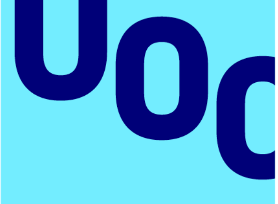

Desarrollo de una herramienta para implementar música adaptativa en el motor gráfico Unity3D.

Alumno: Rafael Carrión García.
Plan de estudios: Máster universitario de diseño y programación de videojuegos.
Área: M7.462 - Trabajo Final de Máster.
Profesor: Jordi Duch Gavaldà
Fecha: 04/01/2021

B) GNU Free Documentation License (GNU FDL)

Copyright © ANY EL-TEU-NOM.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is included in the section entitled "GNU Free Documentation License".

C) Copyright

© (el autor/a)
Reservados todos los derechos. Está prohibido la reproducción total o parcial de esta obra por cualquier medio o procedimiento, comprendidos la impresión, la reprografía, el microfilm, el tratamiento informático o cualquier otro sistema, así como la distribución de ejemplares mediante alquiler y préstamo, sin la autorización escrita del autor o de los límites que autorice la Ley de Propiedad Intelectual.

Ficha del trabajo de final

Título del trabajo	*Desarrollo de una herramienta para implementar música adaptativa en el motor gráfico Unity3D.*
Nombre del autor:	*Rafael Carrión García*
Nombre del consultor/a:	*Jordi Duch Gavaldà*
Nombre del PRA:	*Jordi Duch Gavaldà*
Fecha de entrega (mm/aaaa):	*03/01/2021*
Titulación o programa:	*Máster universitario de diseño y programación de videojuegos.*
Área del Trabajo Final:	*TFM*
Idioma del trabajo:	*Castellano*
Palabras clave	*Unity3D, Música adaptativa,*

Resumen del trabajo (máximo 250 palabras):

Diseño y desarrollo de una herramienta para el motor gráfico Unity3D que permita implementar música lineal y adaptativa, facilitando la gestión de las piezas musicales y ofreciendo diferentes modos de reproducción.

En la actualidad los videojuegos superan la recaudación de otras industrias como la del cine o la industria musical, lo que ha derivado en que cada vez se realicen mayores inversiones en todos los departamentos involucrados en el desarrollo del videojuego, sin excluir al apartado musical. La composición musical para videojuegos es un camino profesional que está ganando popularidad para músicos y compositores, desde la grabación de grandes producciones orquestales, hasta piezas compuestas mediante instrumentos virtuales y librerías en un pequeño ordenador portátil.

Una de las principales diferencias con el resto de medios lineales (cine, televisión, conciertos, etc.) en lo que al apartado sonoro se refiere, reside en la flexibilidad y la adaptación de las piezas musicales y efectos de sonido, de tal manera que todo el apartado sonoro puede reaccionar según las acciones del jugador y el estado del juego.

Abstract (in English, 250 words or less):

Design and development of a tool for Unity3D game engine that allows to implement linear and adaptive music, facilitating the management of musical pieces and offering different playback modes.

Currently, economic returns generated by the video game industry exceed the collection of other industries such as the cinema or the music industry, which has resulted in more economic investment in every department involved in the development of the video game, without excluding the audio department. Music composition for video games is a career path that is gaining popularity for musicians and composers, from recording large orchestral productions, to pieces composed using virtual instruments and libraries on a small laptop.

One of the main differences with the rest of linear media (cinema, television, concerts, etc.) lies in the flexibility and adaptation of the musical pieces and sound effects, where all sounds and music can react according to the actions of the player and the state of the game.

Índice

1. Introducción.
   1.1 Contexto y justificación del trabajo.
   1.2 Objetivos del trabajo.
   1.3 Enfoque y método seguido.
2. Música adaptativa: flujo de trabajo.
   2.1 Introducción al capítulo.
      2.2.1 Muestra en un secuenciador musical (“Demo”).
      2.2.2 Grabación.
      2.2.3 Implementación en el motor gráfico.
3. Análisis de los principales middlewares.
   3.1 Introducción al capítulo.
   3.2 Funcionalidades comunes.
4. Diseño de la herramienta: datos e interfaz.
   4.1 Introducción al capítulo.
   4.2 Estructura de almacenamiento de clips de audio.
      4.2.1 Descripción de “so_TrackContainer.cs”.
      4.2.1 Descripción de “so_ContainerListData.cs”.
   4.3 Interfaz de gestión.
      4.3.1 Interfaz: ContainerManager.
      4.3.2 Interfaz: TrackContainer.
5. Diseño de la herramienta: sincronización de pistas.
   5.1 Introducción al capítulo.
   5.2 Primera aproximación.
      5.2.1 Metrónomo.
      5.2.2 Reproducción de pistas.
   5.3 Problemas y soluciones del primer método.
      5.3.1 Corrección de tipo de variables.
      5.3.2 Solución a la latencia por carga de audio.
6. Diseño de la herramienta: solución final.
   6.1 Programación de las pulsaciones del metrónomo.
   6.2 Selección y reproducción de contenedores
   6.3 Escenas de control
7. Conclusión
   7.1 Aprendizaje durante el desarrollo, análisis de los objetivos, seguimiento y metodología.
   7.4 Líneas de trabajo futuro.

8. Bibliografía

1. Introducción.

1.1 Contexto y justificación del trabajo.

En la actualidad la industria de los videojuegos desarrolla productos de entretenimiento audiovisual que aunan el arte y la tecnología. Una de las consecuencias del aumento del acceso y de la facilidad de creación de videojuegos, ha generado un ambiente laboral que cada vez resulta más competitivo, obligando a los trabajadores de la industria a reinventarse constantemente y desarrollar nuevas técnicas para optimizar la tecnología y ponerla al servicio del arte,la jugabilidad y el entretenimiento del producto digital.

Desde el desarrollo y publicación de los primeros videojuegos, el diseño y la ambientación sonora ha formado una parte fundamental del apartado artístico de un videojuego, convirtiéndose en muchas ocasiones en un elemento diferenciador entre otros títulos y extendiendo la cultura del videojuego a un mercado sumamente competitivo como es el de la música. Para ejemplificar la comunión entre música y videojuegos, podemos citar la figura de *Koji Kondo*, un compositor que comenzó a trabajar para la conocida marca de consolas *Nintendo* cuando las limitaciones tecnológicas formaban parte del reto de componer una banda sonora memorable y emotiva. Durante los primeros años, *Koji Kondo* sólo podía componer para cuatro instrumentos, lo que le bastó para componer piezas que hoy en día siguen sonando en auditorios o en televisión como las canciones de la serie de *"Mario Bross"* o de *"The Legend of Zelda"*, con el beneficio correspondiente obtenido de su explotación económica y mediática.

Con el paso de los años el diseño de sonido y la música para videojuegos se ha ido consolidando como una disciplina compleja, por diferentes motivos. Por un lado, compite con los elevados estándares de calidad marcados por otros medios de comunicación de masas como son el cine y la televisión, mientras que por otro lado, la extensión y la poca linealidad de las experiencias interactivas relacionadas con el videojuego hacen que el diseño de sonido adaptativo con una perspectiva cinematográfica se convierta en una tarea difícil de abordar para los equipos de desarrollo. Indudablemente la calidad sonora de los videojuegos ha crecido drásticamente debido a los avances tecnológicos, al interés de profesionales consagrados del sector (compositores y diseñadores de sonido de cine) y a la mayor inversión económica que pueden permitirse muchos estudios de videojuegos.

En este contexto se han abierto un hueco unas pocas empresas que ofrecen soluciones a la implementación de audio en los distintos motores gráficos, creando un tipo de software comúnmente llamado "*Middleware*" que sirve de bisagra entre la parte más técnica de programación y una interfaz y funcionalidades que son más

familiares a los músicos . En el trabajo se analizará brevemente algunas de las funciones comunes de estos programas, con el fin de situar la herramienta en un contexto profesional actual y coherente.

1.2 Objetivos del trabajo.

En este trabajo se pretende estudiar y desarrollar una herramienta dentro del motor gráfico Unity3D que permita la gestión y la implementación de la música de un videojuego, ofreciendo soluciones sencillas para la creación de bandas sonoras que se adapten al entorno de juego sin tener que hacer uso de herramientas externas o “middlewares”.

Así mismo, este trabajo pretende ser un punto de partida para explorar el motor de audio de Unity3D y posteriormente extender las funcionalidades de la herramienta, adaptándola a las necesidades de diseño del juego y adelantándose a futuras complicaciones tras haber obtenido una visión más concreta del funcionamiento del audio en el motor.

1.3 Enfoque y método seguido.

Para enfocar el trabajo primero se analizará brevemente las soluciones que ofrecen las empresas que cuentan con mayor número de títulos publicados del sector, encontrando los puntos comunes entre ellas. De esta manera se pretende cribar y seleccionar las características fundamentales y más demandadas de los productos que ofrecen.

Una vez decididas las funcionalidades de la herramienta, se realizarán iteraciones para encontrar una solución cómoda, rápida y flexible que tenga como principales premisas:

- Interfaz propia y funcional: la herramienta debe contar con una interfaz exclusiva para su propósito, que agilice la creación y gestión de las piezas musicales.

- Independencia de herramientas externas: todo el contenido de la herramienta debe ser programada en Unity3D sin utilizar ninguna API externa.

- Debe ser extensible y útil en múltiples proyectos.

2. Música adaptativa: flujo de trabajo.

2.1 Introducción al capítulo.

En este primer capítulo se expondrán los procesos que se siguen a la hora de implementar el sonido y la música de un videojuego. Si bien no todos los estudios siguen un flujo de trabajo general, todos ellos disponen a grandes rasgos de las mismas herramientas y procesos, desde los estudios más grandes, hasta estudios más pequeños. La gran diferencia entre las grandes compañías y las pequeñas, es la cantidad de recursos humanos y tecnológicos que pueden invertir en cada proceso de la cadena, algo que acaba siendo un factor tremendamente diferencial.

2.2 Flujo de trabajo general.

Durante la creación de la banda sonora se incluyen diferentes procesos comunes como los que se pueden ver en la *Figura 2.1.*

Figura 2.1 : Flujo de trabajo de la banda sonora de un videojuego.

2.2.1 Muestra en un secuenciador musical ("Demo").

En la actualidad, tanto en la industria del cine como de los videojuegos uno de los primeros pasos es realizar una muestra del trabajo para que los directores del proyecto lo acepten. Esta muestra se realiza en un *DAW (Digital Audio Workstation)* o secuenciador musical. En la actualidad existen numerosos secuenciadores que

comparten las mismas características, aún así existen algunas diferencias notables que los posicionan en el mercado. Algunos de los ejemplos más importantes son:

- Ableton Live: un secuenciador con una curva de aprendizaje muy asequible para compositores y artistas noveles. Ofrece una gran velocidad de trabajo para diseñar efectos de sonido y la producción de música electrónica. Su principal ventaja en el mercado reside en que es un programa diseñado para actuaciones en directo, algo en lo que sus competidores se quedan atrás. En contrapartida, la forma de trabajo que por un lado le dota de velocidad para crear nuevos sonidos, por otro lado limita o en ocasiones puede resultar incómoda a la hora de manejar la ruta de la señal.

- FL Studio: Es un secuenciador que en los últimos años ha crecido rápidamente por la influencia de ciertos estilos urbanos pero que no ha significado un impacto considerable en la industria de los videojuegos.

- Cubase: Es uno de los estándares de la industria audiovisual. Sus múltiples opciones de personalización y flexibilidad para trabajar con librerías orquestales lo ha posicionado para que grandes compositores como *Hans Zimmer* lo haya adoptado como herramienta principal de trabajo. A parte de tener un tratamiento de la señal mejor que otros secuenciadores, dispone de un editor de partituras, mejor gestión de las librerías midi y lo que es más importante, pertenece a la misma casa que desarrolla Nuendo, un producto que está centrándose en el audio para videojuegos, permitiendo una integración total del secuenciador con el middleware.

- Reaper: A medida que pasa el tiempo este secuenciador se acerca más a la industria del videojuego, principalmente porque permite a los usuarios programar sus propias funciones e integrarlas en el programa, de tal forma que resulta interesante para automatizar muchas de las tareas del proceso del audio de videojuegos.

- Pro Tools: Se trata del referente en la industria audiovisual desde hace años para realizar las mezclas de audio y música, tanto para el cine como para otros medios.

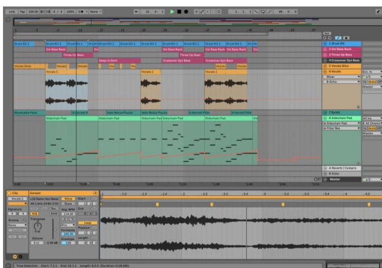

Figura 2.2 : Ableton Live

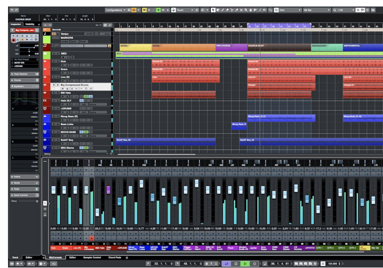

Figura 2.3 : Cubase

Figura 2.4 : Pro Tools

2.2.2 Grabación.

Una vez están compuestas las demos se planifica la grabación de las pistas e instrumentos necesarios. Es una etapa muy importante del proceso que está estrictamente condicionada por la inversión económica del estudio y en la que tiene que estar muy planificadas las sesiones, la distribución de las pistas y cómo se van a implementar en el juego, pues en ocasiones puede resultar muy caro repetir la grabación. En esta fase de producción destaca el coste que supone grabar piezas orquestales sobre otros estilos de música, principalmente porque es un género en el que se suele tener que contar con más profesionales y equipo para el trabajo.

Para ilustrar el proceso se pondrá como ejemplo tres situaciones que sirvan de referencia, no obstante, cada videojuego cuenta con sus peculiaridades concretas que no podemos referenciar:

- Videojuego Indie o poco presupuesto: En la grabación se ven involucrados pocos músicos y se pretende abaratar costes, lo que motiva al uso de recursos fácilmente accesibles. Estos pueden ser librerías comerciales, sintetizadores o instrumentos virtuales.

- Presupuesto medio: Se realizan grabaciones de ciertos instrumentos protagonistas que llevan el peso de las piezas, para posteriormente integrarlos con librerías de sonidos e instrumentos virtuales. Un buen uso de esta técnica consigue aumentar la expresividad que puede faltar en algunas composiciones puramente sintéticas. Si el presupuesto destinado es un poco mayor algunas compañías optan por la posibilidad de alquilar estudios previamente microfoneados por horas, con una orquesta que hace varias tomas leyendo la partitura a primera vista.

- Presupuesto alto: El estudio tiene la capacidad de invertir en una grabación completa con la cantidad de músicos que necesite, microfonía particular, equipo de mayor calidad, contratación de profesionales para mezclar las canciones, etc. Aunque en las grandes producciones también se mantienen las librerías comerciales de las demos e instrumentos virtuales, se utilizan como apoyo de la pieza, no como eje fundamental sobre el que se tiene que construir los demás elementos, como es el caso de producciones anteriormente nombradas.

2.2.3 Implementación en el motor gráfico.

Tras realizar la grabación y la post-producción de la música y efectos de sonido, comienza el proceso para integrarlo en el motor de juego. Al igual que para cualquier apartado artístico, la implementación de audio requiere la colaboración de dos departamentos que tienen áreas de conocimiento distintos, el departamento técnico que se encarga de la programación y el apartado artístico que involucra toda la creación sonora previamente mencionada. En esta confluencia de perfiles profesionales las empresas suelen optar por alguna de estas soluciones:

- Usar el motor de audio del motor gráfico: Normalmente utilizar herramientas de terceros encarece el desarrollo del videojuego, de modo que algunos estudios de bajo presupuesto optan por no utilizar herramientas externas. Esta opción abarata costes pero plantea nuevas limitaciones debido a la inversión de horas de trabajo necesaria para abordar cualquier funcionalidad fuera de las posibilidades que ofrece el motor.

- Utilizar herramientas externas o middleware: Existen compañías que desarrollan herramientas que sirven de puente entre los secuenciadores musicales y los motores gráficos. A parte de encarecer la producción del juego también es necesario contar con profesionales que las conozcan, por otra parte ofrecen numerosas posibilidades complejas de desarrollar.

3. Análisis de los principales middlewares.

3.1 Introducción al capítulo.

En la actualidad hay dos compañías que dominan el mercado, la primera de ellas se llama Audiokinetic y es el propietario de Wwise, mientras que la segunda se llama Fmod y desarrolla un software del mismo nombre. Ambas empresas cuentan con un largo catálogo de videojuegos que han utilizado su software y servicios, los cuales cuentan con muchas características similares. Por ejemplo, ambas cuentan con varias licencias de uso comercial para adaptarse a distintos presupuestos, pero las licencias orientadas al desarrollo de videojuegos "indie" tienen limitaciones distintas, lo que hace que muchos juegos de este tipo se inclinen más por escoger Fmod.

En definitiva, al contratar este tipo de servicios los desarrolladores tienen que analizar la opción que más se adapte a su videojuego, lo que a veces puede resultar complicado y añade costes derivados de la contratación y mantenimiento del servicio.

3.2 Funcionalidades comunes.

Tanto Fmod como Wwise ofrecen soluciones para la implementación de efectos de sonido como de música lineal y adaptativa. Para el desarrollo de la herramienta estudiaremos las funcionalidades orientadas a la gestión de la música adaptativa, pues la música lineal puede ser fácilmente implementada con las herramientas que ofrece Unity3D al no tener que responder a ningún tipo de sincronización musical. Algunas de sus principales funcionalidades son:

- Sincronización: Al cargar las pistas musicales en ambos programas, se ofrece la posibilidad de establecer el tempo de la canción para que los *stingers*, cambios de pista o *loops* puedan realizarse sincrónicamente.

- Modos de reproducción: Existe cierta flexibilidad a la hora de reproducir las pistas cargadas, de modo que se pueden crear distintas combinaciones musicales. Esto contribuye a disminuir la fatiga auditiva del jugador al evitar la repetición constante de las piezas musicales, algo que resulta muy negativo para la experiencia de juego.

- Eventos globales: El cambio de canciones o lanzamiento de clips se puede realizar de una manera global, una característica que facilita las tareas de programación.

- Interfaz intuitiva: Uno de los puntos fuertes de estas herramientas es que ofrece una interfaz que se acerca a la estética y organización de un secuenciador musical. Aunque parece que sea un detalle puramente estético, sin duda sitúa a los artistas que no tienen nociones de un motor gráfico concreto en un punto intermedio en el que se pueden comunicar directamente con los programadores, agilizando parte del proceso.

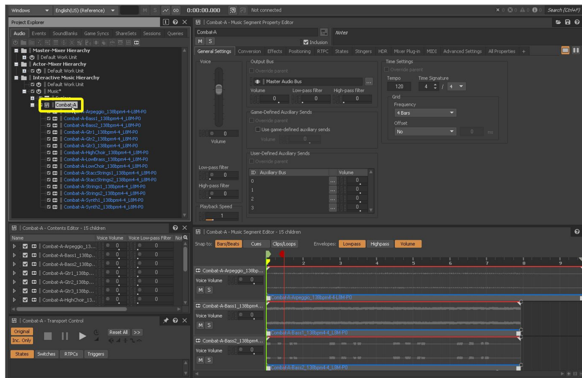

Figura 3.1 : Wwise

Figura 3.2 : Fmod

4. Diseño de la herramienta: datos e interfaz.

4.1 Introducción al capítulo.

En este capítulo primero se detalla la planificación de la herramienta, desde la gestión de los datos y clips musicales, repasando los aspectos a tener en cuenta para la sincronización, hasta las correcciones y la justificación de la toma de decisiones. Pese a las numerosas modificaciones y correcciones, la estructura fundamental de la herramienta se ha mantenido constante.

4.2 Estructura de almacenamiento de clips de audio.

La estructura para guardar los datos se basa en el uso de los *Scriptable Objects,* una forma de almacenar datos que según la documentación oficial de Unity:

<<A ScriptableObject is a data container that you can use to save large amounts of data, independent of class instances. One of the main use cases for ScriptableObjects is to reduce your Project's memory usage by avoiding copies of values. >>

"Unity Documentation"

Entre las ventajas más útiles de los *Scriptable Objects* para el desarrollo de la herramienta se destacan:

- Pueden mantenerse tras las recargas de escena y son fácilmente accesibles en cualquier momento de la ejecución.

- Puede evitar la duplicidad de datos ya que se guardan por referencia y no como las clases y estructuras normales que se serializan como copias completas.

- Se pueden guardar como un asset del proyecto.

- No necesitan estar adjuntos a *GameObjects*.

- Es una forma cómoda para manejar datos y crear instancias desde una interfaz propia.

Tras repasar las ventajas de los *Scriptable Objects* se decide crear dos tipos derivados de esta clase que se describen en los puntos siguientes.

4.2.1 Descripción de “so_TrackContainer.cs”.

En este contenedor se guardan los clips de audio que forman un segmento musical. Depende de la pieza musical estos clips de audio se dividen en los instrumentos o secciones que el compositor o programadores consideren necesarios. En la *Figura 4.1* se puede observar un ejemplo básico compuesto por una percusión, un bajo y un piano en un secuenciador.

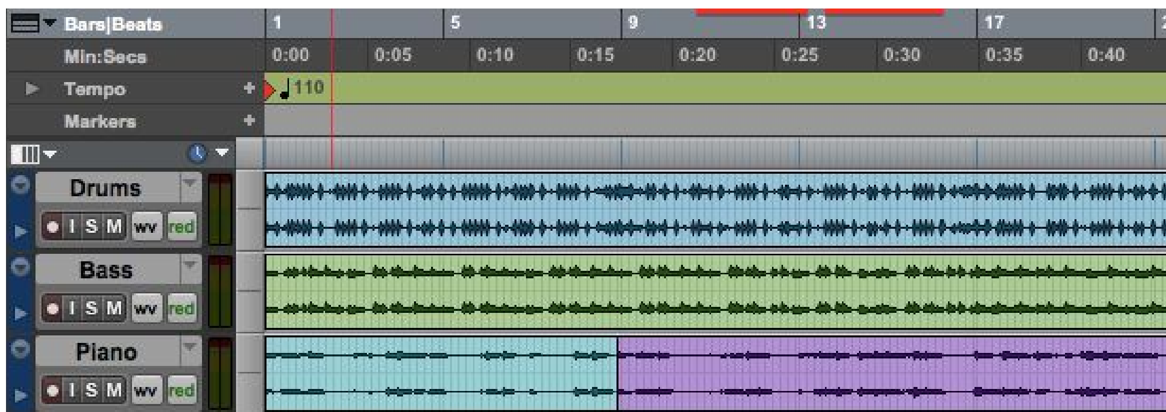

Figura 4.1 : Ejemplo de segmento musical en Pro Tools

Para replicar, en cierto modo, el comportamiento de un secuenciador musical en Unity, nombraremos las variables necesarias para poder armar un segmento musical:

- Tempo de la canción: Indica la velocidad a la que se ejecuta una pieza musical y se mide en pulsaciones por minuto.

- Listas de *AudioClips:* Debido a la naturaleza de la música de videojuegos, la cual cuenta con muchas piezas en bucle se decide utilizar dos listas. Una de ellas son para piezas que siempre se reproducirán en bucle y la otra para clips de audio que variarán según el método de reproducción elegido.

- Stingers: Son *AudioClips* que se utilizan como recurso en la música de videojuegos como señales acústicas que normalmente se sitúan al inicio y al final de un segmento musical para evitar que las piezas musicales dejen de sonar bruscamente. Su particularidad reside en que el momento de inicio de la reproducción del clip viene determinada por la elección del programador, quien determina el momento de la reproducción del clip que debe coincidir con el tiempo fuerte del compás. Como se puede ver en la *Figura 4.2* Wwise cuenta con una interfaz muy intuitiva para programar los *Stingers*, en la que el marcador azul indica el momento del clip de audio que coincidirá sincrónicamente con el tempo de

la canción, calculando el desfase necesario y reproduciendo el clip con anterioridad para que coincidan en la línea temporal.

- Tiempo de los *Stingers:* Al no disponer de una interfaz como la mostrada en la *Figura* 4.2, el desfase será un valor numérico calculado anteriormente en el secuenciador musical.

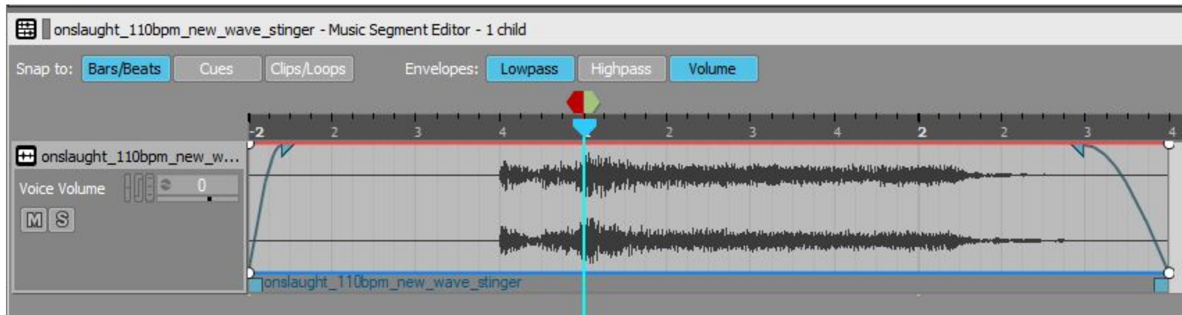

Figura 4.2 : Ejemplo de programación de un Stinger en Wwise

- Campo *AudioMixerGroup:* Esta variable almacena la dirección de salida de la señal de audio. Sólo aparecerán como seleccionables los grupos que hayan sido creados de forma convencional en Unity.

- Modo de reproducción: Es una cadena que sirve para discriminar posteriormente el modo de reproducción.

4.2.1 Descripción de "so_ContainerListData.cs".

La función de este *ScriptableObject* es gestionar los *"so_TrackContainers"*, de tal forma que los métodos para crearlos, borrarlos y leerlos sean más fáciles de acceder desde el código de la interfaz. Aquí se puede ver una de las partes más importantes del código, pues la herramienta genera los *"so_TrackContainers"* en una ruta específica y es la cual tiene que ser leída por la interfaz.

```
public void GetContainers(){
so_ContainerListData containerData
=Resources.Load<so_ContainerListData>("TrackContainerList/TrackContainerList");
            so_TrackContainer[] tempList
=Resources.LoadAll<so_TrackContainer>("TrackContainers");
            List<so_TrackContainer> currentList = containerData.GetListOfContainers();
            foreach(so_TrackContainer containerItem in tempList){
                if(!currentList.Contains(containerItem)) {
                containerData.AddNewContainerToList(containerItem);
                }
            }
```

4.3 Interfaz de gestión.

Uno de los objetivos del trabajo era crear una interfaz con las herramientas que ofrece Unity para facilitar la gestión del contenido musical. Dicha interfaz recibe los datos de los *ScriptableObjects* previamente explicados, pero en este caso se comentará primero la parte de la interfaz que corresponde a los "*so_ContainerListData*" y posteriormente la parte de los "so_TrackContainer", para seguir el orden por el que se rige la interfaz, desde la lectura de la lista de los contenedores hasta los contenedores particularmente.

La interfaz se abre desde una pestaña llamada "Adaptative Music" como se muestra en la *Figura 4.3*

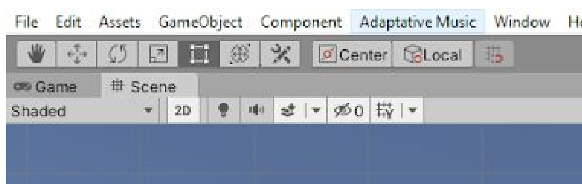

Figura 4.3 : Muestra de la pestaña "Adaptative Music"

Figura 4.4 : Muestra de la pestaña "ContainerManager"

Aquí se muestra un esquema de la navegación por la interfaz y el flujo de datos.

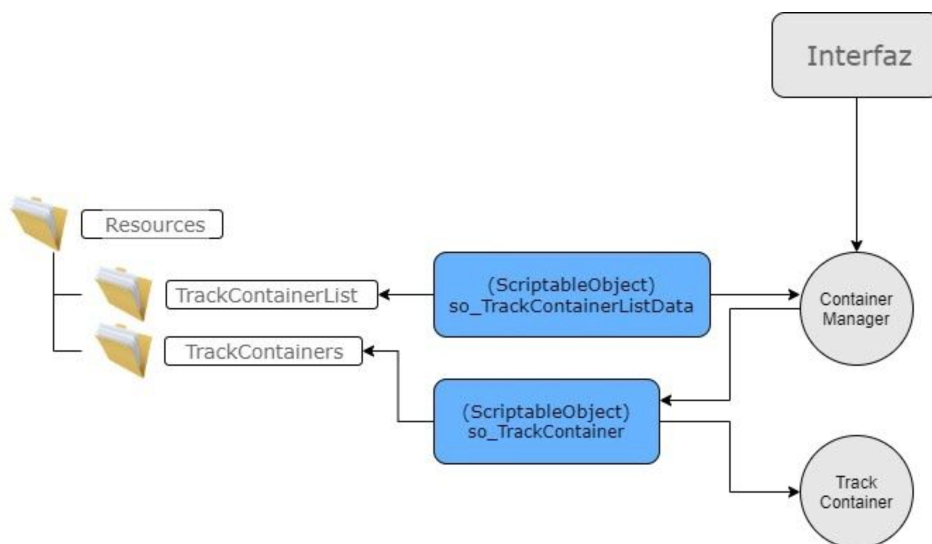

Figura 4.5 : Esquema del funcionamiento de la interfaz.

4.3.1 Interfaz: ContainerManager.

En esta parte de la interfaz se muestra la lectura obtenida de la carpeta donde se encuentran los contenedores. La información se muestra en listas y permite:

- Crear contenedores nuevos: Se introduce el nombre del nuevo contenedor y seleccionando "Create Container" se crea en la ruta especificada.

- Borrar contenedores: Pulsando el botón "Delete" junto a cualquier contenedor se borrará.

- Abrir la interfaz de cada contenedor: Presionando el botón con el nombre de cada contenedor se genera una nueva ventana con las opciones particulares del contenedor.

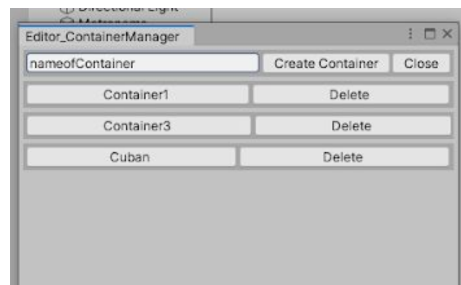

Figura 4.6 : Visualización del ContainerManager

Para poder generar la interfaz es necesario que la clase "*Editor_TrackContainerManager*" herede de "*EditorWindow*", tambíen hay que recurrir a métodos para redibujar la interfaz o guardar los datos cada vez que cambia. Alguno de los métodos de la interfaz y sus particularidades son:

Este método se encarga de renderizar la interfaz y manejar sus eventos, en ella se recoge el nombre del nuevo contenedor en la variable *newContainerName* mediante un campo de texto y al presionar el botón "Create Container" se llama a la función correspondiente para crearlo en la ruta. Finalmente siempre se ejecutan los métodos para renderizar los contenedores disponibles.

```
public void OnGUI() {
     EditorGUILayout.BeginHorizontal("box");
     newContainerName = EditorGUILayout.TextField("nameofContainer");
     if (GUI.changed) newContainerName_Fixed = newContainerName;
         if (GUILayout.Button("Create Container")){
         CreateTrackContainer(newContainerName_Fixed);
         }
     if (GUILayout.Button("Close"))
     {
         Close();
```

```
        }
        EditorGUILayout.EndHorizontal();

PaintAllContainers(Resources.Load<so_ContainerListData>("TrackContainerList/TrackContaine
rList"));
    }
```

Renderización de contenedores disponibles: Se vacía la lista que contiene los contenedores y se refresca para que cada cambio se refleje en el momento. Al recorrerse la lista *so_TrackContainer* se crean tantos botones como contenedores existen, los cuales al presionarlos crean una instancia de la clase *Editor_TrackContainer* que recibe el nombre del contenedor seleccionado y posteriormente la recibe con el método *GetWindow()*.

```
private void PaintAllContainers(so_ContainerListData containerListData){
    containerListData.ClearList();
    containerListData.GetContainers();

    foreach(so_TrackContainer container in
containerListData.GetListOfContainers()){
        GUILayout.BeginHorizontal("box");
        string containerName = container.name;
        if(GUILayout.Button(containerName)){
            sc_ContainerManager.SetCurrentTrackContainer(container);
            Editor_TrackContainer newEditorTrackContainer = new
Editor_TrackContainer() ;
                GUIContent newTitle = new GUIContent();
            newTitle.text = container.name;
            newEditorTrackContainer.titleContent= newTitle;
            newEditorTrackContainer.SetCurrentContainer(container);
            GetWindow(typeof(Editor_TrackContainer), newEditorTrackContainer);
            }
        if(GUILayout.Button("Delete")){
            DeleteContainer(container.name);
            }
        GUILayout.EndHorizontal();
        }
    }
```

La creación y destrucción de los contenedores se realiza mediante los métodos *CreateTrackContainer()* y *DeleteContainer()* respectivamente. Como se puede observar la ruta de los datos es fija, esto se ha decidido así porque en otra asignatura del Máster llamada "Modding y creación de niveles" se comentó con la profesora que las rutas y direcciones de las herramientas no deberían ser modificables por los diseñadores por seguridad. En este caso es la mejor opción, pues los archivos de audio pueden alojarse en cualquier carpeta, de tal forma que no

compromete a la organización del proyecto, mientras que todos los contenedores se encuentran en una misma localización, lo que facilita su reutilización.

```
public void CreateTrackContainer(string nameContainer){
        string path = "Assets/Resources/TrackContainers/" + nameContainer
+  ".asset";
        so_TrackContainer newContainer;
        newContainer =new so_TrackContainer();
        AssetDatabase.CreateAsset(newContainer, path);
        AssetDatabase.SaveAssets();
        AssetDatabase.Refresh();
    }

    private void DeleteContainer(string nameContainer){
        Debug.Log("Name in function::" + nameContainer);
        string path = "Assets/Resources/TrackContainers/" + nameContainer
+  ".asset";
        AssetDatabase.DeleteAsset(path);
    }
```

4.3.2 Interfaz: TrackContainer.

Esta segunda parte de la interfaz se muestran las variables pertenecientes al *ScriptableObject* llamado *"so_TrackContainer".* Se enumerarán de arriba a abajo y de izquierda a derecha cada una de las secciones y elementos modificables:

- Pestaña superior izquierda: Se muestra el nombre del contenedor, es la referencia que se tiene que utilizar para buscarlo y seleccionarlo posteriormente.

- Play mode: nos permite seleccionar el modo de reproducción de los clips variables entre Random, que reproduce clips aleatorios de la lista o Sequence, que los reproduce consecutivamente.

- Sección clips "Play Variable": Los *AudioClips* de esta sección se ven afectados por el modo de reproducción. Se añaden clips nuevos con "Add new clip to play variable section" y se borran con la opción "Remove Clip".

- Sección de clips "Play Always": Se trata de otra sección de *AudioClips* con las mismas opciones que la sección previamente descrita, con la particularidad de que todos los clips se reproducen a la vez en loop.

- Sección de *Stingers*: Añadiendo los clips a sus campos correspondientes llamados "StingerIn" y "StingerOut", se asignan respectivamente los *Stingers* de inicio y pausa de reproducción.

- Time Stinger Beat: Este tiempo es el desfase del clip de audio, desde su inicio hasta donde cae el tiempo del compás.

Figura 4.7 : Visualización del ContainerManager

En el caso de la clase "*Editor_TrackContainer*" las principales dificultades se encuentran en crear campos que posibiliten la elección de ciertos tipos de objetos y su guardado en los *ScriptableObjects* correspondientes. Algunos ejemplos de ello podrían ser:

El método de inicio que se encarga de llamar al resto de funciones.

```
private void OnGUI() {
        if (GUILayout.Button("Close")) {
            Close();
        }
        PaintContainerSettings();
        PaintClipsOfContainer();
        PaintClipsOfPlayAlways();
        PaintStingers();
    }
```

En la sección de opciones del contenedor se encuentra:

- Un campo seleccionable (pm) que almacena el método de reproducción según un enumerado y lo manda al contenedor en forma de cadena.
- Un campo de tipo objeto que permita buscar los *MixerGroups* disponibles para asignarlos.
- Un campo de tipo float para asignar las pulsaciones por minuto del contenedor.

```
public void PaintContainerSettings() {
        pm = (playMode)EditorGUILayout.EnumPopup("Play mode:", pm);
        containerTrack.SetPlayMode(pm.ToString());
        EditorGUILayout.BeginHorizontal("box");
        AudioMixerGroup newMixer;
        newMixer = EditorGUILayout.ObjectField("AudioMixer",
containerTrack.mixerGroup, typeof(AudioMixerGroup), false) as AudioMixerGroup;
        containerTrack.bpm = EditorGUILayout.FloatField("BPM",
containerTrack.bpm);
        if (GUI.changed) containerTrack.mixerGroup = newMixer;
        EditorGUILayout.EndHorizontal();
        EditorGUILayout.LabelField("------Audioclips:  play variable section
-----");
    }
```

Los métodos para renderizar las listas de pistas disponibles son muy similares para la lista de clips variables y la lista que se reproduce en bucle. En ella se comienza creando una lista temporal de *AudioClips* que posteriormente se rellena con los elementos de la lista guardados en el contenedor. Una vez recorrida la lista se crean dos botones, uno para añadir un elemento vacío a la lista y otro para borrar una entrada de la lista. Para borrar un elemento de la lista hay que introducir el índice que se muestra al lado del clip en la interfaz, esto es así porque en un primer momento se utilizaba la variable *counter* pero sólo borraba el último elemento de la lista, pues una vez recorrido el bucle su valor se corresponde al índice del último elemento.

```
public void PaintClipsOfContainer() {

        int counter = 0;
        List<AudioClip> tempArray = containerTrack.audioListPlayVariable;
        if (tempArray.Count != 0)         {
            foreach (AudioClip clip in containerTrack.audioListPlayVariable)
            {
                EditorGUILayout.BeginHorizontal("box");
                AudioClip tempClip;
                tempClip = EditorGUILayout.ObjectField("Track " +
counter.ToString(), containerTrack.audioListPlayVariable[counter],
typeof(AudioClip), false) as AudioClip;
                EditorGUILayout.EndHorizontal();
                if (GUI.changed)
                {
                    tempArray[counter] = tempClip;
                }
                counter++;
            }
        }
        EditorGUILayout.BeginHorizontal();
```

```
        int index = new int();
        index = EditorGUILayout.IntField("Index", index);
        if (GUI.changed) removeIndex = index;
        if (GUILayout.Button("RemoveClip"))
        {
            containerTrack.audioListPlayVariable.RemoveAt(removeIndex);
        }
        EditorGUILayout.EndHorizontal();

        if (GUILayout.Button("Add new clip to play variable section")) {
            containerTrack.audioListPlayVariable.Add(null);
        }
    }
```

El apartado que corresponde a la asignación de *Stingers* no se comentará porque cuenta con métodos similares y sería reiterativo.

5. Diseño de la herramienta: sincronización de pistas.

5.1 Introducción al capítulo.

A la hora de ensamblar todas las pistas de los contenedores y sus modos de reproducción en una línea de tiempo común, se necesita un sistema que ejerza de metrónomo.

<<El metrónomo produce una marca métrica, regular (latidos, clicks), que pueden ser ajustados en latidos por minuto. >>

www.conceptodefinicion.de

Seguidamente se describen las iteraciones que fueron necesarias, argumentando por qué no eran opciones válidas y se tuvo que cambiar el enfoque.

5.2 Primera aproximación.

En la primera solución se comenzaron utilizando los métodos más comunes que ofrece la API de Unity como son *AudioSource.PlayOneShot()* o *AudioSource.Play()*, métodos sencillos y muy prácticos para la mayoría de aplicaciones pero una vez implementadas prácticamente todas las funcionalidades de la herramienta comenzaron a surgir distintos problemas. Algunos estaban directamente relacionados con el uso de estos métodos.

5.2.1 Metrónomo.

Para transformar el tempo de la canción en segundos y poder medirlo durante la ejecución, se programó un script sencillo en el que *timeBetweenBeats* representa el tiempo entre pulsaciones y mediante la función Time.deltaTime se incrementa la cuenta global de tiempo. Al cumplirse la condición que determina que se ha cumplido el tiempo entre pulsaciones, se lanza un evento a la que otras funciones pueden suscribirse.

```
private void GetTimeBtweenBeats(){
       timeBetweenBeats = bpm/60;
}
private void Update() {
              if(startCount){
                  countBetweenBeats += Time.deltaTime;
                  if(countBetweenBeats >= timeBetweenBeats){
```

```
                countBetweenBeats -= timeBetweenBeats;
                barCount++;
                CustomEvents.customEvent.ChangeBar();
            }
        }
    }
```

5.2.2 Reproducción de pistas.

La reproducción de las pistas tal y cómo se comenta en puntos anteriores viene determinada por la pulsación del metrónomo. El lanzamiento de un evento a cada cambio de compás facilita que el script que gestiona la reproducción de las pistas y ejecutar una función según la pulsación del metrónomo, habilitando la reproducción de los clips mediante la función *AudioSource.PlayOneShot()* si éstos habían consumido su tiempo de reproducción. La principal razón por la que utilizar *PlayOneShot()* es porque se pueden reproducir varios clips de audio desde un mismo *AudioSource*, de modo que resulta una solución muy sencilla en la que apenas hay unas pocas clases.

5.3 Problemas y soluciones del primer método.

Una vez se comenzó a probar el sistema se empezaron a observar problemas de sincronización, algunas veces a medida que se aumentaba el número de clips y otras veces de forma aleatoria. Tras buscar información sobre el motor de audio de Unity, se detectó que los problemas podrían deberse al tiempo de descompresión de los archivos y el uso incorrecto del tipo de variables.

5.3.1 Corrección de tipo de variables.

En un primer momento se utilizaban variables de tipo float que tienen una precisión menor que los de tipo double. Algunos de los cambios que se llevaron a cabo fueron:

- Calcular las diferencias temporales y pulsaciones del metrónomo en variables de tipo float.

- Establecer como reloj de consulta el valor extraído de *AudioSettings.dspTime*, que según la definición del manual de Unity es mucho más preciso que otras fuentes:

<<This is a value specified in seconds and based on the actual number of samples the audio system processes and is therefore much more precise than the time obtained via the Time.time property.>>

"Unity Documentation"

- Calcular la longitud de los clips usando la frecuencia de muestreo, que determina la cantidad de muestras que se toman de una señal de audio por unidad de tiempo. La duración del clip es el resultado de dividir la longitud del clip en muestras entre la frecuencia de muestreo.

5.3.2 Solución a la latencia por carga de audio.

Para evitar las imprecisiones de sincronización por la carga del audio se utilizará el método AudioSource.PlayScheduled, que permite reproducir un clip en un momento específico de la línea de tiempo obtenido del dsp de audio. Esta forma de reproducción es independiente de la velocidad de los fotogramas y le da al sistema de audio el tiempo suficiente para preparar la reproducción de sonido sin causar picos repentinos de CPU.

Uno de los problemas derivados de usar este método es la incapacidad del sistema para planificar la reproducción de un clip en un *AudioSource* ocupado por la reproducción de otro clip, algo que causa cortes y vacíos de sonido.

6. Diseño de la herramienta: solución final.

6.1 Programación de las pulsaciones del metrónomo.

Durante el tiempo de ejecución a cada pulsación el metrónomo lanzará un evento, en este caso contendrá el valor numérico en el que se producirá la siguiente pulsación en la línea de tiempo del dsp. Para realizar esta predicción, se suma la duración entre pulsaciones al instante actual. En ocasiones se producen desfases por el tiempo de ejecución del código, de modo que el resultado de la suma está desplazado al no partir desde el origen exacto de la pulsación.

En la *Figura 6.1* se ilustra dicho desfase llamado "remainder", que posteriormente se restará para compensarlo. El "remainder" es el resto de la división del momento actual menos el tiempo de la primera pulsación, entre la longitud entre pulsaciones. De esta forma se obvia el número de pulsaciones y sólo se extrae la diferencia.

Figura 6.1 : Pulsaciones sobre el dsp.

```
private void CalculateNextEvent(int beatCounter)
    {
        double remainder = (AudioSettings.dspTime - startTime) % beatDuration;
        nextEventTime = AudioSettings.dspTime + beatDuration - remainder;
        PlayClick(nextEventTime);
        music_events.ChangeBeat();
        if (beatCounter == 4)
        {
            music_events.Changebar(nextEventTime);
        }
        startTime = nextEventTime;
    }
```

6.2 Selección y reproducción de contenedores

Para la selección y reproducción de los contenedores hay que añadir en un *GameObject* de la escena se añadirá el script llamado "ContainerPlayer_Manager". Esta debe alojar los contenedores que se quieran reproducir en la escena, cabe destacar que el nombre del contenedor es muy importante, pues es la referencia que se tiene que pasar a la función *PlayContainer()* para que seleccione y reproduzca el contenedor.

Figura 6.2 : ContainerPlayer_Manager.

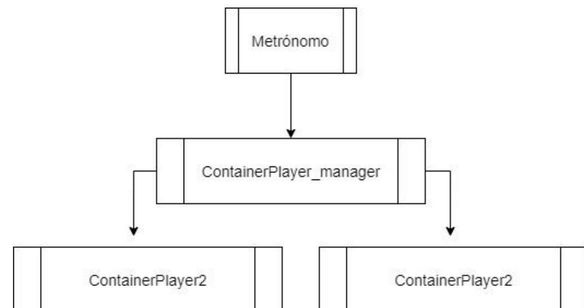

Figura 6.3 : Esquema de ejecución.

A la hora de reproducir un contenedor se ejecutan una serie de métodos que se encargan de generar tantos *AudioSources* como sean necesarios y controlar los tiempos y modos de reproducción, algunos de ellos se encuentran en "*ContainerPlayer_Manager*" y otros en "*ContainerPlayer2*". Como idea general y en orden de ejecución, la metodología de la reproducción consiste en:

- Buscar el contenedor en la lista correspondiente y seleccionarlo.
- Mandar al metrónomo la información necesaria y comenzar la cuenta de las pulsaciones.
- Crear dos clases duplicadas de "*ContainerPlayer2".*
- Crear los *AudioSources* necesarios en la carpeta correspondiente.
- Recibir el evento del metrónomo.
- Discriminar el "*ContainerPlayer2"* que está ocupado.
- Ejecutar las pistas según el modo de reproducción.

A continuación, se exponen algunas de las partes más importantes de esta metodología.

Este es el primer método de ejecución, se encarga de recoger el contenedor mediante su nombre y mandar la información necesaria al metrónomo y a otras variables necesarias. También se encarga de llamar a *CreateMirrorContainer*(), el método que se encarga de duplicar el sistema para generar los *AudioSources* suficientes para evitar los cortes repentinos de audio.

```
public void CreateContainerAuxNeeds(string nameOfContainer)      {
        foreach (so_TrackContainer container in containersAvailabesToPlay) {
            if (container.name == nameOfContainer)
            {
                currentContainer = container;
                contPlayer = CreateContainerPlayer();
                contPlayer.SetContainerProperties();
                CreateMirrorContainer();                       }
            SendPropertiesToMetronome(metronome, currentContainer);
        }
        longestClipDuration = contPlayer.ReturnLongestClipTime();
        playStingerIn = true;
    }
```

El siguiente método está suscrito al evento del metrónomo, de tal manera que se ejecuta cuando hay un cambio de compás (a la cuenta de cuatro pulsaciones). Para determinar si tiene que establecer un nuevo punto de reproducción evalúa si el clip de audio más largo habrá finalizado su reproducción en el siguiente cambio de compás. Si es así, ejecuta los métodos necesarios enviando el nuevo momento de reproducción y asignándolos al sistema que no está ocupado.

```
public void Bar_Event_Listener(double scheduledEvent)
       {
       if (stopClips)
       {
            StingerOut(scheduledEvent);
            contPlayer.Stop_Clips(scheduledEvent);
            contPlayerMirror.Stop_Clips(scheduledEvent);
            metronome.StopMetronomeCount();
            stopClips = false;
            return;
        }
        if (scheduledEvent >= globalEnd_longestCliptime)
        {
            StingerIn(scheduledEvent);
               if (alternator % 2 == 0)
               {
                contPlayer.Play_Clips(scheduledEvent);
               }
               else
               {
```

```
                contPlayerMirror.Play_Clips(scheduledEvent);
               }
            globalEnd_longestCliptime = longestClipDuration + AudioSettings.dspTime;
    if (current_variableClipLength == 0) current_variableClipLength =
globalEnd_longestCliptime;
            alternator++;
        }
        if((scheduledEvent>= current_variableClipLength) &&
currentContainer.audioListPlayVariable.Count!=0)
              {
            PlayNewVariableClip(scheduledEvent);
              }
    }
```

Otra de las funciones importantes es *CheckPlayMode*(), que se encarga de devolver un índice que se usará en la lista que contiene los *AudioClips*, este índice varía según el modo de reproducción. De esta manera se reproducen los clips aleatoriamente o secuencialmente.

```
private int CheckPlayMode()
       {
         switch (currentContainer.playMode)
              {
            case "random":
                Debug.Log("Random" + indexOfVariableClip);
                indexOfVariableClip =
Random.Range(0,currentContainer.audioListPlayVariable.Count-1);
                break;
            case "sequence":
                indexOfVariableClip++;
                if (indexOfVariableClip>
currentContainer.audioListPlayVariable.Count-1)
                          {
                    indexOfVariableClip = 0;
                          }
                Debug.Log("Sequence" + indexOfVariableClip);
                break;
         }
         return indexOfVariableClip;
       }
```

Por otro lado se encuentran los métodos de "*ContainerPlayer2*" cuya función es generar los *AudioSources* necesarios y reproducirlos en base a las condiciones recibidas por "*ContainerPlayer_Manager*". En este mismo script también se encuentra definida una clase llamada "*AudioSources_Creator*" que sirve de clase colaboradora conteniendo ciertas utilidades.

La reproducción de las listas de clips variables y de bucles es muy similar, ambas se recorren para utilizar *PlayScheduled()*. En el caso de los bucles se

reproducen todas indiscriminadamente mientras que en la lista variable se recurre a un índice según el modo de reproducción.

```
public void Play_Clips(double scheduledTime)
      {
      foreach (AudioSource source in l_playAlways_AudioSources)
                  {
                        source.PlayScheduled(scheduledTime);
                  }
      }

public void Play_Clips_Variable(double scheduledTime, int index)
      {
            AudioSource source = l_playVariable_AudioSources[index];
            source.PlayScheduled(scheduledTime);
      }
```

A la hora de detener la reproducción surgen varias cuestiones que marcan el planteamiento de la solución:

- Tienen que destruirse los *AudioSources* que se han creado.
- La destrucción de los canales ocupados tiene que realizarse tras finalizar la pista activa, de lo contrario pararía su reproducción.

```
public void Stop_Clips(double scheduledTime)
      {
            foreach (AudioSource source in l_playAlways_AudioSources)
            {
                  source.SetScheduledEndTime(scheduledTime);
            }
            foreach (AudioSource source in l_playVariable_AudioSources)
            {
                  source.SetScheduledEndTime(scheduledTime);
            }
            AudioSources_Creator utility_AS = new AudioSources_Creator();
            double timeToDestroy = scheduledTime - AudioSettings.dspTime;
            utility_AS.DestroyAudioSources(l_playAlways_AudioSources,
(float)timeToDestroy);
            utility_AS.DestroyAudioSources(l_playVariable_AudioSources,
(float)timeToDestroy);
      }
```

Como la reproducción de *Stingers* es algo puntual, se escoge crear el *AudioSource* y destruirlo conforme el clip acabe de reproducirse.

```
public void PlayStinger(double schedTime, AudioClip stinger)
	{
		AudioSource stingerSource = 
audioSourcesAllocator_go.AddComponent<AudioSource>();
		stingerSource.clip = stinger;
		stingerSource.outputAudioMixerGroup = container.mixerGroup;
		stingerSource.PlayScheduled(schedTime);
		AudioSources_Creator utility_AS = new AudioSources_Creator();
		utility_AS.DestroyStinger(stingerSource, stinger.length);
	}
```

6.3 Escenas de control

Una vez implementado el sistema se crean un par de escenas de control. La primera de ellas muestra una serie de valores para ilustrar la evolución de los datos. La reproducción del contenedor se hace mediante un botón mediante el cual se le pasa a la función el nombre del contenedor deseado, este método es al que se tendría que recurrir desde cualquier parte del código para reproducir la música en un juego real. En la escena se muestra:

- Bar Count: La cuenta del compás.
- Beat Count: La cuenta de la pulsación.
- Dsp time: Es el valor leído de *AudioSettings.dsp.*
- Start Time: El momento en el que empieza el metrónomo.
- Bar Duration: La duración del compás. Está hecho para un 4/4 musical, por lo que representa cuatro veces el *Beat Duration.*
- Beat Duration: Es el tiempo entre pulsaciones.
- Remainder: Es el desfase que se usa para corregir las diferencias entre pulsaciones.
- Next Event: Representa el tiempo en el *dsp* en el que tendrá lugar la siguiente pulsación.
- Longest clip duration: Es el tiempo en segundos del clip más largo.
- Sample Rate processing: La frecuencia de muestreo a la que está trabajando el *dsp.*

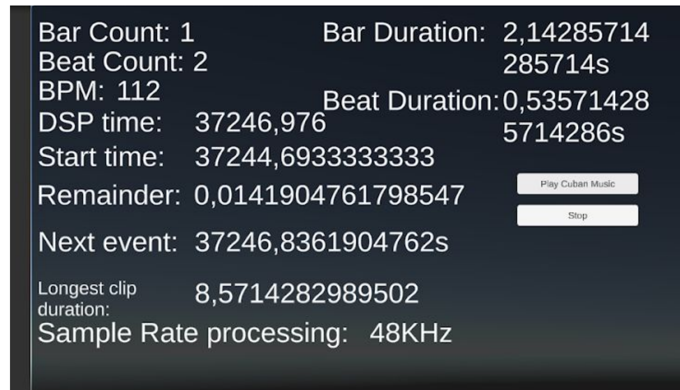

Figura 6.4 : Escena de control

La segunda de ellas se trata de un ejemplo para una de las aplicaciones más comunes en un juego real, en la que la música viene determinada por las zonas en las que se encuentra el jugador. De la misma manera podrían usarse para momentos de combate o de exploración.

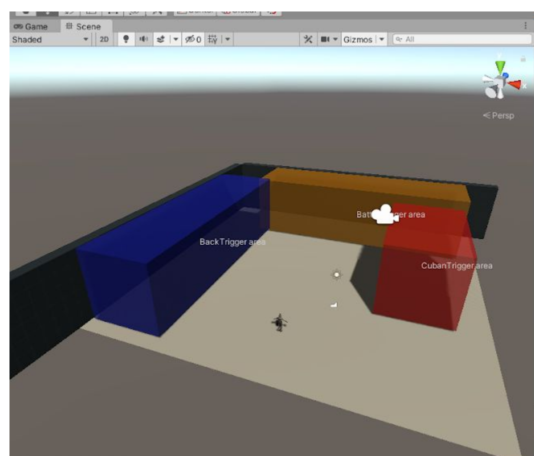

Figura 6.5 : Escena de control con áreas

7. Conclusión

7.1 Aprendizaje durante el desarrollo, análisis de los objetivos, seguimiento y metodología.

Tal y como se comentaba al principio del trabajo, los objetivos consistían en conocer más a fondo cómo funciona el sistema de audio de Unity y encontrar una manera de replicar ciertas funciones que tienen programas comerciales de compañías externas.

Durante el desarrollo el primer gran obstáculo a vencer fue programar la interfaz siendo la primera experiencia de este tipo, entender la manera de gestionar los datos de entrada y los tipos de objetos que se pueden gestionar desde ella. Tras solucionar los problemas de la interfaz, el siguiente aprendizaje consistió en entender cómo se procesa la señal, descubrir los problemas y encontrar la solución. Se siguió la metodología y hoja de ruta presentada al inicio, dirigiéndose según se sucedían las iteraciones y los problemas, pues muchos de los cambios obligaban a reestructurar parte del proceso.

Los objetivos del trabajo se han cumplido, el fin del desarrollo de la herramienta no era competir comercialmente con otras empresas de software que se dedican a este campo, sino explorar y entender los procesos y soluciones que ofrecen para obtener una solución desde el propio motor.

7.4 Líneas de trabajo futuro.

Esta herramienta es una primera aproximación a una de las múltiples soluciones para programar música adaptativa y efectos de sonido. A nivel personal se pretende seguir mejorando la herramienta e implementar nuevas funcionalidades para continuar con el aprendizaje de la programación de audio y música en videojuegos para optar a nuevas oportunidades laborales que puedan presentarse en departamentos de audio, tanto de grandes como pequeños estudios.

8. Bibliografía

La Información [en línea] Disponible en:
https://www.lainformacion.com/estilo-de-vida-y-tiempo-libre/la-historia-de-la-musica-en-los-videojuegos_lbm4azcx4ahijfzwx1t1j5/#:~:text=Los%20primeros%20grandes%20compositores&text=Koji%20Kondo%20comenz%C3%B3%20a%20trabajar,la%20inspiraci%C3%B3n%20y%20la%20genialidad.

Documentación de Unity (ScriptableObject) [en línea] Disponible en:
https://docs.unity3d.com/Manual/class-ScriptableObject.html

moonantonio.github.io [en línea] Disponible en:
https://moonantonio.github.io/post/2018/csharpunity/012/

Wwise [en línea] Disponible en:
https://www.audiokinetic.com/courses/wwise201/?hsCtaTracking=7601e53e-2538-4622-9c5e-a0cdcc680168%7Cfa2f82fc-29bc-4724-89e5-d29c851fd4ee

conceptodefinicion.com [en línea] Disponible en:
https://conceptodefinicion.de/metronomo/