas musicales que suenan de manera agradable y coherente entre sí. Para entender tanto la armonía, como los algoritmos que se quieren mostrar en futuros apartados del TFG, es necesario interiorizar los siguientes conceptos, que se intentarán explicar de la manera más resumida posible. Cabe recalcar que el análisis a continuación está basado en los siguientes libros: Herrera (1988), *Teoría Musical y Armonía Moderna Vol. I* y Herrera (2011), *Teoría Musical y Armonía Moderna Vol. II*. Es decir, esta explicación se realiza bajo el contexto de la armonía moderna, la cual difiere en algunos puntos de la clásica. Sin embargo, al tratarse de una introducción, la mayoría de conceptos que van a ser mostrados se comparten entre ambas doctrinas. A pesar

de ello, una de las primeras cosas en las que sí difieren es en el nombre de las notas. A partir de ahora se utilizará el cifrado americano: (Figura 2.1)

| Nota | Do | Re | Mi | Fa | Sol | La | Si |
|---------|----|----|----|----|-----|----|----|
| Cifrado | C | D | E | F | G | A | B |

Tabla 2.1: Cifrado americano

2.3.1. Intervalos

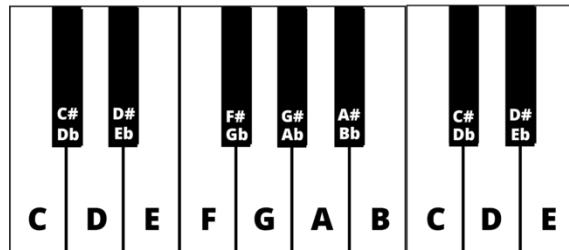


Figura 2.1: Notas en el piano

Un intervalo es la distancia que existe entre dos notas. Un semitono es la distancia mínima que puede existir entre dos notas. Un tono equivale a dos semitonos. Por ejemplo, en la Figura 2.1 se observa que la distancia entre el primer C y el primer E es de 4 semitonos (las teclas negras también cuentan como notas musicales aunque tengan nombres un poco especiales). Entre C y C# o entre E y F, por ejemplo, hay un semitono. Dependiendo del número de semitonos que tenga un intervalo, este tendrá un nombre y simbología diferente: (Tabla 2.2)

| Símbolo | Nombre | Nº Semitonos |
|---------|-------------------|--------------|
| 1 (T) | Tónica | 0 |
| b2 | Segunda menor | 1 |
| 2 | Segunda mayor | 2 |
| b3 | Tercera menor | 3 |
| 3 | Tercera mayor | 4 |
| 4 | Cuarta justa | 5 |
| #4 | Cuarta aumentada | 6 |
| 5b | Quinta disminuida | 6 |
| 5 | Quinta justa | 7 |
| b6 | Sexta menor | 8 |
| 6 | Sexta mayor | 9 |
| b7 | Séptima menor | 10 |
| 7 | Séptima mayor | 11 |
| 8 | Octava | 12 |

Tabla 2.2: Tabla de intervalos

Al observar la Tabla 2.2 se puede deducir por contexto que el símbolo 'b' (bemol) reduce en uno el número de semitonos del grado original, así como el símbolo '#' (sostenido) los aumenta en uno. Por ejemplo, #6 sería equivalente a una séptima menor (b7), con 10 semitonos ambos.

Con la información de esta tabla, ya podríamos entender afirmaciones tales como 'la quinta justa de C es G' o 'la tercera mayor de B es D#', por ejemplo, si nos ponemos a contar los semitonos más detenidamente (piano de referencia en la Figura 2.1).

Cabe recalcar que, evidentemente, existen intervalos que se salen de la octava, con más de 12 semitonos. Algunos no muy lejanos tienen incluso nombre propio, por ejemplo, una novena mayor (9), de 14 semitonos. A pesar de ello, lo que se hará en estos casos es interpretar ese intervalo como su relativo si la nota estuviese en el 'mismo rango' que la tónica (nota desde la cual se mide el intervalo). Así, una novena mayor sería equivalente a una segunda mayor, de 2 semitonos y un intervalo de 42 semitonos será interpretado como una quinta disminuida ($42 \bmod 12 = 6$ semitonos), por ejemplo. De hecho, siguiendo esta definición, el propio intervalo de octava es redundante, ya que esta es equivalente a la tónica y cumple la misma función ($12 \bmod 12 = 0$ semitonos). Por ejemplo, la octava de C4 es C5. Por ello, el intervalo de octava también se puede omitir de la lista.

2.3.2. Escalas

Se define escala como una secuencia de intervalos ascendentes entre los 12 previos a la octava cuyo primero es siempre la tónica. Tradicionalmente, la mayoría de escalas están formadas por 7 notas o intervalos, las más utilizadas en la música que escuchamos hoy en día son la mayor y la menor: (Tabla 2.3)

| | | | | | | | |
|---------------------|-------|---|----|---|---|----|----|
| Escala mayor | 1 (T) | 2 | 3 | 4 | 5 | 6 | 7 |
| Escala menor | 1 (T) | 2 | b3 | 4 | 5 | b6 | b7 |

Tabla 2.3: Escalas mayor y menor

Este conjunto de intervalos representa una especie de esqueleto, un molde con el cual se puede crear una sucesión de notas musicales si se establece como tónica cualquiera de las 12 existentes. Cuando se crea una sucesión de grados respecto a una tónica utilizando cualquiera de estas dos escalas, mayor o menor, se estaría hablando de tonalidad. Un grado es la numeración de una nota dentro de la escala a la que pertenece y se escribe en números romanos. Aquí unos cuantos ejemplos de tonalidades mayores y menores: (Tablas 2.4 y 2.5 respectivamente)

| | I | II | III | IV | V | VI | VII |
|-----------------|----|----|-----|----|---|----|-----|
| C mayor | C | D | E | F | G | A | B |
| D mayor | D | E | F# | G | A | B | C# |
| Bb mayor | Bb | C | D | Eb | F | G | A |

Tabla 2.4: Tonalidades mayores

| | I | II | bIII | IV | V | bVI | bVII |
|-----------------|----|----|------|----|---|-----|------|
| C menor | C | D | Eb | F | G | Ab | Bb |
| D menor | D | E | F | G | A | Bb | C |
| Bb menor | Bb | C | Db | Eb | F | Gb | Ab |

Tabla 2.5: Tonalidades menores

A parte de estas dos escalas, existen muchas otras más que se utilizan regularmente a la hora de componer música. Algunos ejemplos: (Tabla 2.6)

| | | | | | | |
|--------------------------|-------|----|----|----|----|-------|
| Pentatónica mayor | 1 (T) | 2 | 3 | 5 | 6 | |
| Pentatónica menor | 1 (T) | b3 | 4 | 5 | b6 | |
| Hexatónica | 1 (T) | 2 | 3 | #4 | #5 | #6 |
| Menor armónica | 1 (T) | 2 | b3 | 4 | 5 | b6 7 |
| Frigia | 1 (T) | b2 | b3 | 4 | 5 | b6 b7 |
| Mixolidia | 1 (T) | 2 | 3 | 4 | 5 | 6 b7 |

Tabla 2.6: Otras escalas

2.3.3. Acordes

Aunque no todo el mundo estaría de acuerdo con esta definición, vamos a decir que un acorde es un conjunto de dos o más notas tocadas de forma simultánea. La definición es en cierta medida similar a la de una escala, ya que un acorde no deja de ser un conjunto de intervalos ascendentes, en el que el primero es siempre la tónica del acorde. También se cumple esa propiedad de 'molde' a la hora de establecer una nota musical como tónica del acorde. Los acordes más comunes están formados por tres notas (tríadas) y dependiendo del conjunto de intervalos se pueden conseguir diferentes sonoridades: (Tablas 2.7 y 2.8)

| Nombre | Símbolo | Intervalos | | |
|--------------------------|---------|------------|----|----|
| Acorde mayor | | 1 (T) | 3 | 5 |
| Acorde menor | - | 1 (T) | b3 | 5 |
| Acorde aumentado | + | 1 (T) | 3 | #5 |
| Acorde disminuido | -b5 | 1 (T) | b3 | b5 |

Tabla 2.7: Tríadas

| Nombre | Símbolo | Notas | | |
|---------------------|---------|-------|----|----|
| C mayor | C | C | E | G |
| C menor | C- | C | Eb | G |
| C aumentado | C+ | C | E | G# |
| C disminuido | C-b5 | C | Eb | Gb |

Tabla 2.8: Tríadas en C

Se define como armonía de una escala el conjunto de acordes que se pueden formar utilizando únicamente las notas (o intervalos) de la escala. Esto puede chocar con la definición de acorde, ya que, como tal, puede haber miles de tipos de acordes diferentes si atendemos a toda la combinatoria, así que, por ahora, solo tendremos en cuenta los tipos de acordes definidos en la Tabla 2.7. Este concepto es más difícil de deducir y explicar, así que recomiendo la visualización de este¹ vídeo en el que el autor encuentra la armonía de la escala C mayor y pasa el resultado a grados, además de repasar otros conceptos vistos por encima anteriormente. Se obtiene el siguiente resultado: (Tabla 2.9)

| Grado | Acorde | Nota | Acorde |
|-------|--------|------|--------|
| I | | C | C |
| II | - | D | D- |
| III | - | E | E- |
| IV | | F | F |
| V | | G | G |
| VI | - | A | A- |
| VII | -b5 | B | B-b5 |

Tabla 2.9: Escala mayor en grados y en C

Como se puede observar, en la escala mayor se forma una tríada mayor a partir de los grados 1, 4 y 5, una menor a partir de los grados 2, 3 y 6, una disminuida a partir del grado 7 y no existe ningún grado del cual se forme una tríada aumentada. Esto significa que sea cual sea la tónica de una escala, en este caso, la escala mayor, los acordes correspondientes a cada grado serán siempre del mismo tipo. Por lo tanto, si se quiere deducir la armonía de una escala diferente, por ejemplo, la escala menor, los acordes correspondientes a cada grado serán distintos a los de la escala mayor, pero serán del mismo tipo si se varía la tónica. Además, el hecho de que solo se forme un acorde por cada grado es debido a una particularidad propia de la escala mayor y a que hemos escogido un número muy reducido de acordes. Vamos a tener en cuenta ahora el siguiente grupo de acordes de cuatro notas (cuatríadas), que se sumarán al anterior grupo: (Tabla 2.10)

| Nombre | Símbolo | Intervalos | | | |
|---------------------------------|---------|------------|----|----|----|
| Acorde mayor con séptima | maj7 | 1 (T) | 3 | 5 | 7 |
| Acorde menor con séptima | -7 | 1 (T) | b3 | 5 | b7 |
| Acorde de dominante | 7 | 1 (T) | 3 | 5 | b7 |
| Acorde semidisminuido | -7b5 | 1 (T) | b3 | b5 | b7 |

Tabla 2.10: Cuatríadas

A continuación un par de ejemplos que evidencian el anterior párrafo: (Tabla 2.11)

¹<https://www.youtube.com/watch?v=dMwVB3BWcjI>

| Escala mayor | | | | | | | | |
|--------------|---------|------|------|---------|-------|------|---------|--------|
| Grado | Acordes | | Nota | Acordes | | Nota | Acordes | |
| I | | maj7 | C | C | Cmaj7 | D | D | Dmaj7 |
| II | - | -7 | D | D- | D-7 | E | E- | E-7 |
| III | - | -7 | E | E- | E-7 | F# | F#- | F#-7 |
| IV | | maj7 | F | F | Fmaj7 | G | G | Gmaj7 |
| V | | 7 | G | G | G7 | A | A | A7 |
| VI | - | -7 | A | A- | A-7 | B | B- | B-7 |
| VII | -b5 | -7b5 | B | B-b5 | B-7b5 | C# | C#-b5 | C#-7b5 |

| Escala menor | | | | | | | | |
|--------------|---------|------|------|---------|--------|------|---------|--------|
| Grado | Acordes | | Nota | Acordes | | Nota | Acordes | |
| I | - | -7 | C | C- | C-7 | D | D- | D-7 |
| II | -b5 | -7b5 | D | Db-b5 | Db-7b5 | E | E-b5 | E-7b5 |
| bIII | | maj7 | Eb | E | Emaj7 | F | F | Fmaj7 |
| IV | - | -7 | F | F- | F-7 | G | G- | G-7 |
| V | - | -7 | G | G- | G-7 | A | A- | A-7 |
| bVI | | maj7 | Ab | Ab | Abmaj7 | Bb | Bb | Bbmaj7 |
| bVII | | 7 | Bb | Bb | Bb7 | C | C | C7 |

Tabla 2.11: Comparativa entre escalas (mayor y menor) y tónicas (C y D)

2.3.3.1. Inversiones

Un acorde en su estado fundamental es la forma más básica en la que se puede representar dicho acorde. Es el resultado de plasmar en el 'molde' del acorde (Tabla 2.7) la tónica del acorde deseada, por ejemplo C, como se mostró anteriormente en la Tabla 2.8. En la Figura 2.2 se muestra un ejemplo con el acorde de C (Do mayor), conformado por las notas **C**, **E** y **G**.

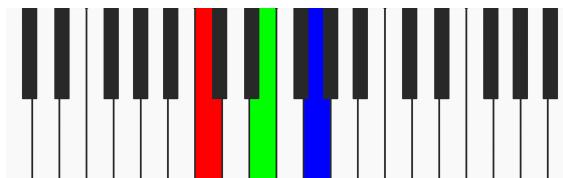


Figura 2.2: Acorde de C en su estado fundamental

Una inversión consiste en la reordenación de las notas del acorde de forma que la nota mas grave no es la tónica del mismo. Por ejemplo, en una triada la nota más grave también puede ser la segunda o tercera nota del acorde, que corresponderían a la 1º y 2º inversión: (Figuras 2.3 y 2.4 respectivamente)

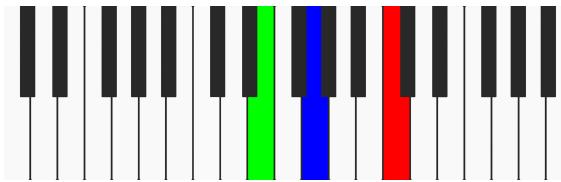


Figura 2.3: Acorde de C en su 1^a inversión

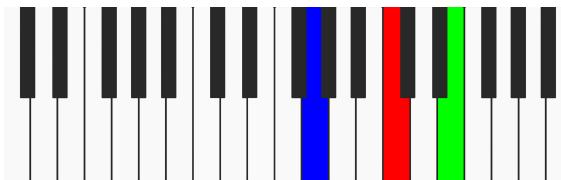


Figura 2.4: Acorde de C en su 2^a inversión

Esta no es la única forma en la que se puede encontrar un mismo acorde. Las notas que lo forman no tienen por qué ir consecutivas dentro de la misma octava o rango: (Figuras 2.5 y 2.6)

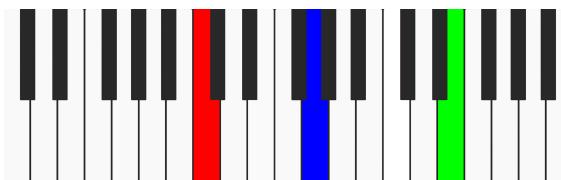


Figura 2.5: Acorde de C en su estado fundamental con sus notas en diferentes octavas

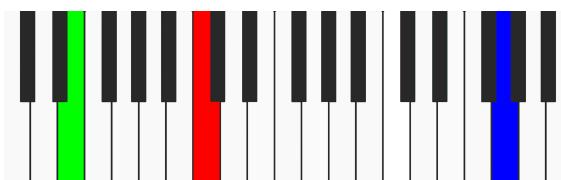


Figura 2.6: Acorde de C invertido con sus notas en diferentes octavas

Por supuesto, repetir notas del mismo acorde en diferentes octavas también se considera tocar dicho acorde: (Figura 2.7)

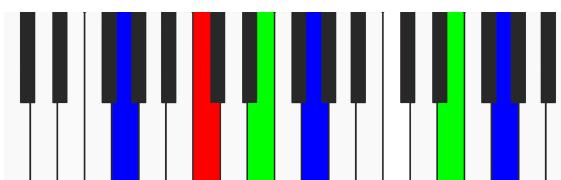


Figura 2.7: Acorde de C representado por múltiples notas

Con todos los ejemplos se quiere dejar claro que, a pesar de toda la combinatoria de representaciones que puede llegar a tener un mismo acorde y de las ligeras diferencias entre sus sonoridades, armónicamente simbolizan lo mismo, en este caso, el acorde de C.

2.3.3.2. Arpegios

Relacionado con las inversiones en el sentido de representar un mismo acorde pero de diferentes maneras, un arpegio se define como la secuencia de las notas del acorde tocadas de manera individual y sucesiva, pudiéndose encontrar estas en multitud de patrones diferentes. De nuevo, aunque la representación del acorde cambie, armónicamente simbolizaría lo mismo.

2.3.4. Modos

Los modos de una escala son las diferentes secuencias de intervalos que se logran al comenzar una nueva escala desde los diferentes grados de la misma. La forma más sencilla de entenderlo es con un ejemplo en términos absolutos y relativizar cada escala obtenida pasándola a intervalos como se hace en la Tabla 2.12 con la escala de C mayor (C jónico, en este contexto modal). Los modos de la escala mayor son los más utilizados, son los conocidos como modos griegos, cada uno conocido con un nombre propio. A pesar de ello, este mismo proceso se puede realizar con cualquier escala.

| | C | D | E | F | G | A | B | C | D | E | F | G | A |
|-----------|---|---|---|----|----|---|----|----|----|---|----|----|----|
| Jónico | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | | | | | |
| Dórico | | 1 | 2 | b3 | 4 | 5 | 6 | b7 | | | | | |
| Frigio | | | 1 | b2 | b3 | 4 | 5 | b6 | b7 | | | | |
| Lidio | | | | 1 | 2 | 3 | #4 | 5 | 6 | 7 | | | |
| Mixolidio | | | | | 1 | 2 | 3 | 4 | 5 | 6 | b7 | | |
| Eólico | | | | | | 1 | 2 | b3 | 4 | 5 | b6 | b7 | |
| Locrio | | | | | | | 1 | b2 | b3 | 4 | b5 | b6 | b7 |

Tabla 2.12: Modos griegos

Viendo la tabla se pueden sacar varias conclusiones. La primera es que cada escala tiene tantas relativas con diferente tónica como modos tenga la propia escala. Aquí un ejemplo que sirve también para consolidar puntos descritos anteriormente: C jónico y D dórico son relativas entre sí porque comparten el mismo conjunto de notas, a pesar de tener distinta tónica; C jónico y C dórico no lo son porque aunque que comparten la misma tónica, el conjunto de notas que lo conforman es diferente, ya que el 'esqueleto' de sus escalas es diferente. La segunda conclusión es que el modo eólico es idéntico a la escala menor y que por lo tanto, en el caso del ejemplo, C mayor (jónico) es relativo a A menor (eólico). Hasta ahora se ha omitido, pero toda escala mayor tiene una relativa menor y viceversa.

Para consolidar aún más el concepto de los modos a continuación se van a mostrar varios ejemplos de modos griegos con tónicas diferentes a las descritas en la anterior tabla: (Tabla 2.13)

| Modo dórico | 1 (T) | 2 | b3 | 4 | 5 | 6 | b7 |
|--------------------|-------|----|----|----|----|----|----|
| C dórico | C | D | Eb | F | G | A | Bb |
| Modo lidio | 1 (T) | 2 | 3 | #4 | 5 | 6 | 7 |
| A lidio | A | B | C# | D# | E | F# | G# |
| Modo locrio | 1 (T) | b2 | b3 | 4 | b5 | b6 | b7 |
| Eb locrio | Eb | Fb | Gb | Ab | Bb | Cb | Db |

Tabla 2.13: Ejemplos de modos con distintas tónicas

A pesar de no ser tan utilizados como las escalas mayor y menor, los modos y sobre todo, los modos griegos, son utilizados también en la música que solemos escuchar hoy por hoy. Difieren de sus relativas mayor y menor en el centro tonal. Como la tónica es distinta, los movimientos que surgen en una composición son distintos también, lográndose así colores nuevos. Aunque los modos formen parte de un capítulo más avanzado dentro de la teoría de la armonía, el conocimiento de su existencia y su origen son cruciales para el entendimiento de varios apartados del TFG.

2.4. MIDI

A la hora de representar música, tradicionalmente hacemos uso una serie de estándares y normas que juntos forman la notación musical occidental. El uso de esta notación musical nos permite escribir partituras que otros músicos pueden entender e interpretar.

Pero, ¿cómo llevamos esto al mundo digital? Si bien hay programas capaces de interpretar partituras, en la mayoría de ocasiones se utilizan otras formas de lo que se denomina música simbólica. La más común es el MIDI (archivos con extensión .mid).

El MIDI, al igual que el resto de formatos de música simbólica, no suena por sí solo, pues el archivo no contiene sonido alguno. Al igual que una partitura, el MIDI nos indica qué nota hay que tocar, cuando hay que tocar esa nota, durante cuánto tiempo, y en ocasiones incluso en qué instrumento recomiendan ser tocadas esas notas (aunque esto último es algo que apenas se utiliza ya).

Como ya hemos dicho, el MIDI no contiene sonido, sólo instrucciones para tocar una serie de notas, por lo que es un formato de archivo muy liviano y cómodo de usar. Además, si bien podría parecer una desventaja no poder escuchar sin más su contenido, los sistemas operativos suelen traer una forma de reproducir el archivo MIDI fácilmente. En el caso de Windows, basta con abrirlo con el reproductor de multimedia por defecto para que lo interprete, eso sí, con timbres algo pobres pero que cumplen su función.

2.5. Software musical

2.5.1. ¿Qué es una DAW?

Una DAW (*Digital Audio Workstation*) o estación de trabajo de audio digital, es un programa capaz de grabar y editar audio, producir, mezclar y masterizar música. También permiten secuenciar música mediante MIDI y hacerla sonar mediante instrumentos virtuales.

Algunos ejemplos de DAWs populares son Ableton Live, Pro Tools, FL Studio o Reaper (Sección 2.5.2), siendo esta última la que usaremos en nuestra herramienta.

2.5.2. REAPER

Reaper es la DAW que usaremos para generar sonido usando nuestra aplicación. Hemos escogido esta DAW por la facilidad que nos brinda para lanzar scripts sobre ella y por su precio, ya que cuenta con un periodo de prueba gratuito de 60 días con todas las funcionalidades del programa completo (sigue permitiendo su uso acabado el periodo de prueba). Además, el precio de Reaper es muy asequible comparado con el de otras DAWs.

En la Figura 2.8 podemos ver la interfaz de Reaper. Hay tres pistas creadas, las dos primeras contienen algunos ítems MIDI y la tercera pista un archivo de audio que hemos cargado en formato .wav. Además, en la parte inferior de la interfaz se encuentra el mezclador o *mixer*, donde podremos, entre otras cosas, regular el volumen de cada pista así como el de la mezcla resultante. También podemos desde este mezclador silenciar una pista (pulsando el botón con la letra M) o hacer que suene en solitario, silenciando el resto (pulsando el botón con la letra S).

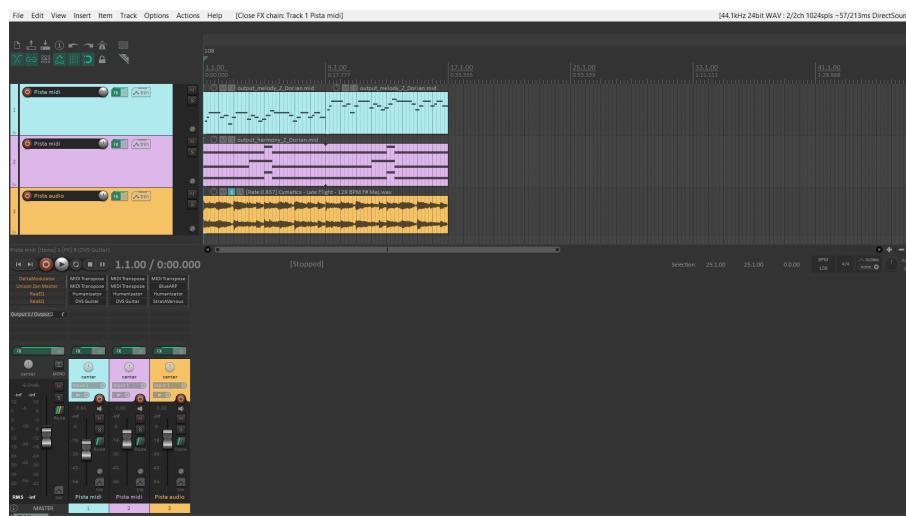


Figura 2.8: Captura de pantalla de Reaper

Para preparar Reaper para trabajar con nuestra herramienta, consultar la guía de uso de la aplicación (Sección 3).

2.5.2.1. ReaScript

Uno de los puntos fuertes de Reaper es su capacidad para ejecutar scripts, contando incluso con un editor de texto integrado donde podemos escribir scripts en Lua y EEL (un lenguaje de scripting propio de Reaper). Además de usando esos dos lenguajes, podemos llamar a las funciones de la API de ReaScript usando Python. Esta es la opción que hemos elegido por la comodidad que ofrece a la hora de programar.

Una vez escrito un script de Python (estando habilitado Python, ver su guía de uso en Sección 3.2.2) podremos usar las acciones de Reaper para lanzar nuestro script. De forma alternativa, podemos lanzar el script sobre Python desde fuera, como lo hacemos desde la ventana de la herramienta (Sección 3.3)

2.5.3. Instrumentos

Para poder generar audio usando una DAW, podemos reproducir sonidos muestrados o sintetizar sonidos en tiempo real. En una DAW, podemos escribir o cargar música simbólica, generalmente en formato MIDI (Sección 2.4), en una pista. A continuación, podemos añadirle a esa pista un plugin de instrumento virtual, el cuál convertirá el MIDI en sonido.

Veamos un ejemplo: en la Figura 2.8 se puede ver en la primera pista un ítem MIDI, abierto en el rodillo de piano en la Figura 2.9. Dicho ítem generará una señal MIDI que, en tiempo real, los plugins añadidos podrán modificar, y en el caso de los instrumentos virtuales (como el de la Figura 2.10), convertir en audio.

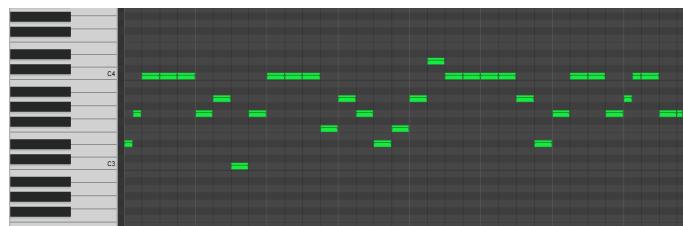


Figura 2.9: Notas MIDI en el rodillo de piano de Reaper.

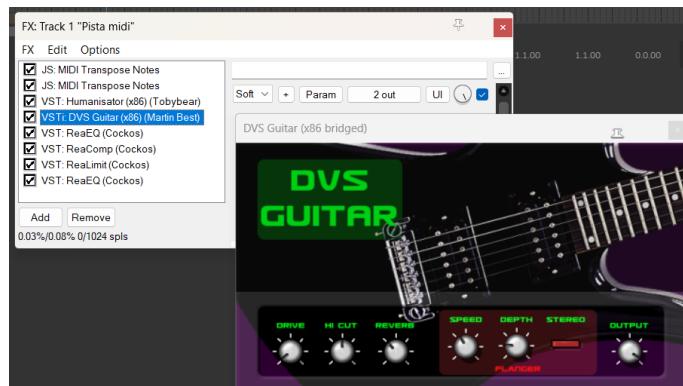


Figura 2.10: Un instrumento virtual de guitarra eléctrica, el DVS Guitar.

2.5.3.1. ¿Qué es un plugin?

Un plugin es un software externo (generalmente) que se carga en una DAW, en nuestro caso Reaper, para añadir distintas funcionalidades. En el ámbito de la producción musical, los plugins son fundamentales, ya que nos proporcionan instrumentos virtuales para generar sonidos y distintos efectos para manipular ese sonido.

El formato más común de los plugins en Windows son los VSTs (archivos con extensión .vst3 o .dll). Hay algunos otros formatos como AAX o AU, por ejemplo para dispositivos MAC, pero los que usaremos en nuestra herramienta son los mencionados VSTs.

Podemos separar los plugins en varios tipos: los efectos de audio, los plugins MIDI y los instrumentos virtuales.

Los efectos de audio reciben una señal de audio y la modifican de diversas formas. Algunos ejemplos de efectos de audio son la ecualización², la reverberación (ver la Sección 8.2.6) o la compresión³. En la Figura 2.11 podemos ver un ejemplo de ecualización: en la primera imagen se ve la señal de audio original que recibe el plugin, mientras que en la segunda se ve la curva de ecualización aplicada, así como su efecto en la señal de audio. En dicho ejemplo, se cortan las frecuencias agudas (las de la derecha del espectro de frecuencias) y además se suben un poco los graves, en torno a los 400Hz.

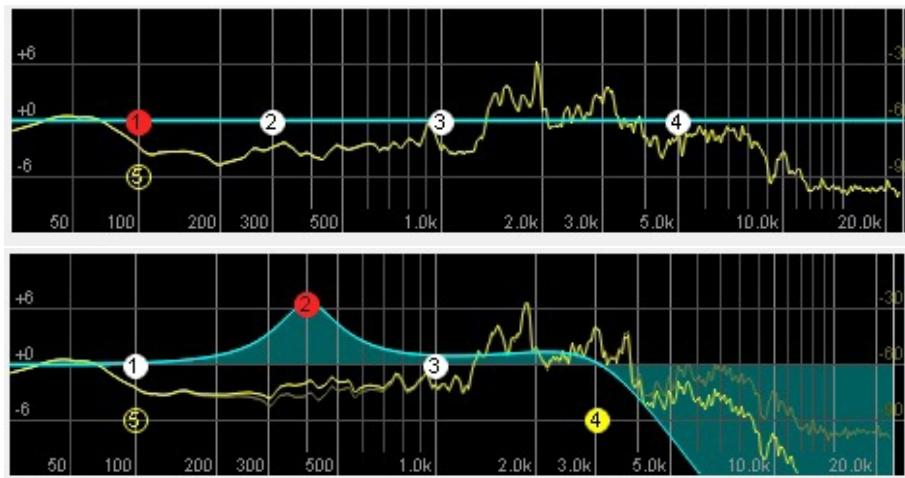


Figura 2.11: Plugin de ecualización por defecto de Reaper, ReaEQ

Los plugins MIDI modifican una señal MIDI y la convierten en otra. Por ejemplo cambiando la octava de las notas, su intensidad, arpegiando dichas notas, etc. Entre los plugins MIDI usados en el trabajo, se destaca el plugin arpegiador *BlueArp*. Usamos este plugin para secuenciar notas MIDI con diversos fines, como crear líneas de bajo, arpegiar acordes, en la percusión, etc. Para más info sobre *BlueArp*, consultar la Sección 8.2.3.

²<https://emastered.com/es/blog/equalizer-music>

³<https://www.aulart.com/es/blog/que-es-la-compresion-y-como-usarla/>

Los instrumentos virtuales son plugins que transforman la señal MIDI en audio. Los instrumentos virtuales pueden separarse en varias categorías:

- **Instrumentos *sampleados*:** Reproducen una muestra pregrabada al recibir un input MIDI. El número de muestras puede variar desde una muestra cada varias notas hasta decenas de muestras por nota, dependiendo de la intensidad de la nota (que en terminología MIDI es la *velocity*), o incluso varias muestras para una misma nota y misma intensidad (por ejemplo en un instrumento de percusión, golpeando en distintas zonas del tambor). Por tanto, para la mayoría de timbres ofrecen la mejor calidad de audio posible. Son instrumentos que suelen ocupar muchos gigabytes de almacenamiento, ya que las muestras en formatos de buena calidad (como el .wav) ocupan mucha memoria.
- **Sintetizadores:** Crean sonido a mezclando distintas formas de onda, envolventes y efectos⁴. Suelen usarse tradicionalmente en géneros de música electrónica, pero hoy en día hay muchos usos para este tipo de instrumentos, ya que pueden usarse para recrear de forma más o menos realista la mayoría de timbres que podamos necesitar. Al no usar sonidos pregrabados, son alternativas más ligeras que los instrumentos *sampleados*.
- **Instrumentos mixtos:** Combinan muestras de sonidos o instrumentos reales con sonidos sintetizados en tiempo real.

⁴<https://es.wikipedia.org/wiki/Sintetizador>

Capítulo 3

Guía de Uso

Este capítulo recoge los pasos que se deben seguir para poder utilizar la aplicación Vanguard Music al completo. El código fuente de la aplicación se encuentra en el repositorio de GitHub del siguiente enlace: https://github.com/miggon23/TFG_AsistenteComposicionConIA. En el README hay adjuntados enlaces a vídeos que muestran lo descrito en este capítulo de forma visual.

La Sección 3.2.4 incluye cómo realizar parte de la configuración de dependencias automáticamente.

3.1. Dependencias de la aplicación

3.1.1. Dependencias de Instalación

La aplicación funciona en los sistemas operativos Windows 10 y Windows 11, y está preparada para ser ejecutada con Python 3.9.

Para hacer uso de la aplicación, se requiere instalar herramientas de terceros. Estas dependencias se pueden diferenciar en dos grupos:

- **Dependencias de código:** Son las dependencias de la aplicación Vanguard Music, necesarias para que esta execute y funcione correctamente. Se compone de Python3.9, Reaper 7 y Node.js, además de las dependencias de bibliotecas de Python que recogemos en el fichero *requirements.txt*.
 - **Python 3.9:** Utilizamos Python3.9 como lenguaje de programación dentro de la aplicación, y como lenguaje de scripting para la comunicación con Reaper a través de ReaScript (Sección 8.3)
 - **Reaper 7:** Es la DAW que utilizamos para secuenciar y sonorizar la música generada por Vanguard Music
 - **Node.js:** Utilizamos Node.js para hacer uso de la API de Magenta en JavaScript. Esta dependencia se puede omitir si no se desea generar con

Magenta, pudiendo hacer uso del resto de generadores que funcionan con Python. Para ello, hay que realizar un *npm install* dentro de la *carpeta MagentaGenerator*. Esta es una de las acciones automatizadas (ver la Sección 3.2.4).

- **Visual Studio C++ Build Tools:** Contiene paquetes de C++ necesarios para el funcionamiento de las bibliotecas que usan los modelos generativos de nuestra aplicación. La Sección
- **Plugins de Reaper:** Contiene los archivos que hay que proporcionar a Reaper para poder sonorizar correctamente las piezas musicales generadas por Vanguard Music. En la Sección 3.1.3 se explica cómo configurar estos archivos, los cuales se dividen en las siguientes carpetas:
 - La *carpeta VSTs*, que utilizamos para Reaper. Contiene los instrumentos virtuales.
 - La *carpeta presets* contiene los archivos de configuración de los VSTs.

La lista de todas las dependencias con sus respectivas versiones recomendadas se puede consultar en el Apéndice A. Incluye también los enlaces de descarga

3.1.2. Interfaz gráfica con *TKInter*

Hemos usado la biblioteca de TKInter para montar la interfaz de usuario de la aplicación en Python. TKInter es la biblioteca por defecto que Python ofrece a los desarrolladores para montar una GUI (Interfaz Gráfica de Usuario por sus siglas en inglés)

3.1.3. Reaper

Reaper es la DAW de producción musical que usamos en nuestra aplicación. Este entorno de edición y producción musical es el que nos permite *renderizar* el audio que generamos en nuestra aplicación. Es decir, nos permite agregar los instrumentos y efectos de audio que hacen sonar el MIDI generado o proporcionado por el usuario.

Para el uso de la herramienta debemos instalar Reaper 7, desde su web oficial. El usuario debe tener en cuenta que Reaper no es una DAW gratuita, pero permite un periodo de prueba con todas sus funcionalidades y donde podrá usar la herramienta sin ninguna limitación, tan sólo hay que esperar unos segundos y cerrar la pestaña que aparece explicando que estamos usando el periodo de prueba.

Junto al código fuente de la aplicación, se adjuntan dos carpetas, la *carpeta VSTs*, que contiene los plugins que utilizamos en Reaper; y la *carpeta presets*, que contiene las distintas configuraciones de los plugins. La Sección 2.5.3.1 contiene más información acerca de qué es un plugin.

Tras descargar los plugins y los presets, hay que proporcionarle a Reaper sendas rutas a los respectivos directorios. A continuación se menciona como realizar esta configuración en Reaper:

- En *Options > Preferences > Plug-ins > VSTs > VST plug-in paths* hay que añadir la ruta a la carpeta *VSTs* que proporcionamos.
- *Options > Show Reaper resources path* abrirá la ruta de recursos de Reaper, hay que sustituir la carpeta *presets* por la carpeta proporcionada por nosotros. Si no se quieren perder los presets de Reaper se puede añadir el contenido de la carpeta en lugar de sustituirlo, pero pueden producirse incompatibilidades entre presets.

3.2. Preparación del entorno

Una vez instaladas las dependencias, se requiere una preparación del entorno de trabajo para poder ejecutar la aplicación.

3.2.1. Entorno virtual

Un entorno virtual de Python es un espacio que permite ejecutar proyectos de Python de forma independiente, aislando el conjunto de dependencias y configuraciones del resto del sistema. Esto permite trabajar de forma que no haya conflictos entre paquetes externos al entorno. Además, las dependencias que se instalen dentro del entorno activado no saldrán de este, evitando dejar residuos en el sistema al borrar el entorno.

Para poder ejecutar la aplicación en un entorno de Python, asegúrese de que la ejecución de scripts está activada en su plataforma Windows.

La ejecución de scripts se puede activar desde un PowerShell de Windows con: *Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass*. Se puede desactivar con: *Set-ExecutionPolicy -Scope Process -ExecutionPolicy Default*. El modificador *-Scope* indica que la política de ejecución se aplicará solo al proceso actual. Si se quiere cambiar a nivel de usuario para que sobreviva al proceso, se puede cambiar *-Scope* por *CurrentUser*.

Desde un CMD, en la raíz del repositorio, ejecute *python -m venv env* para crear un entorno de Python, después *./env/Scripts/activate* para activar el entorno virtual. Crear el entorno solo es necesario hacerlo una vez, lo que valdrá para futuras ocasiones, pero será necesario activarlo cada vez que se abra un CMD, ya que por defecto está desactivado.

Para instalar las dependencias de la aplicación de forma automática, se recomienda ejecutar el comando *pip intall -r requirements.txt* la primera vez que se active el entorno.

3.2.2. Ejecución de scripts con Reaper

Como se menciona en la Sección 8.3, por defecto Reaper solo permite la ejecución de scripts de Lua y de EEL (*Embedded Extensible Language*).

Para que nuestra aplicación se pueda comunicar con Reaper, es necesario tener activada la opción *permitir ejecución de scripts de Python*. Dentro de Reaper, debemos seleccionar el apartado *Opciones > Preferencias* (o atajo de teclado *Ctrl+P*) y buscar el apartado *ReaScript* dentro del apartado *Plug-ins*.

Una vez en esta pestaña, marcaremos la casilla *Enable Python for use with ReaScript* y le proporcionaremos la ruta a nuestro ejecutable de Python 3.9 (Apéndice A), así como el nombre de la DLL de Python (*python39.dll* en el caso de usar la versión que recomendamos). Pulsamos *apply* y debería indicarnos que Python está instalado correctamente, reiniciamos Reaper y estamos listos para usar la herramienta.

3.2.3. Iniciar la Aplicación

Para iniciar la aplicación una vez instaladas las dependencias y preparado el entorno, podremos usar la aplicación Vanguard Music con normalidad. Para ello, con el entorno virtual activado, y situado sobre la raíz del repositorio, hay que ejecutar el script *appRoot.py*. Desde un CMD se puede hacer en Windows con *py appRoot.py*. Recuerde tener el entorno virtual activado a la hora de lanzar la aplicación (ver Sección 3.2.1 para ver como activar el entorno virtual).

3.2.4. Instalación automatizada

La instalación de las dependencias incluidas en el *requirements.txt*, la creación del entorno virtual de Python y el inicio de la aplicación están automatizadas mediante la ejecución de archivos por lotes, evitando tener que realizar de forma manual varios de los pasos anteriormente mencionados. Un archivo por lotes se puede ejecutar haciendo doble click sobre él. Los principales archivos por lotes utilizados para este propósito son los siguientes:

- **install_magenta_dependencies.bat**: Instala las dependencias de JavaScript sobre la carpeta *MagentaGenerator*, ejecutando un *npm install*. Requiere tener instalado Node.js para que funcione.
- **install_requirements.bat**: Este archivo se encarga de crear el entorno virtual de Python si no está creado. Para qué esto funcione, la política de ejecución de scripts debe estar en modo *Bypass* (ver la Sección 3.2.1 para ver cómo cambiar la política de ejecución de scripts en Windows). Tras crear el entorno virtual, instala las dependencias contenidas en *requirements.txt*. Además llama a *install_magenta_dependencies.bat*.

La instalación completa puede tardar unos minutos la primera vez.

- **VanguardMusic.bat**: Este script activa el entorno virtual e inicia la aplicación. Si no existe entorno virtual, llama a *install_requirements.bat* antes de iniciar la aplicación.

La instalación de dependencias automática no cubre la configuración de los VSTs (plugins) y presets de Reaper. Al final de la Sección 3.1.1 se explica como configurar estas carpetas que se adjuntan con el código fuente. Del mismo modo, hay que instalar Node.js, Reaper 7 y las *Visual Studio C++ Build Tools* por separado.

3.3. Generación de audio

Esta es la primera pestaña de Vanguard Music, la que aparecerá nada más abrir la aplicación (Figura 3.1).

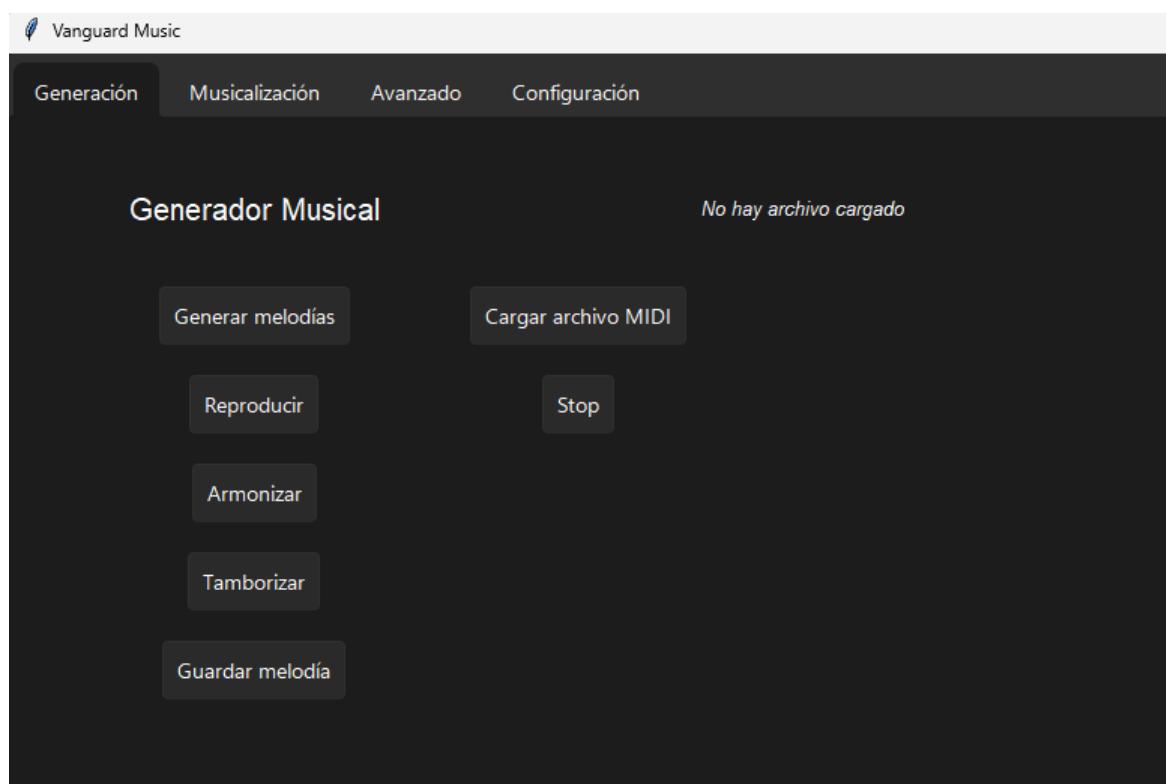


Figura 3.1: Pestaña *Generación*

Esta pestaña reúne las funcionalidades que permiten al usuario generar MIDI. Estas funcionalidades se dividen:

- **Generar:** Genera MIDI de acuerdo con el generador activo. Para cambiar el generador de midi, ver la Sección 3.6.1.
- **Cargar archivo MIDI:** Carga un archivo .mid desde el sistema de ficheros del usuario.
- **Reproducir:** Reproduce la melodía generada a modo prueba con un sintetizador simple.

- **Armonizar:** Armoniza la melodía generada. Ver la Sección 2.2 para más detalle.
- **Tamborizar:** Genera el MIDI asociado a la percusión. La sección 7.1.2 contiene más información acerca de cómo se genera la percusión.
- **Guardar melodía:** Guarda el fichero .mid generado en el sistema de ficheros del usuario.

3.4. Selección de Temática y Sonorización

Esta sección describe la pestaña Musicalización de la aplicación: (Figura 3.2).

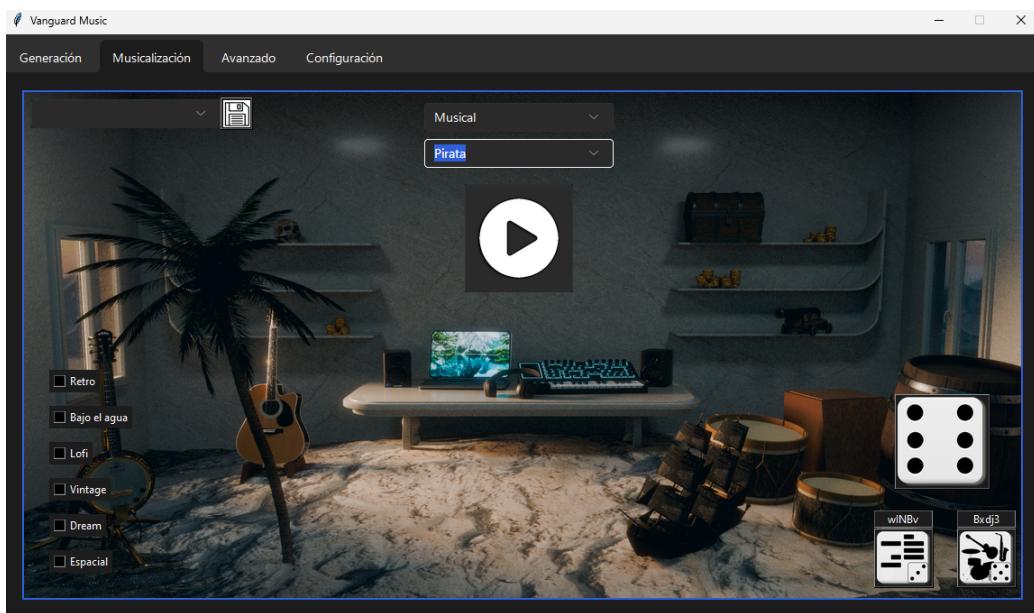


Figura 3.2: Pestaña Musicalización

En esta pestaña disponemos de varias opciones para dirigir la sonorización del MIDI generado en la pestaña Generación de audio.

3.4.1. Temáticas

Las temáticas afectan a la instrumentación del arreglo musical. En la parte superior de la Figura 3.2, encontramos un desplegable con las posibles temáticas que se pueden seleccionar. El estilo del fondo irá cambiando según la temática seleccionada, reflejando la configuración activa. Las temáticas disponibles son las siguientes:

- **Pradera:** Sonidos de guitarra, piano y cuerdas. Genera la sensación de estar al aire libre en una pradera o un bosque. En ocasiones puede llegar a sonar a aventura.

- **Piano:** Un único piano interpretará todo el arreglo.
- **Desierto:** Sitars, flautas, guitarras y percusiones que dan la sensación de estar en un desierto.
- **Nieve:** Sonidos brillantes, ukeleles y kalimbas que nos sitúan en un paisaje nevado.
- **Pirata:** Guitarras, percusiones, acordeones y violines que bien podría llevar un pirata en su barco.
- **Selva:** Marimbas y una gran variedad de sonidos de percusión sonando simultáneamente interpretando los ritmos de la selva.
- **Épico:** Una orquesta entera interpretando el arreglo.
- **Tenebroso:** Sonidos lúgubres con timbres disonantes que generan tensión constante.
- **Agua:** Sonidos suaves y burbujeantes generan la sensación de estar flotando.
- **Asiático:** Instrumentos tradicionales asiáticos mezclados con sonidos modernos.
- **Rock:** Guitarras eléctricas con mucha distorsión.
- **Pop:** Sonidos modernos presentes en la música pop actual.
- **Tecno:** Percusión de música disco y varios sintetizadores sonando a la vez.

3.4.2. Entorno

Justo encima del desplegable de temáticas encontramos el desplegable de entornos. Este desplegable nos dará la opción de elegir dónde se simulará que suena nuestra canción, es decir, cambiará la reverberación que se aplicará a la mezcla.

3.4.3. Efectos

En la parte inferior izquierda se encuentran varias *checkboxes* asociadas a efectos que se pueden aplicar a la canción. Los cambios en estos modificadores también se ven reflejados en el estilo del fondo de esta pestaña de la aplicación.

Los efectos disponibles son los siguientes:

- **Retro:** Disminuye la tasa de bits y la frecuencia de muestreo, reduciendo la calidad del sonido. Esto hace que la mezcla suene como lo haría en una consola antigua.
- **Bajo el agua:** Ralentiza todos los sonidos para generar la sensación de estar buceando.

- **Lo-fi:** Agrega un filtro *lo-fi* de entre varios posibles. Esto conlleva la disminución de la calidad del sonido, el filtrado de frecuencias agudas, la adición de ruido de fondo (como el ruido de una cinta antigua) y algunas imperfecciones en el tono del sonido.
- **Vintage:** Añade saturación analógica y pedales de reverberación produciendo la calidez característica de una grabación en cinta antigua.
- **Dream:** Ralentiza los sonidos y los desafina ligeramente para generar una sensación psicodélica.
- **Espacial:** Coloca todos los sonidos en el espacio. Ideal para temas espaciales o de música *ambient*.

3.4.4. Semillas

En la parte inferior derecha de la Figura 3.2 se encuentra un botón con forma de dado, mostrando un seis. Este dado aleatoriza las semillas de instrumentos y de arreglos. Estas semillas serán las que se utilicen para sonorizar el MIDI una vez se pulse el botón de reproducir que hay en medio de la pantalla. Una misma semilla generará los mismos instrumentos y pistas siempre que la configuración de Temáticas sea la misma. Es decir, para dos casos de uso donde la semilla sea la misma pero la temática o los modificadores (*checkboxes*) sean distintos, la generación final de instrumentos y pistas será distinta.

La semilla de instrumentos es la que está encima del botón de dado que muestra un 5, junto a varios instrumentos (Figura 3.2). Esta semilla cambia la asignación de instrumentos sin modificar el MIDI de las pistas. Esta semilla se puede cambiar, por ejemplo, si se desea modificar los instrumentos sin afectar al arreglo o distribución de los ítems MIDI. El Capítulo 8 contiene más información acerca de la selección automática de los instrumentos.

La semilla de arreglos es la que se sitúa encima del botón de dado que muestra un 3, junto a unas barras horizontales negras (Figura 3.2). Esta semilla cambia la disposición del MIDI generado en las distintas pistas de Reaper. Esta semilla se puede cambiar, por ejemplo, si se desea modificar el arreglo sin afectar a la asignación de instrumentos. La Sección 9.4 y las siguientes de ese capítulo, explican cómo se crea el arreglo.

Cada semilla se puede cambiar mediante su botón asociado. También se puede introducir una semilla a mano directamente escribiendo sobre la entrada de texto de la semilla.

3.4.5. Presets

Los presets son configuraciones de la pestaña Musicalización que se guardan dentro de la aplicación. Estas configuraciones albergan el estado de la temática, las

checkboxes y la semilla seleccionadas al momento de haberlas guardado. La configuración actual de la pestaña Musicalización se puede guardar haciendo clic en el botón que tiene el icono de la Figura 3.3.



Figura 3.3: Icono de guardado

Al pulsar este botón, se abrirá una ventana solicitando un nombre para el preset (Figura 3.3). Para crear un nuevo preset, basta con escribir un nombre de preset que no esté registrado ya. Si se introduce un nombre de preset ya registrado, este se sobrescribirá con la información del nuevo estado de la pestaña Musicalización.

Para recuperar un preset, basta con utilizar el desplegable situado en la parte superior izquierda de la pestaña y seleccionar el nombre del preset. Esto actualizará la información de la pestaña con la que había en el momento en el que se guardó ese preset, es decir: temática, *checkboxes* y semilla.

3.5. *Renderizado*: Descarga de los resultados

Al pulsar el de *Play* se abrirá Reaper automáticamente, disponiendo las pistas con el MIDI y los instrumentos generados en los pasos previos de la aplicación.

A partir de este punto el usuario tiene total libertad de trabajar con Reaper de forma autónoma, con la propia interfaz de Reaper. Para *renderizar* y guardar el resultado, en *File > Render* se puede seleccionar el formato de compresión, el nombre de archivo y la ruta de guardado, así como otras características propias de Reaper.

3.6. Modo avanzado

El objetivo principal de la aplicación es ofrecer a los usuarios una forma sencilla de generar temas musicales. No obstante, está incluida la posibilidad de realizar acciones avanzadas para los usuarios más avezados de la aplicación, así como los que cuentan con conocimiento musical previo.

La posibilidad de editar en Reaper los temas musicales generados por nuestra aplicación sigue existiendo, tal y como está planteado en la sección anterior, al pulsar el botón de *Play*. Esto permite al usuario poder trabajar directamente en Reaper en lugar de centrarse en nuestra aplicación, a partir de los MIDIs e instrumentos generados por Vanguard Music.

En los apartados siguientes veremos las acciones posibles dentro de la pestaña Avanzado de la aplicación: (Figura 3.4)

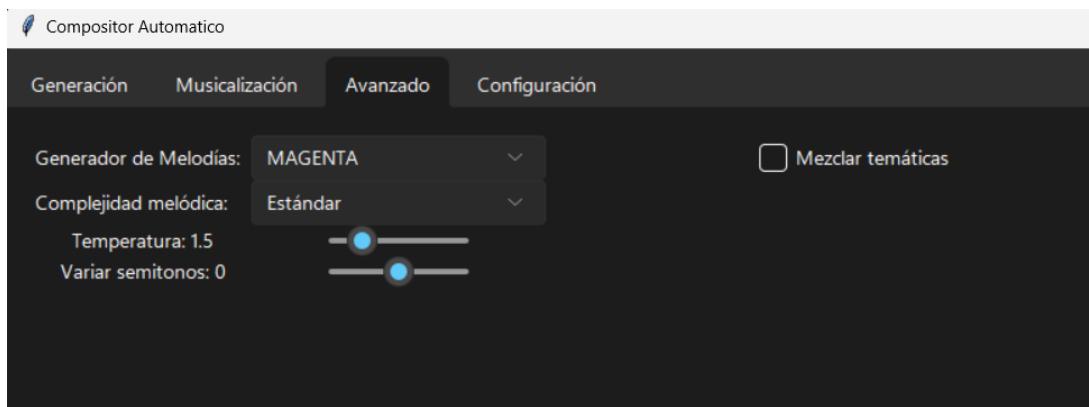


Figura 3.4: Pestaña avanzada

3.6.1. Alternar Generador

Esta opción permite cambiar el modo de generación de la melodía en la pestaña Generación. Los generadores disponibles son *Cadenas de Markov*, *Redes Neuronales Recurrentes* y *Magenta*. Ver la Sección 4 para más información acerca de los modos de generación musical.

3.6.2. Temperatura

La temperatura influye en cómo el generador produce el MIDI para las melodías. Un valor bajo hace que el generador tome decisiones menos arriesgadas, con melodías más simples, mientras que valores altos dotan de un carácter experimental al generador. Por defecto, como muestra la Figura 3.4, el valor de temperatura se encuentra a 1'5.

3.6.3. Mezclar temáticas

La opción mezclar temáticas permite asignar a cada pista de Reaper una temática distinta, en lugar de aplicar la misma temática a todas.

En la Figura 3.5 se puede observar qué pista se asociará con qué temática. Por defecto, cuando la opción temáticas aleatorias está desactivada, se aplicarán a todas las pistas la temática seleccionada en la pestaña Musicalización. De este modo, el usuario puede ir modificando la temática de cada pista a partir de una temática raíz.



Figura 3.5: Mezclar temáticas

Las temáticas añadidas al desplegable se incluirán en el preset al momento de guardarlo, pudiendo ser recuperado, como se indica en la Sección 3.4.5. No obstante, solo se aplicará en la musicalización en Reaper si la opción Mezclar temáticas está activada.

La opción *Temáticas aleatorias* asigna una temática aleatoria a cada pista, como se refleja en la Figura 3.6.

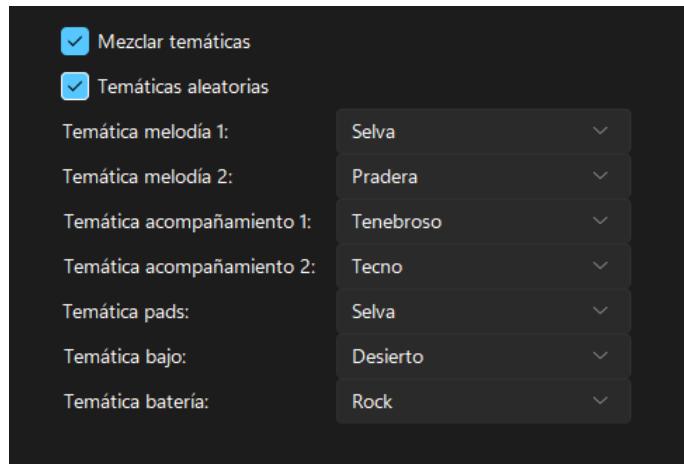


Figura 3.6: Aleatorización de temáticas

3.6.4. Variar semitonos

Varía la tonalidad de la canción el número de semitonos indicado en la Figura 3.4. Esta variación se aplica a la señal MIDI que reciben los instrumentos virtuales que generan el sonido en Reaper. No modifica en modo alguno el sonido una vez generado, por tanto no se pierde calidad de la mezcla final por usar esta funcionalidad (como sucedería si se varía la tonalidad en el sonido ya generado).

3.7. Configuración

La pestaña de configuración permite reorganizar los recursos de la aplicación.

- Ruta de Reaper: La ruta absoluta a Reaper es utilizada por la aplicación para poder encontrar el ejecutable de Reaper a la hora de reproducir el contenido generado por Vanguard Music. Por defecto, Reaper se busca en *C:/Program Files/Reaper (x64)/Reaper.exe*. Si su ruta absoluta a Reaper es distinta a esta, es importante que actualice esta opción para poder usar la aplicación correctamente.

3.8. Ayuda y Tooltips

Las *Tooltips* son descripciones de texto que muestran el efecto o función de un elemento. Esto incluye botones, entradas de texto, desplegables, etc. Para ver la *Tooltip* de una herramienta, sitúe el cursor encima de la herramienta.

Capítulo 4

Generación de Melodías con *Machine Learning*

En este capítulo se explicará cómo se han generado melodías. Se van a explorar distintas formas de generación de melodías, entre ellas 2 modelos propios entrenados y un modelo ya existente de Magenta.

4.1. Cómo representamos la música

Para poder trabajar con las melodías, necesitamos alguna forma de representación interna para tratar las notas en los diferentes modelos. A continuación se detallan las representaciones utilizadas en este módulo.

4.1.1. Representación "*pitch_duration*"

Esta representación simplifica al máximo la información crucial de una nota, es utilizada tanto por las cadenas de Markov (Sección 4.3), como por las redes neuronales recurrentes (Sección 4.4).

La duración suele venir dada en *steps*, ya que nos proporcionan una unidad entera que es independiente del tiempo absoluto.

Las notas se supone que se encuentran seguidas una de otra y no pueden sonar varias a la vez, por lo que no son necesarios atributos como el tiempo de inicio o fin de cada nota.

El silencio viene codificado como una nota de pitch 0.

Por ejemplo, si quisiéramos codificar un Do de la cuarta octava (C4 en inglés) que dura 2 tiempos se codificaría como "60_2", siendo 60 el pitch MIDI de la nota C4 (consultar Wolfe (s.f.) para más información sobre el pitch MIDI).

4.1.2. Magenta *NoteSequence*

Para poder guardar archivos MIDI de forma sencilla y comunicarnos con modelos de Magenta, empleamos el estándar definido por este llamado *NoteSequence* (Magenta, 2022c).

Este estándar nos proporciona una forma cómoda y rápida de interactuar con la API de Magenta (Sección 4.5), así como de poder leer y convertir a MIDI fácilmente.

Existe un paquete de pip para utilizar *NoteSequence* en Python llamado *note-seq*.

En el módulo propio *noteseqConverter* se definen una serie de funciones para poder convertir de *NoteSequence* al formato simplificado "*pitch_duration*" y a JSON, además de funciones para guardar y cargar archivos MIDI.

4.2. Dataset utilizado y procesamiento de datos

Para poder utilizar algoritmos de *machine learning* lo primero es tener un dataset con una gran cantidad de datos y procesarlo adecuadamente.

A la hora de buscar el dataset es importante buscar uno adecuado para poder generar melodías básicas que podamos utilizar en otras partes de la aplicación.

Magenta posee una gran cantidad de datasets que se utilizaron para el entrenamiento de sus modelos. Dichos datasets se pueden consultar en Magenta (2022a).

El dataset elegido ha sido el *Bach Doodle Dataset* de Magenta¹.

Este dataset consiste en una serie archivos JSON con melodías que componían los usuarios del Bach Doodle (Huang et al. (2019)). Contiene más de 6 años de música de usuarios y sus melodías están en formato *NoteSequence* (Sección 4.1.2).

Al utilizar un dataset con melodías sencillas podemos entrenar de forma más simple y directa nuestros modelos. Pero para poder utilizar el dataset tenemos que procesar algunos aspectos de este:

- Lo primero es filtrar las melodías. Existe un campo en cada entrada del dataset llamado "*feedback*", que representa el feedback de usuarios con 0 (malo), 1 (neutral) o 2 (positivo). Ya que tenemos una gran cantidad de melodías, podemos descartar todas las melodías que no tengan un feedback de 2.
- Para poder realizar ajustes posteriormente a la melodía es conveniente que se encuentre en una escala determinada. El dataset incluye un campo llamado "*key_sig*", el cual indica la nota fundamental de la escala. Por simplicidad descartamos todas las melodías que no se encuentren en Do Mayor (C Major en inglés). Además, limitamos las notas a un rango de 2 octavas realizando trasposiciones, consistiendo estas en, sencillamente, mover las notas a otras octavas.

¹<https://magenta.tensorflow.org/datasets/bach-doodle>

- Finalmente, es vital normalizar el tempo de las canciones, por lo que convertimos todas las melodías a 120bpm. Para esto basta con convertir adecuadamente los *steps* de las melodías del dataset, ya que existe un campo llamado *tempos* en la propia *NoteSequence* de la melodía.

Una vez realizados estos ajustes, podemos guardar el dataset procesado como CSV con los atributos

$$\text{pitch, start, end, duration, next_note_pitch, next_note_start, next_note_duration.}$$

Con estos parámetros podemos implementar distintos modelos de *machine learning* utilizando unos parámetros u otros dependiendo de las necesidades del modelo.

4.3. Cadenas de Markov

Las cadenas de Markov son un modelo estocástico que describe una secuencia de eventos en la que la probabilidad de cada evento es dependiente únicamente del estado anterior.

Por lo tanto, el modelo de Markov no tiene memoria (sin entrar a modelos más complejos).

Simplificando, podemos pensar en las cadenas de Markov como una máquina de estados en la que cada estado está conectado a todos los demás y las probabilidades de pasar a cada estado desde uno cualquiera suman 1.

Las cadenas de Markov son comúnmente representadas gráficamente mediante un grafo dirigido tal y como se muestra en la Figura 4.1.

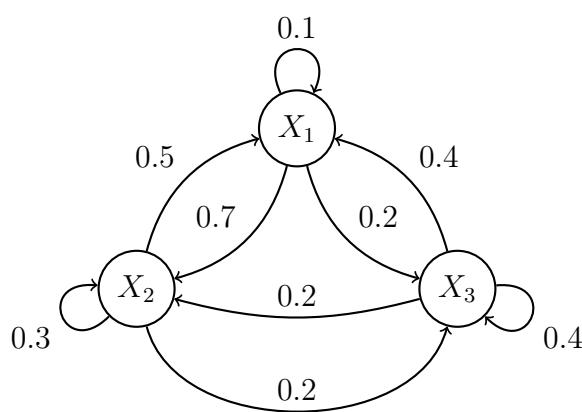


Figura 4.1: Ejemplo de una cadena de Markov

Internamente, las cadenas de Markov se suelen representar con matrices de transición, tales como la de la Tabla 4.1

| | X_1 | X_2 | X_3 |
|-------|-------|-------|-------|
| X_1 | 0.1 | 0.7 | 0.2 |
| X_2 | 0.5 | 0.3 | 0.2 |
| X_3 | 0.4 | 0.2 | 0.4 |

Tabla 4.1: Ejemplo de una matriz de transición

4.3.1. Entrenamiento de las Cadenas de Markov

Una ventaja de las cadenas de Markov es su fácil entrenamiento. Una vez tenemos nuestro dataset limpio y normalizado (Sección 4.2) podemos recorrerlo para construir la matriz de transición.

En nuestro caso, cargamos todas las secuencias de notas del dataset, descartamos las notas que no tienen otra a continuación (serían las que se encuentran al final de la melodía) y rellenamos una tabla de ocurrencias. Dicha tabla indica el número de veces que una nota aparece después de otra en las melodías.

Por ejemplo, si hubiera sólo 4 notas (cabe destacar que las notas se encuentran en notación *pitch_duration*, dicha notación se explica en la Sección 4.1.1) podría quedar la matriz de ocurrencias dada en la Tabla 4.2 tras recorrer todo el dataset.

| | 60_2 | 64_1 | 65_2 | 67_1 |
|------|------|------|------|------|
| 60_2 | 238 | 119 | 280 | 63 |
| 64_1 | 120 | 50 | 185 | 145 |
| 65_2 | 117 | 108 | 15 | 60 |
| 67_1 | 120 | 20 | 36 | 24 |

Tabla 4.2: Ejemplo de matriz de ocurrencia

Posteriormente sumamos cada fila y convertimos a probabilidades cada entrada de la tabla dividiendo entre la suma de su fila. Con esto obtenemos una matriz de transición como la de la Tabla 4.3.

| | 60_2 | 64_1 | 65_2 | 67_1 |
|------|------|------|------|------|
| 60_2 | 0.34 | 0.17 | 0.4 | 0.09 |
| 64_1 | 0.24 | 0.1 | 0.37 | 0.29 |
| 65_2 | 0.39 | 0.36 | 0.05 | 0.2 |
| 67_1 | 0.60 | 0.1 | 0.18 | 0.12 |

Tabla 4.3: Ejemplo de matriz de transición calculada a partir de la matriz de ocurrencia

Con la matriz de transición ya podríamos ejecutar la cadena de Markov N veces para obtener una melodía. En nuestro caso utilizamos la librería de Python PYDTMC, que proporciona modelos de Markov ya implementados (para más información sobre dicha librería consultar Beluzzo (2024)). Con esta librería podemos:

- Crear una cadena de Markov a partir de la matriz de transición.

- Guardar las cadenas a archivo.
- Ejecutar la cadena. Ya sea N iteraciones o ejecutar sólo una iteración.
- Dibujarla con *matplotlib*.

La representación gráfica de la cadena se puede ver en la Figura 4.2.

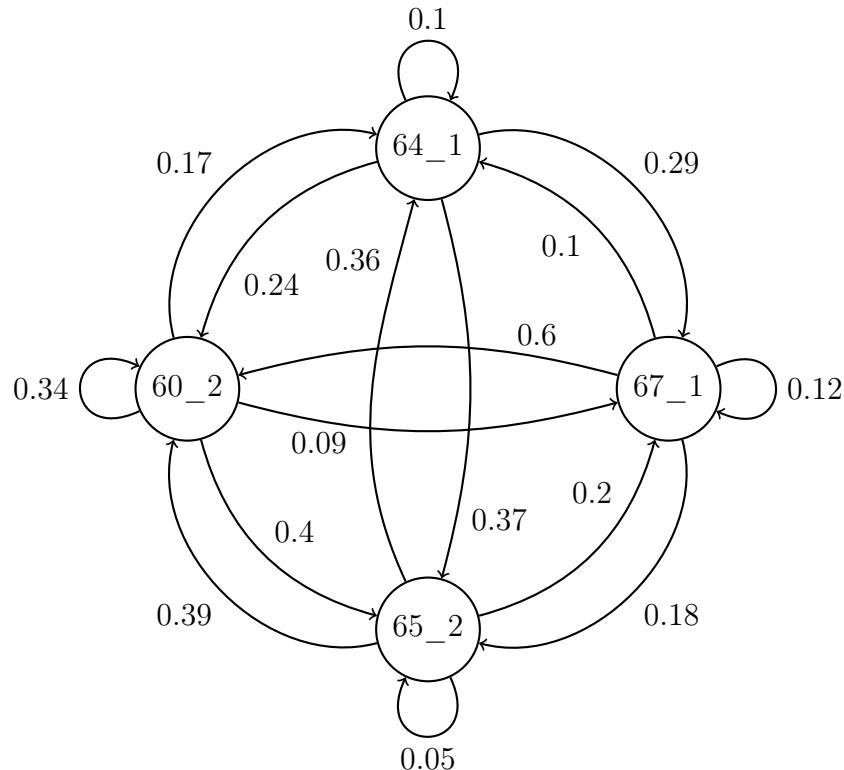


Figura 4.2: Representación visual de la cadena obtenida a partir de la matriz de transición

4.3.2. Generar melodías con Cadenas de Markov

Para crear melodías, podemos realizar el número de iteraciones que queramos sobre la cadena para crear una melodía de la longitud deseada, el acercamiento más simple sería generar N notas.

En nuestro generador (definido en el archivo *MarkovGenerator.py*) se puede especificar el número de *steps* deseado y se generarán iteraciones suficientes hasta llegar al límite. Al ejecutar comenzamos siempre en "C4_2", por conveniencia.

4.3.3. Puntos fuertes y débiles de la generación de melodías con Cadenas de Markov

Las cadenas de Markov resultan muy potentes como primer acercamiento, pues son un modelo simple y fácil de entender. El entrenamiento es sencillo y su ejecución

una vez entrenada es prácticamente instantánea.

Además, una parte importante es que al existir aleatoriedad es un modelo que no es determinista, por lo que cada ejecución será distinta.

Sin embargo, tienen algunas desventajas:

- Primero, hablando de rendimiento y escalabilidad, cada nodo de la cadena tiene que representar una nota con su duración, por lo que en la práctica se crea una cadena inmensa aunque limitemos a 2 octavas el posible rango melódico. Esto hace que la cadena ocupe bastante en archivo, al menos con la librería que utilizamos.
- Desde el punto de vista musical, no proporcionan una melodía muy rica, ya que dependen completamente de la probabilística, las melodías generadas no tendrán una coherencia aparente. Aunque ese punto no resulta muy crítico para nuestro trabajo por el resto de etapas que realizamos, es conveniente obtener una melodía lo más agradable posible.

A pesar de las desventajas que presentan, resultan muy interesantes como modelo más experimental y liviano en la ejecución. Por estos motivos se mantienen como opción de generación en la aplicación final.

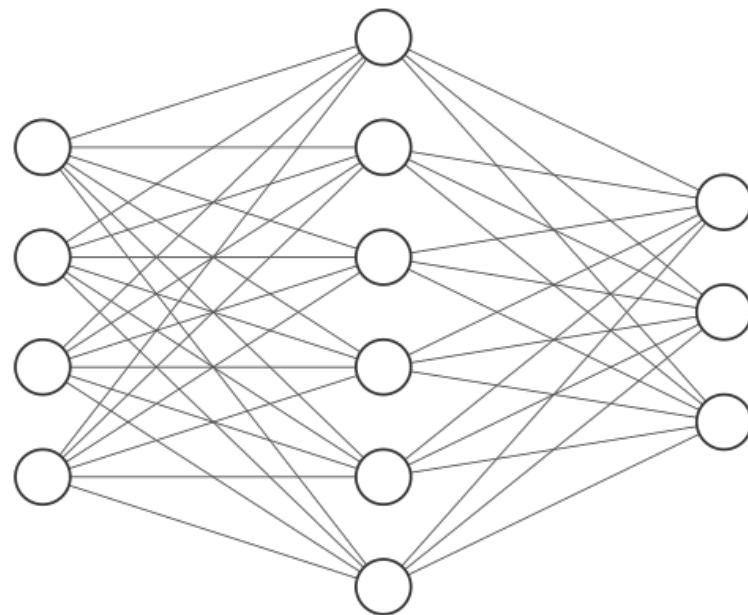
4.4. Redes Neuronales Recurrentes

4.4.1. Introducción a las Redes Neuronales Recurrentes

4.4.1.1. ¿Qué es una red neuronal?

Las redes neuronales son uno de los principales modelos de *machine learning* y de los más estudiados por su versatilidad y capacidad de resolver problemas no lineales. A lo largo de los años se han explorado muchos tipos de redes neuronales, pero antes de profundizar debemos de entender qué es una red neuronal. En esta sección se explicarán las redes neuronales de una forma muy simple y superficial para poder seguir el resto del trabajo. Para más información sobre el tema y profundizar en sus bases es recomendable consultar Nielsen (2015a).

Las redes neuronales son, de forma concisa, una serie de nodos (llamados *neuronas*) que contienen una función (llamada *función de activación*). Los nodos se agrupan por capas y cada uno está conectado a todos los de la capa siguiente mediante unas conexiones llamadas *pesos*. Esto implica que podemos representar una red de neuronas con un grafo como el de la Figura 4.3.



Input Layer $\in \mathbb{R}^4$ Hidden Layer $\in \mathbb{R}^6$ Output Layer $\in \mathbb{R}^3$

Figura 4.3: Red neuronal simple, con 1 capa oculta

En toda red neuronal poseemos 3 tipos de capas principales:

- **Capa de entrada:** Codifica, como su nombre indica, la entrada de la red. La entrada de la red puede ser de muchos tipos dependiendo del problema, pero es típico adaptarla para que sea una serie de números reales, generalmente normalizados.
- **Capa de salida:** Representa una codificación de la salida de la red. Al igual que la entrada, la salida de la red puede ser codificada de diversas maneras, siendo común la representación mediante *one-hot encoding*.
- **Capa oculta:** Puede ser una o varias dependiendo de la red. Esta capa se conecta a la entrada y la salida y actúa como intermediario de la red, permitiendo el aprendizaje de esta.

Las técnicas de codificación más comunes incluyen:

- La normalización de valores que son números enteros o reales. En este caso la entrada puede ser, por ejemplo, una distancia, la intensidad de un color, etc. Con la normalización conseguimos reducir la posible diferencia entre valores, lo que permite un mejor aprendizaje de la red.
- El *One-hot encoding* es un tipo de codificación resulta vital para poder codificar valores de tipo *labeled*, es decir, valores discretos que se encuentran

etiquetados, como, por ejemplo, un tipo de animal (perro, gato, koala, hurón...). El *one-hot encoding* consiste en convertir estos valores en un array de tantos elementos como etiquetas existan y posteriormente rellenarlo con ceros en todos los elementos exceptuando el que se corresponda con la etiqueta a codificar, que tendrá valor 1.

El entrenamiento de una red consiste en dar valores a las conexiones entre neuronas (pesos) para que la salida sea la esperada en cada caso. El proceso más común de entrenamiento es utilizar el *stochastic gradient descent*, que consiste en realizar diversas iteraciones con subdivisiones de todos los datos de entrenamiento (llamadas batches). El *stochastic gradient descent* irá minimizando el error de la salida hasta un punto de estancamiento.

4.4.1.2. Introducir recurrencia a las redes

Una red neuronal clásica es buena realizando predicciones sobre una situación concreta o clasificando estados fijos, pero, ¿qué pasa si los datos que tenemos dependen de los que vinieron antes?

Las redes neuronales recurrentes (RNN para abreviar) son redes neuronales en las que la salida se conecta a la entrada de la propia red, por lo que la salida en un instante depende de las salidas anteriores de la red.

En estas redes podemos interpretar que la salida en un instante t depende de la entrada en ese instante y de las salidas en los instantes $t-1$ y $t-2$ de la red, como se muestra en la Figura 4.4.

Las redes neuronales recurrentes son el modelo ideal para realizar predicciones de modelos financieros, modelos de reconocimiento y tratamiento de texto (conocido como NLP) y, en nuestro caso, generar melodías que tienen en cuenta las notas anteriores de esta.

Sin embargo, las RNN tienen un problema grave, al ser entrenadas utilizando el método del *stochastic gradient descent* podemos apreciar el problema del *desvanecimiento del gradiente* (consultar más información en Nielsen (2015b)). Este problema se encuentra en redes neuronales profundas de muchas capas y en redes recurrentes, pues al calcular el gradiente a lo largo de todas las capas este va disminuyendo, llegando a ser muy pequeño a medida que se llega a las primeras capas de la red, esto hace que el entrenamiento se estanke y la red no consiga aprender correctamente. Este problema es intrínseco de estos tipos de redes, y es muy complicado evitarlo, para ello se utilizan modelos de redes neuronales recurrentes adaptados específicamente para reducir este problema.

4.4.1.3. Redes Long-Short Term Memory

Una forma de evitar el problema del *desvanecimiento de gradiente* es utilizar el modelo Long-Short Term Memory, dicho modelo fue presentado en Hochreiter y Schmidhuber (1997).

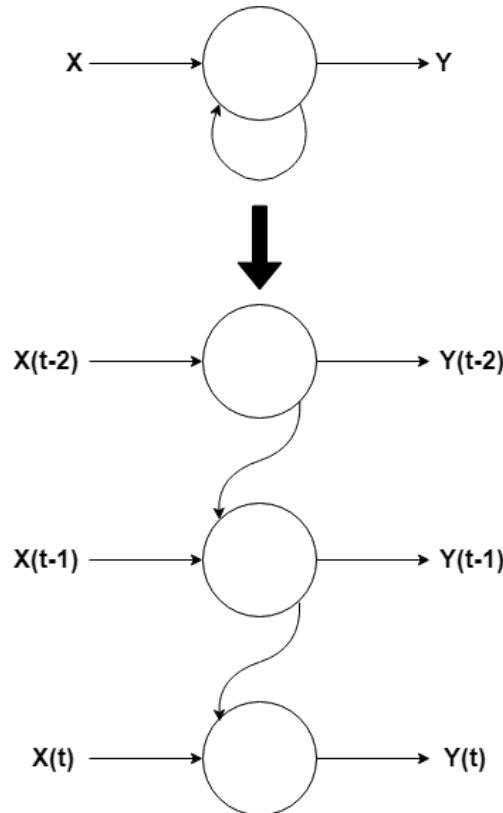


Figura 4.4: Despliegue temporal de una RNN

La idea es introducir 2 vías de memoria a la red, uno de memoria a corto plazo y otro a largo plazo. La memoria a largo plazo se conecta a toda la red y la memoria a corto plazo se conecta a cada capa de la red. Combinando ambos tipos de memoria conseguimos que la red no desvanezca el gradiente. Consultar Wikipedia (2017-2024) para más información.

4.4.2. Diseño de nuestra RNR

Para crear y entrenar nuestra red neuronal recurrente hemos utilizado la librería *Keras*, propiedad de Google y que utiliza *Tensorflow* como backend.

La red tiene las siguientes capas:

- **Capa input:** Una capa de entrada de 3 neuronas. La entrada consiste en el *pitch* de la nota actual, la duración de la nota en *steps* y el instante en el que suena la nota, también en *steps*.
- **Capa LSTM:** Una capa, como su nombre indica, de tipo *Long-Short Term Memory*. Esta capa hace el trabajo principal de la red, pudiendo procesar secuencias de notas para generar notas con un mayor contexto.
- **Capa output:** Una capa de salida con 2 salidas independientes. Por un lado la nota predicha y, por otro lado, el *start_time* de dicha nota.

La entrada se normaliza antes de entrenar y se guardan los parámetros de normalización (la media y desviación típica) en un archivo JSON, para posteriormente poder normalizar la entrada cada vez que se ejecute.

Ya que utilizamos una capa LSTM, la entrada tiene que venir dada en forma de secuencias. En nuestra red, se utilizan secuencias de 5 notas como entrada, para las cuales se predice una salida.

En el proceso de pruebas y mejoras del diseño de la red, se tomaron muchos enfoques. Una idea fue utilizar, como se dice anteriormente, una salida que prediga el tiempo de inicio de la nota, para introducir mayor dinamismo a la melodía. Sin embargo, dicho parámetro no aprendía correctamente patrones en el entrenamiento, aún utilizando una gran cantidad de datos, por lo que se decidió ignorarlo en la versión final. Aún así, para introducir silencios a la melodía se definen notas adicionales que representan silencios, lo cual producen mejores resultados que no tener silencios en toda la melodía.

Finalmente, en la salida no se elige una nota arbitrariamente, sino que existe una serie de notas que han aparecido a lo largo del entrenamiento. La salida se encuentra codificada utilizando *one-hot encoding* y representa la nota de tipo *label* que predice la red que es la más adecuada. Este acercamiento nos permite utilizar una última capa de tipo *softmax*, la cual permite añadir un factor aleatorio a la generación, ya que si no sería un modelo determinista y, por lo tanto, cada ejecución sería idéntica.

4.4.3. Resultados

Con el uso de RNR conseguimos mejorar la calidad de las melodías que se generaban utilizando cadenas de Markov. Al introducir secuencias de notas, la red posee mayor contexto y puede realizar predicciones más acertadas, así como disminuir el factor aleatorio.

Su ejecución resulta más cómoda, pudiendo variar la temperatura con la que generar para conseguir melodías más simples o más experimentales.

Como aspectos negativos cabe destacar que su ejecución es más lenta, teniendo que calcular cada iteración individualmente, aunque no supone un problema grave de rendimiento.

En definitiva, las redes neuronales recurrentes suponen un paso más en la calidad musical de las melodías, pero vamos a explorar modelos ya entrenados a continuación.

4.5. Generación con Magenta

Tal y como se dijo en el estado de la cuestión, Magenta contiene una serie de modelos de *machine learning* ya entrenados para disciplinas como la generación musical (Sección 2.1.4.2). Magenta posee varios paquetes que podemos utilizar para generar. A continuación se explican los diferentes paquetes y la integración en nuestra

aplicación.

4.5.1. Paquete de Magenta para Python

Existe un paquete de Magenta para Python, que puede ejecutar todos los modelos preentrenados.

Dicho paquete tiene instrucciones de instalación para sistemas Linux y MacOS (consultar Magenta (2023)). Sin embargo, a día de hoy no parece tener una versión compatible con Windows, por lo que no podemos utilizarlo en nuestra aplicación.

4.5.2. Paquete de Magenta para JavaScript

Magenta tiene también una versión para JavaScript, que se ejecuta sobre TensorFlowJS. Dicha versión es análoga a la de Python y permite ejecutar los modelos preentrenados.

Magenta Studio utiliza esta versión de Magenta, tanto para el plugin de Ableton como en la versión de escritorio, por lo que podemos utilizar esta versión para incluir Magenta en nuestro proyecto.

4.5.3. Magenta en nuestro proyecto

Para poder comunicar nuestro proyecto de Python con Magenta utilizamos el módulo de Python *subprocess*.

Podemos tener diversos scripts de NodeJS que son ejecutados por un script Python y se comunican por los argumentos del proceso y la salida estándar de este en formato JSON.

Actualmente tenemos 2 scripts:

- Uno dedicado a continuar melodías ya creadas (*magentaContinue.js*). El script utiliza *MusicRNN* para continuar la melodía dada.
- Otro para generar melodías (*magentaGenerator.js*). Este script utiliza *MusicRNN* para continuar una melodía de 1 nota (Do4) y así generar las melodías.

En la aplicación tenemos la posibilidad de generar melodías con Magenta, requiriendo una conexión a internet para descargar y ejecutar el modelo preentrenado.

4.5.4. Ventajas y desventajas de la generación con Magenta

Generar melodías con Magenta nos aporta varias ventajas:

- Son modelos entrenados con una gran cantidad de datasets y que utilizan técnicas avanzadas de *machine learning*, por tanto, la generación de melodías es bastante rica y estas poseen más coherencia interna.
- El *continue* nos permite alargar melodías y mantenerlas coherentes para poder trabajar con ellas posteriormente.

Sin embargo, como desventaja principal tenemos la necesidad de una conexión a internet para poder utilizar el modelo, así como el tiempo que tarda el modelo en inicializar, sobre todo al tener que manejar subprocessos.

Este modelo de generación nos aporta bastantes ventajas, pero es necesario mantener algún modelo que se pueda ejecutar de forma local y no dependa de módulos que a futuro puedan ser descontinuados.

Capítulo 5

Armonización

5.1. ¿Qué es armonizar?

Aunque armonizar pueda tener diversas connotaciones dependiendo del contexto, a partir de ahora, en este módulo del TFG, el término 'armonizar' se referirá específicamente a la tarea de encontrar una armonía adecuada para una melodía dada. Es decir, buscar la mejor secuencia de acordes, según unas normas establecidas, que acompañen y enriquezcan a la melodía. Esta secuencia de acordes será utilizada por posteriores módulos de la aplicación.

Cabe dejar claro que, buscar 'la mejor' armonía para una melodía en términos absolutos no es algo factible, ya que esta depende de la subjetividad de cada persona. Es por ello por lo que se prefiere hablar de buscar una armonía lo suficientemente coherente para una melodía dada, que forme parte del espectro de soluciones razonables, ya que, por lo general, una melodía puede ser acompañada por varios conjuntos de acordes distintos.

Obsérvese que, al resolver este problema, se estaría construyendo de forma implícita un analizador armónico. Si la entrada a este módulo fuese, en vez de únicamente una melodía, una canción completa, la cual contiene mucha más información, la salida esperada sería la armonía de dicha canción. Esto tiene mucha utilidad, ya que el análisis armónico es fundamental para el estudio en profundidad de una obra. La armonía conforma los cimientos de una composición musical.

5.2. Armonización por ventanas

Este fue el primer algoritmo diseñado, el cual escalará y evolucionará en posteriores apartados. Se basa en una idea, a priori, sencilla: dividir la canción en ventanas. Una ventana es una idea propia, y se define como conjunto de pulsos consecutivos que comprenden las notas que se hallan en dicho segmento. Un pulso también se puede traducir como una negra, un *beat* o un *step*. La idea es dividir la canción en ventanas de un determinado número de pulsos y asignar a cada una, es decir, al

conjunto de notas que abarca, el acorde que mejor encaje según una heurística.

Tanto la idea de las ventanas como la heurística utilizada, están estrechamente relacionadas con como funcionan los compases en la música real. En castellano, se define como compás tanto a la fracción que se encuentra al principio de un pentagrama, como a cada uno de los espacios comprendidos entre las líneas (horizontales) divisorias (Figura 5.1). El denominador de la fracción representa una figura musical y cada figura musical representa un determinado número de pulsos. En este caso, el denominador es el número 4, que representa una negra, la cual dura un pulso. El numerador nos indica la cantidad de figuras, por ende, la fracción $4/4$ nos está diciendo que un compás abarca $4 * 1 = 4$ pulsos.



Figura 5.1: Partitura en 4/4

Veamos otro ejemplo con el compás 6/8 (Figura 5.2). El denominador es el número 8, que representa una corchea. La corchea dura medio pulso. El numerador nos indica que en cada compás caben 6 corcheas, es decir $6 * 0.5 = 3$ pulsos.



Figura 5.2: Partitura en 6/8

Sabiendo esto, se puede establecer una clara analogía entre los compases y el concepto de ventana. La fracción nos indica el número de pulsos que abarca cada ventana, y las notas encerradas entre las líneas divisorias de cada compás serían las que cada ventana tiene en cuenta para elegir un acorde.

La idea es llenar una lista en la que cada índice represente cada ventana. Dependiendo de cómo de larga sea una canción y cómo de grandes sean las ventanas, el tamaño de la lista variará. Cada ventana contendrá información del peso de todos los acordes que hayan sido valorados para acompañar al conjunto de notas que abarque la ventana. Se elige finalmente el acorde de mayor peso para cada ventana. Si un acorde no aparece en la lista de acordes es porque tiene peso 0, es decir, ni siquiera se ha valorado como candidato para encajar en la ventana. El peso de los acordes se calcula a partir de la heurística que se explicará a continuación.

Antes que nada, se debe elegir qué acordes se quieren detectar. Por ejemplo, podríamos definir como candidatos las cuatro tríadas de la Tabla 2.7. Ahora bien, como existen 12 notas musicales, se van a valorar 48 acordes diferentes por cada

ventana, y eso solo teniendo en cuenta 4 tipos de acordes. Dicha cifra podría crecer mucho a la hora de considerar más tipos de acordes. A pesar de esto, el problema no está tanto en el rendimiento, sino en el hecho de que se estarán valorando acordes que contienen notas que no se encuentra en el conjunto de notas de la melodía, es decir, notas fuera de la escala, pudiendo así aparecer soluciones poco deseadas. Para paliar este problema, se escogerán únicamente acordes cuyo conjunto de notas esté comprendido en la escala que forma el conjunto de notas de la melodía. Es decir, se utilizará la armonía de la escala que forma el conjunto de notas de la melodía. Esto mitiga considerablemente el problema, pero no lo llega a solucionar, además de generar otros. Estos problemas se explicarán y se intentarán solucionar en apartados posteriores pero, por ahora, nos sirve esta solución, ya que esto no es más que una primera versión del algoritmo que se pretende construir.

A partir de ahora, se trabajará en términos relativos. De hecho, todo lo explicado anteriormente se ha implementado así, aunque en la explicación se haya querido abstraer. Toda la arquitectura de este módulo ha sido pensada para trabajar en estos términos relativos. Eso significa que, en vez de con notas musicales, se trabajará con grados e intervalos. Esto se logra estableciendo cualquier nota del conjunto de notas de la melodía como tónica (al menos por ahora; esto cambiará en futuros apartados), la primera mismamente, y relativizando el resto de las notas respecto a esta, es decir, transformando las notas en grados contando el número de semitonos que hay desde la tónica hasta cada nota (Tabla 2.2). Véase que, relativizar un conjunto de notas elimina implícitamente la información de la octava. Esto podría suponer un problema, pero en realidad nos beneficia al tener en cuenta el contexto armónico en el que estamos trabajando. Realmente, no se necesita información sobre la octava de cada grado, solamente saber a qué grado de la escala pertenece cada nota. También recalcar que el conjunto de grados o intervalos que se generan a la hora de relativizar todas las notas de la canción formarían ese esqueleto o molde de la escala del que se hablaba en la Sección 2.3.2. Aunque cueste más de entender, de esta manera se trabaja de forma más sencilla a la hora de ampliar funcionalidades en el módulo. Nótese que también se puede realizar el proceso inverso; es decir, a partir de una tónica y un conjunto de grados, hacer la traducción a notas musicales.

Terminando ya los preparativos previos a la explicación de la heurística, falta por entender el concepto de tick. La distancia entre dos ticks es la unidad simbólica mínima e indivisible que puede durar una nota. En cada canción se debe definir el número de ticks que dura un pulso. Cuanto mayor sea el número de ticks por pulso, mayor será el número de partes en el que puedes dividir el pulso. Haciendo una analogía con la representación simbólica que se utiliza en las partituras, si los ticks por pulso son iguales a 4, en nuestra canción la semicorchea sería la representación mínima que se podría utilizar, mientras que si fueran igual a 8, la fusa sería la representación mínima (una fusa es la mitad de una semicorchea). De todas formas, lo normal es que el número de ticks por pulso sea más alto para permitir mayor número de divisiones y combinaciones. Por lo tanto, sintetizando en lo importante, un tick representa un instante de tiempo dentro de la canción.

5.2.1. Heurística

Las distintas técnicas heurísticas que se han utilizado se basan en la idea de que las notas de una melodía coinciden en gran medida con las de la armonía, es decir, con las notas del acorde que le corresponden. Aunque esto puede no ser así siempre, por lo general, una melodía suele estar formada bien por notas del acorde, las cuales tienen más relevancia dentro del fragmento o por tensiones, notas fuera del acorde, que suelen utilizarse como notas de paso o adorno, con menos relevancia. Cada nota afecta a los pesos de los acordes de la ventana que comprende dicha nota. Por lo tanto, este proceso consiste en un recorrido de todas las notas de la canción.

Para mejorar la comprensión de la heurística se facilitará un ejemplo práctico para ver cómo se van calculando los pesos de los acordes en una ventana. A continuación, una vista previa del fragmento a analizar representado por un secuenciador arriba y por una partitura tradicional abajo: (Figura 5.3)

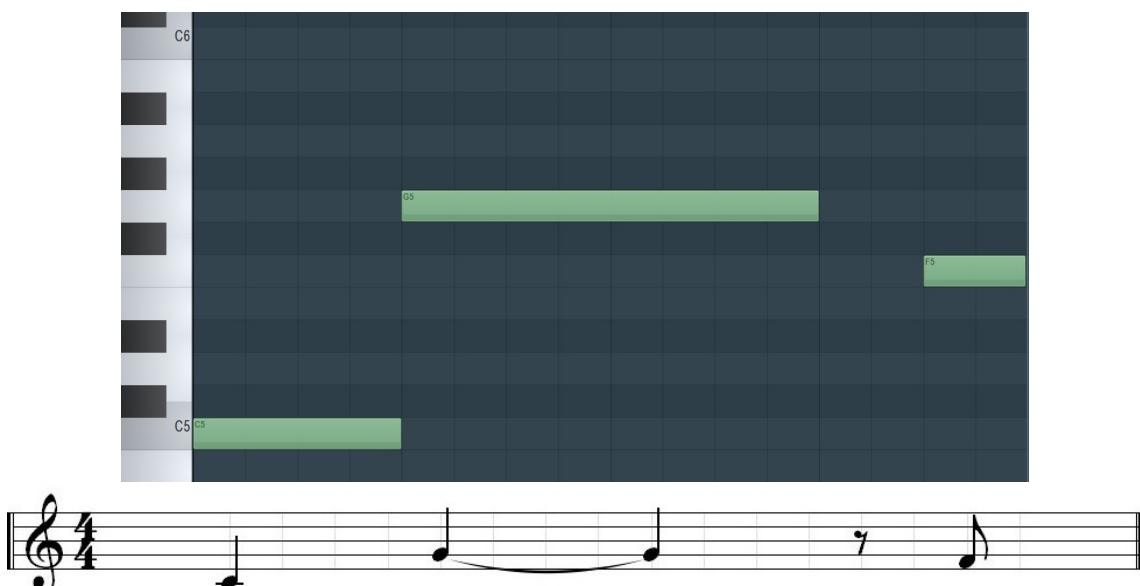


Figura 5.3: Ventana o compás a armonizar

Para simplificar el ejemplo asumimos que estamos en la tonalidad de C mayor. Como ya se ha explicado anteriormente, esto significa que la melodía se relativiza respecto a C y que se están utilizando como acordes candidatos los que comprenden la armonía de la escala mayor. Como se trabaja en términos relativos los acordes están representados por un par grado-tipo de acorde. El estado inicial de la ventana de pesos tiene, por ende, el siguiente aspecto: (Figura 5.1)

| Acorde | I | II- | III- | IV | V | VI- | VII-b5 |
|--------|---|-----|------|----|---|-----|--------|
| Peso | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Tabla 5.1: Estado inicial de los pesos de la ventana

Dicho esto, lo que se ha tenido en cuenta para valorar los pesos de los acordes en cada ventana ha sido lo siguiente:

- **Posición de la nota dentro del acorde:**

Partimos de la base de que si una nota no pertenece al acorde este será valorado con peso 0. En el caso contrario, el peso que la nota aporte depende de si esta pertenece al primer (tónica), tercer, quinto o séptimo (en caso de ser una cuatríada) grado del acorde, es decir, si es la primera, segunda, tercera o cuarta nota del acorde respectivamente. Los pesos aportados por cada grado se pueden cambiar a voluntad, pero, tal y como suele estar escrita la música, lo más sensato es que sigan el siguiente orden: tónica > quinta > tercera > séptima.

En el ejemplo propuesto los pesos aportados por cada nota dependiendo de su posición en el acorde son: (Tabla 5.2)

| Posición | Tónica | Tercera | Quinta |
|----------|--------|---------|--------|
| Peso | 1 | 0.25 | 0.5 |

Tabla 5.2: Peso por posición de la nota

Como existen tres notas en toda la ventana (C, F y G), los pesos de los acordes quedarían así: (Tabla 5.3)

| Acorde | Explicación | Peso |
|--------|--|------|
| I | C es la tónica del grado I: +1 G es la quinta del grado I: +0.5 | 1.5 |
| II- | F es la tercera del grado II: +0.25 | 0.25 |
| III- | G es la tercera del grado III: +0.25 | 0.25 |
| IV | F es tónica del grado IV: +1 C es la quinta del grado IV: +0.5 | 1.5 |
| V | G es la tónica del grado V: +1 | 1 |
| VI- | C es la tercera del grado VI: +0.25 | 0.25 |
| VII-b5 | F es la quinta del grado VII: +0.5 | 0.5 |

Tabla 5.3: Pesos tras la valoración de la posición de la nota

- **Posición de la nota dentro de la ventana:**

Aquí se vuelve a encontrar otra analogía con el cómo funcionan los compases en la música. El compás (fracción) no solo nos indica la duración del compás, sino que también nos da pistas de cómo se distribuyen las notas dentro del mismo. Por ejemplo, la fracción 4/4 nos está informando de que en cada compás caben cuatro negras. Aunque se pueda llenar el compás con figuras distintas a la negra, el compás 4/4 nos dice que los pulsos 1, 2, 3 y 4 marcan el ritmo de la canción, por lo que las notas que coincidan con estos pulsos van a ser las que lleven la voz cantante dentro de la melodía. Las notas fuera de estos pulsos clave tienen más papeletas de ser tensiones del acorde. Otro ejemplo para que se entienda mejor: la fracción 2/2 nos dice que en un compás caben dos blancas (el denominador 2 corresponde a la figura blanca, la cual dura el doble que una negra). Aunque la duración del compás sea la misma que la del 4/4, es

decir, 4 pulsos, esta nueva fracción nos estaría diciendo que los pulsos clave dentro de un compás son solo el 1 y el 3, difiriendo del primer ejemplo. De nuevo, aunque esto suela ser así, el compositor puede saltarse estas normas, pero tener en cuenta estos conceptos nos es beneficioso para acertar con la heurística.

Por lo tanto, a la hora de valorar los pesos de los acordes dentro de una ventana, esta se divide en ticks clave. Si la nota coincide con un tick clave, los acordes que contengan esa nota serán mejor valorados. Por eso la configuración de las ventanas está definida por tres parámetros: la distancia entre ticks clave (numerador de un compás tradicional), la cantidad de ticks clave (denominador; junto con la distancia entre ticks clave definen el tamaño de ventana) y los multiplicadores de cada tick clave. De forma similar a como pasa con la posición de la nota dentro del acorde, cada tick clave dentro de una ventana (o pulso dentro de un compás) ofrece un incremento distinto. Por ejemplo, en un compás 4/4, los pesos más sensatos atendiendo a cómo está escrita la música deberían tener el siguiente orden: Pulso 1 > Pulso 3 > Pulso 2 = Pulso 4.

Siguiendo esa configuración de ventana, en el ejemplo propuesto se han decidido elegir los siguientes multiplicadores por tick clave: (Tabla 5.4)

| Posición del tick clave | Primera | Segunda | Tercera | Cuarta |
|-------------------------|---------|---------|---------|--------|
| Multiplicador | 1.4 | 1.1 | 1.2 | 1.1 |

Tabla 5.4: Multiplicadores por posición del tick clave

Los ticks clave dentro de la ventana se verían así: (Figura 5.4)

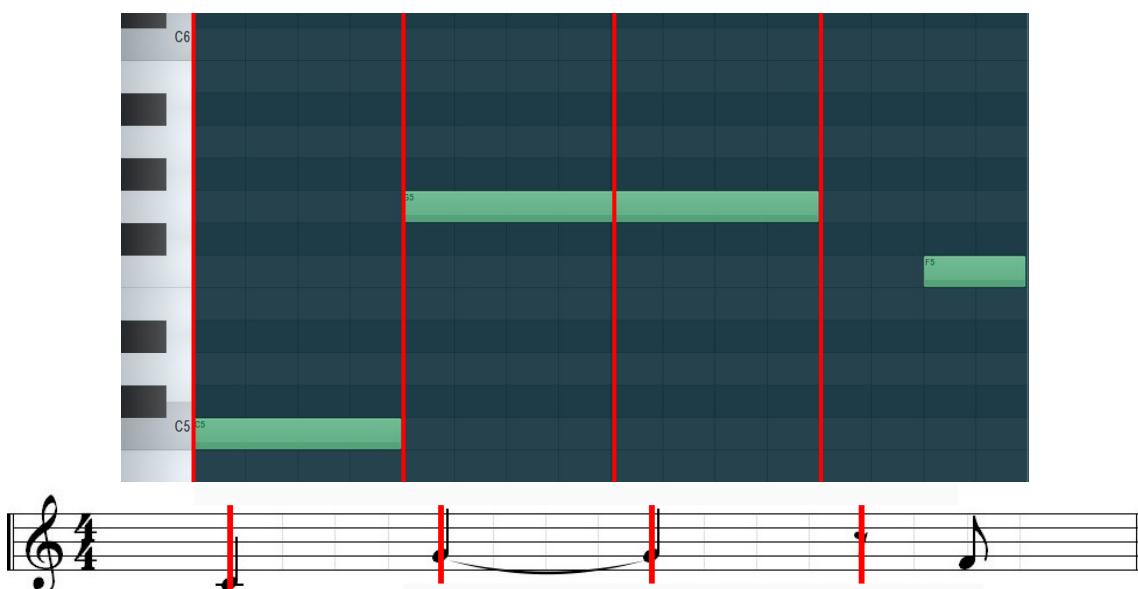


Figura 5.4: Ventana o compás a armonizar con los ticks clave marcados

Aplicando los multiplicadores, los pesos quedarían así: (Tabla 5.5)

| Acorde | Explicación | Peso |
|--------|---|-------|
| I | C comienza en el primer tick clave: $1*1.4$ G comienza en el segundo tick clave: $0.5*1.1$ | 1.95 |
| II- | F no comienza en tick clave: $0.25*1$ | 0.25 |
| III- | G comienza en el segundo tick clave: $0.25*1.1$ | 0.275 |
| IV | F no comienza en tick clave: $1*1$ C comienza en el primer tick clave: $0.5*1.4$ | 1.7 |
| V | G comienza en el segundo tick clave: $1*1.1$ | 1.1 |
| VI- | C comienza en el primer tick clave: $0.25*1.4$ | 0.35 |
| VII-b5 | F no comienza en tick clave: $0.5*1$ | 0.5 |

Tabla 5.5: Pesos tras tener en cuenta el multiplicador por tick clave

El programa está pensado para poder elegir a voluntad estos tres parámetros descritos anteriormente con la configuración que se deseé. Sin embargo, y debido a la incertidumbre de la melodía generada, por defecto se utiliza la configuración de ventana descrita en el ejemplo.

- **Discriminación entre notas que acaban de empezar y notas ya sonando:**

Esto tiene que ver con el anterior punto. Si la nota comienza en un tick clave, el peso que aporte será distinto a si esta pasa por tick clave cuando ya había sido emitida anteriormente, en cuyo caso, el peso que sume al acorde que la contenga se verá mermado.

La penalización utilizada en el ejemplo es de 0.75. La única nota afectada sería G, la cual se reproduce durante el tercer tick clave también. Los pesos de los acordes quedarían así: (Tabla 5.6)

| Acorde | Explicación | Peso |
|--------|------------------|-------|
| I | $+0.5*1.1*0.75$ | 2.363 |
| II- | $+0$ | 0.25 |
| III- | $+0.25*1.1*0.75$ | 0.481 |
| IV | $+0$ | 1.7 |
| V | $+1*1.1*0.75$ | 1.925 |
| VI- | $+0$ | 0.35 |
| VII-b5 | $+0$ | 0.5 |

Tabla 5.6: Pesos tras sumar las valoraciones por tick clave penalizado

- **Duración de la nota:**

Las notas que más duran otorgan más peso. Esto se logra de manera implícita por como funcionan los ticks clave. La nota siempre modificará los pesos de los acordes en su tick de inicio y cada vez que pase por un tick clave. Por lo que las notas más largas pasarán por más ticks clave, siendo entonces más relevantes a la hora de establecer cuál es el acorde ganador en cada ventana.

Este punto se podría haber implementado de otra manera (multiplicando el peso otorgado por la duración de la nota), pero entraría en conflicto con los dos puntos anteriores. Por lo que haciendo balance entre ambas soluciones se ha decidido mantener esta heurística ya que, por lógica, tendería a dar mejores resultados.

- **Velocidad de la nota:**

La velocidad de una nota es la intensidad de esta, es decir, como de fuerte o suave se toca. Por lo general, al interpretar una pieza musical, se destaca la importancia de ciertas notas tanto dentro de la melodía individual como en el contexto general de la canción. Por ello, valorar los acordes teniendo en cuenta la velocidad de las notas coincidentes es una manera sencilla de acertar con la heurística.

Sin embargo, este apartado no ha sido implementado ya que las melodías generadas no varían la velocidad de sus notas, por lo que no supondría ninguna mejora de cara a la aplicación. Si la melodía fuese grabada por un humano o generada artificialmente teniendo en cuenta las velocidades, este punto sí tendría más sentido.

El acorde ganador de la ventana del ejemplo sería C mayor (grado I). Los pesos finales quedarían así: (Tabla 5.7)

| Acorde | I | II- | III- | IV | V | VI- | VII-b5 |
|--------|-------|------|-------|-----|-------|------|--------|
| Peso | 2.363 | 0.25 | 0.481 | 1.7 | 1.925 | 0.35 | 0.5 |

Tabla 5.7: Pesos finales

5.2.2. Conclusiones

Uno de los primeros contras que ya se había mencionado en el párrafo 5.2 era el conjunto de acordes que se querían valorar. La solución propuesta en dicho apartado era utilizar la armonía de la escala que formaba el conjunto de notas de la melodía. Esto puede generar varias situaciones las cuales giran alrededor del hecho de que en la música a la que estamos acostumbrados, tal y como la conocemos, las escalas suelen tener 7 notas. Parte de estas situaciones son favorables, pero otras generan problemas, algunos de ellos mitigables, otros asumibles y otros insalvables:

- **La escala formada tiene menos de 7 notas:** en este caso se asume que la escala que forma la melodía es un subconjunto de una escala completa de 7 notas. En principio se utilizan las escalas mayor y menor (Sección 2.3), pero se podrían utilizar otras escalas menos utilizadas también. Se busca la tónica de una escala mayor o menor que englobe a todas las notas que forman la melodía, priorizando aquellas tónicas que estén representadas dentro de la melodía, de forma que, la armonía de la escala encontrada será la utilizada para armonizar la canción. En caso de encontrarse varios pares tónica-escala que resuelvan

el problema, se elige aquel cuya tónica sea la nota con mayor frecuencia de aparición en la melodía. En el caso de no haber solución se realizará el mismo proceso pero con las notas que no están representadas en la melodía. Esto, aunque raro, podría llegar a ocurrir. También es posible que, a pesar de todo, no haya solución. Una manera de paliar esto sería añadiendo más escalas a parte de la mayor y la menor, pero no es una solución completa. En estos casos se ignora el problema y se armoniza con la armonía de la escala que forma la melodía original, aunque el resultado pueda llegar a ser algo pobre.

- **La escala formada tiene 7 notas:** lo más probable en estos casos es que la escala formada sea una mayor, su relativa menor o un modo griego relativo. Se podría pensar a priori que es un problema que se elija aleatoriamente la tónica de la escala (recordemos del párrafo de la Sección 5.2 que se establecía como tónica la primera nota de la melodía), ya que se estaría armonizando para un modo o para una escala menor cuando realmente la melodía podría estar escrita en una escala mayor, por ejemplo. Pero como esta primera versión del algoritmo no tiene en cuenta las relaciones entre los acordes, la salida esperada en estos casos es siempre la misma sea cual sea la tónica (siempre y cuando esta pertenezca a la melodía original, claro).

También puede pasar que la escala formada no sea la mayor ni ninguna relativa a esta. En estos casos no se puede optar por la solución del apartado anterior ya que esta escala no puede ser ningún subconjunto de otra escala de 7 notas. Así que simplemente se utiliza la armonía de dicha escala para armonizar la melodía de forma natural.

- **La escala formada tiene más de 7 notas:** ocurre cuando se intentan armonizar canciones más largas o en melodías más 'vanguardistas'. De nuevo, se utiliza la armonía de la escala formada de forma natural. Aunque se pueda pensar que los resultados son un tanto caóticos debido a la cantidad elevada de acordes utilizados, la heurística consigue contradecir lo mencionado en el párrafo de la Sección 5.2, controlando este caos y consiguiendo resultados decentes. Lo que suele ocurrir en canciones más largas es que se suele modular a otras tonalidades y escalas durante su transcurso, de ahí que en la canción puedan llegar a aparecer las 12 notas musicales. Como la heurística funciona únicamente por pesos, esta es capaz de detectar estas modulaciones 'sin querer', obteniéndose resultados interesantes. De todas formas, esto no siempre es así y a veces sí que ocurre ese caos.

El siguiente problema, el cual se erradica en la siguiente versión del algoritmo, es el hecho de que se asume que todos los acordes que acompañan a la melodía tienen la misma duración. Como se divide la canción en fragmentos uniformes, todos los acordes que se forman tienen el tamaño de una ventana.

Por último está la problemática de que no se tiene en cuenta las relaciones entre los acordes. En futuras versiones se explicará por qué esto es un problema y cómo se ha solucionado.

En conclusión, y a pesar de las críticas descritas, los resultados obtenidos son bastante decentes. Esta primera versión supone las bases para la evolución del algoritmo y es un buen punto de partida.

5.3. Armonización por ventanas +

Como se comentó anteriormente, en esta sección se pretende arreglar el hecho de que se asume que todos los acordes que acompañan a la melodía tienen la misma duración. Esto supone un problema ya que no se ajusta a la realidad. En la mayoría de canciones dos o más acordes pueden compartir un mismo compás o un solo acorde puede abarcar varios compases, por ejemplo. Por ello, se han propuesto varias soluciones.

5.3.1. Armonización por ventanas de diferentes tamaños

La idea detrás de esta solución es armonizar la canción con diferentes configuraciones de ventana. Más concretamente, armonizar la canción con tamaños de ventana múltiplos entre sí y sumar los resultados en una estructura tan grande como la configuración de ventana que más fragmente la melodía, es decir, la que tenga el tamaño de ventana más pequeño. Esta estructura no deja de ser también una lista de ventanas, con la diferencia de que estas tienen mayor contexto de la melodía debido al sumatorio de los resultados de todas las configuraciones. Cada ventana afectará al resultado de todas las ventanas finales que abarque.

Con esta solución se consigue un resultado en el que es común que varias ventanas finales contiguas tengan el mismo acorde como el de más peso, de forma que cuando se imprima la solución los acordes idénticos contiguos se fusionarán en un solo acorde, consiguiendo así que la canción se armonicé con tamaños de acordes variables.

Cabe recalcar que, a pesar de que este algoritmo esté pensado para que los tamaños de ventana utilizados sean múltiplos entre sí, este está implementado de forma que se pueden elegir las configuraciones de ventana que se deseen para ser combinadas. De todas formas, no tiene sentido utilizar tamaños de ventana que no sean múltiplos entre sí, ya que la idea de esta solución es fragmentar la melodía respetando cierta estructura. Por ejemplo, un grupo de configuraciones sensatas teniendo en cuenta cómo está escrita la música sería: 4/4 (4 pulsos de duración), 2/4 (2 pulsos), 1/4 (1 pulso; este último se podría llegar a omitir, ya que puede generar mucha fragmentación no deseada). A continuación un ejemplo gráfico que pretende esbozar dicha configuración: (Figura 5.5)

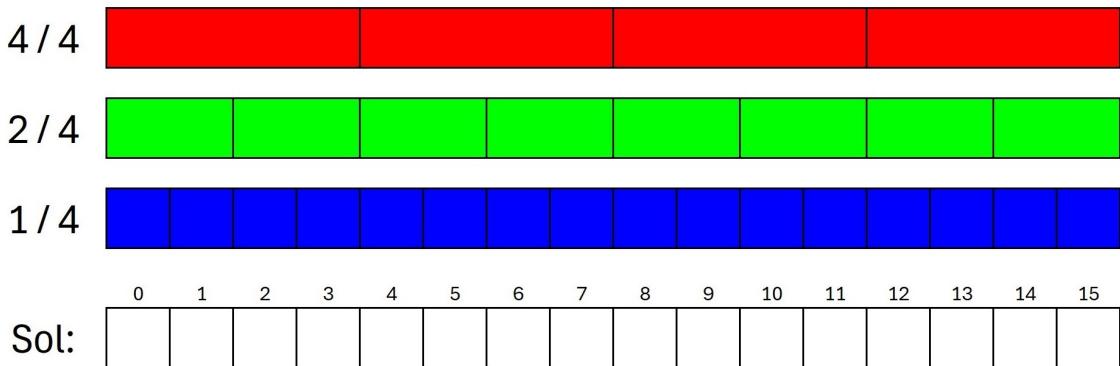


Figura 5.5: Armonización por ventanas de diferentes tamaños

5.3.2. Armonización por desplazamiento de ventana

Esta otra idea consiste en utilizar una única configuración de ventana, es más, abstrayendo para que se entienda mejor, consiste en utilizar una única ventana, la cual se va desplazando una determinada distancia a la que llamaremos *offset*. A medida que la ventana va recorriendo la melodía se va llenando una estructura de ventanas finales similar a la que se habló en el anterior método. Con lo dicho, se puede sacar como conclusión que el tamaño que tendrán esas ventanas finales será el tamaño del *offset*. De nuevo, lo que se busca con esta idea es que la estructura final tenga más contexto de la melodía que si hubiésemos hecho una armonización por ventanas básica con el tamaño de ventana del *offset*. Al igual que con el anterior método, a la hora de imprimir la salida los acordes idénticos y contiguos se combinan.

Haciendo una analogía al ejemplo descrito en el anterior algoritmo y, de nuevo, teniendo en cuenta cómo está escrita la música, una configuración sensata sería: ventanas de 4 pulsos ($4/4$) y tamaño del *offset* a 1 pulso (negra). A continuación otro ejemplo que ilustra la configuración descrita: (Figura 5.6)

5.3.3. Conclusiones

Sendos algoritmos consiguen resolver el problema planteado, pero las salidas de ambos para una misma melodía son diferentes. Al utilizarse la armonización por ventanas de diferentes tamaños (Sección 5.3.1), se consigue una fragmentación dentro de cada compás, pero respetando la separación entre sus contiguos, ya que los tamaños de ventana son múltiplos entre sí y, por lo tanto, múltiplos del más grande. Mientras que, con la armonización por desplazamiento de ventana (Sección 5.3.2), compases contiguos pueden 'contaminarse' entre sí. Esto no es algo malo como tal, ya que en algunas melodías esta característica es algo beneficioso, pero en general, probando ambos algoritmos, el que mejores resultados ha dado en promedio ha sido el primero.

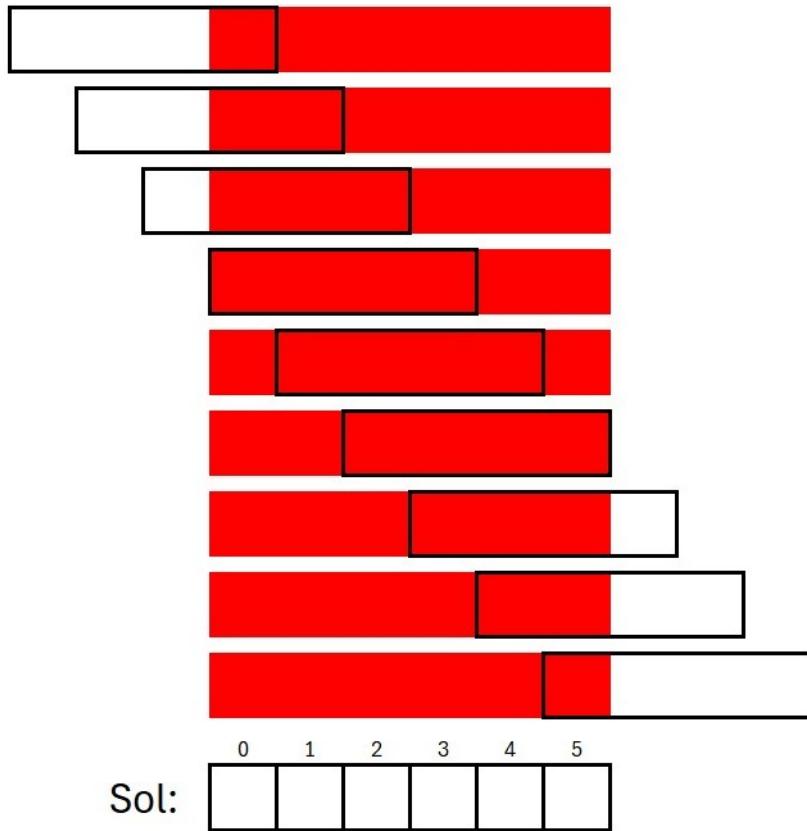


Figura 5.6: Armonización por desplazamiento de ventana

5.4. Relacionando acordes

Ahora toca resolver el hecho de que no se tienen en cuenta las relaciones entre acordes. Esto supone un problema debido a que los acordes no existen en un vacío, que es lo que erróneamente se ha estado haciendo hasta ahora. Cada uno tiene una relación con los acordes que le preceden y los que le siguen. Los acordes deben relacionarse de manera lógica y orgánica entre sí para formar una estructura musical sólida y convincente. En la música, abunda la teoría sobre las funciones individuales de cada tipo de acorde, así como de su papel dentro de la tonalidad a la que pertenecen y sus relaciones con sus predecesores y sucesores. Sin embargo, no se entrará en detalle en este mundo, basta con entender el problema: hasta ahora se está eligiendo siempre el acorde con más peso de cada ventana, cuando podría haber soluciones en las que elegir un acorde con menos peso en determinados fragmentos beneficiaría a la progresión armónica sin estropear la cohesión con la melodía, obteniéndose así salidas más ricas.

Las siguientes soluciones se basan en la misma idea: modificar los pesos de los acordes de cada ventana favoreciendo aquellos que encajen mejor con sus predecesores, cogiendo, de nuevo, el acorde con mayor peso después de dicha modificación. Se va adelantando que este método no ha sido el elegido para resolver este problema en la aplicación final, ya que los resultados obtenidos no han sido los deseados. Sin em-

bargo, es de bien entender sus debilidades, para comprender mejor el funcionamiento del módulo y futuros apartados.

5.4.1. Matriz de correspondencia

La idea de crear una matriz de correspondencia de acordes se basa en la de las cadenas de Markov (Sección 4.3). Se utilizará una matriz cuadrada (grafo dirigido) cuyas entradas sean pares grado-tipo de acorde, indicando la probabilidad que tiene un acorde de ir a cualquiera de los demás existentes. Los datos de dicha matriz se utilizarán para modificar los pesos de los acordes de una ventana dependiendo de qué acorde es el anterior. También existe una entrada para el inicio de una canción que modifica los pesos de la primera ventana.

El primer problema de este método se debe a la cuestión del cómo se va a rellenar (entrenar) dicha matriz de correspondencia. La principal idea sería utilizar el propio armonizador con canciones completas ya existentes y no solo con melodías. Como una canción completa es la unión entre melodía y armonía, el armonizador en este caso se comportaría como un analizador armónico. La salida obtenida se utilizaría para llenar la matriz. Al haber más notas que contribuyen a los pesos de los acordes, el resultado final no es tan ambiguo como en una melodía en solitario. Es decir, la progresión armónica obtenida es la propia que la canción original tiene. Cabe mencionar que, a pesar de la información extra que te ofrece una canción completa, el algoritmo seguiría estando sujeto a producir variaciones respecto a la armonía original pensada por el propio autor de la obra. Parte de estas variaciones sí que se podrían considerar errores que enturbiarían la matriz, pero no tendrían por qué. El hecho de que existan ambigüedades a la hora de realizar un análisis armónico es más común de lo que parece y no es algo problemático como tal: dos músicos expertos pueden diferir en opiniones a la hora de analizar ciertas piezas musicales.

El siguiente problema es heredado de las limitaciones que tiene el algoritmo en su estado actual: no existen métodos rigurosos para conocer ni la tónica de las canciones utilizadas para el entrenamiento, ni la tónica de la melodía que se pretende armonizar. Si recordamos, como hasta ahora no se habían tenido en cuenta las relaciones entre acordes, tampoco se había prestado mucha atención a cuál era la tónica de la canción. Esto ahora sí que supone un problema, ya que, en el primer caso, si se elige como tónica una nota errónea, la canción al completo se relativizará de forma errónea también, lo que desembocaría en el hecho de que toda la información que queremos recopilar para conocer las relaciones entre los acordes se transformaría en ruido para la matriz de correspondencia, ya que, como se comentó anteriormente, los datos se almacenan en entradas representadas por pares grado-tipo de acorde. De todas formas, juega a nuestro favor el hecho de que muchos archivos MIDI contienen mensajes especiales que nos facilitan la tónica de la canción. El verdadero problema surge en el segundo caso: aunque tengamos certeza de que los datos recopilados son de calidad, como desconocemos la tónica de la propia melodía, el sesgo ofrecido por la matriz nos sirve de poco y enturbiaría el resultado más que ayudar a mejorarlo (en el caso de equivocarnos a la hora de elegir la tónica de la melodía, claro). Se podría pensar en que almacenar los datos en la matriz de correspondencia de manera

absoluta, es decir, en pares tónica del acorde-tipo de acorde podría ser una solución viable. Nada más lejos de la realidad, esto supondría ignorar el contexto musical en el que se mueve la armonía de la canción analizada, enturbiándose así la matriz. Aquí un ejemplo: si estamos en un acorde de C (Do mayor) y pasamos a un acorde de G (Sol mayor), la transición no sería la misma si estuviéramos en la tonalidad de C mayor, donde la progresión en términos relativos sería de un primer grado a un quinto grado (I - V), que si estuviéramos en la tonalidad de G mayor, donde la progresión sería de un cuarto grado a un primer grado (IV - I).

De este apartado se ha desarrollado parte de la infraestructura necesaria para llevarlo a cabo, por ejemplo, los métodos que transcriben los datos en diferentes formatos o el propio algoritmo que aplica el sesgo a los pesos de los acordes. Debido a la falta de tiempo y a la dificultad de llevarlo a cabo, se decidió dejarlo a mitad. El problema de la tónica se podría haber mitigado de alguna manera; de hecho, están implementados métodos que pretenden, de vaga manera, resolver este inconveniente. Por otra parte, el hecho de encontrar de forma cómoda un *dataset* rico y abundante para construir la matriz bloqueó definitivamente que la idea se llevase a cabo (el *dataset* de Magenta (Sección 4.2), utilizado por otros módulos, solo ofrece melodías individuales). De todas formas, la matriz de correspondencia sí que se ha llegado a poner en práctica en el contexto de la armonía modal, poniendo las probabilidades de transición 'a mano', con cierta lógica, claro. Los resultados, aunque no horribles, no fueron convincentes. En definitiva, hubiese sido interesante poder escuchar los resultados que hubiese podido generar esta idea, pero por falta de tiempo se ha quedado pendiente como posible trabajo futuro.

5.4.2. Modelo de predicción de acordes

Una idea prometedora que consistiría en entrenar un modelo que, dado un acorde o secuencia de acordes previos, te devuelva una lista de probabilidades que afectarán a los pesos de los acordes de la siguiente ventana. A las problemáticas de la tónica y de la búsqueda de un *dataset*, se le suma la dificultad de buscar y entrenar al modelo adecuado. Lo único implementado de este apartado es la transcripción de los datos a un formato adecuado para dicho modelo.

Relacionado con esta sección, cabe citar la existencia de Hooktheory (2013-2024), una plataforma educativa diseñada para enseñar teoría musical y análisis de canciones populares. Hooktheory (2013-2024) es una herramienta en línea que permite a los usuarios crear, analizar y explorar progresiones de acordes y melodías, proporcionando retroalimentación en tiempo real sobre la teoría musical, ayudando a los usuarios a entender cómo funcionan las diferentes progresiones y estructuras musicales. La parte interesante relacionada con este apartado es que Hooktheory (2013-2024) cuenta con una base de datos interactiva que analiza las progresiones de acordes de miles de canciones populares. Dicha base de datos se podría utilizar para extender y profundizar en esta idea en un futuro.

5.5. Reconocimiento de progresiones de acordes

En esta sección exploraremos una solución alternativa al problema de las relaciones entre acordes que ataca el problema desde una perspectiva distinta: dada una lista predefinida de progresiones de acordes, se busca la secuencia de progresiones que mejor encaje con la melodía atendiendo a las restricciones establecidas por dicha lista. Esta solución es muy útil teniendo en cuenta cómo está escrita gran parte de la música: muchas de las canciones que escuchamos con regularidad comparten exactamente las mismas progresiones de acordes. Son miles y miles las canciones que utilizan los mismos clichés armónicos, normalmente de 4 acordes, aunque esto suele variar. Debido a esto, esta solución tienen bastante sentido; se busca la máxima cohesión posible con la melodía (según la heurística aplicada, claro), respetando una progresión armónica coherente, ya que, al ser el usuario quien define la lista, se sabe de antemano que dicha progresión funciona.

A continuación se mostrarán dos variantes que solucionan el problema. La primera busca una secuencia de progresiones sin importar que la última está cortada por la mitad y la segunda, una secuencia de progresiones de forma que la última *loopee* coherentemente con el primer acorde de la primera progresión. Previo a ambos algoritmos, se ha de definir una serie de políticas que determinen qué progresión se puede elegir en cada momento y de qué manera esta puede transicionar a otra. Aunque los resultados de ambos algoritmos puedan ser distintos, las normas utilizadas por estos son las mismas.

5.5.1. Política de elección de progresiones de acordes

Toca definir el formato de dicha lista y lo que representa su información. En la Tabla 5.8 se muestra un pequeño ejemplo del aspecto que tiene.

| Progresión | | | | | Tran. | |
|------------|-----|-----|----|----|-------|-----|
| 1 | I | VI- | IV | V | I | VI- |
| 2 | I | IV | I | IV | I | V |
| 3 | VI- | II- | V | I | IV | V |
| 4 | I | IV | V | | I | VI- |
| 5 | V | I | | | VI- | V |

Tabla 5.8: Tabla de progresiones

La lista está definida por un conjunto de progresiones conformadas por, al menos, dos acordes, definidos por un par grado-tipo de acorde. Cada progresión está ligada a una serie de acordes de transición. Podría no existir ningún acorde de transición asociado. El resultado es una secuencia de una o más progresiones definidas en la lista. Las progresiones ganadoras pueden diferir entre sí; aunque parezca obvio, con esta afirmación se quiere dejar claro que el resultado no tiene por qué ser una secuencia de la misma progresión todo el rato. Cuando una progresión termina, la siguiente progresión con la que enlace debe empezar obligatoriamente por alguno de

los acordes de transición definidos o por el último acorde de la propia progresión. Por el ejemplo, desde la primera progresión de la Tabla 5.8 se puede transicionar a todas las demás, ella misma incluida. Pero desde la segunda progresión, no se puede transicionar a la tercera.

Se ha elegido esta política como se podría haber elegido cualquier otra. Algunas de las otras valoradas son simplificaciones de esta. Por ejemplo, que no se tenga en cuenta el último acorde de la progresión actual para transicionar a la siguiente, que desde una progresión se pueda transicionar a cualquier otra sin ninguna restricción, que solamente se pueda utilizar una única progresión repetida en bucle hasta que la melodía termine o que, directamente, solo se pueda utilizar una única progresión una única vez, la cual abarque toda la melodía. Una vez entendido esto toca hablar de cómo se ha hecho y de las variantes que existen.

5.5.2. Secuencia de progresiones sin temor al corte

Esta es la primera variante creada. El título de la sección hace referencia a que el algoritmo buscará la mejor secuencia de progresiones sin importar que la última progresión elegida esté cortada por la mitad, es decir, puede caber la posibilidad de que los últimos acordes de la última progresión ganadora no formen parte de la mejor solución. Además, cabe aclarar que la duración de los acordes dentro de cada progresión es variable también, no solo para maximizar el resultado obtenido, sino también para evitar volver a convertir en problema algo que ya se había resuelto en versiones anteriores (Sección 5.3).

Se parte de la base de una lista de ventanas con la información de los pesos de los acordes candidatos obtenida por alguno de los algoritmos explicados anteriormente. Se ha de mencionar que, a partir de este momento, el conjunto de acordes que se valoran en dichos algoritmos ya no es el de la armonía de ninguna escala, sino el conjunto de acordes diferentes que el usuario haya definido en la tabla de progresiones. Por cada ventana se elige un acorde, respetando siempre las políticas establecidas (Sección 5.5.1), de forma que, la solución ganadora es aquella cuyo sumatorio de pesos entre todos los acordes sea el mayor. Este mismo proceso se repite 12 veces, una por cada posible tónica que podría llegar a tener una canción, y se elige la solución que más peso consiga. Resumiendo, el algoritmo se basa en una búsqueda en anchura. Se crea una estructura arbórea que recorre todas las posibilidades. Se utilizan ciertos mecanismos de poda, en los que no se entrará en detalle, que alivian bastante el coste del algoritmo hasta el punto de ser prácticamente imperceptible para el uso que se le da en la aplicación final. Es posible que si se ejecutase en canciones muy largas, con muchas notas, sí que se notara un poco más.

5.5.3. Secuencia de progresiones que *loopea*

La razón de la creación de esta solución se basa en una necesidad natural de la propia aplicación por cómo se van a construir los arreglos. Prácticamente todas las canciones contemporáneas más famosas se basan en una progresión de acordes que

se repite en bucle durante toda la canción.

Esto complica el algoritmo, ya que no solo se exige que la última progresión no esté cortada por la mitad, sino además, que esta transicione correctamente con el primer acorde de la secuencia para que el bucle sea válido. Vamos a profundizar más en lo que se considera un bucle válido, teniendo en cuenta las políticas establecidas (5.5.1), analizando las distintas situaciones posibles. Hay bucle si...

...el último acorde de la secuencia es el último acorde de la última progresión y este coincide con el primero de la secuencia.

...el último acorde de la secuencia es el penúltimo de la última progresión y el último acorde de la última progresión coincide con el primero de la secuencia.

...el último acorde de la secuencia es el último acorde de la última progresión y alguno de los acordes de transición de la última progresión coincide con el primer acorde de la secuencia.

En caso contrario, no hay bucle. Cabe mencionar que existe la posibilidad de que la mejor solución sea una secuencia de una única progresión. De hecho, es muy común en melodías de corta duración las cuales son utilizadas en la aplicación. Y también recalcar que se puede dar el caso de que una progresión no pueda *loopear* consigo misma, dependiendo de cómo se hayan definido los acordes de transición, siempre y cuando el primer y último acorde de la progresión no sean los mismos.

Una vez explicado todo esto, la implementación se basa en descartar aquellas soluciones finales que no generen un bucle con el principio. Además, el árbol pasa a aumentar su tamaño, ya que se debe ramificar una vez más por cada acorde inicial distinto que exista entre el conjunto de progresiones definidas, ya que, la solución final con más peso no tiene por qué *loopear*, siendo otra con menos peso la que sí que cumple los requisitos, limitando parte de la poda que se hacía en la anterior versión.

5.5.4. Conclusiones

5.5.4.1. Pros

Centrándome primero en las conclusiones de este apartado, ambos algoritmos, a parte de solucionar el inconveniente por el que se idearon en un principio, consiguen resolver implícitamente otros problemas. Se desvela por fin la incógnita de la tónica de la melodía. Como el recorrido en anchura se realiza tantas veces como notas existen, relativizando la melodía 12 veces respecto a dichas tónicas, se crean evidencias lo suficientemente convincentes como para afirmar que la tónica asociada a la solución ganadora es la real. No solo eso, sino que además se puede obtener información adicional sobre la escala de la melodía. A cada progresión de acordes le corresponde implícitamente una escala, por ejemplo, todas las progresiones descritas en la Tabla 5.8 pertenecen a la escala mayor, ya que dichos grados solo se

pueden encontrar, en un principio, en esa escala. Si se ampliase la tabla con algunas progresiones más y se añadiesen progresiones propias de la escala menor, o incluso de otro tipo de escalas más 'vanguardistas', se podría ampliar el armonizador para detectar la escala de la melodía, que junto a la tónica, supondría la posesión de una información muy valiosa para realizar multitud de tareas.

Otro punto a favor es la flexibilidad de ambos algoritmos de procesar cualquier salida de las heurísticas implementadas hasta ahora: tanto la armonización normal (Sección 5.2), como la armonización por ventanas diferentes (Sección 5.3.1) y por desplazamiento (Sección 5.3.2) devuelven una salida con el mismo formato, pudiéndose elegir con libertad cual procesar, explorando así con las diversas soluciones.

Por último, como ya se comentó anteriormente, la cuestión del conjunto de acordes valorados por dichas heurísticas se simplifica, pasando estos a ser los descritos por el usuario, eliminando la necesidad de tener que calcular la armonía de ninguna escala.

5.5.4.2. Contras

El principal punto en contra viene en el hecho de que la salida puede variar considerablemente dependiendo de las progresiones definidas en la tabla, no solo variando los acordes, sino también la tónica detectada, incluso pudiendo no haber solución si la lista está demasiado sesgada. De todas formas esto es algo normal: si le dices a dos músicos expertos, que en teoría, haciendo una analogía, deberían tener una tabla de progresiones de acordes bastante rica en sus cabezas, que armonicen la melodía, debido a la poca información y ambigüedad que esta ofrece, la libre interpretación de dichos músicos desembocaría también en secuencias de acordes dispares, incluso variando la tónica de la canción también. Que el algoritmo sea más predecible que versiones anteriores debido a la necesidad de la tabla de progresiones no es algo malo per se: le estás dando al usuario la capacidad de crear sus propias normas con las que generar música.

El otro punto en contra sería el coste asintótico del algoritmo respecto a anteriores versiones. Si se utilizase sobre composiciones más largas, con mayor número de notas, los tiempos de ejecución sí que serían más notorios. De todas formas, como ya se ha comentado anteriormente, para el propósito que se le quiere dar en la aplicación el coste del algoritmo es prácticamente imperceptible, además de la existencia de otros métodos de poda que mejorarían su rendimiento que no se han llegado a implementar por falta de tiempo y necesidad.

5.5.4.3. Conclusiones finales

Para concluir, simplemente decir que los resultados obtenidos con los métodos descritos en este apartado son más que convincentes. Se ha conseguido arreglar el último problema asociado a las relaciones entre acordes y, con ello, se ha conseguido un algoritmo que resuelve el problema planteado en este módulo con creces.

Búsqueda de nuevos cromatismos musicales

Este capítulo representa un punto de enlace entre melodía (Capítulo 4) y armonía (Capítulo 5), en relación con apartados posteriores que tratan sobre la creación de diversas temáticas (Sección 9.1). Dichas temáticas no solo varían en la utilización de instrumentos y técnicas a la hora de hacer arreglos musicales, sino en el uso de escalas y armonías que aporten sonoridades diferentes. Básicamente, se busca modificar las melodías originales con el objetivo de crear nuevos colores, que aporten una capa más de profundidad a la hora de crear nuevas temáticas.

6.1. Tratamiento de la melodía original

Debido a la incertidumbre y pseudo aleatoriedad de las melodías generadas en el Capítulo 4, sobre todo por Magenta (Sección 4.5), es necesario realizar un tratamiento previo de las melodías generadas.

El primer y más sencillo paso consiste en transportar toda la entrada a un rango de octavas razonable para una melodía. Para ello se calcula el *pitch* medio y se transportan todas las notas de la melodía de forma que el nuevo *pitch* medio se encuentre entre la quinta y la sexta octava. Las notas se pueden transportar una o varias octavas arriba o abajo, es decir, en un número negativo o positivo de semitonos múltiplo de 12. De esta forma se respeta la supuesta tónica original de la melodía.

El siguiente paso consiste en corregir las notas que se salen de la escala. Para ello, se ha de identificar primero la tonalidad de la melodía. El proceso es el siguiente: se armoniza la canción dos veces, la primera para un conjunto de progresiones propias de la escala mayor y la segunda, para un conjunto de progresiones propias de la escala menor. Recordemos que el armonizador nos facilita las tónicas para sendas generaciones. Se comparan los pesos de la mejor solución de ambas salidas y finalmente se elige la escala y tónica de la armonización ganadora. Con estos datos se ajusta la melodía a la tonalidad y se vuelve a armonizar fijando la tónica, mitigando

así el coste del proceso. No es necesario realizar el algoritmo para cada nota posible ya que conocemos la tónica. Aunque se podría pensar que este último paso es redundante ya que se podría utilizar la última salida asociada a la tonalidad ganadora, el hecho de ajustar las notas de la melodía podría provocar que la solución anterior variase, pudiéndose mejorar aún más la solución final.

La melodía ajustada y su armonía asociada se consideran la generación base y también serán utilizadas en parte de las temáticas.

6.2. Modos griegos

Una vez procesada la melodía original, se hará uso de la armonía modal (Sección 2.3.4) para crear nuevos colores. La idea es que dependiendo de la temática elegida se utilice la melodía base o uno de los modos griegos dependiendo del sonido buscado. El primer paso para conseguir dicho objetivo es ajustar la escala a la del modo griego, utilizando como tónica la calculada anteriormente y priorizando el ajuste de las notas ajenas a la propia tónica y a la nota de color.

6.2.1. Nota de color

La nota de color es una característica propia de los modos griegos. Para saber de dónde viene hay que entender que los modos griegos se pueden dividir en mayores y menores, dependiendo del acorde que se forme en el primer grado de la escala: si se forma una triada mayor, será un modo mayor, si por el contrario, se forma una triada menor en el primer grado, será un modo menor. La excepción es el modo locrio, en cuyo primer grado se forma una triada disminuida. Se considera un modo menor al presentar más similitudes con la escala menor que con la mayor, ya que su tercer grado también es una tercera menor.

| | | | | | | | | |
|------------------|--|---|---|---|----|---|---|----|
| Jónico | | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Lidio | | 1 | 2 | 3 | #4 | 5 | 6 | 7 |
| Mixolidio | | 1 | 2 | 3 | 4 | 5 | 6 | b7 |

Tabla 6.1: Modos mayores

| | | | | | | | | |
|----------------|--|---|-----------|----|---|-----------|----------|----|
| Eólico | | 1 | 2 | b3 | 4 | 5 | b6 | b7 |
| Dórico | | 1 | 2 | b3 | 4 | 5 | 6 | b7 |
| Frigio | | 1 | b2 | b3 | 4 | 5 | b6 | b7 |
| *Locrio | | 1 | b2 | b3 | 4 | b5 | b6 | b7 |

Tabla 6.2: Modos mayores

La nota de color de un modo se deduce comparándolo con la escala mayor o menor natural, según le corresponda. La nota de color es aquella que difiere entre ambas escalas. Dada esta definición se puede deducir que los modos jónico y eólico

no tienen nota de color, ya que, como ya se explicó en la Sección 2.3.4, estos son los nombres que reciben las escalas mayor y menor en un contexto modal, es decir, son, respectivamente, la misma escala. Un ejemplo de nota de color: si se observa la Tabla 6.2, la nota de color del modo dórico es la sexta, ya que difiere de la sexta menor de la escala menor natural. Por último, volver a resaltar la peculiaridad del modo locrio, a al cual le pertenecen dos notas de color. Entre la comunidad de compositores hay mucha divergencia de opiniones; algunos piensan que el modo locrio es demasiado raro y no merece la pena componer para él, mientras que otros sí que le dan una oportunidad.

6.2.2. Armonía modal

Una vez ajustada la melodía a la escala, priorizando la tónica y la nota de color, se han de elegir los acordes que pueden acompañar a dicha melodía. La teoría de la armonía modal nos ofrece varias pautas a seguir.

La primera consiste en evitar el uso de acordes disonantes, que contengan un intervalo de quinta disminuida entre algunas de sus notas. Quedan por lo tanto descartadas las triadas y cuatríadas semidisminuidas y los acordes de dominante, los cuales se pueden encontrar de manera natural en la armonía tonal.

La siguiente pauta consiste en establecer una jerarquía entre los acordes restantes, pudiendo estos ser:

- **Primario:** todo acorde que pertenezca al primer grado de la escala.
- **Secundario:** todo acorde que contenga la nota de color y no sea primario.
- **De paso:** todo acorde que no sea primario ni secundario.

A continuación un ejemplo en el que se analiza la armonía de la escala del modo dórico teniendo en cuenta las normas establecidas por la teoría de la armonía modal utilizada: (Tabla 6.3)

| Grados | Acordes | |
|--------|---------|------|
| I | - | -7 |
| II | - | -7 |
| bIII | | maj7 |
| IV | | 7 |
| V | - | -7 |
| VI | -b5 | -7b5 |
| bVII | | maj7 |

Tabla 6.3: Armonía del modo dórico

La idea a la hora de armonizar las melodías modales consiste en utilizar progresiones que hagan inciso en el uso de los acordes primarios y secundarios, consiguiéndose así 5 variaciones de la melodía y armonía base con las que crear diversas temáticas.

Nótese que son 5 los nuevos colores generados, ya que primero, se ha decidido utilizar también el modo locrio a pesar de sus peculiaridades y segundo, los modos jónico y eólico no son susceptibles de generar nuevas variaciones dada la teoría armónica modal utilizada.

Capítulo 7

Generar Percusión por Ordenador

La percusión¹ es una parte fundamental de una canción que aporta ritmo y sopor te al resto del arreglo musical. Para la generación de percusión hemos barajado dos opciones, la generación usando Magenta (2022b) (Sección 4.5) y la implementación de nuestro propio generador. La opción por la que nos hemos decantado finalmente es la de la generación propia, explicada en la Sección 7.1.2.

7.1. Generación de percusión

7.1.1. Generación de percusión con Magenta

Drumify es un programa sencillo dentro de la suite de aplicaciones de Magenta. Su uso es simple pero eficaz: se carga un archivo MIDI (Sección 2.4) de una canción, ya sea con acordes o sólo melodía, y devuelve otro archivo MIDI con un arreglo de batería que funciona para la canción proporcionada.

Este flujo de trabajo se puede completar usando otra herramienta que nos da Magenta, que en la versión de escritorio llaman *Groove*. El *groove* en una canción, en este caso aplicado a la parte de la percusión, es algo así como la sensación de movimiento que genera esta al escucharla².

La aplicación *Groove* de Magenta recibe un archivo MIDI, esta vez de percusión, y humaniza las notas, poniendo más velocidad en las notas que suenan en los golpes fuertes del compás, que generalmente serán el bombo y muchas veces la caja, mientras que otros instrumentos como los platillos tendrán menos velocidad. También varía ligeramente la posición de las notas, de forma que no quedan colocadas en el instante de tiempo exacto marcado por el compás, algo que hace que la percusión (y todos los instrumentos en general) suenen bastante robóticos.

¹<https://rayrojodrums.com/es/que-son-las-percusiones/>

²[https://es.wikipedia.org/wiki/Groove_\(musica\)](https://es.wikipedia.org/wiki/Groove_(musica))

7.1.2. Generación de percusión propia

Una vez vistas las herramientas que nos brinda Magenta para la generación de MIDIs de percusión, llegamos a la conclusión de que era algo que, si bien funcionaba, era demasiado genérico para lo que buscábamos. Por esta razón, no usaremos Magenta en la herramienta.

Hemos implementado nuestro propio generador de percusión, que nos permite trabajar mejor en los estilos que buscábamos. Se presentarán en la Sección 7.1.4.1. Este generador crea tres archivos MIDI de un compás 4/4 de duración del estilo especificado. A continuación se explica su funcionamiento.

7.1.3. ¿Cómo enfocamos la percusión?

Con el fin de poder programar algún tipo de generador de ritmos de batería, vamos a tratar de simplificar al máximo la forma de crear partes de percusión para una canción.

El enfoque que vamos a darle es similar al que se explica en el vídeo de YouTube del canal "8-bit Music Theory", (Dru (2021))

Separamos las baterías en tres partes principales:

- **Beat:** Suelen ser notas en los golpes fuertes. Es la parte fundamental del ritmo. Generalmente suele ser el bombo el encargado de esta función.
- **Engine:** Son golpes que contrastan con el *beat*. Añaden movimiento al ritmo. Si en el *beat* tenemos un bombo en el *engine* podríamos tener una caja. Pueden ir en los golpes fuertes o en los débiles.
- **Constant:** No destaca especialmente pero añade consistencia al ritmo. Esta función suelen desempeñarla los platillos. Suelen ir en los golpes débiles.

A lo largo de esta sección, vamos a ir desarrollando un ejemplo de ritmo generado. Comenzamos usando una codificación propia más sencilla que el MIDI: con tres arrays de tamaño 16 representamos nuestras notas de percusión, siendo cada posición del array una semicorchea. De esta forma:

- El patrón para el *beat* básico es el siguiente: [1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0], siendo 1 un golpe de percusión y 0 un silencio. Para generar variedad, se cambian un poco las notas de forma aleatoria siguiendo unas normas. Para el ejemplo obtenemos este patrón en el *beat*: [1,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0]
- De igual manera, el *engine* básico es el siguiente: [0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0]. Para el ejemplo, se ha retrasado el segundo golpe, resultando en un patrón [0,0,0,0,1,0,0,0,0,0,0,0,1,0,0].
- Para el patrón de *constant* básico, el patrón es [1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0]. Se usará este mismo patrón sin modificaciones para el ejemplo.

Ya que lo que estamos representando son percusiones, no es necesario guardar el final de la nota en la representación que estamos usando, puesto que en los instrumentos de percusión alargar la nota MIDI no alarga el sonido.

Juntando estas tres partes obtenemos un ritmo de batería que podemos mantener durante una parte o la totalidad de la canción. Crearemos un archivo MIDI con las notas del ritmo de percusión a partir de los arrays que hemos visto previamente.

En la Figura 7.1 vemos el ritmo básico en partitura y en la Figura 7.2, en el rodillo de piano. En ambas figuras la línea de abajo representa el *beat*, en este caso un bombo, la de arriba representa la *constant*, en este caso platillos, y la línea central el *engine*, en este caso un golpe de caja. En la Figura 7.3 vemos el ritmo que ha creado el generador en el ejemplo que hemos visto anteriormente.

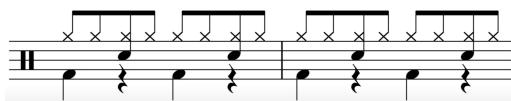


Figura 7.1: Ritmo básico de percusión.

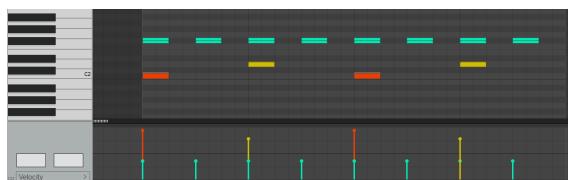


Figura 7.2: Ritmo básico de percusión en el rodillo de piano.

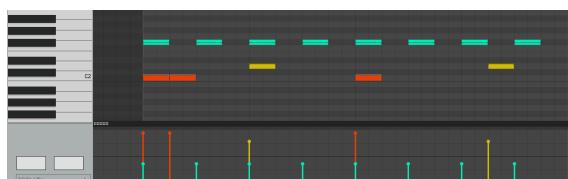


Figura 7.3: Ejemplo del patrón de percusión resultante en el ejemplo.

Una vez tenemos el ritmo base, podemos añadir algunos adornos. Algunos ejemplos de adornos podrían los *drum fills* o las intros.

7.1.4. *Drum Fills*

Un *drum fill* es un pequeño motivo musical que sirve para añadir variedad a un ritmo de batería y para hacer transiciones entre secciones de un arreglo musical³. A veces los *drum fills* pueden aparecer indicados en las partituras de los bateristas, pero en otras ocasiones sólo viene indicado dónde tocar un *fill* y es el propio baterista el que improvisa un *drum fill*.

³[https://es.wikipedia.org/wiki/Fill_\(musica\)](https://es.wikipedia.org/wiki/Fill_(musica))

La primera idea para lograr tener *drum fills* fue la de implementarlos como el resto de las partes de percusión. Esta idea fue descartada debido a que los *drum fills* usan más elementos de la batería y son más creativos que los ritmos que hemos visto en el punto anterior.

Por tanto, para obtener *drum fills*, cargamos en Reaper (Sección 2.5.2) un archivo MIDI (Sección 2.4) con algunas notas de percusión marcadas, concretamente las de la Figura 7.4. A continuación, añadimos a la pista el plugin arpegiador *BlueArp* (Sección 8.2.3) para que genere una secuencia de notas a partir de las notas de percusión del ítem MIDI cargado. Para nuestro ejemplo los *drum fills* quedarán como en la Figura 7.5.

Por último, añadimos un plugin humanizador (Sección 8.2.4) que además de variar la velocidad y *timing* de las notas, varía la nota que se interpreta. Esto lo que hace en un instrumento de batería es cambiar el elemento de la batería que se toca (este paso se obvia en el ejemplo por motivos de claridad). Lo que obtenemos como resultado es una secuencia de golpes de batería generadas de forma aleatoria que al combinar con el ritmo base da como resultado un *drum fill*.



Figura 7.4: Ítem MIDI auxiliar usado para generar *drum fills*



Figura 7.5: Arpegiador y resultado final de un *drum fill*

Además, si esta secuencia que hemos generado suena sin el ritmo base, cumple la función de una intro de batería, que da pie a que entre el ritmo al comienzo del siguiente compás.

Los *drum fills* los colocaremos cada 4 compases, siendo cada 8 algo más largos, al igual que cuanto más avanzado se encuentren en la canción.

7.1.4.1. Estilos

Todos los patrones que se generen serán variaciones de un patrón básico. Por ejemplo, en la Figura 7.6 podemos ver un patrón básico de batería como el que vimos anteriormente y a su derecha un patrón de jazz. Si nos fijamos, podemos observar que el *beat* y el *engine* son similares, cambiando los golpes de percusión con los que se tocan dichas partes (platillos en ambos casos).

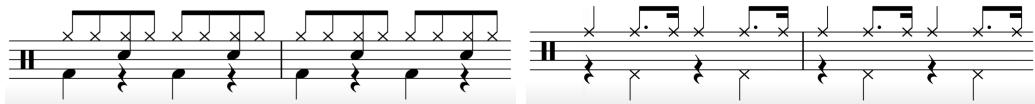


Figura 7.6: A la izquierda, ritmo estándar básico de percusión, a la derecha, ritmo estándar de jazz

Los patrones básicos son los propios de varios géneros musicales. Con estos patrones tratamos de cubrir la mayor parte de ritmos posibles:

- **Estilo básico:** Ritmo más básico de bombo y caja con platillos como *constant*. Este es el estilo del patrón de ejemplo que hemos visto anteriormente.
- **Palmas:** Ritmo de palmas con varias voces simultaneas
- **Maracas:** Ritmo con una maraca como *constant* con un bombo ocasional.
- **Jazz:** Platillos como *beat* y haciendo un ritmo contrastante como *constant* y un platillo de pedal haciendo el *engine*.
- **Disco:** Bombo a negras con caja y platillos.
- **Metal:** Platillos haciendo el *beat* y doble bombo como *constant*
- **Latin:** Bombo, caja y platillos haciendo ritmos que contrastan entre sí.
- **Rock:** Caja haciendo el *beat* y bombo la *constant*
- **Dembow:** Bombo y caja en el que cada segunda caja se retrasa ligeramente.

7.1.4.2. Generación de patrones

Como se ha visto antes, se parte de un patrón de batería básico, como los de la Figura 7.6. De manera aleatoria se decide omitir algunas notas, adelantarlas, tocar dos veces en lugar de una, etc. De esta forma obtenemos un patrón de batería derivado de uno de los patrones básicos, como el que hemos visto anteriormente en el ejemplo en la Figura 7.3.

Variando ligeramente este patrón, crearemos otros dos que serán muy similares al original y también entre ellos.

7.1.4.3. Combinación de patrones

Como se ha explicado en el punto anterior, hemos generado tres patrones distintos de un mismo estilo que además funcionan bien juntos. Llamaremos a estos patrones A, B y C.

Es común en los arreglos de percusión combinar estos patrones de cierta manera para añadir variedad al ritmo básico. Por ejemplo, podemos encontrarnos una secuencia bastante común ABAC o una secuencia AABB, que no sería más que tocar

de una en una en secuencia los patrones correspondientes, volviendo a empezar al llegar al final de la secuencia.

Estas secuencias las generamos de manera aleatoria una vez, por lo que para cada canción generada saldrá una secuencia distinta pero a lo largo de una canción la batería mantiene esa secuencia continuamente.

Dependiendo de la temática escogida para la canción, se cargarán unos estilos de batería u otros, generalmente tres estilos distintos que van cambiando a lo largo de la canción de manera aleatoria.

En la Figura 7.7 vemos el MIDI de batería de ejemplo completo de 8 compases (sin contar el compás de intro) con una secuencia ABAC del ritmo generado y los *drum fills* añadidos, tanto para hacer una intro como cada cuatro compases.

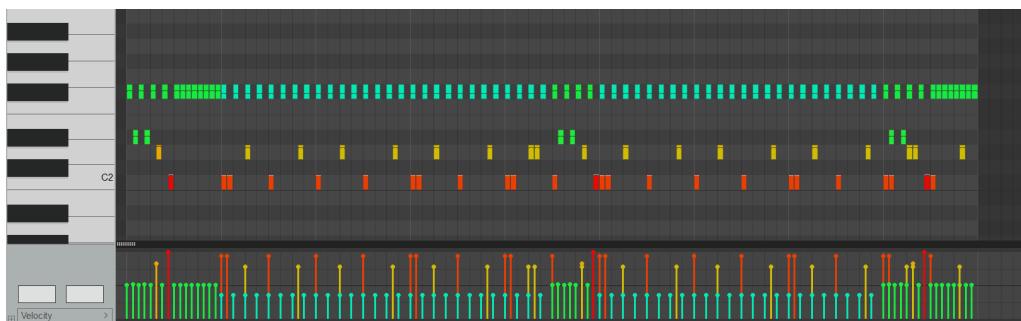


Figura 7.7: Ritmo resultante completo

7.1.4.4. Percusión en MIDI

A la hora de interpretar un archivo MIDI (Sección 2.4) de percusión, la mayoría de instrumentos virtuales mapean las notas de la misma forma. Por ejemplo el bombo es la nota MIDI 36 y la caja la nota MIDI 38 (C3 y D3 respectivamente).

La velocidad de las notas MIDI en los instrumentos de percusión indican con cuánta fuerza se realiza el golpe. En algunos instrumentos virtuales, la velocidad de la nota se ignora; en otros, se reproduce el sonido correspondiente con más o menos volumen; y en otros, existen muestras específicas grabadas para distintas velocidades de la nota. Asimismo, es común que en los instrumentos virtuales de percusión haya varias muestras para una misma nota y una misma velocidad, con el fin de que no suene siempre el mismo golpe exacto y el ritmo resultante suene más realista.

Capítulo 8

Sonorización del Proyecto

Con el fin de cubrir todas las temáticas disponibles (Sección 9.1), hemos reuniendo una serie de plugins que nos permiten contar con un catálogo de timbres muy diversos. Todos los instrumentos utilizados son gratuitos. Además, hemos priorizado que sean plugins ligeros, de forma que la herramienta final no requiera de demasiado almacenamiento. No obstante, el usuario final de la herramienta puede cambiar fácilmente dentro de Reaper los instrumentos cargados por otros plugins externos de su elección que haya instalado.

Además de generar sonidos, hay multitud de tareas que pueden realizar los plugins, como transformar el sonido o el MIDI, añadir reverberación, eco u otros efectos que veremos a continuación. Estos plugins se denominan plugins de efectos.

Reaper incluye algunos plugins de efectos de base, pero apenas tiene instrumentos virtuales, al contrario que otras DAWs populares que incluyen una buena cantidad de estos en su versión básica. Los efectos principales se nombran más adelante (Sección 8.2).

8.1. Presets

Reaper nos permite guardar configuraciones de un plugin para usarlas más adelante, esto es lo que se conoce como preset. La mayoría de plugins también permiten la opción de guardar presets dentro de su propia interfaz. Y de hecho, suelen traer algunos presets ya creados por el fabricante del plugin con sonidos o efectos que funcionan y que hemos aprovechado para nuestros presets.

8.2. Plugins utilizados

Vamos a detallar los timbres y los efectos más importantes presentes en las canciones generadas por la herramienta así como los plugins concretos que hemos usado para generarlos.

En la Figura 8.1 podemos ver la cadena de efectos que utilizamos para cada pista. Cabe destacar que el orden de los plugins importa, ya que se deben colocar los plugins de efectos MIDI los primeros (empezando por arriba), a continuación los instrumentos virtuales y por último los plugins de efectos de audio.



Figura 8.1: Efectos utilizados en Reaper

La lista completa de plugins que hemos utilizado junto con sus respectivos enlaces de descarga puede encontrarse en el Apéndice A.

8.2.1. Instrumentos virtuales

Hemos usado un gran número de plugins de instrumentos virtuales para lograr un amplio catálogo de timbres que cubren las distintas temáticas (Sección 9.1) y ofrecen variedad tímbrica para estas, ya que tenemos varios presets para cada pista (Sección 9.5) y temática. Hemos priorizado la facilidad de instalación a la vez que la variedad, por eso usamos principalmente instrumentos sintetizados que simulan los timbres que buscamos.

Tanto en la Figura 8.2 como en la Figura 8.3 se pueden ver las interfaces de dos de los instrumentos virtuales utilizados en la temática de desierto (Sección 9.1): el primero es un instrumento virtual con varios sonidos de palmas, mientras que el segundo es un instrumento virtual de un sitar, un instrumento tradicional de cuerda propio de la India y Pakistán. En ambos casos se pueden observar controles para modificar el sonido obtenido usando el plugin, por ejemplo la cantidad de reverberación o el volumen. En el caso del plugin Clap Machine hay menos parámetros que en el otro, que es más completo y personalizable.



Figura 8.2: Interfaz del plugin Clap Machine de 99Sounds



Figura 8.3: Interfaz del plugin Zither Renaissance de Sample Science

8.2.2. Plugins auxiliares

Utilizamos algunos plugins que nos proporciona Reaper para realizar algunas tareas:

- **Transponer notas:** Modificamos la señal MIDI recibida para subir o bajar semitonos dependiendo de la configuración recibida desde la app. En algunas pistas también usamos este plugin para subir o bajar una octava entera.
- **Comprimir y limitar:** Cargamos dos plugins, ReaComp y ReaLimit, para intentar igualar el volumen lo máximo posible entre los distintos instrumentos.
- **Ecualizar:** Cargamos un plugin ReaEQ para ecualizar el sonido. Junto con la compresión, logramos que el sonido de la pista se mezcle con el del resto de pistas de la mejor forma posible, dando a cada pista su espacio en el espectro de frecuencias.

8.2.3. BlueAmp

BlueAmp es uno de los plugins MIDI principales que utilizamos. Es un tipo de plugin que recibe una señal MIDI y la convierte en otra. Para que esto funcione, se debe colocar antes que los instrumentos virtuales en la cadena de efectos, para transformar el MIDI antes de que sea convertido en sonido.

El principal uso del plugin *BlueAmp* es hacer el trabajo de un arpegiador. Los arpegiadores reciben un acorde y lo transforman en un acorde arpegiado (Sección 2.3.3.2). En nuestro caso, a parte de usarlo como arpegiador, lo usamos también para generar ritmos de acordes, líneas de bajo (Sección 9.8), *drum fills* (Sección 7.1.4) y sonidos extra para los *ear candy* (Sección 9.10).

En la Figura 8.4 podemos ver la interfaz del plugin *BlueAmp*. En la parte derecha de esta imagen podemos observar el patrón que transformará el MIDI entrante, un patrón de arpegio común. Este patrón tiene en cuenta la posición de las notas en un acorde comenzando por la nota más grave teniendo en cuenta que el acorde puede estar invertido (Sección 2.3.3.1).

En la Figura 8.5 podemos ver primero un acorde de C en su forma básica. Esta será la señal MIDI que reciba el plugin arpegiador. A su derecha, la transformación que ha sufrido el MIDI tras pasar por *BlueAmp* (con la configuración de la Figura 8.4).



Figura 8.4: Interfaz de *BlueAmp* haciendo un arpegio

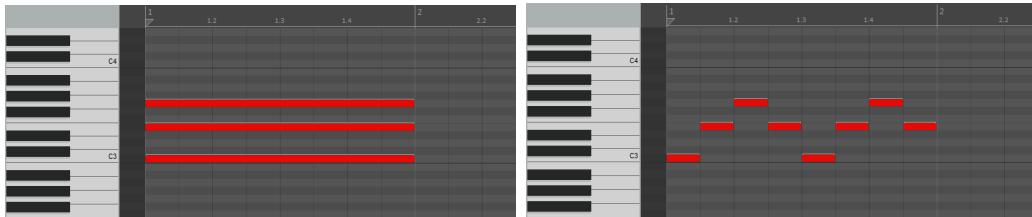


Figura 8.5: Acorde de C en MIDI arpegiado

BlueArp no sólo sirve para arpegiar las notas, también puede ser usado para crear patrones melódicos. En este contexto los patrones melódicos corresponderían a células rítmicas que se repiten (cada poco tiempo generalmente) variando las notas pero no el ritmo. Un ejemplo de patrón melódico sería un arpegio.

Como puede verse en la Figura 8.4, el plugin cuenta con varios parámetros que podemos personalizar de. Entre los más importantes se encuentran el cambiar el número de *steps* o pasos que tiene el patrón melódico, la figura rítmica en la que se basa el arpegio (en la imagen una corchea), la intensidad de cada una de las notas, cambios de octavas, etc.

A parte de generar arpegios, puede usarse *BlueArp* para generar acordes enteros en una misma figura (en lugar de sólo una nota).

Para crear una línea de bajo, cargamos un MIDI con acordes en la pista del bajo y usamos *BlueArp* para que la señal MIDI resultante sólo tenga una nota cada vez, que será principalmente la nota fundamental del acorde (puede verse un ejemplo en la Sección 9.8).

También se usa el arpegiador para hacer *drum fills*, como se explica en la Sección 7.1.4.

8.2.4. Humanizador

A la hora de producir una canción, se van a crear los sonidos usando instrumentos virtuales en lugar de grabarlos, el productor puede encontrarse con que la música generada puede sonar robótica, sin vida. Esto se debe a que el MIDI es muy preciso en velocidad y tempo, hasta el punto de que no es realista el sonido que se obtiene, ya que ningún músico toca con la precisión perfecta de un ordenador.

Para solucionar este problema existe la humanización. En Reaper, podemos humanizar las notas en el propio rodillo de piano pulsando la tecla H. Como la idea es que nuestra herramienta requiera la menor intervención humana posible, hemos explorado otras alternativas para humanizar el MIDI.

Usamos un plugin MIDI que transforma el MIDI en otra señal MIDI. En concreto, hemos elegido el Humanisator de Toby Bear. Con este plugin transformamos una señal MIDI con una velocidad fija y a tempo exacto en una señal MIDI humanizada. Por ejemplo, en la Figura 8.6 convertimos el acorde que arpegiamos con *BlueArp* en la Figura 8.5 en un acorde arpegiado humanizado (el color representa la velocidad de la nota), que ahora se parecería bastante a lo que un teclista grabaría con un

controlador MIDI¹.

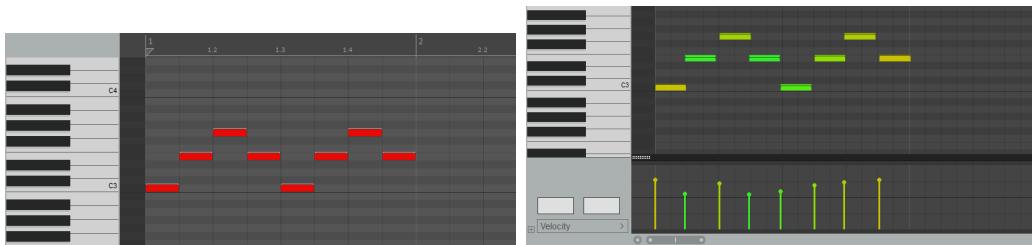


Figura 8.6: Acorde de C en MIDI arpegiado humanizado

8.2.5. *Delay*

El efecto de *delay* genera un eco de un sonido recibido, repitiendo ese sonido de forma atenuada cierto tiempo después.

Usamos este efecto para añadir algo más de presencia a los *ear candy* (Sección 9.10) en algunas ocasiones. Para lograr este efecto, usamos el plugin CRMEL de UnplugRed.

8.2.6. *Reverb*

La reverberación es el efecto encargado de recrear espacios acústicos, pues simula el rebote del sonido en distintas superficies. Sin nada de reverberación, es más difícil lograr que un sonido suene natural.

La reverberación puede añadirse a un sólo sonido o a una mezcla de sonidos, lo que hace que suenen mejor en conjunto, como si estuvieran situados en el mismo espacio acústico.

La manera correcta de añadir reverberación a una mezcla es realizar un envío² desde todas las pistas a una nueva pista, en la cual cargaremos nuestro plugin de *reverb* con la señal *dry* a menos infinito (como se ve en la Figura 8.7), es decir, en esta pista sólo sonará la *reverb*. De esta forma podemos controlar cuanta señal enviamos desde cada pista a la *reverb*. Esto es muy útil por ejemplo para enviar más señal de los instrumentos principales y para enviar menos señal del bajo y la batería y así no ocupen más espacio del que les toca en la mezcla. En esta pista cargaremos el plugin OrilRiver de Denis Tihanov (Figura 8.7), aunque el usuario puede reemplazarlo por cualquier otro plugin de reverberación de su elección que tenga instalado.

¹<https://soundsmarket.com/blog/controlador-midi>

²<https://dnamusic.edu.co/los-efectos-de-envio/>



Figura 8.7: Interfaz del plugin de reverberación Oril River

8.2.7. Mezcla

El máster en Reaper es una pista por la que pasan todos los sonidos del resto de pistas, es la pista con el sonido de la mezcla final. Por tanto, cuando cargamos un plugin en la pista de máster, afecta a todos los sonidos de la canción. Los plugins que cargamos en el máster de base se pueden ver en la primera pista de la izquierda en la Figura 8.1.

A continuación usamos un compresor y un limitador para darle algo más de volumen y cohesión a la mezcla final, un ecualizador para tratar distintas frecuencias dependiendo de la temática de la canción, de nuevo otra instancia del OrilRiver para expandir el estéreo de la mezcla y por último, un plugin AFTER de TWestProductions, que se encarga de masterizar³ de forma muy sencilla la canción.

8.2.8. Efectos en la mezcla

Desde la ventana de la herramienta (Sección 3.4) podemos marcar algunos *checkboxes* que añadirán varios plugins de efectos adicionales al máster para modificar el sonido final de la canción.

Estos plugins son los encargados de que la mezcla final suene retro, *vintage*, *lofi*, etc. Pueden llegar a modificar bastante la señal de audio original que llega al máster, por lo que al aplicar varios de ellos a la vez la calidad del sonido puede disminuir drásticamente y llegar a hacer que la mezcla suene mal.

Los efectos disponibles son los siguientes:

- **Retro:** Utilizamos un plugin *bitcrusher* que reduce las profundidad de bits del audio y por tanto, su calidad para simular el sonido de una consola antigua.
- **Bajo el agua:** Utilizamos el plugin Deja Vu de Cymatics que reduce la velocidad del audio a la mitad para ralentizar el ritmo de la canción y lograr una sensación de que está sonando dentro de agua.

³<https://www.landr.com/es/que-es-la-masterizacion/>

- **Lofi:** Aplicamos un filtro de lofi a la muestra que reduce la calidad de forma creativa, filtra frecuencias y añade sonidos de polvo en una grabación. El plugin usado para lograr este efecto es el Unison Zen Master.
- **Vintage:** Utilizamos una cadena de efectos formada por varios plugins: una reverberación analógica, un pedal de *delay*, saturación y un filtro que simula que el sonido ha pasado a través de una cinta de grabación antigua.
- **Dream:** Usamos un plugin que ralentiza un poco la velocidad de la canción y un efecto de *chorus* analógico que además añade desafinación ocasional al sonido. Para lograr sendos efectos usamos los plugins Deja Vu y Memory respectivamente, ambos de Cymatics.
- **Espacial:** Colocamos una reverberación muy grande con el *dry* a menos infinito, por lo que no suena la señal que llega al máster si no únicamente la reverberación. En este caso la *reverb* usada es Valhalla Super Massive.

8.3. ReaScript

Reaper ofrece una API llamada ReaScript, la cual permite a los usuarios crear y ejecutar scripts personalizados. Esto es útil a la hora de automatizar tareas, mejorando la funcionalidad del programa. Por defecto, Reaper solo permite la ejecución de scripts de Lua y de EEL (*Embedded Extensible Language*), pero también cabe la posibilidad de ejecutar scripts de Python cambiando la configuración.

Una de las posibilidades que ofrece ReaScript es la de comunicar Reaper con otros sistemas, pudiendo ampliar las características de la DAW con programas de terceros.

Nuestra herramienta se conecta con Reaper a través de ReaScript, ejecutando sus propios scripts en Reaper. Esto permite sonorizar el MIDI generado en nuestra herramienta a través de la API de ReaScript, haciendo uso de los plugins mencionados en este capítulo. De este modo, aplicamos cada plugin pertinente en función de la temática sonora que queramos conseguir (Sección 9.1).

Capítulo 9

Cómo arreglar una canción

Hacer un arreglo de una canción consiste en utilizar una idea musical, una composición o una canción entera para crear una canción nueva.

En nuestro caso vamos a usar un motivo o idea musical para arreglar una canción que funcione como música para un videojuego. Hay muchas formas de abordar este tema, y como es común en la música y el arte, no hay una verdad absoluta.

Por nuestra parte, priorizamos que el arreglo funcione para que sea rico y variado, con varias secciones y sonidos posibles, de manera que dos generaciones distintas no sean nunca iguales. Bajo esta premisa, hemos definido una serie de normas para la estructura y timbres de la canción, que se generarán de manera pseudoaleatoria siguiendo dichas normas.

9.1. Temáticas

Con el fin de abordar distintas ambientaciones de videojuegos, así como distintos géneros musicales, hemos decidido separar las canciones generadas en varios grupos, que llamaremos temáticas.

Para lograr que una canción suene a una temática en concreto o transmita una sensación son importantes varios factores, entre ellos los timbres, el tempo de la canción o la tonalidad.

Tanto para abordar algunas temáticas como para la idea de separar en temáticas teniendo en cuenta los factores mencionados, nos hemos inspirado en los vídeos de YouTube del canal "Ludofonia" (Lud (2020)).

La mayoría de adaptaciones según el tema se realizan en Reaper (Sección 2.5.2) mediante el uso de plugins, efectos, selección de tempo y selección de la escala de la canción. Por lo general, la distribución de los ítems MIDI no varía, por lo que dos generaciones con la misma semilla pero temáticas distintas tendrán la misma estructura pero con distintos timbres, tempo, ritmos de percusión, etc.

A continuación, desglosaremos cada temática disponible en la herramienta, ex-

plicando de qué forma hemos enfocado cada una:

- **Pradera:** Canciones en modo lidio con un tempo de en torno a 120BPM. Usamos timbres de pianos, guitarras y algunos instrumentos de cuerda, junto con un bajo eléctrico y distintas cajas de ritmos para la batería.
- **Piano:** Usamos la tonalidad original del MIDI cargado o generado por la herramienta. El BPM es aleatorio dentro de un amplio rango. A la hora de hacer el arreglo sólo tenemos dos pistas sonando simultáneamente como máximo, y todas las pistas tienen el mismo instrumento virtual de piano.
- **Desierto:** Usamos el modo frigio para esta temática, además de usar varios timbres característicos como sitars, flautas, guitarras, percusiones, palmas, etc. Para más información ver Lud (2020).
- **Nieve:** Modo dórico con BPM lento. Sonidos suaves y agradables, timbres característicos como campanas o kalimbas.
- **Pirata:** Modo mixolidio y BPM elevado. Timbres característicos como acordeones, violines, percusiones variadas, guitarras, etc. Algo muy importante mencionado en el vídeo de YouTube del canal "Ludofonia" sobre la música pirata (Lud (2023a)) es que la melodía suele ir en 3/4 o atresillada. Mediante el uso del plugin *BlueArp* (Sección 8.2.3) modificamos las melodías para que encajen mejor bajo esa premisa.
- **Selva:** La mayoría de pistas cargan percusiones además de otros sonidos típicos de jungla como las marimbas. Para esta temática nos ayudamos del exotismo del modo frigio, además, el tempo utilizado es bastante rápido. Para más información ver Lud (2022a).
- **Épico:** Varios instrumentos orquestales, en su mayoría de cuerda y viento metal, además de timbales para la percusión. Usamos el modo mixolidio.
- **Tenebroso:** Modo locrio, sonidos no muy agradables sonando en varias pistas. Para más información ver Lud (2022b).
- **Agua:** Modo mixolidio y BPM lento. Sonidos suaves y algunos efectos de agua generados con sintetizadores. Para más información ver Lud (2023b).
- **Asiático:** Usamos el modo lidio y mezclamos instrumentos tradicionales asiáticos con una batería moderna y un bajo eléctrico.
- **Rock:** Modo dórico. Cargamos varias guitarras eléctricas con amplificadores con distorsión junto con una batería de rock.
- **Pop:** Usamos la tonalidad original del MIDI cargado o generado por la herramienta. Varios instrumentos de música pop actual como guitarras, pianos, sintetizadores, etc.

- **Tecno:** Usamos la tonalidad original del MIDI cargado o generado por la herramienta. Percusión electrónica golpeando el bombo a negras haciendo un efecto de *sidechain*¹ sobre varios sintetizadores (este efecto de *sidechain* lo simulamos con una envolvente).

9.2. Las partes fundamentales de una producción

El proceso de producir una canción es complejo y bastante creativo, pero podemos reducirlo a una serie de normas básicas con el fin de simplificarlo.

Proponemos hacer producciones a partir de los siguientes cuatro elementos fundamentales: La melodía, el acompañamiento, el bajo y la percusión.

- **Melodía:** La melodía es el elemento protagonista y más distintivo de una canción. Es una sucesión de notas que se perciben como una única entidad, siendo generalmente la parte más memorable de una composición. Más info sobre la generación de melodías en la herramienta en el Capítulo 4.
- **Armonía:** La armonía acompaña a la melodía. Para ello, usaremos progresiones de acordes, como se describe en el Capítulo 5.
- **Bajo:** El bajo se encarga de las frecuencias graves de una canción, proporcionando la base rítmica y armónica sobre la cual se sustentan los demás elementos. El bajo no solo apoya la armonía, sino que también refuerza el ritmo y añade profundidad a la mezcla. En la Sección 9.8 se puede ver cómo creamos las líneas de bajo usadas en la herramienta.
- **Percusión:** La percusión es el componente principal encargado de llevar el ritmo en una canción. A través de diversos ritmos, la percusión establece el tempo y la dinámica de la pieza, siendo fundamental para la cohesión de la mezcla. En la Sección 7.1.2 se puede ver cómo generamos los ritmos de percusión usados en la herramienta.

9.3. Secciones

Estructuramos la canción en un máximo de 8 secciones, las cuales tienen una duración de 8 compases cada una. Lo ideal para que una canción no se vuelva repetitiva es añadir o quitar elementos cada poco tiempo, siendo lo más común cada 8 compases.

El usuario puede, desde Reaper, elegir si quiere *renderizar* la canción entera de 8 secciones, o seleccionar un rango de secciones para *renderizar* únicamente esas, ya que mientras sean secciones enteras y no se corten a la mitad, *loopearán* de forma correcta sea cual sea la selección.

¹<https://es.wikipedia.org/wiki/Sidechain>

9.4. Arreglo

Generamos una matriz de booleanos de 7x8 la cuál rellenamos de maneras aleatoria, y que posteriormente corregimos siguiendo unas normas:

- No puede haber dos pistas de acompañamiento sonando simultáneamente.
- En la primera sección no puede haber más de tres pistas principales sonando a la vez. Entendemos como pistas principales las que cubren los fundamentos de la producción (Sección 9.2), es decir, las 7 primeras pistas.
- Siempre ha de haber al menos una pista principal sonando por sección.

En la Figura 9.1 podemos ver una captura de pantalla que muestra un arreglo que la herramienta ha cargado en Reaper.

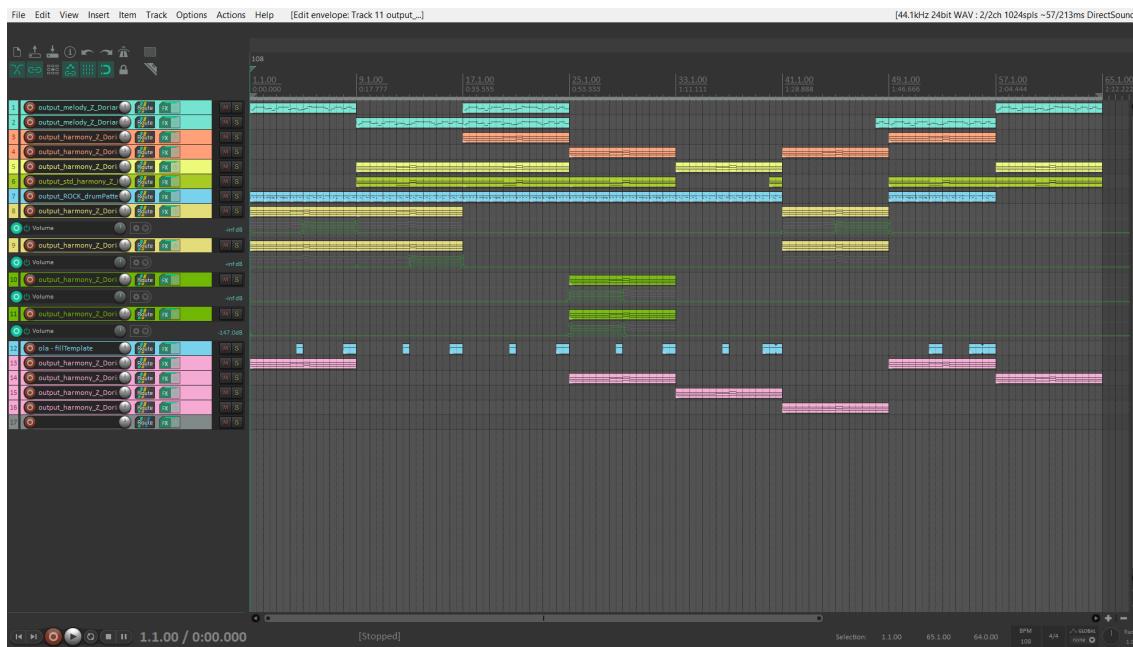


Figura 9.1: Arreglo generado en Reaper

9.5. Pistas

Dependiendo de la temática (Sección 9.1) de la canción, la herramienta cargará en Reaper varios efectos (Sección 2.5.3.1) por pista. La mayoría de los plugins MIDI y de efectos cargados son los mismos siempre, pero el instrumento virtual que interpreta el MIDI cargado se selecciona de manera aleatoria de entre una serie plugins VST seleccionados para cada temática concreta. Las cadenas de efectos de las pistas se puede ver en la Sección 8.2.

Las pistas de la 1 a la 16 de la Figura 9.1 contienen distintos ítems MIDI que son interpretados, mientras que la pista 17 sirve como pista de reverberación. Dicha pista de reverberación se explica en la Sección 8.2.6.

A continuación se detalla el contenido de las pistas 1 a 16.

9.6. Melodías

Una vez tenga una melodía, el usuario puede determinar, desde la ventana de la aplicación de la herramienta, la complejidad con la que se cargarán las melodías en Reaper. Hay tres complejidades:

- **Estándar:** Cargamos el MIDI de melodía entero en Reaper, por tanto la melodía se repetirá cada 8 compases, como se puede ver en la Figura 9.2.

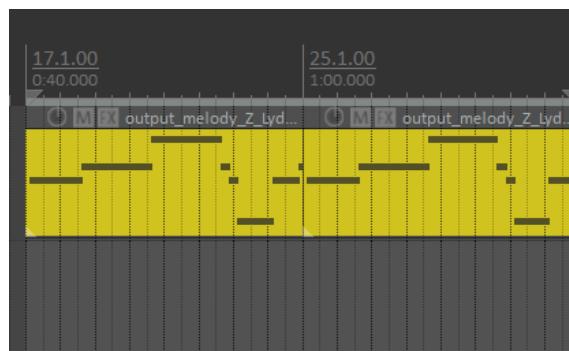


Figura 9.2: Ítems MIDI en complejidad estándar

- **Repetitiva:** La repetición en la melodía es clave para que la canción sea predecible y sea fácil de recordar. En esta complejidad las melodías se repiten cada dos compases. Para lograr esto, troceamos el MIDI de melodía original proporcionado en cuatro partes y las cargamos de forma pseudoaleatoria, como se puede ver en la Figura 9.3.

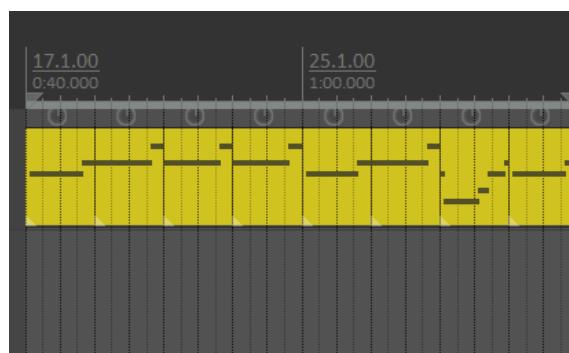


Figura 9.3: Ítems MIDI en complejidad repetitiva

- **Súper repetitiva:** Similar a la complejidad anterior, pero esta vez con ítems MIDI todavía más cortos. Se repite la secuencia de ítems dos veces por cada 8 compases, como se puede ver en la Figura 9.4.

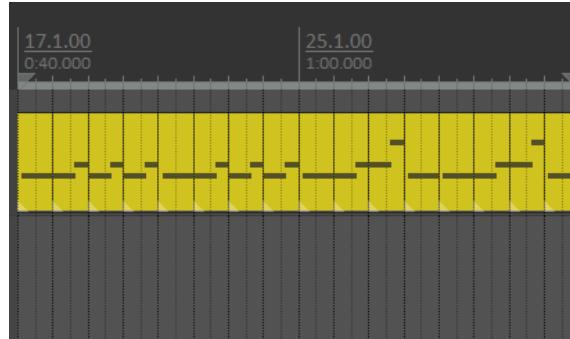


Figura 9.4: Ítems MIDI en complejidad súper repetitiva

Como se puede ver en la Figura 9.1, se cargan en Reaper dos pistas de melodía, las dos primeras, en color azul en este caso. En esta pista van los MIDIs de melodía.

9.7. Armonía

Para cubrir la parte de la armonía en el arreglo generado por la herramienta, se usarán tres pistas: dos de acompañamiento de la melodía propiamente dicho y una pista de *pads* más en segundo plano.

Como se puede ver en la Figura 9.1, se cargan en Reaper dos pistas de acompañamiento, las pistas 3 y 4 en color naranja. Estas pistas utilizan un arpegiador (Sección 8.2.3) que hará que en ocasiones los acordes sean arpegiados o tocados con otro ritmo distinto al del MIDI proporcionado. Por esta razón no pueden sonar al mismo tiempo, ya que es probable que arpegios generados suenen mal en conjunto. Estas pistas cargan el MIDI de armonía (Capítulo 5).

La pista de *pads* también carga el MIDI de armonía, por lo que también interpreta los acordes de la canción. En la Figura 9.1 se ve en la pista 5 de color amarillo. Cuando suena sin el acompañamiento hace la propia función de acompañar a la melodía algo más en segundo plano, y cuando suena a la vez que el acompañamiento complementa el sonido de este.

9.8. Línea de bajo

Al comienzo del desarrollo de la herramienta implementamos un generador de líneas de bajo que creaba, a partir de una armonía dada, una melodía monofónica para ser interpretada por el bajo, usando principalmente la nota más grave de cada acorde (que puede no ser la fundamental, ya que estospueden estar invertidos). Este generador es similar al generador de percusión (Sección 7.1.2) pero simplificado.

Este algoritmo tenía algunas limitaciones y a la hora de ampliarlo para añadir variedad, decidimos optar por una alternativa más cómoda y flexible.

En la versión actual de la herramienta, cargamos la armonía (Sección 5.1) (sin inversiones) en Reaper en la pista de bajo y utilizamos el plugin *BlueAmp* (Sección 8.2.3) para interpretar la línea de bajo usando principalmente la nota fundamental del acorde y en ocasiones el resto de notas de dicho acorde.

Veamos un ejemplo: en la Figura 9.5 se puede ver el MIDI cargado mencionado previamente. Tanto en la Figura 9.6 como en la Figura 9.7 se puede ver un patrón de *BlueAmp* de línea de bajo junto con el MIDI resultante en el rodillo de piano.

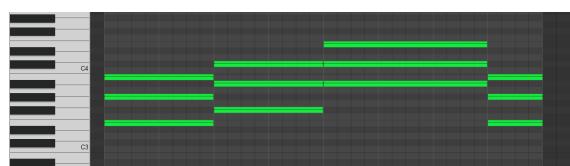


Figura 9.5: Progresión de acordes sin inversión

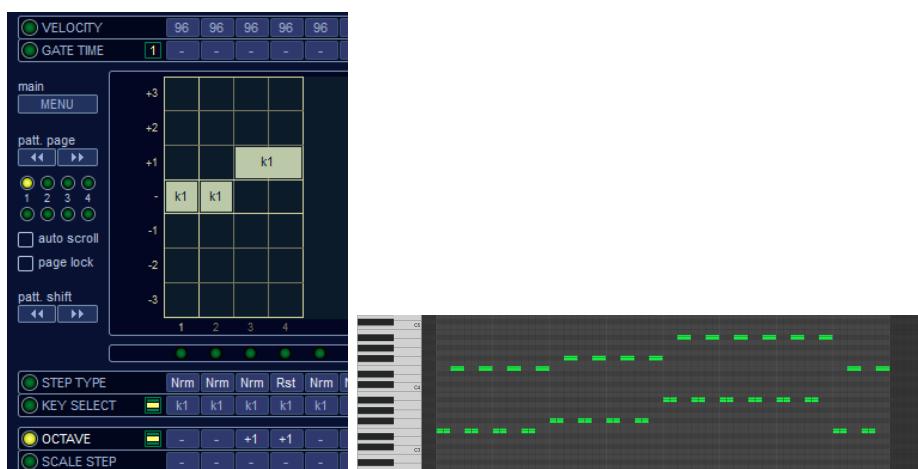


Figura 9.6: Ejemplo de configuración de *BlueAmp* y línea de bajo resultante

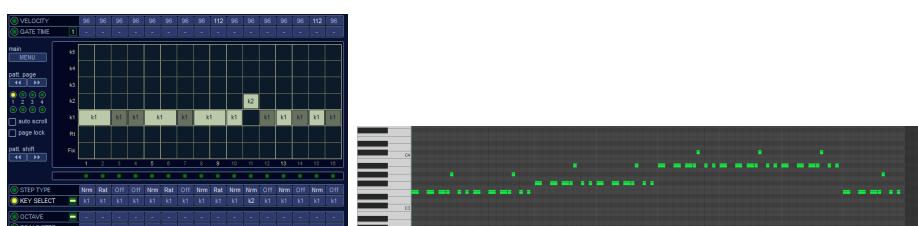


Figura 9.7: Otra configuración de *BlueAmp* y línea de bajo resultante

Como se puede ver en la Figura 9.1, se carga en Reaper una pista de bajo, la pista 6, en este caso de color verde. Esta pista carga el MIDI de armonía (Sección 5.1) pero esta vez sin inversiones (Sección 2.3.3)). Esto es porque en el bajo generalmente queremos una melodía monofónica con las notas fundamentales de cada acorde,

y necesitamos que la fundamental sea la nota más grave porque le metemos un arpegiador *BlueArp* (Sección 8.2.3) para tocar únicamente esa nota, un arpegio o una línea de bajo.

9.9. Arreglos de batería

Dependiendo del estilo de la canción (Sección 9.1), se genera cada cuatro compases de forma aleatoria un estilo de batería o género musical de entre hasta tres posibles. Se cargan alternando las tres variaciones distintas que nos proporciona el generador de percusión (Sección 7.1.2) formando un patrón calculado de forma aleatoria una única vez. Por ejemplo, si el patrón calculado es ABAC, patrón típico de batería, cargaríamos el MIDI correspondiente al ritmo A, de longitud de un compás, a continuación el ritmo B, de nuevo el A y después el C, y repetiríamos el mismo patrón ABAC durante toda la canción.

Como se puede ver en la Figura 9.1, se carga en Reaper una pista de batería, la pista 7, de color azul. Esta pista carga varios MIDIs de batería (Sección 7.1.2) dependiendo de la temática de la canción (Sección 9.1).

9.10. *Ear candy*

El *ear candy* en la producción musical consiste en añadir pequeños elementos a una canción para volverla más interesante, que no son imprescindibles para la canción, pero que captan la atención del oyente o ayudan a dar coherencia a la mezcla. Podríamos decir que el *ear candy* es la guinda del pastel en una canción.

A partir del arreglo inicial, nuestro programa sigue unas reglas para añadir varios tipos de *ear candy*: transiciones, *drum fills* y sonidos adicionales que enriquecen la mezcla final:

- **Transiciones:** cuando una sección contiene un número determinado de instrumentos sonando simultáneamente y la sección siguiente contiene al menos un instrumento más, se añade un *riser* formado a partir de un sonido aleatorio con mucha reverb que va aumentando en volumen según nos acercamos al cambio de sección. En el caso contrario, es decir, que haya varios elementos en una sección y en la siguiente al menos dos elementos menos, se agrega un *downriser*, que no es más que un sonido reverberado también pero esta vez decreciente en volumen.

Como se puede ver en la Figura 9.1, se cargan en Reaper dos pistas de *uprisers*, las pistas 8 y 9, de color amarillo. Cargan el MIDI de acompañamiento para crear transiciones usando un sonido reverberado que deja de sonar al entrar la nueva sección. Sólo puede sonar una simultáneamente. Son dos para añadir variedad.

También se cargan dos pistas de *downrisers*, las pistas 10 y 11, de color verde. Al igual que los *uprisers*, cargan el MIDI de acompañamiento pero esta vez

para generar una bajada, usando una *reverb* con menos ataque (alcanza más volumen antes) que baja de volumen progresivamente.

- **Drum fill:** (Sección 7.1.4) Cuando va a entrar un instrumento de batería en la próxima sección, y sólo si no hay un *riser* colocado para la transición, se coloca un *drum fill* que sonando sin otros instrumentos de batería, funciona como un ritmo de introducción al patrón de batería. Adicionalmente, cada cuatro compases de percusión, colocamos un *drum fill* que suena a la vez que el ritmo principal para añadirle movimiento, siendo de mayor duración en los compases múltiplos de ocho y cuanto más avanzados en la canción estemos.

Como se puede ver en la Figura 9.1, se cargan en Reaper una pista de *drum fills*, la pista 12, de color azul. Suenan de forma simultánea a las baterías o cuando va a entrar una sección con batería y no estaba sonando en la sección actual. Carga un MIDI especial que tiene marcadas las notas de bombo, caja y dos platillos, que serán interpretados por el arpegiador *BlueArp* (Sección 8.2.3) y con el humanizador (Sección 8.2.4) configurado con algo de aleatoriedad en el *pitch*, lo que en instrumentos de batería significa que sonarán distintos golpes (cajas, platillos, toms, palmas) cada vez.

- **Entrada adelantada de sonidos:** Si entre dos secciones no hay ningún elemento común, uno de los elementos adelanta su entrada para que el cambio no sea muy brusco. Al elegir instrumento para hacer esto, tiene prioridad el instrumento de bajo. Si no hay bajo presente, será la melodía o el acompañamiento el que resuelva esto en ese orden de prioridad. En la Figura 9.1 puede verse un ejemplo en la transición de la sección 6 a la 7 (compás 48), donde se adelanta la entrada de la pista 2, una pista de melodía.
- **Sonidos adicionales:** Si en una sección no hay muchos sonidos presentes, se agregarán de forma aleatoria algunos sonidos (con más o menos protagonismo en la mezcla) a lo largo de la sección. Como se puede ver en la Figura 9.1, se cargan en Reaper cuatro pistas de sonidos puntuales, las pistas de la 13 a la 16, de color rosa. Cargan el MIDI de la armonía y suenan al menos dos veces por sección durante un periodo corto de tiempo si la sección cumple los requisitos planteados en la sección de *ear candy*. Usan un arpegiador para interpretar una melodía tocando las notas presentes en el acorde y llevan un efecto de delay creado con el plugin CRM BL (Sección 8.2.5).

Capítulo 10

Conclusiones y Trabajo Futuro

Los objetivos del trabajo se han cumplido, obteniendo así una aplicación capaz de generar música simbólica y, que se comunica con Reaper para reproducirla con una selección de instrumentos acordes a unas temáticas predefinidas. A continuación desglosamos los hitos más relevantes conseguidos en cada apartado del sistema.

10.1. Generación de melodías

La generación de melodías con *machine learning* tiene la ventaja principal de poder ser automática, lo que permite crear melodías a todo tipo de usuario, sin requerir conocimientos musicales. En nuestro trabajo se exploran varios modelos, pero todos tienen algo en común, el usuario debe de ir escuchando cada melodía para comprobar si son de su agrado.

En general, los algoritmos más sofisticados de Magenta no consiguen una melodía aceptable el 100 % de las veces, pero la generación con esta metodología permite ir probando distintas melodías, lo que se ajusta perfectamente al ciclo de uso de nuestra aplicación.

Finalmente podemos destacar que los modelos más complejos, Magenta y nuestra red neuronal recurrente, generan melodías interesantes. Resulta evidente que los resultados no parecen simplemente notas aleatorias, sino que se pueden reconocer estructuras de composiciones típicas.

10.2. Armonización

La armonización por ventanas implica dividir la canción en fragmentos uniformes, en los cuales se selecciona el acorde predominante. Los pesos de los acordes en dichos fragmentos se determinan mediante una heurística rigurosa. La armonización por ventanas supuso un buen punto de partida y las bases para la evolución del algoritmo.

Aún así, esta primera versión del algoritmo tenía un problema: asume que todos los acordes de la canción tienen la misma duración. Tanto la armonización por ventanas de diferentes como la armonización por desplazamiento de ventana solucionaron este problema. Ambas otorgan a la heurística una perspectiva más amplia del contexto musical, permitiendo fragmentar los acordes en tamaños irregulares de forma coherente.

Tener en cuenta las relaciones entre acordes fue lo más difícil de solucionar. Con un primer intento fallido, con la matriz de correspondencia entre acordes, se consiguió resolver el problema desde una perspectiva distinta: el reconocimiento de progresiones de acordes. El usuario define las progresiones de acordes que quiere que se detecten y el algoritmo maximiza el sumatorio de los pesos de las ventanas atendiendo a las restricciones impuestas. Esta solución permite tener más control y personalización de la salida esperada, consiguiéndose así un resultado que resuelve el problema planteado con creces.

El algoritmo de armonización ha ido evolucionando, introduciendo las mejoras necesarias para satisfacer las problemáticas y necesidades que un compositor puede llegar a tener a la hora de acompañar una melodía, consiguiéndose finalmente superar incluso las expectativas iniciales.

10.3. Arreglos e instrumentalización

Hemos logrado proporcionar una interfaz sencilla y versátil para el usuario final a la hora de conectar con Reaper, ya que hemos logrado automatizar el mayor número de tareas posibles.

En términos de sonoridad, hemos logrado resultados más que aceptables teniendo en cuenta la poca intervención humana requerida. Las canciones resultantes son bastante variadas y el amplio abanico de timbres disponibles permiten cubrir varios géneros musicales y temáticas de videojuegos. Además, si el usuario no está del todo convencido con una generación, puede hacer uso de las temáticas y las semillas para cambiar alguna parte de la canción. Por ejemplo puede cambiar el arreglo manteniendo los instrumentos o cambiar la temática pero mantener la estructura de la canción.

Destacar que la herramienta está diseñada con la idea de generar música para videojuegos. Esto es muy útil si el usuario final quiere, por ejemplo, usar una misma canción para distintas zonas de su videojuego, simplemente cambiando la temática, la reverberación o el *pitch*. Otro ejemplo de uso puede ser música del estilo de los niveles de Super Mario, donde se puede usar una misma idea musical (las melodías generadas por la herramienta) para crear canciones de varios niveles, como un nivel de agua, un nivel de desierto, de selva, etc. (cambiando las temáticas u otros parámetros pero sin cambiar la melodía generada).

Todas las canciones generadas son *loopables*, al igual que las distintas secciones dentro de una propia canción, pues las canciones generadas se basan en la repetición de una serie de patrones que no generan disonancias al reproducirse en bucle por

su ritmo constante, transiciones y continuidad melódica y armónica. Debido a esto, dichas canciones son ideales para ser usadas en un videojuego, donde incluso puedes usar las distintas secciones y capas para montar fácilmente un proyecto de música adaptativa en una herramienta externa (por ejemplo, usando FMOD).

10.4. Aplicación

TKInter ha demostrado ser una biblioteca muy útil para el desarrollo de aplicaciones de prototipado rápido, por lo que ha cumplido con nuestras expectativas.

Con TKInter hemos podido desarrollar una aplicación acorde con la idea inicial, ofrecer una interfaz sencilla que permita a los desarrolladores crear piezas musicales para sus videojuegos, sin necesidad de conocimientos musicales previos.

Hoy en día el desarrollo de aplicaciones conlleva más infraestructura y servicios que el alcance que tiene Vanguard Music, como el inicio de sesión o base de datos en la nube, por mencionar un par. Somos conscientes de la extensibilidad de nuestra aplicación, por lo que lo hemos recogido en la sección de trabajo futuro.

10.5. Conclusiones generales

La aplicación *Vanguard Music* cumple los objetivos que nos propusimos al inicio del trabajo. Permite generar música con distintas temáticas manteniendo la idea musical original, además, dentro de esta se generan 8 secciones que funcionan independientemente y se pueden combinar con cualquier número del resto de secciones y, *loopan* entre sí perfectamente.

El trabajo ha requerido combinar campos como el *machine learning*, teoría y composición musical, producción musical, desarrollo de aplicaciones en Python y diseño de arquitecturas modulares. Esto nos ha permitido salir de nuestros campos de conocimiento, específicamente el desarrollo de videojuegos, e investigar diversas áreas para la construcción de una herramienta funcional.

10.6. Trabajo futuro

Aunque hemos conseguido buenos resultados en base a los objetivos que nos propusimos, existen diversas áreas en las que se puede expandir y mejorar el proyecto. A continuación se exponen algunas mejoras que se proponen para cada apartado de este.

La parte de generación de melodías es altamente extensible. En primer lugar, al tener una aplicación modular que permite utilizar distintos módulos de generación, es posible desarrollar módulos para cualquier modelo de generación simbólica, por lo que se puede experimentar con modelos radicalmente distintos a los actuales para comparar resultados. Resulta especialmente interesante explorar modelos como

KNN (*K Nearest Neighbours*) o modelos que simulen las estrategias comunes de composición. Además, otra ruta de mejora es solucionar la dependencia de internet de Magenta, utilizando modelos preentrenados de dicha herramienta que se incluyan con la aplicación, sin embargo, esto no resulta sencillo con su API en JavaScript.

En cuanto al armonizador, parte del trabajo futuro ya ha sido esbozado en las secciones que lo explican. Por ejemplo, una forma sencilla de mejorar el algoritmo sería mejorando la heurística inicial. En la Sección 5.2.1 se habla de tener en cuenta también la velocidad de la nota para calcular los pesos, pero se podrían tener en cuenta otros factores u otra metodología para llevar a cabo la heurística inicial. Las partes que más rango de mejora tendrían serían las de la matriz de correspondencia y el modelo de predicción de acordes (Secciones 5.4.1 y 5.4.2 respectivamente) que pasaron a un segundo plano en nuestra implementación por falta de tiempo. Si los resultados fueran buenos, supondrían una mejora respecto al reconocimiento de progresiones de acordes ya que el coste del algoritmo volvería a ser lineal. Los resultados obtenidos representarían una especie de equilibrio entre aquello que compone comúnmente la gente (entrenamiento de los modelos) y las peculiaridades que pueda generar la heurística a la hora de calcular los pesos de los acordes. Más centrado en la parte del reconocimiento de progresiones de acordes, se podrían implementar las variantes de las políticas de elección de progresiones de las cuales se habla en la Sección 5.5.1. Además, se podría mejorar la poda de los algoritmos ya existentes para mejorar sus rendimientos. Por último, y relacionado con todo el módulo en general, sería ideal implementar una interfaz accesible desde la aplicación para configurar los parámetros relacionados con el armonizador, ya que ahora mismo solamente se pueden modificar a nivel de código. Estos parámetros abarcarían desde la configuración de las ventanas de las primeras heurísticas, hasta la elección de las progresiones de acordes que se quieran detectar y sus transiciones.

En cuanto el apartado de la búsqueda de nuevos colores existen muchas cosas que modificar y ampliar. La primera sería dividir el concepto de melodía y armonía base en mayor y menor, con el objetivo de tener un color más a la hora de generar temáticas. En vez de elegir la configuración con más peso, que es lo que está implementado actualmente, utilizar dos soluciones, una armonizando exclusivamente con progresiones mayores y otra con progresiones menores. Esto tendría más sentido y aportaría mayor riqueza a las temáticas. Así mismo, utilizar este mismo par de soluciones para realizar la modificación a los modos mayores y menores correspondientes también supondría una metodología más coherente que lo implementado en este producto final. Relacionado también con los modos, sería muy interesante y supondría una mejora atacar la armonía modal desde otra perspectiva. Ahora mismo se ajusta la melodía base a la escala del modo y se armoniza únicamente con los acordes que pertenecen a dicho modo. Es decir, no se sale del modo elegido. A la hora de componer con los modos griegos esto no es lo que se suele hacer realmente. Es mucho más común continuar en una escala diatónica (mayor o menor) y utilizar acordes de intercambio con el modo escogido para realizar la composición. Este es el cambio de perspectiva que estaría interesante explorar en un trabajo futuro. Lo último por mencionar sería el uso de las numerosas escalas adicionales que existen y que vagamente se han mencionado en este TFG para crear nuevos colores. Algunos

ejemplos rápidos serían la doble armónica, con una sonoridad muy característica que mejoraría la temática de desierto actual o la hexatónica, que encajaría con temáticas más misteriosas. Estas y muchas más escalas podrían ser utilizadas para mejorar las temáticas existentes y crear otras nuevas.

Por parte de la sonorización de la canción, sería interesante tratar de condensar todos los plugins necesarios en un número más reducido de estos. Un posible camino a seguir sería el uso de alguna suite de plugins (de pago o gratuita) como podría ser Kontakt de Native Access. Otra opción sería el uso de algún plugin de sintetizador mucho más completo que los que se utiliza, del estilo de Vital o de Serum, pudiendo de esta forma utilizar ese plugin para generar la mayoría de los timbres. Además de eso, actualmente es una herramienta bastante cerrada, que no permite ser extendida por los usuarios. Esto es algo que podría cambiar mediante la implementación de algún sistema de presets descargables (por ejemplo de manera similar a cómo lo hacen Vital o Serum), quizás de temáticas nuevas, soporte para plugins que los usuarios elijan, etc.

La aplicación tiene aún mucha extensibilidad, la cual se escapa al alcance de este TFG. Actualmente, la mayoría de aplicaciones cuentan con registro de usuario y su correspondiente inicio de sesión, que permite distinguir a los usuarios. Combinado con el guardado de datos en la nube y una base de datos en línea, permite operar desde dispositivos distintos, recuperando la información de una misma cuenta y asegurando los datos en caso de desinstalar la aplicación. En el caso de Vanguard Music serviría para no perder los presets de la aplicación que defina el usuario.

Introduction

Introduction to the subject area. This chapter contains the translation of Chapter 1.

Motivation

Music has always been a powerful means of expression, capable of conveying a wide range of emotions. In fields such as cinema and video games, it is crucial, as it can evoke strong feelings in the audience, immersing them in fictional worlds. Any video game developer who wants to captivate the player needs to rely on music and sound effects.

Today, there are many ways to approach music composition and production, but the most common is the use of a DAW (Digital Audio Workstation). A DAW is a software tool that allows recording, editing, and mixing audio tracks, as well as sequencing music. Sequencing music involves programming and playing musical events, with the most widespread format for these events being MIDI. MIDI is a standard that defines basic note-on and note-off events, as well as others that contain more specific information about the musical piece. These types of events can be interpreted by plugins. Specifically, the plugins that interpret MIDI are called virtual instruments. Virtual instruments can emulate the timbre of real instruments, either by using recorded samples or digital audio synthesis, to generate sound in real-time.

The great advantage of MIDI is the ability to modify and edit a song easily, without having to re-record the performance of the piece. This is because the musical notation is separated from the instruments. A MIDI file represents a digital score. It is, in essence, a symbolic musical notation of the music.

The other alternative, where there is no separation between musical notation and instruments, involves working directly with digital audio. In this approach, musicians record live performances of instruments and vocals, capturing the sound waves generated by them. The recordings are edited and mixed to create the final track. This method allows capturing the unique performance and timbre of each musician and instrument, providing an authentic and natural character to the music,

but it is very limited when making changes to the composition once the recordings are done.

In the field of video games, traditional music composition does not quite fit, as a song is traditionally composed to be listened to from beginning to end as a complete experience. In video games, we do not know in advance how long a player will stay in an area listening to a song, so it is crucial that it does not become repetitive and cycles properly. But the problems do not end there; in video games, unlike cinema, we cannot adjust which song will play at each minute, as it is a medium with a higher degree of freedom. For these reasons, adaptive music emerges. Adaptive music is a style of music specifically composed to change based on a series of parameters.

When composing adaptive music, we can work with horizontal or vertical composition. Horizontal composition involves using temporal jumps in a song already divided into sections that can be cycled. These jumps are defined for sections with different intensity or instruments, to better adapt to a specific area or moment. In vertical composition, several tracks or layers are created that can be activated or deactivated to generate different sections. Although this largely depends on the musical style, it is common to find the following layers: melody, harmony, bass, and rhythmic base, among others. The melody is the leading and most distinctive element, usually a voice or instrument with defined patterns that leads the song. The harmony accompanies and complements the melody, usually appearing as chords played by one or more instruments secondary to the melody. The bass creates a foundation over which the melody and harmony stand out, generating the lowest frequencies of the song. Finally, the rhythmic base, as its name suggests, defines the rhythm of the song that the rest of the elements will follow. By combining all these layers, we achieve a complete musical composition, and in adaptive music, we activate or deactivate them to generate variety and make the music adjust to what is happening in the video game, introducing more or less intensity, for example. This is just a typical example; adaptive music can consist of many more layers and different elements that allow a wide range of combinations. In short, it is a very rich field open to imagination.

This work aims to develop a tool that automatically generates music of various themes for use in video games.

In recent years, there have been many advances in automatic music generation with AI and machine learning techniques. There are two main ways to generate music: audio generation and symbolic level generation.

Audio generation produces the actual musical piece in formats like MP3 or WAV, usually using deep learning or attention techniques. This type of generation is not explored in this work, but it is a valid methodology that each year continues to demonstrate its potential.

At the symbolic level, generation produces the notes that make up the song in MIDI format or similar, so it can be rendered using virtual instruments and a DAW, as mentioned earlier. As we have discussed, symbolic generation, unlike audio generation, allows adjustments to be made to the song and parts or instruments to be varied. This is the approach we have selected for our work.

Objectives

As mentioned, the main objective of this work is to create a tool that automatically generates music for video games. The tool will have different modules that generate various musical layers, including melody, harmony, bass, and rhythmic base. This generation is done at a symbolic level, and the result is generated in a DAW project with predefined timbres. This approach allows for great flexibility in terms of editing by the user, both musically, by editing the notes, and in terms of timbres, by changing timbres and virtual instruments in the DAW.

Specifically, the following specific objectives are proposed:

1. Achieve melody generation through different means. On one hand, we will investigate the use of existing tools, such as Google's Magenta. On the other hand, we will explore the development of our own machine learning models for this purpose.
2. Achieve harmony for the generated melody so that it is pleasing to the human ear, either through the use of artificial intelligence or our own heuristics.
3. Using the generated melodies and harmonies, create the complete arrangement of a song, following a structure that facilitates its use as video game music, whether in an adaptive or linear form.
4. Sonorize the generated musical notes by making an appropriate selection of virtual instruments based on predefined themes.
5. Create a desktop application for Windows that integrates the aforementioned possibilities in a user-friendly manner.
6. Connect the developed application with Reaper, adding the necessary instruments and producing the mix based on the theme and modifiers selected in our application.

Work Plan

Having clearly separated objectives in terms of functionality allows us to efficiently separate the work into independent modules, like a production chain divided into stages that come together in the final application.

1. Research and study the field of artificial intelligence applied to music generation. During this phase, we will review existing literature, investigate the most relevant approaches and algorithms, and familiarize ourselves with available tools and resources.
2. Develop the melody generation algorithm. We will use machine learning techniques and generative models to create systems capable of composing original and varied melodies. Additionally, we will explore the existing Magenta tool.

3. Implementation of an algorithm that harmonizes generated melodies. At this stage, we will design and develop heuristics based on music theory in order to generate the chords that accompany the melody.
4. Study, understand, and apply the ReaScript API to program in Reaper. We will use ReaScript with Python to load all the generated MIDI files into Reaper, arrange them according to an arrangement to create a complete song, and load the plugins that will make those MIDI files sound.
5. Selection and configuration of instruments. We will investigate and select a variety of virtual musical instruments that fit the predefined themes of the video games. We will configure these instruments to convincingly reproduce the generated melodies to the human ear.
6. Develop the desktop application. We will use appropriate technologies to develop an intuitive and user-friendly interface that allows users to generate, harmonize, and sonorize their own musical compositions for video games. Specifically, we will use Tkinter, Python's default library for creating graphical user interfaces (GUIs).

Structure of the Document

Chapter 2 will present the preliminary concepts that need to be understood for further sections. This expands on some concepts presented in this chapter.

Chapter 3 consists of a user guide for the tool, explaining all aspects from installation and configuration to its use for generating songs.

Chapter 4 describes the models used to generate melodies, comparing their results and detailing their design and implementation. It also includes the integration of Magenta into our application.

Chapter 5 delves into the harmonization of a melody, applying concepts of musical theory in our own algorithms.

In Chapter 6, the melodies are modified and various harmonization techniques are used to obtain new colors from the same generation.

Chapter 7 details the generation of the percussion base used in song composition.

Chapter 8 explains how the sounds and virtual instruments applied to the different generated parts are chosen.

Chapter 9 delves into the arrangement of songs for various themes, describing each part of the song production process.

Finally, Chapter 10 contains brief conclusions on the results obtained with the work, as well as a series of improvements and future work to be done.

Conclusions and Future Work

The objectives of this work have been met, resulting in an application capable of generating symbolic music and communicating with Reaper to play it using a selection of instruments aligned with predefined themes. Below, we detail the most relevant achievements for each part of the system.

Melody Generation

The main advantage of melody generation with machine learning is its automation, allowing users without musical knowledge to create melodies. Our work explores several models, but they all share one thing in common: the user must listen to each melody to determine if it is satisfactory.

In general, even the most sophisticated algorithms from Magenta do not produce an acceptable melody 100% of the time. However, this methodology allows users to try different melodies, which fits perfectly within the usage cycle of our application.

Finally, it is worth noting that the more complex models, such as Magenta and our recurrent neural network, generate interesting melodies. It is evident that the results are not just random notes; recognizable structures of typical compositions can be identified.

Harmonization

Window-based harmonization involves dividing the song into uniform fragments, in which the predominant chord is selected. The weights of the chords in these fragments are determined using a rigorous heuristic. Window-based harmonization was a good starting point and laid the foundation for the evolution of the algorithm.

However, this first version of the algorithm had a problem: it assumed that all chords in the song had the same duration. Both window-based harmonization and sliding window harmonization solved this problem. Both approaches provided the heuristic with a broader perspective of the musical context, allowing chords to be fragmented into irregular sizes coherently.

Accounting for the relationships between chords was the most difficult problem to solve. An initial failed attempt using the chord correspondence matrix led us to solve the problem from a different perspective: chord progression recognition. The user defines the chord progressions they want to detect, and the algorithm maximizes the sum of the window weights based on the imposed constraints. This solution allows for more control and customization of the expected output, achieving a result that satisfactorily resolves the posed problem.

The harmonization algorithm has evolved, introducing necessary improvements to meet the problems and needs that a composer may have when accompanying a melody, ultimately exceeding the initial expectations.

Arrangements and Instrumentation

We have provided a simple and versatile interface for the end user to connect with Reaper, automating as many tasks as possible.

In terms of sound, we have achieved more than acceptable results considering the minimal human intervention required. The resulting songs are quite varied, and the wide range of available timbres allows for covering various musical genres and video game themes. Additionally, if the user is not entirely satisfied with a generation, they can use themes and seeds to change parts of the song. For example, they can change the arrangement while maintaining the instruments or change the theme while keeping the song's structure.

It is noteworthy that the tool is designed to generate music for video games. This is very useful if the end user wants to use the same song for different zones of their game, simply changing the theme, reverb, or pitch. Another example of use could be music similar to Super Mario levels, where the same musical idea (the melodies generated by the tool) can be used to create songs for various levels, such as a water level, a desert level, a jungle level, etc. (changing the themes or other parameters without changing the generated melody).

All generated songs are loopable, as are the different sections within a song. The generated songs are based on the repetition of a series of patterns that do not generate dissonances when played in a loop due to their constant rhythm, transitions, and melodic and harmonic continuity. Therefore, these songs are ideal for use in a video game, where the different sections and layers can be easily used to create an adaptive music project in an external tool (e.g. using FMOD).

Application

TKInter has proven to be a very useful library for rapid prototyping application development, meeting our expectations.

With TKInter, we have been able to develop an application in line with the initial idea: offering a simple interface that allows developers to create musical pieces for

their video games without requiring prior musical knowledge.

Today, application development involves more infrastructure and services than Vanguard Music's current scope, such as login or cloud database features. We are aware of our application's extensibility, which is addressed in the future work section.

General Conclusions

The Vanguard Music application meets the objectives we set at the beginning of the project. It allows generating music with different themes while maintaining the original musical idea. Additionally, within this application, eight sections are generated that function independently and can be combined with any number of other sections, looping perfectly.

The work has required combining fields such as machine learning, music theory and composition, music production, Python application development, and modular architecture design. This has allowed us to step out of our knowledge areas, specifically game development, and explore various areas to build a functional tool.

Future Work

Although we have achieved good results based on our initial objectives, there are various areas where the project can be expanded and improved. Below are some proposed improvements for each part of the project.

The melody generation part is highly extensible. Firstly, having a modular application that allows using different generation modules makes it possible to develop modules for any symbolic generation model, allowing for experimentation with radically different models to compare results. It is particularly interesting to explore models like KNN (K Nearest Neighbors) or models that simulate common composition strategies. Additionally, another improvement route is to eliminate Magenta's internet dependency by using pre-trained models included with the application, though this is not straightforward with its JavaScript API.

As for the harmonizer, part of the future work has already been outlined in the sections explaining it. For example, a simple way to improve the algorithm would be to enhance the initial heuristic. In Section 5.2.1, considering the note's velocity for calculating weights is mentioned, but other factors or methodologies could also be considered for the initial heuristic. The parts with the most room for improvement would be the chord correspondence matrix and the chord prediction model (Sections 5.4.1 and 5.4.2, respectively), which were deprioritized in our implementation due to time constraints. If the results were good, they would represent an improvement over chord progression recognition since the algorithm's cost would return to linear. The results would represent a balance between commonly composed elements (training models) and the peculiarities that the heuristic might generate when calculating chord weights. More focused on chord progression recognition, im-

plementing variations of the progression selection policies discussed in Section 5.5.1 could be considered. Additionally, improving the pruning of existing algorithms to enhance their performance would be beneficial. Finally, related to the entire module in general, implementing an accessible interface from the application to configure harmonizer-related parameters would be ideal, as these parameters can currently only be modified at the code level. These parameters would range from configuring the initial heuristics' windows to choosing the chord progressions to be detected and their transitions.

Regarding the search for new colors, there are many things to modify and expand. The first would be to divide the concept of base melody and harmony into major and minor, to have an additional color when generating themes. Instead of choosing the configuration with the most weight, which is currently implemented, using two solutions—one harmonizing exclusively with major progressions and another with minor progressions—would provide more richness to the themes. Similarly, using this same pair of solutions to make modifications to the corresponding major and minor modes would also be a more coherent methodology than what is currently implemented. Related to modes, it would be very interesting to approach modal harmony from a different perspective. Currently, the base melody is adjusted to the mode's scale, and only chords belonging to that mode are used for harmonization. This is not how composing with Greek modes is typically done. It is much more common to continue in a diatonic scale (major or minor) and use borrowed chords from the chosen mode for composition. This perspective shift would be interesting to explore in future work. Lastly, the use of numerous additional scales mentioned briefly in this work for creating new colors would be noteworthy. Examples include the double harmonic scale, with a very characteristic sound that would enhance the current desert theme, or the hexatonic scale, which would fit more mysterious themes. These and many more scales could be used to improve existing themes and create new ones.

In terms of song sound, it would be interesting to try to condense all necessary plugins into a smaller number of these. One possible path would be to use a plugin suite (paid or free) like Kontakt from Native Instruments. Another option would be to use a much more complete synthesizer plugin than those currently used, such as Vital or Serum, allowing that plugin to generate most of the timbres. Additionally, the tool is quite closed, not allowing for user extension. This could change by implementing a downloadable presets system (similar to how Vital or Serum do), perhaps for new themes, plugin support that users choose, etc.

The application still has a lot of extensibility, which goes beyond the scope of this project. Currently, most applications include user registration and login, allowing for user distinction. Combined with cloud data storage and an online database, this enables operation from different devices, retrieving information from the same account and ensuring data security in case the application is uninstalled. For Vanguard Music, this would help preserve user-defined presets.

Contribuciones Personales

Rodrigo Sánchez Torres

Procesamiento de datos

Para poder utilizar algoritmos de inteligencia artificial en nuestro trabajo era necesario tener un dataset limpio y adecuado para nuestros objetivos.

La primera tarea fue buscar un dataset que contenga información sobre melodías. En la búsqueda se valoraron muchos tipos de datasets, finalmente, en la exploración de los datasets de Magenta se decidió utilizar el *Bach Doodle Dataset*.

Una vez elegido el dataset se procedió al tratamiento de este, con distintos acercamientos se fueron realizando limpiezas en principio superficiales, pero pronto se decidió que lo mejor era tener un dataset con solamente melodías que estén en la misma escala e incluso en un rango reducido de octavas.

También se exploraron las opciones de usar o no silencios, siendo en algoritmos como cadenas de Markov mejor no emplear silencios, pero en redes neuronales recurrentes se obtuvieron mejores resultados con silencios en el dataset.

Finalmente se testeó tanto a usar tiempos absolutos en segundos como a normalizar el tiempo utilizando *steps* y BPM (*beats per minute*), dando mejor resultado tener el tiempo normalizado y homogéneo en el dataset.

Algoritmos de *Machine Learning*

Antes de poder elegir qué modelos serían más adecuados para la generación de melodías fue necesario realizar una amplia investigación sobre el tema, para poder conocer la mayor cantidad de técnicas y modelos que podemos aplicar a nuestro trabajo.

Tras la investigación quedó claro que era necesario buscar un modelo que no fuera determinista, para poder generar melodías diferentes en cada ejecución. Es por esto que se decidió explorar los 3 modelos actuales: cadenas de Markov, redes neuronales recurrentes con capa *softmax* y modelos de Magenta.

En primer lugar se empezó a explorar la generación con Markov, para esto se realizó una investigación sobre el tema y sus posibles aplicaciones. Para poder utilizar correctamente las cadenas se utilizó una notación común a lo largo del proyecto en los algoritmos propios de generación, como era "*pitch_duration*".

Para poder utilizar cadenas de Markov se procesó manualmente el dataset, creando matrices de transición que sean interpretables por la librería utilizada. Finalmente, se consiguió tener un primer modelo de generación listo, que permitió continuar el desarrollo de otras ramas que se apoyan en éste.

Una vez se tuvo el primer modelo de generación listo se podía dedicar más tiempo a desarrollar un modelo más complejo. Se decidió utilizar entonces una red neuronal recurrente. Pero esta tenía el problema de ser un modelo determinista, así que se añadió finalmente una capa *softmax*, que permitía introducir una aleatoriedad a la generación controlada con un parámetro de temperatura.

Este modelo fue el más complejo de desarrollar, pues había que realizar mucho trabajo de ajuste de *súperparámetros* y diseño de la propia red. Además, se realizaron varias modificaciones al dataset, que finalmente no aportaban un mejor resultado. Al final tras un largo período de prueba se decidió adoptar el diseño actual, teniendo una entrada de número reales, con el *pitch*, duración y tiempo de inicio de la nota actual, y una salida con valores de tipo *label*, siendo estas las posibles notas que podía elegir la red. Al utilizar salida con valores de tipo *label* codificados mediante *one-hot encoding* se podía utilizar la capa *softmax* mencionada anteriormente.

La red conseguía finalmente un rendimiento aceptable, por lo que se realizaron pruebas para entrenarla de forma que indicara también el tiempo de inicio de cada nota, lo que podía generar melodías más interesantes sobre el papel. Sin embargo, la red no conseguía aprender patrones significativos sobre este, obteniendo una precisión tan baja que no merecía la pena utilizar dicho parámetro. Es por esto que se decidió introducir los silencios como notas adicionales.

Este modelo consiguió una generación mucho mejor a las cadenas de Markov, manteniendo patrones más típicos en la composición pero con limitaciones. Aún así se decidió mantener el modelo con Markov para tener una generación más experimental.

Intercalado con el desarrollo de la red neuronal recurrente se integró Magenta al proyecto, con el fin de proporcionar una generación más elaborada que la de Markov para poder trabajar en otras ramas y producir mejores resultados.

La integración de Magenta supuso una gran cantidad de problemas, pues principalmente no se podía utilizar el paquete de Python en Windows. Tras realizar varias pruebas con binarios precompilados para Windows, migraciones a Ubuntu y diversos acercamientos se decidió que no era viable utilizar el paquete de Python, por lo que se exploró el uso del paquete de JavaScript.

El principal conflicto era que el proyecto estaba íntegramente desarrollado en Python, por lo que el módulo de JavaScript tenía que estar separado de éste. Finalmente se consiguió comunicar el proyecto con un script (*magentaPython.py*) que actúa de puente entre Python y JavaScript. Este script utiliza el módulo *subprocess*

para lanzar un proceso de JavaScript, con el que nos podemos comunicar por medio de argumentos de dicho proceso y la salida estándar de éste. Con esta metodología podemos mandar melodías serializadas en JSON al proceso y recibir resultados de generación recogiendo la salida estándar con el mismo formato JSON.

Este acercamiento permite utilizar Magenta en la aplicación, pero conlleva otros problemas de dependencia de paquetes y sobre todo de una conexión a internet para utilizar los modelos alojados en la base de datos de Google.

Aportaciones a la aplicación

Además, se han realizado aportaciones al código de la aplicación. Se han introducido los módulos de generación a ésta, así como la posibilidad de elegir uno u otro desde código. También se han añadido las funcionalidades de carga de melodías existentes y guardado de una melodía generada, pudiendo elegir el nombre y ubicación del archivo.

Víctor Manuel Estremera Herranz

Arquitectura base

Antes de comenzar a realizar el algoritmo de armonización, el primer paso consistió en implementar una serie de clases y métodos que me permitiesen hablar en el lenguaje de la música. Es decir, llevar todo lo explicado en el apartado de principios de la armonía a código (Sección 2.3). Se empezó poco a poco y la arquitectura fue escalando. Primero se representó al intervalo (Sección 2.3.1). Un conjunto de intervalos formaron una escala, que a la vez representa un acorde (Secciones 2.3.2 y 2.3.3). Se idearon métodos con los que calcular la armonía de la escala y sus modos (Secciones 2.3.3 y 2.3.4). Posteriormente, se incorporaron clases auxiliares para representar una nota musical o un compás.

El siguiente paso consintió en gestionar la entrada y salida de archivos MIDI. Para operar cómodamente con las canciones era necesario idear a una representación propia con la que poder leer las melodías y escribir acordes. Esta tarea se me alargó más de lo que tenía previsto en un primer momento debido a los diversos mensajes MIDI que existen y la forma en la que estos se pueden encontrar. Muchos de ellos no son necesarios para la tarea del armonizador, pero otros rompían de nuevo la última traducción implementada. Finalmente, tras explorar la combinatoria de mensajes a base de probar MIDIs sacados de multitud de sitios, se terminó de arreglar la traducción. Las representaciones de una canción utilizadas en este módulo utilizan el tick como unidad simbólica de tiempo. Si se desea profundizar más en estas representaciones, se puede consultar el Apéndice B.

Algoritmo de armonización

Una vez establecidas unas bases sólidas, era momento de implementar la heurística. La idea de la heurística inicial surge de mis conocimientos previos en la música y de todo lo asimilado a lo largo de los años dedicados a esta disciplina. Todo lo aprendido, tanto de manera implícita como explícita, al interpretar un instrumento, me permitió valorar correctamente la importancia de cada nota en una pieza musical y, por ende, de los acordes que las acompañan.

Las buenas bases de la heurística permitieron una evolución cómoda del algoritmo. El siguiente paso a resolver fue la uniformidad de los tamaños de los acordes. Con sendas mejoras descritas en la Sección 5.3 la armonía se fragmentó y los resultados obtenidos mejoraron considerablemente. En su momento, se puso en práctica esta nueva versión del algoritmo no solo con obras generadas por IA, sino con composiciones clásicas ya existentes. Los resultados fueron muy satisfactorios, sobre todo con estos últimos. El algoritmo era capaz generar acompañamientos muy interesantes. Además, al cambiar las configuraciones de ventana, dichos acompañamientos variaban. Se creaban nuevas perspectivas armónicas no evidentes a simple vista que seguían sonando bien. Esto hace referencia a una de las primeras afirmaciones descritas en esta módulo: una misma melodía puede ser acompañada por diversas armonías, la mejor depende de la subjetividad del oyente.

Tener en cuenta el contexto armónico en el que se mueven los acordes fue lo más complicado. El intento fallido de la matriz de correspondencia y el hecho de que la anterior versión del algoritmo ya fuera considerablemente buena, me hicieron desistir un poco a la hora de ponerle la guinda al pastel. Sin embargo, en los últimos meses de desarrollo se me ocurrió atacar el problema desde una perspectiva diferente con el reconocimientos de unos patrones de acordes definidos previamente por el usuario. Para facilitarme un poco la vida creé un primer prototipo en la que se recorría en anchura las ventanas de los pesos de los acordes de forma que la última progresión pudiese quedar partida por la mitad. Esta versión base evolucionó rápidamente a la utilizada en la aplicación final: ahora el algoritmo no solo maximiza el peso de los acordes teniendo en cuenta las restricciones establecidas, sino que también considera que la última progresión forme un bucle coherente con el primer acorde y, por ende, tampoco corte la última progresión.

Modificaciones de la melodía original

Aprovechando toda la arquitectura creada en este módulo, también me he encargado del procesamiento de las melodías para su utilización en los arreglos. Estas modificaciones consisten desde su troceo y combinación, hasta su trasposición y normalización con el objetivo de ser utilizados en los arreglos. Debido a las facilidades que ofrece el módulo su implementación ha sido sencilla.

Búsqueda y creación de cromatismos musicales

También me he encargado de todo lo relacionado con la creación de nuevos colores a partir de una misma entrada con el objetivo de ser utilizados por las distintas temáticas, es decir, todo lo relacionado con el Capítulo 6. En resumen, no solo me he dedicado a la búsqueda de progresiones de acordes típicas de las tonalidades mayor y menor, sino a la transformación del producto base para crear composiciones modales.

Otras tareas colaborativas

Me he involucrado en la limpieza y normalización del *Bach Doodle Dataset* con el propósito de mejorar las salidas generadas por el módulo de generación de melodías a través de inteligencia artificial.

Por último, he implementado parte del primer algoritmo de generación de líneas de bajo atendiendo a las indicaciones de mi compañero. Aunque finalmente, dichos patrones no se utilizan en la aplicación final.

Javier Callejo Herrero

Generación de percusiones.

Lo primero que se hizo fue investigar los diferentes modelos de Magenta, tanto para tomar ideas como para decidir si podíamos usar dichos modelos en nuestra herramienta.

Una vez vistas las herramientas disponibles, se decidió que los resultados obtenidos no ofrecían la variedad que requería la herramienta, por lo que se implementó un generador de percusión propio. Este generador permite crear patrones de batería de varios géneros distintos (como el jazz, el rock, la música disco, etc.) Estos patrones generados nos servirán para las siguientes partes del trabajo, en concreto la del diseño de temáticas.

Generación de líneas de bajo.

Tras haber implementado el generador de percusión, se implementó una primera versión de un generador de líneas de bajo, que cogía un ítem MIDI con acordes y usaba las notas más graves para crear de forma pseudoaleatoria una melodía monofónica que si sirviera como línea de bajo. Más adelante se descartó el uso de este generador al descubrir formas mejores de afrontar este programa (el uso de un plugin arpegiador).

Selección de plugins.

De forma paralela a los apartados anteriores, se realiza una investigación exhaustiva de plugins (de instrumentos virtuales principalmente). Continuamente nos encontramos con problemas de almacenamiento e instalación muy tediosa.

Es más adelante cuando se reúnen finalmente todos los plugins de instrumentos virtuales que se usarán en la herramienta, plugins ligeros, de fácil instalación, pues son colocados los .vst o .dll en una única carpeta comprimida para facilitar la instalación tanto al usuario final como a los propios compañeros del trabajo.

Mezcla.

Se investiga sobre algún plugin o flujo de trabajo que permita algún tipo de mezcla automática o al menos regulación automática de volumen, ya que los sonidos que se usan son aleatorios y a veces no están bien balanceadas las pistas. Las distintas opciones disponibles son en su mayoría de pago, por lo que no son óptimas para su uso en este trabajo. Se decide, por tanto, nivelar el nivel de las pistas usando plugins de stock de Reaper (compresión, limitación y ecualización).

Ecualización.

Se investiga sobre los efectos de la ecualización y la importancia de cada región de frecuencias en distintos tipos de instrumentos. A continuación se incluyen en cada pista y en la mezcla final ecualizadores con configuraciones específicas para melodías, baterías, bajos, etc.

Programar en Reaper.

Se investiga el uso de ReaScript con Python para comenzar a automatizar tareas en Reaper. Primeramente se logran cargar instrumentos virtuales e ítems MIDI llamando a un script desde una acción de Reaper.

A continuación, se diseña y se implementa la carga de plugins que cubren las necesidades concretas de cada pista. (Se continúa buscando plugins que cubran nuestras necesidades, en particular plugins de efectos).

Desde ahí, se comienza a desarrollar una arquitectura que permite crear un arreglo para la primera temática (pradera). En este punto tenemos un script que carga ítems MIDI de melodía, armonía, batería y bajo. Además, añade de forma pseudo-aleatoria plugins a las pistas y al máster: tenemos las primeras canciones generadas.

En este momento se empiezan a diseñar las temáticas. Se habla de ello en el siguiente apartado.

Se continúa hasta tener todas las temáticas programadas, añadiendo progresivamente mejoras y funcionalidades nuevas, como el subir o bajar semitonos, añadir

diversos efectos a la mezcla o, finalmente, cambiar la complejidad de la melodía cargada.

Cabe destacar que existe una investigación sobre conectar scripts externos con Reaper, pero no se logra dicho objetivo y pasa a un segundo plano hasta más avanzado el desarrollo (Sección de Miguel Comunicación entre Vanguard Music y Reaper).

Diseño e implementación de las temáticas.

Se investigan los elementos tanto técnicos como creativos necesarios para crear las distintas temáticas para las canciones. A continuación, se diseñan e implementan en Reaper todas las temáticas que están disponibles en la herramienta.

Para la implementación de las temáticas, ha sido requerido un exhaustivo trabajo de selección de timbres usando los instrumentos virtuales disponibles. El proceso es el siguiente: selección de plugin, elección o diseño del timbre dentro del plugin, guardado de un preset de Reaper y adición de dicho preset al script de Python para su posterior recuperación al lanzar el script en Reaper.

Diseño de la interfaz gráfica.

Finalmente, se *renderizaron* en 3D usando Blender los diversos fondos presentes en la pestaña de sonorización de la aplicación y se colocaron en TKInter.

Miguel González Pérez

Primera aproximación a la generación de MIDI con Magenta.

Uno de los primeros objetivos era conseguir generar melodías con música simbólica a través de la herramienta de Magenta. Dado que no fue posible para nosotros utilizar la API de Magenta desde Python en Windows, surgió la alternativa de utilizar la API de JavaScript, lo que implicaba integrar Node.js con el resto del desarrollo en Python.

Si bien logramos adquirir la melodía en forma de NoteSequence de Magenta en el contexto de JavaScript, no podíamos acceder a ello desde Python, por lo que más tarde se encargaría mi compañero Rodrigo de continuar y completar esta tarea.

Arquitectura de la Aplicación en TKInter.

Tras el anterior apartado, mi siguiente tarea fue la de montar la arquitectura de la aplicación en TKInter con Python. Gran parte del trabajo en este apartado fue el del estudio de la documentación de *TKInter*, la biblioteca estándar de Python para la creación de interfaces gráficas.

Utilizando estos conocimientos, diseñé la aplicación de forma que disgregase las funcionalidades principales en pestañas diferenciadas. Esto requería en cuanto a implementación diseñar el código de forma que gestionar y añadir varias pestañas fuera escalable y sostenible. Esto incluye también la arquitectura interna de cada pestaña, así como la colocación de los elementos en cada una de ellas, **callback** de eventos al interactuar los elementos y la visibilidad selectiva de los elementos en función del estado interno de la aplicación.

La separación en carpetas de los distintos módulos de la aplicación fue esencial para mantener orden y limpieza en el código. Algunos de estos son las constantes globales de la aplicación, nos permite conocer las rutas de archivos usadas por la aplicación, sin repetir cadenas en cada lugar; las estrategias, usadas como patrón de comportamiento por la aplicación; las imágenes, que ayudan a reflejar el estado interno de la aplicación; y los presets de la app, las distintas configuraciones de la app guardadas por el usuario.

El mantenimiento de un estado interno de la aplicación durante la ejecución y la posterior serialización a JSON y persistencia en ficheros en forma de presets son parte del trabajo que he realizado en el desarrollo de la aplicación. De este estado interno se sustrae la información necesaria, como temáticas y semillas, para la instrumentalización en Reaper. Esto implica también recuperar las configuraciones del estado interno de la aplicación guardadas por el usuario, de forma que no produzca errores.

La configuración propia de la aplicación, como la ruta al ejecutable de Reaper, también fue parte de este proceso de persistencia en formato JSON.

Para el desarrollo de la aplicación he aplicado algunos patrones de diseño software, como *Strategy* a la hora de asignar el generador musical en la pestaña *avanzado* o el propio sistema de pestañas de la aplicación, que aplica el patrón *State* para manejar la entrada, actualización y salida de cada módulo de la aplicación.

En un momento avanzado del desarrollo de la aplicación, encontramos la necesidad de explicar los distintos elementos de la interfaz de usuario de la aplicación de forma rápida y sencilla. Para esto se creó la clase *Tooltip* que permitía mostrar un pequeño texto cuando pasabas el ratón por encima de un elemento de la interfaz. Más tarde, cuando la aplicación evolucionó, optamos por usar un plugin de TKInter que traía *tooltips* integradas que funcionaban mejor, adaptándose al tema oscuro de la aplicación.

Acercándonos a los últimos momentos del desarrollo, programé los eventos que muestran una ventana de error o warning cuando hay una excepción dentro de la aplicación. Esto se hizo ampliando las excepciones de Python, creando excepciones de la propia app, con un método para mostrarse a través de una ventana emergente, diferenciando entre errores críticos y advertencias.

Finalmente, hice uso de archivos de ejecución por lotes para ayudar a la configuración e instalación automática de dependencias, con el objetivo de disminuir la intervención manual por parte del usuario y facilitar el uso de la herramienta.

Comunicación entre Vanguard Music y Reaper.

Uno de los principales retos de la aplicación era la comunicación con Reaper. Nuestra intención inicial era la de permitir al usuario utilizar la aplicación sin tener que preocuparse en exceso por Reaper. Esto planteaba varios desafíos, siendo uno de ellos el de añadir nuestros scripts de ejecución sin necesidad de que el usuario tenga que registrarlos manualmente en Reaper.

Reaper permite la ejecución de scripts en Python, *LUA* y *EEL* a través de su API *ReaScript*. No obstante, estos scripts tienen que haber sido registrados en Reaper antes de poder ser ejecutados. Si bien éramos capaces de ejecutar los scripts ya registrados en Reaper desde nuestra aplicación gracias al trabajo de Javi, no podíamos registrar nuevos scripts desde código. Mi trabajo en esta sección fue la de estudiar la documentación de *ReaScript* con el fin de enviar nuestros scripts a Reaper desde código.

Los scripts registrados en Reaper tienen un ID único asociado, no obstante este identificador hemos visto que se puede perder en algunos casos, por ejemplo, en cambios de versiones de Reaper, por lo que hacemos ahora es registrar de nuevo los scripts cuando lanzamos Reaper desde la aplicación, sustituyendo los scripts homónimos que hubiese registrado antes en el proceso.

Bibliografía

Vídeo: La música en los niveles de desierto ¿por qué suena así? | análisis musical (en español). <https://www.youtube.com/watch?v=Havwy8-1HNc>, 2020. Video de YouTube de Ludofonia (último acceso, mayo, 2024).

Vídeo: How to write drum parts (for non drummers). <https://www.youtube.com/watch?v=FoMmVlAvjmM>, 2021. Video de YouTube de 8-bit Music Theory.

Vídeo: La música en los niveles de jungla / selva ¿por qué suena así? | análisis musical de videojuegos. <https://www.youtube.com/watch?v=CCfNsBHM3fg>, 2022a. Video de YouTube de Ludofonia (último acceso, mayo, 2024).

Vídeo: Tipos de música tétrica en videojuegos | análisis musical (en español). <https://www.youtube.com/watch?v=01o0p9bv1hA>, 2022b. Video de YouTube de Ludofonia (último acceso, mayo, 2024).

Vídeo: Música de piratas ¿por qué suena así? | análisis musical de videojuegos. <https://www.youtube.com/watch?v=JB-rgtbxkhQ>, 2023a. Video de YouTube de Ludofonia (último acceso, mayo, 2024).

Vídeo: ¿qué hace a la música en los niveles de agua sonar así? | análisis musical de videojuegos. <https://www.youtube.com/watch?v=4L16jwe01rM>, 2023b. Video de YouTube de Ludofonia (último acceso, mayo, 2024).

BELUZZO, T. Pydtmc. <https://github.com/TommasoBelluzzo/PyDTMC>, 2024. (último acceso, abril, 2024).

DAHLHAUS, C. *Estudios sobre la música del siglo XIX*. Alianza Editorial, 1978.

HERRERA, E. *Teoría Musical y Armonía Moderna Vol. I*. Antoni Bosch editor, 1988. ISBN 978-84-85855-31-5.

HERRERA, E. *Teoría Musical y Armonía Moderna Vol. II*. Antoni Bosch editor, 2011. ISBN 978-84-85855-45-2.

HOCHREITER, S. y SCHMIDHUBER, J. Long Short-Term Memory. *Neural Computation*, vol. 9(8), páginas 1735–1780, 1997. ISSN 0899-7667.

- HOOKTHEORY. Hooktheory: Learn how to write songs. <https://www.hooktheory.com>, 2013-2024. (último acceso, abril, 2024).
- HUANG, C.-Z. A., HAWTHORNE, C., ROBERTS, A., DINCULESCU, M., WEXLER, J., HONG, L. y HOWCROFT, J. The bach doodle: Approachable music composition with machine learning at scale. 2019.
- ILANA SHAPIRO, M. H. Markov Chains for Computer Music Generation. *Journal of Humanistic Mathematics*, vol. 11(2), páginas 167–195, 2021. Disponible en: <https://scholarship.claremont.edu/jhm/vol11/iss2/8> (último acceso, mayo, 2024).
- MAGENTA. Magenta datasets. <https://magenta.tensorflow.org/datasets/>, 2022a. (último acceso, abril, 2024).
- MAGENTA. Magenta studio. <https://magenta.tensorflow.org/studio>, 2022b. (último acceso, abril, 2024).
- MAGENTA. Notessequence. <https://github.com/magenta/note-seq>, 2022c. (último acceso, abril, 2024).
- MAGENTA. Magenta. <https://github.com/magenta/magenta>, 2023. (último acceso, abril, 2024).
- MAGENTA. Magenta website. <https://magenta.tensorflow.org>, s.f. (último acceso, abril, 2024).
- NIELSEN, M. A. *Neural Networks and Deep Learning*. Determination Press, 2015a. Disponible en <http://neuralnetworksanddeeplearning.com/> (último acceso, abril, 2024).
- NIELSEN, M. A. *Why are deep neural networks hard to train?*. Determination Press, 2015b. Disponible en <http://neuralnetworksanddeeplearning.com/> (último acceso, abril, 2024).
- SCHOENBERG, A. *Theory of Harmony*. University of California Press, 1911.
- WIKIPEDIA. Long short-term memory. https://en.wikipedia.org/wiki/Long-short-term_memory, 2017-2024. (último acceso, abril, 2024).
- WOLFE, J. Note names, midi numbers and frequencies. <https://newt.phys.unsw.edu.au/jw/notes.html>, s.f. (último acceso, abril, 2024).

Apéndice A

Dependencias de Instalación

Para poder hacer uso nuestra aplicación, se requiere tener instaladas herramientas de terceros. Algunas de estas herramientas son esenciales para el funcionamiento correcto de nuestra aplicación, mientras que otras, como se indica más adelante, son orientativas. Cada dependencia tiene asociado el enlace donde se puede descargar este contenido.

A.1. Dependencias fundamentales

En la Tabla A.1 tenemos los programas que se requiere tener para el correcto funcionamiento de la herramienta. Aunque es posible que todo funcione correctamente con otras versiones de los programas, a veces hay problemas con eso, por lo que recomendamos usar las versiones indicadas en la tabla.

| Herramienta | Versión recomendada | Enlace | Fecha Enlace |
|-------------------------------|---------------------|----------|--------------|
| Python | 3.9 | descarga | Mayo 2024 |
| Reaper | 7.14 | descarga | Mayo 2024 |
| Node | 20.12.2 | descarga | Mayo 2024 |
| Visual Studio C++ Build Tools | - | descarga | Mayo 2024 |

Tabla A.1: Programas externos fundamentales

En la Tabla A.2 indicamos los plugins de efectos que recomendamos instalar, debido a que la herramienta está preparada para cargar presets de estos. Aún así, si el usuario quiere cambiar cualquiera de estos por otro plugin de su elección, podrá hacerlo. Eso sí, deberá añadirlos manualmente a sus pistas correspondiente, ya que la herramienta no los cargará en Reaper. Todos los plugins utilizados son gratuitos, pero instamos al usuario final a leer las licencias individuales de cada plugin.

| Plugin | Versión recomendada | Enlace | Fecha Enlace |
|-----------------------|---------------------|----------|--------------|
| Humanisator | 1.0 | descarga | Mayo 2024 |
| BlueARP | 2.5.1 | descarga | Mayo 2024 |
| Cymatics Diablo Lite | 1.1.0 | descarga | Mayo 2024 |
| Valhalla Supermassive | 3.0.0 | descarga | Mayo 2024 |
| Flux Mini 2 | 1.0.0 | descarga | Mayo 2024 |
| CRMBL | - | descarga | Mayo 2024 |
| OrilRiver | 2.0.3 | descarga | Mayo 2024 |
| After | 1.2 | descarga | Mayo 2024 |
| Delta Modulator | 1.0 | descarga | Mayo 2024 |
| Unison Zen Master | 1.0.0 | descarga | Mayo 2024 |
| Cymatics Deja Vu | 1.0.0 | descarga | Mayo 2024 |
| Overheat | 1.0.6 | descarga | Mayo 2024 |
| Ratshack Reverb | 2.2.0 | descarga | Mayo 2024 |
| Devil Spring Reverb | 1.1.7 | descarga | Mayo 2024 |
| BPB Dirty VHS | 1.0.0017 | descarga | Mayo 2024 |
| FreeTILT | 1.0.0 | descarga | Mayo 2024 |
| Cymatics Memory | 1.0.0 | descarga | Mayo 2024 |

Tabla A.2: Plugins de efectos fundamentales

A.2. Instrumentos virtuales recomendados

En la Tabla A.3 y la Tabla A.4 indicamos los plugins de instrumentos virtuales recomendados para el uso completo de la herramienta. Estos plugins son los encargados de dar sonido a las canciones generadas, no son un requisito indispensable para que la herramienta funcione, pero si el usuario los tiene descargados se cargarán automáticamente en Reaper con los presets adecuados en cada momento. Igual que sucedía con los plugins de efectos, el usuario puede cargar sus propios plugins de forma manual en Reaper si no quiere usar los instrumentos virtuales recomendados. Todos los plugins utilizados son gratuitos, pero instamos al usuario final a leer las licencias individuales de cada plugin.

| Plugin | Versión recomendada | Enlace | Fecha Enlace |
|----------------------|---------------------|----------|--------------|
| DSK Guitars Acoustic | 1.0 | descarga | Mayo 2024 |
| DSK Guitars Nylon | 1.0 | descarga | Mayo 2024 |
| Guitars Steel | 1.0 | descarga | Mayo 2024 |
| AkoustiK GuitarZ | 1.0 | descarga | Mayo 2024 |
| DSK Saxophones | 1.0 | descarga | Mayo 2024 |
| Strings | 1.0 | descarga | Mayo 2024 |
| AkoustiK KeyZ | 1.0 | descarga | Mayo 2024 |
| Room Piano | 3.2 | descarga | Mayo 2024 |
| Zither Renaissance | 2.0 | descarga | Mayo 2024 |
| Indian DreamZ | 1.0 | descarga | Mayo 2024 |
| SIM-DIZI | 1.002 | descarga | Mayo 2024 |
| Virtual Handpan | - | descarga | Mayo 2024 |
| SIM-MBIRA | 1.00 | descarga | Mayo 2024 |
| Dulcimator2 | 1.0 | descarga | Mayo 2024 |
| World StringZ | 1.1 | descarga | Mayo 2024 |
| Accordion | - | descarga | Mayo 2024 |
| Ukulele | 1.1 | descarga | Mayo 2024 |
| VSCO2 Marimba | 1.02 | descarga | Mayo 2024 |
| Sonatina Xylophone | 1.1 | descarga | Mayo 2024 |
| Iowa Alto Flute | 1.0 | descarga | Mayo 2024 |
| Sonatina Oboe | 1.1 | descarga | Mayo 2024 |
| Sonatina Trombone | 1.1 | descarga | Mayo 2024 |
| Iowa Trumpet | 1.0 | descarga | Mayo 2024 |
| Brass | 1.0 | descarga | Mayo 2024 |
| Darksichord 3 Lite | 3.0 | descarga | Mayo 2024 |
| DVS Guitar | 1.05a | descarga | Mayo 2024 |

Tabla A.3: Instrumentos vitrtuales recomendados

| Plugin | Versión recomendada | Enlace | Fecha Enlace |
|-----------------------|---------------------|----------|--------------|
| Keyzone | 1.3 | descarga | Mayo 2024 |
| ChoirZ | 1.0 | descarga | Mayo 2024 |
| Toy Keyboard | 3.0 | descarga | Mayo 2024 |
| Nu Guzheng | 2.0 | descarga | Mayo 2024 |
| SIM-SHENG | 1.002 | descarga | Mayo 2024 |
| OMB2 | 1.0 | descarga | Mayo 2024 |
| SamsaraSitar | 1.0 | descarga | Mayo 2024 |
| Cute Emily Guitar | 1.1 | descarga | Mayo 2024 |
| Strat-A-Various | - | descarga | Mayo 2024 |
| Upright Piano | - | descarga | Mayo 2024 |
| DPiano-A | 1.2 | descarga | Mayo 2024 |
| TAL-NoiseMaker | 5.0.6 | descarga | Mayo 2024 |
| Spicy Guitar | 1.3 | descarga | Mayo 2024 |
| Electrik GuitarZ | 1.0 | descarga | Mayo 2024 |
| MF Concert Guitar | 1.0 | descarga | Mayo 2024 |
| SIM-DJEMBE | 1.0 | descarga | Mayo 2024 |
| SIM-DUNUN | 1.001 | descarga | Mayo 2024 |
| Dynamic Guitars | 1.1 | descarga | Mayo 2024 |
| Abstract Crystal Pads | 1.0 | descarga | Mayo 2024 |
| Sonatina Glockenspiel | 1.1 | descarga | Mayo 2024 |
| Water | - | descarga | Mayo 2024 |
| Oscine Tract | - | descarga | Mayo 2024 |
| B3x | 1.0 | descarga | Mayo 2024 |
| BassZ | 1.0 | descarga | Mayo 2024 |
| mini DrumZ 2 | 1.0 | descarga | Mayo 2024 |
| Tenpan | - | descarga | Mayo 2024 |
| 8bitZ | 1.0 | descarga | Mayo 2024 |
| Clap Machine | 1.0 | descarga | Mayo 2024 |
| Drumplayer | - | descarga | Mayo 2024 |
| Afroplugin | 1.1 | descarga | Mayo 2024 |
| Cassette Drums 606 | - | descarga | Mayo 2024 |
| Cassette Drums 808 | - | descarga | Mayo 2024 |
| Cassette Drums 909 | - | descarga | Mayo 2024 |
| Sonatina Timpani | 1.1 | descarga | Mayo 2024 |
| Ocarina | 1.0 | descarga | Mayo 2024 |
| SIM-MIJWIZ | 1.00 | descarga | Mayo 2024 |
| Pianotone 600 | 2.0 | descarga | Mayo 2024 |
| EVM Grand Piano | 1.1 | descarga | Mayo 2024 |
| Free Amp 3 | 3.6 | descarga | Mayo 2024 |

Tabla A.4: Instrumentos virtuales recomendados

Apéndice B

Representaciones de una canción

En el módulo al que se hace referencia durante toda la Sección 5 se manejan dos representaciones distintas de las notas:

1. **Lista de notas:** es la representación estándar en este módulo de la aplicación.

Es la más cómoda para realizar operaciones simples con las notas, como por ejemplo recortar una canción, transponer todas las notas, etc... Se almacenan en una lista desordenada de estructuras con el siguiente formato:

- **note:** almacena el *pitch* MIDI (tono) de la nota. Guardar el *pitch* MIDI es similar a guardar la nota musical, ya que este almacena de forma implícita el nombre de la nota y su octava.

| Nombre | C | Db | D | Eb | E | F | Gb | G | Ab | A | Bb | B |
|--------|---|----|---|----|---|---|----|---|----|---|----|----|
| Pitch | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

Tabla B.1: Relación entre nota y su *pitch* en la primera octava

Por ejemplo, si calculamos a qué nota le corresponde el *pitch* 40, sabemos que el nombre de la nota es E si nos fijamos en la tabla, ya que $40 \bmod 12 = 4$ y sabemos que pertenece a cuarta octava, ya que $40 \div 12 = 3$ (se empieza en la octava 0).

- **start_time:** tiempo en ticks en el que empieza a sonar una nota desde que empieza la canción en el tick 0.
- **duration:** tiempo en ticks desde que empieza a sonar la nota hasta que para.

2. **Diccionario de eventos:** se almacena en cada tick clave el evento que ha ocurrido. Los eventos están ordenados de menor a mayor según el tick en el que ocurren. Como tal, solo existen dos tipos de eventos: **note_on** y **note_off**.

A cada evento viene asociado el *pitch* de la nota afectada. Esto puede suponer, a priori, un inconveniente, ya que, si hay dos notas del mismo *pitch* superpuestas en el mismo espacio de tiempo, existe una ambigüedad a la hora

de saber qué evento *note_off* le corresponde a cada una. Sin embargo, como esta representación es únicamente utilizada en la armonización por ventanas (Sección 5.2) para hacer un recorrido lineal de las notas de la melodía, este inconveniente no les afecta.

Existen métodos para pasar de la anterior representación a esta, pero no al revés. De nuevo, esto se debe a que esta representación solo se utiliza para realizar la armonización por ventanas.