YONDULIB
Herramienta para el uso de sonidos como método de control de videojuegos Unity

Por
Gonzalo Alba Durán
Nuria Bango Iglesias

UNIVERSIDAD COMPLUTENSE
MADRID

Grado en Desarrollo de Videojuegos
Facultad de informática

Dirigido por
Manuel Freire Morán

YONDULIB
Tool for using sounds as inputs for Unity videogames

Madrid, 2021–2022

YONDULIB

Herramienta para el uso de sonidos como método de control de videojuegos en Unity

Memoria que se presenta para el Trabajo de Fin de Grado

Gonzalo Alba Durán
Nuria Bango Iglesias

Dirigido por

Manuel Freire Morán

Facultad de Informática
Universidad Complutense de Madrid

Madrid, 2022

Agradecimientos

A nuestras familias, que nos han apoyado en todo momento y que nos han brindado la posibilidad de estudiar este grado.

A Manuel, sin ti esto obviamente no habría sido posible. Gracias por tu paciencia y tu confianza.

A nuestras psicólogas, que nos han mantenido lo más estables posibles a lo largo de este recorrido.

Por último, no menos importante, a nuestros amigos, de aquí y de allí, de la UCM y del UCM.

Resumen

La industria del videojuego está en crecimiento constante y hace tiempo que dejó de ser un sector enfocado a un público infantil. Hoy en día existen infinidad de juegos que abarcan ámbitos muy diversos, por lo que es fácil encontrar alguno que nos guste y entretenga. Sin embargo, casi todos los juegos se controlan mediante la combinación teclado y ratón en ordenador, o con mandos con botones y joysticks en consola, con esto nos referimos a las entradas de usuario. Este trabajo de fin de grado busca favorecer el desarrollo de videojuegos donde los usuarios puedan interaccionar mediante sonidos, ofreciendo una interfaz para cambiar los controles predefinidos por otros más accesibles y potencialmente divertidos.

Esto se suma a la cantidad de nuevas interacciones que se pueden desarrollar en el ámbito de la experiencia de usuario, sumando nuevas formas de jugar que a su vez implican nuevas maneras de concebir el propio diseño y desarrollo de mecánicas y dinámicas de videojuegos. Para ello, hemos desarrollado un paquete para el conocido entorno de desarrollo de videojuegos *Unity* que permite vincular las diferentes acciones del juego con comandos de sonido como silbar, chasquidos o golpes. Como *Unity* no dispone de APIs integradas para el reconocimiento de sonidos en tiempo real, nos hemos apoyado en la librería *Libsoundio*, que proporciona funciones de entrada de audio de baja latencia.

Gracias a la librería desarrollada, YONDULIB, podemos analizar la entrada y, mediante algoritmos de detección de frecuencias e intensidades, identificar los sonidos. Un desarrollador que use YONDULIB puede asociar sonidos a acciones en el juego, recibiendo retroalimentación gráfica en tiempo real sobre el porcentaje de acierto de cada asociación. Un jugador puede ver en tiempo real cómo son reconocidos sus sonidos por el juego.

Como resultado, este trabajo describe el proceso de diseño y desarrollo de YONDULIB, disponible como código fuente en un repositorio público, y también como un paquete de *Unity* que contiene un ejemplo práctico de uso de la librería. También describimos el proceso de integración del paquete en dos juegos ya desarrollados.

Tras este proyecto se deja un camino marcado sobre el que continuar respecto a *inputs* novedosos por sonido, además de poder servir de comienzo para investigaciones paralelas en cualquier ámbito de entradas de usuario innovadoras en el desarrollo de videojuegos que es, sin duda, un terreno por explotar.

Palabras clave: reconocimiento de sonidos en tiempo real, interacción mediante sonido, *input* de usuario, *input* en videojuegos, *Unity*, *Unity Package*, *Plugins en Unity*, *Input System*

Abstract

The video game industry is constantly growing and has long since ceased to be a sector focused on children. Nowadays, there are countless games covering a wide range of fields, so it is easy to find one that we like and that entertains us. However, almost all games are controlled by a combination of keyboard and mouse on computer, or by controls with buttons and joysticks on console, by which we mean user input. This final degree project aims to support the development of videogames where users can interact with sounds, offering an interface to change the predefined controls for more accessible and potentially fun ones.

This adds to the number of new interactions that can be developed in the field of user experience, adding new ways of playing that in turn imply new ways of conceiving the design and development of videogame mechanics and dynamics. To this end, we have developed a package for the popular videogame development environment *Unity* that allows linking the different actions of the game with sound commands such as whistling, clicking or tapping. Since *Unity* does not have built-in APIs for real-time sound recognition, we have relied on the *Libsound* library, which provides low-latency audio input functions.

Thanks to the developed library, YONDULIB, we can analyse the input and, using frequency and intensity detection algorithms, identify the sounds. A developer using YONDULIB can associate sounds to actions in the game, receiving real-time graphical feedback on the success rate of each association. A player can see in real time how their sounds are recognised by the game.

As a result, this study describes the design and development process of YONDULIB, available as source code in an public repository, and also as a *Unity* package containing a practical example of using the library. We also describe the process of integrating the package into two already developed games.

This project provides a path to be followed in terms of innovative inputs for sound, and can also serve as a starting point for parallel research in any area of new user input in videogame development, which is certainly an area still to be looked into.

Keywords: real-time sound recognition, sound interaction, user input, videogame input, *Unity*, *Unity Package*, *Unity* plugin, *Input System*.

Índice general

Página

Capítulo 1

Introducción

1.1. Antecedentes

Los videojuegos tienen una historia, respecto a la interacción del usuario con la máquina, con una evolución progresiva, pero sin grandes saltos innovadores en su gran mayoría.

Los métodos de entrada se ven en gran parte limitados por el hardware de cada época, por lo que es comprensible ver como en los inicios de los videojuegos se veían claramente influenciados por los controles de las recreativas (*joystick* y botones) y los propios teclados de los ordenadores.[1]

Figura 1.1: Controlador *Joystick* de la consola *Atari 2600* (1977)[1]

Hay grandes hitos en la historia de los videojuegos. Sin duda, uno de ellos fue la innovación en métodos de entrada en 2006, cuando salió la *Nintendo Wii* con su *Wii Remote* o *WiiMote*. El *WiiMote* supuso un cambio respecto a los *gamepads* hasta la fecha con un nuevo tipo de *input*. Consiste en la detección del movimiento del mando en el espacio a través de su giroscopio (y en versiones posteriores su acelerómetro) y su puntero de infrarrojos[2]. Unos años más tarde, en 2010 salió *Kinect*, un dispositivo para la *Xbox360* que permitía interactuar con la consola a través de gestos y comandos de voz, sin necesidad de utilizar un controlador de videojuegos tradicional. El dispositivo estaba compuesto por una cámara, sensores de audio y tecnología capaz de captar el movimiento[3].

Los videojuegos hoy en día tienen una gran variedad de recursos con relación al *gameplay* y a la interacción del usuario con el juego. Las lineas entre accesibilidad, innovación y evolución se difuminan en aspectos como la realidad virtual.

En la realidad virtual se abre un abanico entero de nuevos métodos de entrada y controladores ya que existen desde cámaras hasta trajes hápticos, que se complementan para proporcionar experiencias distintas en cada juego.

Ante el mercado actual y sus variados dispositivos de entrada surgen distintos problemas o cuestiones a tener en cuenta como son la compatibilidad, el precio y la accesibilidad.

Por ejemplo, a pesar de lo innovador que fue en su momento el *WiiMote* era conocido también por sus altos precios. En la actualidad, se puede ver como la realidad virtual está poco extendida debido a restricciones técnicas de los equipos que lo soportan, o simplemente por su precio elevado, dando así datos como que solo algo más del 2,1 % de usuarios totales de *Steam* utilizan gafas de realidad virtual en 2022. [4]

1.2. Motivación

Este trabajo se inspira en cómo, cada nuevo tipo de *input* de usuario, abre puertas a la innovación en el desarrollo videojuegos. La motivación nace de la exploración de nuevos controladores en la búsqueda por ampliar nuestros conocimientos en este ámbito, así como hacer accesible esta frontera a otros desarrolladores. [5]

YONDULIB nace para encontrar nuevas formas de control, inspirándose en el principal objetivo de los videojuegos, entretener. Buscamos algo potencialmente novedoso, pero sin renunciar a la diversión y que fuese viable en un trabajo de estas características. De aquí nace el nombre y parte de la funcionalidad, inspirado en el célebre personaje de *Guardianes de la Galaxia*, *Yondu*, que controla su arma mediante silbidos.

El sonido se puede entender como una herramienta al alcance de todos que nos permite comunicar cualquier cosa. Utilizando estos conceptos como pilares fundamentales, nace una idea que permite y facilite el manejo de controladores distintos en el desarrollo de videojuegos.

1.3. Objetivos

Con este trabajo se pretende explorar la gestión de sonidos como input de usuario, con el fin de proporcionar un mecanismo de control para videojuegos usando sonidos, que sea

[1]Fotografía de Evan Amos en Wikipedia, con licencia de dominio público

aplicable de manera cómoda a cualquier videojuego en desarrollo.

Una de las referencias de reconocimiento de sonidos es el reconocimiento de voz en los *smartphones*. Esta funcionalidad está muy optimizada y tiene una latencia baja en el reconocimiento de palabras y frases. A pesar de esto, para controlar un videojuego, este tiempo de respuesta sigue siendo demasiado largo.

Tratándolo desde la perspectiva de desarrolladores, buscamos características que nos gustaría encontrarnos en un proyecto semejante, estableciendo tres objetivos principales:

- Objetivo 1: reducir la complejidad de los sonidos reconocidos, simplificándolas en golpes o silbidos, para reducir el tiempo de procesamiento y, por ende, el tiempo de respuesta. Para ello, necesitaremos que el acceso al audio tenga baja latencia y el reconocimiento tenga lugar en tiempo real.
- Objetivo 2: que sea posible, para desarrolladores, añadir nuevos tipos de sonidos asociados a acciones. Para ello, necesitamos que la estructura sea ampliable.
- Objetivo 3: que todo el paquete sea utilizable. Para ello, necesitaremos simplificar la integración en juegos lo más posible.

Desde la perspectiva del jugador se quiere conseguir una experiencia intuitiva y sencilla a la hora de cambiar los controles y ofrecer alternativas a los controles clásicos del videojuego. Este objetivo se apoya en propuestas innovadoras del pasado como los sistemas de input de la consola *Nintendo Wii* o *Xbox Kinect*, que crearon un dispositivo, respectivamente, que captaba el movimiento del jugador.

1.4. Plan de trabajo

Nuestro plan de trabajo se ha basado en iteraciones cortas, y está inspirado en metodologías ágiles. En este caso con reuniones bisemanales (cada dos semanas) con el tutor y semanales entre los miembros del proyecto, de las cuales se tienen actas que plasman el trabajo a realizar para la siguiente reunión, así como hitos completados y no completados de esa reunión.

En total, hemos realizado 27 reuniones de seguimiento. Además, hemos hecho uso de un repositorio en *GitHub* para mantener nuestro código, y para elaborar la memoria hemos usado el sistema de edición colaborativo de *LaTeX OverLeaf* con la plantilla de *TeFlon*[2] proporcionada por la UCM, que permite un acercamiento a esta herramienta sin previo conocimiento.

La plataforma que hemos elegido para desarrollar este proyecto es *Unity* con C# como lenguaje de programación. Esto nos permite agilizar procesos y ahorrar tiempo de desarrollo frente a otras opciones como *Unreal*, en las que tenemos menos experiencia de desarrollo.

Para acompañar el desarrollo en *Unity* hemos utilizado el IDE *Visual Studio 2019* y *GitHub* tanto como para el repositorio en el que se encuentra el código fuente [3] del

[2]LaTeX OverLeaf con la plantilla de TeFlon `https://www.ucm.es/oficina-de-software-libre/publicaciones`

[3]GitHub del proyecto de desarrollo de YONDULIB `https://github.com/nubango/yondulib-project`

trabajo como para el repositorio ligado al enlace de descarga del paquete [4].

Una visión general de los pasos a seguir para cumplir con los objetivos son los siguientes:

- Creación de la librería de reconocimiento de sonido
- Creación del dispositivo específico en *Unity*
- Integración de la librería con el input de *Unity*
- Utilización de la librería en juegos reales

Los objetivos cumplidos se verán reflejados en las integraciones del paquete en distintos tipos de proyectos.

- Demo: Una escena en primera persona que sirve de ejemplo de cómo utilizar YONDULIB.
- Juego 1: Integración de YONDULIB en un videojuego desarrollado previamente en el que muestra el uso de la mecánica del silbido.
- Juego 2: Integración de YONDULIB en un videojuego desarrollado previamente en el que se muestra la combinación de YONDULIB con diferentes dispositivos para realizar las diferentes mecánicas.

Los diferentes contratiempos a lo largo del desarrollo del proyecto nos permiten dividirlo en dos etapas. La primera comenzó con 3 integrantes, con propuestas que iban enfocadas entorno a interfaces de usuario y herramientas similares a *Android Studio Layout Editor*, pero en *Unity*, surgiendo así "Herramienta para la Autoría y Generación de Interfaces de Usuario en *Unity*".

Avanzando hasta ya entrado el curso académico 2020-2021, surgen dos novedades que cambian el rumbo del proyecto. Por una parte la implementación por parte de *Unity* del *UI Builder* [5], que a grandes rasgos era la idea que estábamos desarrollando. Por otra parte, tras los resultados de los exámenes extraordinarios, el tercer alumno implicado en un principio en el proyecto no pudo continuar en el proyecto.

La segunda etapa, en la que ya únicamente hay 2 integrantes, la compone el desarrollo de este proyecto que, tras evaluar la situación, nos llevó a buscar alternativas a la idea original. Finalmente, decidimos tomar un camino diferente relacionado con el *input* de usuario y el control por sonidos. En la siguiente tabla se muestran los hitos temporales y las diferentes etapas explicadas:

[4]Repositorio del paquete a descargar en *Unity* `https://github.com/nubango/yondulib`
[5]`https://docs.unity3d.com/Packages/com.unity.ui.builder@1.0/manual/index.html`

Hitos temporales		
Año	**Fecha**	**Contenido**
2020	18 Junio	Presentación de participantes y alta del proyecto
	6 Julio	Repartir tareas para el verano y evaluar riesgos
	13 Julio	Reunión sin profesor para organizar investigaciones de verano
2021	23 Abril	Estructurar calendario de entrega para septiembre
	30 Abril	Propuestas de nuevas direcciones para el proyecto
	7 Mayo	Procesamiento de audio e input en *Unity*
	14 Mayo	Ejemplo de movimiento con audio y búsqueda de APIs
	21 Mayo	*Debugs* con LASP
	4 Junio	Eventos diferenciados de sonidos y menú de controles *ingame*
	25 Junio	*Delegates* y separación de frecuencias para reconocedores de sonidos
	30 Julio	Planificación verano
	3 Septiembre	Reconocedor con valores continuos y no discretos
	18 Octubre	Depuración visual y estructuración de memoria
	4 Noviembre	Integración en escena y gestión de fechas Navidad
	15 Noviembre	Estructura de puntos de la memoria
2022	23 Marzo	Corrección de memoria y estructura fechas
	30 Marzo	Creación repositorio para *Unity Package*
	6 Abril	Paquete con una demo y secciones de la memoria
	13 Abril	Corrección de la memoria
	22 Abril	Corrección de la memoria: objetivos
	27 Abril	Corrección de la memoria: resumen, introducción y estado del arte
	4 Mayo	Corrección de la memoria: diseño
	11 Mayo	Planificación: entrega en septiembre
	3 Septiembre	Revisión del borrador completo
	6 Septiembre	Entrega del borrador
	12 Septiembre	Revisión de las pruebas implementadas
	15 Septiembre	Reunión final con revisión completa

Cuadro 1.1: Hitos temporales con fechas e información de las reuniones.

Capítulo 2

Estado del arte

Resumen: Este capítulo detalla conceptos importantes que ayudan a entender el desarrollo del proyecto. En el primer apartado (2.1) explicamos brevemente el sonido y como lo detectamos con los micrófonos. Estos dispositivos son de entrada/salida y en el siguiente punto (2.2) explicamos su funcionamiento y repasamos algunos usos interesantes de los periféricos en videojuegos. Finalmente, el último punto (2.3) expone qué es un motor de videojuegos y cuáles son las opciones que hay en la actualidad.

2.1. ¿Qué es el sonido?

El sonido son ondas que se propagan a través de un medio (fluido o sólido) que están generadas por el movimiento vibratorio de un cuerpo. El oído es el órgano sensorial capaz de detectar y transformar estas fluctuaciones del medio en impulsos nerviosos que son enviados al cerebro.

2.1.1. Física del sonido

La física del sonido es estudiada por la acústica, que trata la propagación de las ondas sonoras por los diferentes tipos de medios y las interacciones que se producen con los cuerpos físicos.[6]

Las ondas sonoras se producen cuando un cuerpo vibra rápidamente. Dichas vibraciones producen una diferencia de presión en el medio que genera una onda que se propaga a través de este.

La frecuencia es el número de oscilaciones o variaciones de presión por segundo. El ser humano es capaz de detectar ondas sonoras con una frecuencia comprendida entre 20 y 20.000 hercios. Un sonido grave corresponde a una onda sonora con frecuencia baja mientras que las frecuencias más altas corresponden con sonidos más agudos.

La amplitud indica la magnitud de las variaciones de presión producidas por las vibraciones de un cuerpo. El volumen de un sonido aumenta en función de la amplitud de su onda. La unidad que se utiliza para medir este valor es el decibelio (dB). El rango del oído humano se comprende entre 0 dB, umbral de audición, y 120 dB, considerado como

umbral de dolor y a partir del cual se producen daños irreversibles en el sistema auditivo. Cada 10 dB representa aproximadamente el doble de volumen de sonido percibido.

2.1.2. Detección del sonido. Procesamiento del sonido.

El micrófono es un dispositivo de entrada capaz de captar ondas sonoras y transformarlas en energía eléctrica para aumentar su intensidad y poder transmitirla. Para realizar este proceso el micrófono se divide en los siguientes componentes: diafragma, dispositivo transductor, rejilla y carcasa.

Los micrófonos pueden transmitir el audio recogido de dos formas diferentes: de forma analógica o digital. En una señal analógica, la información, para pasar de un valor a otro, pasa por todos los valores intermedios, es decir, es continua. Por el contrario, en una señal digital pasa de un valor al siguiente sin poder tomar valores intermedios, por lo tanto, es discontinua, y solo puede tomar valores discretos.[1]

Esta señal es recogida por otro dispositivo que la procesa. Según el tipo de dispositivo, la señal debe ser analógica o digital, por lo que hace falta un intermediario que pueda convertir y gestionar dicha señal.

La tarjeta de sonido, a través de un ordenador, es el dispositivo intermediario que convierte las señales analógicas en digitales. El ordenador gestiona la señal digital utilizando un programa informático o *driver*.

Los ordenadores genéricos, sobre todo portátiles, suelen llevar una tarjeta de audio integrada en la placa base. También existen tarjetas de sonido independientes, que en el ámbito profesional son conocidas como interfaces de audio. Suelen tener múltiples conectores de entrada y salida de audio para un procesamiento del sonido más profesional.

2.2. Entrada/Salida

Un computador debe tener la capacidad de comunicarse con el exterior para que podamos interactuar con él. Los dispositivos de entrada/salida (E/S) son aquellos que permiten dicha comunicación. Las computadoras disponen de un sistema de E/S, que está formado por hardware y software específico, para gestionar de manera efectiva dicha comunicación.

2.2.1. Micrófonos como dispositivos de E/S

Los micrófonos se conectan al computador, habitualmente, a través de la interfaz externa *minijack* o *jack* de 3.5mm. Este puerto es el más común en dispositivos electrónicos debido a que es compatible con la mayoría de auriculares con cable del mercado. También existen micrófonos, digitales, que se conectan a través del conector USB tipo A.

Con la llegada del estándar USB tipo C, cada vez más empresas están optando por eliminar el puerto *jack* de 3.5mm, sobre todo en *smartphones* donde prima la impermeabilidad y el aprovechamiento del espacio.

[1]Señal analógica y digital `https://www.maupe.com/Empresa/senal-analogica-vs-senal-digital/`

Para la gestión de los micrófonos, el sistema operativo brinda a los programadores una interfaz específica con la capacidad de gestionar parámetros básicos de la captación de audio, administrar canales, conexión al dispositivo, etc. Cada sistema operativo tiene una o más APIs específicas que realizan esta función, lo que dificulta el soporte multiplataforma de programas de audio.

En *Windows*, el sistema operativo de *Microsoft*, existen unas interfaces de programación para aplicaciones de audio bajo el nombre *Windows Core Audio API*, que fueron incorporadas por primera vez en la versión *Windows Vista*. Esta API se divide en 4 partes que gestionan diferentes sectores del audio [7]:

- *API Windows Multimedia Device* (*MMDevice*): sirve para enumerar los dispositivos con punto de conexión de audio en el sistema.
- *Windows Audio Session API* (*WASAPI*): sirve para crear y administras secuencias de audio hacia y desde dispositivos con punto de conexión de audio.
- *DeviceTopology API*: sirve para acceder directamente a las características topológicas, como controles de volumen y multiplexores, que se encuentran a lo largo de las rutas de acceso de datos dentro de los dispositivos de hardware en adaptadores de audio.
- *EndpointVolume API*: sirve para acceder directamente a los controles de volumen de dispositivos de audio. Esta API la utilizan principalmente las aplicaciones que administran secuencias de audio en modo exclusivo

Las cuatro APIs que forman *Windows Core Audio API* sirven como base para otras API de nivel superior como *DirectSound* que elevan las funcionalidades y proporcionan mayor flexibilidad a la hora de gestionar el sonido proporcionado por los dispositivos conectados.

En *GNU/Linux*, *Advanced Linux Sound Architecture* (ALSA) es un software libre perteneciente al núcleo de *Linux* que se comunica con el hardware relacionado con el sonido, como tarjetas de sonido e interfaces MIDI. Desde la versión 2.6, es la API oficial para audio en el *kernel Linux*.

ALSA proporciona una API llamada *libasound* que recomiendan utilizar para desarrollar aplicaciones, ya que proporciona una interfaz de programación de mayor nivel que la interfaz del *kernel* y más fácil de manejar para los desarrolladores. Esta API se puede dividir en las siguientes interfaces principales:[8]

- Interfaz de control: Permite administrar registros de tarjeta de sonido y consultar los dispositivos disponibles.
- Interfaz PCM: Sirve para gestionar la captura y reproducción de audio digital.
- Interfaz *Raw* MIDI: Esta API proporciona acceso al bus MIDI de la tarjeta de sonido. MIDI es un estándar para instrumentos musicales electrónicos.
- Interfaz de temporizador: proporciona acceso al temporizador hardware de la tarjeta de sonido para sincronizar eventos de sonido.
- Interfaz de secuenciador: es una interfaz de nivel superior construida sobre la interfaz *Raw* MIDI para programadores MIDI y síntesis de sonido. Maneja gran parte del

protocolo y sincronización MIDI.

- Interfaz de mezclador: API construida sobre la interfaz de control que gestiona los dispositivos en las tarjetas de sonido que enrutan y controlan los niveles de volumen.

ALSA proporciona compatibilidad con la antigua API *Open Sound System* (OSS) lo que permite que la mayoría de aplicaciones con esta API puedan seguir ejecutándose sin cambios.

En *Mac OS*, *Apple* proporciona un conjunto de herramientas para gestionar una amplia variedad de funcionalidades del sonido en sus dispositivos. Permite interactuar a bajo nivel con el hardware de los dispositivos de audio, gestionar la comunicación con dispositivos MIDI, administrar la reproducción de archivos de audio, manejar grabaciones en tiempo real y procesamiento de audio de baja latencia y creación de complementos (*plugins*) para crear efectos de audio, instrumentos y utilidades para utilizarlas dentro de las aplicaciones.[9]

Al margen del sistema operativo, existen APIs que intentan unificar el desarrollo y facilitar la generación de software multiplataforma. Estas librerías hacen un complicado ejercicio de abstracción de funcionalidades comunes de los sistemas para proporcionar una interfaz idéntica y coherente sin importar en el entorno en el que se encuentre el programador. Un ejemplo claro es *Libsoundio* que es una abstracción ligera sobre varios controles de sonido que proporciona una API que funciona independientemente del controlador de sonido que se conecte y que que soporta los tres sistemas operativos nombrados anteriormente.

2.2.2. E/S en videojuegos

A lo largo de la historia del videojuego han existido multitud de plataformas *hardware* que, junto con los avances tecnológicos, han ido evolucionando, aumentado la capacidad de cómputo y explorando formas de interacción innovadoras.

La estandarización actual es una comodidad que no siempre ha existido y en una época más temprana de la industria del videojuego, los ordenadores y consolas eran mucho menos potentes y su acceso y utilización no estaba tan extendido como en la actualidad.

Las primeras consolas de videojuegos fueron las máquinas recreativas que tenían integrados los controles en la propia estructura. Se componían de varios botones y un *joystick*. Más adelante llegaron las consolas domésticas, que tenían un tamaño mucho más reducido, y con ellas los mandos de videojuegos. Estos dispositivos se conectaban a la consola por un puerto mediante un cable.

Con el avance de la tecnología y el aumento de la potencia de los ordenadores y consolas, los mandos también se convirtieron en dispositivos de salida con la respuesta háptica a través de la vibración del mismo. Los mandos se libraron de los cables con la llegada de las conexiones inalámbricas.

El videojuego *Guitar Hero* fue novedoso debido a la forma de guitarra del periférico con el que se jugaba. La *Wii* de *Nintendo* supuso una innovación en cuanto a interacción con los videojuegos se refiere. El mando era inalámbrico y tenía varios botones, pero la novedad principal fue en la incorporación de sensores que percibían el movimiento del mando, que supuso un gran avance en el sector. Fue acompañado de diferentes accesorios

que transformaban el modo de uso del mando. También lanzaron el videojuego *Wii Fit* que combinaba el mando de la consola con una tabla que media el peso que había sobre ella, lo que permitió una mayor inmersión y una experiencia innovadora.

El avance de las consolas va unido al desarrollo de la computación genérica y esto supone una tendencia a la estandarización de las interfaces *hardware*. Actualmente las consolas y los periféricos son compatibles con estándares como USB o *Bluetooth* (para dispositivos inalámbricos).

2.2.3. Juegos que utilizan sonido como mecánica

Existen videojuegos que han utilizado los sonidos como mecánica. El videojuego LOOM[2] es una aventura gráfica lanzado por la compañía *LucasFilm Games* en el año 1990. La jugabilidad consiste en lanzar hechizos a través de un bastón mágico. Lo interesante es que para lanzar el hechizo correctamente sonará unas notas musicales que tendrás que repetir en el orden correcto. Aun así, no utiliza el sonido como *input*, aparece un pentagrama en la pantalla y es con el ratón, pulsando en la zona correcta, con el que generas el sonido.

Actualmente, hay varias propuestas que sí utilizan realmente el sonido como *input* para el control de los videojuegos. *Saltar pollo* (2021)[3], *Scream Go Hero* (2017)[4] y *Yasuhati* (2017)[5] son tres juegos de plataformas simples donde controlas al personaje a través de sonidos. En los tres, el esquema de control con sonido es muy simple: si hay cualquier sonido más allá de un silencio de base, se realizan ciertas acciones, en función de la intensidad de los sonidos. Para mover al personaje hay que realizar sonidos poco intensos. La otra mecánica que tienen es saltar, que está mapeada a la detección de sonidos más intensos. Así que cuanto más alto emitas sonidos, más alto saltará el personaje. Estas mecánicas son poco precisas por lo que es común que salte cuando queremos que avance.

2.3. Motores de videojuegos

Los motores de videojuegos son un programa y conjunto de herramientas que permiten agilizar todo el proceso de desarrollo de un videojuego. A su vez, proporciona distintos motores como el de renderizado, el de sonido, el de físicas, así como inteligencia artificial, gestión de memoria y *scripting*.

El término motor de videojuegos fue acuñado a principios de los 90, asociándolo al inicio de los videojuegos en 3D. Sin embargo el primer motor de videojuego en 3D del que se tiene conocimiento fue *Freescape Engine*, desarrollado por *Incentive Software* en 1986.[10] [11]

Muchos motores de videojuegos están especializados en géneros concretos para atender necesidades específicas del mismo y de esta manera conseguir una mayor eficiencia en el desarrollo del producto final.

[2]LOOM `https://store.steampowered.com/app/32340/LOOM/`

[3]Saltar pollo `https://play.google.com/store/apps/details?id=com.IsaacGame.JumpingChicken&hl=es&gl=US`

[4]Scream Go Hero `https://play.google.com/store/apps/details?id=com.ketchapp.screamhero&hl=es_HN`

[5]Yasuhati `https://play.google.com/store/apps/details?id=jp.ne.freem.YASUHATI&hl=es_HN`

Hoy en día, existen distintos tipos de motores dependiendo de su origen: código abierto, *shareware*, pago, etc. Existe una gran variedad de motores de videojuegos y entre las grandes empresas los más conocidos (junto con algunos de los juegos en los que se utiliza) son: [12]

- *Decima (Horizon Zero Down, Death Stranding, Until Down)*
- *Source Engine 2 (Dota 2, Titanfall 2, Portal 2)*
- *UniArt FrameWork (Rayman, Child of Light))*
- *AnvilNext (Assasin's Creed Origins, For Honor)*
- *Id Tech 5 (Doom 4, Dishonored 2)*
- *FOX Engine (Metal Gear Solid V, PES 2017)*
- *Frostbite (EA: Battlefield 1, Battlefront II, FIFA18)*

Todos ellos son motores que han sido desarrollados para el uso interno y desarrollo de juegos de una compañía específica.

Pero desarrollar un motor es muy costoso y la mayoría de los estudios no se pueden permitir desarrollar uno, por lo que existen empresas que se dedican al desarrollo de motores de uso más genérico, no con ello de peor calidad, para que puedan licenciarlo terceros. Entre los más utilizados están *Unreal Engine* y *Unity*.

En otra escala, se pueden mencionar también *RPG Maker*, *Construct*, *Phaser*, *Gamemaker* o *Scratch*, que se utilizan en ámbitos distintos como la enseñanza o para proyecto a menor escala, pero siguen siendo motores con sus características diferenciadoras.

Cada motor gestiona las físicas, el sonido o las animaciones de una manera diferente. Para el motor de físicas de los más populares está *PhysiX*, en el apartado gráfico *DirectX* u *OpenGL* y en sonido FMOD. Además, el lenguaje en el que se programa con cada motor varía: Unity utiliza lenguajes orientados al scripting como *UnityScript* y C# mientras que Unreal trabaja con C++.[13][14]

Capítulo 3

Audio como *input* en videojuegos

Resumen: Este capítulo detalla el diseño e implementación de YONDULIB, una librería que permite usar sonidos recogidos por el micrófono de un ordenador como entradas (controles) en videojuegos *Unity* que la emplean. El primer apartado (3.1) presenta las consideraciones de diseño que llevaron a la implementación en sí (apartado 3.2). Finalmente, un último apartado (3.3) describe las pruebas realizadas para asegurar su buen funcionamiento. *(Desde la figura 3.7 hasta la 3.16 ha sido utilizado en Lucid app*[1] *para crearlas)*

3.1. Diseño

Para cumplir el objetivo de crear una herramienta que permita manejar un videojuego a través de sonidos, primero tuvimos que decidir para qué entorno queríamos desarrollar esta herramienta. *Unity* es el entorno más familiar para nosotros y es la principal ventaja frente a otros entornos. Lo primero que hicimos fue examinar las posibilidades que nos ofrecía en el ámbito de los dispositivos de input y si existía alguna herramienta relacionada con el audio que nos pudiese ayudar. *Unity* contiene un sistema de *input* pero no ofrece herramientas para el control de micrófonos, así que tuvimos que buscar alguna librería externa al motor que nos proporcionase las funcionalidades que estábamos buscando. Las librerías más famosas de gestión de audio, que tienen compatibilidad con *Unity* como FMOD, no ofrecen acceso al audio con baja latencia. Por lo que tuvimos que buscar librerías fuera del ámbito de *Unity*.

Empezamos por el acceso al audio en *Windows*, pero nos dimos cuenta de la complejidad de acceder directamente al audio y lidiar con librerías de bajo nivel. Entonces, volvimos a *Unity* y empezamos a buscar *plugins* y librerías de particulares que pudiesen proporcionar el acceso al audio con baja latencia. Encontramos el proyecto *LASP*[2] (procesamiento de señal de audio de baja latencia para *Unity*) desarrollado por Keijiro Takahashi, que permite acceder al audio de baja latencia a través de un *wrapper*, desarrollado en c#, de la librería *Libsoundio*[15]. En la subsección 3.2.3 se profundiza más en esta librería y el *wrapper* nombrado, el cual es multiplataforma y concede acceso al audio de baja latencia en *Windows*, *Linux* y *MacOS*.

[1] Lucid app `https://lucid.app/`

[2] LASP `https://github.com/keijiro/Lasp`

Una vez obtuvimos el acceso al audio en *Unity*, pudimos desarrollar los algoritmos de reconocimiento de sonidos. El desarrollo pasó por varias fases de refactorización de código y mecánicas hasta llegar a dos prototipos de reconocedores, uno de golpes, palmas y chasquidos y otro de silbidos. La implementación de estos reconocedores se explica más adelante en la subsección 3.2.5 A la hora de generar eventos específicos de los reconocedores en el input de *Unity* nos encontramos con la inviabilidad de extender el *Input Manager*, debido a la nula flexibilidad de este motor para generar nuevos tipos de eventos. Finalmente, decidimos utilizar *Input System*, que es un nuevo motor de *Input* de *Unity* que fue lanzado en 2019 y que ofrece la flexibilidad que necesitamos.

3.1.1. Entorno de desarrollo

El proceso de creación de un videojuego abarca desde el concepto inicial hasta el videojuego en su versión final. Es una actividad multidisciplinaria, que involucra profesionales de la programación, diseño gráfico, animación, sonido, música, actuación, etc.[16]

Para crear videojuegos existen multitud de caminos. Para escoger de forma adecuada primero hay que tener en cuenta las necesidades propias, las especificaciones del *software* y la documentación existente, así como los conocimientos relacionados con el lenguaje de programación utilizado. Aquí algunas opciones:

- DIY (*Do It Yourself*)
- Web
- Multiplataforma
- *All In One* (Unity , Unreal, etc)

La opción que más nos conviene es la utilización de motores *All In One*, ya que a lo largo de la carrera hemos adquirido conocimientos suficientes en este área. Como motor de videojuegos, Unity destaca, entre los desarrolladores, por la comunidad de usuarios activos en foros y la documentación extensa que proporciona. Si buscas en *Google "create a video game"* recibirás 5.000 millones de resultados. De todos, *Unity* estará en 70 millones de ellos. *Unreal Engine* en 36 millones. *Game Maker Studio* en 2 millones [3]. Con estos datos podemos asumir que un usuario principiante no coja el camino predeterminado por estas búsquedas y mucho menos que cree su propio motor.

No es algo que se resuma a simples búsquedas. En el caso de *Unity*, y según datos de la propia compañía, hablamos de una base de 3.000 millones de usuarios y una presencia del 50 % en el catálogo de juegos de móviles. Un 90 % del mercado si nos ceñimos a la corriente de realidad aumentada y virtual. Un segmento en el que los motores propios apuntan a una media muy inferior.[17]

Por todos estos datos, y sumándose a que a lo largo de toda la carrera hemos utilizado *Unity*, se decide utilizar este motor, con *Unity Packages* como forma de instalación de la librería.

[3]Búsquedas "create a video game" `https://www.xataka.com/videojuegos/indies-tenian-razon-unity-motores-terceros-le-han-ganado-partida-a-motores-propios-a-hora-crear-juegos-1`

3.1.2. Acceso a audio de baja latencia

Primero, optamos por investigar las opciones que nos brindaba un motor de audio para videojuegos como FMOD, que tiene compatibilidad en *Unity*. FMOD ofrece multitud de funcionalidades en post-procesado pero ninguna función para manejar audio en tiempo real ni acceso a audio con baja latencia. Por lo que decidimos buscar *plugins* del motor que proporcionasen estas funcionalidades. Encontramos un *plugin* que encajaba a la perfección con lo que estábamos buscando, llamado ADX2. ADX2 es un *middleware* de audio para videojuegos creado por *Criware*[4]. El principal inconveniente de este *software*, es que la licencia para poder utilizarlo es de pago, y aunque para estudios pequeños y desarrolladores la licencia es gratuita, para poder descargarlo hay que realizar una solicitud.

Paralelamente investigamos otras opciones. La siguiente fue acceder directamente a la librería proporcionada por el sistema operativo. Esta opción sí que nos proporcionaba acceso al micrófono y a audio de baja latencia, pero nos encontramos con el inconveniente de que nos limitaba el desarrollo exclusivo para el sistema operativo del desarrollo, en este caso *Windows*. La librería que proporciona *Windows* se llama WASAPI[5] y está programada en C, lo que supone otro inconveniente ya que en *Unity* se programa en el lenguaje C#.

Para solucionar el primer problema, buscamos librerías que idealmente fuesen multiplataforma y lidiasen con el sistema operativo, facilitándonos el acceso al micrófono y al audio. Existen varias librerías de audio que cumplen esta premisa: *PortAudio*[6], *RtAudio*[7], SDL2[8], JUCE[9] y *Libsoundio*[15].

Una vez solucionado el problema de la abstracción de la librería del sistema operativo, teníamos que encontrar la forma de unir alguno de estos *backends* con *Unity*. Ninguno de ellos ofrecía soporte para este motor de forma oficial, por lo que investigamos fuentes no oficiales que ofrecieran una solución. Entonces encontramos el proyecto LASP (procesamiento de señal de audio de baja latencia para *Unity*) desarrollado por *Keijiro Takahashi*, que consiste en crear imágenes y formas geométricas a partir de audio detectado en tiempo real por un micrófono. *Keijiro*, en este proyecto, para acceder al audio de baja latencia desarrolló un *wrapper* en C# que permite utilizar la librería *Libsoundio*. Este *wrapper* era la solución que andábamos buscando. Permite acceder al audio con baja latencia, ofreciendo un flujo de datos sin procesar. Es una librería multiplataforma y compatible con multitud de *backends* como JACK, *PulseAudio*, ALSA, *CoreAudio* y WASAPI.

3.1.3. Reconocedores de sonidos

Una vez obtuvimos el acceso al audio y a los micrófonos desde *Unity*, pudimos empezar a desarrollar el reconocedor de sonidos. Comenzamos con dos reconocedores distintos, uno de palmas y otro de chasquido de dedos. En ellos analizábamos tres parámetros: la frecuencia con más intensidad, la intensidad total de la muestra (suma de la intensidad de todas las frecuencias) y la duración en el tiempo de esa frecuencia. De esta

[4]Criware `https://www.criware.com/en/index.html`
[5]WASAPI `https://docs.microsoft.com/es-es/windows/win32/coreaudio/wasapi`
[6]PortAudio `http://www.portaudio.com/`
[7]RtAudio `https://www.music.mcgill.ca/~gary/rtaudio/index.html`
[8]SDL2 `https://www.libsdl.org/`
[9]JUCE `https://juce.com/`

manera conseguíamos distinguir entre chasquidos y palmas, sin embargo, la fidelidad era bastante baja ya que si un chasquido era demasiado fuerte se detectaba como palmada y, al contrario, si una palmada era demasiado débil, podía detectarse como chasquido. Finalmente, optamos por unificar el reconocimiento de los dos sonidos en un mismo reconocedor. A continuación, se muestra una comparación entre las frecuencias detectadas de una palmada contra las frecuencias generadas por un chasquido.

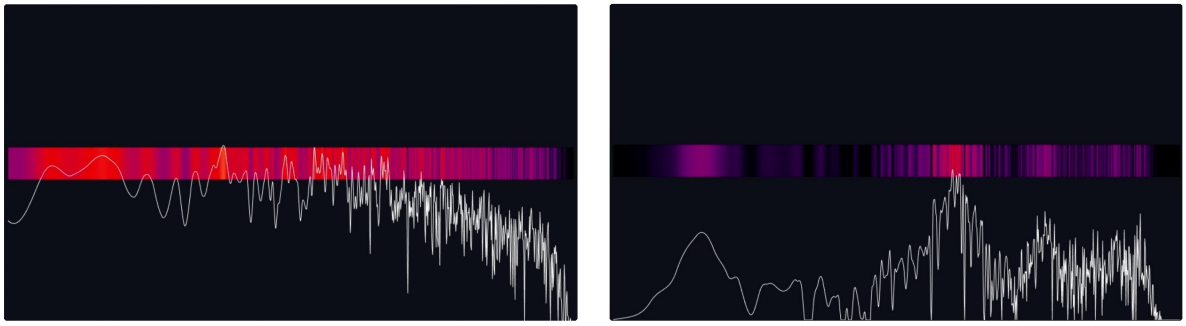

Figura 3.1: Frecuencias de una palmada (izquierda) vs. las de un chasquido (derecha)

Al unificar los dos reconocedores, sentimos la necesidad de buscar otro sonido fácil de realizar y que aportara un poco más de profundidad. Decidimos realizar un reconocedor de silbidos. Comenzamos partiendo de la base del reconocedor ya hecho, analizando los tres parámetros nombrados anteriormente. De esta manera, obteníamos la frecuencia predominante en intensidad y tuvimos que ajustar el rango donde se movían los valores de intensidad total de los silbidos y los valores de duración del silbido. Los silbidos tienen menor intensidad total que una palmada, pero la frecuencia con más intensidad solía ser más elevada. La duración del silbido tuvimos que ajustarla para que fuese mayor que lo que duraba una palmada, ya que al realizar pruebas con silbidos de mucha intensidad se solapaba con el otro reconocedor. En la siguiente figura se muestra las frecuencias generadas por un silbido.

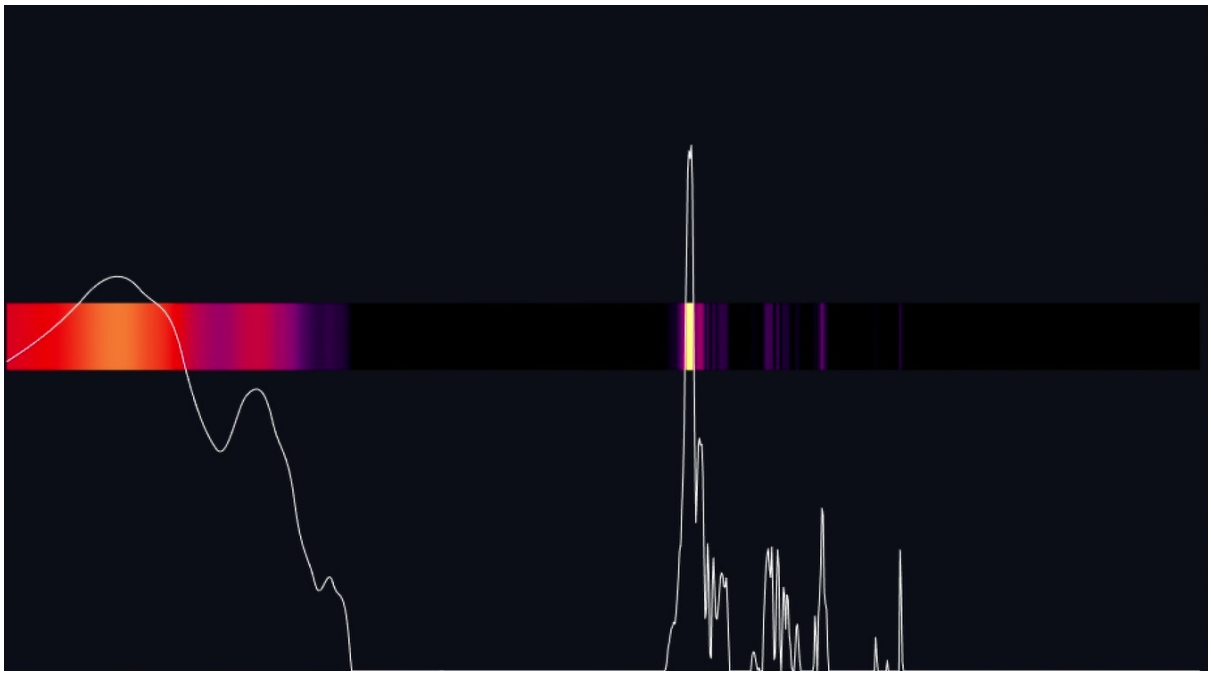

Figura 3.2: Frecuencias de un silbido

Una vez tuvimos los reconocedores pensamos en cómo íbamos a generar los eventos. Desarrollamos un pequeño reconocedor de combos en el que se permitía grabar combos y su posterior reconocimiento. La generación de eventos no estaba integrada en ningún sistema de input por lo que las pruebas se reducían a mensajes por consola. Gracias al desarrollo de este sistema pudimos realizar las suficientes pruebas, en diferentes entornos, para recoger datos de fidelidad de los reconocedores. Descubrimos que el reconocimiento difería mucho al cambiar de micrófono o al cambiar de localización, debido al eco. Debido a esto, tuvimos que replantear el algoritmo de reconocimiento.

Realizamos una refactorización del reconocedor de silbidos, empezando por el principio y comparando las muestras de silbidos de diferentes frecuencias para encontrar patrones comunes. También analizamos las muestras de palmas y chasquidos, ya que necesitábamos identificar los patrones en los que se diferenciaban. Junto con el análisis de los parámetros, creamos el sistema de puntuación de identificación, en el que la suma de todos los parámetros analizados se encuentra entre el intervalo 0 y 1. Este mismo proceso lo realizamos para el reconocedor de palmas y chasquidos de dedos. Tras la refactorización, se experimentó un incremento considerable en el reconocimiento exitoso y un decremento en los falsos positivos.

Todo este trabajo se realizó sobre el proyecto LASP, nombrado anteriormente, que no solo aporta acceso al audio, si no que aporta muchas otras funcionalidades que para este proyecto no son relevantes. Debido a esto, y aprovechando la refactorización del sistema de reconocimiento, decidimos abordar la refactorización el acceso al audio. Eliminamos casi todo el proyecto LASP, dejando solo el procesamiento que realiza al audio sin procesar, y el *wrapper* en C# de la librería *Libsoundio*, que es la que realmente proporciona el acceso al audio.

3.1.4. Input en Unity y generación de eventos

Input System estandariza la forma en la que se implementan los controles. Incorpora una capa de abstracción que sirve para separar el significado lógico de una entrada de los dispositivos físicos que generan dicha entrada. Con ello, permite acceder a los valores de *input* asignados a una acción del juego, sin importar de qué dispositivo provenga la señal [18].

Esta capa de abstracción tiene una representación gráfica en el editor, que permite asignar los controles de los dispositivos a las diferentes acciones que tiene el juego. Todo comienza con la creación de un fichero de tipo *Input Action Asset*. Este archivo tiene la estructura de un JSON y define el objeto sobre el que se va a trabajar. A continuación, se muestra una imagen de ejemplo de un objeto de tipo *Input Action Asset* en el editor:

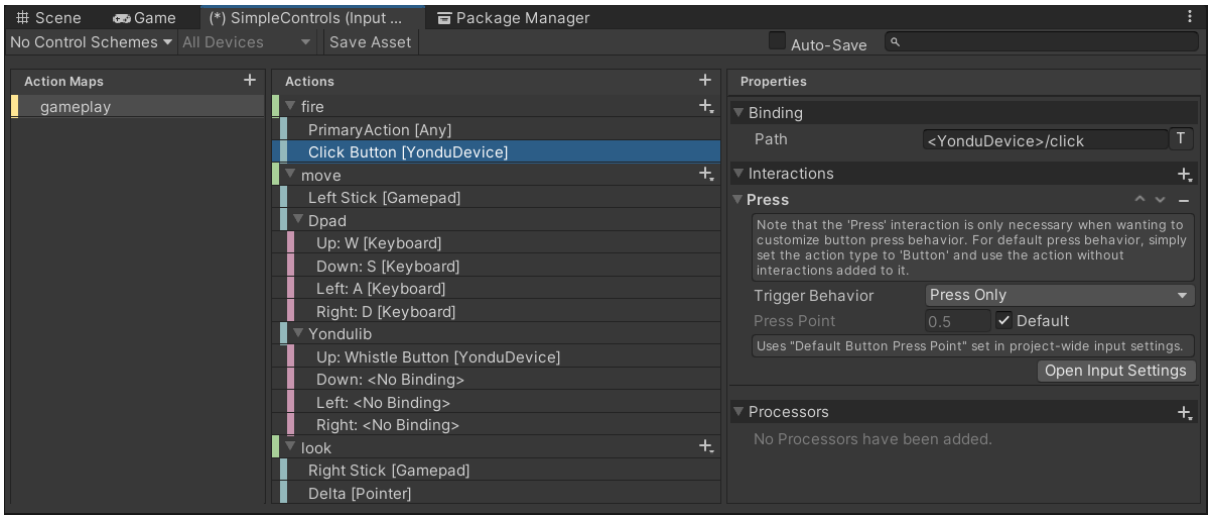

Figura 3.3: *Input Action Asset* con un *Action Map* configurado

Como se puede observar en la figura, hay tres columnas: *Action Maps, Actions* y *Propierties*. Cada *Action Asset* puede tener varios *Action Maps*, que son conjuntos de *Actions*. Los *Action Maps* se pueden usar, por ejemplo, para separar el input relacionado con la interfaz del *input* relacionado con el *gameplay*. De esta forma, se puede dar que el botón de aceptar un evento en un menú del juego pueda ser el mismo que el del salto del personaje y cada uno responde solo en el contexto que corresponde.

En la columna de *Actions* podemos ver todas las acciones definidas. No son botones o teclas, son *bindings*, como saltar, moverse o disparar. Existen tres tipos de acciones: *Value, Button* y *Pass Through*. Sus principales diferencias son en cuanto a como reaccionan ante la llamada de eventos y qué eventos se activan.

Dentro de una acción hay que añadir un *binding*. Por ejemplo, para el típico movimiento de personaje que usa las teclas WASD debemos añadir un *2D Vector Composite*, que son 4 nuevos *bindings* relacionados. Una vez añadidos los *bindings* hay que asignar las teclas o el *Input Path* que queramos relacionar con esta acción. Por ejemplo, puede ser *"<Keyboard>/w"* para referirnos a la tecla W del teclado. Es importante ver aquí que el *path* se refiere a un *Device* como *"Keyboard"* en el ejemplo. En nuestro caso, el dispositivo se llama *YonduDevice* y los controles disponibles son *Click* y *Whistle*.

Una vez definido el fichero *Input Action Asset*, la opción más sencilla es utilizar un componente llamado *Player Input*, que proporciona *Input System*, y asignar el fichero creado a dicho componente. Si queremos tener más control sobre el procesamiento de la entrada, *Unity* nos brinda la oportunidad de generar a partir del fichero *Input Action Asset* un fichero en C#, que contiene todos los datos y permite acceder a través de código a ellos.

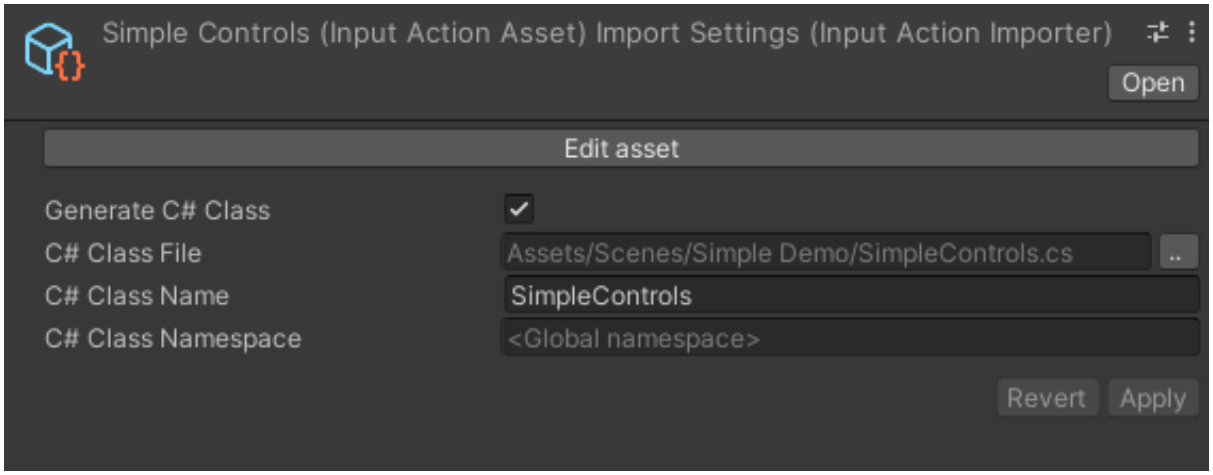

Figura 3.4: Generación automática del fichero C# a partir de un *Input Action Asset*

Esta clase permite simplificar enormemente el acceso a los datos de los dispositivos conectados, debido a todas las funciones nombradas anteriormente que proporciona *Input System*. Por ejemplo, el siguiente código es perteneciente al sistema de input antiguo y accede a los valores de dos dispositivos distintos para realizar la acción *look*:

```
var look = new Vector2();

var gamepad = Gamepad.current;
if (gamepad != null)
    look = gamepad.rightStick.ReadValue();

var mouse = Mouse.current;
if (mouse != null)
    look = mouse.delta.ReadValue();
```

Se puede observar que accede directamente a los dispositivos para obtener el valor. Esto hace que para la obtención del valor haya que comprobar todos los dispositivos para los que queremos que sea compatible el videojuego. Con el nuevo *Input System*, este mismo acceso quedaría de la siguiente forma:

```
myControls.gameplay.look.performed +=
        context => look = context.ReadValue<Vector2>();
```

Sin duda, es una mejora sustancial, ya que independientemente del número de dispositivos utilizados, el código es inmutable y no cambiará. De esta forma, cambios en el funcionamiento de los dispositivos no afecta al código del juego, si no que bastaría con

modificar en el editor gráfico las propiedades afectadas por los cambios realizados en el dispositivo.

De la misma forma que se pueden leer los datos de los controles de un dispositivo, también se pueden generar esos mismos datos mediante eventos por código. *Input System* está dirigido por eventos y permite crear tanto eventos generales de un dispositivo como eventos de controles específicos. La generación de estos eventos es útil para desarrolladores que quieren crear dispositivos nuevos o que pretenden modificar el funcionamiento de dispositivos ya existentes[19].

3.1.5. Paquetes en Unity

A día de hoy la práctica más extendida gracias a las funcionalidades integradas que proporciona *Unity* es el uso de *Custom Packages*. Para crear un paquete hay que seguir una convención de diseño que propone Unity, en la que se detalla la estructura de ficheros que hay que seguir.[20]

```
<root>
  ├── package.json
  ├── README.md
  ├── CHANGELOG.md
  ├── LICENSE.md
  ├── Third Party Notices.md
  ├── Editor
  │   ├── [company-name].[package-name].Editor.asmdef
  │   └── EditorExample.cs
  ├── Runtime
  │   ├── [company-name].[package-name].asmdef
  │   └── RuntimeExample.cs
  ├── Tests
  │   ├── Editor
  │   │   ├── [company-name].[package-name].Editor.Tests.asmdef
  │   │   └── EditorExampleTest.cs
  │   └── Runtime
  │       ├── [company-name].[package-name].Tests.asmdef
  │       └── RuntimeExampleTest.cs
  ├── Samples~
  │       ├── SampleFolder1
  │       ├── SampleFolder2
  │       └── ...
  └── Documentation~
      └── [package-name].md
```

Figura 3.5: Estructura típica de un paquete de *Unity* usado para añadir funcionalidad adicional a un juego. Puede contener tanto código que modifica el comportamiento del editor (carpeta "editor") como el juego en sí mismo (carpeta *"runtime"*)
[10]

Hay que crear los ficheros *.asmdef*, *Assembly Definition*. Estos archivos gestionan las dependencias y con ello mejora los tiempos de compilación y facilita la ampliación del proyecto.

[10]Estructura del paquete `https://docs.unity3d.com/Manual/cus-layout.html`

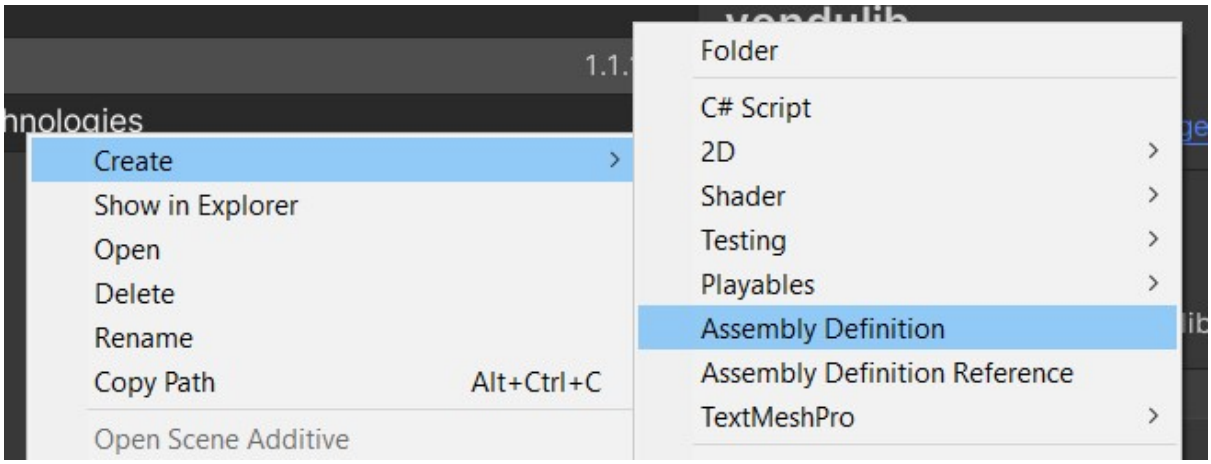

Figura 3.6: Creación del fichero *Assembly Definition*

Otro de los ficheros imprescindibles es el *package.json*, que es un manifiesto de todo el paquete.

```
{
  "name": "com.companyname.packagename",
  "version": "1.2.3",
  "displayName": "Package Example",
  "description": "This is an example package",
  "unity": "2019.1",
  "unityRelease": "0b5",
  "dependencies": {
    "com.unity.some-package": "1.0.0"
  },
  "keywords": [ "keyword1", "keyword2" ],
  "author": {
    "name": "My Company",
    "email": "hello@example.com",
    "url": "https://www.mycompany.com"
  }
}
```

El *workflow* que hemos seguido en el desarrollo para *Packages* con esta estructura ha sido trabajar en un proyecto el desarrollo del código del paquete y crear, en un directorio a parte, la estructura recomendada. En otro proyecto importar esa estructura desde el *Package Manager* para realizar las pruebas pertinentes. Cuando los cambios sean estables se puede utilizar un repositorio de *GitHub* subiendo únicamente el directorio con la estructura comentada anteriormente para que la importación desde *Git* integrada en el *Package Manager* de *Unity* lo reconozca.

En cuanto al punto de vista del usuario que se baja el paquete desde *GitHub*, solo se requiere añadir previamente en el *manifest.json* de la carpeta *Packages* los *Scoped Registries* si son necesarios, que se mencionan en la guía de instalación de cada paquete.

3.2. Implementación

La implementación de YONDULIB se puede dividir en dos grandes sistemas: el sistema de reconocimiento de sonidos y el sistema de *input* y generación de eventos. El primero abarca el acceso al audio (3.2.3), su procesamiento (3.2.4) y su posterior análisis para la generación del evento (3.2.5). El segundo utiliza la librería *Input System* para gestionar los dispositivos y los eventos (3.2.2).

A continuación, resumimos brevemente las decisiones de diseño tomadas antes de profundizar en las diferentes partes en las que se dividen estos sistemas y cómo fue su implementación en el proyecto.

3.2.1. Decisiones de diseño tomadas

El sistema de reconocimiento está compuesto por tres fases: acceso al sonido, procesamiento del audio y análisis del audio. Como explicamos en el punto anterior, para acceder al audio se barajaron múltiples opciones válidas optando finalmente por la librería *Libsoundio* y un *wrapper* para C# y *Unity*. El procesamiento del audio es aprovechado de la librería LASP, que es de donde obtuvimos el *wrapper*. Por último, el análisis del audio fue desarrollado por nosotros, de acuerdo a los controles que queríamos obtener del sonido detectado.

El sistema de generación de eventos está estrechamente relacionado con el motor de desarrollo. En *Unity* existen dos opciones para generar eventos de *input*: *Input Manager* y *Input System*. Finalmente nos decantamos por *Input System* debido a la versatilidad y control sobre los eventos que ofrece.

3.2.2. Estructura general de clases

El proyecto se divide en dos módulos principales, uno encargado del reconocimiento de sonidos y la generación de sus respectivos eventos, y otro encargado de gestionar esos eventos generados y enlazarlos con las acciones correspondientes del juego.

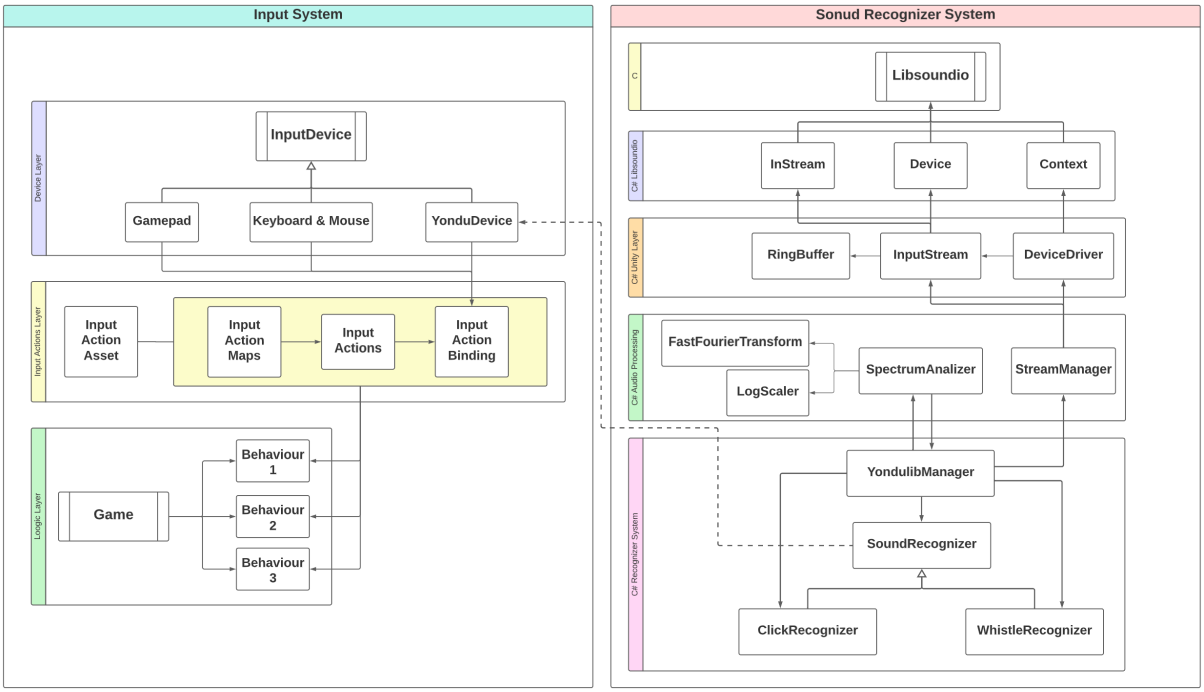

Figura 3.7: Estructura general de YONDULIB

Reconocimiento de sonidos

Para poder aplicar los algoritmos de reconocimiento de sonidos primero tenemos que poder acceder al sonido. El *wrapper* creado por *Keijiro Takahashi* de la librería *Libsoundio* creada por *Andrew Kelley* nos proporciona el acceso al audio con baja latencia.

La estructura principal se compone de tres clases: *InStream*, que envuelve el flujo de datos, *Device*, que representa el dispositivo que proporciona esos datos, y *Context*, que recoge todos los datos necesarios para establecer el contexto necesario para acceder a las funciones de bajo nivel que gestionan los dispositivos.

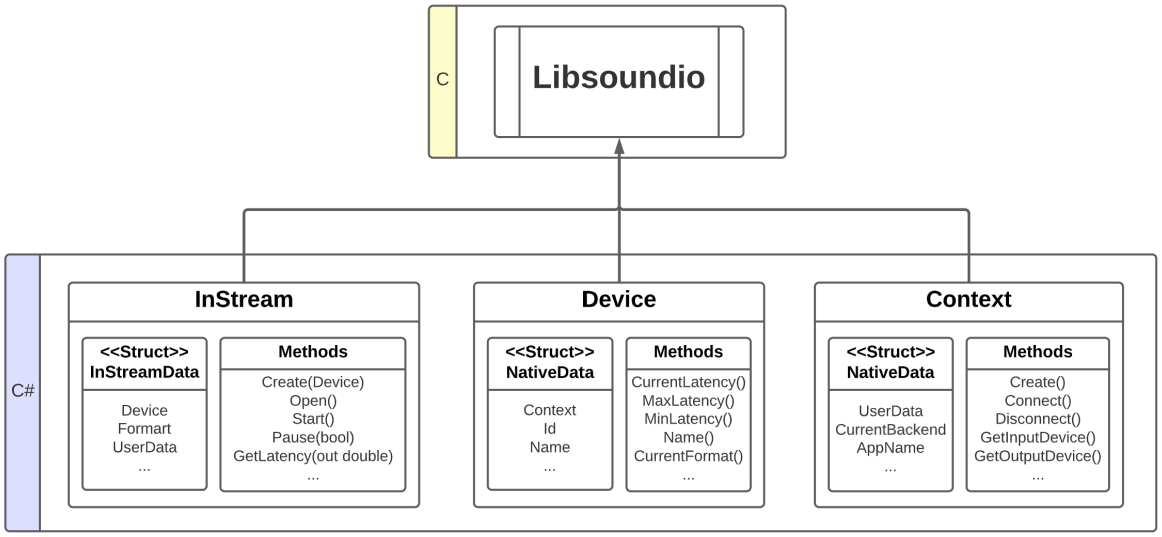

Figura 3.8: *Wrapper*

Para hacer uso del *wrapper* necesitamos una capa de abstracción que nos permita acceder a los micrófonos, y a los datos que capta, de una forma organizada y segura. Las clases que componen esta capa son: *DeviceDriver*, que gestiona el acceso al contexto y los dispositivos, así como el acceso al flujo de datos, *InputStream*, que encapsula el modo de mostrar los datos al exterior, y *RingBuffer*, que representa el patrón del *buffer* circular utilizado para mostrar los datos en *InputStream*. La clase *StreamManager* es la encargada de utilizar esta capa para proporcionar las opciones de gestión al resto del proyecto.

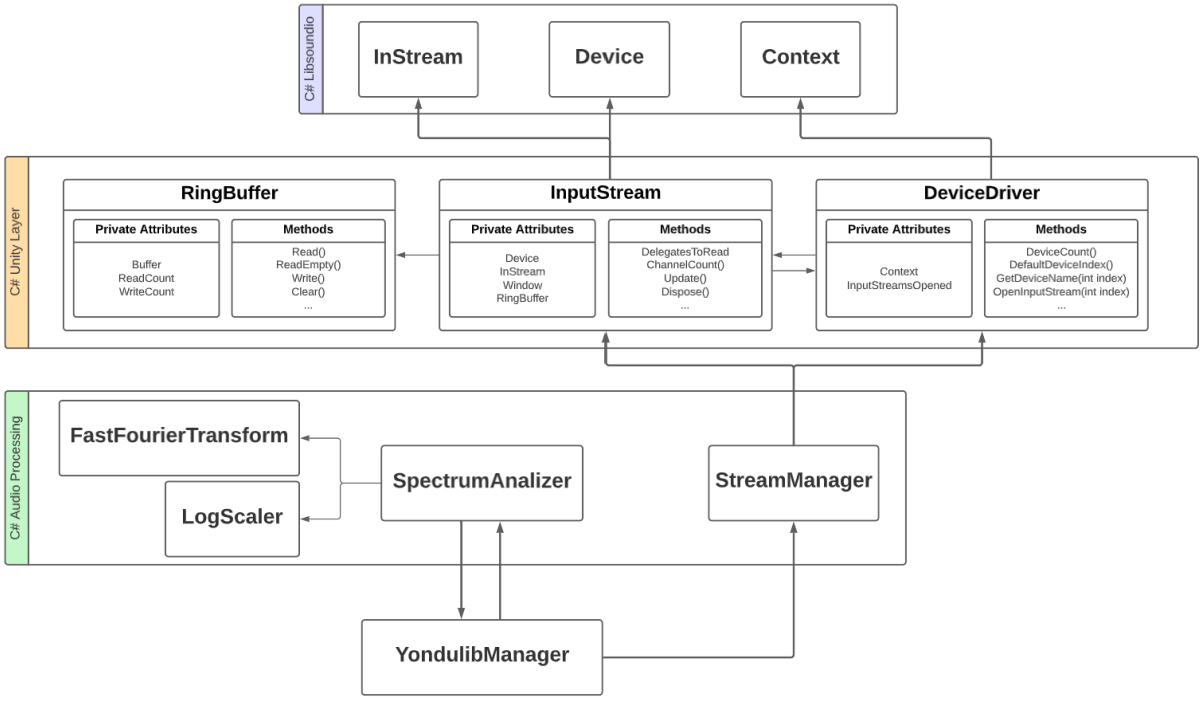

Figura 3.9: Capa de abstracción

Una vez tenemos acceso al flujo de datos captado por el micrófono, se necesitan procesar los datos para poder analizarlos en el formato correcto. *SpectrumAnalyzer* es la clase encargada de realizar este procesado, que consiste en aplicar la transformación rápida de Fourier y un escalado logarítmico.

Una vez procesado los datos, se pueden aplicar los algoritmos de reconocimiento de sonidos. Estos algoritmos se basan en una estructura de herencia simple, donde la clase base es *SoundRecognizer* y de la que heredan *ClickRecognizer*, encargada de reconocer los golpes, chasquidos y palmas, y *WhistleRecognizer*, encargada de reconocer los silbidos.

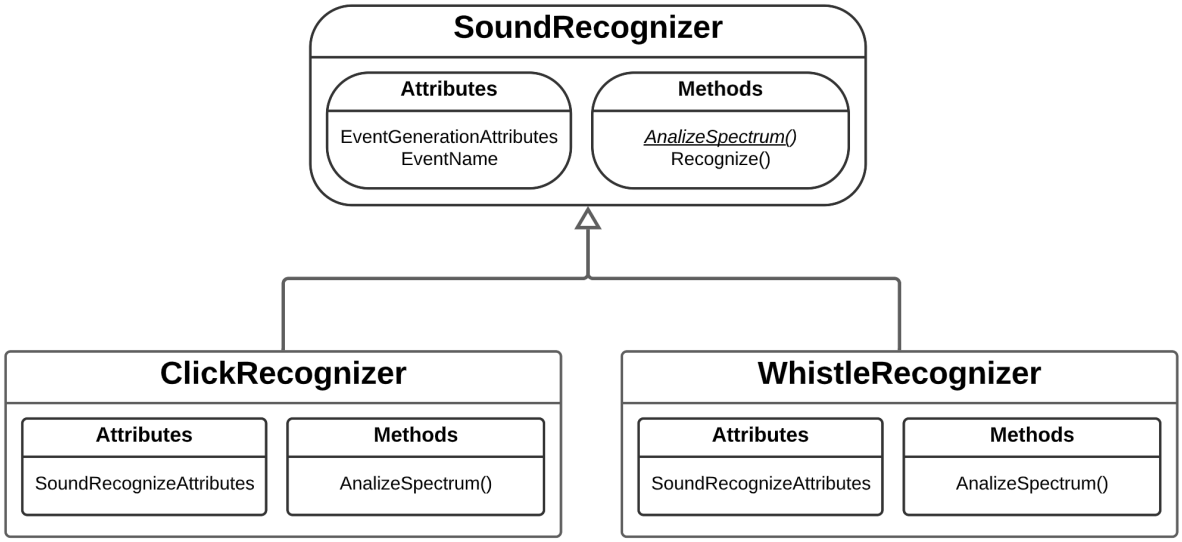

Figura 3.10: Estructura de clases de los reconocedores

La clase encargada de unir los datos procesados con los algoritmos de reconocimiento y de gestionar la parte visual de la configuración es *YondulibManager*. La estructura de clases completa del módulo de reconocimiento de sonidos seria la mostrada en la siguiente imagen.

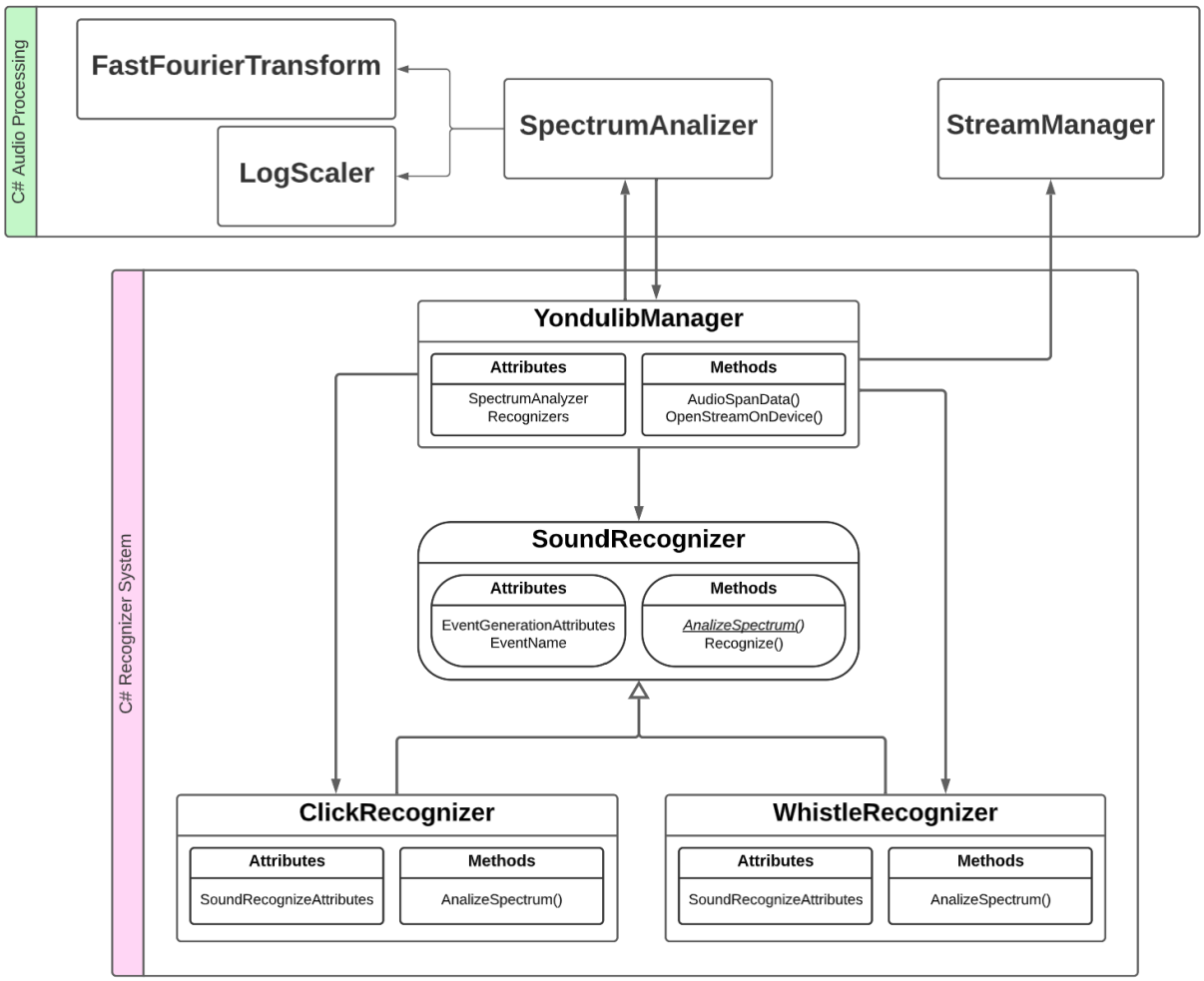

Figura 3.11: Estructura de clases completa del módulo de reconocimiento de sonidos

Input System

Es un sistema de entrada nuevo de *Unity* que de momento convive con el anterior, y que trae consigo alguna mejoras significativas. La principal novedad es que crea una capa intermedia entre los dispositivos y la lógica del juego, que aporta flexibilidad y modularidad a ambas secciones.

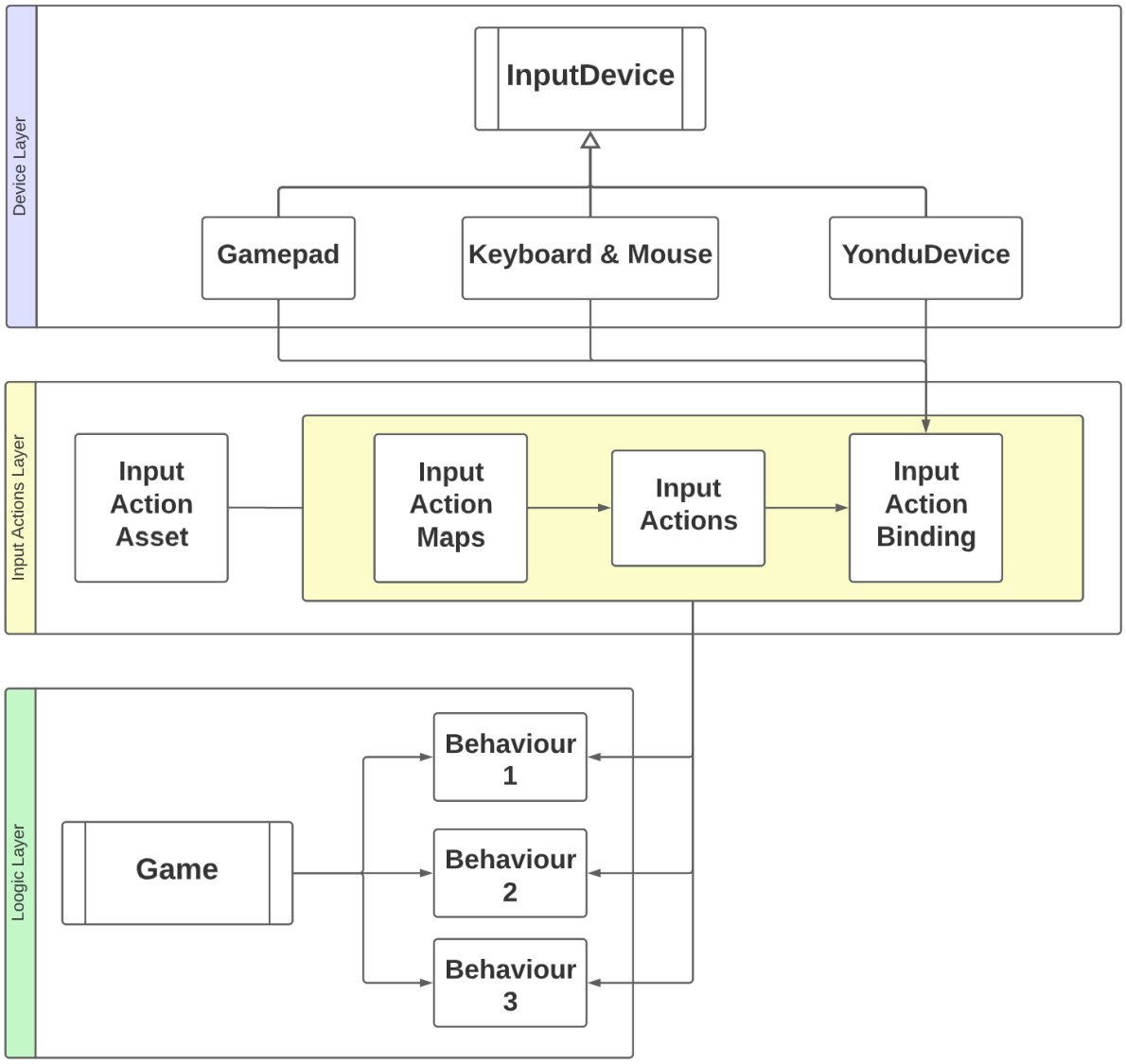

Figura 3.12: Estructura básica de *Input System*

Esta capa intermedia la componen los objetos *Input Action Assets* que son archivos de extensión JSON que contiene una representación de las acciones que se pueden hacer en el juego. *Unity* permite generar una clase en código C# a partir de este JSON, para poder acceder a los valores asignados a cada acción a través del código.

Los *Input Action Asset* se componen de *Input Action Maps* que son mapas que engloban acciones similares según el contexto, por ejemplo, un mapa de *gameplay* contiene las acciones de disparar y moverse, y un mapa de menús contiene las acciones de interacción con la interfaz de usuario. Estas acciones, que se llaman *Input Action*, contiene el elemento *Input Binding* que representa la conexión entre la acción y uno o más controles. De esta forma puede haber más de un control asociado a una misma acción. Esto es útil para establecer una compatibilidad con diferentes dispositivos de una forma rápida y sencilla.

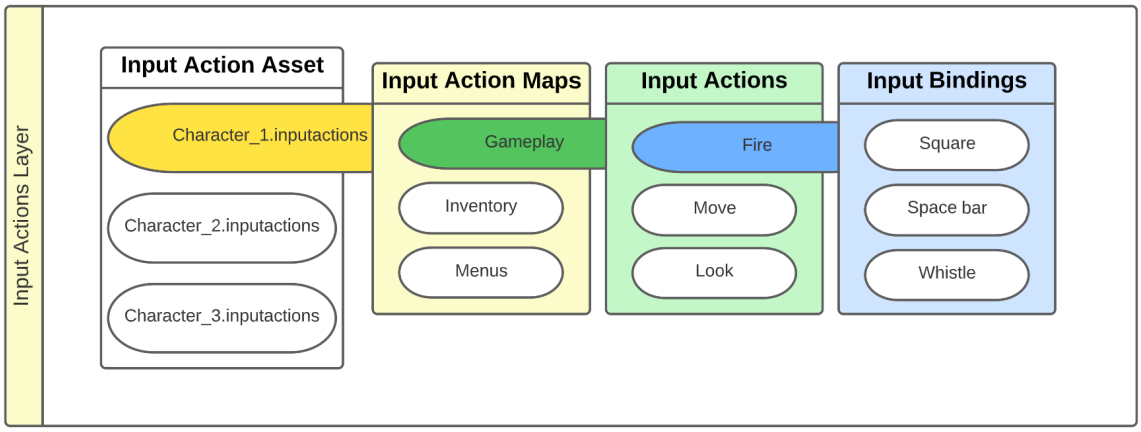

Figura 3.13: Esta capa permite agrupar las acciones con un mismo contexto (en verde) en mapas de acciones (en amarillo) para que se ejecuten al detectar las entradas asociadas (en azul)

InputSystem permite incluir dispositivos nuevos. De esta manera podemos utilizar dispositivos poco convencionales y trabajar directamente con ellos en el editor. Cada dispositivo tiene su estado que define los eventos que se pueden generar con dicho dispositivo. Todos los dispositivos heredan, directa o indirectamente, de la clase *InputDevice*. De forma predeterminada vienen creados los dispositivos más comunes como el teclado y ratón, y multitud de *gamepads*. La flexibilidad que ofrece *Input System* es enorme ya que permite modificar y extender cualquier dispositivo [21][22].

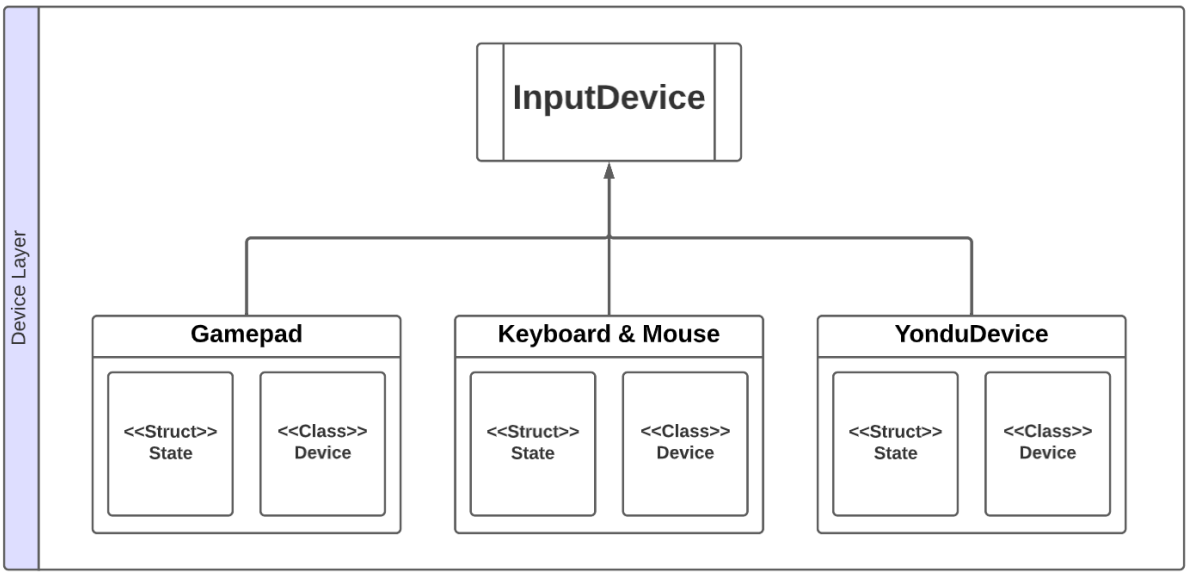

Figura 3.14: Estructura de la capa de dispositivo. *YonduDevice* se comporta como un dispositivo de entrada más

3.2.3. Acceso al audio

Para acceder a las funciones de bajo nivel que gestionan el acceso al sonido y sus dispositivos utilizamos la librería *Libsoundio*. Es una librería multiplataforma creada por *Andrew Kelley* en el lenguaje de programación C que permite gestionar la entrada y salida de audio, adecuada para su utilización en software de tiempo real. [11] *Libsoundio* es compatible en los principales sistemas operativos: *Linux, Mac OS y Windows*, soportando las librerías de audio de bajo nivel: JACK, *PulseAudio*, ALSA, *CoreAudio* y WASAPI.

Debido a que en *Unity* se programa en C#, aprovechamos el trabajo realizado por *Keijiro Takahashi* en su proyecto de procesamiento de señal de audio de baja latencia (LASP)[23] en el que utiliza un *wrapper* de *Libsoundio* creado por él mismo para poder utilizarla en este lenguaje. [24]

En YONDULIB, el acceso al audio se realiza a través de la clase *StreamManager* que accede automáticamente al micrófono por defecto, aunque también brinda la posibilidad de elegir el micrófono, a través del *prefab DeviceSelector*. Internamente se apoya en un *driver* simple que conecta con el wrapper anteriormente nombrado. Este *driver* se compone de tres clases: *DeviceDriver, InputStream y RingBuffer*.

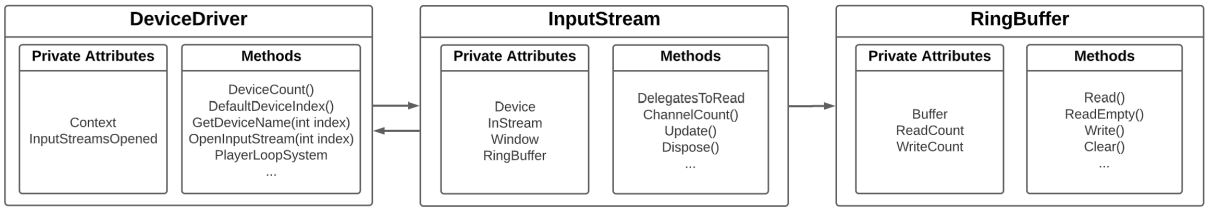

Figura 3.15: Estructura interna del *driver*

DeviceDriver

Esta clase tiene una interfaz publica que brinda información sobre los dispositivos conectados y permite abrir un flujo de entrada (*InputStream*) para acceder a los datos que recoge el micrófono.

Para poder acceder al audio y los dispositivos es imprescindible saber en qué sistema operativo estamos y qué *backend* se va a utilizar. Para ello, es preciso recoger la información necesaria que establezca un contexto que defina las características del entorno.

El *wrapper* de *Keijiro* contiene la clase *Context* que proporciona toda la información necesaria para inicializar exitosamente el contexto y que conecta directamente con *Libsoundio*. Contiene una interfaz pública que permite acceder a los dispositivos por índice, tanto de entrada como de salida, y al dispositivo por defecto que se utiliza en el ordenador.

La clase *DeviceDriver* contiene una instancia estática de la clase *Context*, que utiliza para exponer una interfaz que permite acceder a los índices de los dispositivos de entrada y a sus nombres. Evita exponer el dispositivo entero de manera pública porque la gestión del mismo se hace de manera interna. Contiene un método que devuelve un flujo de entrada abierto a partir del índice del dispositivo.

[11] Libsoundio `https://github.com/andrewrk/libsoundio`

DeviceDriver se encarga, internamente, de gestionar todos los flujos abiertos, actualizándolos a través de un método *Update()*. Aunque esta clase se actualiza en cada vuelta de bucle, no es una implementación *Monobehaviour*. *Unity* brinda la oportunidad de crear clases que se actualicen sin necesidad de implementar dicha interfaz, a través de *PlayerLoopSystem*. *PlayerLoopSystem* es una estructura que representa un subsistema de actualización en *Unity*. Existen tanto subsistemas nativos integrados en *Unity* como subsistemas personalizados creados por el propio programador. La clase *PlayerLoop* representa el sistema que engloba a todos los subsistemas permitiendo acceder al orden de actualización y modificarlo.

DeviceDriver aprovecha esta funcionalidad para actualizar los flujos abiertos, creando un subsistema propio asignando el método *Update()* como punto de actualización.

InputStream

Es la clase que representa el flujo de entrada. Tiene una interfaz pública que permite acceder a datos importantes como el número de canales, la frecuencia de muestreo o la latencia, así como a los datos leídos por el dispositivo. También contiene un método público *Update()* que es el encargado de que los datos se lean correctamente.

Para crear un *InputStream* es necesario tener acceso al dispositivo al que va asociado el flujo. Por lo que la única clase capaz de crear uno es *DeviceDriver*, ya que, como explicamos con anterioridad, internamente es la que contiene el *Context*, que a su vez es el encargado de acceder a los dispositivos.

InputStream contiene un objeto de tipo *InStream*, que también pertenece al *wrapper*, y que es el encargado de gestionar todas las acciones relacionadas con la lectura de datos. La estructura *InStreamData* es la encargada de contener todos los datos necesarios para realizar correctamente dicha lectura, dos métodos, que pertenecen directamente a la librería *libsoundio* y que se llaman desde el contexto del *callback ReadCallback*:

- *BeginRead*: función que se llama cuando está todo listo para leer los datos del *buffer* del dispositivo. Recibe como parámetros el flujo de entrada del que se va a leer, las direcciones de memoria de donde puede leer datos y la cantidad de *frames* que se van a leer.
- *EndRead*: función que se llama cuando la lectura ha resultado exitosa y ha finalizado. Recibe como parámetro el flujo de entrada y elimina el número de *frames* indicados al llamar a *BeginRead*.

y tres *callbacks*:

- *ReadCallBack*: es donde se realiza la acción de lectura de datos. Recibe un parámetro de tipo *InStreamData* que se pasa por referencia y dos parámetros más que representan el mínimo y el máximo de *frames* que se pueden leer. Dentro de esta función primero se llama a *BeginRead*, despues se leen los datos y por último se llama a *EndRead*.
- *OverFlowCallback*: esta función lanza un mensaje de advertencia en el editor de que el *buffer* del dispositivo está lleno.
- *ErrorCallback*: esta función se llama cuando ocurre un error irrecuperable.

A través de una interfaz pública en *InStream* podemos acceder a los datos y asignar dichos *callbacks*. Esta clase también contiene los métodos necesarios para crear, abrir, iniciar, pausar y acceder a la latencia del flujo de datos.

Finalmente, contiene un *buffer* circular (*RingBuffer*) que se utiliza como herramienta de comunicación entre *libsoundio*, que vuelca los datos leídos, y *InputStream*, donde los datos se guardan en un *array de bytes*. La escritura en el *RingBuffer* es realizada en una función delegada requerida en la clase *InStream* llamada *OnReadInStream()*, y es en el método *Update()* de InputStream donde se realiza la lectura de los datos transferidos al *RingBuffer()*.

RingBuffer

Para leer los datos correctamente *Keijiro* utiliza el patrón de *buffer* circular, que como su nombre indica, es un *array* con una política de reemplazo FIFO (*first in first out*), en el que el primer elemento en entrar al *buffer* es el primero en ser reemplazado. Este patrón tiene la ventaja de que no necesita realizar reservas de memoria, ya que los datos nuevos sobrescriben los más antiguos, lo que evita recolecciones de memoria que impacten en el rendimiento.

Esta estructura sirve como conexión entre *Libsoundio*, que recoge el audio y lo expone a través del método *ReadCallBack* volcando los datos en el *RingBuffer*, y el *driver*, que lee los datos del audio escritos en *RingBuffer*. Como se puede apreciar, este entorno presenta una estructura de productor-consumidor, en el que *RingBuffer* es el medio de comunicación, por lo que contiene una sección crítica que debe ser controlada. Esta sección crítica se encuentra en el método *OnReadInStream* que es la función que se le pasa a *ReadCallback*.

La interfaz pública expone información sobre el estado del *buffer* y los datos leídos y escritos y tres funciones que representan las acciones de lectura y escritura: *Write*, que escribe los datos pasados por parámetro en el *buffer*, *WriteEmpty*, sobre escribe el *buffer* con ceros, y *Read*, que vuelca los datos del *buffer* en una región de memoria pasada por parámetro.

3.2.4. Procesamiento del audio

La clase encargada de gestionar el procesado del audio es *SpectrumAnalyzer* que hereda de *MonoBehaviour*, por lo que se actualiza en cada vuelta de bucle para realizar varios cálculos.

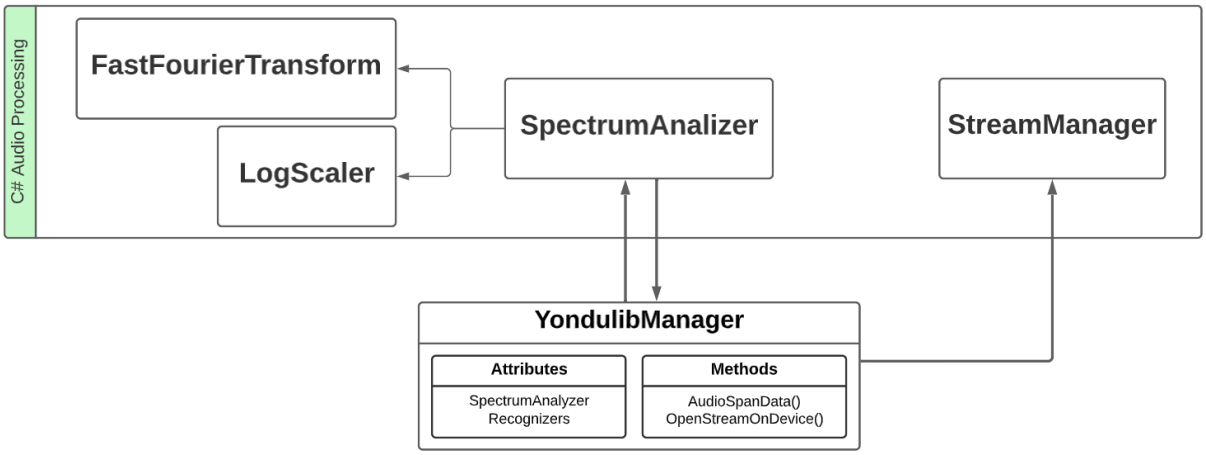

Figura 3.16: Estructura del procesamiento de audio

El acceso a los datos, como explicamos anteriormente, se lo concede la clase *StreamManager*, a través de *YondulibManager*, utilizando una región de memoria no administrada (*NativeArray*) donde se vuelcan los datos. Al realizar este proceso, *StreamManager* modifica el volumen de la muestra multiplicándola por un valor para así conseguir ampliar la intensidad de toda a muestra.

A los datos recogidos se le aplica el algoritmo de la transformación rápida de Fourier (FFT)[12] para cambiar los datos del dominio del tiempo a una representación en el dominio de la frecuencia. Este algoritmo permite calcular la transformada discreta de Fourier con un coste computacional reducido, lo que se traduce en mayor eficiencia y rapidez para poder analizar los datos posteriormente.

Para realizar esta transformación se definen los valores de la ganancia y el rango dinámico. La ganancia se utiliza para ajustar la intensidad de la señal para evitar la distorsión del sonido. El rango dinámico también influye para evitar distorsiones y conseguir una muestra de audio más limpia.

Posteriormente, se realiza un escalado logarítmico de la intensidad ya que la medida estándar, el decibelio (dB), es logarítmica.

3.2.5. Análisis del audio. Fase de reconocimiento

La primera fase, en el análisis del audio procesado, es la fase de reconocimiento del patrón específico. Esta fase es específica al sonido que queremos identificar. En este proyecto hemos realizado dos reconocedores diferentes, uno para palmas, chasquidos y golpes (*ClickRecognizer*), y otro para silbidos y sonidos armónicos (*WhistleRecognizer*).

Para identificar cada tipo de sonido hemos seguido un patrón similar. Nos fijamos en diferentes parámetros y cada uno obtiene una puntuación de reconocimiento. La suma de todas estas puntuaciones es como máximo 1, siento este número el valor de coincidencia máxima y pudiendo afirmar que el sonido encaja al 100 % en un silbido o en un golpe.

[12]Transformación rápida de Fourier `https://en.wikipedia.org/w/index.php?title=Fast_Fourier_transform&oldid=1100383485`

Los parámetros comunes, tanto al reconocimiento de palmas como al de silbidos, son: el número de frecuencias altas, la diferencia entre el valor máximo y el valor mínimo de intensidad y la comparación de los picos máximos. En el reconocedor de silbidos añadimos un parámetro adicional, que es el estudio de la posición de la frecuencia predominante. Los parámetros del reconocedor de chasquidos son tres por lo que cada uno puede conseguir una puntuación máxima de 0.33. Por el contrario, el reconocedor de silbidos tiene cuatro parámetros, por lo que el valor máximo de cada uno de ellos es de 0.25.

Esta estructura es un prototipo experimental de reconocimiento que funciona en un trabajo de estas características. Todos los parámetros, que a continuación se desarrollan, han surgido de pruebas experimentales realizadas por nosotros, que nos han dado buenos resultados en el reconocimiento de los sonidos nombrados. Estas pruebas son realizadas en combinación con el proyecto LASP[23] que nos proporciona texturas y animaciones interesantes.

Número de frecuencias altas

Número de picos de intensidad que contiene el *array* de datos. Cuantas más frecuencias con intensidades altas contenga más probabilidades hay de que sea un golpe/chasquido. Para ello recorremos todo el *array* de datos comparando la posición actual con la anterior y la posterior para saber cuándo deja de crecer la intensidad y empieza a decrecer y contar el número de picos de intensidad que tiene la muestra.

Una vez tenemos el parámetro calculado, lo analizamos en función de las características del sonido que queremos identificar. En golpes y palmas, la cantidad de picos de intensidad es elevada, por lo que hemos definido que, si la cantidad de picos de intensidad es igual o mayor que un séptimo del total de frecuencias, entonces reconocemos al 100 %.

Por el contrario, en un silbido el número de picos de intensidad es bajo y predomina uno sobre el resto, por lo que cuanto menos picos haya mayor será la puntuación obtenida. Si se detectan picos de frecuencia superiores a un sexto del del total de frecuencias, el reconocimiento de silbidos recibirá una puntuación de 0.

El valor de un séptimo y de un sexto nombrados son valores experimentales que han sido calculados y basados en distintas pruebas realizadas con diferentes formas de silbar y de dar palmas y chasquidos.

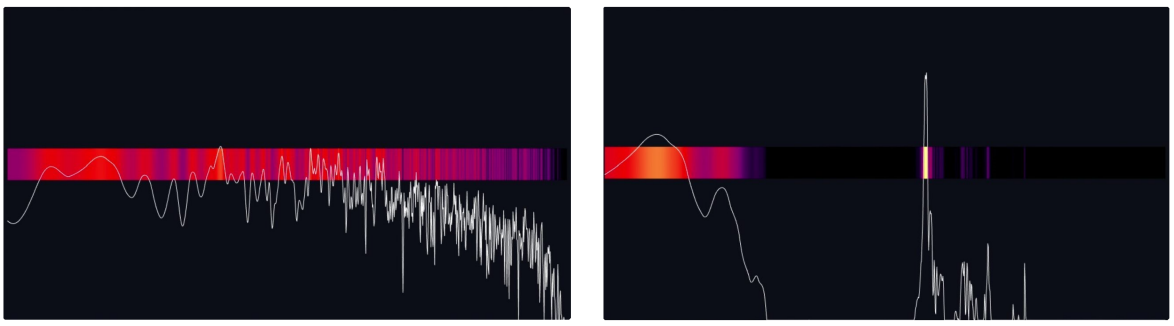

Figura 3.17: Picos de intensidad de una palmada (izquierda) vs de un silbido (derecha)

Diferencia de intensidad

La diferencia de intensidad es la diferencia entre el valor máximo de frecuencia y el mínimo. Hablamos de diferencia de intensidad entre frecuencias cercanas en el espacio, ya que en todas las muestras suele tener frecuencias con valor nulo. Lo que queremos detectar en este análisis es la amplitud de los picos de frecuencia en relación con las frecuencias cercanas. Si la amplitud es grande hay más probabilidades de que sea un silbido. Por lo contrario, si la amplitud es baja lo más probable es que haya sido un golpe.

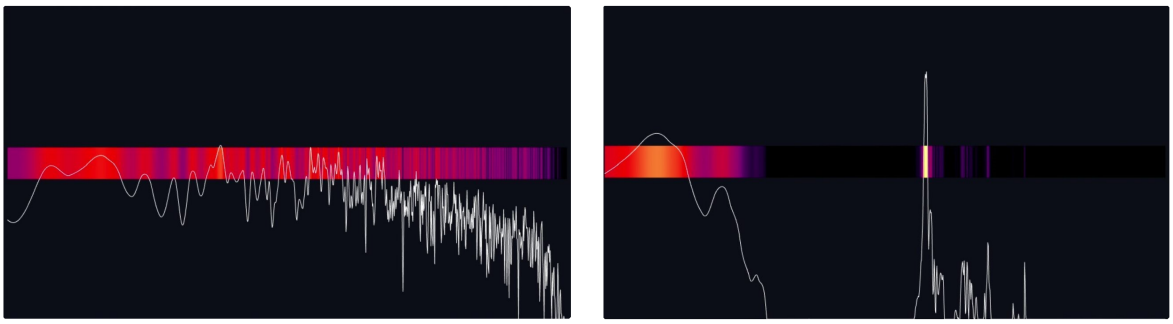

Figura 3.18: Amplitud de los picos en una palmada (izquierda) vs amplitud de los picos en un silbido (derecha)

Como vemos en la imagen de la derecha, la diferencia entre el valor máximo y el mínimo es muy grande, lo que nos sugiere que es un silbido. Por el contrario, en la imagen de la izquierda la diferencia es bastante menor, lo que indica que es una palmada.

Para poder calcular este parámetro nos hemos apoyado en el patrón de la ventana deslizante. Este patrón consiste en formar una ventana sobre una parte de los datos y desplazarla sobre el *array* para capturar diferentes partes de ellos.

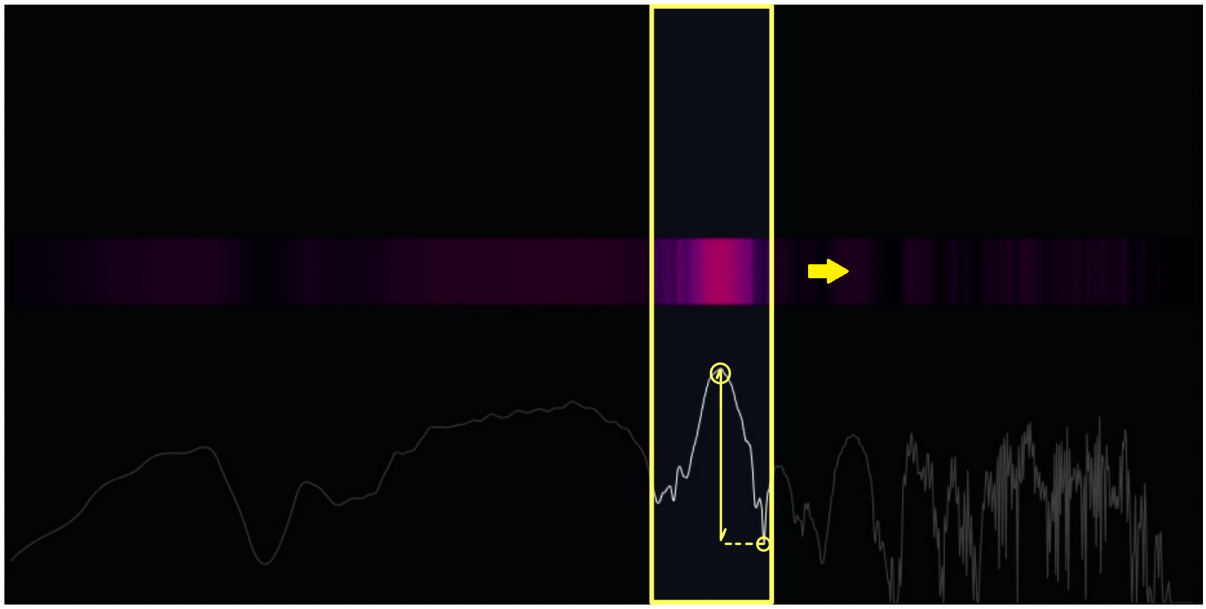

Figura 3.19: Ventana deslizante sobre la imagen tomada de un chasquido en un instante que sirve para calcular la diferencia de intensidad

La ventana tiene una extensión del 10 % de la dimensión del *array* y lo recorre calculando la diferencia de intensidad en cada posición. Este porcentaje lo consideramos bueno después de la realización de pruebas con diferentes tamaños de ventana.

El cálculo de este parámetro lo hemos implementado utilizando dos colas de prioridad, una de máximos y otra de mínimos. En estas colas se encuentran los valores que están en la ventana deslizante en ese instante, por lo que para calcular la diferencia de intensidad solo hay que restar el primer valor de la cola de máximos con el primer valor de la cola de mínimos. Este valor calculado se compara con el valor máximo de las iteraciones anteriores. Si es mayor se actualiza y se guarda para futuras comparaciones, y si es menor se desecha.

Una vez obtenido el valor de amplitud obtendremos un valor entre 0 y 1. En el reconocedor de silbidos premiamos que la amplitud sea elevada, por lo que si el valor es igual o superior a 0.8 recibirá la máxima puntuación. Por el contrario, en el reconocedor de palmas y chasquidos, esta situación se produce si el valor es inferior a 0.4. Una vez más, estos valores son experimentales y, en el marco de pruebas realizado, han sido reconocidos como buenos umbrales de reconocimiento.

Comparación de picos de intensidad máximos

Este parámetro mide la diferencia de intensidad entre los picos de frecuencia máximos. Para ello, aprovechamos la ventana deslizante del parámetro anterior para ordenar todas las frecuencias de mayor a menor intensidad, a través de una cola de prioridad descendente. Esta cola de prioridad maneja objetos de la estructura *Note* que se compone de dos valores, intensidad y frecuencia.

Una vez ordenados los datos de mayor a menor intensidad se calcula la diferencia de intensidad máxima entre ellos. Compara la primera intensidad de la cola con la siguiente,

y después esta segunda con la tercera, y así sucesivamente, restando y calculando la diferencia de intensidad entre los dos valores. Este proceso se repite con todas las intensidades mayores que 0.

Si las frecuencias de los valores comparados son frecuencias contiguas entonces no realizamos la comparación porque asumimos que se trata del mismo pico de intensidad. La distancia establecida para esta criba es el doble del tamaño de la ventana deslizante, es decir, del 20 % de la dimensión del *array*. Tras realizar varias pruebas con diferentes longitudes, este porcentaje es el tamaño que mejores resultados nos ha dado en las pruebas experimentales.

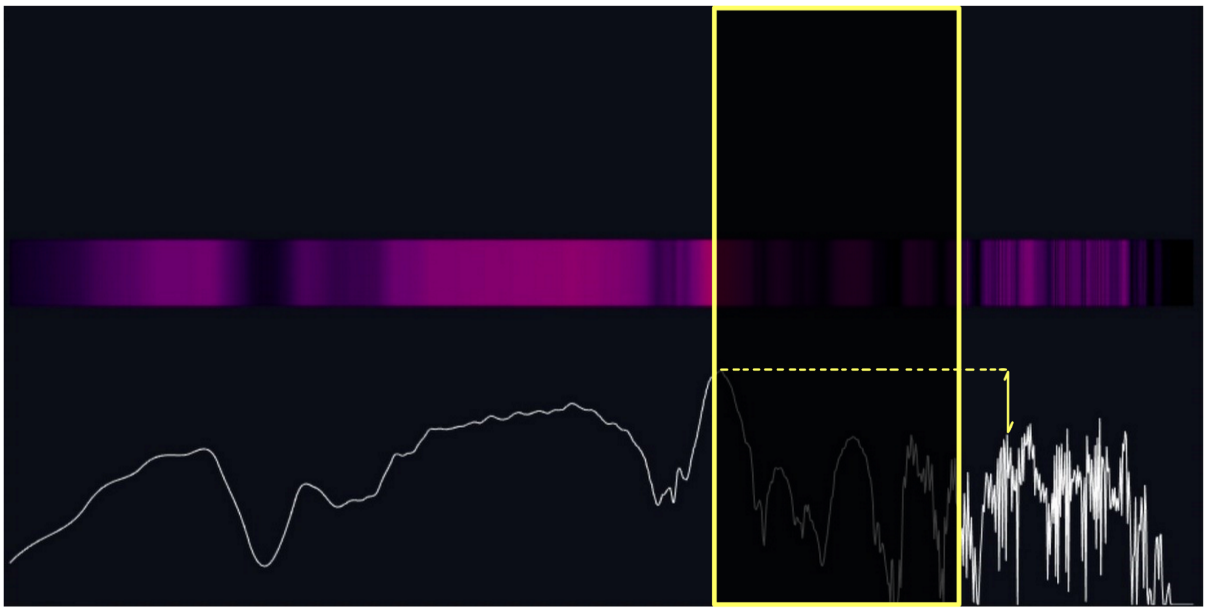

Figura 3.20: Comparación de los picos de intensidad máximos con una distancia mínima del 20 % del tamaño total del *array*

Cuanta más diferencia haya entre los picos máximos registrados más probabilidades hay de que sea un silbido, ya que se suele registrar una mayor intensidad en frecuencias aisladas. Por el contrario, a menor diferencia entre los picos máximos mayor probabilidad de que el sonido registrado sea un golpe. Esto es debido a que en este tipo de sonidos se registran intensidades parecidas en la mayoría de las frecuencias.

Posición de la frecuencia predominante

Este parámetro solo se calcula en el reconocedor de silbidos debido a que en la mecánica de chasquidos no es relevante la frecuencia predominante. La posición de la frecuencia de un silbido no llega a notas ni muy graves ni muy agudas. Según las pruebas realizadas sobre distintas formas de silbar, la frecuencia máxima se queda en la zona central del espectro. Los valores recogidos revelan un intervalo aproximado que está entre el primer quinto y el tercer quinto del *array*. Hemos recogido silbidos fuera de este intervalo, pero son poco frecuentes. La frecuencia con la máxima intensidad se obtiene en la ventana deslizante del segundo parámetro explicado en esta sección (3.2.5), aprovechando que itera sobre cada uno de los elementos del *array*.

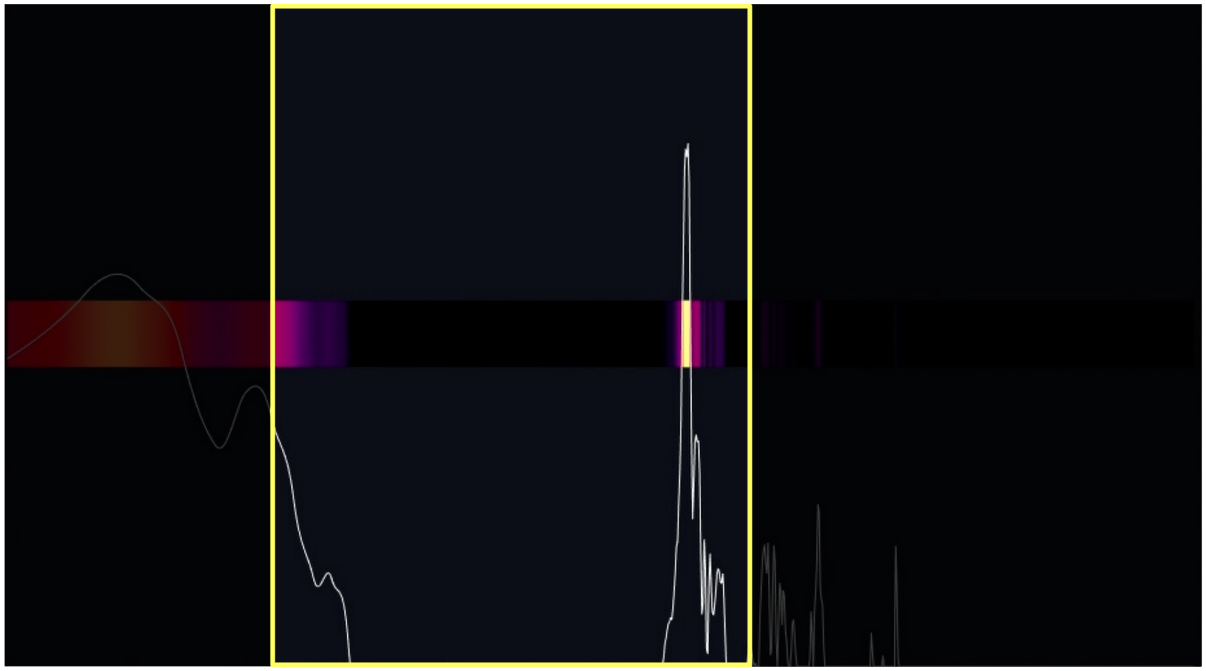

Figura 3.21: Imagen de un silbido donde se observa que la frecuencia máxima se encuentra dentro del intervalo propuesto

Las frecuencias detectadas que se encuentren dentro del intervalo del 20 % nombrado recibirán una puntuación de 0.25 (máxima puntuación). Los valores que se quedan fuera recibirán una puntuación entre 0.15 y 0, siendo 0.15 la frecuencia más próxima al intervalo y 0 la frecuencia más alejada.

3.2.6. Análisis del audio. Fase de generación de evento

Una vez obtenemos el nivel de reconocimiento del sonido, tenemos que decidir si la puntuación obtenida es lo suficientemente buena como para generar el evento correspondiente a dicho sonido. La clase *SoundRecognizer* es la encargada de generar el evento. Es la clase base de la que heredan *ClickRecognizer* y *WhistleRecognizer*, citadas anteriormente. El valor establecido a partir del cual se da por válido el reconocimiento del sonido es del 87 % de compatibilidad.

En la generación del evento influyen más factores a parte del nivel de reconocimiento. Inicialmente generábamos un evento en cada vuelta de bucle que se detectara el valor de reconocimiento establecido, pero nos dimos cuenta de que se generaban demasiados eventos y que había un porcentaje de error bastante alto en el que se detectaban los dos tipos de sonidos a la vez. La solución que encontramos fue generar el evento solo cuando se detectara repetidamente el mismo sonido. De esta manera, neutralizamos el porcentaje de error y disminuimos la generación de eventos. Esta solución también sirve para evitar que las réplicas del sonido producidas por el eco generen nuevos eventos.

La propia generación del evento está ligada a *Unity* y al sistema de *input*. El dato con el que se genera el evento es la frecuencia predominante en la muestra, obtenida gracias al parámetro de la posición de la frecuencia predominante. En el caso del evento de los golpes y chasquidos, la frecuencia predominante no se calcula por lo que su valor es 0.

En función del tipo de evento que queremos generar, se necesita realizar transformaciones en la frecuencia predominante para poder adecuar dicho valor, que es de tipo *float*, al tipo de estructura de datos necesaria para cada evento.

Para el tipo de evento *click* se necesita generar un booleano que representa si ha ocurrido o no el evento. El comportamiento que tiene es como un botón estándar, 'pulsado-no pulsado'.

El evento de tipo *whistle* es más complejo. Tiene dos ejes principales, uno equivale a la frecuencia a la que se silba, que tiene el comportamiento de un eje del *joystick* de un mando convencional, y el otro es el propio silbido, como un botón 'silba-no silba'.

Para generar los datos necesarios a partir de un solo valor de frecuencia, hicimos una representación gráfica de la función que queríamos que tuviese este control.

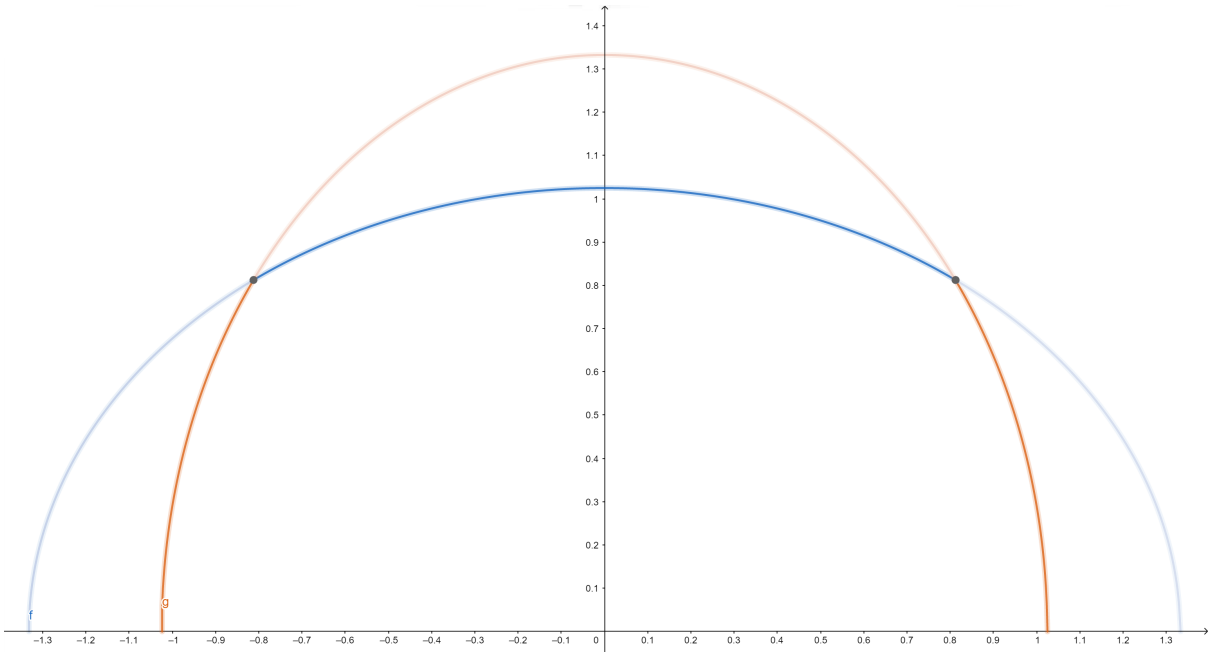

Figura 3.22: Visual del rango del input *Whistle*

El eje Y equivale a la detección del silbido y el eje X representa el rango *agudo-grave* de la frecuencia a la que se silba. En esta representación conseguimos, del círculo completo de direcciones, la mitad de ellas, pudiendo desplazar a la izquierda, delante y derecha, y todos los valores intermedios de la función.

La gráfica es la unión de la función de la mitad de una elipse orientada en la horizontal y otra en la vertical. El punto de corte entre las dos funciones es el (0.812, 0.812) y es donde se produce el cambio de una función a otra. Escogimos esta función porque las frecuencias de los extremos son más complicadas de silbar, por lo que teníamos que escoger una función en la que los valores cercanos a 1 y -1 se vieran beneficiados.

$$f(x) = \sqrt{1{,}05 - (x^2/1{,}3^2)} \text{ si } x <= 0{,}812$$

$$g(x) = \sqrt{(1{,}3^2 * (1{,}05 - x^2)} \text{ si } x > 0{,}812$$

Para hallar estas funciones hemos utilizado la calculadora gráfica que proporciona *Google*[13]. Estas ecuaciones son totalmente experimentales y después de probar con funciones más sencillas como la circunferencia o la recta, decidimos que una buena solución es la mostrada en la figura 3.22.

3.2.7. *Input System. Yondu Device*

Yondu Device, como todos los dispositivos en *Input System*, hereda de *InputDevice*, que actúa como contenedor de los controles. Este comportamiento es igual en todos los dispositivos y es el punto diferencial que tiene este sistema de entrada con el antiguo.

Cada control tiene un nombre único, pero se pueden asignar alias para mejorar la visualización en el editor. Esto es útil en dispositivos de diferentes marcas, como ocurre con los *Gamepads* de *Xbox* y de *PlayStation*.

A cada control se le puede asignar una utilidad. Estas utilidades especifican para qué se va a usar el control asociado. De esta manera se define un esquema de utilidades general, que después se aplica a los diferentes dispositivos y facilita la compatibilidad de las acciones de un videojuego. Por poner un ejemplo, existe la utilidad "*PrimaryAction*"que indica cual es el botón de acción principal del dispositivo.

Los controles en realidad no almacenan valores. Existe un estado de entrada que se localiza en bloques de memoria sin procesar. El estado es administrado por el sistema de entrada. Cada control recibe un bloque de estado de entrada (*InputStateBlock*) que lo utiliza para leer los valores del dispositivo. Estos valores componen el estado del dispositivo y representan los datos resultantes del estado de lectura.

El dispositivo *Yondu Device* está formado por dos controles: *Click* y *Whistle*. El primero representa un botón y el segundo la mitad de un joystick.

La inicialización de los dispositivos se realiza en el momento que el sistema detecta el dispositivo hardware. En ese momento se desencadena un proceso de búsqueda para encontrar la representación que más se asemeje al dispositivo conectado.

En el dispositivo *Yondu Device* este proceso es diferente. Nuestro dispositivo solo se inicializa y, por tanto, el sistema lo detecta, al ejecutar la escena en el editor. El componente *YondulibManager* es el encargado de crear el dispositivo en el momento en el que se ejecuta dicha escena.

3.2.8. Dependencias de YONDULIB

Una dependencia es una aplicación o una librería requerida por un programa para poder funcionar correctamente. En *Unity* existe un gestor de paquetes llamado *Unity Package Manager* que permite gestionas las dependencias del proyecto.

Una de las principales demandas que se les piden a estas librarías es que puedan ser utilizadas en la mayor parte de entornos. La librería que utilizamos para gestionar el sonido, *Libsoundio*, es multiplataforma, por lo que tiene una gran compatibilidad para la mayoría de ordenadores. [15]

[13]Calculadora gráfica de Google `https://www.geogebra.org/`

Como se explica en el repositorio de descarga del YONDULIB [14], antes de comenzar con la instalación, hay que definir las dependencias en el JSON de las librerías de *Unity*. Estos paquetes son: *unity.inputsystem*, *unity.mathematics*, *unity.burst*, *jp.keijiro.libsoundio* y *org.nuget.system.memory*.

Todas las dependencias son compatibles en *Unity* pero dos de ellas están alojadas en un sistema diferente. Estos sistemas son *NuGet* y NPM.

Scoped Registries

NuGet es el sistema de gestión compatible con *Microsoft* para compartir código en .NET, al igual que lo es *Maven* con Java y NPM con JS. Estos sistemas son especialmente utilizados en proyectos grandes.

Los *Scoped Registries*[25] permiten comunicar a *Unity* la localización de cualquier registro *custom* de paquetes al *Package Manager* para que pueda acceder a distintas colecciones de paquetes al mismo tiempo. Para simplificar, esta herramienta permite a *Unity* acceder a paquetes alojados en otros servidores, descargarlos e instalarlos.

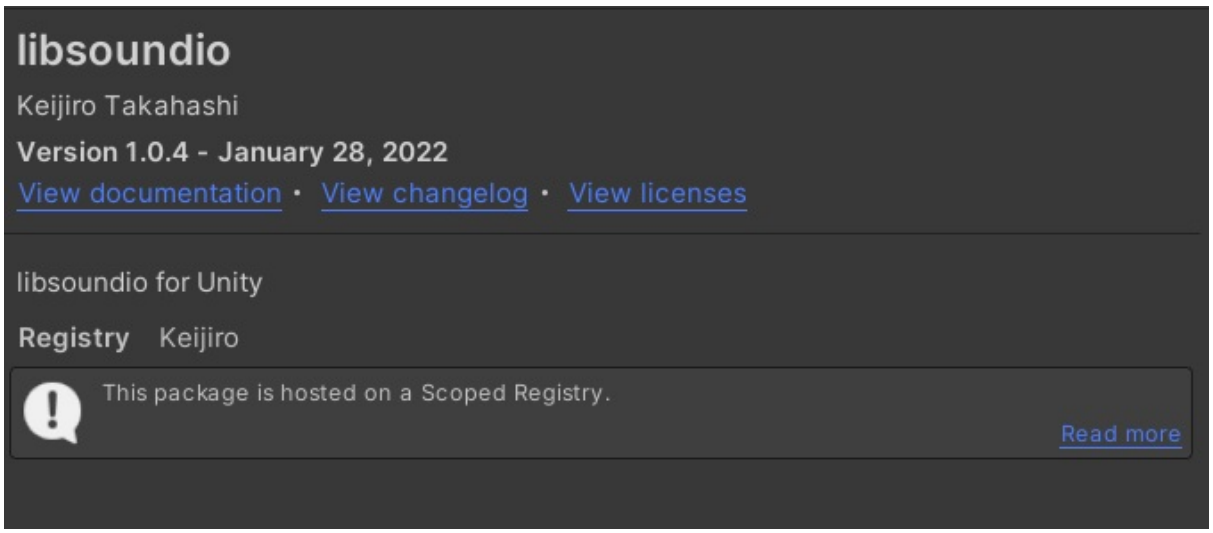

Figura 3.23: libsoundio pertenece a un *Scoped Registry*

Para poder añadir este tipo de paquetes a un proyecto de *Unity* es necesario editar el archivo *manifest.json*:

```
"scopedRegistries": [
    {
      "name": "Unity NuGet",
      "url": "https://unitynuget-registry.azurewebsites.net",
      "scopes": [
        "org.nuget"
      ]
    },
```

[14] `https://github.com/nubango/yondulib` Repositorio de YONDULIB

```
    {
      "name": "Keijiro",
      "url": "https://registry.npmjs.com",
      "scopes": [
        "jp.keijiro"
      ]
    }
  ]
```

Como vemos en el texto aparece el nombre, la url y el paquete que se quiere descargar. Una vez realizado este paso, podemos añadir la dependencia del paquete en la sección de *dependencies* y *Unity* lo detectará e instalará en el proyecto automáticamente:

```
"dependencies": {
    "com.unity.inputsystem": "1.0.2",
    "com.unity.mathematics": "1.2.5",
    "com.unity.burst": "1.4.11",
    "jp.keijiro.libsoundio": "1.0.4",
    "org.nuget.system.memory": "4.5.3"
 }
```

3.2.9. Integración con entorno de desarrollo de videojuegos

La integración de YONDULIB es diferente dependiendo de la situación de partida, pero la instalación sigue el mismo proceso (todos los detalles en el *GitHub* del *plugin*[15]). Si partimos de un proyecto nuevo, la integración de YONDULIB es sencilla y rápida, ya que es un plugin integrado a la perfección en *Unity*.

Si queremos integrar YONDULIB en un proyecto en desarrollo o ya terminado, surgen diferentes complicaciones en función del sistema de *input* que se esté utilizando. Si el proyecto está desarrollado con *Input System* la integración será más fácil y sencilla, ya que esta dependencia es la que más problemas de compatibilidad genera. En este supuesto, una vez instalado, deberíamos poder acceder al dispositivo *Yondu Device* en el editor de *Input Actions* y a todos los controles que este ofrece. Para poder utilizar estos controles es preciso que se incluya en la escena un objeto con los componentes de *YondulibManager* y *YonduDeviceSupport*. Con la descarga del plugin se proporciona un ejemplo de prueba en el que se muestra las estructuras necesarias para que todo funcione correctamente.

En el supuesto de que se quiera integrar YONDULIB en un proyecto que utiliza el *input* antiguo de *Unity*, el tiempo de integración será más largo. La duración de este proceso depende principalmente de tres factores: cuanto desarrollado esté el *input*, lo complejo que sea y cuanto de independiente es este módulo del resto del videojuego.

Si el proyecto no tiene muy desarrollado el módulo de *input*, la integración será más fácil ya que este módulo tendrá poca influencia con el resto de las secciones del proyecto.

La complejidad también afecta al tiempo de integración, ya que si es muy complejo significa que habrá muchas conexiones entre las diferentes partes del videojuego. Esto supone un aumento del tiempo de integración porque cada acceso al sistema de input antiguo hay que cambiarlo y adaptarlo al nuevo sistema.

[15]Github de YONDULIB `https://github.com/nubango/yondulib`

Por último, la calidad del código y el grado de abstracción de los accesos al *input* antiguo son dos factores de gran relevancia a la hora de reducir el tiempo de integración.

3.3. Pruebas

Las pruebas que hemos realizado se pueden dividir en tres grupos: pruebas sobre los reconocedores de sonidos, pruebas sobre *YonduDevice*, y las pruebas de integración en otros proyectos.

3.3.1. Pruebas de reconocedores

La primera parte del desarrollo la realizamos en el proyecto LASP, ya que proporciona efectos visuales a la entrada de audio detectada. Utilizamos una escena de *Unity* en la que muestra la intensidad de cada frecuencia en el eje longitudinal a través de una función y una textura que simula un mapa de calor, mostrando tonos cálidos para las intensidades altas y tonos fríos para las intensidades bajas.

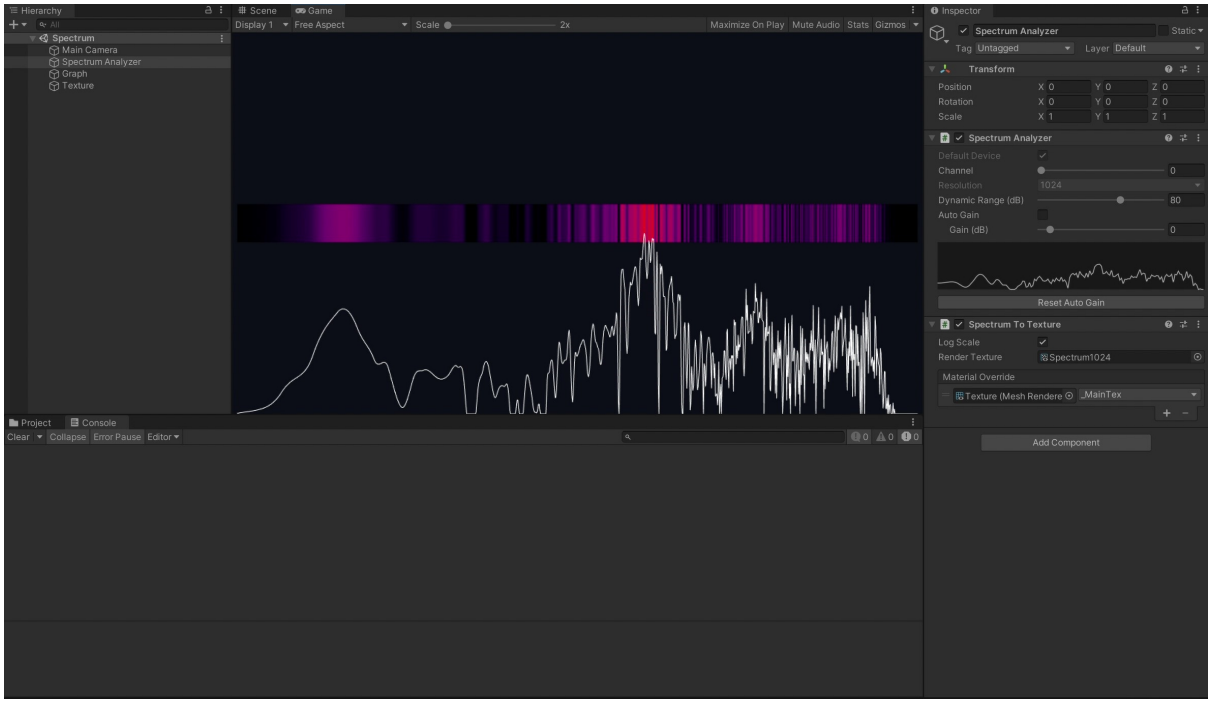

Figura 3.24: Entorno de pruebas

Las pruebas las íbamos realizando en micrófonos distintos ya que la recogida del audio varía en función de la calidad de los mismos. La interfaz permite elegir el micrófono de una forma muy sencilla por lo que facilitó conmutación entre el micrófono del ordenador y otro conectado externamente.

Podemos dividir las pruebas en dos fases principales: la fase de creación y la fase de efectividad. En la fase de creación, las pruebas que realizamos fueron orientadas a la búsqueda de parámetros en el audio que nos ayudasen a asociarlo a un sonido concreto. Primero observamos cómo cambia el espectro con cada sonido y buscamos las diferencias

entre ellos. De esta manera obteníamos pistas de lo que debíamos analizar. Después, programamos algoritmos que obtuviesen los datos que queríamos y observamos si realmente eran útiles o no a través del depurador de mensajes, mostrando los valores obtenidos y comparándolos con los diferentes sonidos.

En la fase de efectividad, las pruebas estaban orientadas a mejorar la efectividad del reconocedor ya programado. Emitimos mensajes de los datos recogidos para poder ajustarlos y mejorar la efectividad de los reconocedores. Una vez creada la escena de prueba, pudimos probar la efectividad de los reconocedores en ella, aunque para el ajuste de parámetros se sigue utilizando el depurador de mensajes.

3.3.2. Pruebas del input

Input System proporciona una interfaz gráfica donde comprobar los eventos generados por los dispositivos de entrada conectados. A través de esta interfaz hemos podido comprobar el formato de los eventos y nos ha permitido depurar más a fondo el dispositivo *YonduDevice*.

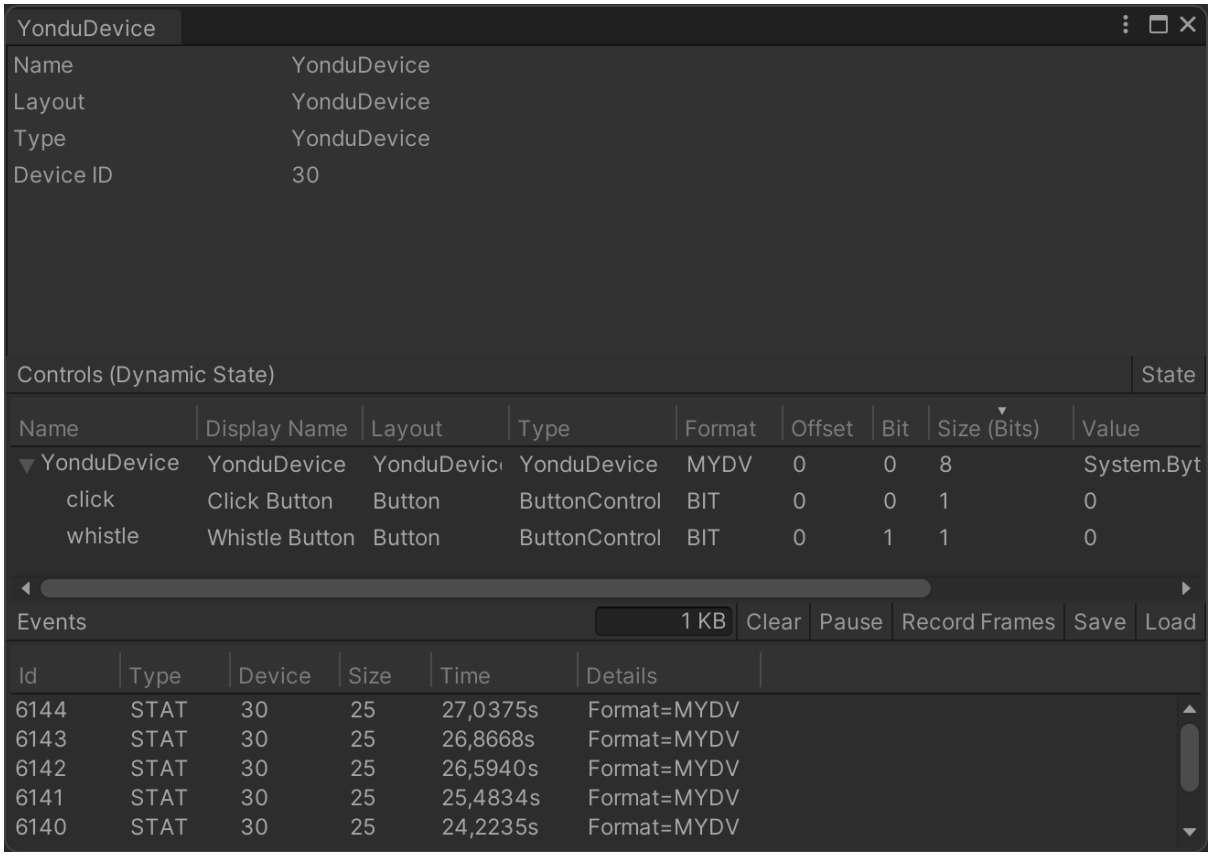

Figura 3.25: Depurador de dispositivo

Este depurador muestra los eventos en tiempo real y permite acceder a los valores que toma cada evento. También permite visualizar el formato en el que se capturan los datos y el tamaño que ocupan en memoria. Combinamos esta herramienta con los mensajes por consola para, de esta manera, poder ajustar los parámetros necesarios, tanto para la generación del evento como para ajuste en el reconocimiento.

Donde más hemos podido explotar estas pruebas ha sido en el control de silbidos ya que los datos recogidos se presentan como un rango de valores y precisa de mayor ajuste y depuración.

3.3.3. Pruebas de integración en otros proyectos

Este tipo de pruebas están orientadas a la búsqueda de errores en el *plugin* al realizar la importación e integración de YONDULIB en un videojuego ya creado.

Ejemplo de prueba

El ejemplo incluido con el *Package* está formado por un escenario FPS creado a base de cubos simples sin texturas. El control de los silbidos es utilizado para mover al personaje por el escenario. El intervalo agudo-grave corresponde al intervalo izquierda-derecha. El control de los chasquidos, golpes y palmas es el asignado para realizar la mecánica de disparar cubos de colores.

Esta escena ha sido sacada de un ejemplo de la la librería *Input System* en la que se mostraba el uso de la nueva interfaz de acciones. Aprovechamos esta escena ya que contiene una infraestructura básica del nuevo sistema de input adecuada para integrar YONDULIB.

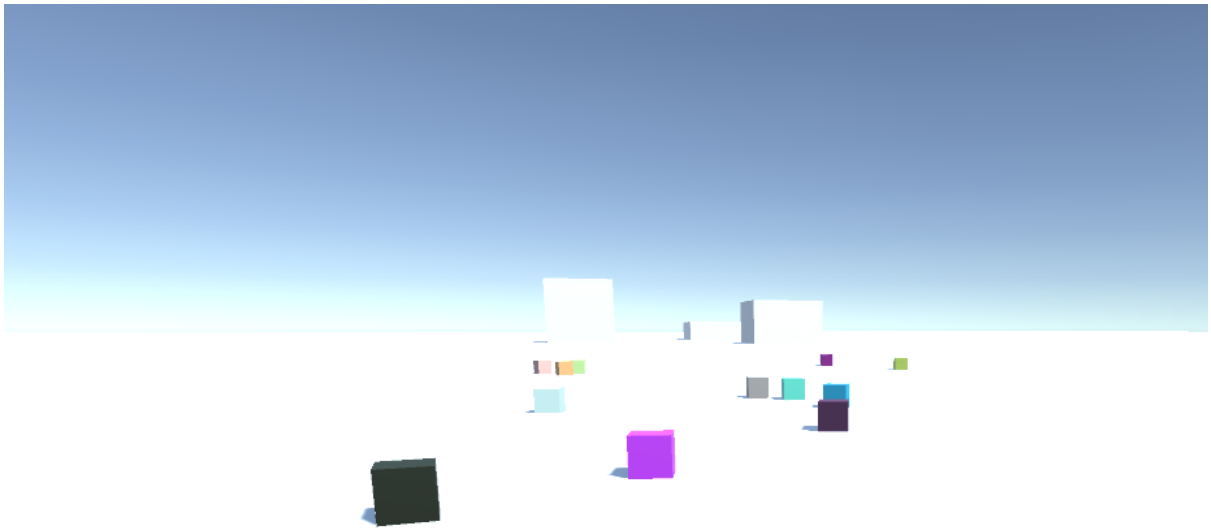

Figura 3.26: Escena del ejemplo de prueba

La integración fue sencilla y rápida, siguiendo los pasos descritos en el *GitHub* de YONDULIB[16] el plugin se instaló directamente. Solo hizo falta añadir los controles del *Yondu Device* a las acciones de moverse y disparar para que todo funcionase.

[16]GitHub de YONDULIB `https://github.com/nubango/yondulib`

Karting

Karting microgame es el nombre de un videojuego de prueba, descargado de *Unity Learn*[17], que consiste en manejar un coche en un circuito simple para pasar por unos puntos de control antes de que se acabe el tiempo. Este juego es perfecto para mostrar el control de silbidos en la mecánica del manejo del coche, donde el coche gira a derecha o izquierda en función de si se está silbando agudo o grave.

Figura 3.27: Escena de YONDULIB integrado en el juego *Karting* de *Unity Learn*

La integración de YONDULIB en este proyecto ha sido más complicada que en el ejemplo anterior. Esto se ha debido, en gran medida, a que el videojuego está construido sobre el sistema de *input* antiguo de *Unity*.

Inicialmente se instala el *plugin* en el proyecto siguiendo los pasos habituales. Una vez instalados, salta un mensaje de que es necesario reiniciar *Unity* para hacer el cambio de motor de *input*. Una vez reiniciado, si vamos a los ajustes del proyecto, veremos que por defecto *Unity* ha puesto la opción de utilizar los dos motores a la vez.

Inicialmente probamos cambiando los ajustes para que solo utilizase el nuevo *input*, pero al ser un juego ya acabado, esta opción nos presentó algunos problemas de compatibilidad, sobretodo a la hora de manejar las interfaces. Por eso decidimos utilizar la combinación de los dos motores de *input*. El nuevo *input* lo implementamos en los controles del *gameplay* y el antiguo, lo utilizamos en el control de los menús.

Este proyecto utiliza los objetos *Assembly Definitions* para organizar el código, por lo que para poder utilizar el nuevo *Input System* en los *scripts* del juego hemos tenido que añadir dicha dependencia en el fichero *assambly* correspondiente.

[17] *Karting microgame* `https://learn.unity.com/project/karting-template`

First person shooter

Este videojuego también pertenece a *Unity Learn*[18]. La finalidad de este videojuego es derrotar a todos los enemigos con el arma equipada. Contiene diferentes tipos de enemigos que no dudarán en vaciar tu barra de vida, que si llega a cero se acaba el juego.

Este juego presenta la oportunidad de mostrar la utilización de YONDULIB junto con controles de otros dispositivos. La acción de moverse por el escenario se realiza mediante silbidos, como sucede en el juego de *Karting*, silbar más agudo hace que el personaje se mueva hacia la izquierda y silbar más grave hacia la derecha. El control de que detecta los chasquidos y las palmas lo hemos asociado a la acción de saltar. La acción de mirar la realizamos con el movimiento del ratón y la acción de disparar con el botón izquierdo, también del ratón.

Figura 3.28: Escena de YONDULIB integrado en el juego FPS de *Unity Learn*

Como en el juego de *Karting* la integración de YONDULIB tiene algunas complicaciones ya que también utiliza el sistema de *input* antiguo de *Unity*. Este juego tiene más mecánicas que el anterior ya que podemos disparar, saltar y cambiar de arma. Al realizar el cambio al sistema nuevo de *input* hemos optado por simplificar alguna de estas mecánicas para la correcta integración del *plugin*.

Los pasos a seguir para la correcta integración son los mismos que en el juego explicado en el punto anterior (3.3.3). La herramienta se instala a través del *Package Manager* insertando la dirección del repositorio de *GitHub*, como especificamos en la descripción del mismo repositorio. Una vez instalado hay que reiniciar *Unity* para que el cambio al nuevo sistema de *input* se haga efectivo. Este proyecto también utiliza los objetos *Assembly Definitions* para organizar el código, por lo que hay que añadir a las dependencias la librería de *Input System*.

[18]FPS de *Unity Learn* `https://learn.unity.com/project/fps-template`

Capítulo 4

Conclusiones y trabajo futuro

4.1. Recapitulación

El resultado del proyecto es YONDULIB, una librería que permite la utilización de sonidos como input en el desarrollo de videojuegos. Para ello se han desarrollado dos sistemas. El sistema de reconocimiento de sonidos y el sistema de entrada.

El primer sistema es el encargado de captar los datos del sonido recogidos por el micrófono. Para ello, hacemos uso de la librería *Libsoundio*[15], que permite acceder a los dispositivos de audio en *Unity*. Esta librería es multiplataforma y es compatible con la mayoría de los motores de audio actuales.

Una vez obtenido el flujo de datos, analizamos la señal para identificar el sonido que ha ocurrido. Hemos desarrollado dos reconocedores de sonidos diferentes, el de chasquidos, golpes y palmas, y el de silbidos. Cada uno analiza el flujo de datos en base a unos parámetros específicos del sonido que reconoce.

Por último, generamos el evento asociado al reconocedor. Esta fase es el punto de unión entre los dos sistemas ya que la generación del evento está ligada al sistema de entrada que se utilice.

El segundo sistema es el encargado de gestionar los eventos generados y asignar dichos eventos a las acciones del juego. Este sistema se llama *Input System* que ha sido desarrollado por *Unity* y lanzado recientemente. Permite la creación de dispositivos personalizados. Aprovechando esta funcionalidad, hemos creado el *Yondu Device*. Es un dispositivo que contiene dos controles, un botón y un joystick, que corresponden a los reconocedores de chasquidos y silbidos nombrados anteriormente.

La combinación de estos dos sistemas da como resultado el plugin YONDULIB para desarrollo de videojuegos en *Unity*.

4.2. Conclusiones

Con este trabajo hemos explorado las posibilidades que presentan los sonidos como mecanismo de control para videojuegos. Podemos concluir que es viable el desarrollo de una herramienta que permita utilizar este tipo de controles en *Unity*.

En la sección 1.3 detallábamos los objetivos que nos planteamos al principio del proyecto. A continuación, detallamos el grado de consecución de dichos objetivos:

> *Objetivo 1: reducir la complejidad de los sonidos reconocidos, simplificándolas en golpes o silbidos, para reducir el tiempo de procesamiento y, por ende, el tiempo de respuesta. Para ello, necesitaremos que el acceso al audio tenga baja latencia y el reconocimiento tenga lugar en tiempo real.*

YONDULIB contiene dos tipos de reconocedores que generan dos tipos de controles. Esto es posible, como se explica en la sección 3.2.3, gracias al acceso al audio en baja latencia, que nos permite acceder y procesar el audio en un tiempo lo suficientemente corto como para que podamos emitir una respuesta viable en un videojuego (3.3). Por lo tanto, podemos afirmar que el Objetivo 1 que nos habíamos propuesto, lo hemos conseguido implementar.

> *Objetivo 2: que sea posible, para desarrolladores, añadir nuevos tipos de sonidos asociados a acciones. Para ello, necesitamos que la estructura sea ampliable.*

Nuestro proyecto es ampliable lo que permite aumentar el número de reconocedores ya que la estructura del proyecto está diseñada para ello, como se explica en el punto 3.2.2. Hemos implementado dos controles, que se apoyan en los reconocedores de palmas y silbidos respectivamente.

> *Objetivo 3: que todo el paquete sea utilizable. Para ello, necesitaremos simplificar la integración en juegos lo más posible.*

Este proyecto proporciona un *plugin* funcional que se puede descargar añadiendo la dirección de *GitHub* en la pestaña *Package Manager* en *Unity*. Como se muestra en las pruebas (3.3.3), la implementación de esta herramienta es totalmente viable tanto en proyectos de nueva creación como en proyectos donde el producto final ya está terminado.

4.3. Trabajo futuro

Este proyecto tiene varias vías que permiten la ampliación y el desarrollo del plugin. Principalmente, hemos identificado tres regiones que se pueden aprovechar como punto de partida para seguir desarrollando y mejorar la herramienta.

Reconocimiento de sonidos

El módulo de reconocimiento de sonidos tiene gran potencial de desarrollo. El reconocedor de golpes actualmente no hace distinción entre palmas y chasquidos. Un trabajo futuro puede ser la mejora de este reconocedor para que haga la distinción. De esta forma se ampliarían los controles del *Yondu Device*.

Lo mismo ocurre con el reconocimiento de silbidos, se podría mejorar acercándose a un comportamiento de un instrumento musical, diferenciando las distintas notas de la escala. Esto implicaría un mapeo de cada nota como un *input* diferente y habría que analizar las ondas con mayor precisión. Partiendo del reconocedor de silbidos, ajustando parámetros existentes y añadir, si es necesario, nuevos parámetros.

Trasladando estos nuevos reconocedores al módulo de input (*Yondu Device*) tendrían el mismo comportamiento que los silbidos. De esta forma tendríamos tantos *joystick* como notas.

Pruebas

La automatización de las pruebas sobre los reconocedores ayudaría a reducir los tiempos de desarrollo. Se podría desarrollar un módulo que permitiese cargar archivos de audio para sustituir la entrada del micrófono por estos archivos. Este módulo permitiría probar los reconocedores en cualquier localización, eliminando los agentes externos como el ruido ambiente o el eco, ya que se dejaría de utilizar el micrófono para realizar las pruebas.

Otra mejora que ayudaría a la realización de pruebas sobre los reconocedores sería el uso de retroalimentación gráfica para el análisis de los parámetros de los reconocedores. La lógica y las texturas se podrían aprovechar del proyecto ya nombrado LASP[1] de *Keijiro Takahashi*. De esta forma podríamos analizar más fácilmente el grado de acierto de cada parámetro y calibrarlo con mayor precisión.

Editor

A nivel del editor, se podría seguir trabajando para mejorar la experiencia de usuario al utilizar YONDULIB. Un camino que se puede explorar es la inicialización del *Yondu Device* en tiempo real, sin necesidad de iniciar el juego. Profundizando en esta idea, se podría dar la posibilidad de elegir cuando queremos que se inicialice el dispositivo. Esto sería útil tanto para el desarrollo del propio plugin, como para el desarrollo de juegos.

Otro camino a seguir, podría ser la mejora de la interfaz de los componentes del plugin. Añadir controles en el propio editor para elegir el micrófono que utilizará el dispositivo, así como parámetros modificables que permitan calibrar el micrófono para optimizar el reconocimiento de los sonidos.

[1]LASP `https://github.com/keijiro/Lasp`

Capítulo 4

Conclusions and future work

4.1. Recapitulation

The result of the project is YONDULIB, a library that allows the use of sounds as input in the development of videogames. Two systems have been developed for this purpose. The sound recognition system and the input system.

The first system is in charge of capturing the sound data collected by the microphone. In order to do this, we make use of the *Libsound* library, which allows access to audio devices in *Unity*. This library is cross-platform and is supported by most current audio engines.

Once the data stream is obtained, we analyse the signal to identify the sound that has been produced. We have developed two different sound recognisers, one for clicks, taps and claps, and one for whistles. Each one analyses the data stream based on specific parameters of the sound it recognises.

Finally, we generate the event related to the recogniser. This stage is the link between the two systems, as the generation of the event is linked to the input system being used.

The second system is in charge of managing the events generated and mapping these events to the actions of the game. This system is called *Input System* which has been developed by *Unity* and released recently. It allows the development of custom devices. By taking advantage of this functionality, we have created the *Yondu Device*. It is a device that consists of two controls, a button and a joystick, which correspond to the click and whistle recognisers named above.

The two systems combine to form the YONDULIB plugin for game development in Unity.

4.2. Conclusions

Through this work we have explored the possibilities presented by sounds as a control mechanism for videogames. We can conclude that it is technically feasible to develop a tool that allows the use of this type of controls in *Unity*.

In the section 1.3 we detailed the goals we set ourselves at the beginning of the project. Now, we detail the level of accomplishment of these objectives:

> *Objective 1: to reduce the complexity of the recognised sounds, by simplifying them into beats or whistles, in order to improve the processing time and, therefore, the response time. To do this, we will need the audio access to be low latency and the recognition to take place in real time.*

YONDULIB contains two types of recognisers that generate two types of controls. As explained in the 3.2.3 section, this is made possible by low latency audio access, which allows us to access and process audio in a short time enough for us to be able to produce a viable response in a videogame (3.3). Therefore, we can affirm that we have managed to accomplish Objective 1.

> *Objective 2: to make it possible for developers add new types of sounds associated with actions. To achieve this, we need the structure to be extendable.*

Our project is extensible, which allows us to increase the number of recognisers, since the project structure is designed for this, as explained in the point 3.2.2. We have implemented two controls, which rely on the clap and whistle recognisers respectively.

> *Objective 3: to make the whole package usable. For that, we will need to simplify the game integration as much as possible.*

This project provides a functional plugin that can be downloaded by entering the *GitHub* address in the *Package Manager* tab in *Unity*. As the tests show (3.3.3), the implementation of this tool is fully viable in both start-up projects and projects where the final product is already finished.

4.3. Future work

This project has several paths that allow further extension and development of the plugin. Mainly, we have identified three areas that can be used as a starting point for further development and tool improvement.

Sound recognition

The sound recognition module has great potential for development. The tap recogniser currently does not make a distinction between clapping and clicking. Future work could be to improve this recogniser to distinguish them. This would expand the controls of the *Yondu Device*.

The same applies to whistle recognition, which could be improved by approaching the behaviour of a musical instrument, by differentiating between the notes of the musical scale. This would mean mapping each note as a different *input* and the waves would require a more accurate analysis. Starting from the whistle recogniser, by adjusting existing parameters and adding, if necessary, new parameters.

If these new recognisers were transferred to the input module (*Yondu Device*), they would have the same behaviour as the whistles. This way we would have as many *joystick* as notes.

Testing

Test automatization on the recognisers would help reduce development times. A module could be developed to load audio files in order to replace the microphone input with these files. This would allow testing of the recognisers in any location, removing external agents such as ambient noise or echoes, as it would no longer use the microphone for testing.

Another improvement that would help testing of recognisers would be the use of graphical feedback for the analysis of recogniser parameters. Logic and textures could take advantage of the already mentioned LASP project by *Keijiro Takahashi*. This would make it easier to analyse the accuracy of each parameter and calibrate it more accurately.

Editor

At the editor level, work could be done further to improve the user experience when using YONDULIB. One path that could be explored is the initialisation of the *Yondu Device* in real time, with no need to launch the game. Further along this idea, we could provide the option of choosing the time when we want the device to be initialised. Both for plugin development itself and for game development, this would be useful.

Another path to follow could be to enhance the interface of the plugin components. Adding controls in the editor to select the microphone to be used by the device, as well as adjustable parameters to calibrate the microphone in order to optimise sound recognition.

Capítulo 5

Contribuciones

Este proyecto abarca tanto programación con librerías de sonido como el uso de *Unity*. Esto implicó que en los primeros pasos del proyecto se pudo trabajar sin dependencias entre las dos partes citadas y que, siendo dos integrantes, no hubo complicaciones a la hora de repartir el trabajo.

A pesar del reparto de las tareas, el proyecto se ha visto marcado por la comunicación para que ambos integrantes tuvieran conocimiento de los avances, independientemente de las primeras decisiones de investigación que llevaron a cada uno a profundizar en temas separados.

A continuación, cada participante expone sus aportaciones.

5.1. Gonzalo Alba Durán

Después del contratiempo que se explica en la sección 1.4, comenzamos una serie de reuniones, con nuestro tutor Manuel Freire, para decidir qué dirección tomar. Finalmente, decidimos indagar en la idea de controlar un videojuego a través de sonidos.

Las primeras investigaciones giraron en torno a la búsqueda de posibles herramientas para *Unity* que nos sirviesen de referencia a la hora de desarrollar nuestra idea. Al no encontrar nada parecido, tanto Nuria como yo, procedimos a buscar la viabilidad del proyecto y si era posible desarrollarlo. Yo me encargué de buscar herramientas de acceso a sonido fuera de *Unity* o librerías de bajo nivel que ofreciesen compatibilidad con el lenguaje C#, que es con el que se programa en *Unity*. Finalmente, encontramos el proyecto LASP[1] creado por *Keijiro Takahashi* que parecía acceder al audio en tiempo real y con baja latencia.

A partir de este momento, nuestros esfuerzos fueron dirigidos a descubrir qué partes de LASP podíamos utilizar para desarrollar nuestra idea. Después de un tiempo probando el contenido del nuevo descubrimiento, decidimos utilizar una escena llamada *Spectrum Analyzer* debido a su simplicidad y al aporte visual del sonido que nos ofrecía, ya que podía ser nos útil a la hora de desarrollar el reconocimientos de sonidos.

[1]LASP `https://github.com/keijiro/Lasp`

En este momento decidimos optimizar nuestra fuerza de trabajo, mientras un integrante se encarga de la investigación y desarrollo del reconocimiento de sonidos, la otra persona investiga cómo crear eventos personalizados en *Unity* y que posibilidades nos brinda la plataforma.

En mi caso, me tocó la investigación y desarrollo del reconocimiento de sonidos. Inicialmente tuve que profundizar en cómo se accedía al sonido en LASP y qué parte del código era útil y cual no. Con nociones básicas de cómo funcionaba el proyecto de referencia, comencé a trabajar en formas de reconocimiento de sonidos. Los primeros prototipos reconocían palmas y chasquidos por separado. Paralelamente, para poder realizar pruebas, desarrollé un pequeño sistema de combos con la salida de texto como *feedback*.

Después de un tiempo, y de varias pequeñas mejoras en el algoritmo creado, tomé la decisión de unificar los dos reconocedores, debido a que solo distinguía bien los dos sonidos en condiciones ideales de entorno y sonido. Esta unificación dio pie al desarrollo de un nuevo reconocedor, el de silbidos. Este reconocedor me costó más desarrollarlo ya que guardaba pocas similitudes con los reconocedores creados anteriormente.

Una vez creados los dos reconocedores juntamos el trabajo realizado por las dos partes. La integración la hicimos conjuntamente. Fue un camino largo, ya que los reconocedores necesitaban refactorizaciones debido al alto porcentaje de fallos. Como la detección de silbidos todavía estaba en una fase temprana, la integración con *Input System* se hizo primero con la detección de palmadas. La mejora del reconocimiento de silbidos desembocó en una refactorización bastante profunda en las bases de los reconocedores. Esto afectó también a la detección de palmas, que experimentó una mejoría considerable. Esta actualización también trajo consigo la eliminación del sistema de combos que había creado, ya que dejó de ser útil en la integración con *Input System*.

Al tener encauzadas las dos partes del proyecto, comenzamos a trabajar en la creación del *plugin*. Para ello necesitábamos abstraer y organizar el código de las dos partes. Respecto a la parte de los reconocedores, decidí refactorizar el acceso al sonido, ya que había demasiados ficheros. Esto se debía a que había utilizado una escena del proyecto LASP como base. Finalmente, llegué a la librería base que utilizaba LASP para acceder al sonido, *Libsoundio*, y realicé los cambios pertinentes para integrarla en nuestro proyecto.

Después de este gran cambio, relevé a Nuria en la integración de los nuevos cambios. Mientras ella se dedicaba a la creación de la infraestructura del *plugin*, yo me dedicaba a pulir los reconocedores ya integrados en una escena de ejemplo con *Input System*.

Finalmente, con el *plugin* ya creado, realizamos retoques en la escena de ejemplo e integramos YONDULIB en varios videojuegos ya hechos.

5.2. Nuria Bango Iglesias

Teniendo en cuenta las motivaciones (1.2) comunes que teníamos desde un inicio, junto con los imprevistos que se pueden apreciar en el plan de trabajo (1.4), comienza el desarrollo con investigación a fondo.

Estas primeras investigaciones van de la mano con reuniones con nuestro tutor, Manuel Freire, que nos ayudó a ambos a centrar los esfuerzos en un camino que podría llegar a

ser fructífero. La idea que nació de estas reuniones fue adaptar el sonido y convertirlo en controles válidos para los jugadores.

Obviamente nos sentíamos más cómodos con investigaciones que rodeasen al motor *Unity*, ya que tenemos conocimientos previos. Puliendo la idea anterior, ahora tenemos el objetivo más claro de crear un plugin para *Unity* con el que controlar mediante sonidos un videojuego ya creado.

Aprovechando otros conocimientos del grado, mis primeras búsquedas avanzadas fueron sobre la librería de sonido FMOD. En un principio pensé que podría sacar algo en claro usando de forma más específica FMOD, descartamos la idea ya que no permite acceder al micrófono para su uso en situaciones que requieran de respuesta rápida, es decir, baja latencia.

Llegamos a un punto en el que las opciones ya existentes son muy limitadas, incluso de pago. Por una parte esto me impresionó por no saber qué información encontraríamos, pero por otra me impulsó a crear algo de lo que no mucha gente disponía en *Unity*. La esperanza vino al encontrar la librería LASP, con la que podríamos acceder en tiempo real al audio en *Unity*. Tras un periodo de pruebas con el entorno que nos proporcionaba *Keijiro Takahashi*, confirmamos que nuestro proyecto era prometedor.

Nuestra forma de continuar es dividir esfuerzos. Por una parte estaba la investigación sobre el reconocimiento de sonidos y por otra, sobre los dispositivos de input personalizados de *Unity*. Para trabajar sin dependencias entre nosotros, creamos dos proyectos distintos en un mismo *GitHub* compartido.

En mi proyecto comencé a usar el input que proporciona *Unity* por defecto. Esto me llevó al uso de *Delegates* para el enlace entre pulsación y acción. En las primeras pruebas, con estructuras propias, desarrollé un menú de configuración de controles que permitía intercambiar las teclas de cada acción en ejecución. Una vez que la estructura era estable, el siguiente paso fue crear una escena de prueba 3D sencilla en la que el objetivo era abrir una puerta mediante una palanca. Esta demo pretendía ser un acercamiento a lo que sería enlazar cualquier evento a una acción específica y me topé con la limitación de que no era posible crear un evento de un controlador distinto a los predefinidos, como lo son un mando o el teclado.

Este inconveniente me llevó a la investigación que me cambió la perspectiva del trabajo, el *Input System*. Únicamente había información en el manual oficial de *Unity* y en contados espacios de desarrolladores que están al tanto de las últimas novedades que presenta *Unity*, ya que este sistema está aún en desarrollo.

Una vez conocido el concepto de *Input System* tenía que comprender que era lo que nos permitía y hasta qué punto solucionaba los problemas sobre eventos y controladores. Intenté aprovechar la escena que tenía e integrarle este nuevo sistema, pero no era tan trivial como intuí por los pocos ejemplos que encontré. Opté por un nuevo enfoque gracias a los ejemplos que incluye el paquete del nuevo *input*.

Los ejemplos me llevaron hasta el concepto de *Custom Device*, que era lo que buscábamos, un dispositivo nuevo totalmente personalizable. Esta estructura permitía la integración de los reconocedores, en los que estaba trabajando Gonzalo, independientemente de las acciones que yo tuviese implementadas. Esto nos llevó al esperado punto de unificar el trabajo.

El momento de unificación desencadenó un periodo importante de refactorización, revisión de código y estructuras en general. El trabajo fue conjunto y terminó con un proyecto mucho más manejable.

Mientras Gonzalo hacía pruebas con los reconocedores para su optimización, yo me enfoqué en el desarrollo del plugin, que en *Unity* se conoce como *Package*. El *Package* implicó la creación de un nuevo GitHub que tenía que cumplir los requisitos que se mencionaban en la documentación oficial de *Unity*. Uno de los más importantes era la estructura del directorio. Gracias a la refactorización previa, el paso del proyecto sobre el que desarrollamos todo a esta estructura de carpetas fue mucho más asequible.

Como broche final, integramos el *Package* en dos juegos ya desarrollados que confirmaron que nuestro proyecto es funcional.

Índice de figuras

Índice de cuadros

Bibliografía

[1] Wikipedia, "Game controller." https://en.wikipedia.org/w/index.php?title=Game_controller&oldid=1109976863, 2022.

[2] Romulo Ochoa, Frank G. Rooney, and William J. Somers, "Using the Wiimote in Introductory Physics Experiments." https://aapt.scitation.org/doi/abs/10.1119/1.3527747, 2011.

[3] Einnews.com, "Microsoft Fully Unveils Kinect for Xbox 360 Controller-Free Game Device." https://world.einnews.com/pr_news/56709028/microsoft-fully-unveils-kinect-for-xbox-360-controller-free-game-device, 2010.

[4] Antonio Delgado, "El uso de la Realidad Virtual crece un 30 % en Steam, con las Quest 2 acaparando el 46 % de usuarios." https://www.geeknetic.es/Noticia/24383/El-uso-de-la-Realidad-Virtual-crece-un-30-en-Steam-con-las-Quest-2-acaparando-el-46-de-usuarios.html, 2022.

[5] Michael Schreiber, Margeritta von Wilamowitz-Moellendorff Ralph Bruder, "New Interaction Concepts by Using the Wii Remote." https://link.springer.com/chapter/10.1007/978-3-642-02577-8_29, 2009. Online; accessed 07 April 2022.

[6] Wikipedia, "Sonido." https://es.wikipedia.org/w/index.php?title=Sonido&oldid=145277145, 2022. Online; accessed 14 April 2022.

[7] Windows API, "Acerca de las API de audio principal de Windows." https://docs.microsoft.com/es-es/windows/win32/coreaudio/about-the-windows-core-audio-apis, 2022.

[8] Jeff Tranter, "Introduction to Sound Programming with ALSA." https://www.linuxjournal.com/article/6735, 2004.

[9] Apple Developer, "Working with Audio." https://developer.apple.com/audio/.

[10] U. Ruelas, "¿qué es un motor de videojuegos (game engine)?." https://codingornot.com/que-es-un-motor-de-videojuegos-game-engine, 2017.

[11] Wikipedia, "Motor de videojuego." https://es.wikipedia.org/w/index.php?title=Motor_de_videojuego&oldid=141146960, 2022.

[12] Pedro Pablo Fernández, "Motores Gráficos y de juego: Definición, tipos y modelos de Negocio." https://www.hyperhype.es/motores-graficos-y-de-juego-definicion-tipos-y-modelos-de-negocio/, 2020.

[13] J. M. Martínez Burgos, "Pros y contras de usar un motor de juegos." https://hafo.biz/videojuegos/pros-y-contras-de-usar-un-motor-de-juegos/, 2021.

[14] Unity, "Creating and using scripts." https://docs.unity3d.com/es/530/Manual/CreatingAndUsingScripts.html, 2016.

[15] Andrew Kelley, "libsoundio Documentation 2.0.0." http://libsound.io/doc-2.0.0/index.html.

[16] Wikipedia, "Desarrollo de videojuegos." https://es.wikipedia.org/w/index.php?title=Desarrollo_de_videojuegos&oldid=145175539, 2022.

[17] R. Márquez, "Los indies tenían razón: Unity y los motores de terceros le han ganado la partida a los motores propios a la hora de crear juegos." https://www.xataka.com/videojuegos/indies-tenian-razon-unity-motores-terceros-le-han-ganado-partida-a-motores-propios-a-hora-crear-juegos-1, 2021.

[18] Unity, "Actions." https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/manual/Actions.html, 2021.

[19] Unity, "Input system events." https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/manual/Events.html, 2021.

[20] Unity, "Asset packages." https://docs.unity3d.com/es/2020.2/Manual/AssetPackages.html, 2020.

[21] Unity, "Input system." https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/manual/index.html, 2021.

[22] Unity, "Devices." https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/manual/Devices.html, 2021.

[23] Keijiro Takahashi, "keijiro/Lasp." https://github.com/keijiro/Lasp/blob/v2/README.md, 2022.

[24] Andrew Kelley, "andrewrk/libsoundio." https://github.com/andrewrk/libsoundio/blob/master/README.md.

[25] Unity, "Scoped registries." https://docs.unity3d.com/2020.3/Documentation/Manual/upm-scoped.html, 2020.

[26] Wikipedia, "Microfono." https://es.wikipedia.org/w/index.php?title=Micr%C3%B3fono&oldid=145224402, 2022.

Gonzalo Alba Durán
Nuria Bango Iglesias

Ult. actualización 15 de septiembre de 2022
LaTeX lic. LPPL & powered by TEFLON