
UNIVERSIDAD
COMPLUTENSE
MADRID

# GENERACIÓN DE MÚSICA PROCEDURAL Y ADAPTATIVA PARA VIDEOJUEGOS

# PROCEDURAL AND ADAPTATIVE MUSIC GENERATION FOR VIDEOGAMES

## Curso académico 2019-2020

Trabajo de fin de grado del Grado en Desarrollo de Videojuegos, Facultad de Informática, Universidad Complutense de Madrid.


## Autores

Alberto Córdoba Ortiz · Juan Ruiz Jiménez · Ramón Arjona Quiñones

## Director

Jaime Sánchez Hernández

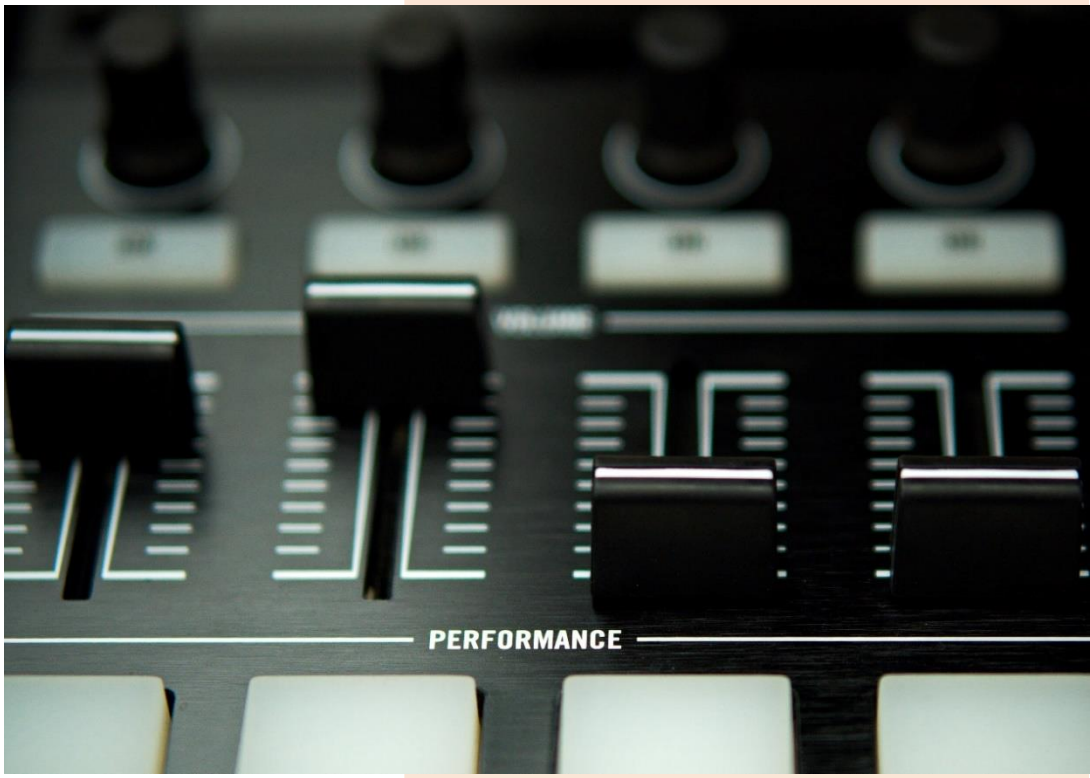
## Codirector

Marco Antonio Juan de Dios Cuartas

2019-2020

# Generación de música procedural y adaptativa para videojuegos

## Procedural and adaptative music generation for videogames




Alberto Córdoba Ortiz

Juan Ruiz Jiménez

Ramón Arjona Quiñones

UNIVERSIDAD COMPLUTENSE MADRID

*“La música es para el alma lo que la gimnasia para el cuerpo”*

*— Platón*

# ÍNDICE

## PALABRAS CLAVE

Música, Procedural, Adaptativo, Videojuego, Supercollider, Unity, Plugin

## RESUMEN

La música procedural permite crear piezas de manera completamente automática en tiempo real. Esta se forma siguiendo un conjunto de reglas definidas por el compositor que crea el sistema. Además, si dichas reglas tienen la capacidad de modificarse adaptándose a eventos y cambios decimos que el sistema es adaptativo y, por lo tanto, creará música adaptativa.

Esta idea ha sido explorada en otras áreas, y es muy útil en el ámbito de los videojuegos, donde la música desempeña un papel esencial potenciando las emociones que se desean transmitir al jugador, de forma análoga a la narrativa. El hecho de que esta música pueda llegar a sonar durante muchas horas supone un reto para los compositores, que deben evitar caer en melodías repetitivas o agobiantes para no romper esta inmersión.

Es por esto por lo que se ha creado una herramienta para Unity que, trabajando en conjunto con una plataforma de síntesis de audio y composición algorítmica llamada Supercollider, es capaz de crear distintos tipos de piezas procedurales y adaptarlas a distintos videojuegos. Para ello se ofrece al usuario la posibilidad de seleccionar entre tópicos clásicos de los videojuegos, así como seleccionar la forma en que su videojuego afecta a la composición.

De esta forma, se trabaja uno de los aspectos menos tratados en el desarrollo de videojuegos (comparado con otros como el arte, diseño y programación) gracias a una herramienta distinta a las usuales y que hace hincapié en el apartado sonoro y musical de la obra.

## KEY WORDS

Music, Procedural, Adaptative, Videogame, Supercollider, Unity, Plugin

.

## SUMMARY

Procedural music allows the creation of musical pieces in a totally automatic way, all in real-time. Said music is created following a set of rules defined by the composer who creates the system. In addition, if said rules have the capacity to change, adapting themselves to certain events, we say the system is adaptative and therefore, creates adaptative music.

This idea has been widely explored in other areas, and it's useful in videogames, where the music performs a key role enhancing the emotions destined to the player, in a similar way as the narrative does. The fact that this music can last so much time playing is a challenge for composers, who must avoid using repetitive or oppressive melodies in order not to break this immersion.

Thus, a tool for Unity has been created; a tool which, working along an audio synthesis and algorithmic composition platform called Supercollider, can create different types of procedural pieces and adapt them to the videogame. To do so, the user is offered the possibility of selecting among classic videogame topics, as well as choose the way he wants his videogame to affect the composition.

This way, we treat a side aspect in terms of videogame development (compared to others as art, design and programming), thanks to a different tool specialized on the sonorous and musical area of the videogame.

# CAPÍTULO 1 – INTRODUCCIÓN

## 1.1 Estado del arte y antecedentes

### 1.1.1 Música procedural

La **música procedural** es definida por **Karen Collins** como "una composición que evoluciona en tiempo real, de acuerdo con un conjunto específico de reglas lógicas de control". (Collins - An Introduction to Procedural Music in Video Games, 13, 2009)

En este tipo de música, el papel del compositor no es el de crear las estructuras, notas y demás elementos sonoros de la pieza, sino el de diseñar un **sistema** capaz de generarlas en base a ciertas **reglas**. El objetivo de estas reglas es conseguir que no se pueda anticipar qué será lo próximo que vamos a escuchar, de manera que en el momento en el que el sistema se pone a funcionar, el compositor ya no tiene control sobre este (Brian Eno - In Motion Magazine, 1996).

"*Aunque pueda tener el placer de descubrir procesos musicales y componer el material musical para ejecutarlos, una vez que el proceso está configurado y cargado, se ejecuta solo*" (Steve Reich, 1980)

El estilo más explorado por los autores pioneros en la música procedural es la **música ambiental**. Es un estilo que hace hincapié en el tono y la atmósfera sonora por encima de otros parámetros como la estructura musical o el ritmo.

#### *Steve Reich*

**Steve Reich** es un compositor estadounidense conocido por ser pionero en la producción minimalista. La técnica que usaba es el "cambio o ajuste de fase" y consistía en reproducir dos o más cintas a distintas velocidades en bucle de manera que el resultado obtenido siempre suena diferente.

En 1965 aparece la primera pieza procedural creada por Steve Reich utilizando un sistema mecánico para su composición, incluyendo la técnica del ajuste de fase. El sistema consistía en dos grabadoras de cinta donde grabó su voz diciendo ***"It's gonna rain"*** y las puso a reproducir en bucle a distintas velocidades, la primera a 0,998% de la velocidad original y la otra a 1,002%. Así pues, con esta regla tan sencilla se crea una pieza que suena distinta hasta que las cintas se sincronizan otra vez. Puede escucharse en el siguiente enlace:

- Steve Reich - It's Gonna Rain:

  https://drive.google.com/file/d/1QvjG7nhhkqDOSJQNWP08rSaawx8YprRe/view

## *Brian Eno*

**Brian Eno**, considerado el padre de la **música generativa**, es el mayor creador de piezas ambientales generativas de la segunda mitad del siglo XX y principios del XXI. Se expone cual es según su criterio, la diferencia entre la música común, entendiendo por común música compuesta por compositores, y la música generativa:

"*La música clásica, al igual que la arquitectura clásica o como otras formas clásicas especifican la entidad a construir de antemano. La música generativa no hace eso, especifica un conjunto de reglas y luego las pone a trabajar. En palabras del libro de Kevin Kelly, la música generativa está fuera de control. Ahora, fuera de control significa que no sabes con exactitud qué es lo que puede hacer. Tiene vida propia*".(Generative Music - Brian Eno - In Motion Magazine, 1996.)

Este autor se vio motivado en sus inicios tras escuchar la pieza anteriormente mencionada de Steve Reich ("*It's gonna rain"*) ya que, como señala el compositor: "Se trata de una pieza muy interesante porque es muy simple. Es una pieza que cualquiera podría hacer, pero resulta muy compleja a nivel sonoro" (Eno, 1996). Esto se debe a que, a medida que la vas escuchando, el cerebro se va habituando de la misma manera que el ojo humano lo hace cuando mira algo durante mucho tiempo. El cerebro cancela la información común y deja de procesarla, y empieza a darse cuenta de las diferencias.

Tal y como relata en su libro, uno de sus mayores intereses ha sido siempre la invención de **'maquinas'** y **'sistemas'** que puedan producir experiencias musicales y visuales y,

aunque en muchas ocasiones estas máquinas son solo conceptuales y no físicas, su misión consiste en hacer obras sonoras con materiales y procesos especificados, pero en combinaciones e interacciones que no lo estén (Eno, A Year with Swollen Appendices, 330, 1996).

La primera obra generativa de este autor fue **"*Discret Music*"** en 1975. Para ella, combinó dos simples ciclos melódicos de diferentes duraciones que se repetían por separado solapándose de manera arbitraria (técnica similar a la de *Steve Reich* antes mencionada). La duración de la primera pieza es de 29 segundos y de la segunda de 33. Así pues, consigue que, con tan solo 2 melodías de escasos 30 segundos de duración, se pueda crear una pieza de hasta 957 segundos (29 * 33) hasta volver a solaparse y generar el mismo patrón.

La obra generativa más relevante de *Brian Eno* se llama **"*Music for airports*"** (1978)**.** El propio Eno comenta que se trata de una pieza sencilla, con melodías cantadas por 3 mujeres y por él mismo. Una de las notas se repite en bucle cada 23,5 segundos, la siguiente cada 25 7/8, y las otras dos siguientes a intervalos muy similares. De esta manera, pasa mucho tiempo hasta que las reproducciones puedan sincronizarse otra vez (Eno - In Motion Magazine, 1996). Al tratarse de una persona no formada en la música de una forma académica, Eno tenía dificultades para interpretar partituras y usaba sus propios símbolos gráficos para denotar las frases musicales o bucles tal y como muestra la Figura 1.1.

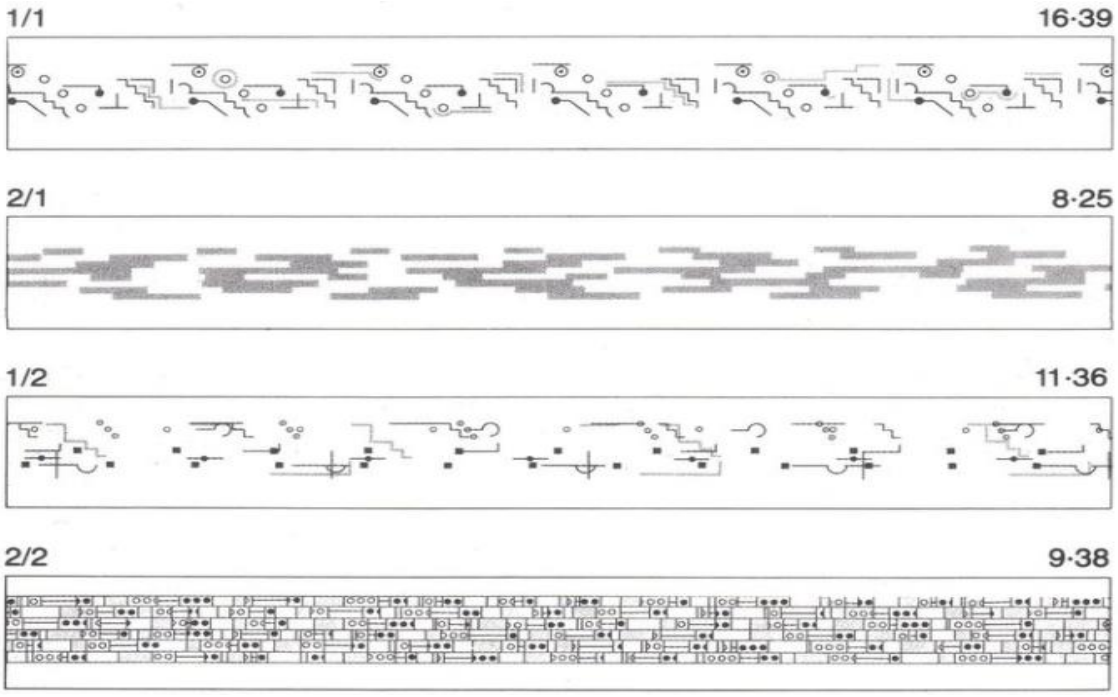



**Figura 1.1 Gráficos de "Music for airports" – reverbmachine.com**

El autor promueve que la música ambiental no debe sobrecargar al usuario, sino formar parte de la experiencia del entorno sin alterarlo. Esto se refleja en los títulos de algunas de sus obras generativas más importantes a las que se puede acceder en el apartado de anexo.

Cabe destacar que la frontera entre los términos “generativo” (propio de Eno) y “procedural” es incierta incluso para los profesionales de este sector, por lo que suelen usarse indistintamente según la situación.

### 1.1.2 Música adaptativa

La **música adaptativa** es aquella que se modifica en tiempo real de acuerdo con eventos que va recibiendo. Esto la relaciona estrechamente con la música procedural ya mencionada, con la diferencia de que aquí no se trata de una evolución progresiva y natural de la pieza, sino de cambios repentinos disparados por el entorno que rodea a la misma.

## *Karen Collins*

Karen Collins es una investigadora que explora la relación entre sonido y tecnología. Centrada principalmente en la importancia del **audio dinámico** en videojuegos, lo define en "*An Introduction to Procedural Music in Videogames*" como un audio no lineal de elementos variables de los aspectos sonoros del videojuego, y lo divide en dos categorías:

- **Audio interactivo**: Eventos sobre los cuales el jugador tiene el control y puede realizar en diferentes momentos, como pueden ser los pasos del personaje, los disparos de arma, etc.
- **Audio adaptativo:** Eventos sobre los que el jugador no tiene control y cambian en función de parámetros del juego. Son, por ejemplo, los cambios de música al cambiar de ubicación o la reproducción de sonidos cuando hay una transición del día a la noche.

Collins encuentra una **serie de problemas** a los que la música procedural debe hacer frente. El principal es **las sesiones largas de juego**, donde el jugador empieza a cansarse de la música y acaba apagándola, llegando a reemplazarla con la suya propia. La autora describe que la manera más eficiente de hacer música que no sature al jugador es reduciendo la densidad de la instrumentalización, y pone como ejemplo en el videojuego **Spore** (Maxis, 2008).

Otro de los problemas es la **sinergia entre la música y la escena de juego**. Si esta primera no está de alguna forma ligada a la imagen y narrativa del juego, se pierde la inmersión, y por tanto el sentido de tener música en un videojuego.

Collins distingue dos tipos de procesos en la música procedural para videojuegos:

- **Algoritmos transformativos**: Son aquellos que afectan a la estructura de la pieza, como la alteración del tono, la escala o las capas de sonido.
- **Algoritmos generativos:** crean la pieza a partir de una estructura definida.

## 1.1.3 El ámbito del videojuego

A partir de las investigaciones de Collins, se puede sacar en claro que es posible crear un sistema de música procedural para usarlo en un **videojuego**, con la ventaja de poder enriquecerlo si se añaden elementos de audio dinámico o adaptativo.

A continuación, se presentan algunas obras relevantes para este trabajo que han implementado dichos sistemas de una forma u otra:

### *Spore*

Es un juego de simulación y estrategia para Windows y Mac creado por la compañía Maxis en 2008. Consiste en la evolución de una célula hasta la colonización de la galaxia.

Los programadores de sonido **Kent Jolly** y **Aaron McLeran** basaron la música de este título en las obras de Brian Eno. Todo el juego depende en gran medida de la generación procedural para la creación de criaturas, mundos y música.

La música después de ser prototipada en **Max/MSP** en su versión final fue trasladada a **PureData** y consta de muchas muestras pequeñas de sonido que generan la banda sonora en tiempo real. Las melodías y los ritmos se generan en un marco de reglas muy limitadas.

El jugador también puede construir y editar su propia música para crear himnos para sus ciudades. Sin embargo, se establecieron mecanismos para limitar el "*input*" del usuario de modo que los cambios no afectaran radicalmente a la pieza que se generaba y la música siguiera teniendo sentido.

### *No Man's Sky*

**No Man's Sky** es un videojuego de exploración espacial multiplataforma publicado el 9 de agosto de 2016 por la compañía Hello. Una de las principales características de este juego es que casi todo su contenido está generado de manera procedural; los planetas, su fauna, su flora, su relieve y, lo que más interesa, la música.

El director de sonido de No Man's Sky, **Paul Weir**, habla en una charla de 2016 sobre sonido procedural en *Sónar+D* de la dificultad de crear música y eventos sonoros en este proyecto puesto que, tradicionalmente, están vinculados a eventos del juego y estos están siempre bajo control. Por otro lado, en un juego donde todo se genera de forma procedural, no se puede tener nada demasiado controlado.

La música del juego está compuesta por un grupo de música ambiental electrónica conocido como ***65daysofstatic***, el cual no hace música procedural habitualmente. Sin embargo, en este juego crearon los "*Assets*" necesarios para la generación procedural de música en colaboración con el equipo de sonido de Hello Games.

Estas son las herramientas desarrolladas por este equipo de sonido:

- **VocAlien:** consiste en un plugin de síntesis de audio para Wwise que crea las voces de las criaturas. Uno de los problemas principales es que el tipo de sonido de la criatura depende también de muchos de sus aspectos físicos, como la longitud del cuello o el tamaño de la caja torácica (estos modifican la resonancia, el tono u otros parámetros).
- **Pulse:** programa que utiliza diversas técnicas algorítmicas para generar piezas musicales en función de parámetros del juego. El programa es capaz de identificar el estado actual y toma una serie de "*presets*" con una sonoridad similar a la del grupo *65daysofstatic* para crear la pieza en cuestión en tiempo de ejecución.

Cabe destacar la influencia de Spore sobre este juego, ya que muchas de las técnicas usadas aquí son mejoras de las ya realizadas por el equipo de Jolly y McLeran.

El auge de este tipo de música ha causado la aparición de **tecnologías** que facilitan la creación de piezas procedurales y/o adaptativas. Se presentan, por tanto, algunas de las más interesantes entre las que se encuentra la elegida para el propio plugin:

## *Pure data*

**Pure data** es una herramienta de código abierto para multimedia desarrollado por Miller Puckette que permite a creación de música de forma visual a músicos, investigadores o desarrolladores sin necesidad de escribir código. Puede ser usado en procesos de sonido, video, graficos 2D y 3D, dispositivos de entrada y MIDI.

Las funciones algorítmicas se representan como cajas visuales llamados *objetos* que se establecen en una ventana llamada *canvas*. El flujo de datos entre objetos se realiza mediante conexiones visuales llamadas *patch cords*. Cada objeto juega una función específica que puede ir desde complejas operaciones matemáticas a bajo nivel hasta funciones de alto nivel de audio y video. Los objetos de Pure data se compilan en el lenguaje de C o C++. En la figura 1.2 podemos visualizar el canvas con objetos conectados mediante patch cords. En la figura 1.3 se muestra un ejemplo de interfaz más complejo de un programa de música procedural llamado Sim Cell creado con pure data.

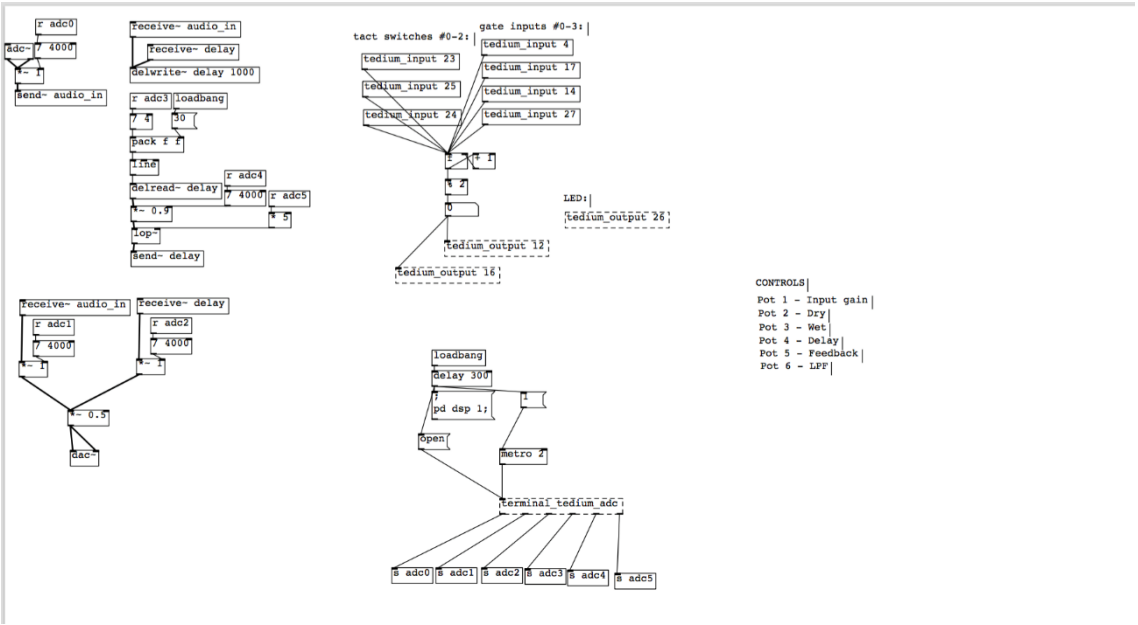



**Figura 1.2 Pure Data, elementos básicos – forum.pdpatchrepo.info**

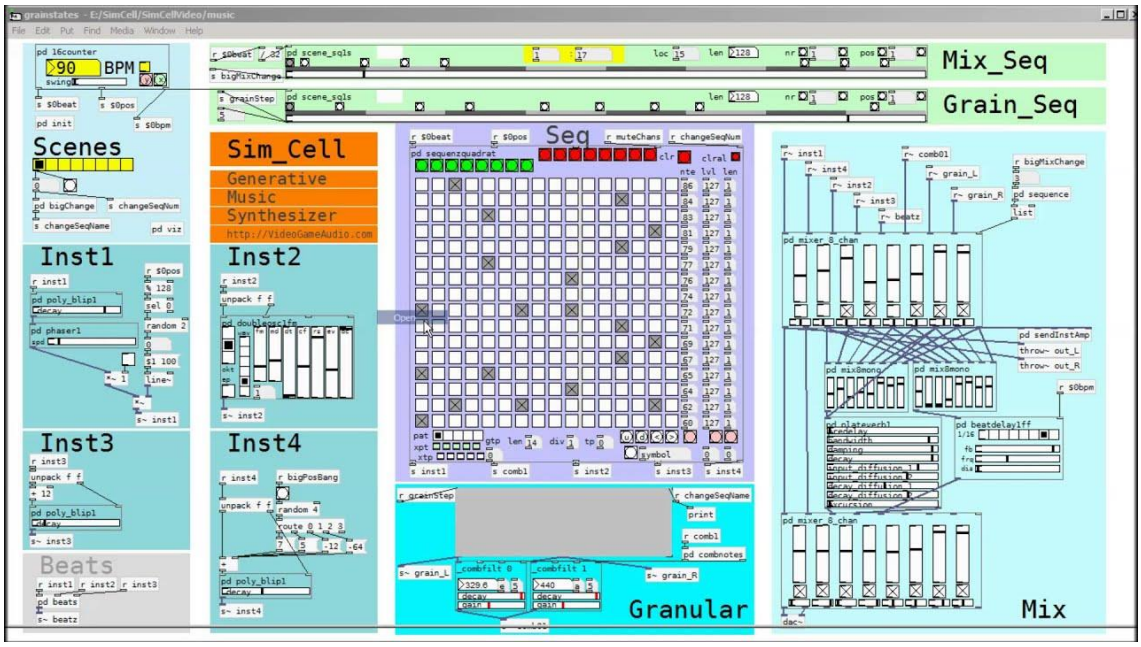



**Figura 1.3 Proyecto de Pure Data para el videojuego "Sim Cell" – School of Video Game Audio (YouTube)**

## *Electronic Arts Pure Data*

El interés por usar lenguajes como Max/MSP o Pure Data en juegos ha existido siempre, pero la oportunidad de usarlo siempre ha sido baja. La compañía de juegos Electronic Arts tenía su propio software interno para realizar una función similar a la de Pure Data, pero en 2002 dejó de estar en desarrollo. Tanto el equipo de *Spore* como el equipo del videojuego *Dead Space* se interesaron por Pure Data, aunque finalmente el segundo usó Lua en su lugar. El equipo de *Spore* tomó la decisión de utilizar Pure Data por tener una clara división entre interfaz gráfica y motor de sonido que permitían un uso más amigable para los diseñadores de sonido.

Lo primero fue eliminar elementos del motor de audio de Pure Data y reemplazarlos por objetos construidos sobre este que interactuasen con su sistema de audio, a esto lo llamaron EApd. EApd no es el único reproductor de audio integrado puesto que se usaba en casos circunstanciales. Esto se debe a que Pure Data no funcionaba tan rápido como se deseaba en algunas situaciones, a menudo los sonidos tenían que instanciarse muchas veces todos con distintas posiciones 3D y la asignación de memoria en tiempos de carga a menudo generaba fallos. Kent Jolly (programador de sonido de *"Spore"*)

explica que, aunque a pesar de ser herramientas potentes para ser usadas en videojuegos, no están adecuadas para su uso en entornos de producción de videojuegos.

## *Supercollider*

Supercollider es una plataforma de síntesis de audio y composición algorítmica usada por músicos artistas e investigadores que trabajan con sonido. Es gratuito, de código abierto y puede usarse en las plataformas de Windows, macOS y Linux. Supercollider se compone de tres partes principales:

- **Scsynth**: servidor de audio en tiempo real. Es la parte principal de la plataforma. Permite complejas combinaciones de técnicas de síntesis aditiva y sustractiva, FM, síntesis granular, FFT.
- **Sclang**: Es un lenguaje de programación interpretado, de tipado dinámico y basado en objetos. Sclang controla los sintetizadores vía OSC (Open Sound Control), también permite conectar la aplicación a hardware externo incluyendo controladores MIDI o música en red. También tiene extensiones creadas por usuarios llamadas Quarks.
- **Scide**: Es el editor donde se escribe código en lenguaje sclang con sistema de ayuda integrado.

Actualmente la plataforma se mantiene en desarrollo gracias a la comunidad que crean extensiones, llamadas "*Quarks*", para ampliar su funcionalidad

Open sound control (OSC) es un protocolo de comunicaciones que define un formato de mensajes que facilita la comunicación entre dispositivos capacitados para recibir o enviar datos a través de la red.

## *Fmod y WWise*

**FMOD** es un motor de audio que permite añadir música y sonido posicional a los proyectos. Puede incluirse la versión de FMOD Core en los proyectos para acceder a la API de bajo nivel compatible con C, C++, C# y Javascript.

FMOD Studio es una herramienta basada en FMOD que proporciona un entorno gráfico y un abanico de funcionalidades para diseñar, compilar y optimizar audio adaptativo. Su interfaz permite a los usuarios incluir eventos de sonido que pueden ser manipulados enrutando la salida de estos a una lista muy amplia de efectos como reverberación, tremolo, filtros de frecuencias, etc. También permite la creación de parámetros que realicen cambios sobre el efecto orden de reproducción de las pistas. Es compatible con Unity, Unreal Engine y Cry Engine.

Como se visualiza en la figura 1.4, los eventos de sonidos pueden agruparse en carpetas y ser asignados a bancos de sonido. Cada evento se compone de una línea de tiempo donde se muestra el audio a reproducir y multitud de opciones para su manipulación.

Este sistema de sonido permite la creación de piezas adaptativas, ya que al poder manipular parámetros desde el código del juego pueden realizarse cambios en tiempo real de lo que se está escuchando.


**Figura 1.12 Imagen de la interfaz de Fmod, captura propia**

Compañías como Rockstar games, 2K games, 505games, Activision, Bethesda o Sony han usado la herramienta para sus proyectos.

Cabe también mencionar a **WWise**, se trata de un motor de audio similar a FMOD, también diseñado para añadir música y sonidos posicionales y adaptativos a los videojuegos. Incluye de la misma forma una interfaz visual llamada *Wwise Audio Lab*. La

versión de *Wwise Authoring API* permite la integración de Wwise en cualquier motor, herramienta o aplicación. También posee un plug-in para Unity y Unreal Engine.

### 1.1.5 Métodos algorítmicos para composición musical

En este apartado describiremos los algoritmos más utilizados para la composición de música computacional. El criterio más importante es cómo reacciona a los cambios de situación del medio interactivo. La mayoría de los métodos de composición computacional se centran en la generación de forma compositiva dividiéndose y estructurándose algorítmicamente.

Las decisiones se pueden tomar de dos maneras:

- **Determinista:** Es un algoritmo completamente predictivo si se conocen sus entradas.
- **Estocástico:** Los procedimientos de generación integran la elección aleatoria para tomar decisiones.

Ambas técnicas se pueden usar para música de medios interactivos ya que las variables de dicho medio pueden modificar la generación. Los medios estocásticos pueden no ser coherentes gran parte de las veces, pero cumplen la función de evitar las reiteraciones y motivos recurrentes ofreciendo mayor variedad a la obra.

#### *Procesos aleatorios*

Es la técnica estocástica más simple. Genera valores aleatorios entre un mínimo y un máximo, lo que llamaremos frontera, dichos valores pueden variar en el tiempo haciendo que las posibilidades de generación sean cambiantes.

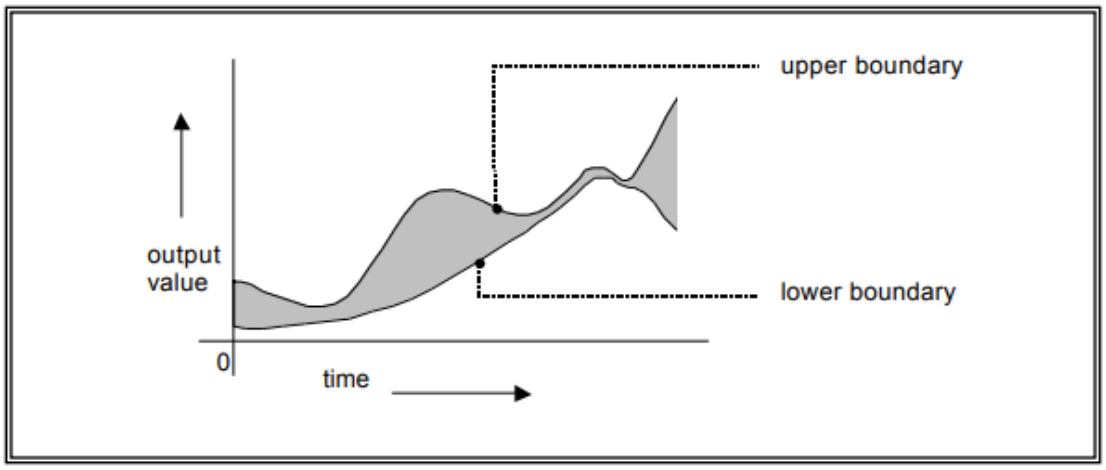



**Figura 1.13** (Paulus, The use of generative music systems for interactive media, 2001)

Este método se utiliza tanto en técnicas de síntesis granular como en sistemas de música procedural para medios interactivos ya que puede usarse para aleatorizar los elementos del ámbito melódico como las notas, la progresión harmónica, etc.

### *Distribución probabilística*

Un método estocástico similar a la técnica aleatoria, pero en este caso los valores que se puede tomar entre las fronteras de valor máximo y mínimo dependen de la probabilidad de cada valor, es decir, la probabilidad de que un valor sea seleccionado depende de la distribución de probabilidad. De esta manera podemos controlar que los eventos musicales que más nos interesen suenen con mayor frecuencia. La manera habitual de extender este algoritmo es combinándolo con los procesos aleatorios para variar las probabilidades en el tiempo en función de parámetros del sistema interactivo.

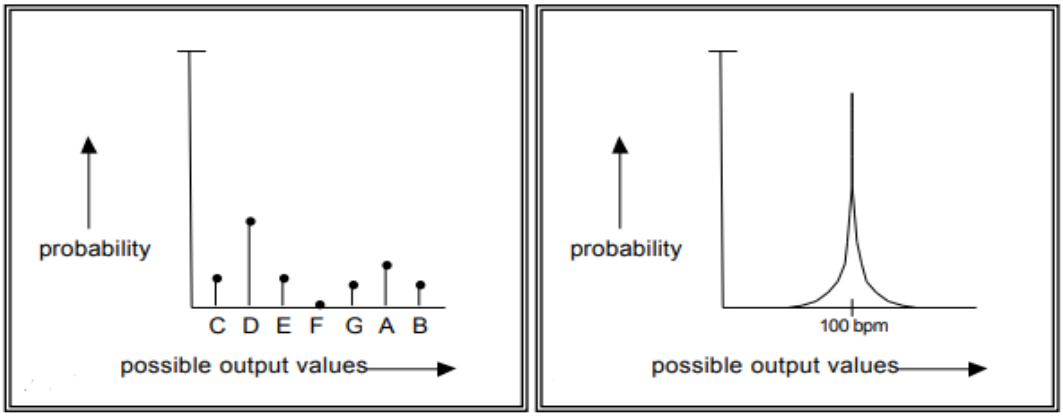


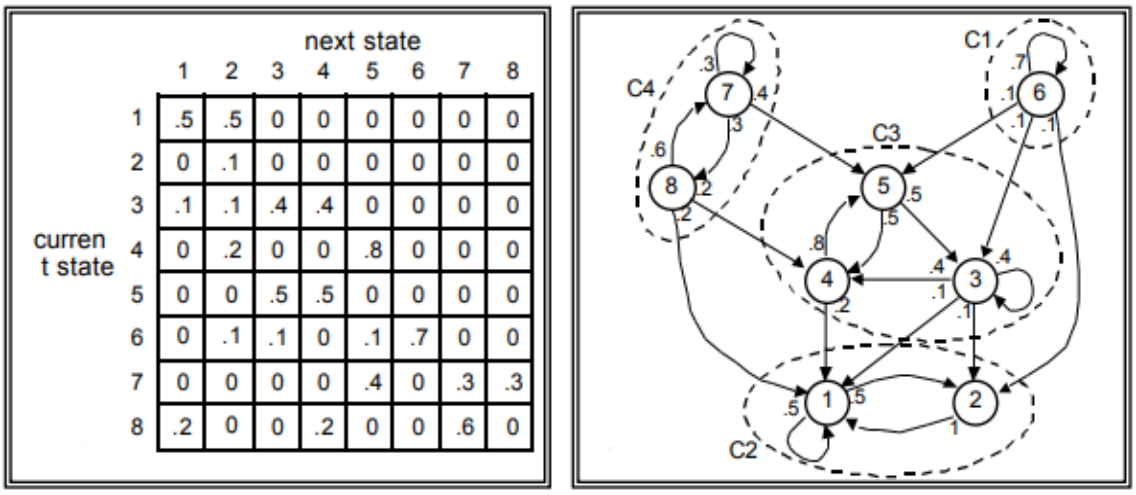

**Figura 1.14** (Paulus, The use of generative music systems for interactive media, 2001)

## *Ruido Marrón*

El ruido marrón es un derivado del proceso aleatorio. Consiste en otra técnica estocástica y difiere del proceso aleatorio en que su salida depende del valor tomado anteriormente. Habitualmente el valor generado es igual, uno más alto o uno más bajo que el anterior, esto da como resultado un cambio gradual y permite controlar lo que el valor puede aumentar o disminuir. Esta técnica es muy recurrente, por ejemplo, en la gestión automática del volumen de un sonido de viento de manera que los cambios sean notables, pero no bruscos.

## *Cadenas de Markov*

Es un sistema estocástico de probabilidad donde el evento futuro está determinado por el estado de uno o más eventos del pasado. Los estados de transición se disponen en forma de matriz. El funcionamiento consiste en que la fila de probabilidades correspondiente al estado actual se usa para derivar el siguiente estado. La fila del nuevo estado se usará a su vez para derivar nuevamente el estado siguiente, y así sucesivamente. El orden de la cadena de Markov determina el número de estados que se tiene en cuenta para la siguiente iteración. Así pues, en una cadena de orden uno tendrá en cuenta el estado inmediatamente anterior y se representará en una matriz bidimensional, si miramos dos estados anteriores usaremos una matriz tridimensional y así sucesivamente (orden N Matriz N+1-dimensional)



| curren t state \ next state | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | .5 | .5 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | .1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | .1 | .1 | .4 | .4 | 0 | 0 | 0 | 0 |
| 4 | 0 | .2 | 0 | 0 | .8 | 0 | 0 | 0 |
| 5 | 0 | 0 | .5 | .5 | 0 | 0 | 0 | 0 |
| 6 | 0 | .1 | .1 | 0 | .1 | .7 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | .4 | 0 | .3 | .3 |
| 8 | .2 | 0 | 0 | .2 | 0 | 0 | .6 | 0 |



**Figura 1.15** (Paulus, The use of generative music systems for interactive media, 2001)

Como podemos visualizar en el ejemplo en función de los valores que obtengamos en el estado actual en el que nos encontramos podremos saltar de un estado a otro. Los usos que le podemos dar en música procedural son muy variados ya que cada estado puede estar conectado con un patrón musical, un ritmo, una melodía o valores más independientes como el volumen, el tono, etc.

### *Algoritmos genéticos*

Utilizados para la composición algorítmica, estos procesos usan técnicas evolutivas con procesos de evaluación de aptitud, apareamiento y mutación para conocer cuáles son los más aptos para obtener datos musicales más apropiados para ciertas condiciones. Esta técnica es poco útil para la generación en tiempo real.

## 1.2 Objetivos del trabajo

El **objetivo central** de este TFG es la creación de una extensión para Unity que permita, a un usuario que no posee conocimientos previos del tema, añadir música para su videojuego.

En concreto:

- Crear un sistema en Supercollider extensible con la capacidad de crear música procedural de distintas temáticas típicas en videojuegos.
- Controlar la música generada por el sistema mediante variables que puedan modificarse en tiempo real para hacer que ésta sea adaptativa.
- Crear una extensión para Unity que permita a un usuario sin conocimientos musicales previos, seleccionar qué tipo de música quiere y qué elementos de su videojuego quiere que afecten a la adaptación de esta.
- Conectar la extensión de Unity con el sistema creado en SuperCollider para que ambas herramientas funcionen en conjunto en la creación de piezas procedurales y adaptativas.
- Ayudar al desarrollador independiente y sin conocimientos musicales previos a incluir piezas con sentido a su proyecto.
- Aportar a desarrolladores con conocimientos musicales o estudios con equipos de producción musical la maqueta para sus escenas sin necesidad de compositores ni músicos que realicen dicha tarea. Los estudios podrán entonces realizar test para confirmar si la pieza que quieren para su proyecto funciona.

## 1.3 Plan de Trabajo

Para alcanzar el objetivo propuesto por el TFG se ha dividido en 3 bloques principales y, aunque todos los integrantes han trabajado en los tres bloques, cada uno está liderado por uno de los autores de este TFG. Los bloques, junto a sus principales subpuntos, son los siguientes:

- Creación del plugin de Unity — Ramón Arjona Quiñones
    - Interfaz y funcionalidad de cara al usuario
    - Arquitectura del código en Unity
- Creación de paquetes con sentido musical — Juan Ruiz Jiménez
    - Selección de tópicos propios de videojuegos
    - Traducción de la teoría musical a SClang
- Arquitectura y comunicación entre sistemas — Alberto Córdoba Ortiz

- Comunicación de Unity con Supercollider
- Arquitectura del código en SuperCollider para asegurar extensibilidad

### 1.3.1 Aportaciones

*Alberto Córdoba Ortiz*

Uno de los problemas principales a resolver es la comunicación de Unity con Supercollider. Para ello se usa el protocolo de comunicaciones Open Sound Control que permite compartir información musical a través de una red.

Para poder comunicarnos desde el punto de vista de Unity se han usado como base unos scripts de libre uso modificación y distribución disponibles en un repositorio de github mencionado en los apéndices. Estos archivos componen una arquitectura básica de envío y recepción de mensajes OSC los cuales han sido estudiados para comprender su uso y han sido modificados gran parte de ellos para que realicen la tarea que se requiere. Los cambios más grandes consisten en la creación de métodos que desempaquetan el mensaje recibido y lo gestionan para asegurarse de que no está duplicado y sobrescribirlo en ese caso.

Los mensajes que se pueden enviar a SuperCollider están acotados y especificados por los desarrolladores de este TFG y el usuario tiene acceso a unas modificaciones controladas que, gracias a una capa intermedia de abstracción, pueden interpretarse y enviarse correctamente a Supercollider. Por otra parte, desde SuperCollider se crea el cliente y el servidor para poder enviar y recibir paquetes de información. Para obtener Información más ampliada puede consultarse en el apartado 2.4, Comunicación Unity – SuperCollider.

Por otro lado, se ha estudiado el funcionamiento del lenguaje SCLang y se ha puesto en práctica para poder crear la distribución de clases y la jerarquía usando herencia y polimorfismo para la distribución de paquetes.

La arquitectura consiste en una instancia de la clase MusicMaker que hace de controlador de los paquetes, así como del servidor, los efectos, los parámetros y los

mensajes. Este controlador creará las instancias de todo esto mencionado y a partir de este momento empezará a recibir los cambios a vía OSC de Unity.

Otra de las aportaciones es la creación de un generador de acordes. Este generador se almacena en una variable global para que se pueda acceder a él desde cualquier lugar del código o de cualquier paquete y permite crear acordes en una escala y una nota base indicada. El generador recibe como argumentos un valor del 0 al 6 abarcando así los modos griegos y una nota MIDI base de la que se crean los acordes. Se devolverá de esta manera un array cuyo contenido son un total de 10 acordes de 2 a 5 notas cada uno. Este generador de acordes permite crear combinaciones rápidamente de acordes aleatorios pero que funcionan entre sí para crear música ambiental de distintos tópicos.

También ha sido relevante la aportación al paquete de terror creando la estructura básica de cómo debe sonar el paquete y creando una versión primitiva de dicho paquete que posteriormente ha sido muy ampliado con los conocimientos musicales del Juan Ruiz Jiménez.

La creación de escenas de Unity sobre las que probar el plugin también ha abarcado parte del trabajo y las aportaciones a esta parte del trabajo son las siguientes:

- escena de un pequeño juego de naves cuya música varía cuando suceden distintos eventos. Es una demo muy primitiva que debido a todos los cambios realizados ya no está operativa.
- escena de botones genérica para testear el uso de cualquier paquete musical.

### *Ramón Arjona Quiñones*

Las aportaciones al trabajo han sido, principalmente, las relacionadas con la parte de Unity vistas en el punto 2.2. Esto puede resumirse en 5 puntos, ordenados cronológicamente:

1. **Investigación previa** sobre las tecnologías mencionadas, especialmente:
    - **Propiedades y reflexión en C#:** cómo funcionan y si realmente podían proveer la funcionalidad que se necesitaba para llevar a cabo la parte adaptativa del proyecto.

- **Editor de Unity:** estudio del modelo de ejecución y de las clases que incluye (*EditorWindow*, *CustomPropertyDrawers*, *SerializedProperties*, *GUILayout*, etc)
- **Nociones básicas sobre SuperCollider:** introducción al lenguaje SCLang, arranque del servidor y definiciones básicas de sintetizadores, funciones y mensajes.

2. **Diseño del plugin** en Unity atendiendo al modelo "**MVC"** (*Model-View-Controller),* un estilo de arquitectura muy válido sobre multitud de plataformas y que por tanto sería de gran utilidad:
   - **El modelo**; el conjunto de clases, enumerados, constantes y funciones auxiliares que soportaría luego el resto del proyecto (MMFoundation.cs)
   - **El controlador**; que se aprovecha de este modelo para realizar las funciones ya descritas y, en general, hacer que el plugin funcione (MusicMaker.cs)
   - **La vista**; que expone el modelo al usuario de forma filtrando la información para no sobresaturarlo (MMEditor.cs)

3. **Implementación** de cada una de estas partes, atendiendo a otros patrones de diseño como el "**Singleton"** (clase *MusicMaker*, de instancia única) y agrupando las infraestructuras necesarias bajo un espacio nombre común. En general, se busca priorizar la claridad y eficiencia de código para permitir posibles ampliaciones futuras. Con cada iteración del desarrollo, y tras hablar con los demás integrantes, se fue cambiando la estructura de las clases para dar cabida a nuevas ideas.

4. **Persistencia** de los datos introducidos por el usuario, guardando sus preferencias en varios archivos en disco de forma que no pierda la información al salir y volver a entrar al proyecto.

5. **Unión** del trabajo de los 3 integrantes en la fase tardía del proyecto, apoyándose en el trabajo ya realizado por cada uno. Se define, por tanto, una estructura común para los mensajes que se envían de Unity a SuperCollider y se modifica lo necesario en ambos lados para que la conexión sea óptima.

- **En Unity**, se normalizaron los mensajes para que solo hubiera 2 tipos, cada cual contenía además índices para saber de qué aspecto y qué capa se trataba.
  Se añadieron también mensajes fijos al inicio (para iniciar el servidor y la música) y al final (para parar todo y matar el servidor) de la ejecución del juego.
- **En SuperCollider**, se creó un método en **Packages.sc** que recibe estos mensajes y se encarga de procesarlos para activar/desactivar la capa correspondiente.

Cabe destacar también alguna **aportación transversal**, como la creación de escenas y componentes de prueba en Unity que se han ido realizando para testar el correcto funcionamiento del plugin, pero que no se entregan con el código adjunto por su propia definición.

Estas son todas las aportaciones realizadas al proyecto, no queriendo profundizar más en los resultados de cada una de ellas dado que han quedado suficientemente explicadas en el capítulo 2, y no se desea volver a entrar en detalles sin haber necesidad de ello.

### *Juan Ruiz Jiménez*

La carga de trabajo ha sido centralizada en la composición musical a partir de algoritmos generativos en SuperCollider. Para poder desarrollar la parte del plugin encargada de la generación de música con sentido fue necesaria una investigación previa del lenguaje a utilizar: SCLang. Esto incluye un estudio exhaustivo del lenguaje, así como la comprobación de las utilidades que ofrece este y cómo pueden ser trasladadas a un terreno que pueda resultar útil para los desarrolladores.

Se utilizó como ejemplo base el código de muestra que fue utilizado para el desarrollo de música ambiental en SuperCollider hecho por el autor Eli Fieldsteel, lo cual sirvió de gran ayuda para estructurar el proyecto y definir los sintetizadores y samples. A partir de las conclusiones y los recursos obtenidos de esta serie de ejemplos se desarrolló el código que se utiliza en los distintos paquetes del plugin. (Ver apartado *Apéndices*).

Las aportaciones se pueden resumir en, por una parte, el desarrollo de los cuatro paquetes generadores de música, lo que incluye los algoritmos utilizados y el desarrollo de los samples y efectos sobre los que se apoyaran estos. Para su elaboración fue

necesario un estudio previo de cada género musical, tanto a nivel musical (lo cual incluye el desarrollo cultural y la evolución histórica de ese tipo de música) como a nivel tímbrico (estudio de los instrumentos utilizados teniendo en cuenta como de posible es su incorporación en el plugin).

Concretizando el apartado anterior, esa carga de trabajo se desglosa en la programación del paquete ambiental, fantástico, desértico y de terror (sin olvidar en este último la ayuda de Alberto Córdoba con la elaboración del generador de acordes y la base del paquete). En cada paquete se incluyen los algoritmos generativos de las capas percutivas, armónicas, melódicas y de efectos, así como el sintetizador de dientes de sierra, el sampler o la reverberación, comunes a todos los paquetes.

Por otro lado, también se han aportado funcionalidades comunes al plugin como pueden ser la lectura de archivos de audio jerarquizada en carpetas, permitiendo su uso y reproducción desde el sintetizador sampler o la definición de buffers, buses y nodos desde SuperCollider.

Cabe destacar también el diseño de la jerarquía utilizada por el plugin que se basará en el uso de capas musicales distribuidas en tres categorías: percusión, armonía, melodía y efectos. De esta forma, el plugin utilizará un sistema de generación de música a partir del concepto de capa el cual, siguiendo con la idea de que todos los generadores implementarán el mismo número de capas y seguirán esta distribución de la misma manera, ha sido utilizado para su implementación posterior en la arquitectura. El desarrollo de esta idea derivó en el concepto de **paquete musical** y los métodos que utilizará de manera teórica, la idea que se siguió definitivamente para encapsular las funcionalidades relativas a la generación de música.

Por último, también han sido aportados todos los archivos de audio utilizados en el plugin y presentes en la carpeta *buffers*. Esto incluye un trabajo de amplia búsqueda de samples de uso libre, así como la elaboración o edición a mano de la gran mayoría de ellos con el uso de herramientas externas como Reaper. Para el desarrollo de esta labor también fue necesario un estudio detallado del entorno a utilizar y de los conceptos teóricos a la hora de editar o crear efectos, como pueden ser el uso adecuado de la ecualización, compresión y uso de plugins externos de instrumentos.

# CAPÍTULO 2 – TECNOLOGÍAS UTILIZADAS Y DISEÑO DEL PLUGIN

## 2.1 MusicMaker

A partir de este punto se designará con el nombre de "**MusicMaker***"* a la herramienta o *plugin* desarrollado para el TFG, siendo este también el nombre oficial con el que un usuario pueda referirse a él.

Antes de entrar en detalles sobre la implementación, se va a explicar qué es lo que ofrece desde un punto de vista de usuario, así como los pasos a seguir para ponerlo en funcionamiento.

Para usar **MusicMaker,** es necesario tener una serie de archivos que, en caso de distribución comercial, se colocarían en las carpetas del sistema correspondientes mediante un instalador:

- 12 archivos de código fuente de SuperCollider **(.sc)**
- 13 archivos de código fuente para Unity **(.cs)**
- Librerías de sonido divididas en 23 carpetas contenedoras de las muestras **(.wav y .aiff)**

Una vez instalado, el usuario hace uso de la herramienta apoyándose en un entorno gráfico. Distinguimos 2 partes, cada una con una interfaz distinta:

1. **Elección de música procedural:** se escogen los parámetros con los que se desea que empiece la música, como se observa en la figura 2.1

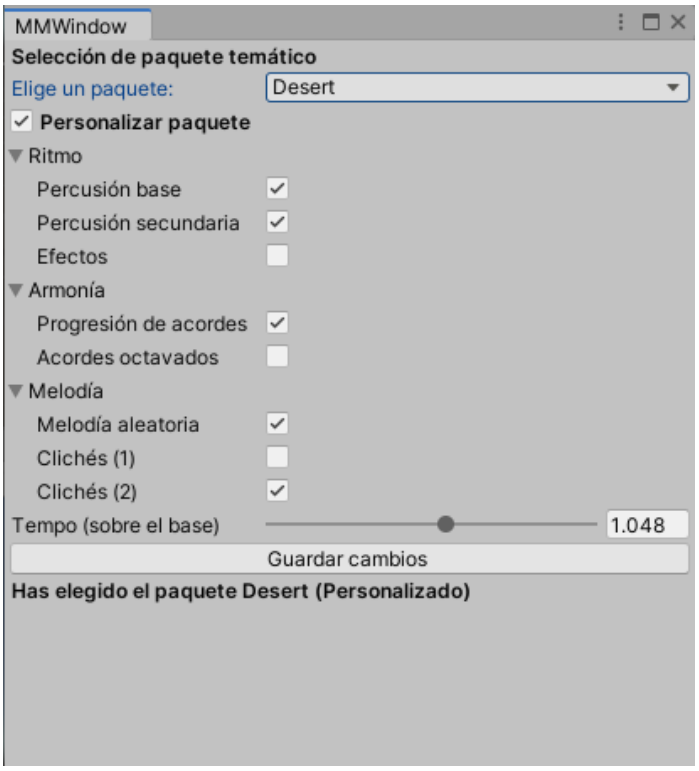



**Figura 2.1: Interfaz en Unity para la parte procedural – Fuente propia**

Estos parámetros son:

- **Paquete:** se elige entre una serie de paquetes temáticos (Ambiente, Desierto, Fantasía y Terror). Cada uno utiliza una serie de instrumentos y/o algoritmos distintos para generar la música. Esta lista de paquetes es fácilmente ampliable.
- **Capas activas:** existen 4 aspectos principales (Ritmo, Melodía, Armonía y Efectos) dentro de la música generada. Cada uno tiene a su vez una serie de **capas** (entre 2 y 5 dependiendo del aspecto) que se corresponden con **pistas musicales**. Se puede seleccionar qué capas empiezan activas desde el inicio del juego, con cualquier combinación posible, siendo la opción predeterminada la de comenzar con la primera capa de cada aspecto activa (4 pistas en total). Esto escala hasta un máximo de 12 pistas sonando simultáneamente.
- **Tempo relativo:** se elige un multiplicador entre 0.5 y 2 que afectará al tempo musical base del paquete en cuestión. Como ejemplo, si el paquete Desierto tiene un tempo musical base de 100bpm, se podría dejar tal cual (tiempo relativo = 1) o establecerlo a un valor entre 50bpm (tempo relativo = 0.5) y 200bpm (tempo relativo = 2).

2. **Configuración de la música adaptativa:** una vez configurado el paquete con estos parámetros se crea un **objeto** que permite configurar cómo va a comportarse la música durante la ejecución del videojuego.

   Esto es gracias a una serie de **tuplas,** de las que se hablará más tarde, que permiten seleccionar de una forma cómoda cómo afectan variables del propio código a las capas antes vistas; principalmente activándolas y desactivándolas. en la figura 2.2 se ve cómo los diferentes elementos del array "***ActivatedLayers***" se mapean para ***activar*** distintas ***capas rítmicas*** de la música.

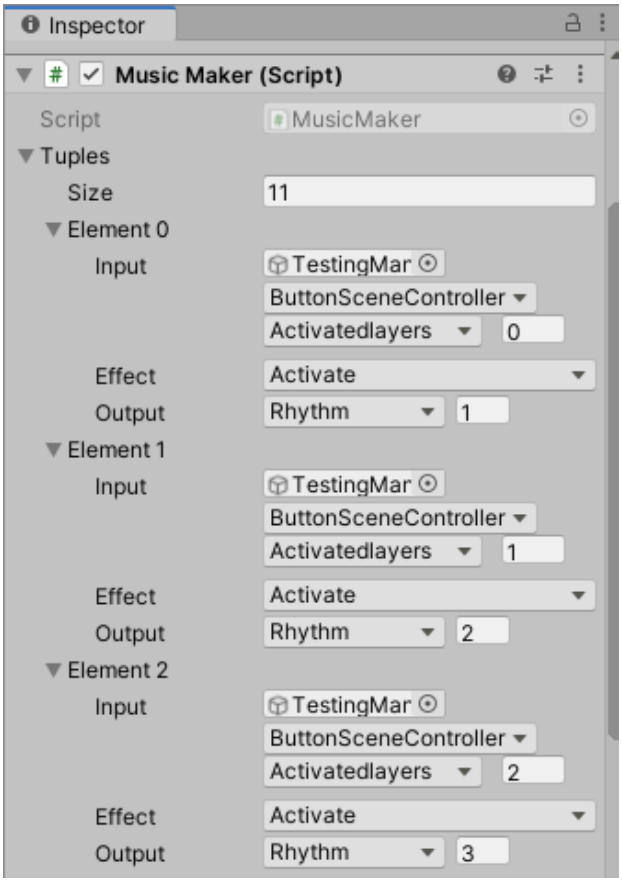



**Figura 2.2 Interfaz en Unity para la parte adaptativa – Fuente propia**

Cabe destacar que esta parte del plugin es opcional para el usuario, i.e., puede limitarse a reproducir la música procedural, con el inconveniente de que suenan solo aquellas capas que se eligieron en un primer momento.

## 2.2 El *plugin* en Unity

Unity es un motor de videojuegos multiplataforma creado por Unity Technologies, el cual es hoy, una de las herramientas más potentes y conocidas tanto por desarrolladores independientes como por grandes compañías de videojuegos.

En los siguientes puntos se explica de una forma gradual el uso que se ha hecho de Unity para desarrollar la herramienta, empezando por una base sobre el motor y el lenguaje C#, y siguiendo con la arquitectura, interfaz y, por último, funcionamiento del plugin.

### 2.2.1 Unity y C#

#### *Unity*

Unity utiliza una **arquitectura por componentes**, principal razón que impulsó a elegirlo frente a otros posibles candidatos. Se explica a continuación esta arquitectura de forma progresiva:

- Un **componente** es un *script*, generalmente escrito en C#, que realiza una serie de funcionalidades. Los componentes de Unity heredan todos de una clase padre llamada ***MonoBehaviour*****,** incluida en las librerías del propio motor. Esta clase contiene una serie de atributos, y también de ***callbacks*** (i.e., funciones que se llaman en determinados momentos de la ejecución) que proporcionan la base al usuario para empezar a programar, ya que cualquier componente que herede de *MonoBehaviour* heredará por tanto estas características. Algunos ejemplos de los callbacks:
    - **Awake:** llamado una sola vez al inicio de la vida del componente.
    - **Start:** llamado cada vez que el componente se activa.
    - **Update:** el más interesante de todos, llamado por el motor una vez cada *tick* de ejecución. Constituye la herramienta principal para programar el bucle de juego.

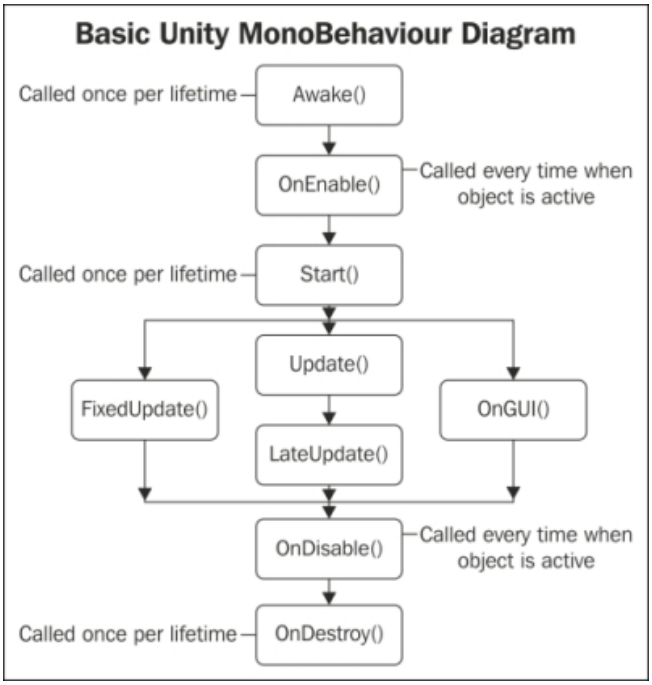



**Figura 2.3 Ciclo de vida de un MonoBehaviour – retro.relaxate.com**

La gran ventaja de los componentes es que permiten que sus atributos públicos sean visibles desde el propio Editor de Unity.

- Un ***GameObject*** es una entidad contenedora de componentes. De por sí no proporcionan funcionalidad alguna; es la lista de componentes que contienen lo que les da la utilidad. Todos los GameObjects tienen al menos 1 componente asociado, llamado *Transform,* que les dota de una posición, rotación y escala. La referencia entre GameObject y componentes es doble; un GameObject contiene una lista de componentes, a la vez que cada componente posee una referencia al GameObject del que forma parte. Esto permite la comunicación directa entre componentes que posee GameObject.
    - Cabe destacar la existencia de una extensión de los GameObject llamada ***Prefab*****.** Mientras que un GameObject pertenece a una escena de juego, un Prefab pertenece al proyecto en sí. Esto permite tener copias de esta entidad en diferentes escenas, con la utilidad de que los cambios que se le hagan afectarán a todas las copias existentes (una especie de “molde”)
- Una **escena de juego** es, en esencia, una lista de GameObjects. Las escenas pueden reproducirse, pausarse o pararse completament. Una escena necesita, como

mínimo, una **cámara** y una **luz** para que el resto de GameObjects puedan verse correctamente.

De la misma forma que un GameObject se diferencia de otro por los componentes que posee, una escena de juego se diferencia de otra por los GameObjects que reúne.

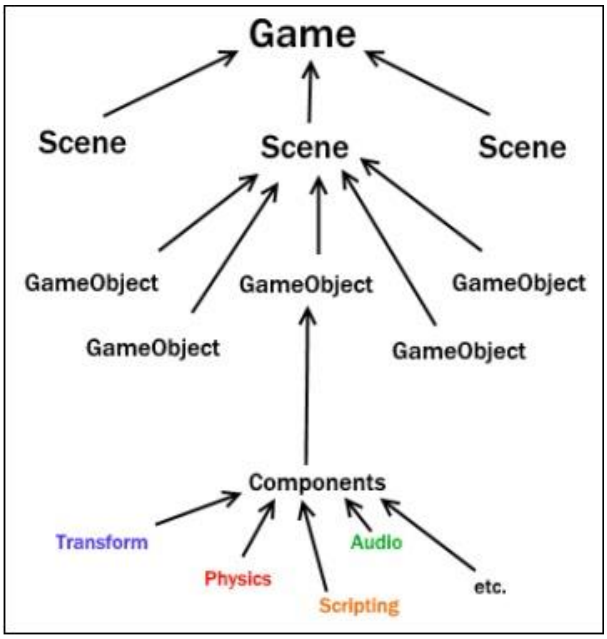



**Figura 2.4 Arquitectura por componentes en Unity – Tutorialspoint.com**

Estos elementos conforman el grueso de este motor a nivel de ejecución. Sin embargo, cabe destacar algo más sobre Unity, y es el hecho de que cuenta con **2 modelos de ejecución:**

1. **Motor (*UnityEngine*):** el propio motor de Unity que proporciona el acceso a los componentes y objetos antes mencionados. Es el responsable de llamar a los métodos *Awake()*, *Start()* y *Update()*. Contiene librerías para usar los MonoBehaviour, *Transform*, *GameObject*, y un largo etcétera en el que no se va a profundizar.
2. **Editor (*UnityEditor*):** conforma el GUI del programa, es decir, toda la parte visual que ofrece a programadores y diseñadores para poner a puesta las escenas. Ofrece métodos parecidos al motor, como el método *OnGUI() (*una especie de *Update()).*
   En esta parte hay un elemento muy importante llamado el **Inspector** (figura 2.5)**.** Este se muestra cada vez que el usuario selecciona un GameObject de la escena, y permite añadirle componentes, quitarlos y modificar sus atributos públicos.

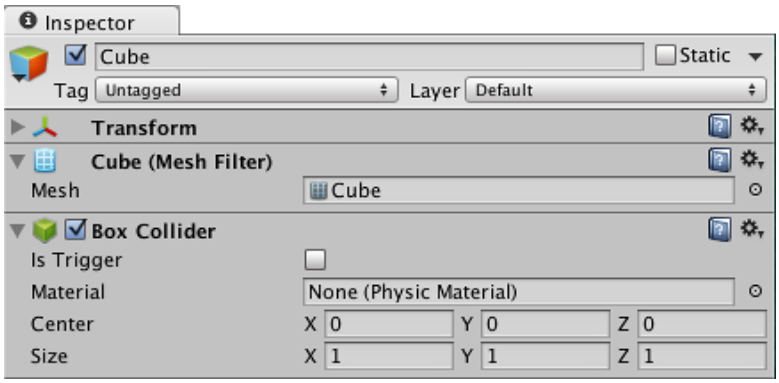



**Figura 2.5 Inspector de un GameObject – docs.unity3d.com**

Como última característica a destacar, tenemos la **serialización**. Unity proporciona una "*etiqueta*", **System.Serializable,** que sirve para indicar que la clase a la que precede puede ser serializada automáticamente, esto es, el propio motor se encarga de almacenarla y sacarla de archivo con un par de simples llamadas a un API.

## *C#: propiedades y reflexión*

Como ya se ha mencionado, Unity basa su sistema de scripts en el lenguaje **C#**. Una de las características de este lenguaje **indispensables** para el desarrollo del plugin son las **propiedades** y la **reflexión**:

- Las **propiedades** son una mezcla entre variable y método que permite el acceso a campos privados. En la figura 2.6 vemos la forma más común de declarar propiedades en C# (de forma ***autoimplementadas***):

```
#region Variables
public float floatParam { get; set; }
public bool boolParam { get; set; }
public bool[] boolArrayParam { get; set; }
public int intParam { get; set; }
#endregion
```

**Figura 2.6 Propiedades autoimplementadas de distintos tipos – *Fuente propia***

- La **reflexión** (*System.Reflection*) es una librería que proporciona una API para obtener información sobre los ensamblados cargados y los tipos definidos dentro de ellos. La clase de esta librería "***PropertyInfo"*** proporciona acceso a los metadatos de una propiedad (clase declaradora, nombre, tipo y valor que contiene)

En la figura 2.7 vemos un ejemplo simplificado sacado del código del plugin, que usa propiedades y reflexión para obtener el valor de la variable "*Mass*" de un componente de tipo "*Rigidbody*".

```
//Guardamos el componente en una variable tipo "object"
object obj = new Rigidbody();
//Hallamos el tipo del componente
System.Type type = obj.GetType();
//Buscamos la propiedad con el nombre "Mass"
PropertyInfo prop = type.GetProperty("Mass");

//No contiene esa propiedad
if (prop == null) return null;
//Sí la contiene; devolvemos el valor
return prop.GetValue(obj);
```

**Figura 2.7 Ejemplo de uso de la clase PropertyInfo** - ***Fuente propia***

## 2.2.2 Estructura de clases

La estructura de clases en Unity, así como algunos métodos estáticos y constantes de acceso global, se definen en el archivo **MMFoundation.cs**. Todo lo expuesto en este apartado forma parte de este archivo, que engloba las estructuras y clases bajo el **espacio de nombres "MM"** (de MusicMaker).

Una vez decidida la funcionalidad completa que se quiere dar al usuario, toca pensar en la implementación que más conviene. Se separaron, por tanto, las 3 partes más obvias que el usuario querría configurar a la hora de crear música adaptativa, cada cual representado por una clase hecha en C#:

1. **La entrada (*MusicInput*):** la clase se encarga, principalmente, de almacenar una propiedad (aunque por seguridad almacena el nombre de esta, una referencia a su componente y otra al GameObject que pertenece).

Esta propiedad puede ser de un componente programado por el usuario o de uno de los componentes nativos de Unity. Soporta 4 tipos distintos de datos; booleanos, arrays de booleanos, números enteros y de punto flotante. Los 2 primeros se usan para activar/desactivar capas musicales, mientras que los 2 últimos sirven para otros parámetros como el tempo.

2. **La salida (*MusicOutput*):** esta clase es la representación de una "capa musical", por lo que se compone de un tipo enumerado, que indica el aspecto musical a cambiar, y un entero indicando el número de capa de este aspecto.
3. **El efecto (*MusicEffect*):** este tipo enumerado contiene una serie de efectos que pueden producirse sobre distintos parámetros musicales:
   - None: valor predeterminado para evitar errores
   - Activate y Deactivate: usados para activar y desactivar capas musicales, solo funcionan con tipos booleanos de entrada
   - Increase y Decrease: usados para el tempo y posibles extensiones de la parte adaptativa.

Haciendo uso de la composición de clases nace la clase más importante de esta arquitectura: **MusicTuple**. Se trata de una tupla compuesta por una entrada, una salida y un efecto que los relaciona de alguna manera. Estas tuplas son las que el usuario puede configurar desde el Inspector.

Cabe destacar también la existencia de 2 clases **estáticas** y que proporcionan **acceso global** a funcionalidades útiles para el resto del plugin:

- **Utils:** incluye métodos útiles relacionados con las **PropertyInfo** y la ruta de guardado para los archivos de MusicMaker.
- **Constants:** incluye constantes para el número de aspectos y capas existentes, teniendo en mente una posible ampliación de estas.

## 2.2.3 Interfaz para el usuario

La interfaz para el usuario se apoya en el uso de la librería ya vista **UnityEditor.** Es en el archivo **MMEditor.cs** donde se gestiona toda la parte de entrada y configuración por parte del usuario, habiendo 2 partes bien diferenciadas siguiendo el uso en 2 pasos del plugin:

1. **Ventana del plugin:** Unity proporciona una clase para crear ventanas en su editor, de forma análoga a la que hacía con MonoBehaviour. Se puede así crear una ventana "***MMWindow"*** en la que poder personalizar los parámetros de inicio de la música. Una vez seleccionados, se vuelca esta configuración a 2 archivos distintos y se crea en la escena actual de juego el objeto "MusicMaker" con el script "**MusicMaker.cs**" ya añadido.

2. **Editor de Unity:** conviene hablar aquí de los ***PropertyDrawer***. Estos son etiquetas asociadas a una clase (al igual que *System.Serializable*) que sirven para implementar la interfaz gráfica con la que la clase se verá desde el Editor.

   Significa que podemos personalizar la interfaz en tiempo real, usando la información introducida por el usuario para filtrar y mostrarle solo lo que necesita ver.

   En las siguientes figuras vemos cómo sería la interfaz sin PropertyDrawers (arriba) frente al resultado final con PropertyDrawers (abajo)


**Figura 2.8 Interfaz sin PropertyDrawers** – ***Fuente propia***

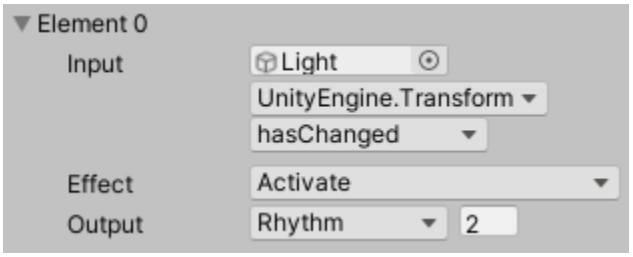



**Figura 2.9 Interfaz con PropertyDrawers** – ***Fuente propia***

La 2ª interfaz es más **compacta**, permite seleccionar los ítems desde ***popups*** y se evitan **campos innecesarios** (como los campos *min*, *max*, *index* y *LayerNo* en el caso de la figura 2.7)

Esto nos sirve para configurar las tuplas en el ya creado objeto **MusicMaker**. En este objecto se puede elegir el número de tuplas deseadas, además de configurar el input, efecto y output de cada una.

La interfaz visual para elegir el output y el efecto son bastante sencillas, por lo que se procede a explicar el proceso que sigue el usuario para elegir el **input**, el cual mezcla la reflexión de C# con los PropertyDrawers de Unity y ha sido la parte más complicada de implementar:

1. Se muestra un campo para arrastrar el GameObject que se desea usar de entre los presentes en la escena de juego.

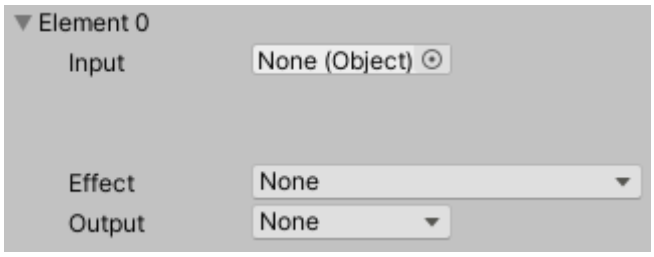



**Figura 2.10 Paso 1, elección de GameObject** – ***Fuente propia***

2. Una vez elegido, y gracias a la reflexión, podemos acceder a sus *propiedades* mostrando una lista de componentes disponibles.

- Es posible que un componente tenga variables, pero no propiedades. Por ello, y usando por 2º vez la reflexión, se omiten los componentes en esta lista que no contengan propiedades.

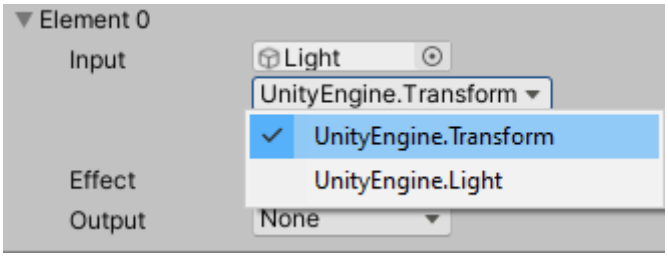



**Figura 2.11 Paso 2, elección de componente** – ***Fuente propia***

3. Por último, y una vez elegido el componente, se muestra una lista similar pero esta vez con todas las propiedades del componente, que deben ser de los tipos mencionados (bool, bool[], int o float).

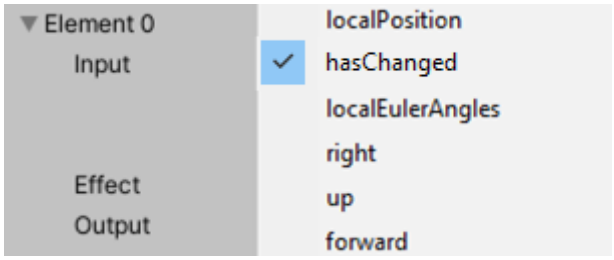
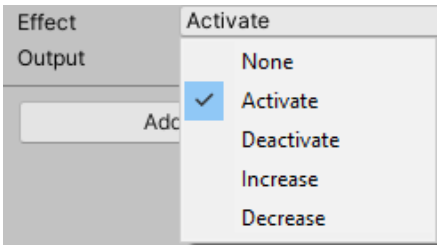
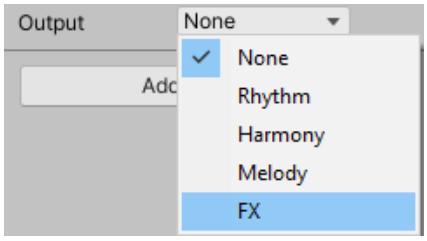



**Figura 2.12 Paso 3, elección de propiedad** – ***Fuente propia***

- En los tipos bool[] se añade un campo extra para indicar el índice
- En los tipos numéricos se añaden 2 campos extras para indicar el rango entre el que se moverán y poder así normalizarlos.

Una vez seleccionado el **input**, es cuestión de elegir un **efecto** y una capa de **salida**, terminando así de configurar la tupla.




**Figuras 2.13 y 2.14, elección de efecto y salida– *Fuente propia***

En la figura 2.14 vemos la tupla ya configurada, siendo la propiedad booleana "*hasChanged*" del componente Transform la responsable de activar el efecto de sonido nº2.

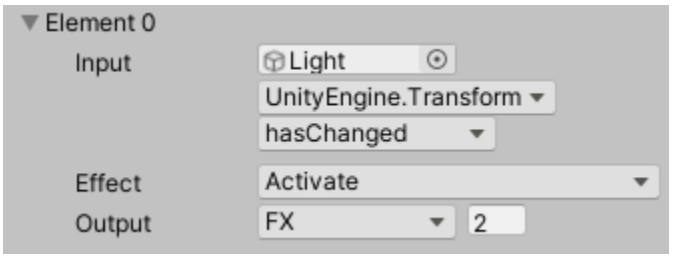



**Figura 2.15 Tupla ya configurada – *Fuente propia***

## 2.2.4 Funcionamiento del plugin

Una vez explicada la infraestructura interna y los parámetros que el usuario introduce a través de la interfaz, se puede entender el funcionamiento del plugin, basado en el uso del **UnityEngine**.

Esta parte está contenida en el script **MusicMaker.cs**, una clase **singleton** que contiene:

- La lista de tuplas (*tuples*) seleccionadas por el usuario, ya mencionada en el punto anterior.
- Una lista de valores (*varValues*) correspondiente al valor de cada tupla en el frame anterior.
- Un array bidimensional de capas *(activeLayers),* mapeado con las capas activas que están sonando.
- El paquete (*package*) elegido

- El identificador (*ClientId*) para conectarse a SuperCollider

Al iniciarse el juego, este script recoge la información guardada por el usuario desde **MMWindow** y las manda a SuperCollider una vez se establece la conexión (explicado en el apartado 2.4: Comunicación Unity-SuperCollider).

En este momento, la escena comienza y **la música empieza a reproducirse** a la espera de que alguna de las propiedades contenidas en la lista de tuplas cambie su valor.

Esta comprobación se hace en cada vuelta del bucle *update()*. Cuando el propio desarrollo del juego haga que cambie el valor de una de ellas, se dan una serie de pasos, mostrados en la figura 2.15:

1. Se actualiza la copia local de la propiedad
2. Se procesa el mensaje, distinguiendo el tipo de dato que contiene la propiedad para ajustarlo al estándar de los mensajes.
3. Se manda el mensaje a SuperCollider

```
//La propiedad ha cambiado desde el frame anterior
if (!actualVal.Equals(varValues[i]))
{
    //Actualizamos la copia
    varValues[i] = actualVal;
    //Procesamos el mensaje en función de la tupla
    message, arg = ProcessMessage(tuples[i]);
    //Lo mandamos a SuperCollider
    OSCHandler.Instance.SendMessageToClient(ClientId, message, arg);
}
```

**Figura 2.16 Versión simplificada del bucle principal de MusicMaker – *Fuente propia***

Con estos 3 simples pasos, podemos tener control sobre el cambio de las propiedades en la escena a la vez que gestionamos la información que queremos mandar a SuperCollider.

Una vez termina el juego, el plugin cierra la conexión con el cliente de SuperCollider y libera la memoria que estaba utilizando.

## 2.3 SuperCollider

Se utilizará SuperCollider como herramienta para poder abordar tanto la generación de música en tiempo real como su capacidad adaptativa. Su servidor permite ejecutar líneas de código en tiempo real, mientras el juego está funcionando, por lo que se puede tener acción directa sobre la música a cada momento.

El protocolo de comunicaciones OSC facilita la comunicación con Unity, permitiendo encapsular así las funcionalidades discretas del código y enviarlas directamente al motor para tener un efecto directo en el videojuego.

Al ser un lenguaje de tipado dinámico y basado en objetos, proporciona una facilidad enorme a la hora de programar y un estilo de trabajo muy similar al estudiado en Desarrollo de Videojuegos, por lo cual, resulta muy familiar al uso (esto nos permitirá cambios en el sonido en tiempo de ejecución, de los cuales se hablará más adelante). Esta característica resulta clave para poder afrontar la generación de música desde un punto de vista más cercano al del compositor, puesto que podremos usar la idea de objeto para desarrollar cada generador de audio y las funciones para poder establecer cambios sobre el sonido.

Por otra parte, SuperCollider es una plataforma pensada para la síntesis de audio y la composición algorítmica, por lo que contiene funcionalidades propias del terreno musical, como puede ser el uso de la notación MIDI, acción sobre el tempo, selección de escalas y tonalidades, construcción de acordes, etc. Esto último permite al programador utilizar términos que le facilitarán mucho su labor como compositor.

### 2.3.1 Recursos de SuperCollider

En este apartado se detallarán las funcionalidades básicas ofrecidas por SuperCollider que han sido utilizadas en todos los paquetes para generar las piezas musicales (ver capítulo 3).

**- Pbind**: A partir de un par de valores (*key-value*) donde se define el tipo de evento y el instrumento a utilizar, genera un stream constante (por defecto un array de un valor por segundo). Se utilizará para generar **patrones** con sentido ajustados a un tempo definido y que se mantienen en el tiempo.

**- Pseq**: Realiza un número de ciclos determinados sobre una lista dada en orden. En los paquetes será comúnmente utilizado dentro de los Pbind, para poder definir como búfer sonoro una lista de eventos dada. De esta manera se generarán patrones cíclicos que se repiten en el tiempo y mantienen una sucesión ordenada, lo cual será aprovechado para lanzar los golpes percutivos y los ataques melódicos y armónicos siguiendo un patrón constante y estable.

**- Pdef**: Registra los patrones en una *key*, de tal manera que se pueda acceder a la instancia desde cualquier punto del código del paquete. A la hora de reproducir los patrones definidos con el Pdef (acción .play()), se añadirá el argumento **quant**, lo que permitirá que solo se reproduzca cuando entre el primer golpe de compás definido según un tempo dado. Gracias a esta funcionalidad se asegura que todas las capas suenen respetando el compás de la obra y se mantenga un tempo estable.

**- Synth**: Representa una unidad de producción sonora, un sintetizador. Su uso resultará especialmente útil en el campo de los efectos *one-shot*, puesto que permitirá condensar y reproducir la sonoridad de un único sintetizador con una única acción.

**- Prand, Pxrand**: Funciones utilizadas para realizar selecciones aleatorias. Prand escoge un objeto de una lista de manera aleatoria en cada repetición mientras que Pxrand hace lo mismo, pero asegurando que nunca se escogerá el mismo objeto dos veces seguidas.

- **Pexprand, Pwhite**: Generadores de números aleatorios, utilizados sobre todo para proporcionar riqueza sonora a partir de un margen aleatorio en un marco definido dentro de los argumentos de los sintetizadores. Pexprand sigue una distribución exponencial mientras que Pwhite sigue una distribución uniforme.

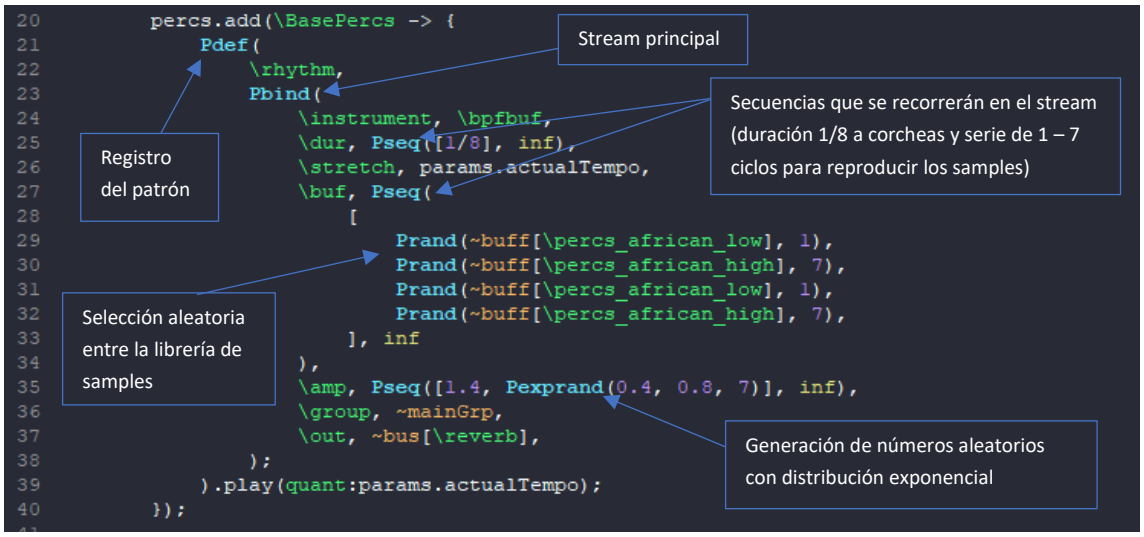



**Figura 2.17 Demostración del uso de recursos de SuperCollider – *Fuente propia***

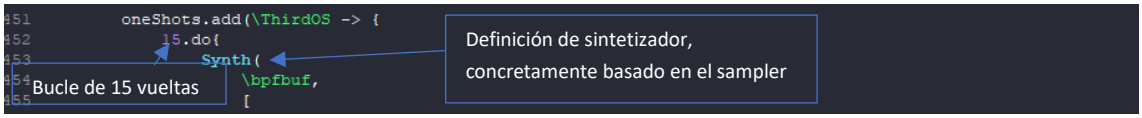



**Figura 2.18 Demostración del uso de recursos de SuperCollider – *Fuente propia***

## 2.3.2 Arquitectura del plugin en SuperCollider

### *Introducción*

Como se menciona anteriormente Supercollider está basado en programación orientada a objetos, por esta razón, en la creación del plugin ha sido necesario establecer una arquitectura basada en clases jerarquizadas que permita la correcta comunicación entre ellas.

### *Clases, Herencia y Polimorfismo*

Para que Supercollider encuentre las clases deben estar almacenadas en un directorio que se creará automáticamente al ejecutar la línea `Platform.systemExtensionDir;` Los archivos pueden estar en formato .scd o .sc pero para que la herramienta los entienda como clases deben estar guardados en formato .sc y no contener código fuera de las propias clases, además el nombre de la clase debe comenzar por una letra mayúscula. A partir de este momento cada vez que se lance Supercollider compilará

automáticamente estas extensiones o podrán recompilarse a mano desde la opción Language >> Recompile class library En la barra de herramientas. Para ejecutar código de Supercollider hay que seleccionar las líneas a ejecutar y pulsar el comando *Ctrl+Intro*. Esto implica que, si se quiere llamar a un método, antes de hacerlo, hay que seleccionarlo y pulsar dicho comando para que la ejecución funcione. Sin embargo, el código de las clases se compila al ejecutar Supercollider lo que permite que solo se tenga que ejecutar la línea que inicializa el plugin. Es por esta razón que algunas clases solo inicializan variables globales, para poder ser usadas en otras partes del código.

Las clases de Supercollider pueden heredar de otra clase mediante el operador ':' de esta manera se adquieren todas los atributos y métodos de la clase padre y se puede acceder a ellos desde la clase hija.

Las clases de Supercollider también permiten el polimorfismo, así pues, si el padre implementa una funcionalidad que el hijo no reimplementado al hacer una llamada a dicha funcionalidad la llamada se realizará sobre el método heredado del padre, pero, si el hijo si ha reimplementado dicha funcionalidad será a esa pues a la que se llame.

Todas las clases heredan de una superclase y la creación de esta se hace mediante la función new de la siguiente manera: var = SomeClass.new; Para crear una constructora distinta y hacer que esta reciba argumentos habrá que rescribir el método new de la superclase.

*Clases del plugin en Supercollider*

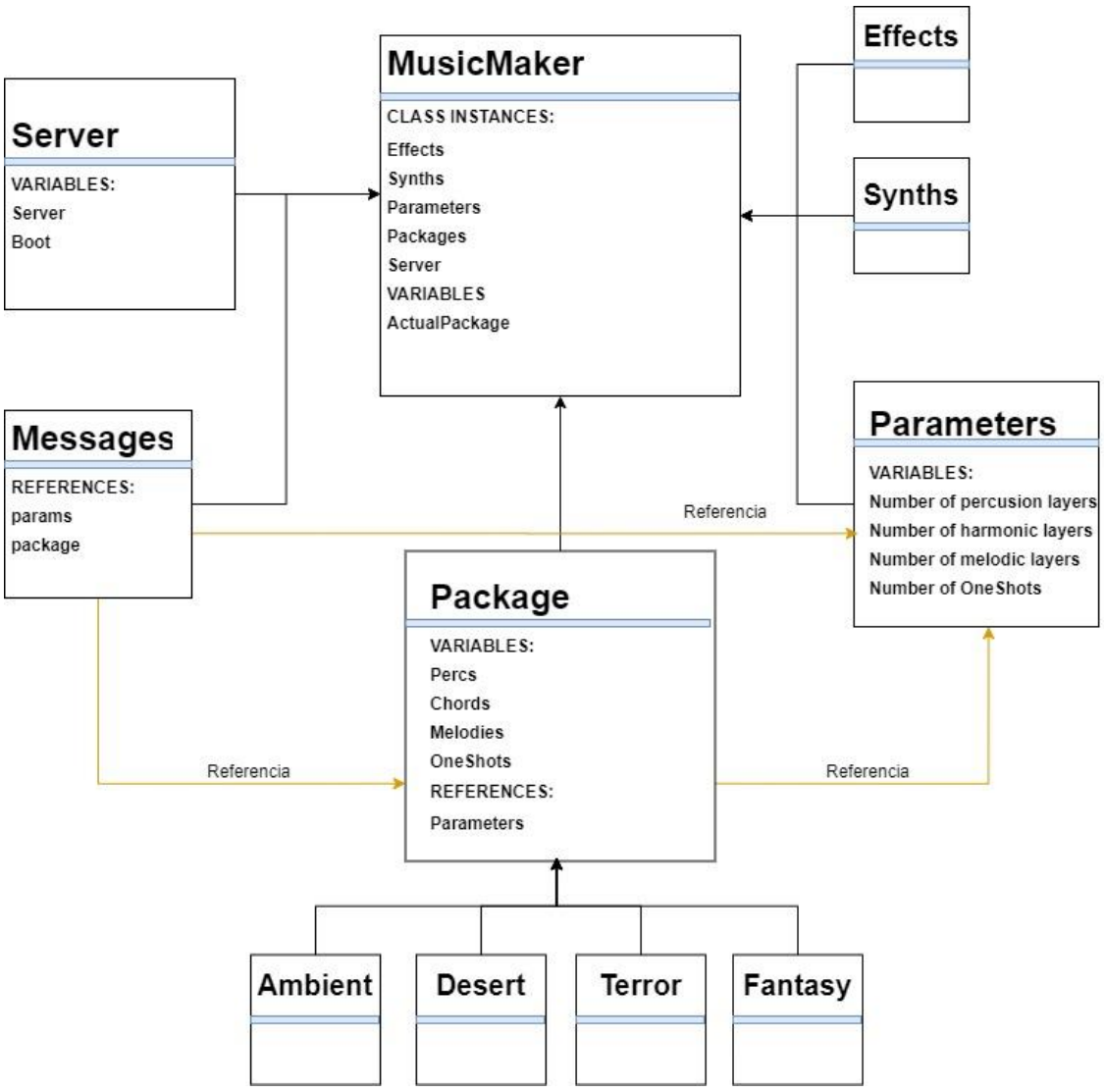



**Figura 2.3 Diagrama de clases en SuperCollider**

En la figura anterior de puede visualizar como están distribuidas las clases que conforman el plugin desde el punto de vista de Supercollider. Al iniciar el juego desde Unity se llama a un mensaje que crea la instancia de MusicMaker y llama al método inicializador de la clase, inicializando el servidor, los efectos, parámetros, sintetizadores, y el paquete que recibe como parámetro desde Unity. La llamada se recoge en el archivo startup.scd que contiene una porción de código con el mensaje que inicializa el MusicMaker al ser llamado desde Unity.

### *MusicMaker.sc*

La clase MusicMaker es la clase que engloba toda la funcionalidad. Se encarga de iniciar todas las clases que conforman el plugin. El servidor tarda un tiempo en lanzarse y es necesario saber cuándo está operativo para crear la conexión con el cliente de Unity además de inicializar cada uno de los paquetes. Para ello el servidor proporciona el método `s.waitForBoot()` que recibe como parámetro una función "*callback*" que será llamada una vez se inicialice el servidor. Este método se llama desde esta clase después de iniciar el servidor.

### *Server.sc*

Es en esta clase donde se crea un servidor local y se configura para poder empezar a reproducir audio. Además, en esta clase se encuentran los métodos que crean los *buffers* de audio, los Buses y los grupos de reproducción ya que deben añadirse al arrancar el servidor.

### *Parameters.sc*

Es la clase que se encarga de almacenar los parámetros adaptativos que afectan directamente sobre la generación de la música. Esta clase se pasa como parámetro a los distintos paquetes musicales para que puedan hacer uso de ellos.

### *Messages.sc*

Esta clase posee los mensajes de tipo OSC genéricos para cualquier paquete musical. Es por eso por lo que posee una referencia al paquete que está sonando actualmente, para saber sobre cual debe actuar.

### *Effects.sc*

Es la clase que se ocupa de inicializar los efectos en forma de variables globales. También almacena el método de generación de acordes automático que es almacenado en una variable global para poder ser ejecutado desde cualquier parte.

### *Package.sc*

Es la clase padre de la que heredan los paquetes específicos de cada tópico. Posee las listas de percusiones, acordes, melodías, "*one shots"* y una referencia a los parámetros. Además de poseer métodos de funcionalidad genérica que pueden ser sobrescritos por las clases hijas en caso de necesitar una implementación específica. Las clases que heredan de esta clase son:

- FantasyPackage.sc
- DesertPackage.sc
- AmbientPackage.sc
- TerrorPackage.sc

## 2.4 Comunicación Unity - SuperCollider

Ahora que ya hemos introducido tanto Unity como Supercollider vamos a explicar cómo se comunican ambas herramientas.

La comunicación entre estos dos sistemas fue el primer problema a resolver. Jorge García Martin es el autor de un conjunto de archivos que implementan la conexión entre ambas plataformas a través de mensajes OSC y están disponibles en el siguiente repositorio de GitHub: https://github.com/jorgegarcia/UnityOSC.[1]

[1] Estos archivos son de libre uso, modificación, distribución y publicación, además de estar permitido sublicenciar y vender copias del software. La única condición es mantener el aviso de copyright anterior. Su sistema es genérico para conectar Unity con cualquier plataforma capaz de enviar y recibir datos utilizando el protocolo OSC.

### 2.4.1 Comunicación desde Unity

Open Sound Control (**OSC**) es un protocolo de comunicaciones que permite compartir información musical en tiempo real sobre una red. En nuestro caso lo usamos para abrir un túnel de comunicación entre Unity y Supercollider.

Para mejor comprensión vamos a separar la explicación del funcionamiento desde el punto de vista del motor Unity y el punto de vista de Supercollider.

La instalación de la extensión en Unity es muy sencilla, es suficiente con arrastrar los archivos (**.cs**) a una carpeta del proyecto y pueden empezar a usarse. Los archivos principales son:

**OSCPacket:** Conjunto de datos serializables para ser enviados como una secuencia OSC. Tiene métodos para convertir la información a secuencias de bytes y para realizar el proceso opuesto.

**OSCMessage:** Hereda de **OSCPacket** y amplía su funcionalidad.

**OSCClient:** Se encargan de enviar mensajes OSC a la dirección y puerto especificado en la constructora. Para ello se crea un cliente UDP que utiliza la clase **UdpClient** del espacio de nombres **System.Net.Sockets** y se conecta usando la IP y el puerto indicados. Para enviar mensajes el cliente posee un método **Send(OSCPacket packet)** que convierte la información de los paquetes OSC a un array de bytes y lo envía.

**OSCServer:** Se encarga de recibir los mensajes OSC. Para crearlo hay que especificar un puerto y se conecta a través de UDP. Posteriormente se crea una hebra que se encarga de recibir los mensajes.

**OSCHandler:** Se trata de un singleton para controlar las conexiones del cliente y servidor, así como como el envío y la recepción de datos OSC. En la inicialización del controlador se crean los clientes que transmiten los datos OSC. A estos clientes se les asigna un ID, una IP y un puerto y después se crea el servidor con un ID y un puerto. Se pueden abrir tantos clientes y servidores como se quiera y se almacenan en un diccionario. Para nuestro caso necesitamos un único cliente y un servidor. El código de

este archivo ha sido ampliado por nosotros para poder obtener una funcionalidad más cercana a la que necesitábamos.

**OSCReciver:** Se trata de un receptor de OSC simplificado para recibir los mensajes en una cola de tipo OSCMessage segura para subprocesos. Permite poner un candado a la cola de mensajes para evitar que la hebra que recibe estos mensajes la utilice al mismo tiempo y extrae y devuelve la información recibida en orden.

De esta manera ya tenemos creado un sistema que gestiona la conexión desde el punto de vista del motor de videojuegos permitiendo serializar, decodificar, enviar y recibir mensajes. Ahora para enviar un mensaje desde Unity solo hay que hacer una llamada al controlador OSC. Indicando el cliente, ruta y los datos que se quieren enviar: OSCHandler.Instance.SendMessageToClient("nombre del cliente", "ruta", "datos"). La ruta es el nombre por el que después SuperCollider sabrá a que mensaje hay que llamar.

### 2.4.2 Comunicación desde Supercollider

Para recibir mensajes solo hay que crear una instancia del tipo **OSCdef** cuyos argumentos son:

- **Clave**: Clave con la que se almacena el OSCdef en la colección global.
- **Función**: Función que ejecuta el mensaje. Le llega como parámetros un array de valores, así como la marca de tiempo, la dirección IP del remitente y el puerto del cual se ha recibido el mensaje.
- **Ruta**: Indica la ruta de dirección OSC de la instancia que se crea. Cuando Unity envié un mensaje se llamará al mensaje que coincida con la ruta.
- **ID fuente**: Es opcional, se trata de la dirección IP del remitente, si se establece, este objeto solo responderá a los mensajes de esa fuente.
- **Puerto de recepción**: Indica el puerto por el cual se recibirán los mensajes.

Solo con la creación de esta instancia Supercollider está preparado para recibir mensajes

Para enviar información desde Supercollider a Unity ser realiza mediante el objeto **NetAddr**. Para establecer comunicación con otra aplicación necesitas conocer cuál es el

puerto que está escuchando la aplicación, en nuestro caso está escuchando a través del puerto 7771. Para enviar un mensaje solo hay que crear la instancia indicando el puerto y la dirección y una vez creada hacer una llamada al método que envía un mensaje a la ruta indicada con el mensaje.

Toda la información aquí explicada puede consultarse en la documentación de Supercollider.

# CAPÍTULO 3 – ALGORITMOS DE COMPOSICIÓN

## 3.1 Música procedural en SuperCollider

*Para explicar este apartado, será necesario abordar cuestiones musicales de nivel básico-medio.*

### 3.1.1 Distribución por paquetes

Para poder abordar las cualidades generativas de los algoritmos y conseguir encapsular la música que va a producir el plugin dentro de un contexto cerrado, se desarrollará la idea de **paquete** musical.

Un paquete contiene la funcionalidad al completo para poder producir música con sentido en un mismo marco artístico, es decir, música de un estilo concreto. La idea de paquete es común e invariable entre todos los estilos, todos los paquetes realizados en este plugin ofrecen las mismas posibilidades, la diferencia radica en que cada uno **debe sonar diferente** y debe ser coherente con el estilo al que hace referencia.

### 3.1.2 Acción y superposición de capas

Para poder entender el funcionamiento de un paquete es necesario conocer la idea de **capa**. El MusicMaker genera música a partir de capas sonoras con sentido compatibles entre sí.

Las funcionalidades que se deben implementar en estas capas se pueden resumir en:

- **Percusiones**: Aportan sentido rítmico a la pieza musical, su labor será esencial para determinar la intensidad y el pulso de lo que está sonando. El tempo y el compás que deberán seguir dependerá directamente del estilo musical a tratar.

- **Acordes**: Un acorde consta del sonido producido por tres o más notas sonando simultáneamente y cuya funcionalidad es aportar sentido armónico. Dependiendo del tipo de música, se necesita una armonía (secuencia de uno o varios acordes) concreta que será fuertemente determinada por el contexto de la pieza y sus características.

- **Melodías**: Debe ser coherente dentro del contexto armónico proporcionado por las capas de acordes. Su funcionalidad consiste en extender y complementar las capas anteriores. Una melodía se genera a partir del sonido de notas únicas y de varias alturas distribuidas a lo largo de una línea de tiempo.

- **Efectos**: Esta última funcionalidad está fuertemente ligada al mundo de los videojuegos pues su objetivo consistirá en aportar *feedback* al usuario a partir de elementos sonoros que tendrán lugar dentro de la propia pieza.

El origen de esta división se debe a que, de manera teórica, la música se puede desglosar esencialmente en percusión, armonía y melodía. Esta división por bloques es muy habitual y representa la manera en la que se entienden la composición y la improvisación.

El bloque de efectos extra se debe a la estrecha relación existente entre los videojuegos y la interacción con el usuario. El plugin necesita proporcionar sonoridades cortas y directas, lanzables en momentos muy concretos y sin una musicalidad tan concreta (pueden ser simples efectos de audio) que servirán para mostrar reacción a las acciones del usuario.

Todos los paquetes ofrecerán al usuario la posibilidad de generar tres capas percusivas, dos capas armónicas, tres capas melódicas que dialogan entre sí y cinco capas de efectos. Cada capa es independiente, pero se acopla perfectamente a las demás. De esta manera, el usuario podrá generar y reproducir capas conjuntamente, alternándolas a su gusto y generando sonoridades en función de lo que se quiera transmitir en el videojuego en cada momento.

Una capa de percusión, armonía o melodía se puede dejar sonando indefinidamente, reproduciendo y pausando a decisión del usuario, mientras que las capas de efectos, de

tiempo corto, podrán ser lanzadas en un momento determinado, pero se pararán automáticamente cuando termine de sonar dicho producto sonoro.

Todos los paquetes implementan la misma funcionalidad, permiten reproducir y parar cada una de las capas sonoras detalladas anteriormente. Además, permiten cambiar el tempo de toda la pieza simultáneamente, para que este pueda adaptarse al contexto necesario para el videojuego (salvo excepciones que se detallarán más adelante). Estas funcionalidades de generar capas sonoras y actuar sobre ellas se implementarán a través de **métodos**. Por una cuestión logística y de arquitectura, todos los métodos tendrán la misma función y se llamarán igual (Ejemplo: reproducir cierta capa de percusiones, parar cierta capa de armonía, aumentar tempo de la pieza…). De esta manera, la diferencia y la riqueza de cada paquete residirá en **cómo se implementa cada método**, ajustándose a las características del estilo y al conjunto de la pieza. Cada paquete implementará los métodos de una manera concreta y diferente, explorando qué algoritmo se adapta mejor a las necesidades de la capa. Para poder cambiar el tempo de la pieza en tiempo real será necesario redefinir el método afectado en otra función a parte cuya diferencia radica en la forma en la que se lanza en la obra (espera a que termine el compás y se reincorpora sobre el método original pero con el tempo cambiado). Además, el estudio de qué necesidades puramente musicales, como pueden ser las escalas utilizadas, los acordes, los compases y la armonía en general, así como la exploración en la riqueza tímbrica de acuerdo con las particularidades y el contexto histórico de cada género, será lo que marque una clara diferencia entre los paquetes haciendo que cada uno suene único y coherente.

### 3.1.3 Contexto musical

Antes de explicar cada paquete en concreto, es necesario conocer qué estilos se han decidido abordar y por qué resultan interesantes. El plugin cuenta con la implementación de **cuatro paquetes**: **desértico, ambiental, fantasía** y **de terror**.

La explicación de por qué se han decidido implementar estos estilos radica en un previo estudio acerca de cuáles son los estilos más habituales en música para videojuegos. El

plugin debe ser lo más versátil posible y cubrir un elevado rango de potenciales videojuegos. Estos cuatro estilos pueden abarcar las sonoridades más típicas de la industria en cuanto a una cuestión de tópicos.

El paquete desértico se puede utilizar en aquellos juegos ambientados en sitios como Asia Occidental o África del Norte, música con clara influencia de la cultura árabe, india, turca o mesopotámica. Servirá para ambientar aquellos videojuegos que se sitúen en lugares basados en dichos emplazamientos, muy típico en juegos de acción-aventuras, niveles especiales de plataformas, juegos de rol, etc.

El paquete ambiental cubrirá un área muy amplia, ideal para juegos con tendencias minimalistas y un estilo difuso que necesiten música de fondo, pero sin ningún remarque de estilo o desarrollo motívico. Su función será generar música agradable al oído y que consiga pasar desapercibida, sin cadencias fuertes ni melodías marcadas.

El paquete fantástico cumple su función para aquellos juegos de rol o aventuras con un toque mágico, aportando al conjunto una sonoridad mística con un ritmo muy característico basado en la división ternaria y melodías que siguen la concepción cultural relacionada con el mundo fantástico en oriente y en occidente. Este es uno de los géneros más explorados en el mundo de los videojuegos por lo que era necesario el desarrollo de un paquete completamente dedicado.

Por último, el paquete de terror complementará todos aquellos juegos con una ambientación tenebrosa o que requieran de música que genere tensión y cause malestar o nerviosismo en el usuario. Este es otro género muy explotado en la industria, el cual, al ser tan cerrado y concreto, necesita una implementación propia y exclusiva capaz de generar música que se acople perfectamente a esta idea.

Para poder desarrollar los paquetes dentro de un marco coherente y con sentido, se necesita conocer con exactitud qué tópicos conforman cada género musical y cómo se traslada la teoría musical a código escrito en lenguaje SClang. Más adelante, se responderán estas cuestiones individualmente para cada paquete, además de una explicación ordenada de los algoritmos seguidos para la creación de las distintas capas musicales y los recursos generales utilizados en cada paquete.

## 3.2 Implementación con SCLang

### 3.2.1 Recursos generales

Antes de explicar cada paquete por separado, se realizará un análisis de los recursos generales (todos los paquetes requieren de ellos) utilizados para la generación de sonidos en el plugin.

En primer lugar, se explicarán los efectos utilizados en los paquetes, su sentido y su implementación. El primero de los efectos para tener en cuenta es la **reverberación**, un efecto básico e imprescindible para cualquier pieza musical. La reverberación es un efecto muy utilizado en grabación musical que consiste en la permanencia durante un determinado periodo de tiempo del sonido una vez que la fuente original ha dejado de emitirlo. Este efecto será aplicado a todas las capas sonoras de todos los paquetes y su objetivo consiste exclusivamente en lograr una mayor profundidad y apertura del sonido, haciéndolo menos seco, más natural y agradable para el oído. En cuanto a la implementación, la reverberación será un SynthDef que procesará la señal de entrada (conocida como "dry signal") y enviará un output de esta señal con dicho efecto aplicado ("wet signal"). Se puede pensar en la reverberación como la emulación de las reflexiones de las ondas en un espacio determinado. Desde el punto de vista de programación en SuperCollider es el resultado de una suma de muchos feedbacks, donde los parámetros del delay dependerán del tamaño de la onda, su forma y las características del espacio en el que rebote la señal (dimensiones, materiales, coeficientes de absorción, reflexión, etc). Conociendo esta idea el último paso será la implementación. En primer lugar, se tomará la señal Dry del bus de entrada y se definirá una variable temporal. En esta variable se irán guardando cada uno de los delay feedbacks con sus respectivos parámetros durante un número determinado de iteraciones (se tomaron 16). El resultado de cada iteración se acumulará en la señal Wet. Por último, la señal definitiva de salida consistirá en una mezcla balanceada de ambas señales wet y dry, que se enrutará al bus de salida correspondiente.

El otro efecto desarrollado para el plugin consiste en un **generador de acordes**. Se decidió establecer una implementación que permita generar una serie de acordes con

sentido dentro de una tonalidad y un modo dados para explorar las posibilidades del lenguaje y a modo de investigación. Se comprueba su funcionalidad en el paquete de terror que será explicado más adelante. Para su implementación, se utilizará un array base que contendrá las distancias entre notas existentes en los diferentes modos de una tonalidad, un índice que indicará el modo concreto y la nota que definirá la tonalidad. A partir de estos datos, en otro array auxiliar se generarán diferentes acordes, juntando las notas generadas en el array base (acordes de entre tres y cinco notas) de manera aleatoria.

Una vez explicados los efectos, es necesario detallar y conocer el funcionamiento de los **sintetizadores** utilizados en el plugin (todos los instrumentos se conocen como sintetizadores en el ámbito de SuperCollider). A partir de los tres sintetizadores programados se generarán todas y cada una de las muestras sonoras que se puedan escuchar en las piezas musicales. Un sintetizador es, básicamente, un instrumento musical electrónico (oscilador) capaz de generar señales eléctricas que posteriormente son convertidas a sonidos audibles.

El primero de los sintetizadores desarrollados en el plugin consiste en un oscilador que genera ondas simples de **diente de sierra** (lo llamaremos sintetizador de diente de sierra), uno de los tipos más utilizados y versátiles, especialmente en el campo de los videojuegos. Como su nombre indica permite generar ondas con forma de picos de sierra y que, a través de la edición de sus parámetros, creará la gran mayoría de sonidos nativos (creados desde 0, sin samples, siguiendo la definición de síntesis) por parte de cada paquete. Su implementación se basará en los recursos básicos proporcionados por el propio lenguaje, como son los generadores de frecuencias y la posibilidad de control sobre la señal, que permitirá enriquecerla con parámetros como el paneo, el *detune*, las frecuencias de corte, etc.

El siguiente sintetizador es un **sampler**. Su función consiste en ofrecer la capacidad de lanzar fragmentos de audio a partir de archivos pregrabados. Resulta extremadamente útil para conseguir sonidos adaptados y coherentes con cada género en concreto, ya que el hecho de poder reproducir archivos previamente grabados y editados permitirá conseguir timbres y sonoridades mucho más ricas y cuidadas que serían imposibles de conseguir a partir de sintetizadores más básicos. Este sampler se utilizará en todos los

paquetes, especialmente para generar pistas de percusión (ya que estas necesitan sonidos muy concretos, como puede ser un golpe de caja, de bombo, de pandereta…) y melodías (estas necesitarán fragmentos de notas con sonoridades muy características como pueden ser flautas, violines, etc). Todos los archivos que puede reproducir este sampler, se encuentran en una carpeta llamada "buffers" situada a la altura de los archivos de SuperCollider.

El último de los sintetizadores es un **órgano**. El objetivo de este sintetizador consiste en ampliar las sonoridades anteriores, permitiendo generar un sonido de cero (sin samples) pero más rico y concreto que el sintetizador de diente de sierra. El origen de este instrumento es externo, está hecho por un miembro de la comunidad de SuperCollider que posteriormente publicó su implementación en Internet. Se decidió incluirlo en el plugin para demostrar que resulta muy sencillo adaptar sintetizadores externos y extender el rango de sonoridades, por si en el futuro algún usuario decide ampliar el plugin y los paquetes, pueda contar con esta opción comunitaria. Su funcionalidad se demostrará y probará en el paquete de terror.

A continuación, se detallará la implementación concreta de cada paquete, cómo se genera cada capa y por qué se han decidido escoger los timbres y el contexto armónico utilizados.

### 3.2.2 Paquete desértico

Este paquete condensará tópicos relativos a la musicalidad relativa al norte africano y oeste asiático. Para conseguir un acercamiento a esta cultura, será necesario el uso de timbres relativos a sus costumbres musicales. De esta manera, se utilizarán sonidos de percusiones tradicionales africanas a modo de samples para desarrollar las capas percutivas. Se definirá un tempo medio de 128 beats por minuto y un compás 4/4 con golpes definidos a velocidad de corcheas.

La primera de las capas percusivas servirá para definir el **pulso** de la pieza, marcando con golpes graves y contundentes los acentos del compás y rellenando el resto de los golpes con percusiones más agudas. Todos los sonidos percusivos serán resultado de

samples de djembé pregrabados y lanzados a través del sintetizador definido anteriormente.

La segunda capa, será una extensión de la primera, con la misma filosofía e instrumento, pero con golpes tímbricas más graves. Ambas capas cuentan con una librería de samples diversa, con varios sonidos grabados de cada instrumento, pero con ligeras variaciones sonoras que serán elegidos de manera aleatoria, para conseguir una mayor riqueza sonora.

La última capa de percusiones consiste en una serie de maracas étnicas sonando a un ritmo constante e invariable, pero con ligeras variaciones entre cada sample (como el caso anterior) para enriquecer, dar cuerpo y completar la percusión al completo, pero a la vez poder funcionar como elemento por separado.

En cuanto al contexto melódico y armónico, las composiciones se basarán en la **escala doble armónica**, ya que es la escala más representativa y utilizada en la música relativa a la cultura que recoge el paquete.

Para el desarrollo armónico y el cuerpo del tema se utilizará el sintetizador de dientes de sierra, puesto que en este campo buscaremos exclusivamente ganar cuerpo y contexto musical. La primera capa consta de un acorde de do menor (primer grado de la escala doble armónica) mantenido y constante. La segunda capa armónica alterna, entre el primer y el segundo grado de la escala una octava por encima a la capa anterior. Esta progresión, aunque resulta básica en cuanto a cantidad de acordes, es completamente funcional y representativa dentro del contexto utilizado, ya que muestra la sonoridad de segunda menor presente entre el cambio del primer acorde al segundo (siendo este intervalo el más representativo de la escala) y la séptima mayor presente en el segundo acorde (siendo este el segundo intervalo más representativo de la escala).

La primera capa melódica, utilizará los samples pregrabados de la escala doble armónica de do, con timbre de cuerda frotada y mantenida. Esta capa escogerá notas aleatorias de dicha librería de samples con duración de negra. Incluirá la opción dentro de esta librería de no tocar ninguna nota en el momento de lanzar el sample, que será situación equiprobable entre el resto de las notas a escoger.

La segunda capa, tendrá definidos una serie de clichés (frases musicales), típicos dentro del contexto musical que se está siguiendo en el paquete. De entre todas estas frases, que contienen secuencias de notas históricamente relacionadas con la cultura desértica, se escogerá una al azar al comenzar cada compás. Para este caso en concreto se utilizará como sample el sintetizador de dientes de sierra para no confundirse con la capa anterior ni pisar su franja de frecuencias.

La última capa realizará un recorrido ascendente de la escala a velocidad de semicorcheas, utilizando como librería de sonidos la misma que la primera capa, pero con una duración más recortada, evitando así solapamientos.

En último lugar, los one-shots (llamamos así efectos dentro del contexto de SuperCollider) de este paquete, también buscan encontrarse dentro del marco cultural y musical de la pieza. Todos los efectos se lanzarán a través del sampler y tendrán una duración corta puesto que su función es muy puntual y con un final definido, no se pueden mantener. El primero de los efectos será un slide o glissando ascendente, aprovechando la sonoridad de las cuerdas frotadas utilizadas en las capas melódicas. Servirá para indicar un hecho puntual en el videojuego, pero manteniendo la cohesión de lo que está sonando de fondo, respetando la sonoridad de la obra. El segundo de los efectos consiste en un slide al igual que el anterior, pero ubicado en otra zona diferente de la escala, para poder trabajar en conjunto con el primero de los one-shots y a la vez contar con una variación sonora que permita al usuario utilizarlo también en contextos diferentes. El tercero de los efectos será un conjunto sonoro finito formado por shakers, trabajando en conjunto con la sonoridad de las capas percutivas.

### 3.2.3 Paquete ambiental

Este tipo de música debe resultar agradable para el oyente y a la vez encontrarse en un segundo plano constante, pues su objetivo es aportar un colchón sonoro de fondo sin ninguna clase de protagonismo.

Para las capas percutivas se utilizarán sonidos suaves y poco agresivos, lo que complementa la idea general buscada en este paquete. Además, la pieza no se apoyará

en ninguna clase de tempo fijado ni distribución de compases concreta, puesto que se busca cierta aleatoriedad en la base rítmica. Esto es debido a que se evita la identificación de patrones rítmicos y repeticiones en la pieza por parte del oyente, ya que el reconocimiento de estos es inherente a la condición humana. Si evitamos dichas características, alejaremos su atención de la sonoridad generada ya que no la identificará como una musicalidad protagonista.

La primera de las capas utilizará el sintetizador de diente de sierra de una manera menos convencional, logrando sonidos completamente diferentes a los esperados gracias a la especificación de valores más inusuales en los argumentos. Utilizando frecuencias situadas por debajo de la franja audible para los humanos (menos de 20hz de frecuencia), solo se pasará a escuchar el ritmo del propio ciclo individual de la onda. Utilizando este recurso, el sintetizador pasará de generar frecuencias audibles al uso a generar ritmos, sirviendo, así como generador de percusiones. Esto unido a los filtros de frecuencias proporcionados por el sintetizador, la aleatoriedad de parámetros como la frecuencia base o la duración y la aportación de la reverberación, permite emular el sonido de una marimba, el cual se acopla perfectamente a la sonoridad buscada y se adapta a partir de patrones en un marco aleatorio como se definió anteriormente.

La segunda capa de percusiones aportará cuerpo a la pieza a partir de un *sustain* de campanas conseguido a partir de la reproducción de samples. Todos los argumentos editables del sintetizador se encuentran dentro de un marco aleatorio, consiguiendo así una sonoridad más rica y variada en el tiempo.

La última de las capas, emula el sonido de un dron, un efecto grave, contundente y mantenido en el tiempo. El motivo de esta sonoridad radica en que este paquete no debe sobrecargar la labor percutiva, siendo esta bastante minimalista en cuanto a estímulos. Para conseguir esto, se utilizará el sintetizador de dientes de sierra, pero en esta ocasión, utilizando el do más grave del teclado como nota base y la modificación de parámetros como el *detune* y los filtros de frecuencias para emular la vibración de la onda.

En cuanto a la armonía base, se buscarán timbres suaves, poco agresivos y mantenidos en el tiempo. Para lograr esto se utilizará el sintetizador de dientes de sierra, con muy

poco ataque, mucho *release* y una reverberación muy presente (se cuenta con la posibilidad de manejar en e l sintetizador unos parámetros de envolvente ASDR *Attack-Sustain-Decay-Release*).

La capa principal se basará en una progresión de cuatro acordes. Una de las mayores características de la música ambiental radica en el uso de acordes con distancias interválicas muy grandes, superiores a una octava, por lo que los acordes generados cumplirán esta condición. Los acordes se formarán a partir de intervalos amplios, sin embargo, las distancias entre notas en cada paso de acordes no serán muy grandes para evitar llamar demasiado la atención del oyente (esta es otra de las características más destacadas de la música ambiental, se evitan cadencias agresivas). La progresión de la capa generada se basará en estos comportamientos, logrando así un colchón armónico típico de este género.

La segunda de las capas armónicas será una extensión simplificada de la capa anterior, basando su sonoridad en mantener un único acorde abierto, constante y con frecuencias más agudas para no enmascarar el resto de los sonidos.

Este paquete no contempla la generación de líneas melódicas. Esto es debido por el mismo motivo que han seguido el resto de las capas, no se busca captar la atención del oyente en ningún momento si no hacer música de fondo, que pase desapercibida. Si hubiera melodía, se atentaría directamente con la definición anterior, puesto que resultaría muy llamativa para el usuario y desviaría su atención a la pieza.

Por último, los one-shots utilizados, se complementarán con la obra siguiendo las características tímbricas seguidas en las demás capas.

El segundo y el tercero serán un conjunto de shakers que se irán desvaneciendo con el tiempo. Cada uno de ellos se origina a partir de la misma librería de samples, sin embargo, los valores de sus argumentos son distintos, lo que provoca dos sonoridades complementarias a la par que distintas. El primero de ellos jugará con valores de sustain más cortos en el tiempo mientras que el segundo tendrá la capacidad de variar las frecuencias de manera aleatoria dentro del marco que sigue la distancia interválica de una escala mayor concreta.

### 3.2.4 Paquete fantástico

Este paquete es el más abierto a nivel de posibilidades y de usabilidad. Su objetivo es generar música que suene a fantasía, lo cual, es un término muy amplio y variante según diferentes culturas y situaciones. Para su desarrollo se ha decidido utilizar los clichés más característicos y comunes entre todos los tipos de fantasía, como pueden ser los timbres de instrumentos de madera, tanto para percusiones como para vientos o la composición basada en el modo lidio (surge a partir del cuarto grado de la escala mayor).

La pieza está compuesta a 140 bpm y sobre un compás de 3/4, ya que, históricamente es el tipo de compás más utilizado en este tipo de música fantástica basada en tribus del bosque y un contexto mágico siguiendo el carácter ternario ligado a la danza. Además, ayudará a proporcionar a la obra una sensación de "trote", comúnmente llamado, lo cual será de mucha ayuda para introducir al jugador en juegos con una acción y movimientos constantes.

Para lograr en la primera capa el sonido percutivo anteriormente nombrado, con un cuerpo hueco y de madera, se utilizarán samples pregrabados de instrumentos hechos puramente a partir de madera, como la caja china o las claves. Con una secuencia a ritmo de corcheas y con los samples graves sonando al comienzo de cada pulso, se generará dicha capa, que aportará el grueso de las percusiones a la obra.

La segunda capa complementará la anterior a partir de samples de grandes campanas de metal. El motivo de este cambio tímbrico radica en que es necesario ganar un pequeño contraste entre las capas para aumentar la riqueza de la obra. El metal resulta opuesto a la madera, es más brillante, al contrario que la madera, que es oscura y opaca, por lo que el contraste será notorio. Sin embargo, será algo exclusivo de esta capa aprovechando que el nivel de golpes se reduce a uno por compás, por lo que no se llevará el protagonismo de las frecuencias, manteniendo y complementando el sonido de la madera.

La última de las capas de percusión, consistirá en unas panderetas siguiendo golpes atresillados, también lanzadas a partir del sampler. Al igual que la capa anterior, el algoritmo que las genera se encargará de elegir samples de la librería de manera

ordenada, diferenciando entre el primer golpe del compás y el resto. Las capas percutivas de este paquete son poco abiertas a la aleatoriedad y la diversificación debido a que son compases muy característicos y de índole cerrada, por lo que se ha preferido optar por la cohesión y la coherencia rítmica en pos de la variedad.

En cuanto a las capas armónicas y melódicas, las composiciones se basarán, como se indicó anteriormente, en la escala mayor lidia (escala lidia de do concretamente). Esta escala es igual a la escala mayor, con la diferencia de que cuenta con un intervalo de cuarta aumentada en lugar de cuarta justa. Es esta diferencia la que producirá una sonoridad mucho más mística y fantasiosa a la pieza, ya que, culturalmente siempre ha sido una escala ligada a prácticas mágicas y misteriosas en el lenguaje cinematográfico, la televisión o los videojuegos.

La primera capa armónica consistirá en la reproducción del acorde triada del primer grado de la escala, otorgando cuerpo y contexto modal a la obra. Ligeras variaciones en los campos de afinación, rangos de frecuencias, ataque y *release* del sintetizador de dientes de sierra, provocarán en el oyente una sensación de movimiento y variedad cada vez que se vuelva a reproducir el acorde sin desviar la atención de su función armónica como primer grado de la escala.

La segunda capa armónica será una extensión de la primera, con la misma funcionalidad y capacidad generativa, pero en una escala superior y con menos presencia. Igualmente, es funcional por separado sin necesidad del apoyo de la capa anterior al igual que todo el resto las capas según su propia definición.

En cuanto a las melodías, la primera de las capas se generará a partir de reproducciones de samples aleatorias de las notas que siguen la escala lidia de do en un rango de dos octavas. En cuanto a timbres se mantiene el viento madera, concretamente el sonido pregrabado de una flauta de caña a duración de negra e incluyendo la posibilidad de que no suene ninguna nota de manera equiprobable.

La segunda capa melódica utilizará el sintetizador de diente de sierra para reproducir aleatoriamente una serie de clichés predefinidos, siguiendo así la idea del paquete desértico.

Por último, la tercera capa añadirá la capacidad de reproducir una sucesión rápida y predefinida sobre la escala lidia con samples, pero esta vez de una duración mucho más corta.

Para terminar con el paquete, los one-shots utilizados son relativos al mundo natural y al mundo mágico, puesto que se complementan fácilmente con las sonoridades seleccionadas en el resto de las capas. Todos ellos se apoyarán en el sampler desarrollado puesto que serán una serie de efectos concretos de una duración fija y grabados con anterioridad. La implementación de todos ellos es común, un lanzamiento básico del sample, la diferencia radica en la sonoridad de este propio sample. El primero de ellos será un sonido de grillos, para mimetizar la obra con el mundo natural, seguido del segundo, basado en un sonido de vientos en el bosque. El tercer one-shot será un efecto especial que emulará el sonido de un hechizo mágico, un recurso que se ha considerado que puede resultar muy útil para los usuarios dentro del contexto de la obra.

### 3.2.5 Paquete de terror

Su objetivo será generar música que transmita una sensación de incomodidad y de temor en el usuario. Para lograr esto a nivel tímbrico se utilizarán sonidos artificiales como sintetizadores digitales con variaciones mínimas entre las frecuencias, sonidos metálicos puesto que son más agresivos para el oído o cuerdas con ligeras desafinaciones y sucesiones poco frecuentes y agresivas.

La obra está compuesta a 40 bpm con compás binario. Esto es debido a que se busca transmitir la sensación de pasos, de estar andando a una velocidad muy baja y firme, para transmitir así al usuario la sensación de que está siendo perseguido a un ritmo constante a todo momento. Las percusiones que se generarán en este paquete son muy poco recargadas, su único objetivo es apoyar la idea anteriormente citada, por lo que se limitarán a marcar partes muy concretas de cada compás.

La primera capa percusiva se limitará a marcar el ritmo a velocidad de corcheas de manera constante. Los golpes sonoros se generarán a partir del sampler, el primero de

ellos se creará a partir de una librería de golpes de bombo de frecuencias agudas mientras que el segundo lo hará a partir de frecuencias graves. Será el factor aleatorio a la hora de escoger cada sample lo que distinga la pieza y añada frescura al sonido generado.

La segunda y tercera capa de percusiones, por el contrario, seguirán una distribución cuaternaria fija que se complementará con la anterior añadiendo samples de grandes campanas metálicas en los pulsos segundo y primero respectivamente. Esto provocará en la obra una sensación de subida de intensidad con el objetivo de aumentar la tensión en el usuario, elevando su nivel de nervios y de intranquilidad debido a las disonancias creadas.

Las capas armónicas de este paquete cuentan con la particularidad de haber sido creadas a partir de un generador de acordes programado por los propios desarrolladores de este plugin (explicado en apartados anteriores).

Para la primera capa se utilizará el generador para reproducir acordes aleatorios dentro de la escala de do menor y a partir del sintetizador de dientes de sierra. La escala menor natural ayuda a generar música que suena "triste" al oído, debido al contexto histórico/cultural y al uso que se le ha dado dentro de la música occidental, de modo que, será muy útil para introducir al oyente en una sonoridad lúgubre y tétrica. Además, el sintetizador cuenta con un argumento "*detune*" o desafinador, el cual se exagera mucho para lograr pequeños cambios de altura entre todas las frecuencias de las ondas que genera, logrando así un sonido muy tenso y agresivo para el oído, perfecto para el contexto que mantiene el paquete.

La segunda capa mantiene la filosofía de la primera a nivel generativo, apoyándose también en el generador de acordes sobre la escala de do menor natural. Sin embargo, a nivel tímbrico utilizará un sintetizador nuevo, un órgano, el cual mantiene una sonoridad artificial y metálica, pero con una forma de onda más redondeada, fusionándose así sin problema con la capa anterior cuando sea necesario. Como se indicó anteriormente en este documento, este sintetizador fue programado por la comunidad de SuperCollider y ha sido incluido en este plugin para mostrar la gran

cantidad de posibilidades a nivel tímbrico que se pueden explorar de cara a futuras expansiones.

Relativo a las melodías, el paquete mantiene la filosofía de permanecer dentro del contexto tonal de la pieza. En este caso, la primera capa melódica se apoyará en el sintetizador de diente de sierra para escoger aleatoriamente una línea entre una serie de clichés determinados.

La segunda capa mantendrá el procedimiento de la anterior, con la diferencia de que utilizará el órgano como fuente tímbrica y una serie de clichés distintos.

La tercera y última capa, a partir del sintetizador diente de sierra, escogerá al comienzo de cada compás una nota de la escala de do menor (escala a partir de la que se construye toda la pieza) situada una octava por encima de las anteriores. Gracias a este comportamiento se generará más énfasis en los compases de la obra sin quitar protagonismo a las líneas anteriores y se conseguirá una mayor riqueza sonora.

Para terminar, en cuanto a los efectos one-shot, se han mantenido las técnicas de buscar sonoridades a partir de los propios sintetizadores o de archivos pregrabados. El primero de ellos utiliza el sintetizador de dientes de sierra para, a partir de una serie de repeticiones y una selección aleatoria de frecuencias muy graves (entre 8 y 60 hz) para conseguir un efecto sonoro penetrante cuyo objetivo es inquietar al usuario. Los dos siguientes efectos utilizarán los samples de una agrupación de cuerdas. Esta agrupación realizará sonoridades a partir de la nota do, al igual que el resto de la pieza, con la particularidad de que añadirá frecuencias externas a la escala (como saltos interválicos de segunda menor a partir del propio do). El objetivo de esto será conseguir tensiones extra y lograr una sonoridad agresiva para el oído hasta que llegue a resultar inquietante y conseguir un efecto de aversión directo sobre el oyente.

# CAPÍTULO 4 - CONCLUSIONES

## 4.1 Conclusiones generales

Una vez desarrollado el plugin, podemos concluir que SuperCollider es una herramienta que demuestra tener un gran potencial en el ámbito de los videojuegos. Sin embargo, se han encontrado varios problemas que han dificultado tanto el desarrollo de la herramienta como el uso que le puedan dar los posibles usuarios. El mayor de los problemas ha sido la imposibilidad de crear un ejecutable del programa de Supercollider que, una vez lanzada una versión ejecutable del proyecto de Unity, se lance Supercollider sin necesidad de ejecutarlo previamente a mano. Además, en estas herramientas se requiere un gran esfuerzo para incluir visualización de datos a mayor nivel y depurar funcionamiento y capacidades.

La música generativa juega un gran papel en los videojuegos, por desgracia está poco extendida y lleva un gran esfuerzo creativo y computacional ponerla en práctica y hacer que se adecue a las necesidades del videojuego. Es por esto por lo que lo más habitual es invertir en compositores que creen obras creativas para cumplir el objetivo de no saturar al usuario.

Las posibilidades generativas de Supercollider resultan extensas, sin embargo, están muy pensadas para géneros de música muy concretos como es la música ambiental. De este modo, producir música que salga de este ámbito sin recurrir a las grabaciones, repeticiones y aleatoriedades resulta muy complicado. Las especificaciones concretas de un género musical, así como sus timbres, clichés y particularidades armónicas chocan directamente con las habilidades proporcionadas por la herramienta debido a la pobreza tímbrica que resulta de la creación de tus propios sintetizadores, así como la necesidad imperiosa de encontrar patrones definidos tanto a nivel rítmico como armónico se enfrenta directamente con las habilidades que ofrece el lenguaje, llenas de algoritmos basados en patrones aleatorios y progresiones no constantes.

## 4.2 Overall conclusions

Once the plugin is developed, we can conclude that SuperCollider is a tool that shows a great potential in the field of video games. However, several problems have been found that have hindered both the development of the tool and its use by potential users. The biggest problem has been the imposibility of creating a Supercollider program executable that, once an executable version of the Unity project is launched, Supercollider is launched whitout the need to previously execute by hand. In addition, a great deal of effort is required in these tools to include a higher level data visualization and debugging of performance and capabilities.

Generative music plays a big role in videogames, unfortunately it is not widespread and it takes a big creative and computational effort to implement it and make it fit the needs of the videogame. This is why it is most common to invest in composers who create creative works to meet the objetive of not saturating the user.

The generative possibilities of SuperCollider are extensive; however, they are well tought out for very specific music genres such as ambient music. Thus, producing music that goes beyond this field without resorting to recordings, repetitions and randomness is very complicated. The concrete specifications of a musical genre, as well are its ¿timbres?, ¿cliches? and harmonic particularities collide directly with the skills provided by the tool due to the ¿timbre? poverty that results from the creation of your own synthesizers, as well as the imperative need to find defined patterns at both the rythmic and harmonic leve lis directly confronted with the skills provided by the language, full of algorithms based on random patterns and non-constant proressions.

## 4.3 Trabajo futuro

El proyecto está preparado para poder hacer avances futuros gracias a la alta compartimentación que lo vuelve muy ampliable por varios aspectos.

Crear un nuevo paquete musical sería tan sencillo como crear una clase en SuperCollider que herede de la clase “package.sc” e implementar tanto los métodos como mensajes propios si se quiere que tenga funcionalidades específicas del paquete. Una vez creado el paquete es suficiente con crear una instancia y añadirlo a la lista de paquetes del MusicMaker y desde Unity añadir un mensaje que indique que ese es el paquete que se quiere ejecutar.

Por otro lado, se puede modificar el modo de generación de música incluyendo algoritmos generativos diferentes, incrementando las capas melódicas y rítmicas para dar mayor variedad o incluso creando y cargando nuevas muestras de audio para enriquecer tímbricamente la composición.

Otro de los objetivos a cumplir en un trabajo futuro consistiría en encontrar una manera de crear un ejecutable del proyecto en SuperCollider.

Se decidió usar Unity por el conocimiento y experiencia acumulado de este motor durante los años de carrera y la experiencia obtenida por el desarrollo de proyectos personales o Game Jams. Esto no implica que sea exclusivo de Unity ya que nuestro sistema de SuperCollider podría funcionar sobre cualquier sistema con capacidad de lanzar mensajes OSC. Esto hace que sea muy ampliable a otros motores y plataformas ya que, además, SuperCollider funciona en los sistemas operativos de MacOS y Linux.

# APÉNDICES

## Repositorios importantes

- Repositorio público con el código e instrucciones del plugin desarrollado:
  https://github.com/ramonarj/MusicMaker

- Repositorio para usar el protocolo OSC desde Unity:
  https://github.com/jorgegarcia/UnityOSC

- Código utilizado en los tutoriales de Eli Fieldsteel:
  https://drive.google.com/drive/u/1/folders/1GJWStaKM2EFRVmHqbjsa5ZKX5iTPWrq5

- Repositorio en el que se ha desarrollado el TFG:
  https://github.com/albcor01/TFG

# Muestras de música generada por el plugin

En este apartado mostramos ejemplos de corta duración de la música generada en SuperCollider:

- Ejemplo del paquete ambiental:
  https://drive.google.com/file/d/1qYkf6YkAjNCKg06YperBz6BrdTGz6Ao0/view

- Ejemplo del paquete de terror:
  https://drive.google.com/file/d/1UZEj3sri12cA4aMG_J2GKPbNfP8oAliP/view

- Ejemplo del paquete de desierto:
  https://drive.google.com/file/d/1DlRgK_E0SmPHRwXnMz2oAYTZ4eA0OWlV/view

- Ejemplo del paquete de fantasía:
  https://drive.google.com/file/d/1lUEhRdgnNbPcemyZ23rDSXdEzVqqkONR/view

# Referencias sonoras

A continuación, se mencionan varias de las obras de autores mencionados a lo largo de la memoria y que han servido de referencia para la realización tanto de la investigación como del desarrollo del plugin.

- Steve Reich - It's Gonna Rain:

  https://drive.google.com/file/d/1QvjG7nhhkqDOSJQNWP08rSaawx8YprRe/view

- Spore ambient Galaxy:

  https://drive.google.com/file/d/11n8hii7qWeQ3K1q7PAGnKUv_ULFpOvIO/view

- Brian Eno - Music for airports:

  https://drive.google.com/file/d/1FJAvCdZCGfLqQqMvJsoIyuaK19jgdF86/view

- Brian Eno – Music for the marble palace

  https://drive.google.com/file/d/1bktlkaO7aJfsnZWGE6JntzMUUNsa1sVj/view

- Brian Eno - Thursday afternoon

  https://drive.google.com/file/d/1FBgbWKybX9VRLAH6Tdevriyo9WvvSN1h/view

- 65daysofstatic – Blueprint for a slow machine

  https://drive.google.com/file/d/14ROeXuf3DIsLV6yWQVnoiYNwhdQ0W_wu/view

# Manual de instalación de MusicMaker

Se indican a continuación los pasos para descargar e instalar MusicMaker:

## Parte 1: SuperCollider

1. Descargar los archivos indicados en el apéndice **"Archivos del plugin"**
2. Acceder a la página web de SuperCollider e instalar la última versión disponible.
3. Abrir el entorno de SuperCollider y comprobar la ruta de la carpeta "*Extensions*". Esto puede hacerse escribiendo la línea Platform.systemExtensionDir; y ejecutándola con el comando "**CTRL + .**".
4. Mover a dicha carpeta los 12 archivos de SuperCollider (extensión.sc) y la carpeta "buffers".
5. Compilar las clases recién pegadas con la opción de Lenguage >> recompile class library.
6. Abrir el archivo de arranque (*startup.scd*). Esto puede hacerse con la opción de file >> open startup file.
7. Pegar en este archivo el contenido del archivo main.scd.
8. Cerrar SuperCollider y volverlo a abrir para que los cambios tengan efecto

## Parte 2: Unity

1. Mover a la carpeta "*Assets*" del proyecto de Unity en cuestión la carpeta "Unity" que contiene los archivos de C# (extensión .cs)
2. Una vez hecho esto, se añade una ventana nueva al editor de Unity donde configurar la música procedural (*Window >> Music Maker*)
3. Configurar los parámetros y comprobar que se crea un GameObject en la escena
4. Configurar, si se quiere, las tuplas en el inspector del objeto creado para la música adaptativa.
5. Al pulsar Play en el Editor debería empezar a reproducirse la música elegida.

# BIBLIOGRAFÍA Y WEBGRAFÍA


*A Year with Swollen Appendices*. (1996).

*Brian Eno Biography - In Motion Magazine*. (1996). Retrieved May 25, 2020, from https://inmotionmagazine.com/eno2.html

Collins, K. (2009). An introduction to procedural music in video games. *Contemporary Music Review*, *28*(1), 5–15. https://doi.org/10.1080/07494460802663983

*GDC Vault - The Sound of "No Man's Sky."* (2017). Retrieved June 4, 2020, from https://www.gdcvault.com/play/1024067/The-Sound-of-No-Man

*Generative Music - Brian Eno - In Motion Magazine*. (1996). Retrieved May 25, 2020, from https://inmotionmagazine.com/eno1.html

Jensen, K. (2010). Investigation on Meter in Generative Modeling of Music. *Proceedings of the 7th International Symposium on Computer Music Modeling and Retrieval (CMMR'10)*, 61–69.

Jolly, K. (2011). Usage of Pd in Spore and Darkspore. *PureData Convention*.

Paulus, E. (2001). *The use of generative music systems for interactive media*.

*Programación: Tutorial videojuegos con Unity 2D en español (Parte 2 Scripts) - Tutoriales bartop arcade, raspberry pi y videojuegos. Proyectos retro consolas spectrum*. . Retrieved June 14, 2020, from https://retro.relaxate.com/tutorial-documentacion-manual-programacion-videojuegos-unity-2d-parte-2/

*Propiedades: Guía de programación de C# | Microsoft Docs*. Retrieved June 14, 2020, from https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/properties

*Reflexión (C#) | Microsoft Docs*. Retrieved June 14, 2020, from https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/concepts/reflection

Schwarz, K. R. (1980). Steve Reich: Music as a Gradual Process: Part I. *Perspectives of New Music*. https://doi.org/10.2307/832600

*SuperCollider 3.10.4 Help*. Retrieved May 31, 2020, from https://doc.sccode.org/Help.html

*This Steve Reich visualisation will melt your musical mind - Classic FM*. Retrieved June 20, 2020, from https://www.classicfm.com/artists/steve-reich/music/piano-phase-visualisation/

*Timeline y la ventana Inspector - Unity Manual*. Retrieved June 14, 2020, from https://docs.unity3d.com/es/2018.4/Manual/TimelineInspector.html

*Unity - Quick Guide - Tutorialspoint*. Retrieved June 14, 2020, from https://www.tutorialspoint.com/unity/unity_quick_guide.htm

*Unity - Scripting API: CustomPropertyDrawer*. Retrieved June 14, 2020, from https://docs.unity3d.com/ScriptReference/CustomPropertyDrawer.html

*Unity - Scripting API: Editor*. Retrieved June 14, 2020, from https://docs.unity3d.com/ScriptReference/Editor.html