AN EUCLIDEAN SEQUENCER PLUGIN FOR DIGITAL AUDIO WORKSTATIONS

UN PLUGIN DE SECUENCIADOR EUCLIDIANO PARA ESTACIONES DE TRABAJO DE AUDIO DIGITAL

TRABAJO FIN DE GRADO
CURSO 2021-2022

AUTORES
DAVID BURGOS DÍAZ
GONZALO COSTALES DE LEDESMA

DIRECTOR
MIGUEL GÓMEZ-ZAMALLOA GIL

GRADO EN INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

AN EUCLIDEAN SEQUENCER PLUGIN FOR DIGITAL AUDIO WORKSTATIONS

UN PLUGIN DE SECUENCIADOR EUCLIDIANO PARA ESTACIONES DE TRABAJO DE AUDIO DIGITAL

TRABAJO DE FIN DE GRADO EN INGENIERÍA INFORMÁTICA
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

AUTORES
DAVID BURGOS DÍAZ
GONZALO COSTALES DE LEDESMA

DIRECTOR
MIGUEL GÓMEZ-ZAMALLOA GIL

CONVOCATORIA: Junio 2022
CALIFICACIÓN:

GRADO EN INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

17 DE MAYO DE 2022

Índice general

Resumen

Un plugin de secuenciador euclidiano para estaciones de trabajo de audio digital

La constancia en el desarrollo y la innovación en la informática desde que la conocemos ha supuesto a lo largo del tiempo una revolución en muchos sectores. En esta ocasión nos centraremos en el musical, ya que, echando la vista atrás varias decenas de años, para realizar una composición musical eran necesarios muchos medios e instrumentos que hacían de esta, una tarea a veces ardua y compleja que muy pocos podían llevar a cabo. Sin embargo, todo esto ha cambiado ya que hoy en día cualquier persona puede ser productor de música con el simple hecho de tener un ordenador a su alcance.

Que esto sea posible se lo debemos a las DAWs, del inglés *Digital Audio Workstations* o estaciones de trabajo de audio digital. Una DAW es básicamente un programa o software que permite grabar, editar, procesar y mezclar múltiples pistas de audio, además de escribir música y hacerla sonar mediante instrumentos virtuales. A parte de la infinidad de funcionalidades por defecto, las DAWs pueden ser extendidas mediante plugins externos. Dentro de este gran abanico de plugins, cabe destacar por su relación con este proyecto, a los secuenciadores. La función de estos es la de secuenciar música, o en otras palabras, programar eventos musicales en el tiempo. Esto es posible haciendo uso de cierta información proporcionada por el usuario en forma de acordes, parmetros, etc.. u otros datos recogidos de la interfaz gráfica de usuario.

El fin de este proyecto es desarrollar un plugin de secuenciador euclidiano, que está basado en los ritmos euclidianos, cuyos orígenes provienen, como más adelante se explica, del Algoritmo de Euclides. Resumidamente, usando el máximo común divisor logra colocar elementos en una secuencia de la manera más uniforme posible en una división de tiempo dada, estableciendo un número de pasos y de eventos que definen un patrón rítmico.

De este modo, gracias a la función que realiza este plugin de generación de eventos musicales, si lo combinamos con otro de instrumento virtual, se estaría generando música de manera casi automática.

Palabras clave

audio digital, DAW, plugin, VST, MIDI, JUCE, ritmos euclidianos, secuenciador euclidiano

Abstract

An Euclidean Sequencer plugin for Digital Audio Workstations

The constancy in the development and innovation in computer science since we know it has meant over the years a revolution in many fields. Let us focus on the music field, since, looking back several years, to make a musical composition many tools and instruments were necessary, making it an arduous and complex task that very few people could carry out. However, all this has changed since nowadays anyone can be a music producer with the simple fact of having a computer at their fingertips.

This is possible thank to the DAWs (Digital Audio Workstations). A DAW is basically a program or software that allows to record, edit, process and mix multiple audio tracks, in addition to writing music and making it sound using virtual instruments. In addition to an infinity of default tools, we can acquire and extend these functionalities using plugins. Within this wide range of plugins, we will mention for its relevance int this project, the sequencers. The goal of these plugins is to sequence music, or in other words, program musical events. This is possible by making use of certain information provided by the user in the form of chords, parameters, etc. or other data collected from the graphical user interface.

The purpose of this project is to develop a Euclidean sequencer plugin, which is based on Euclidean rhythms, whose origins come, as explained later, from Euclid's Algorithm. Summarizing, using the maximum common two-number divisor, it manages to place elements in a sequence as evenly as possible in a set division of time. What in a practical way is establishing a number of steps and events that define a rhythmic pattern.

In this way, thanks to the work performed by this plugin generating musical events, if we combine it with another plugin of virtual instrument, music would be generated.

Keywords

digital audio, DAW, plugin, VST, MIDI, JUCE, Euclidean rhythms, Euclidean sequencer

Capítulo 1

Introducción

1.1. Motivación

1.1.1. Un poco de historia

Tras muchas ideas a lo largo del final del siglo XIX y todo el siglo XX sobre la creación musical con el uso de las tecnologías, finalmente en la década de 1980 pasó a la práctica este concepto con el lanzamiento de los sintetizadores digitales por parte de Yamaha, concretamente, la serie DX. La popularidad de estos instrumentos capaces de producir sonidos utilizando puramente circuitería digital fue tal que, el Yamaha DX7, alcanzó unas cifras que lo colocan a día de hoy como uno de los sintetizadores más vendidos de la historia.

Figura 1.1: *Yamaha DX7*

En esta época de transición de lo analógico a lo digital, caben destacar descubrimientos como:

- El lanzamiento del *CD* o disco compacto en 1982, de manera conjunta por Sony y Philips tras varios prototipos individuales, que cambió la manera de compartir la

música ya que introdujo la capacidad de almacenar y reproducir audio digital.

- En 1985 el lanzamiento del *Commodore Amiga*, un ordenador personal con un precio muy asequible y unas capacidades multimedia que impulsaron la producción musical usando la tecnología.
- El desarrollo y la presentación del MIDI (siglas de *Musical Instrument Digital Interface*), un estándar tecnológico que describe un protocolo, una interfaz digital y unos conectores que hacen posible la enviar y almacenar información musical simbólica, o eventos musicales (comumente llamdados eventos MIDI) entre ordenadores, instrumentos musicales y hardware.

1.1.2. Estaciones de Trabajo de Audio Digital

Hoy en día, la producción musical y el audio digital pasan por su mejor momento, ya que cualquier persona con unos conocimientos no demasiado avanzados puede desarrollar proyectos musicales más que respetables sin requerir de un estudio de música profesional o grandes inversiones de dinero. Esto es gran parte debido a las llamadas Estaciones de Audio Digital o DAWs (*Digital Audio Workstations*), sistemas de grabación, edición y producción de audio digital compuestos por herramientas software (plugins, instrumentos virtuales, etc...) y hardware (micrófonos, monitores, etc..).

Durante la segunda mitad del siglo XX, las DAWs estaban compuestas en su totalidad por componentes hardware (con un software exclusivamente dedicado a él) ya que las limitaciones informáticas de la época eran notables. A medida que estas limitaciones de almacenamiento y procesamiento de datos se fueron reduciendo, las DAW fueron evolucionando hasta un punto en el que, todo lo que se necesitaba para producir audio digital que fácilmente llenaba un estudio y era bastante caro, pasó a ser básicamente una herramienta software que puede usarse en ordenadores convencionales e incluso dispositivos móviles, requiriendo nada más de una interfaz para que el usuario pueda introducir audio y MIDI en el dispositivo obteniendo una salida de audio reproducible en otros dispositivos.

Estos complejos softwares son utilizados desde por los usuarios más básicos hasta en los mejores estudios alrededor de todo el mundo por tres principales razones:

- **Entorno multi-pista:** Nos da la capacidad de trabajar con varias pistas de manera independiente.
- **Larga duración:** Nos proporciona una extensa duración de proyecto, superando las 5 horas fácilmente, o incluso más, en función de la frecuencia de muestreo a la que estemos trabajando.
- **Herramientas de calidad incluidas:** Las DAWs vienen equipadas con instrumentos software como sintetizadores o samplers entre otros, y con efectos de audio como

ecualizadores, compresores, reverberadores, etc.. Además, al conjunto de herramientas por defecto le podemos añadir una infinidad más de instrumentos, efectos y plugins que podemos adquirir fácilmente.

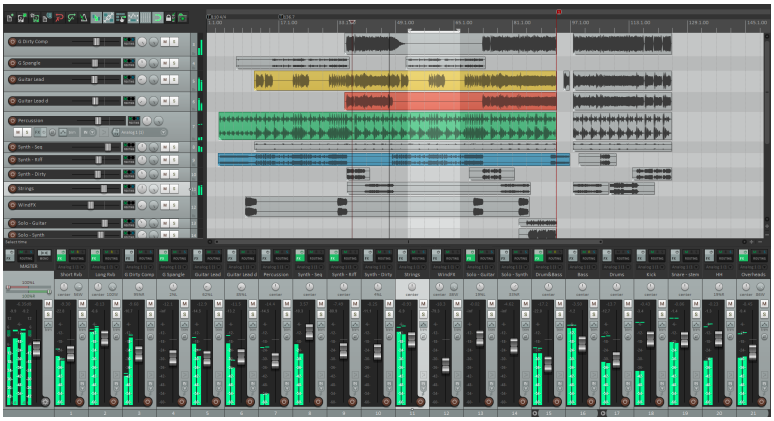

Figura 1.2: *Interfaz de usuario de REAPER*

1.1.3. Plugins

Como hemos comentado previamente, las DAWs vienen con un conjunto de herramientas por defecto a las que se le pueden añadir una infinidad de funcionalidades extra gracias a diferentes extensiones entre las que se encuentran los plugins VST.

VST, de las siglas *Virtual Studio Technology*[11], es un estándar desarrollado por la empresa alemana *Steinberg* cuyo fin es el de ofrecer la posibilidad de desarrollar y conectar plugins de efectos y sintetizadores de audio con las DAWs. A parte de VST, existen otros estándares como AU (*Apple*) o AAX (*Avid Audio*), sin embargo, VST es el más usado de todos.

En el próximo capítulo entraremos más en profundidad hablando tanto de la estructura de las DAWs, como de los tres grandes grupos de plugins que se utilizan en ellas: los efectos de audio (cuya función es la de transformar audio), los instrumentos virtuales (que generan audio a partir de eventos MIDI) y los secuenciadores o generadores MIDI (que asisten en la labor de secuenciación o programación de eventos musicales).

1.2. Objetivos

Con este proyecto el objetivo principal que deseamos cubrir es el de crear un generador de MIDI que permita al usuario probar un paradigma al que no está acostumbrado a la hora de crear y configurar ritmos. La implementación de este generador como un plugin con el estándar VST trae consigo accesibilidad para cualquier usuario y versatilidad de poder

usarlo en la estación de trabajo de audio digital que se prefiera.

Además, se plantean los subobjetivos que se comentan a continuación:

- Presentar una interfaz profesional, al mismo que amigable y fácil de comprender por parte del usuario, sin necesidad de tener que leerse extensos manuales.
- Ofrecer la posibilidad de crear ritmos hasta con cuatro etapas a la vez, es decir, que en la misma instancia del plugin, puedas producir cuatro patrones rítmicos distintos (o no) simultáneamente.
- Allow you to connect the output of the plugin to the MIDI channel of the digital audio workstation you want, connecting each instance of the plugin to sound to the instrument you want.
- Ofrecer un cómodo panel de control desde el cual el usuario pueda configurar como desee los distintos parámetros del ritmo que quiera obtener.
- Posibilidad de sincronizar todos los ritmos que estén sonando en un momento dado para que empiecen de nuevo sus patrones, lo que permitirá al usuario poder tener control sobre la sincronización sin necesidad de andar parando y reanudando los ritmos.
- Ofrecer la funcionalidad de guardado y cargado de *presets* al usuario, de forma se puedan cargar y guardar configuraciones del plugin en el momento que se desee.

Para terminar, y hablando de un nivel de aprendizaje personal, los objetivos que planteamos son los siguientes:

- Desarrollar una buena estructura de trabajo y una buena organización que nos faciliten las labores de aprendizaje y colaboración en un proyecto grupal, donde la sincronización y la comunicación con el otro miembro del equipo sea óptima para que el rendimiento en conjunto sea el mejor posible.
- Aprender a usar y lidiar con nuevas herramientas y software como por ejemplo el *framework JUCE*, con el fin de que el avance del proyecto y el desarrollo de nuestras ideas y objetivos se faciliten.
- Hacer frente a problemas y situaciones que no hayamos conocido durante nuestro aprendizaje en la carrera, y poder solventarlos investigando y adquiriendo nuevos conocimientos que muy posiblemente nos sean útiles en un futuro.

1.3. Plan de trabajo

Una vez establecidos los objetivos del proyecto y habiendo descrito las motivaciones del mismo, es hora de describir el proceso que hemos seguido para llevar a cabo el proyecto con el cumplimiento de estos objetivos mencionados.

Los primeros pasos de este trabajo fueron investigar el origen del secuenciador euclidiano, lo que nos llevó a comprender cómo surgieron los ritmos euclidianos, y como, a día de hoy, están presentes alrededor de todo el mundo. Tras comprender estos, el siguiente paso fue desarrollar un algoritmo que fuera capaz de producir estos ritmos, proporcionándole como entrada el número de pasos del ritmo, y el número de eventos que este tuviera de manera que los distribuyera uniformemente creando el ritmo pertinente.

El lenguaje que usamos para desarrollar este algoritmo fue C++ ya que, nuestro siguiente paso sería el de investigar y familiarizarnos con el *framework JUCE*[6], un entorno de trabajo que ofrece todas las librerías y herramientas necesarias para desarrollar aplicaciones, plugins y librerías orientadas al audio y/o música haciendo uso del mencionado lenguaje de programación C++. Este proceso de familiarización con *JUCE* se vio apoyado enormemente con el hecho de que en su página web hay un proceso de aprendizaje compuesto por pequeños módulos con tutoriales que fuimos completando y una infinidad de vídeos en YouTube donde encontrar ayuda. Nos gustaría destacar concretamente un canal llamado *The Audio Programmer*[8], el cual ha sido un gran apoyo en este proceso de aprendizaje de principio a fin del proyecto.

Figura 1.3: *Logo de JUCE*

Tras haber desarrollado en esta primera etapa de investigación plugins bastantes sencillos como controladores de volumen, wavetables o controladores de envolventes que nos ayudaron a comprender la estructura de un plugin, y el funcionamiento de *JUCE*, nuestro tutor nos aconsejó el desarrollo de un plugin generador/efecto de MIDI, debido a que de esta manera afianzaríamos conceptos similares a los necesarios para desarrollar el secuenciador euclidiano. Nuestra elección fue desarrollar un simple arpegiador que, tomando unos eventos MIDI de entrada, los modificaba cambiándolos a un tono mayor o menor de nuestra elección, o de forma aleatoria.

Una vez finalizada la construcción de este plugin, y entendido el manejo y creación de eventos MIDI, empezamos a desarrollar el proyecto final. Tras numerosas versiones a lo largo de este proceso de desarrollo se consiguieron implementar los componentes y funcionalidades básicos del plugin, destacando el controlador principal del ritmo, que permite al usuario mediante sliders rotativos seleccionar el número de pasos y eventos del mismo, y otros controles adicionales, desde el control de dirección del ritmo, hasta posibilidad de agrupar las notas en tresillos y notas con puntillo. Además, se invirtió una gran cantidad de tiempo a la interfaz debido a su gran complejidad que requiere de señales que la hagan cambiar en tiempo real a una gran velocidad.

Finalmente, se implementaron las funcionalidades de guardar y cargar *presets* que dieran la posibilidad al usuario de almacenar las configuraciones y ritmos que desee para su posterior uso.

Capítulo 2

Estado del arte

Para lograr entender lo útil e importante que puede ser un plugin de estas características dentro del campo de la producción musical debemos desarrollar una explicación del estado en el que se encuentran este tipo de generadores/efectos de MIDI. Para ello explicaremos un poco más en detalle las distintas herramientas que componen las DAWs, los tipos de plugins que hay y lo que es un secuenciador, pero antes, un poco de contexto.

2.1. Origen de los ritmos euclidianos

En 2004, el informático *Godfried Toussaint* descubrió lo que eran los ritmos euclidianos en la música y la capacidad que tenían para generar casi todos los ritmos alrededor del globo, exceptuando la India. En 2005, este mismo profesor describió lo que eran estos ritmos en su paper "*The Euclidean Algorithm Generates Traditional Musical Rhythms*"[9]. En este paper, *Toussaint* aplica el algoritmo de Euclides del maximo común divisor de dos números, correspondientes al número de pasos de un ritmo, y al número de eventos o pulsos del mismo. El resultado obtenido son los números de pulsos y silencios de un ritmo, quedando los pulsos de un ritmo equidistantes entre sí.

El algoritmo a continuación explicado es el cual el profesor desarrolló en el paper, directamente implementado con el algoritmo de Euclides[2]. Pensemos en que un ritmo se puede representar directamente como una cadena binaria de 1s y 0s, donde los 1s representan eventos o pulsos y los 0s representan silencios de manera que, se desean distribuir uniformemente **k eventos** en **n pasos**. Si por ejemplo tomamos $\mathbf{n = 12}$ **pasos** y $\mathbf{k = 4}$ **eventos** el ritmo euclidiano resultante sería $\mathbf{E(4,12) = [1,0,0,1,0,0,1,0,0,1,0,0]}$. El problema viene cuando el máximo común divisor de n y de k no es uno de estos dos números, es decir, que entre todos los eventos no hay el mismo número de silencios (eventos no equidistantes).

Tomemos un ejemplo donde el máximo común divisor de n y k sea 1 para explicar el algoritmo, como por ejemplo, $\mathbf{n = 13}$ y $\mathbf{k = 5}$:

1. Empezaremos colocando los cinco 1s seguidos de los ocho 0s correspondientes:

[1 1 1 1 1 0 0 0 0 0 0 0 0]

2. Ahora procedemos a colocar cada 0 detrás de un 1, obteniendo 5 secuencias de pares [10] y tres 0s restantes:

[10] [10] [10] [10] [10] [0] [0] [0]

3. Lo siguiente será distribuir los tres 0s restantes de una manera similar, colocando cada [0] detrás de una secuencia [10]:

[100] [100] [100] [10] [10]

4. A continuación repetiremos el mismo proceso tantas veces sea necesario hasta acabar con una sola secuencia resultante:

[1001010010010]

Cabe destacar que, al ser una secuencia cíclica, la secuencia obtenida puede ser rotada, de manera que obtendríamos permutaciones del mismo ritmo, desplazando simplemente los eventos por el ritmo:

$$\textbf{rot([1001010010010], 1) = [0100101001001]}$$

Podemos ver diferentes ritmos en función de la rotación, por ejemplo, con el ritmo E(7,12). Este ritmo sin ninguna rotación corresponde al *Bembé* (a), con 1 rotación a la derecha corresponde al ritmo de palmeo del oeste del continente africano (b), y con 2 unidades de rotación a la derecha al ritmo *Tambú de Curaçao* (c):

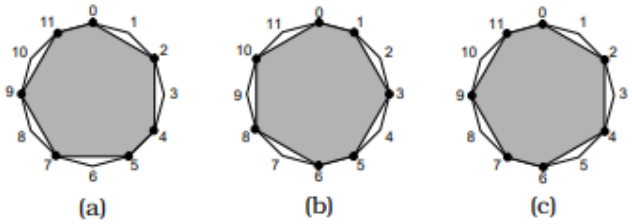

Figura 2.1: *Ejemplos de rotación del Bembé africano*

Para descubrir y conocer muchísimos otros ritmos euclidianos como el tresillo cubano E(3,8), la cumbia colombiana E(3,4) o la *Bossa-Nova* brasileña E(5,16) se recomienda leer el paper de *Godfried Toussaint* anteriormente mencionado.

Tras una puesta en contexto sobre la historia que tiene a sus espaldas el plugin desarrollado en este proyecto, hablemos un poco más en profundidad sobre las DAWs, ya que son las herramientas donde las usaremos, y sobre los plugins, para ver los tipos que hay y como funcionan.

2.2. Estaciones de Trabajo de Audio Digital

A pesar de la compleja interfaz que muestra una DAW al iniciarse, comprendiendo el funcionamiento de los siguientes elementos fundamentales, tendremos la posibilidad de llevar a cabo una producción musical. Para esta explicación tomaremos REAPER como representante, teniendo en cuenta que la gran mayoría de cosas que se presentan en esta DAW son análogas en cualquier otra.

1. **Línea de tiempo:** Es el esqueleto de una DAW, debido que a lo largo de esta línea es donde se irán colocando los diferentes bloques de audio o MIDI. En el eje horizontal está representado el tiempo, separado en diferentes compases, y en el vertical se diferencian las filas o pistas que componen el proyecto.

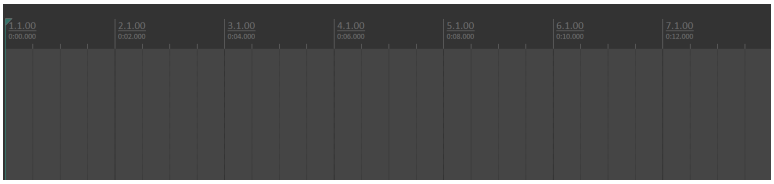

Figura 2.2: *Línea de tiempo en REAPER*

2. **Panel de control:** Esta parte de la DAW es el centro de control de nuestro proyecto, desde donde podemos iniciar, pausar o grabar una secuencia del mismo. Además también se encuentran las características principales de nuestro proyecto como el compás musical o los pulsos por minuto ente otros, que nos indican la velocidad de reproducción. Estas características se pueden ajustar en cualquier momento para adaptarse a las necesidades que se deseen.

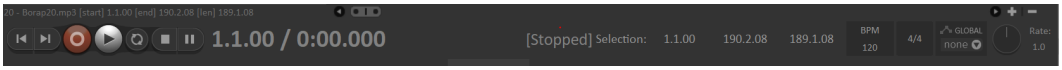

Figura 2.3: *Panel de control en REAPER*

3. **Control individual de pista:** Una vez seleccionada la pista deseada de la línea de tiempo, se mostrará su vista con el control individual, donde se le podrán aplicar efectos y todo tipo de modificaciones, además de silenciarla (pulsando la "M") o aislarla (pulsando la "S") entre otras cosas.

Debemos hacer especial hincapié en esta parte de las DAWs ya que es el componente desde el cual se controlan los efectos o plugins que usamos en cada pista. Como podemos observar en la Figura 2.4, en cada pista destaca el bloque *FX* o bloque de efectos; este es el encargado de administrar el manejo del plugin o plugins que hayamos decidido usar en la pista. En el bloque FX es donde añadiremos los plugins a modo de secuencia, es decir, uno conectado a la salida de otro. Una de las principales características que lo conforman es que, dichos efectos son aplicados en tiempo en real. Veamos esto con un sencillo ejemplo:

Imaginemos que tenemos un teclado MIDI por el cual introduciremos la entrada (las notas que deseemos tocar). A esta entrada se le podría aplicar una cadena de efectos formada por un arpegiador, un instrumento virtual de piano y finalmente un reverberador. De esta manera por ejemplo si se pulsa un acorde de do mayor, sonaría un patrón melódico con las notas do-mi-sol en un piano y con una cierta reverberación, donde si se modifica cualquiera de esos efectos, la salida se vería modificada instantáneamente.

Figura 2.4: *Control individual de pista en REAPER*

4. **Editor MIDI:** Este componente lo que nos ofrece es una interfaz gráfica mediante la cual podemos programar eventos MIDI en el tiempo o, si han sido previamente grabados por ejemplo con un teclado MIDI, nos ofrece también la capacidad de editarlos. MIDI, de las siglas inglesas *Musical Instrument Digital Interface*, es un protocolo basado en la representación y el transporte de eventos cuya información simbólica representa notas musicales; destacando entre otros parámetros la duración, notación musical o intensidad. Este estándar tecnológico es el usado por las DAWs para represntar dicha información y además, como más adelante veremos, puede ser interpretada

por otros dispositivos, programas, o instrumentos electrónicos para generar una salida de audio.

Para crear dichos evento solamente debemos abrir el editor MIDI e introducir valores con el ratón u otros dispositivos como teclados MIDI. Además, gracias a estos editores MIDI, podremos editar las características de cada nota individualmente, de modo que la customización nos facilite alcanzar las preferencias deseadas.

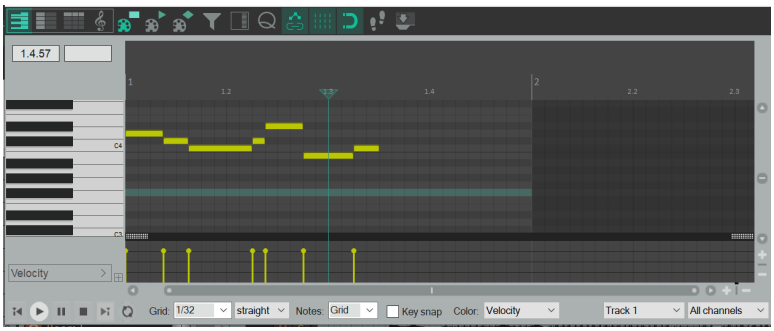

Figura 2.5: *Control individual de pista en REAPER*

5. **Mezclador:** Este componente nos facilita una vista general de todas las pistas donde tendremos la posibilidad de modificar parámetros en todas ellas al mismo tiempo, o solamente en las que deseemos, desde un punto de vista común a todas.

Figura 2.6: *Control individual de pista en REAPER*

2.3. Plugins

Como hemos comentado previamente, VST, de las siglas Virtual Studio Technology, es el estándar de desarrollo y conexión de plugins más usado a día de hoy alrededor de todo el mundo. Dicho esto, se conoce como plugin VST a todo aquel software cuyo fin es ampliar las funcionalidades que una estación de trabajo de audio digital ya tiene. Dos de sus principales ventajas son, la capacidad de utilización y funcionamiento en cualquier DAW (REAPER, Ableton, Logic, etc...) y la respuesta en tiempo real del mismo, es decir, que vemos la respuesta o el cambio que produce el plugin VST en la pista de forma instantanea. Destacan

3 tipos principales de plugins:

1. **Efectos de audio:** Este tipo de plugins desarrollan la función de, tomar una entrada de audio digital, procesarla o transformarla, y devolver ese audio digital modificado. Podemos destacar entre ellos el ecualizador, el compresor o el reverb.

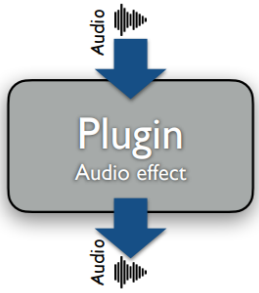

Figura 2.7: *Esquema de un efecto de audio*

2. **Instrumentos virtuales (*VSTi*):** A diferencia de los efectos de audio, estos plugins lo que reciben es información musical simbólica (eventos MIDI) y sintetizan audio a partir de ellos, logrando emular el sonido de instrumentos reales como sintetizadores, bajos, o pianos, sin la necesidad de gastarse grandes cantidades de dinero en ellos.

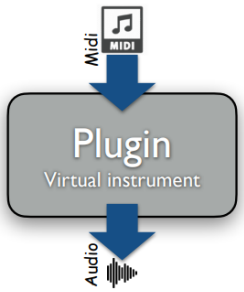

Figura 2.8: *Esquema de un instrumento virtual*

3. **Secuenciadores/generadores MIDI:** Estos plugins son parecidos a los primeros que hemos visto, los efectos de audio, con la diferencia que en lugar de recibir audio digital, pueden recibir MIDI, o no hacerlo. Esto significa que tienen la capacidad de generar, procesar y/o transformar MIDI. Algunos ejemplos destacables son el arpegiador, el generador de acordes, el humanizador o, como en el caso del plugin que se desarrolla en este proyecto, el secuenciador, más concretamente, el secuenciador euclidiano.

Como veremos en la siguiente sección, hay varios tipos de secuenciadores dentro de los plugins generadores/efectos de audio, destacando el convencional, muchas veces confundido

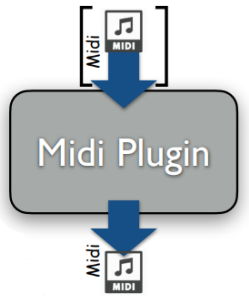

Figura 2.9: *Esquema de un generador/efecto MIDI*

con la propia DAW, y el secuenciador euclidiano. Tras estudiar como funcionan, podremos entender los procesos que llevan a cabo estos plugins de programar y reproducir eventos musicales (MIDI) de una forma secuencial con el fin de componer y llevar un control sobre equipos de música electrónica (sintetizadores, cajas de ritmos, samplers, etc...).

2.4. Secuenciadores

Tras haber definido lo que son los ritmos euclidianos, a continuación veremos como a día de hoy se pueden conseguir haciendo uso de secuenciadores.

La funión principal de estas herramientas es la de programar y reproducir eventos musicales (MIDI) de una forma secuencial con el fin de componer y llevar un control sobre equipos de música electrónica (sintetizadores, cajas de ritmos, samplers, etc...). Estos secuenciadores MIDI permiten al usuario grabar y editar música sin la necesidad de una fuente de entrada de audio. Estos datos luego se reproducen en un instrumento MIDI o módulo de sonido. Con este método, el intérprete puede seleccionar un sonido de piano para un fragmento musical y luego decidir que el fragmento funcionaría mejor como un sonido de órgano.

1. **Secuenciador estándar:** En su definición más amplia, un secuenciador es un programa o máquina física que permite grabar, editar y reproducir eventos musicales, siendo MIDI la interfaz de control más común para ello. Debido a la amplia cantidad de máquinas y sofware que se recogen bajo este término, esta no suele ser la definición de secuenciador estándar, entendiendose normalmente por "secuenciador" al secuenciador de pasos. El término "secuenciador de pasos" recoge máquinas y software que permiten generar secuencias rítmicas dividiendo el ritmo en pasos del mismo tamaño. Para ello se suele usar la representación de rejilla (vease la figura 2.10). Esta consiste en una matriz de huecos del mismo tamaño que se pueden ir rellenando, siendo los huecos silencios, y los bloques rellenos pulsos sonoros. Habitualmente el eje horizontal representa el tiempo, mientras que el eje vertical varía, pudiendo representar, por ejemplo, distintos elementos de un kit de percusión, normalmente llamados *drum ma-*

chine(figura 2.11) o las notas de un teclado, en cuyo caso el secuenciador se conocería como *piano roll* (figura 2.12).

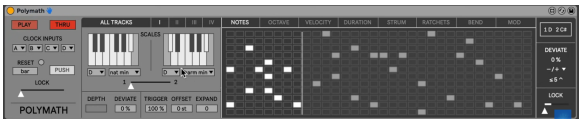

Figura 2.10: *Secuenciador Polymath*

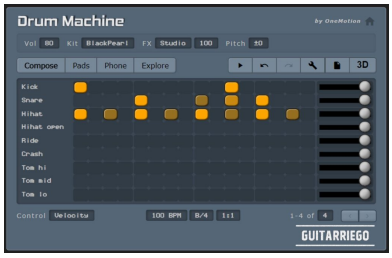

Figura 2.11: *OneMotion drum machine*

Figura 2.12: *Secuenciador Stochas*

2. **Secuenciador euclidiano:** Este tipo de secuenciador es una muy buena alternativa al clásico secuenciador por pasos anteriormente descrito. Cambia el paradigma de programar los eventos MIDI y sus propiedades en una línea temporal de un compás, por la creación de patrones rítmicos euclidianos. En la práctica esto es mucho más sencillo de cómo suena en la teoría ya que, lo único que hay que hacer es introducir el número de total de pasos del compás, y el número de eventos o pulsos que deseamos que tenga, y será el propio plugin el que cree el ritmo, donde los pulsos introducidos estarán distanciados unos de otros de la manera más uniforme posible.

   Además de estos dos parámetros es posible ajustar muchos más, como la rotación del ritmo que anteriormente ha sido nombrada, la nota MIDI que queremos que suene,

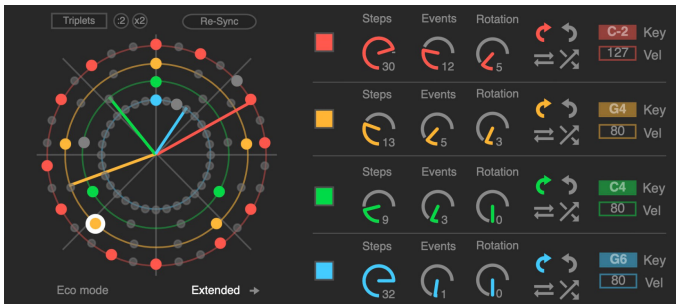

Figura 2.13: *Euclidean sequencer PRO 2.1 - Alkman*

le dirección en la que queremos que se interprete el ritmo, la intensidad de la nota y otros parámetros muy interesantes.

A lo largo de este trabajo se irá estudiando el funcionamiento y la lógica de este plugin con el fin de poder entenderlo y desarrollar, aplicando lo aprendido, nuestro propio plugin de secuenciador euclidiano.

Capítulo 3

Manual de uso del plugin

En esta sección se explicará cómo poner en marcha nuestro plugin en una DAW (en este caso será REAPER, pero siguiendo un proceso muy similar podremos comenzar a utilizarlo en cualquier otra DAW) además de detallarse las diferentes funcionalidades del mismo. Para una mayor agilidad en el proceso de aprendizaje y familiarización recomendamos tener un instrumento virtual descargado que poder conectar a la salida de nuestro secuenciador para poder escuchar los eventos MIDI que este va generando.

3.1. Instalación

La primera tarea a realizar es simple, debemos descargar los ficheros necesarios para instalar nuestro plugin. Podemos obtener dichos ficheros a través del siguiente enlace:[4]. En él encontraremos una carpeta que conforma el plugin en formato VST3.

Una vez descargada dicha carpeta, debemos colocarla en la ruta adecuada para que nuestra DAW reconozca el plugin. Las carpeta por defecto donde se suelen encontrar los plugins VST3 dependiendo del sistema operativo que estemos usando son:

- **Windows**: *C:\Program Files\Common Files\VST3*
- **OS X**: *~/Library/Audio/Plug-Ins/VST3*

Tras haber colocado los archivos en la carpeta correspondiente, solo abrir nuestra DAW y, por si por ella sola no ha sido capaz de reconocer el nuevo plugin al iniciarse, forzamos dicho reconocimiento yendo en el menú superior a ***Options*** > ***Preferences*** > ***Plug-ins*** > ***VST*** y pulsando el botón de ***Re-scan***.

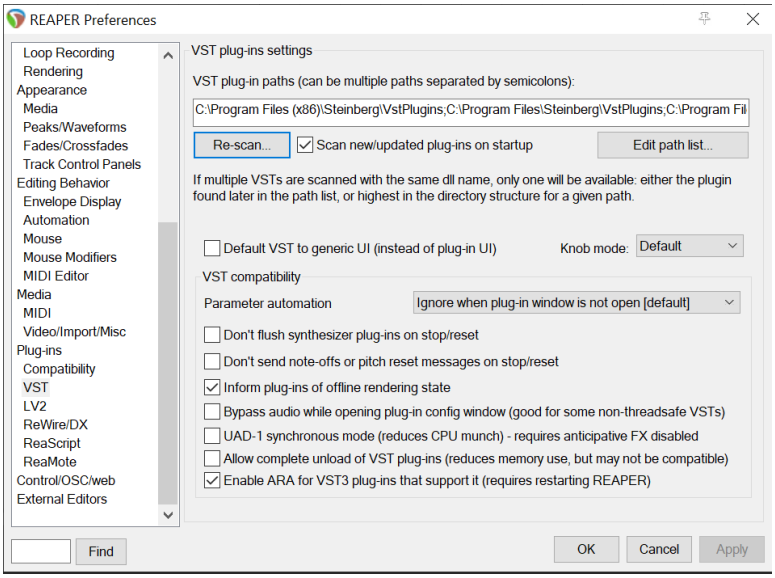

Figura 3.1: *Re-scan de plugins en REAPER*

A continuación, crearemos un nuevo proyecto en el cual insertaremos una pista. Tras esto, pulsando en el botón *FX* de la misma se nos abrirá el bloque de efectos donde buscaremos mediante la barra de búsqueda o yendo a ***Developers*** > **Universidad Complutense de Madrid** y añadiremos nuestro plugin llamado ***EuclideanSequencer***.

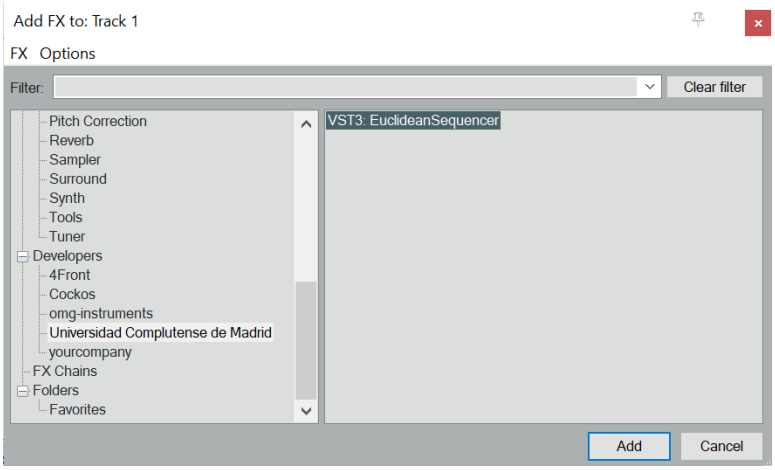

Figura 3.2: *Selección de plugin en REAPER*

Por último, y con el fin de poder escuchar en tiempo real los ritmos MIDI que va creando nuestro secuenciador, debemos añadirle detrás en el bloque de efectos, o enrutándolo desde otra pista, un instrumento virtual (VSTi) como por ejemplo un piano en el caso de la figura que se muestra a continuación.

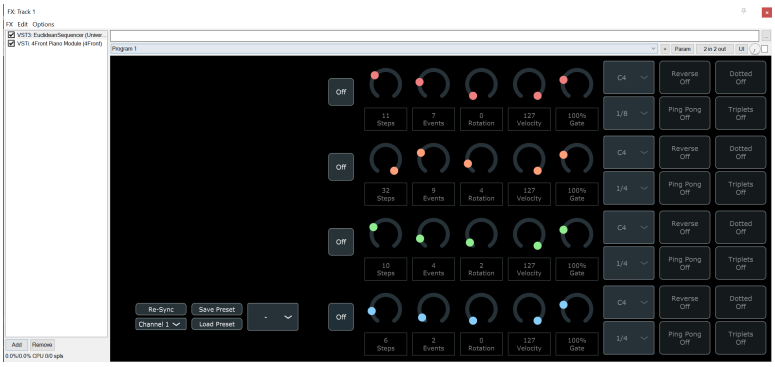

Figura 3.3: *Bloque de efectos de una pista en REAPER*

3.2. Estructura del plugin

Tras su correcta instalación y despliegue en la DAW, y con el fin de cualquier usuario sepa utilizar el plugin de una manera correcta tras este bloque, se proceden a explicar las funcionalidades que tienen cada uno de los componentes que componen este plugin.

3.2.1. Representación gráfica de los ritmos

Como se puede observar en la Figura 3.3, al iniciar el plugin, y sin haber ritmo alguno creado, no hay ninguna representación gráfica. Sin embargo, a medida que vamos creándolos, estos se van representando de manera concéntrica en la parte de la izquierda de la GUI. Cada ritmo está representado por un número de pulsos y de silencios, donde los pulsos están representados por segmentos coloreados de un color distinto en cada ritmo, mientras que los silencios están representados por segmentos más pequeños de color gris. Por último, destacar que en blanco se resalta el paso del compás que está sonando en cada ritmo, como se ve en la siguiente figura:

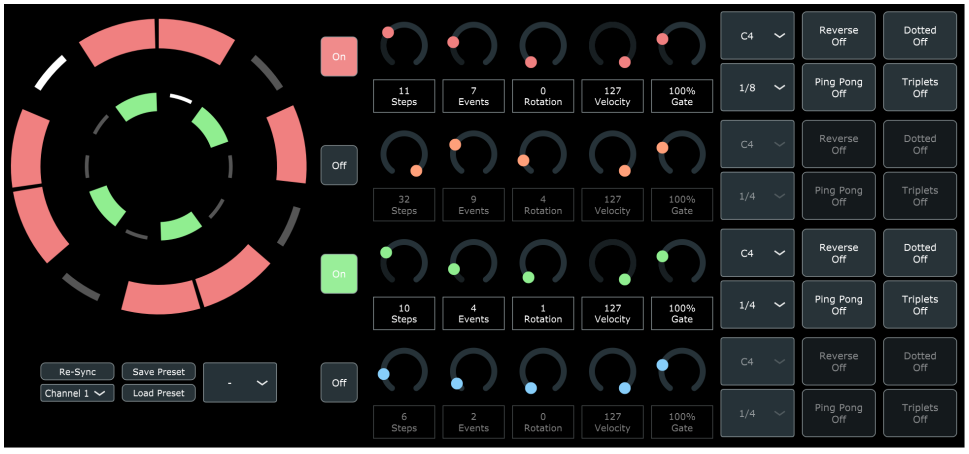

Figura 3.4: *Representación de dos ritmos*

3.2.2. Componentes individuales de cada ritmo

Como se puede ver en anterior figura, el plugin se puede dividir, sin contar la parte izquierda del mismo, en cuatro secciones que son iguales para los cuatro ritmos ya que todos sus componentes son análogos. Estos, de izquierda a derecha, son:

- ***On/Off button***: Con este botón crearemos un ritmo, de forma que será representado gráficamente, y tendremos la posibilidad de interactuar con sus componentes, cosa que no es posible cuando este botón está en *Off*. En la Figura 3.4 podemos observar que los ritmos 1 y 3 tienen sus botones en *On* y todos sus componentes habilitados para interactuar con ellos, mientras que con los ritmos 2 y 4 pasa lo opuesto.

- ***Sliders***: Este conjunto de 5 *sliders* rotativos nos permiten manejar los parámetros de *steps, events, rotation, velocity* y *gate*. La función de cada uno de estos *sliders* es la de indicar, respectivamente, el número total de pasos del ritmo, el número de eventos o pulsos, las unidades de rotación que se le aplican al ritmo (en sentido horario), la intensidad de las notas (siendo 0 una intensidad nula o silencio, y 127 la intensidad máxima) y por último el porcentaje de cada *step* que será ocupado por las notas, donde 0 % sería el valor mínimo, 100 % sería que las notas representasen la misma figura musical que los *steps*, etc...

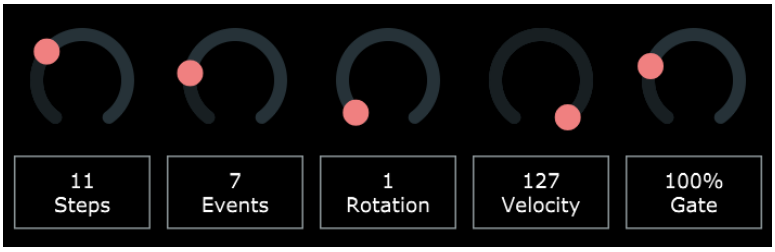

Figura 3.5: *Sliders*

- ***NoteNumber* y *figureStep***: Estas dos *comboBoxes* nos dan la opción de elegir que nota musical queremos que suene en el ritmo y la figura musical representada por cada *step* (tiempo que dura). De esta manera podemos escoger, usando el sistema de notación musical anglosajón, cualquier nota del abanico C0-B6 y cualquier figura musical entre 1 (redonda) y 1/64 (semifusa).

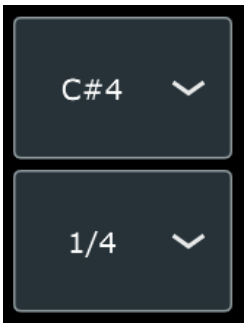

Figura 3.6: *NoteNumber comboBox y stepFigure comboBox*

- **Botones de *reverse, ping-pong, dotted notes* y *triplets***: Los dos primeros botones compiten por la dirección del ritmo, es decir, si ninguno está pulsado (ambos en *Off*) el ritmo ira será en sentido horario. Sin embargo, si *reverse* está en *On*, el sentido será el inverso y si es el botón de *ping-pong* el que está en *On*, cada vez que el ritmo termine (llegue al principio del compás) cambiará de sentido rebotando. Por su parte los botones de *dotted* y *triplets* nos ofrecen la posibilidad de hacer que las notas del ritmo se conviertan en tresillos[10], o en notas con puntillo[7].

Figura 3.7: *Ejemplo con el botón reverse activado*

3.2.3. Otros componentes del plugin

Además de los componentes comunes a los cuatro ritmos del plugin, tenemos en la parte inferior inzquierda otro grupo de componentes que se dedican a desarrollar las siguientes funciones:

- ***Channel Selector***: Esta *comboBox* nos permite seleccionar el canal MIDI de salida que va a ser usado de entre los canales 1 a 16.

Figura 3.8: *Ejemplo con el botón reverse activado*

- ***Re-sync***: Este botón nos permite sincronizar todos los ritmos que haya activos en ese momento. Lo que lleva a cabo este botón realmente es hacer que todos los ritmos empiecen su compás desde el principio, es decir, que se coloquen en su primer paso.

Figura 3.9: *Botón re-sync*

- ***Save/Load preset***: Estos dos botones nos dan la opción de guardar o cargar un preset indicando en la *comboBox* de su derecha el hueco que desean guardar/cargar. Por defecto vemos un guión indicando que no se ha seleccionado ninguna de las 8 posibilidades existentes. Cabe destacar que la ruta de la ubicación de estos *presets* es por defecto */Documents/Euclidean Sequencer/presets/*.

Figura 3.10: *Save/Load Preset and presets' comboBox*

3.3. Ejemplos de uso

Para acabar este capítulo, queríamos mostrar varios ejemplos relacionados con las funcionalidades de guardar y cargar *presets*.

- **Cargar *preset***: En esta operación nos podemos encontrar con 3 escenarios posibles:

  1. **Carga exitosa**: Se produce cuando, tras seleccionar el preset deseado en la *comboBox*, este existe, y es cargado en el plugin con éxito como nos muestra el cuadro de diálogo.

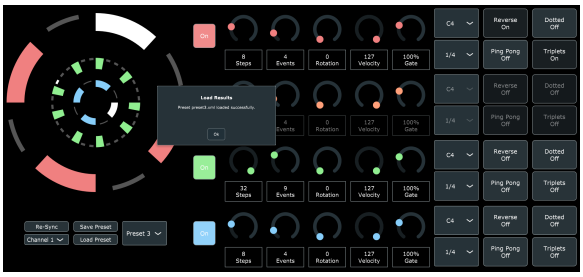

Figura 3.11: *Preset cargado con éxito*

2. **Error - Seleccione Preset**: Producido porque se está intentando cargar un preset sin antes haber sido seleccionado ninguno en la *comboBox*. El cuadro de diálogo indicará que se debe seleccionar un preset.

Figura 3.12: *Error. Seleccione preset para cargar*

3. **Error - Preset Inexistente**: Producido porque se está intentando cargar un preset seleccionado en la *comboBox* que no existe en el sistema, es decir, que ese slot esta vacío. El cuadro de diálogo indicará más detalles.

Figura 3.13: *Error. El preset seleccionado no existe*

- **Guardar preset**: En esta operación nos podemos encontrar con 2 escenarios posibles:

1. **Guardado exitoso**: Se produce cuando, tras seleccionar el preset deseado en la *comboBox*, se guarda el estado del plugin de manera correcta en la ubicación adecuada como se muestra en el cuadro de diálogo.

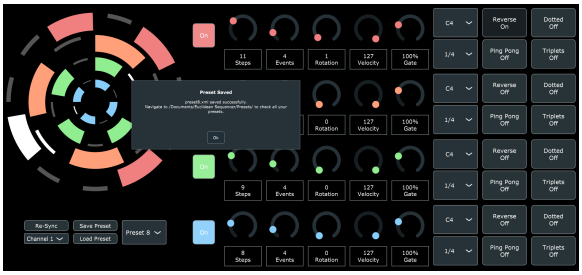

Figura 3.14: *Preset guardado con éxito*

2. **Error - Seleccione Preset**: Producido porque se está intentando guardar un preset sin antes haber seleccionado el slot o hueco en la *comboBox*. El cuadro de diálogo indicará que se debe seleccionar un slot donde guardar el preset de los 8 posibles.

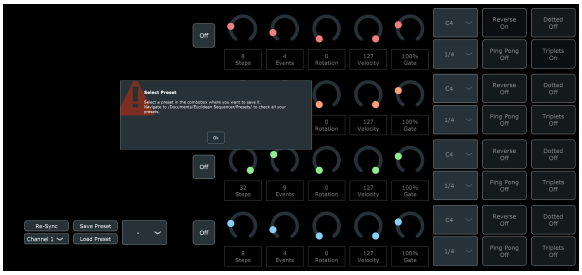

Figura 3.15: *Error. Seleccione slot donde guardar el preset*

Capítulo 4

Implementación

En este capítulo se tratarán los conceptos que conforman el algoritmo del secuenciador euclidiano, así como el software que ha sido empleado para desarrollar el plugin, y el código usado para implementar las distintas funcionalidades. Con el fin de tener un repositorio común donde todos los autores pudiésemos acceder para descargar y subir los archivos necesarios que se fueran desarrollando, y además llevar un claro control de versiones, el software escogido ha sido *GitHub*. En el siguiente repositorio se pueden encontrar todos los que forman el proyecto:[5].

4.1. Conceptos básicos

Al activar cualquiera de las distintas etapas del plugin, esta empezará a generar mensajes MIDI en función de los inputs que introduzca el usuario. La funcionalidad básica la forman dos parámetros: el número de pasos, que indica el total de notas que tiene un compás y los eventos, que indica de estos pasos, cuántos han de sonar y cuantos no (los pulsos del compás). El algoritmo distribuye las notas que suenan y las que no de manera automática, de acuerdo a la definición de ritmo euclidiano.

4.2. JUCE

JUCE es un marco para el desarrollo de aplicaciones multiplataforma orientadas al audio y la música de código abierto y desarrollado en C++. Cuenta con una extensa colección de clases con funcionalidades enfocadas al procesamiento de audio, interfaz de usuario, gráficos y conversión a JSON/XML entre otras, sobresaliendo en el apartado de procesamiento de audio debido a la gran cantidad de funciones que tiene para ello.

De todas las utilidades que tiene, la más interesante de cara a este trabajo es la de desarrollar plugins. Cuenta con soporte para crear plugins con todos los formatos estándar (VST, VST3, AU, AUv3, RTAS o AAX), siendo VST3 el empleado para este proyecto, ya

que se trata de uno de los más eficientes en cuanto al consumo de CPU y es compatible con los principales sistemas operativos.

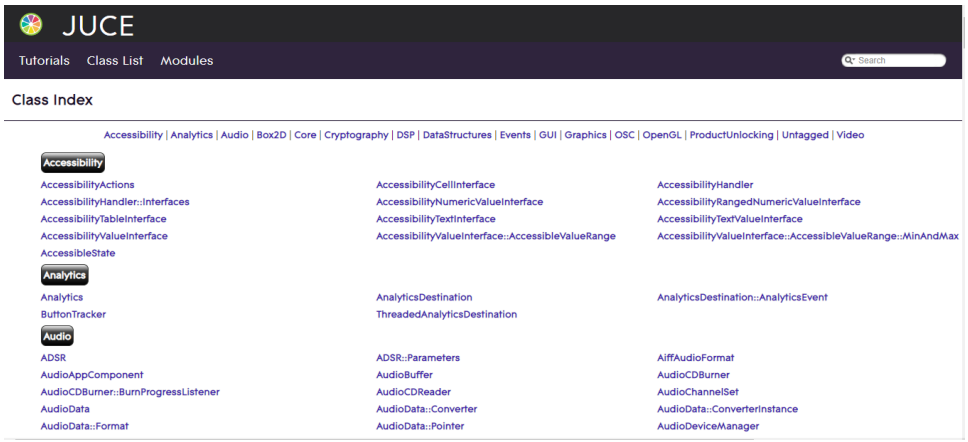

Figura 4.1: *Pequeño extracto del Class index con todas las funcionalidades de JUCE*

4.2.1. Projucer

Projucer es la herramienta de generación de proyectos de JUCE. Su utilidad principal recae en su capacidad de exportar proyectos de JUCE a un IDE, del inglés Entorno de Desarrollo Integrado. Una vez exportados los archivos y configuraciones del proyecto, Projucer genera de manera automática los archivos necesarios para que el proyecto se pueda compilar en la plataforma objetivo. Cuenta con compatibilidad para XCode, Visual Studio, Codeblocks y Linux Makefiles.

4.2.2. Estructura de un plugin de audio en JUCE

Un plugin básico de JUCE cuenta con dos clases: *PluginProcessor* y *PluginEditor*.

PluginProcessor es la clase que se encarga de procesar todos los mensajes MIDI que va a ir generando el plugin. Cuenta con los siguientes métodos para ello:

- ***prepareToPlay()***: este método se llama nada más arrancar el plugin, y en él se incializan todas las variables que sean necesarias para el funcionamiento del mismo.
- ***processBlock()***: este método se llama en intervalos de pocos milisegundos y es el encargado de realizar la generaración y el procesamiento de eventos MIDI que se escriben en el buffer MIDI de salida, además de procesar también las señales de audio que le llegan al plugin y devolverlas en el buffer de salida de muestras de audio, es decir,

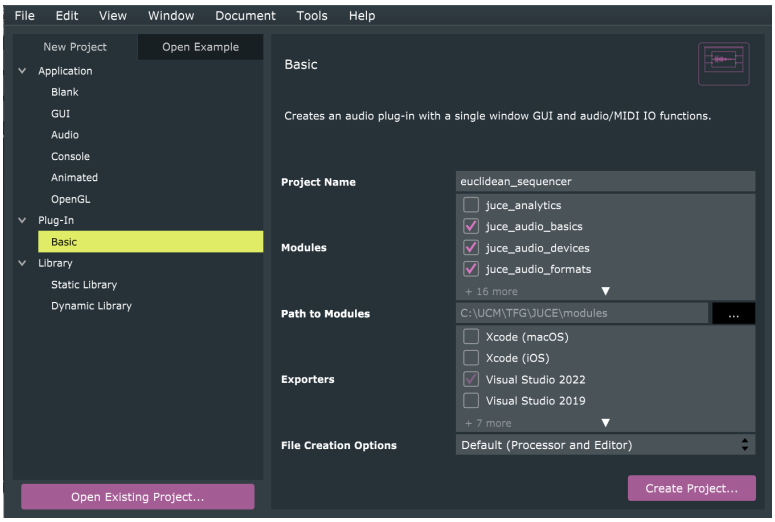

Figura 4.2: *Comienzo de un nuevo proyecto en Projucer*

es la función principal de nuestro procesador. Fijándonos en la Figura 4.4 veamos un sencillo ejemplo donde esta función variará el volumen de la señal de audio que le llegue mediante un parámetro controlado por el usuario:

```
void GainTutorial1AudioProcessor::processBlock (AudioSampleBuffer& buffer, MidiBuffer&
    midiMessages)
{
    const int totalNumInputChannels  = getTotalNumInputChannels();
    const int totalNumOutputChannels = getTotalNumOutputChannels();

    rawVolume = 0.015;

    for (int i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
        buffer.clear (i, 0, buffer.getNumSamples());

    // This is the place where you'd normally do the guts of your plugin's
    // audio processing...
    for (int channel = 0; channel < totalNumInputChannels; ++channel)
    {
        float* channelData = buffer.getWritePointer (channel);

        for (int sample = 0; sample < buffer.getNumSamples(); ++sample)
        {
            channelData[sample] = buffer.getSample(channel, sample) * rawVolume;
        }


    }
}
```

Figura 4.3: *Sencillo ejemplo del funcionamiento de processBlock*

Para este ejemplo solo tendremos en cuenta el parámetro de entrada *buffer*, que como hemos comentado, es el encargado de procesar las señales de audio digital. Como vemos al principio del algoritmo, lo primero que se hace es recoger información sobre el número de canales de salida y entrada a tener en cuenta.

Tras esto, y a modo de buenas prácticas, ponemos el cociente de variación de volumen

rawVolume a un nivel adecuado por defecto, además de vaciar el buffer de salida asegurándonos así de que no haya señales residuales.

A continuación, iteraremos sobre todos los canales de entrada que tenemos y lo que haremos será, para cada uno de ellos, iterar sobre todos los samples que componen la señal de dicho canal y multiplicar cada uno de estos samples por el cociente de variación de volumen *rawVolume*. Con esto lo que conseguimos es variar el volumen de cada señal que nos entra sample a sample por cada canal.

Por último, cabe destacar que en este caso, el valor *rawVolume* iría ligado a algún componente del PluginEditor encargado de recoger la entrada del usuario.

PluginEditor es la clase encargada de mantener la interfaz del plugin, se encarga tanto de leer el input del usuario mediante los diferentes componentes (sliders rotativos, cajas de elección y botones), como de generar los gráficos de forma que se visualice el ritmo que se está generando. Para proporcionar y recibir información del procesador, esta clase cuenta con un campo de tipo *PluginProcessor*. Sus métodos esenciales son:

- ***paint()***: este método se llama cada vez que es necesario redibujar alguna parte del plugin y se encarga de la generación de los gráficos del plugin.
- ***resized()***: se llama cada vez que las dimensiones del plugin varían, y se encarga de la colocación de los distintos elementos que componen la interfaz.

4.3. Clase EuclideanRhythm

A parte de la estructura principal que todo plugin de audio desarrollado en JUCE debe tener, en nuestro caso tenemos además una clase llamada *EuclideanRhythm*, cuya función es representar un ritmo euclidiano una vez es creado (como veremos a continuación) almacenando todos los parámetros necesarios para su correcto funcionamiento en forma de atributos, además de gran cantidad de funciones que permiten tanto al *PluginProcessor* como al *PluginEditor* interactuar con este pudiendo recibir información para saber en qué estado se encuentra y modificar parámetros del mismo, alterando sus características.

4.3.1. Atributos

Entre muchos otros, los atributos más destacables por su importante relación tanto con la parte que procesa el plugin, como con la que mantiene su interfaz son:

- ***Steps*, *events* y *rotation***: Estos valores enteros nos indican el número de pasos totales, eventos o pulsos, y rotación del ritmo respectivamente. Abarcan desde 0 a 32 excepto los steps, cuyo valor mínimo es 1.

- ***EuclideanRhythm***: Esta secuencia de 1s y 0s es la utilizada para almacenar el ritmo donde los 1s marcan los eventos y los 0s los silencios.

- ***Direction*, *reverse* y *pingPong***: Se encargan, mediante valores booleanos, de indicar el sentido hacia el cuál el ritmo es interpretado. *Direction* = *true* indicará que el ritmo va en sentido horario, hasta que bien se ponga *reverse* = *true*, donde hará que el sentido se invierta, o bien *pingPong* tome el valor de *true* con lo que el ritmo, cada vez que llegue a su primer *step*, invertirá su sentido.

- ***NoteNumber*, *velocity***: Encargados de, por medio de valores enteros, indicar el valor de las notas MIDI del ritmo y la intensidad de las mismas (valores de 0-127 donde 0 sería intensidad nula o silencio).

- ***FigureStep* y *gate***: *FigureStep* es un *float* que, conectado a una *comboBox* del *PluginEditor*, nos da la posibilidad de elegir las figuras musicales de los pasos del ritmo (redondas - 1, blancas - 1/2, negras 1/4, etc...). Por otra parte, gate es un entero que nos permite decidir que porcentaje de esa figura (en tiempo) queremos que suene. Oscila entre un 0 y un 400 %, donde 0 % sería un silencio, y un 100 %, el tiempo correspondiente a la figura elegida.

- ***Triplets* y *dottedNotes***: Estos dos booleanos nos dan la posibilidad de hacer que las notas de nuestro ritmo se conviertan en tresillos o en notas con puntillo.

Además de estos parámetros que podemos modificar directamente interactuando con distintos componentes de la interfaz, cabe destacar, por su utilidad e importancia a la hora de hacer el procesamiento, los siguientes atributos:

- ***Index*, *bpm*, *sampleRate***: El valor entero index nos indica la posición de la nota (el paso) del atributo *euclideanRhythm* anteriormente mencionado que se está procesando. El *bpm* y el *sampleRate* por su parte son valores recogidos de la DAW que nos indican la cantidad de pasos por minuto, o velocidad de reproducción el proyecto. Por otra parte el sampleRate nos indica la frecuencia de muestreo del mismo, es decir, el número de muestras por unidad de tiempo que se toman de una señal, normalmente 44100Hz (44100 muestras de la señal en 1s).

- ***stepDuration* y *timeStep***: Estos enteros representan respectivamente, la duración de un step en número de samples y el tiempo en número de samples que lleva procesándose ese step. (mismos dos parámetros respectivamente para las notas, *noteDuration* y *timeNote*, que dependiendo del valor de gate coincidirán o no)

- ***NumSamplesPerBar* y *currentSamplerPerBar***: Mediante dos enteros, expresan el número total de samples que tiene el ritmo, y el número del sample que acabamos de procesar, lo que nos sirve para ver por qué parte del ritmo vamos.

- ***notesDurationMap***: Por último, pero no menos importante, tenemos este mapa de <int, int>donde el primer valor identifica una nota MIDI y el segundo el número de samples que lleva sonando. Esto nos ayuda a mantener las notas MIDI sonando o no, de manera que cuando es el turno de un evento o pulso, se mandará el evento MIDI noteOn de la nota correspondiente al *buffer* MIDI de salida y se añadirá a este mapa. Esta parará de sonar cuando su segundo valor, el correspondiente al número de samples que lleva sonando, llegue al máximo (*timeNote* alcance el valor de *noteDuration*), mandando un evento MIDI noteOff al *buffer* MIDI de salida.

4.3.2. Funciones

Tras haber destacado los atributos fundamentales de esta clase, comentaremos las dos funciones más importantes que, juntos con todos los *getters* y *setters* que la componen, hacen que la clase *EuclideanRhythm* tenga una funcionalidad óptima:

- ***set_ euclideanRhythm(int steps, int events)***: Esta función es la encargada de, en función en número de steps y events que reciba, producir la secuencia de 1s y 0s que conforman nuestro ritmo (atributo *euclideanRhythm*). Además, dependiendo del atributo *rotation*, se le aplicará una rotación u otra (por defecto este valor es 0, a no ser que el usuario lo haya modificado desde la interfaz).

```
void EuclideanRhythm::set_euclideanRhythm(int steps, int events) {
    calculateEuclideanRhythm(steps, events);
    rotateRight(this->_rotation);
}
```

Figura 4.4: *Función set_ euclideanRhythm(int steps, int events)*

Esto es posible, tal y como vemos en la Figura 4.4, gracias a dos funciones auxiliares que dividen este proceso en dos etapas:

- ***calculateRhythm(steps, events)***: Esta primera etapa del proceso consiste en obtener el ritmo euclidiano representado como una secuencia de 1s y 0s, donde los 1s (pulsos del ritmo) quedarán tan equidistantes como sea posible. Para conseguirlo, en lugar de usar el algoritmo que *Godfried Toussaint* desarrolló en su paper con este cometido, y que en capítulos previos ha sido explicado, hemos decidido usar el algoritmo de *Bresenham*[1], el cual permite obtener los mismos objetivos pero, desde nuestro punto de vista, con una implementación más sencilla.
- ***rotateRight(this->_rotation)***: La segunda etapa, tras haber obtenido la secuencia de 1s y 0s que representan el ritmo euclidano, consiste en aplicarle el número de rotaciones que sea pertinente, tomando como parámetro de entrada el valor del atributo *rotation* de la propia clase (nótese que, si es 0, la secuencia de 1s y 0s quedaría igual).

- ***convertBPMToTime()***: Esta función es también fundamental ya que es la que obtiene, desde los *bpm* y el *sampleRate* del proyecto, la duración en samples de los *steps* y de las notas (la duración en samples de las notas también depende del gate como hemos comentado antes):

$$stepDuration = \frac{60 * sampleRate * figureStep}{bpm}.$$

$$noteDuration = \frac{60 * sampleRate * (figureStep * \frac{gate}{100})}{bpm}.$$

4.4. PluginEditor

Como ya se ha descrito anteriormente, esta es la clase encargada de mantener la interfaz, manejando el control de la misma y controlando lo que se está mostrando por pantalla.

4.4.1. Atributos

Los atributos esenciales para el funcionamiento de esta clase son los siguientes:

- ***audioProcessor***: Este atributo es una referencia al *PluginProcessor*. De este modo se tiene una manera rápida y eficiente de acceso a los atributos y funciones del *PluginProcessor*, que, como vamos a ver, ha creado a este objeto *PluginEditor*.

- ***Buttons, sliders* y *comboBoxes***: Contienen la lógica de los botones, *sliders* y demás elementos de la interfaz con los que el usuario puede interactuar. Los elementos que son particulares para cada una de las etapas del secuenciador, como por ejemplo el número de pasos o de eventos, están mantenidos en un mapa, con una entrada para cada una de las cuatro etapas del secuenciador, obteniendo así un código más limpio y fácilmente escalable.

- ***Attachments***: Por cada elemento de lode recien nombrados que corresponden a una etapa (o ritmo) del secuenciador, existe su *Attachments* correspondiente. La función de los *Attachment* es la de ligar cada elemento de la interfaz con un parámetro de la estructura *AudioProcessorValueTreeState*. Esta estructura mantiene el estado completo del plugin en todo momento, y es útil tanto para almacenar configuraciones de parámetros, como para pasar valores al *PluginProcessor*.

4.4.2. Funciones

A continuacion se van a describir las funciones más relevantes que conforman el *PluginEditor*:

- ***EuclideanSequencerAudioProcessorEditor(EuclideanSequencerAudioProcessor&)***: Este es el constructor de la clase. En él se crean todos los mapas que alojarán a botones, *sliders* y *comboBoxes* que van a conformar la interfaz. Además, también se ligan a sus respectivos *Attachments* y una vez inicializados con los valores correspondientes según su función.
- ***paint(juce::Graphics& g)*** y ***paintRhythm(juce::Graphics& g, int seqID, float innerCircleProp)***: El fin de esta función es el de pintar la representación gráfica de los ritmos que haya activos. Tiene un bucle en el que, por cada etapa del secuenciador (máximo 4), coge uno de los *mutex* para que no haya corrupciones en memoria u otros errores, y tiene el correspondiente, llama a la función *paintRhythm*. Dicha función es la que realmente se encarga de representar gráficamente los ritmos, en función de los pasos, eventos y demás parámetros que conforman cada ritmo. Tras esta función el *mutex* queda liberado para que no hayas bloqueos en el plugin.

```
void EuclideanSequencerAudioProcessorEditor::paint(juce::Graphics& g)
{
    // (Our component is opaque, so we must completely fill the background with a solid colour)
    g.resetToDefaultState();
    g.fillAll(juce::Colours::black);

    float innerCircle = 0.8f;
    for (int seqID = 0; seqID < NUM_TOTAL_ETAPAS; seqID++) {

        audioProcessor.my_mutex[seqID].lock();

        if (!audioProcessor.getEuclideanRhythms().count(seqID)) {
            audioProcessor.my_mutex[seqID].unlock();
            continue;
        }

        paintRhythm(g, seqID, innerCircle);
        innerCircle -= 0.05f;
        audioProcessor.my_mutex[seqID].unlock();

    }

    repaint();
}
```

Figura 4.5: *Función paint(juce::Graphics& g))*

- ***setXParams***: Este grupo de funciones son las encargadas de inicializar los distintos elemtentos interactivos de la interfaz. Por ejemplo, *setNoteNumberComboBoxParams* es la función dar los valores iniciales de las *ComboBoxes* que determinan la nota que suena en cada ritmo.
- ***resized()***: Esta función es la encargada de establecer las posiciones de los distintos elementos que componen la interfaz.
- ***buttonClicked()* y *comboBoxChanged()***: Son los *listeners* de los distintos botones y *comboBoxes*. Tras reconocer cual ha sido el elemento con el que se ha interactuado, se encargan de hacer los cambios o llamar a la función apropiada de *PluginProcessor*.
- ***disableComponents(int id)* y *enableComponents(int id)***: Activan y desactivan los componentes de cada etapa del secuenciador. El parámetro id indica de qué etapa se trata de las 4 que hay.

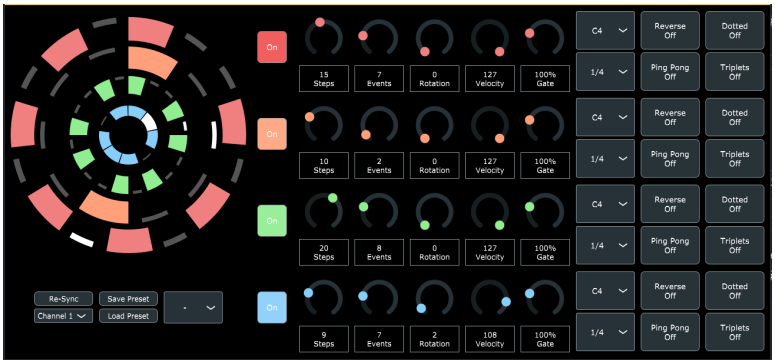

Figura 4.6: *Versión final del secuenciador euclideano*

4.5. PluginProcessor

Esta clase, como hemos dicho anteriormente, es el esqueleto de la gran mayoría de plugins desarrollados con *JUCE* ya que es la encarga de manejar la entrada y salida de audio y MIDI, además de procesar toda la lógica del plugin.

4.5.1. Atributos

- ***euclideanRythms***: Este mapa de *<int, EuclideanRhythm*>* es el encargado de almacenar los ritmos del plugin (máximo 4 ritmos).
- ***playHead***: Este atributo del tipo *AudioPlayHead** es el encargado de proporcionarnos información de la DAW como por ejemplo los bpm a los que está trabajando.
- ***notesOn***: Con la ayuda de este mapa del tipo *<int, int>* sabemos cuándo realmente debemos lanzar un evento MIDI *noteOff* al *buffer* MIDI de salida. Esto es posible ya que el primer parámetro corresponde a una nota MIDI y el segundo, al número de ritmos en los que está sonando, de manera que cuando este segundo parámetro sea igual a 0, se debe mandar dicho evento MIDI *noteOff* de la nota correspondiente.
- ***AudioProcessorValueTreeState***: Usando este tipo de objeto conseguimos manejar el estado del plugin de la manera más óptima posible. Las ventajas principales que nos proporcionan su uso son:
  - Nos da la posibilidad de serializar y deserializar (en formato XML), de forma que podemos guardar y cargar el estado del plugin y de cada uno de sus parámetros en cualquier momento.
  - Se pueden tener los *listeners* de los componentes que queramos unidos a él, de modo que pueda conectarse casi automáticamente a los *sliders*, botones y demás componentes para mantener actualizado el estado de la interfaz de usuario y el procesador.

- ***mutex***: Esta lista de *mutex* (máximo 4) nos permite llevar un control sobre las hebras del plugin, de manera que las modificaciones de una u otra no afecten a la integridad del plugin mediante corrupciones de memoria u otros errores que puedan darse.

4.5.2. Funciones

Una vez que sabemos los atributos fundamentales de esta clase, pasaremos a ver la funcionalidad de los métodos más importantes que la conforman:

- ***EuclideanSequencerAudioProcessor()***: Es el construcor de la clase, donde lo destacable es que se inicializa nuestro objeto de estado *AudioProcessorValueTreeState*.
- ***PrepareToPlay()***: Es la función de encargada de darle un valor inicial a aquellos atributos que lo necesiten para el correcto funcionamiento del plugin.
- ***createEditor()***: Se encarga de crear una instancia del *PluginEditor* llamando a su constructor.
- ***processBlock(juce::AudioBuffer<float> buffer, juce::MidiBuffer midiMessages)***: Esta es la función principal, que se ejecuta una gran cantidad de veces por segundo, donde se lleva a cabo el manejo de entrada y salida de audio y MIDI además de el procesamiento de la lógica. Su función principal en este plugin es, por cada uno de los ritmos que haya en el mapa *euclideanRhtyms*, llamar a la función *processSequencer*.
- ***processSequencer(juce::MidiBuffer midiMessages, EuclideanRhythm* euclideanRhythm, int sequencerID)***: Esta función es la realmente encargada de manejar la lógica de cada ritmo *euclideanRhythm* con ID *sequencerID*. Nada más ser llamada, esta función recoge distintos parámetros de los *inputs* de la interfaz, a través de la estructura *AudioProcessorValueTreeState* y los guarda en el objeto *euclideanRhythm* correspondiente. Para guardar los parámteros es encesario coger un *mutex*, al estar la llamada a esta función entrelazada con la ejecución de la función *paint*, que accede a valores comunes. Una vez actualizados estos valores, el mutex queda libre.

  A continuación, se comprueba si es necesario rotar el ritmo y, en caso de serlo, se realiza esta operación. La siguiente operación consiste en calcular el número de sample por el que va el ritmo actual. Si el número de pasos no ha cambiado, este número no se altera. En caso contrario hay que calcular por donde iría. Cabe destacar que hemos conceptualizado el compás como una circunferencia y la posición en el ritmo como una aguja centrada en la circunferencia que se va desplazando por ella. El fin de esto es que si la aguja lleva media circunferencia recorrida, al cambiar los pasos, su posición debería seguir estando en la mitad de la circunferencia independientemente de lo que dure el nuevo compás. Una vez está calculado el número de sample, se actualiza también el índice del paso que está sonando. Por ejemplo, si el ritmo tenía 4 pasos y el índice (o la aguja) está en el paso 2, si cambiamos a un compás de cuatro pasos, el

índice quedaría en el paso 4.

La siguiente operación a realizar es comprobar si, en el tiempo número de samples que procesa esta llamada de la función, se acaba alguna nota, mandando un MIDI *noteOff* a la nota correspondiente. Para ello basta con recorrer el mapa de duración de notas de la secuencia y comprobar si los samples que le quedan por sonar a cada nota son menores que los que se procesan en esta llamada.

A esto le sigue comprobar si se alcanza un nuevo paso en esta llamada de la función y programar el MIDI *noteOn* correspondiente. El procedimiento es bastante similar al de comprobar si la duración de las notas se acaba.

Finalmente comprobamos el sentido en el que está corriendo el ritmo y actualizamos el valor de la aguja con la nueva posición que tendrá una vez se hayan procesado todos los samples de esta llamada, además de actualizar el tiempo que llevan las notas que siguen sonando.

- ***createParameters()***: Esta función es la encargada de ligar cada componente del *PluginEditor* con el *PluginProcessor* haciendo uso del *AudioProcessorValueTreeState* mediante identificadores únicos.

Además de estas funciones, hay otras muchas que hacen que el funcionamiento interno de todas estas, y del plugin en general, sea el correcto. Destacando por ejemplo los *setters* de muchos parámetros de los ritmos que son llamados por los *listeners* del *PluginEditor*, las funciones de *loadPreset y savePreset* que guardan o cargan el *preset* correspondiente en el directorio */Documents/Euclidean Sequencer/Presets/* de cada usuario o la función *synchronizeAll()* que hace que todos los ritmos activos empiecen su compás desde el principio.

Capítulo 5

Contribuciones al proyecto

Como hemos comentado anteriormente, este trabajo ha sido desarrollado por un grupo compuesto por dos estudiantes de ingeniería informática y amantes de la música. Estos integrantes somos David Burgos Díaz y Gonzalo Costales de Ledesma, quienes sin conocernos de nada antes de la realización de este proyecto, hemos acabado formando una buena relación tanto profesional como de amistad. En este capítulo, y a pesar de que el desarrollo de este trabajo de fin de grado ha sido casi en su totalidad de manera síncrona, ya que hemos trabajado en casi todas las sesiones de manera simultánea (físicamente o de manera remota haciendo videollamadas), comentaremos con detalle cuáles han sido las contribuciones de cada uno.

5.1. David Burgos Díaz

Para empezar, creo que es relevante comentar el punto en el cual se encontraban mis conocimientos tanto sobre audio digital, DAWs, plugins, etc... como de teoría musical, ya que durante este trabajo han ido crecido exponencialmente tanto estos conocimientos como la curiosidad y las ganas de seguir explorando otros conocimientos relacionados con estos campos.

Al inicio del curso, cuando decidimos embarcarnos en este viaje, mis conocimientos relacionados con el audio digital, las DAWs y los plugins eran prácticamente nulos. Algo había aprendido de amigos que se dedican a la producción musical (herramientas básicas como los secuenciadores de pasos y alguna funcionalidad sencilla) pero podría decir que no tenía idea más allá de hacer sonar música y mezclarla a nivel básico con una mesa de mezclas. Además, en cuanto a teoría musical mis conocimientos eran los conceptos básicos que habría adquirido por curiosidad en algún vídeo de YouTube. Fue por esto que lo primero que hice tras elegir que mi TFG iba a estar relacionado con la música, fue matricularme en la asignatura optativa de informática musical. Esta decisión puedo decir con seguridad que fue una de las mejores de la carrera ya que, a parte de terminar de despertar mi curiosidad por el mundo de la informática musical, me metió de lleno en él.

Durante la primera parte de esta asignatura, impartida por el Dr. Jaime Sánchez, mis concocimientos empezaron a ver nuevos horizontes con la introducción a los fundamentos del audio digital. En esta primera etapa, y tras un poco de historia que me puso en contexto, aprendí lo que es el proceso de muestreo y los parámetros que son necesarios para que este se lleve a cabo, como por ejemplo la frecuencia de muestreo o el número de bits por muestra. Tras esto, empecé a familiarizarme y trabajar con otros conceptos relacionados con el procesado o manipulación del sonido, donde se pueden destacar la amplitud, la frecuencia, el timbre, el tono, etc... Una vez teniendo estos conceptos afianzados, Jaime nos enseñó, después de haber jugado con todos ellos haciendo uso de herramientas de edición de audio como *Audacity*[3], a programar audio digital modelando todos estos parámetros. Esta fue una etapa muy enriquecedora ya que terminé de entender el funcionamiento de los generadores y de muchos efectos en el dominio tiempo-amplitud como el de compresión o las envolventes *Fade-In* y *Fade-Out* entre muchos otros. Tras esto, comenzó la última etapa de esta primera parte de la asignatura, donde terminamos de estudiar y programar otros conceptos como el de ecualización, reveberación, filtros como el paso-banda entre otros, el sintetizador, el sampler o la mesa de ondas, haciendo uso de *Python* y sus numerosas librerías dedicadas a este campo, entre las que destaca *pyAudio*.

Figura 5.1: *Logo de la librería pyAudio usada para programar audio digital en Python*

Tras esta parte de la asignatura en la cual afiancé y entendí muchos conceptos que me estaban ayudando paralelamente a investigar y comprender nociones sobre el secuenciador euclidiano, empezó la segunda etapa impartida por nuestro tutor del trabajo de fin de grado, el Dr. Miguel Gómez-Zamalloa. Este período de la asignatura a su vez estuvo dividido en dos:

- Un primer período donde Miguel nos enseñó de una manera impecable lo que es la producción musical; empezando por lo que es el MIDI y lo que son los secuenciadores, y acabando con la automatización de parámetros en una DAW (en este caso, la DAW utilizada fue REAPER). Durante todo este proceso, por supuesto pasamos por lo que es JUCE y todo lo que nos ofrece, los efectos de audio, los instrumentos virtuales, etc... Fue en esta etapa donde realmente el trabajo de fin de grado empezó a tomar

forma con todo el conocimiento mucho más específico que estaba adquiriendo, y con la cantidad de reuniones físicas (a partes de las remotas vía videollamada) que tenía con Miguel. Esto era debido a que dos veces por semana asistía a sus clases, por lo que al final de las mismas podía quedarme hablando con él sobre las inquietudes y dudas acerca de nuestro trabajo para posteriormente comunicárselas también a Gonzalo y trabajar conjuntamente sobre ellas.

- Tras este proceso de aprendizaje y familiarización con la producción musical, las DAWs, el proceso de mezcla y demás conceptos relacionados con esta cuestión, comenzó la última etapa de la asignatura y más complicada para mi debido a mis escasos conocimientos en el campo. Esta etapa consistió en estudiar la teoría musical básica y los principios de composición. A la par que complicada me pareció súper interesante ya que, como le dije a Miguel en su día, me enseñó a ver y entender la música desde un punto de vista que no conocía, y me motivó a seguir indagando en ello. Durante este período aprendí conceptos muy interesantes como el de notas, intervalos, escalas, acordes, armonía básica o ritmos entre otros. Aplicándolo al trabajo de fin de grado, me sirvió por ejemplo para entender y saber implementar las funcionalidades del tresillo y de las notas con puntillo.

Figura 5.2: *Logo de REAPER*

Habiendo explicado mi proceso de aprendizaje de la mano de Jaime y de Miguel en la asignatura de informática musical y entendiendo el contexto en el que estaba en cuanto a mis conocimientos, a continuación explicaré cuáles fueron mis contribuciones al trabajo de fin de grado.

Como he comentado, la primera etapa de nuestro trabajo, coincidió con la primera etapa de la asignatura, en la que, tanto Gonzalo como yo, estuvimos descubriendo como funcionaba JUCE. Durante esta estapa de investigación y familiarización con este *framework* desarrollé algunos plugins muy sencillos siguiendo los distintos tutoriales que hay en la web de JUCE o los que tiene en su canal de YouTube el anteriomente mencionado *The Audio Programmer*. Entre ellos están el del controlador de volumen y frecuencia, con el que aprendí a controlar audio, y una variación de arpegiador, con el que aprendí a manejar eventos MIDI.

Tras este proceso de enriquecimiento, fue hora de ponerse manos a la obra y empezar a trabajar de manera conjunta con Gonzalo en nuestro plugin de secuenciador euclidiano.

Después de haber estado investigando y estudiando cada uno de manera individual, empezar a trabajar de manera conjunta fue fácil ya que los conocimientos que teníamos, además de las ideas para el plugin, eran similares, de manera que lo que no sabía uno lo sabía el otro, y sino, siempre conseguíamos investigar un poco y acabar solucionando el problema que nos surgiera.

Como ya he dicho, el desarrollo del plugin fue casi en su totalidad de manera síncrona, es decir, que las sesiones de trabajo que empleábamos para desarrollar el plugin eran simultáneas. Ya fuera de forma presencial, o trabajando cada uno de manera remota (estando en videollamada y haciendo uso de editores de código compartidos), los avances que el trabajo fue teniendo se debieron a que ambos estábamos involucrados en ellos. A continuación comentaré de manera más explícita cuáles fueron las aportaciones al trabajo en las que yo contribuí:

- **Algoritmo de creación de ritmos euclidianos**: Basándome en el anteriormente mencionado Aloritmo de Bresenham[1], desarrollé un algoritmo que dándole el número de pasos totales y de eventos del ritmo, devuelve una cadena de 1s y 0s donde los 1s, correspondientes al número de eventos, están repartidos de la manera más uniforme posible.
- ***Steps slider* y *Events slider***: Creación y asociación de dos controladores a los parámetros *steps* y *events*.
- **Algoritmo principal**: Desarrollo del algoritmo principal de *processBlock* que itera sobre cada paso del ritmo y crea y maneja los eventos MIDI de forma correcta.
- ***Rotation slider* y *NoteNumber comboBox***: Creación y asociación de dos controladores a los parámetros *rotation* y *noteNumber*.
- ***NoteDuration comboBox***: Creación y asociación del controlador al parámetro *noteDuration*.
- ***Multi etapa***: Diseño e implementación del modo multi etapa del plugin (máximo 4 etapas).
- ***Velocity slider***: Creación y asociación del controlador al parámetro *velocity*.
- ***Limpieza y optimización de código***: Limpieza y optimización de todo el código del proyecto para un óptimo escalado.
- ***On/Off buttons***: Implementación de un botón de On/Off por cada etapa, que crea o destruye el ritmo correspondiente.
- ***Reverse mode***: Implementación del botón *Reverse*, encargado de invertir la dirección del ritmo (hacer que se procese en sentido anti-horario).

- ***Ping-Pong mode***: Implementación del botón Ping-Pong, encargado de hacer que la dirección del ritmo pase a modo ping-pong (cambie de sentido cada vez que llega al principio/final del compás).
- ***Re-sync button***: Implementación del botón *Re-sync*, encargado de sincronizar todos los ritmos haciéndolos empezar a todos de nuevo el compás.
- **Diseño e implementación de la interfaz**: Diseño e implementación de la representación gráfica de los ritmos del plugin.
- ***Triplets button* y *Dotted button***: Implementaciónes de las funcionalidades de tresillo y notas con putillo haciendo uso de dos botones.
- **Reconocimiento y extinción de error añadiendo concurrencia**: Extinción de un error que saltaba al corromperse el *heap* porque el *PluginEditor* accedía a partes de memoria que no debía. Esto era debido a modificaciones en ese instante de la memoria por el *PluginProcessor*. Éxito en la corrección haciendo uso de un *mutex* en el instante óptimo.
- **Guardar/Cargar estado**: Implementación del cargado y guardado de presets mediante un par de botones y una *comboBox*.
- **Extinción de errores**: Extinción de múltiples *crashes* que se producían al dejar de visualizar (minimizar) y luego volver a hacer visible el plugin en una DAW.
- **Versión final del plugin**: Retoques finales en el plugin y revisión de código.

5.2. Gonzalo Costales de Ledesma

Previo al comienzo de este trabajo mis conocimientos sobre el procesamiento de audio eran escasos. Si bien sabía manejar DAWs, plugins y conocía el protocolo MIDI, mi conocimiento sobre como se implementaban estos era nulo.
Como ya ha comentado mi compañero David, nuestra manera de abordar este proyecto consistió en consultar de manera asidua la documentación de JUCE, realizar los tutoriales propuestos en su web y ver videos de Youtube. En concreto, el ejemplo que más me sirvió para este trabajo fue el del arpegiador simple, que ilustra de manera muy simple la generación y procesamiento de mensajes MIDI. Fue a partir de este ejemplo como empezamos a conceptualizar y programar el bloque de procesamiento de nuestro plugin.
El desarrollo del secuenciador se realizó en su gran mayoría de manera conjunta con David, como ya ha explicado previamente, por lo que es difícil separar las tareas entre los dos. A continuación voy a explicar más en profundidad las tareas que se realizaron y distinguir mi parte en ellas, en la medida de los posible:

- **Algoritmo principal**: Esta es la función básica del algoritmo y es, sin lugar a dudas, la tarea que más tiempo conllevó, ocupando alrededor de 12 horas, todas ellas

trabajadas con David. De atribuirme algo específicamente de esta etapa, sería la idea de la representación interna del secuenciador, basándome en la idea de representar el ritmo como una circunferencia y mantener una aguja que apunta al tiempo actual, así como las fórmulas para mantener la posición de esta aguja en caso de que cambiase el compás.

- ***NoteDuration comboBox***: Creación y asociación del controlador al parámetro *noteDuration*. Se programó enteramente de manera conjunta junto a David. Llegamos juntos a la conclusión de que era necesario un mapa para mantener todas las notas que estuvieran sonando a la vez (en el caso de que la duración de cada nota fuera superior al tiempo que se tarda en laanzar la siguiente nota) por lo que en esta parte es imposible separar las participaciones de cada uno.

- ***Multi etapa***: Diseño e implementación del modo multi etapa del plugin (máximo 4 etapas). Esta parte fue sencilla de implementar, simplemente fue necesario crear cuatro objetos donde antes había solo uno. La programación se realizó de manera conjunta, así que esta parte tampoco se puede separar.

- ***Velocity slider***: Creación y asociación del controlador al parámetro, al igual que el caso anterior, fue sencillo y se programo de manera conjunta. *velocity*.

- ***Limpieza y optimización de código***: Limpieza y optimización de todo el código del proyecto para un óptimo escalado. En esta parte se agruparon todos los elementos de cada etapa en mapas de cuatro elementos(uno por cada etapa), además de implementar un código de nombrado para los ids de cada elemento (por ejemplo el id del *slider* de *steps* de la etapa 1 sería "STEP_SLIDER_1"). Se realizó en su enteridad de manera conjunta.

- ***Triplets button* y *Dotted button***: Implementaciónes de las funcionalidades de tresillo y notas con putillo haciendo uso de dos botones. La idea de la implementación fue mía y la programación conjunta.

- **Reconocimiento y extinción de error añadiendo concurrencia**: Extinción de un error que saltaba al corromperse el *heap* porque el *PluginEditor* accedía a partes de memoria que no debía. Esto era debido a modificaciones en ese instante de la memoria por el *PluginProcessor*. Éxito en la corrección haciendo uso de un *mutex* en el instante óptimo. Realizado de manera conjunta mientras haciamos el debug de la interfaz.

- **Selector de canal MIDI**: Implementación del botón del selector de canal MIDI para poder mandar distintas instancias a distintos instrumentos. Impelmentación mía, pero David ayudó en el debug.

- **Guardar/Cargar estado**: Implementación del cargado y guardado de presets mediante un par de botones y una *comboBox*. Realizamos la investigación de amnera conjunta y programamos una parte, si bien la mayoría de la implementación fue labor de David.

- **Extinción de errores**: Extinción de múltiples *crashes* que se producían al dejar de visualizar (minimizar) y luego volver a hacer visible el plugin en una DAW. Realizado de manera conjunta.
- **Versión final del plugin**: Retoques finales en el plugin y revisión de código. Realizado de manera conjunta.

Capítulo 6

Conclusiones y Trabajo futuro

6.1. Conclusiones

Este proyecto, como bien sabemos, ha consistido en el diseño e implementación de un secuenciador euclidiano, finalizando con un plugin que ha cumplido los objetivos iniciales y ofrece la funcionalidad esperada. Además. cuenta con una interfaz gráfica de usuario que permite a los usuarios generar hasta cuatro ritmos euclidianos simultáneos de forma sencilla e intuitiva.

Además de su función básica, se han desarrollado numerosas funcionalidades adicionales que le dan mayor versatilidad a la hora de generar ritmos. Con la función de rotación es posible obtener cualquier ritmo euclidiano posible. La capacidad de hacer ritmos utilizando tresillos o notas con puntillo da otra capa de posibilidades interesantes a los ritmos generados. Poder seleccionar la nota, implica que el propio plugin es capaz de generar eventos MIDI sin necesidad de un teclado ni ningún tipo de controlador. Otra característica destacable es la capacidad de guardar y cargar *presets*, lo que hace que sea muy cómodo volver a trabajar con ritmos que se han generado con anterioridad, y poder experimentar con ellos, sin temer que no puedas volver a recuperar ese ritmo inicial.

Respecto al proceso de desarrollo del plugin, los miembros del equipo podemos afirmar que ha sido un reto debido a que nunca habíamos trabajado con JUCE, ni nos habíamos enfrentado al procesamiento audio/MIDI o a la generación de una interfaz gráfica de usuario como esta. A pesar de que ha sido difícil, hemos disfrutado aprendiendo cómo funciona el procesamiento de audio digital, las DAWs y todas las tecnologías que hemos empleado en el desarrollo de este plugin. Resumiendo, ha sido un proceso muy enriquecedor y gratificante.

6.2. Trabajo futuro

Partiendo del hecho de que el plugin ha cumplido con los objetivos iniciales esperados, algunas otras características y funcionalidades que podrían ser interesantes de implementar en el futuro para amplían aún más su funcionalidad son:

- En primer lugar, sería interesante la posibilidad de implementar un arpegiador para cada ritmo. Esto extendería la capacidad de elegir la nota que suena en cada secuenciador, con la posibilidad de poder elegir varias notas, ya sean acordes, escalas o cualquier combinación de notas que el usuario desee, de modo que el ritmo itere sobre la lista resultante de notas en cada paso de la secuencia, proporcionando una personalización melódica a los ritmos.

- Otra característica interesante sería la de implementar diferentes vistas para la interfaz gráfica, de modo que el usuario pueda cambiar y usar la que más cómoda le parezca según sus preferencias. Dos posibles ideas que se han considerado son: una opción donde la representación de los ritmos está separada, y no representados todo ellos concéntricamente como ahora sucede, y otro diseño con los ritmos desplegados en una recta, yendo esta representación más acorde con los secuenciadores estándar.

- Nuestra última idea relacionada con el trabajo futuro consiste en desplegar este plugin y todas sus funcionalidades en un dispositivo físico. Esto permitiría, entre otros usos múltiples, el brindar al usuario la oportunidad de realizar shows en vivo utilizando esta increíble herramienta, cambiando la típica mesa de DJ por este dispositivo y abandonando la vía convencional.

Capítulo 7

Introduction

7.1. Motivation

7.1.1. Historical context

After many ideas throughout the late XIX century and the entire XX century about musical creation with the use of technologies, finally in the 1980s this concept became into a real practice with the launch of digital synthesizers by Yamaha, specifically, the DX series. The popularity of these instruments capable of producing sounds using purely digital circuits was such that, the Yamaha DX7, reached a number of sales that place it today as one of the best-selling synthesizers in the history.

Figura 7.1: *Yamaha DX7*

In this transition era from analog to digital, it is worth highlighting some events such as:

- The release of the CD or compact disc in 1982, by Sony and Philips together after several individual prototypes, which changed the way music is shared by introducing the ability to store and play digital audio.

- In 1985 the launch of *Commodore Amiga*, a personal computer with a very affordable price and great multimedia capabilities that boosted music production using technology.
- The development and presentation of MIDI (Musical Instrument Digital Interface), a technological standard that describes a protocol, a digital interface and connectors that make it possible to transport and store symbolic musical information, or musical events (commonly called MIDI events) between computers, musical instruments and hardware.

7.1.2. Digital Audio Workstations

Nowadays, music production and digital audio are going through their best moment, since anyone with not an advanced knowledge can develop more than respectable musical projects without no requirements of a professional music studio or large investments of money. This is largely due to the so-called Digital Audio Workstations or DAWs (Digital Audio Workstations), digital audio recording, editing and production systems composed of software tools (plugins, virtual instruments, etc ...) and hardware (microphones, monitors, etc ..).

During the second half of the XX century, DAWs were composed entirely of hardware components (with exclusive software dedicated to it) since the computers limitations of the time were remarkable. When these limitations of data storage and processing were reduced, DAWs evolved to a point where everything that was needed to produce digital audio, that easily fullfilled a studio and was quite expensive, became basically a software tool. This tool can be used on conventional computers and even mobile devices requiring nothing more than an interface so the user can input audio and MIDI events into the device obtaining a playable audio output on other devices.

These complex software are used from the most basic users to the best studios around the world due to three main reasons:

- **Multi-track environment:** It offers the ability to work with multiple tracks independently.
- **Large extension:** It provides us with an extensive project duration, easily exceeding 5 hours, or even more, depending on the sample rate we are working with.
- **Default quality tools included:** DAWs come equipped with software such as synthesizers or samplers among others, and with audio effects such as equalizers, compressors, reverbs, etc. In addition, to the default set of tools we can add an infinity of virtual instruments, effects and plugins that everybody can easily acquire.

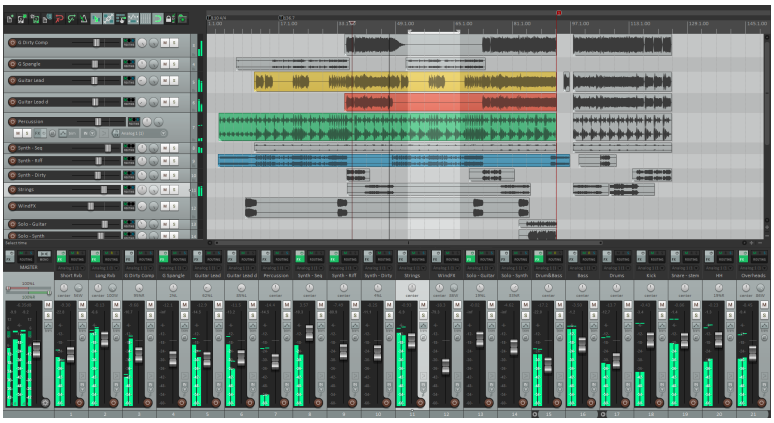

Figura 7.2: *REAPER's graphical user interface*

7.1.3. Plugins

As we have previously mentioned, DAWs come with a set of default tools where which you can also add an infinity of extra functionalities thanks to different extensions among which are the VST plugins.

VST, from the acronym Virtual Studio Technology[11], is a standard developed by the German company Steinberg whose purpose is to offer the possibility of developing and connecting effect plugins and audio synthesizers with DAWs. Apart from VST, there are other standards such as AU (Apple) or AAX (Avid Audio), however, VST is the most used of all.

In the next chapter we will go deeper talking about both the structure of DAWs, as well as the three large groups of plugins that are used in them: audio effects (whose function is to transform audio), virtual instruments (which generate audio interpreting MIDI events) and MIDI sequencers or generators (which assist in the work of sequencing or programming musical events).

7.2. Objectives

With this project the main objective we want to cover is to create a MIDI generator that allows the user to test a paradigm you are not used to when it comes to creating and configuring rhythms. The implementation of this generator as a plugin with the VST standard brings accessibility for any user and versatility of being able to use it on your favorite digital audio workstation.

In addition, the following secondary objectives are proposed:

- Show a professional interface, at the same time that is friendly and easy to understand by the user, without having to read large manuals.

- Offer the possibility of creating rhythms with up to four of them at the same time. This means that in the same instance of the plugin, you can produce four different rhythmic patterns simultaneously.
- Allow you to connect the output of the plugin to the MIDI channel of the digital audio workstation you want, connecting the desired instrument to sound in each instance.
- Offer a comfortable control panel where the user can configure as desired the different parameters of the rhythm they want to obtain.
- Possibility to synchronize all the rhythms that are playing at a given moment so that their patterns start again from the beginning. This will allow the user to have control over the synchronization without having to stop and resume the rhythms all the time.
- Offer the functionality of saving and loading presets to the user, so that plugin configurations and rhythms parameters can be loaded and saved at the time you want.

To conclude, and speaking of our personal learning, the objectives we propose are the following ones:

- Develop a good work structure and a good organization that facilitate the learning and collaborating processes, where synchronization and communication with the other member of the team is optimal so that the performance is the best.
- Learn how to use and deal with new tools and software such as the JUCE framework, so that the progress of the project and the development of our ideas and objectives are way easier.
- Face problems and situations that we have not faced during our learning in the career, and being able to solve them by researching and acquiring new knowledge that will very possibly be useful to us in the future.

7.3. Work Plan

Once the objectives of the project have been established and having described the motivations of the same, it is time to describe the process that we have followed to carry out the project with the fulfillment of these mentioned objectives.

The first steps of this work were to investigate the origin of the Euclidean sequencer, which led us to understand how Euclidean rhythms emerged, and how, nowadays, they are present around the whole world. After understanding these, the next step was to develop an algorithm that was capable of producing these rhythms, providing it as input the number of steps of the rhythm, and the number of events that it had so that it distributed them evenly creating the relevant rhythm.

The language we used to develop this algorithm was C++ since, our next step would be to investigate and familiarize ourselves with the JUCE framework[6], a work environment that offers all the libraries and tools necessary to develop applications, plugins and libraries oriented to audio and/or music using the mentioned C++ programming language. This process of familiarization with JUCE was greatly supported by the fact that on its website there is a learning process composed of small modules with tutorials that we completed and an infinity of videos on YouTube where you can find help. We would like to highlight specifically a channel called The Audio Programmer[8], which has been a great support in this learning process from the beginning to end of the project.

Figura 7.3: *JUCE's logo*

After having developed in this first stage of research quite simple plugins such as volume controllers, wavetables or envelope controllers that helped us to understand the structure of a plugin, and the operation of JUCE, our tutor advised us to develop a MIDI generator /effect plugin, because this way we would strengthen concepts similar to those necessary to develop the Euclidean sequencer. Our choice was to develop a simple arpeggiator that, taking some MIDI events from the input, modified them by changing them to a greater or lesser tone of our choice, or randomly.

Once the construction of this plugin is finished, and understanding the management and creation of MIDI events, we began to develop the final project. After numerous versions throughout this development process, the basic components and functionalities of the plugin were implemented, highlighting the main rhythm controller, which allows the user, through rotating sliders, to select the number of steps and events of the rhythm. We also created other additional controls, highlighting among many others, the rhythm direction control or the possibility of grouping the notes in triplets and dotted notes. In addition, a great part of time was invested in the graphical user interface due to its huge complexity that requires a lot of signals that make it change in real time at a high speed.

Finally, the functionalities of saving and loading presets were implemented that would give the user the possibility of storing the configurations and rhythms that the user wants for later use.

Capítulo 8

Conclusions and future work

8.1. Conclusions

This project has been about the design and implementation of a Euclidean sequencer, ending with a plugin that has met the initial goals and offers the expected functionality. It has a graphical user interface that allows the users to generate up to four simultaneous Euclidean rhythms in a simple and intuitive way.

In addition to the basic functionality, numerous additional functionalities have been developed that give greater versatility when it comes to generating rhythms. With the rotation function it is possible to obtain any possible Euclidean rhythm. The ability to make rhythms using triplets or dotted notes gives another layer of interesting possibilities to the rhythms generated. Being able to select the note, implies that the plugin itself is able to generate MIDI events without needing a keyboard or any type of controller. Another notable feature is the ability to save and load presets, which makes it very comfortable to retrieve interesting rhythms that have been generated, and to be able to experiment with them, without fearing that you will not being able to get that initial rhythm again.

Regarding the development process of the plugin, the members of the team can affirm that it has been a challenge due to we had never worked with JUCE nor had we faced the audio/MIDI processing or a graphical user interface generation like this one. Even though it has been difficult, we have enjoyed learning how digital audio processing, DAWs, and all technologies we have used developing this plugin work. Summarizing, it has been a very enriching and rewarding process.

8.2. Future work

Starting with the fact that the plugin has met the expected initial goals, some other features and options that could be interesting to implement in the future expand even more

its functionality are:

- First of all, it would be interesting the possibility of implementing an arpeggiator for each rhythm. This would extend the functionality of choosing the note that sounds in each sequencer with the possibility of being able to choose several notes, whether chords, scales or any combination of notes that the user desires, so that the rhythm iterates over the resulting list of notes in each step of the sequence, providing a melodic customization to the rhythms.
- Another interesting feature is to implement several different views in order to being able to change between all of them according to the most comfortable preferences of the user. Two possible ideas of views that have been considered are: an option where the representation of the rhythms is separated, and not concentrically and another design with the rhythms unfolded on a straight, being this representation more in line with standard sequencers.
- Our last idea related with future work consists on deploying this plugin and all of its functionalities in a physical device, so between other multiples uses, the user could have the opportunity to perform live shows using this amazing tool, changing the typical DJ table for this device and leaving the conventional way.

Bibliografía

[1] Algoritmo de Bresenham. Algoritmo de Bresenham — Wikipedia, the free encyclopedia. https://es.wikipedia.org/wiki/Algoritmo_de_Bresenham.

[2] Algoritmo de Euclides. Algoritmo de Euclides — Wikipedia, the free encyclopedia. "https://es.wikipedia.org/wiki/Algoritmo_de_Euclides.

[3] Audacity. Audacity: software de edición de audio. https://audacity.es/.

[4] EuclideanSequencer.VST3. Archivo .vst3 del plugin. https://www.dropbox.com/sh/oxfqw92vgh6crv4/AADG9_KRxzJFCDvtP1H-Ew-va?dl=0.

[5] GitHub. Carpeta con todo el código fuente del proyecto. https://github.com/SpringJoe1/Euclidean-Sequencer.

[6] JUCE. Juce framework webpage. https://juce.com/.

[7] Notas con puntillo. Puntillo — Wikipedia, the free encyclopedia. https://es.wikipedia.org/wiki/Puntillo.

[8] The Audio Programmer. The Audio Programmer — YouTube channel. https://www.youtube.com/c/TheAudioProgrammer.

[9] Godfried Toussaint. The euclidean algorithm generates traditional musical rhythms. In *Renaissance Banff: Mathematics, Music, Art, Culture*.

[10] Tresillo musical. Tresillo — Wikipedia, the free encyclopedia. https://es.wikipedia.org/wiki/Tresillo.

[11] Virtual Studio Technology (VST). Virtual studio technology — Wikipedia, the free encyclopedia. https://es.wikipedia.org/wiki/Virtual_Studio_Technology.