UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DEPARTAMENTO DE INGENIERÍA ELECTRÓNICA

DISEÑO DE MÓDULO AUDIO REACTIVO PARA LA REPRESENTACIÓN GRÁFICA DE AUDIO PARA EL PROYECTO SOUNDCOOL

Autor: Alberto Miguel Quirós Pérez

Tutor: Dr. Jorge Sastre Martínez

Cotutor: Dr. Germán Ramos Peinado

Trabajo Fin de Máster presentado en el Departamento de Ingeniería Electrónica de la Universitat Politècnica de València para la obtención del Título de Máster Universitario en Ingeniería de Sistemas Electrónicos

Curso 2019-20

Valencia, enero de 2020

Resumen

Este trabajo trata la implementación de un módulo audio reactivo para la plataforma Soundcool, el cual tendrá la función de representar el audio tratado de manera gráfica, devolviendo señales de vídeo basadas en el comportamiento del sonido de entrada.

Soundcool es un sistema que a partir de módulos sencillos puede crear obras de música, audio y de vídeo. Estos módulos se controlan desde los dispositivos que utilicen los colaboradores, que pueden ser móviles, tablets e incluso gafas de realidad aumentada. De esta manera se puede realizar la creación colaborativa de contenidos multimedia. Está dirigida a músicos, educadores y creadores de contenidos multimedia. Es un sistema gratuito desarrollado por el grupo PerformingARTech de la Universitat Politècnica de València con la colaboración de Carnegie Mellon University y dirigido por el por el Dr. Jorge Sastre.

Para llevar a cabo el trabajo se ha utilizado el lenguaje de programación Max/MSP/Jitter, el cual es un lenguaje gráfico de programación por bloques especializado en soluciones de audio y multimedia interactivas. Se utilizará para el análisis de audio principalmente las herramientas de análisis de señales. Para la generación de vídeo se utilizarán las herramientas de OpenGL de las que dispone MAX/MSP para la generación de objetos 2D y 3D, con el fin de aprovechar los recursos de la GPU y dejar el análisis del audio a la CPU. Todo esto se ha hecho con la finalidad de poder representar el audio tratado de manera gráfica y útil para las actividades tanto profesionales como educativas para el sistema.

Palabras Clave:

Soundcool, Max/MSP/Jitter, audio, vídeo, procesado de señal, OpenGL, GPU, multimedia, 2D, 3D, visualización de audio.

Resum

Este treball tracta la implementació d'un mòdul àudio reactiu per a la plataforma Soundcool, el qual tindrà la funció de representar l'àudio tractat de manera gràfica, tornant senyals de vídeo basades en el comportament del so d'entrada.

Soundcool és un sistema que a partir de mòduls senzills pot crear obres de música, àudio i de vídeo. Estos mòduls es controlen des dels dispositius que utilitzen els col·laboradors, que poden ser mòbils, tablets i inclús ulleres de realitat augmentada. D'esta manera es pot realitzar la creació col·laborativa de continguts multimèdia. Està dirigida a músics, educadors i creadors de continguts multimèdia. És un sistema gratuït desenrotllat pel grup PerformingARTech de la Universitat Politècnica de València amb la col·laboració de Carnegie Mellon University i dirigit pel pel Dr. Jorge Sastre.

Per a dur a terme el treball s'ha utilitzat el llenguatge de programació Max/MSP/Jitter, el qual és un llenguatge gràfic de programació per blocs especialitzat en solucions d'àudio i multimèdia interactives. S'utilitzarà per a l'anàlisi d'àudio principalment les ferramentes d'anàlisi de senyals. Per a la generació de vídeo s'utilitzaran les ferramentes d'OpenGL de les que disposa MAX/MSP per a la generació d'objectes 2D i 3D, a fi d'aprofitar els recursos de la GPU i deixar l'anàlisi de l'àudio a la CPU. Tot açò s'ha fet amb la finalitat de poder representar l'àudio tractat de manera gràfica i útil per a les activitats tant professionals com educatives per al sistema.

Paraules Clau:

Soundcool, Max/MSP/Jitter, àudio, vídeo, processat de senyal, OpenGL, GPU, multimèdia , 2D, 3D, visualització d'àudio.

Abstract

This work deals with the implementation of an audio reactive module for the Soundcool platform, which will have the function of representing the audio graphically, returning video signals based on the input sound's behavior.

Soundcool is a system that can create works of music, audio and video from simple modules. These modules are controlled from the devices used by collaborators, which can be mobile phones, tablets and even augmented reality glasses. This way you can perform collaborative creations of multimedia content. It is aimed at musicians, educators and creators of multimedia content. Soundcool is a free system developed by the PerformingARTech group of the Polytechnic University of Valencia with the collaboration of Carnegie Mellon University and directed by Dr. Jorge Sastre.

Soundcool is based on the programing languaje called MAX / MSP / Jitter, which is a graphic block programming language specialized in interactive audio and multimedia solutions. Mainly the signal analysis tools will be used for audio analysis. For the video generation, the OpenGL tools that MAX/MSP has available for the generation of 2D and 3D objects will be used, in order to take advantage of the GPU resources and leave the audio analysis to the CPU. All this has been done in order to be able to represent the audio treated in a graphic and useful way for both professional and educational activities for the system.

Keywords:

Soundcool, Max / MSP / Jitter, audio, video, signal processing, OpenGL, GPU, multimedia, 2D, 3D, audio display.

Índice general

I Memoria

1. Introducción **1**
- 1.1. Motivación . . . 1
- 1.2. Antecedentes . . . 1
- 1.3. Objetivos . . . 3

2. Materiales y métodos **5**
- 2.1. Herramientas utilizadas . . . 5
  - 2.1.1. Software Utilizado . . . 5

3. Desarrollo del proyecto **9**
- 3.1. Análisis y procesado de audio . . . 9
  - 3.1.1. Primeros Pasos con MSP . . . 9
  - 3.1.2. De audio a vídeo . . . 9
  - 3.1.3. Análisis espectral . . . 12
  - 3.1.4. Adquisición de pulsos . . . 18
- 3.2. Generación de vídeo . . . 24
  - 3.2.1. Primeros Pasos con *Jitter* . . . 24
  - 3.2.2. Primeros módulos con OpenGL . . . 25
  - 3.2.3. Shaders . . . 30
  - 3.2.4. Definición del entorno . . . 32
  - 3.2.5. Gráficos 2D . . . 33
    - 3.2.5.1. Audio Wave: Representación de la onda de audio . . . 33
    - 3.2.5.2. 2D LED Spectrum: Espectrograma amplitud/frecuencia . . . 34
    - 3.2.5.3. Espectrograma: Aplicación de colores . . . 38
    - 3.2.5.4. Complex Analysis: Representación del audio en planos complejos 40
    - 3.2.5.5. Otros módulos 2D: Coordenadas polares . . . 43
  - 3.2.6. Gráficos 3D . . . 44
    - 3.2.6.1. Ray Marching . . . 45
    - 3.2.6.2. Primeros pasos . . . 48
    - 3.2.6.3. Twisted Toro: Visión interna de una figura . . . 52
    - 3.2.6.4. Spheres: Creación de múltiples objetos y efectos de reflexión . 56
    - 3.2.6.5. Sonograph 3D: Espectro temporal de frecuencias . . . 59
- 3.3. Implementación del módulo . . . 61
  - 3.3.1. Módulo de Soundcool . . . 61
  - 3.3.2. Interfaz móvil de usuario . . . 63

4. Conclusiones **65**

5. Líneas futuras de trabajo **67**

6. Presupuesto **69**

7. Bibliografía **71**

II Anexos

A. Listados adicionales **77**

Índice de figuras

1.1. Logo de Soundcool 2

2.1. Logo de MAX/MSP 8 5
2.2. Ventana principal de MAX/MSP. 6
2.3. Ejemplo de un módulo de MAX/MSP. 7

3.1. Segmento de espacio de color RGBA y su codificación 10
3.2. Planos en la matriz RGBA en *Jitter* 10
3.3. Modos de adquisición de audio y conversión a vídeo. 11
3.4. Adquisición de audio 11
3.5. Onda compleja (derecha) y ondas sinusoidales que la forman (izquierda) 12
3.6. Transformada de Fourier. 13
3.7. Espectrograma de un violín 14
3.8. Espectrograma frecuencia_amplitud. 15
3.9. Espectro de frecuencias de 0 Hz a f_0 para sinusoide de 3kHz. 15
3.10. Módulo Espectrograma frecuencia-amplitud. 16
3.11. Bloque jit fft viz. 17
3.12. Bloque pfft amp ppft. 17
3.13. Bloque p logtable. 17
3.14. Algoritmo del espectrograma tridimensional. 19
3.15. Espectrograma obtenido. Modo logaritmico (izquierda) y lineal (derecho). 20
3.16. Espectrograma tiempo-frecuencia-amplitud. 20
3.17. Algoritmo para el seguimiento del ritmo. 20
3.18. Espectrograma de la señal original (arriba) y espectrograma de la señal filtrada (abajo). 21
3.19. Obtención de envolvente: Señal original (izquierda), valores absolutos (centro) y señal envolvente (derecha) 21
3.20. Ondas a generar: rampa ascendente, rampa descendente, sinusoidal y cuadrada 22
3.21. Bloque de adquisición de pulsos 23
3.22. Reproductor básico de vídeo 24
3.23. Logo de OpenGL. 25
3.24. jit.world y jit.gl.gridshape 26
3.25. jit.world, jit.gl.gridshape y jit.gl.camera 27
3.26. jit.gl.texture y adquisición de audio 27
3.27. jit.gl.material 28
3.28. jit.gl.light 28
3.29. Espectrograma 3D (Primera versión) 29
3.30. Módulos realizados con jit.gl.multiple 29

3.31. Rendering pipeline. 31
3.32. Entorno de representación de fragment shaders 32
3.33. Entorno de representación de *fragment shaders*. 32
3.34. Coordenadas normalizadas de la pantalla 33
3.35. Resultado de sample 34
3.36. Onda resultante. 34
3.37. Vumeter digital 35
3.38. Coordenadas normalizadas discretizadas. 35
3.39. Evolución de los colores de los LEDs 36
3.40. FFT discretizada con la resolución de la pantalla 36
3.41. FFT discretizada con la resolución de la pantalla 36
3.42. Búsqueda de los límites de las celdas(derecha) y generación de los bordes de los LEDs (izquierda) 37
3.43. Espectro 2D 37
3.44. Escalas de colores de matplotlib 38
3.45. Comparación de los mapas de colores viridis y magma con otros mapas utilizados 39
3.46. Resultados de la aplicación de los mapas de colores de la librería matplotlib al sonograma. 39
3.47. Representación de las coordenadas de los planos complejos. 41
3.48. Planos complejos con muestra del espectro de frecuencias. 42
3.49. Modelo de color HSV. 42
3.50. Módulos de planos complejos con coloración RGB. 43
3.51. Coordenadas polares de la pantalla: θ (derecha) y r (izquierda). 44
3.52. *Audio tunnel* (derecha) y *lotus* (izquierda). 44
3.53. Hombre dibujando un laúd con una técnica similar al ray casting (grabado en madera de Alberto Durero, 1525). 45
3.54. *Ray casting*. 46
3.55. Algoritmo de *Sphere Tracing*. 47
3.56. Formas planas (gris) y sus funciones de distancia con signo(rojo). 47
3.57. Distancias obtenidas por el algoritmo de *ray marching*. 50
3.58. Vectores normales obtenidos 51
3.59. Escena iluminada. 52
3.60. Escena iluminada con varias figuras. 53
3.61. *Torus*, de Reinder Nijhoff (2019). 53
3.62. Coordenadas polares del toroide: θ(derecha) y r(izquierda). 54
3.63. Espiral alrededor del toroide. 54
3.64. Vista interna del toroide. 55
3.65. Vista interna del toroide con texturas aplicadas. 55
3.66. Variaciones de las dimensiones del toroide y de las propiedades de la cámara. . . 56
3.67. Variaciones de las dimensiones del toroide y de las propiedades de la cámara. . . 56
3.68. Generación de múltiples cubos. 57
3.69. Coloración de los cubos por su posición en el espacio. 57
3.70. Cálculo de reflexiones. 58
3.71. Módulo con colores, reflexiones y sombras. 58
3.72. Módulo final con esferas. 59
3.73. Espectrograma de Chrome Music Lab 59
3.74. Entorno de desarrollo del espectrograma 60

3.75. Altura del espectrograma. 60
3.76. Espectrograma con el mapa de colores *viridis*. 61
3.77. Espectrograma 3D 61
3.78. Plantilla para módulo de soundcool. 62
3.79. Módulo Audio-Reactivo completo con otros módulos de Soundcool. 63
3.80. Interfaz del módulo en Unity. 64

Índice de tablas

3.1. Ecuaciones para análisis de planos complejos. 40
3.2. Código de configuración de la cámara para ray marching. 49
3.3. Código de ray marching/sphere tracing. 49
3.4. Código para la obtención de vectores normales de la superficie. 51
3.5. Código de iluminación de la escena. 52

6.1. Presupuesto. 69

Parte I

Memoria

Capítulo 1

Introducción

1.1. Motivación

Las motivaciones principales para la realización del siguiente trabajo son la oportunidad de pertenecer al equipo de *Soundcool* [1] y el ámbito técnico

La principal motivación que tiene este trabajo es la unificación de la música con los diferentes ámbitos de la ingeniería. Desde hace pocos años se ha adquirido experiencia por medio del estudio, modificación y diseño de diferentes pedales de efectos o *stompboxes* y otros dispositivos para alterar el sonido de los instrumentos.

Por otra parte, el máster en ingeniería de sistemas electrónicos nos introdujo a la generación de gráficos por medio de la programación en CUDA y al procesado digital de señales, entre otras muchas competencias. Este trabajo unifica las experiencias previas con el audio con los conocimientos recién adquiridos en generación de gráficos y procesado de señales.

Por último está la posibilidad de trabajar en el equipo del proyecto *Soundcool* [1], el cual consiste en una plataforma de producción colectiva de contenidos audiovisuales. El trabajar en este proyecto implicaría trabajar en equipo con un producto utilizado en varios países y contribuir con ideas para su mejora y evolución.

En este documento se ha intentado describir de una manera sencilla todo lo que se ha realizado a lo largo de este trabajo, con el fin de que cualquiera que esté interesado en la generación de gráficos y el análisis de audio en plataformas virtuales pueda, al menos, tener un manual de consejos y unas ideas claras de como abordar estos problemas.

1.2. Antecedentes

Desde hace ya algunas décadas, el trabajo del mundo audiovisual ha cambiado radicalmente desde la introducción de la tecnología y la informática. El desarrollo de estas tecnologías ha permitido crear escenarios y situaciones impensables hace algunos años, asimismo ha favorecido que, a día de hoy, el vídeo sea el método de entretenimiento más consumido en todo el mundo [2].

El auge del vídeo ha repercutido también en la forma con la que se disfrutaba de la música y de los espectáculos en vivo. Fue el 1 de agosto de 1981 cuando se estrenó la cadena de televisión

MTV y emitió el videoclip de ”Video Killed the Radio Star”, el cual fue el primer videoclip que se dio a conocer por televisión. Desde entonces, los músicos han intentado unificar sus canciones con un clip de vídeo que encaje con el carácter de la canción, con el fin de reforzar lo que quieren transmitir con ella y que sea capaz de impresionar a mucha más gente, convirtiendo lo que en un principio eran canciones en cortometrajes. Esto cambió la manera que había de disfrutar la música. Ahora las canciones disponen de un vídeo que las respalda para poder llegar a más gente. Algo muy similar ha pasado con los espectáculos en vivo, hoy por hoy no es raro ir a un concierto y encontrarse con proyectores de vídeo enormes o pantallas para la visualización de vídeos creados para el concierto o de los artistas, todo con la finalidad de amplificar las sensaciones producidas por el espectáculo.

Además de esto, la oferta musical ha aumentado con el avance de la tecnología, de manera que cualquiera puede producir música con un ordenador y un micrófono de forma casera y obtener una grabación con una calidad igual o incluso superior a la que se podría obtener en un estudio profesional hace años. Además, con internet estas grabaciones pueden llegar a cualquier lugar del mundo con un coste ínfimo, lo que ha obligado a los músicos a desmarcarse mediante el uso de medios audiovisuales en sus espectáculos o en sus vídeos promocionales.

Soundcool es una aplicación para la creación y edición colectiva de contenido multimedia, por medio de ordenadores y dispositivos móviles. El sistema se basa en un sistema modular, basado en bloques funcionales que se interconectan entre sí y cada uno de los usuarios se encarga de controlar un módulo a través de un dispositivo móvil o varios/todos a través del PC. Estos bloques están relacionados con la generación, reproducción edición y mezcla de audio y vídeo, y pueden ser controlados remotamente mediante una conexión WiFi. Los dispositivos no generan o reproducen sonido, sino que envían las instrucciones a la unidad central por medio del protocolo *Open Sound Control* (OSC) [3] y es esta unidad la que realiza las operaciones de procesado de audio y vídeo.

Figura 1.1: Logo de Soundcool

El proyecto tenía como objetivo primario ser una herramienta para la enseñanza musical en las aulas, objetivo que se cumplió de manera muy satisfactoria, llegando en 2015 a ser utilizada en aulas de España, Italia, Portugal y Rumanía, reportando excelentes resultados. En 2016 se realizó el estreno de “La Mare dels Peixos”, ópera realizada por el equipo de Soundcool, en el Palau de les Arts (Valencia). También se realizó un preestreno del primer acto de la obra en el Colegiul de Are Baia Mare (Rumanía). Ya en 2017 se realizó el preestreno del primer acto en el Instituto Tecnológico de Estudios Superiores de Monterrey. En 2019 se dio el preestreno del la obra en México y en 2020 llegará a Estados Unidos [4].

Desde hace unos años, Soundcool se está enfocando más en el mercado de aplicaciones profesionales, sin dejar de lado el ámbito educacional. Está buscando su uso en eventos, conciertos y festivales, sin dejar de lado el uso en escuelas, institutos y hogares. Por ello, Soundcool quiere realizar módulos de calidad profesional sin abandonar la sencillez y accesibilidad de la plataforma.

Este proyecto busca comenzar con la unificación entre la plataforma Soundcool y las necesi-

dades actuales de los eventos en vivo, creando un módulo capaz de representar audio de manera gráfica, aplicable a diferentes estilos y así poder adaptarse a la mayoría de las situaciones en las que se requiera completar una actuación escénica con un vídeo musical.

1.3. Objetivos

El objetivo del siguiente trabajo de investigación es el desarrollo del módulo audio reactivo para la plataforma Soundcool, capaz de generar gráficos que representen características del audio de forma visualmente atractiva.

Este objetivo se desglosa en tres objetivos parciales:

- Analizar el audio para adquirir sus propiedades más notables.
- Generación de gráficos atendiendo a los valores obtenidos por el objetivo parcial anterior.
- Implementación completa del módulo y puesta en marcha dentro de la plataforma.
- Implementación de una interfaz móvil para el control del módulo, tal y como el resto de módulos de la aplicación.

Capítulo 2

Materiales y métodos

2.1. Herramientas utilizadas

2.1.1. Software Utilizado

Para poder realizar este proyecto, se ha utilizado principalmente el programa MAX, de *Cycling'74* [5]. MAX es un entorno de desarrollo de software que utiliza un lenguaje de programación visual. Está orientado a las necesidades de artistas, educadores e investigadores que quieren trabajar realizando programas capaces de realizar tareas con recursos audiovisuales. Este programa está disponible para WindowsOS y MacOS.

Figura 2.1: Logo de MAX/MSP 8

MAX/MSP es una plataforma de desarrollo de software que incluye una gran variedad de herramientas para la creación y manipulación de sonido, gráficos, música y acciones interactivas. Se vale de un lenguaje de programación por bloques, con lo que ofrece un entorno de desarrollo flexible, potente y accesible a la mayoría de los usuarios. Con esta programación por bloques se puede tener un programa funcional en pocos minutos incluso sin ser un experto en programación.

MAX, con cada edición que publica, quiere abarcar el máximo rango de tecnologías que pueda, y lo consigue, pero sigue siendo un programa que está muy enfocado a la producción multimedia. En un inicio, la compañía Cyclin'74 publicó MAX dividido en programas diferentes, los cuales ahora están unificados bajo el nombre de MAX. Estos programas son:

- MAX: es la herramienta principal. En ella se encuentran todos los bloques de operaciones básicas y de comunicación con distintos dispositivos.

- MSP: es la herramienta que abarca todos los bloques de operaciones con señales de procesado, edición y síntesis de audio.Con esta herramienta se pueden generar sintetizadores virtuales, *samplers* y procesadores de efectos capaces de realizar el procesamiento de señales de audio a tiempo real.
- *Jitter*: se encarga de la edición de señales de vídeo a tiempo real y de la generación de gráficos en 2D y 3D.

El eslogan de Cycling'74 dice: "Conecta con todo"(*"Connect Everything"*). La compañía creadora de MAX/MSP ha conseguido que el programa funcione con una gran cantidad de herramientas capaces de interconectar módulos de diferentes áreas e incluso diversos dispositivos al programa que esté desarrollando el usuario, de manera que el usuario pueda extender los limites de su proyecto más allá del ordenador. Entre todos los dispositivos a los que puede conectarse, estos son los más destacados:

- Controladores MIDI [6]: MAX posee una librería de comunicación MIDI directa que le permite conectarse directamente a cualquier dispositivo MIDI y poder recibir o enviarle información y actuar en consecuencia.
- Entradas y salidas de audio: Se pueden conectar, a través de una tarjeta de sonido, todo tipo de instrumentos, sintetizadores y generadores de señal para poder controlar y editar el sonido en vivo.
- Proyectores e Iluminación DMX [7]: Con el fin de poder aplicar los programas de MAX a conciertos y festivales.
- Ableton [8]: Ableton permite el desarrollo y la aplicación de módulos en las pistas gracias a la librería de MAX, *MaxForLive*.
- Arduino [9]: MAX se puede configurar para conectarse a estos microcontroladores a través de un puerto USB [10] y empleando el protocolo de comunicación serie para el intercambio de datos como el control de elementos hardware y la lectura de sensores.

La ventana principal de MAX es un cuadro de diálogo en la que aparecen los errores cometidos y las líneas de comando en ejecución (fig. 1.3).

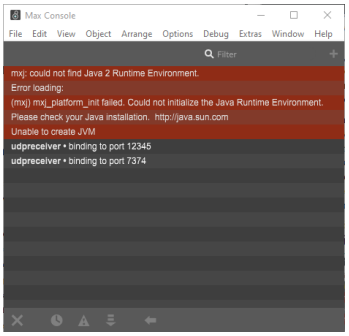

Figura 2.2: Ventana principal de MAX/MSP.

Los proyectos que se crean comienzan como una pantalla en blanco, con las herramientas distribuidas y organizadas en los margenes de esta:

- En el margen superior se encuentran los bloques funcionales básicos, como *buttons, switches, toggles y sliders*, entre otros bloques gráficos e interactivos.
- En el margen izquierdo se encuentran los bloques de programación disponibles separados en categorías: audio, imagen y vídeo.
- En el margen derecho se encuentran las herramientas de búsqueda de información y edición de propiedades de los objetos del módulo.
- En el margen inferior se encuentran las propiedades de la pantalla de edición del módulo.

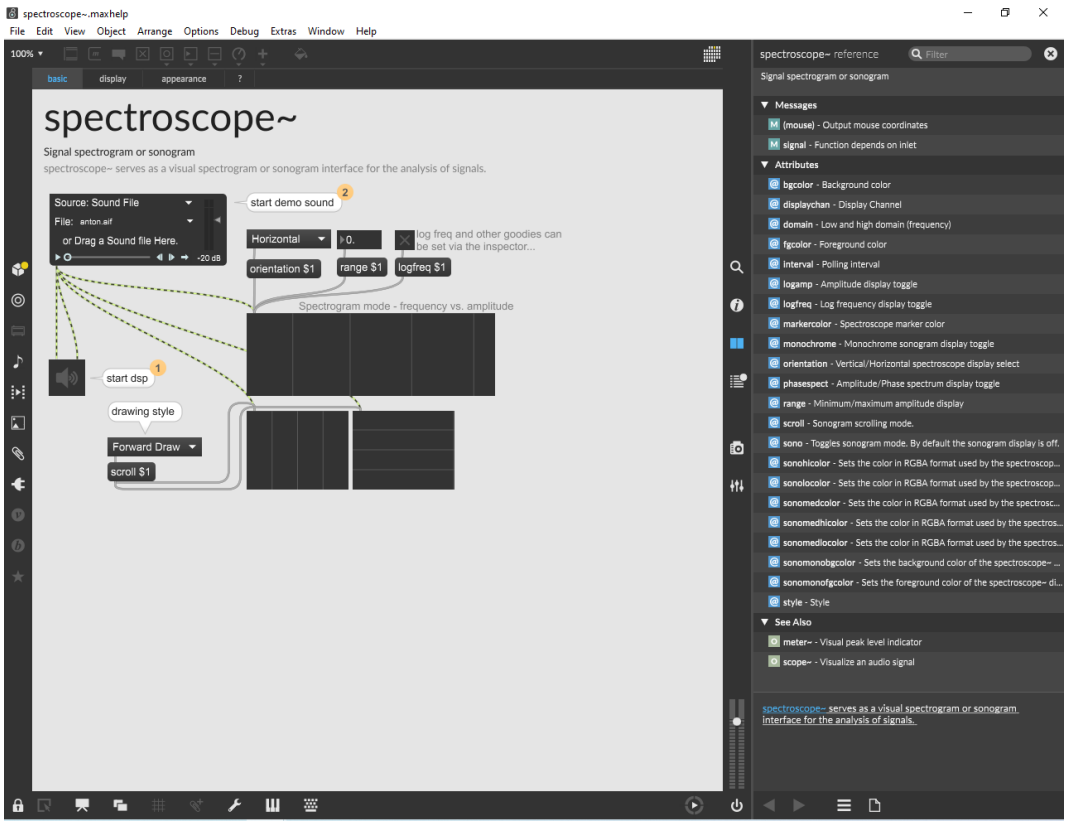

Figura 2.3: Ejemplo de un módulo de MAX/MSP.

Capítulo 3

Desarrollo del proyecto

3.1. Análisis y procesado de audio

3.1.1. Primeros Pasos con MSP

La herramienta MSP es la herramienta de MAX que se encarga principalmente del análisis, síntesis y modulación de señales de audio. Tiene una gran cantidad de objetos e instrucciones para poder llevar a cabo cualquier tarea relacionada con el audio. Estas se diferencian de las demás por acabar con un "~"[10].

Las primeras pruebas con esta herramienta consistieron en pruebas muy simples: generación de tonos, control de volumen, ecualización de la señal, puertas de ruido, etc [11]. Más adelante, la investigación comenzó a indagar en la aplicación de la transformada de Fourier rápida (FFT) a las señales y al análisis espectral de las señales y, después, se comenzó a dar forma a lo que iba a ser el módulo de análisis de audio.

3.1.2. De audio a vídeo

Las señales de vídeo son representadas como matrices de datos multidimensionales [13]. Una pantalla está conformada por muchos píxeles individuales que conforman los elementos de la imagen a mostrar, cada uno de estos se encarga de mostrar un color y, para representar imágenes de una manera satisfactoria, hay que tener un rango muy amplio de colores. Hay muchas formas diferentes de representar colores digitalmente, una forma estándar de describir el color de cada píxel en las computadoras es dividir el color en sus tres componentes de color diferentes: rojo, verde, azul y el componente de transparencia, conocido como alfa (RGBA). Para que cada celda de una matriz represente un píxel de color, cada celda debe contener cuatro valores numéricos (alfa, rojo, verde y azul), no solo uno. Por lo tanto, un fotograma de un vídeo se representa en *Jitter* como una matriz bidimensional, donde cada celda representa un píxel del fotograma y cada celda contiene cuatro valores que representan alfa, rojo, verde y azul en una escala de 0 a 255. En la figura 3.1 se puede observar un ejemplo de codificación RGBA.

Para mantener este concepto de varios datos por celda sin complicar demasiado la noción de las dimensiones de la matriz, *Jitter* organiza estas matrices con planos de datos. Al asignar memoria para los números en una matriz, *Jitter* necesita saber el alcance de cada dimensión y, también,

Figura 3.1: Segmento de espacio de color RGBA y su codificación

el número de valores que se deben mantener en cada celda. Para realizar un seguimiento de los diferentes valores en una celda, *Jitter* utiliza la idea de que cada uno existe en un plano separado. Cada uno de los valores en una celda existe en un plano particular, por lo que podemos pensar en un cuadro de vídeo como una matriz bidimensional de cuatro planos de datos intercalados. Organizando así las matrices es posible acceder a cada una de las componentes de color de una imagen de una manera más sencilla. En la figura 3.2 se muestra un ejemplo gráfico de la administración de planos de *Jitter*.

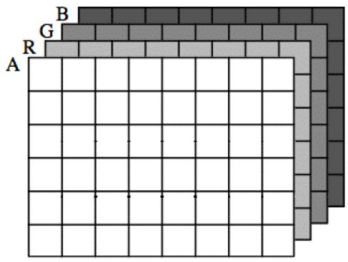

Figura 3.2: Planos en la matriz RGBA en *Jitter*

Al no tener las señales de audio y de video en un formato común, no se puede esperar que al hacer operaciones con ambas se obtenga un resultado deseable. Así pues, el primer paso es encontrar un formato común para ambas señales. Lo más lógico es convertir la señal de audio en una señal de vídeo, ya que la salida del módulo va a ser una señal de vídeo. Para ello, se utiliza el objeto jit.catch~ [14], el cual está compartido por las herramientas MSP y *Jitter*. Su función es convertir los datos aportados por la señal de audio a un formato de matriz. Los datos de señal de entrada recibidos por el objeto serán convertidos en un *array* de datos de *Jitter* y será enviado al exterior por la salida del mismo. El objeto especifica como de grande ha de ser el array generado, especificándolo en las dimensiones del *buffsize*.

Para este objeto hay 4 diferentes modos de capturar una señal y dependiendo de cual se utilice, esta se mostrará de una manera u otra. Estos modos de captación de señal son los siguientes:

- Modo 0: Devuelve toda los datos recogidos entre dos pulsos. Por ejemplo: si se obtienen 1024 muestras entre dos pulsos, el objeto devuelve una matriz de 1024 valores tipo *float32*, y si son 512 muestras, 512 valores, independientemente del *buffer* configurado.

- Modo 1: Genera a la salida una matriz bidimensional de datos, la que más se ajuste para la cantidad de datos obtenidos. Por ejemplo, si como en el caso anterior, se obtienen 1024

muestras entre 2 pulsos y el *buffer* tiene un tamaño de 100 valores, el objeto genera una matriz de 100x10, y los 24 valores sobrantes serán mostrados en el siguiente pulso.

- Modo 2: La salida del objeto son los datos más actuales que entraron en el *buffer*. Usando el ejemplo anterior de 1024 muestras y *buffer* para 100 valores, la salida del objeto serian las útimas 100 muestras recibidas.
- Modo 3: jit.catch ~ se comporta como un osciloscopio. Si no se ha superado el umbral durante todo el *buffer* de datos interno (100 ms), jit.catch~ vuelve temporalmente al comportamiento del modo 2 y genera los datos más recientes hasta que se vuelva a cruzar el umbral.

En la figura 3.3 se puede apreciar los 4 modos de funcionamiento del objeto jit.catch~.

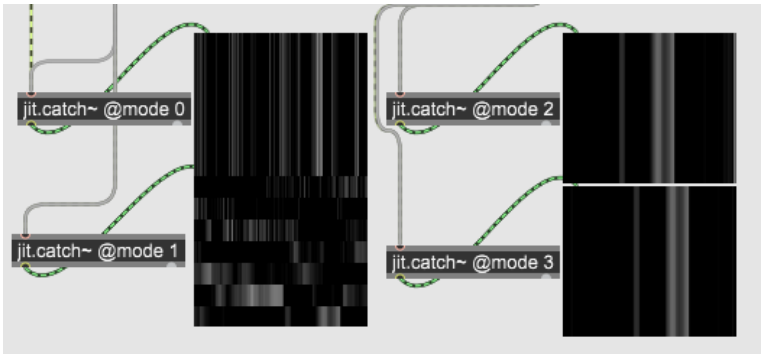

Figura 3.3: Modos de adquisición de audio y conversión a vídeo.

El modo que más se ajusta al análisis de la señal de audio es el modo 3 de jit.catch~, ya que ofrece los datos organizados, aunque no se descartan los demás modos por ahora, por si se da la necesidad de utilizarlos como texturas para aplicar a los objetos que se vayan generando. Por ahora, para el análisis de audio crudo, con este objeto y este modo será suficiente. Se captura una matriz unidimensional de 320 valores que representan la señal de audio y que puede utilizarse en la edición de vídeos e imágenes (fig. 3.4).

Figura 3.4: Adquisición de audio

En este bloque aparecen dos salidas, una es la señal de vídeo y otra es una textura basada en el audio capturado. La primera se trata como un vector de datos en el que basarse a la hora de generar gráficos que se basen en estos valores, y la textura se emplea con la finalidad de poder ver esta señal en la superficie de algún objeto virtual generado para su visualización.

3.1.3. Análisis espectral

La RAE define el *análisis* como la distinción y separación de las partes de algo para componer su composición. El oído humano se comporta como un analizador de tiempo-frecuencia. Cada una de ellas estimula una parte diferente de la cóclea, y cuando se percibe un sonido con una frecuencia predominante, esta vibra más en una zona determinada, y el cerebro puede clasificar esta información con una nota o una fuente de sonido concreta, pero el oído humano tiene limitaciones: cuando el sonido contiene múltiples frecuencias, y no hay ninguna aparentemente predominante, el oído no puede distinguir unas de otras, y esto se percibe como ruido [15].

El espectro de frecuencia de un sonido, o espectro sonoro, es el conjunto de frecuencias con diferentes amplitudes y fases que lo forman. El espectro sonoro de una onda se puede obtener matemáticamente con una herramienta llamada transformada de Fourier [16], que se emplea para transformar señales del dominio temporal al dominio frecuencial.

La transformada de Fourier es un análisis matemático frecuencial de las señales. Para poder aplicarla a una onda, esta onda ha de tener las siguientes características:

- Ha de ser periódica. Una onda periódica de periodo *T* para cualquier valor de *a* ha de cumplir lo siguiente (3.1):
$$f(x) = f(x+T) \tag{3.1}$$
- Ha de estar acotada. Todos los valores de la onda han de estar definidos dentro de un rango de números reales.
- En cualquier intervalo finito, ha de tener un número finito de máximos y mínimos locales y un número finito de discontinuidades.

Fourier consiguió demostrar que cualquier onda que cumpla estas condiciones, es la suma de múltiples ondas sinusoidales con frecuencias que son múltiplos de la frecuencia fundamental de la onda original.

Figura 3.5: Onda compleja (derecha) y ondas sinusoidales que la forman (izquierda)

Esto se demuestra matemáticamente con las series de Fourier [17] (3.2):

$$p(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty}[a_n * cos(2\pi * n * f_0 * t) + b_n * sin(2\pi * n * f_0 * t)] \tag{3.2}$$

Siendo t la variable que representa el tiempo, f_0 la frecuencia fundamental de la señal y a_n y b_n los coeficientes que le dan peso a las ondas básicas que forman la onda a estudiar. Para obtener estos coeficientes hay que realizar un producto interno entre la onda a estudiar y la señal básica que acompaña al coeficiente e integrar el resultado (3.3 y 3.4):

$$a_n = \frac{2}{T}\int_{-T/2}^{T/2} cos(2\pi * n * f_0 * t) * p(t)dt \tag{3.3}$$

$$b_n = \frac{2}{T}\int_{-T/2}^{T/2} sin(2\pi * n * f_0 * t) * p(t)dt \tag{3.4}$$

De esta manera se compara cuanto se parece la señal $p(t)$ a la onda sinusoidal de frecuencia $n * f_0$, devolviendo un escalar que será el coeficiente a utilizar.

Por ejemplo, en el caso de la figura 3.5, la onda compleja se obtiene de la siguiente suma de señales:

$$p(t) = 0 + [0 * \cos(2\pi * 1 * f_0 * t) + [4 * \sin(2\pi * 1 * f_0 * t) + \tag{3.5}$$
$$+[0 * \cos(2\pi * 2 * f_0 * t) + 2 * \sin(2\pi * 3 * f_0 * t)] + \tag{3.6}$$
$$+[0 * \cos(2\pi * 3 * f_0 * t) + 1 * \sin(2\pi * 3 * f_0 * t)] + \tag{3.7}$$
$$+[0 * \cos(2\pi * 4 * f_0 * t) + 0{,}5 * \sin(2\pi * 4 * f_0 * t)] \tag{3.8}$$

$$p(t) = [4{*}sin(2\pi{*}f_0{*}t){+}2{*}sin(2\pi{*}2{*}f_0{*}t){+}1{*}sin(2\pi{*}3{*}f_0{*}t){+}0{,}5{*}sin(2\pi{*}4{*}f_0{*}t)] \tag{3.9}$$

Al igual que se pueden formar ondas muy complejas a partir de ondas sinusoidales, también se pueden obtener las señales básicas a partir de las señales complejas. Para esto vale la transformada de Fourier (3.10).

$$X(f) = \int_{-\infty}^{\infty} x(t) * e^{-j*2\pi*f*t}dt \tag{3.10}$$

Esta transformada es una correlación entre 2 señales: la señal a estudiar y la señal compleja $c(t) = e^{-j*2\pi*f*t}$, se trata de una exponencial compleja, periódica de frecuencia f. Si se fija la frecuencia, la señal a estudiar se compara con la señal exponencial que corresponde a esa frecuencia. Si se parece, da un valor alto en esa frecuencia, indicando que la señal de estudio tiene una componente muy fuerte en la frecuencia fijada. Barriendo un rango grande de frecuencias, se puede obtener el espectro de frecuencias de la onda analizada.

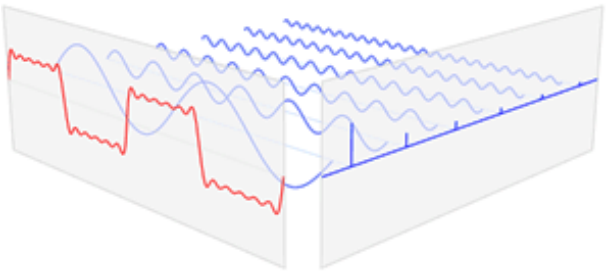

Figura 3.6: Transformada de Fourier.

En la figura 3.6 se puede observar la correlación de una onda en su dominio temporal y en su dominio frecuencial, a los cuales se puede llegar con la transformada de Fourier y la antitransformada de Fourier. Aquí se muestra la onda original (roja, izquierda) descomponiéndose en todas las señales primitivas que la componen (azul, centro) y el resultado de la transformada de Fourier, que es la amplitud de cada onda fundamental en la frecuencia a la que se encuentran (azul, derecha).

Por lo general, los instrumentos emiten señales muy complejas, plagadas de frecuencias armónicas y con más de una frecuencia fundamental, si se toca mas de una nota en ellos. Estos armónicos definen el timbre del instrumento o de la fuente sonora y el oído humano no puede distinguir frecuencias más allá de la frecuencia fundamental de las notas que se están tocando. De hecho, cuando el oído humano, cuando escucha ruido no puede distinguir notas ya que el ruido no tiene una frecuencia fundamental. En estos casos, hay que recurrir a una solución visual para facilitar el análisis del espectro de frecuencias.

El espectrograma [18] es una herramienta visual capaz de mostrar cómo varía el espectro de frecuencias de una señal a lo largo del tiempo. Los espectrogramas son herramientas muy utilizadas en música para el estudio de frecuencias, pero también se utiliza en los campos de estudio de radares, sonares, procesamiento digital de voz, sismología e incluso comunicación animal. Representa tres dimensiones del espectro frecuencial: tiempo, frecuencia y amplitud de la distribución de energía de la señal. La representación del espectro de una señal en el dominio frecuencial puede ayudar a entender mejor su contenido que una representación de la señal en el dominio temporal.

En la figura 3.7 se puede observar el espectro de frecuencias de un violín, el cual muestra las frecuencias (eje 0Y) y las amplitudes de las frecuencias que el violín ha generado (brillo) durante un tiempo (eje 0X)

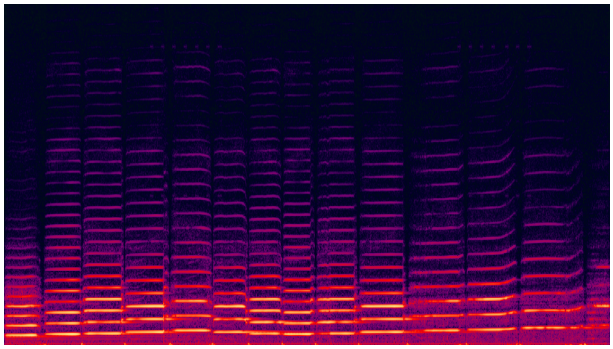

Figura 3.7: Espectrograma de un violín

Para comenzar a desarrollar esta herramienta en MAX, el primer objetivo fue crear un analizador de espectro bidimensional, en el cual solo se muestra la amplitud y la frecuencia a tiempo real. En la figura 3.8 se muestra el actual espectrograma del proyecto Soundcool, el cual representa de esta manera el espectro de frecuencias.

Se utilizó el bloque pfft~ de la herramienta MSP, el cual está diseñado para simplificar el procesamiento de audio espectral utilizando la transformada rápida de Fourier (FFT) [19]. Además de realizar la FFT y la transformada rápida inversa de Fourier (IFFT), pfft~ (con la ayuda de sus objetos complementarios fftin~ y fftout~) gestiona el enventanado de la señal, la superposición y la adición necesarias para crear un sistema de análisis en tiempo real basado en FFTs.

Los segmentos de audio a analizar se guardan en vectores de datos de 512 valores, a los cuales se les aplica el enventanado necesario para que no se den discontinuidades al principio y al final de las secciones. Para realizar el espectrograma 2D solo es necesaria la amplitud de la señal, no es necesaria la fase. Tras aplicar la FFT, los valores obtenidos de manera cartesiana se descomponen a notación polar, y la fase se descarta. La salida directa de este bloque fuerza a la aplicación de la IFFT, con lo que la solución planteada fue guardar los valores dentro de un objeto buffer~. A

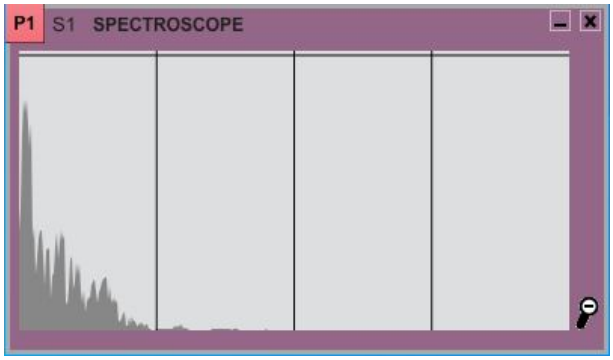

Figura 3.8: Espectrograma frecuencia_amplitud.

este objeto se puede acceder desde fuera del bloque, con lo que se pueden obtener los valores de amplitud de la transformada desde fuera del bloque pfft~.

Siguiendo el ritmo de un contador externo, se generan y se leen las muestras recogidas dentro del bloque pfft~, pero solo hasta la mitad del buffer~, los primeros 256 valores. Las funciones FFT son simétricas, lo que significa que presenta frecuencias negativas que existen como propiedades matemáticas de la misma transformada. La primera mitad de la salida de FFT contiene frecuencias desde 0 Hz hasta la frecuencia de Nyquist [20], que viene a ser $f_{nyq} = f_s/2$, y la segunda mitad es una imagen con las frecuencias negativas.

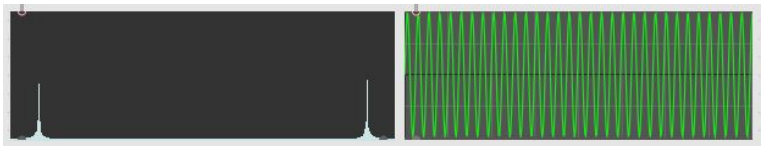

Figura 3.9: Espectro de frecuencias de 0 Hz a f_0 para sinusoide de 3kHz.

Por esta razón se muestrea al doble de la frecuencia audible máxima y solo se recogen los datos de la primera mitad del buffer~.

El espectrograma diseñado es capaz de representar el espectro de manera lineal y de manera logarítmica. Para ajustarse a la ventana en la que se vaya a visualizar, el espectrograma se basa en la altura de esta y adapta la distribución de las frecuencias de la primera mitad del buffer~ a las dimensiones de la pantalla. Debido a que la representación está discretizada en 256 valores, el espectro también está discretizado, y se muestra en rangos de frecuencias. Cada uno de estos rangos viene dado por esta ecuación (3.11):

$$f(x) = \frac{x}{256} \tag{3.11}$$

Siendo x los intervalos de la frecuencia discretizada y $f(x)$ la posición del rango en la pantalla. Los intervalos aumentan en 82.03 Hz de manera lineal.

La visualización logarítmica del espectrograma tiene como finalidad enfocar la atención del usuario a la información que aportan las bandas de frecuencia más graves. Debido a que el análisis espectral ya se ha realizado de manera lineal, es necesario alterar la manera en la que este se representa en la pantalla, procurando que las bandas más graves ocupen mas espacio en la pantalla.

Para ello se utiliza una ecuación exponencial con un rango de valores de 0 a 1 para la representación del espectro de frecuencias. La ecuación hallada ha sido la siguiente (3.12)

$$f(x) = \frac{10^{\frac{x}{427,0314} - 1,60205}}{10000} \tag{3.12}$$

Siendo x los intervalos de la frecuencia discretizada y f(x) la posición discretizada en la pantalla. Los intervalos siguen aumentando en 21.4843 Hz de manera lineal, pero con esta función, los intervalos más graves ocupan más sitio en la pantalla.

En las figuras 3.10, 3.11, 3.12 y 3.13 se muestran los bloques que han conformado el módulo generador del espectrograma bidimensional.

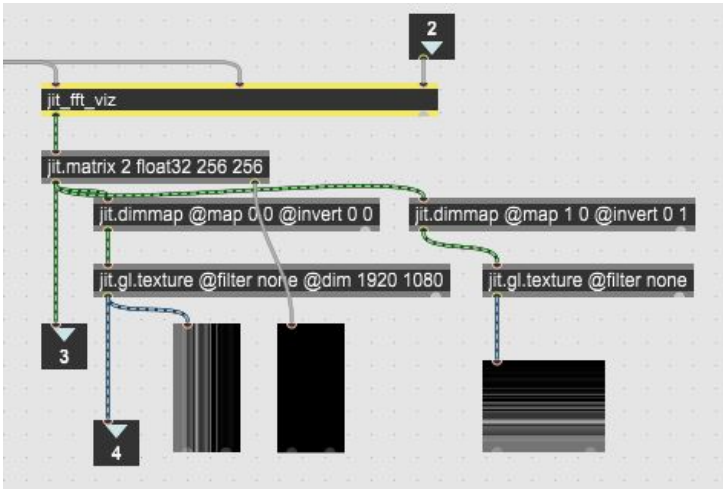

Figura 3.10: Módulo Espectrograma frecuencia-amplitud.

El espectrograma bidimensional ya está definido. El siguiente paso es añadirle el tiempo al espectrograma para representar una nueva dimensión. Actualmente, el espectrograma representa la frecuencia con líneas rectas paralelas a la base de la gráfica, definidas por la posición en la pantalla, y la amplitud de esta según el brillo de la línea. Los valores que toma la amplitud están normalizados: van de 0 a 1 y los valores toman el formato de float32, con lo que adquieren una buena precisión decimal. Añadirle la dimensión de tiempo al espectrograma implicará poder visualizar la evolución temporal de las frecuencias.

Para ello es necesario un *buffer*, en el cual se almacenan las muestras que aparecerán por la pantalla con el mismo orden por el que han entrado en él. Para este fin, se utilizará el objeto jit.gl.slab [21], este proporciona una interfaz optimizada para realizar operaciones gráficas basándose en la GPU. Este bloque utiliza el lenguaje de programación GLSL [22], el cual es un lenguaje nativo de OpenGL [23] para la programación de *shaders* [24] (véase apartado 3.2.3). La función de este objeto es la de guardar y devolver a cada momento el estado de la pantalla, lee la pantalla como si fuera la textura de un *shader*, y la envía también como si fuera una textura, para que otro bloque pueda leerla y mostrarla cuando sea necesario. Este objeto se llama jit.gl.pix [25], y es uno de los objetos más utilizados en este proyecto. Dicho objeto, al igual que el jit.gl.slab, se emplea para el procesamiento de píxeles utilizando la GPU, pero este, a diferencia de jit.gl.slab, utiliza el lenguaje de programación *GEN patcher* el cual es nativo de MAX. Este bloque tiene la función de leer la información del *buffer* y del espectrograma 2D cuando es necesario.

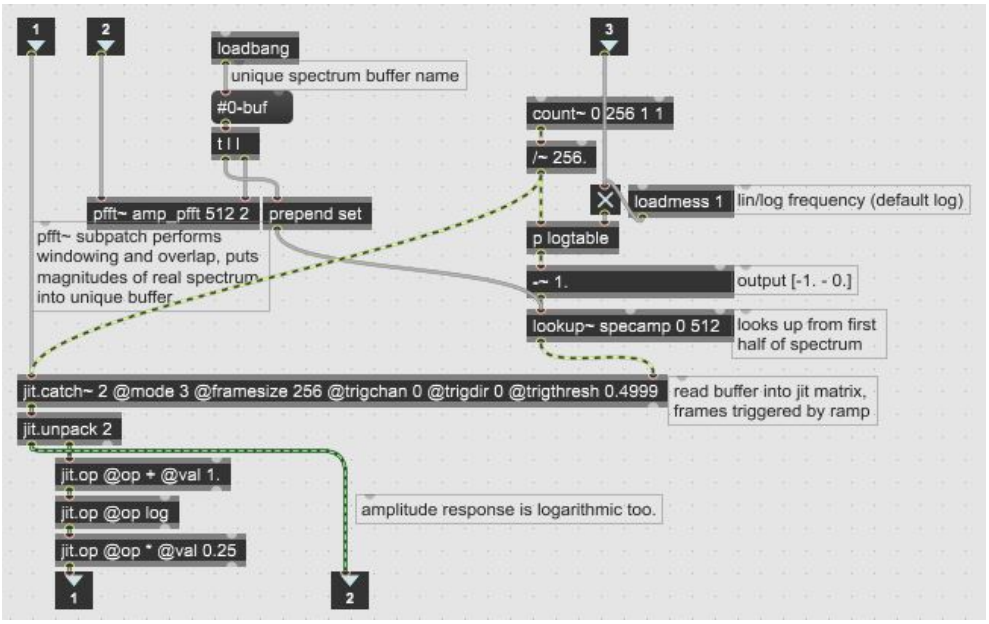

Figura 3.11: Bloque jit fft viz.

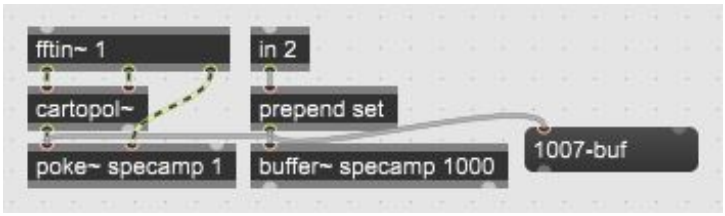

Figura 3.12: Bloque pfft amp ppft.

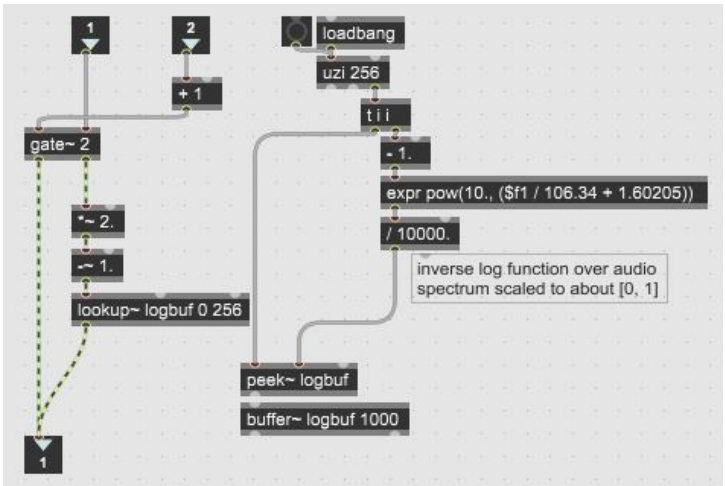

Figura 3.13: Bloque p logtable.

En la figura 3.14 se muestra el diagrama del algoritmo para añadir al espectrograma la dimensión del tiempo.

Así queda un espectrograma que da la ilusión de que se desplaza en el tiempo mientras evoluciona el espectro de frecuencias. El resultado se puede observar en la figura 3.15.

A diferencia de la obtención del audio, para este submódulo solo se ha enviado al exterior el espectrograma a modo de textura, ya que no se van a utilizar los datos de este para modificar formas. El fin que tiene la obtención del espectrograma era la de representarlo en los gráficos a generar, no en realizar más módulos de análisis de audio con él (fig. 3.16).

3.1.4. Adquisición de pulsos

El ritmo es un patrón fuerte, regular y repetitivo de sonidos o movimientos. En música, el ritmo se define por la organización de los pulsos (unidad de tiempo en la música) y de los acentos (el énfasis de cada pulso) en la estructura de la canción. Su influencia en la interpretación de la música: puede determinar el movimiento físico y el estado de ánimo del oyente [26]. Esta información es útil para la generación de gráficos audio reactivos, ya que al reaccionar con el ritmo de una canción, da la impresión de que los gráficos van acompasados, que "bailan" con la música.

Para hacerlo más genérico y simple, el análisis del ritmo se hace sobre la pista de audio a tiempo real: se analizará mientras suena, por lo que no hace falta un análisis previo de las pistas, ni una base de datos con los pulsos por minuto (*beats per minute* o BPM) de una serie de canciones para poder funcionar. Por otra parte tiene la desventaja de que necesita un breve tiempo para poder deducir el ritmo de la pieza y según lo complejo que sea, puede no ser extremadamente preciso, aun así reacciona correctamente a las pulsaciones de la canción. El algoritmo para procesar la señal sigue el guión representado en la figura 3.17.

Las frecuencias graves son las que aportan el ritmo y la melodía básica a una canción, con lo que el primer paso para poder analizar el ritmo es aislar estas frecuencias graves del resto de sonidos. Para ello se emplea un filtro paso bajo (LPF), el cual filtra la canción y evita que pasen frecuencias altas (superiores a 200Hz) a la siguiente fase, dejando a la señal solo con las componentes más graves, que son las que se estudian para definir el ritmo de la canción. En la figura 3.18 se muestra el espectrograma de una canción y su resultado tras el paso por el filtro paso bajo.

Una vez aisladas las frecuencias graves de la onda, se estudia la evolución de la onda envolvente de la señal [27] (fig. 3.19). La onda envolvente de una señal es la función definida por los máximos alcanzados por la señal. Para poder hallarla, primero hay que obtener el valor absoluto de la señal, y luego realizar un promedio entre estos valores para obtener una señal envolvente clara. La señal que se está analizando es la señal que acaba de pasar por el LPF, con lo que la señal envolvente representa la evolución de las frecuencias graves, lo que quiere decir que lee cuando se dan las pulsaciones que marcan el ritmo. Cuando la pendiente del filtro es muy elevada y su amplitud muy grande, entonces es el momento en el que se da un golpe que tiene una alta probabilidad de repetirse de forma periódica durante la canción, esta es la señal que delata su ritmo.

El estudio de la onda envolvente ya de por sí ofrece un seguimiento del ritmo bastante fiel, pero puede darse el caso de que la base rítmica sea muy compleja y que el estudio de la envolvente arroje demasiados datos, llegando a confundir las pulsaciones por minuto de la canción con el ritmo que lleve la batería con respecto a este. Para poder hacer más fiable la obtención de las pulsaciones por minuto, se compara un segmento previo de la canción con otro segmento de la misma magnitud. Al

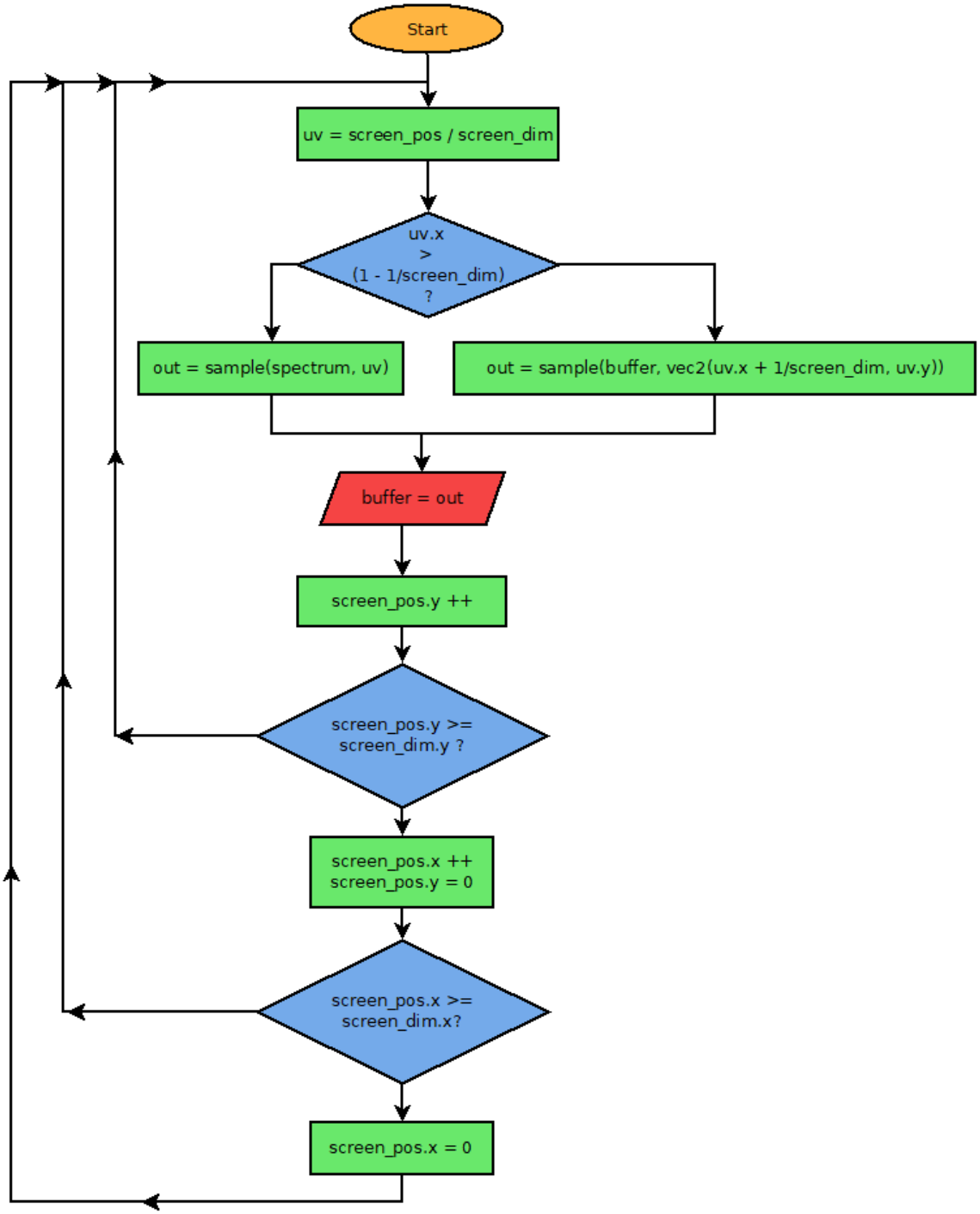

Figura 3.14: Algoritmo del espectrograma tridimensional.

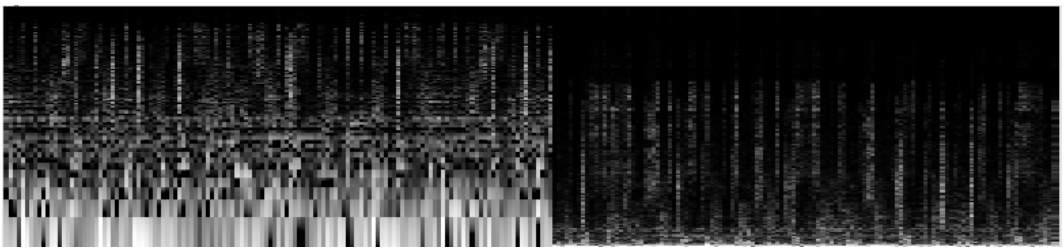

Figura 3.15: Espectrograma obtenido. Modo logaritmico (izquierda) y lineal (derecho).

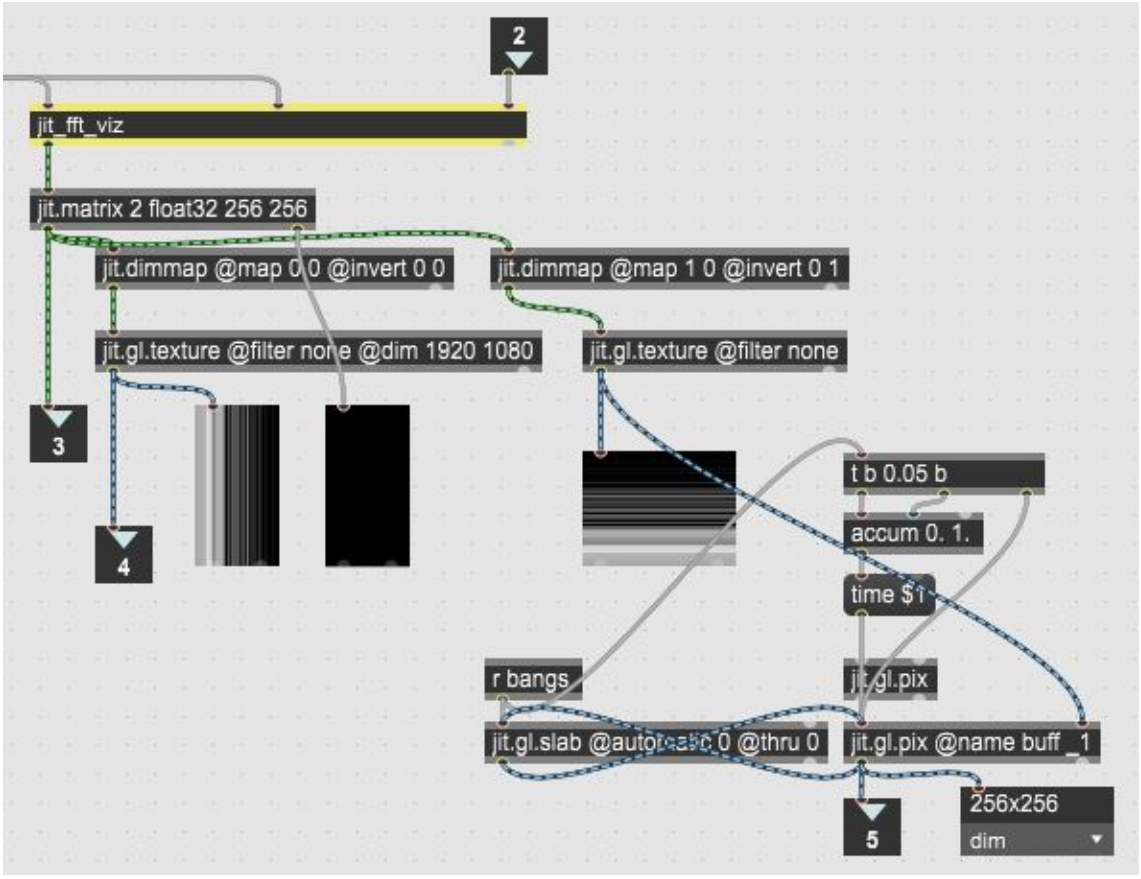

Figura 3.16: Espectrograma tiempo-frecuencia-amplitud.

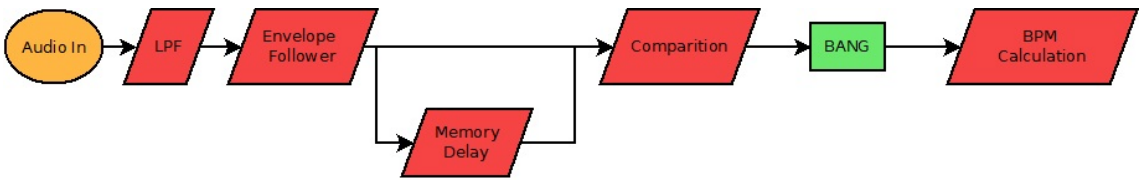

Figura 3.17: Algoritmo para el seguimiento del ritmo.

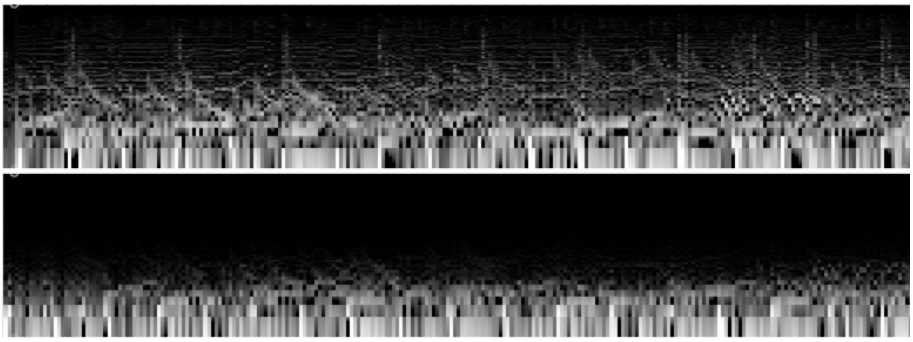

Figura 3.18: Espectrograma de la señal original (arriba) y espectrograma de la señal filtrada (abajo).

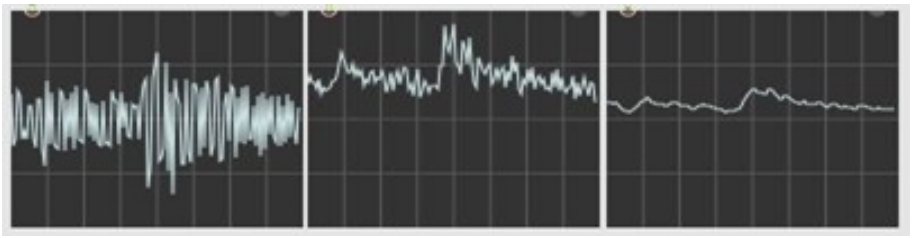

Figura 3.19: Obtención de envolvente: Señal original (izquierda), valores absolutos (centro) y señal envolvente (derecha)

compararlos, se puede observar donde coinciden los golpes más importantes, descartando así todos los arreglos que se hayan hecho y quedando solo las pulsaciones fundamentales que conforman las pulsaciones por minuto de la canción.

Para indicar que ha habido una coincidencia, se ha programado un impulso, un *bang* en la terminología de MAX. De esta manera se obtiene una señal periódica y fiable de las pulsaciones por minuto. El siguiente paso es obtener un número que respalde estas pulsaciones, para conocer exactamente el tempo y poder hacer arreglos con relación a este. Para ello se ha utilizado un bloque llamado *timer*, el cual mide el tiempo entre pulsos en milisegundos. A partir de aquí, obtener las pulsaciones por minuto es tan sencillo como realizar una regla de tres, pero para las aplicaciones que hay previstas, con conocer el tiempo medio que hay entre pulsos es más que suficiente.

Este planteamiento para la obtención del ritmo es comprimido en el bloque de MAX llamado op.beatitude~ [28]. Este bloque, creado por Olivier Pasquet, funciona con la entrada de audio y la entrada del tamaño del *buffer* de memoria para comparar las envolventes de las señales. El tamaño de la memoria define el ritmo máximo que puede obtener el bloque. Cuanta más memoria tenga, podrá detectar ritmos más lentos. Tras muchas pruebas, se llegó a la conclusión de que una memoria de 100 ms funciona bastante bien con un buen rango de canciones, pero a veces falla y requiere un valor de memoria mayor. Debido a que el programa tardaría muchísimo tiempo en detectar por sí mismo el tamaño adecuado de la memoria, se le ha dado la oportunidad al usuario de que modifique este tamaño, por si el programa no reacciona bien a alguna canción.

El estudio de las pulsaciones por minuto está destinado al control del tamaño de los objetos a generar en vídeo, con lo que un número estático y un impulso periódico no van a ser de mucha ayuda. Por ello se planteó hacer señales periódicas con la misma frecuencia que las pulsaciones para poder controlar los objetos de una manera fluida y coordinada. Para ello se realizaron 4 tipos

de ondas: rampa, rampa inversa o negativa, sinusoidal y cuadrada (fig. 3.20).

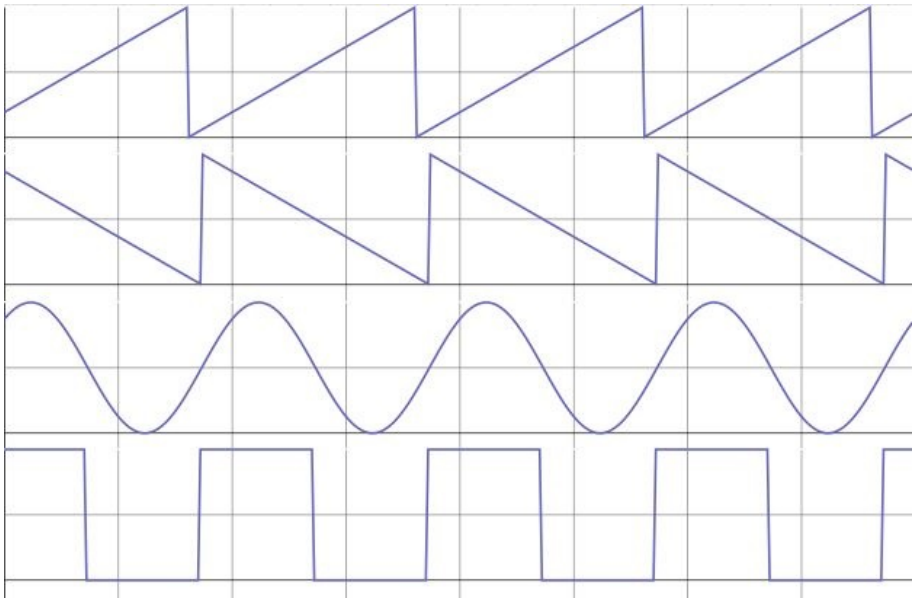

Figura 3.20: Ondas a generar: rampa ascendente, rampa descendente, sinusoidal y cuadrada

La primera onda a generar fue la rampa. El planteamiento fue sencillo: realizar un contador de 0 a 1 discretizado. Para ello se recurrió al bloque *counter*, el cual se configuró para que contara a la frecuencia de generación de vídeo (30 Hz) de manera ascendente hasta llegar al valor en ms que había entre cada pulso. Para mantener la correlación entre la cuenta y el tiempo entre pulsaciones el valor máximo del contador tenía que ser de $t_{pulse}/33ms$. Una vez obtenida la cuenta, se normalizan los datos obtenidos por el contador dividiendo el resultado de la cuenta entre el tiempo entre pulsaciones. De esta manera se obtiene la rampa positiva.

La generación de la rampa negativa es muy sencilla una vez realizada la rampa positiva: no hay más que restar a 1 el valor de la rampa positiva para invertir su pendiente.

Una vez obtenida la rampa positiva, la generación de la onda sinusoidal se efectúa realizando la función seno de amplitud 1 con el valor de la rampa como frecuencia.

La onda cuadrada fue también muy simple de realizar a partir de las anteriores. Al ser una función a trozos, no hubo más que definir en qué momentos obtendría su valor positivo y su valor negativo: cuando el valor de la rampa positiva fuera mayor o igual a 0.5, esta valdría 1, y cuando fuera menor, esta valdría 0.

Las señales de control estaban hechas, pero su percepción era demasiado brusca. Para evitarlo, se puso un control de amplitud manual, con el fin de que el usuario controlase manualmente la amplitud de la oscilación de las ondas y poder ajustar los límites de variación de los objetos. De esta manera no se limita al usuario a las opciones de la máquina, sino que puede hacer lo que quiera con el control del tamaño de los objetos.

Una vez acabado el control de la amplitud de las señales, el submódulo de adquisición de pulsos está acabado (fig. 3.21), y con ello, el módulo de análisis de audio.

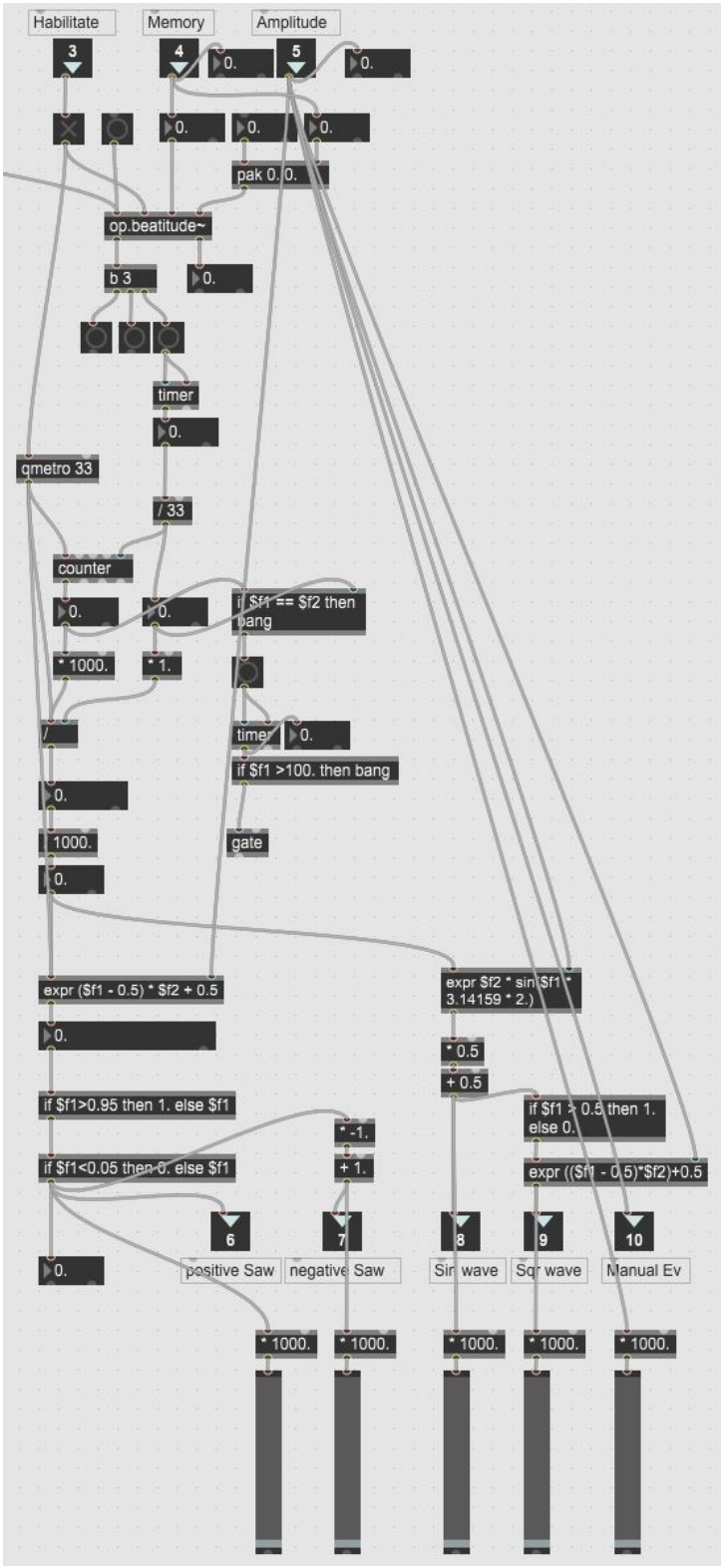

Figura 3.21: Bloque de adquisición de pulsos

3.2. Generación de vídeo

En este apartado se explica las maneras con las que MAX permite la edición y generación de vídeo y gráficos, qué posibilidades se plantearon y cual fue el camino seguido para llevar a cabo cada uno de los submódulos.

3.2.1. Primeros Pasos con *Jitter*

A la hora de trabajar con vídeo en MAX, hay que comenzar explorando la herramienta *Jitter*, la cual contiene todos los bloques funcionales necesarios para poder reproducir, editar y generar imágenes y vídeos [29, 30].

Al igual que los objetos pertenecientes a MSP se diferencian de los demás por llevar un guión al final del nombre de estos(“ ~”), los objetos de *Jitter* se diferencian de los demás por empezar con el nombre de su librería “jit”.

La primera prueba que se hizo con esta herramienta fue un simple reproductor de vídeo. Para ello fue necesario un archivo de vídeo, un medio para reproducirlo y un medio para visualizarlo. El bloque que sirvió como medio para poder reproducir el archivo de vídeo fue el bloque “jit.movie”. Este bloque lee el archivo de vídeo, y emite la imagen leída a ritmo de un tren periódico de pulsos. Por ello, este objeto necesita de un metrónomo que le indique, la frecuencia a la que ha de emitir las imágenes recibidas. Este objeto se llama“qmetro”, y se conecta al bloque “jit.movie”para poder reproducir el vídeo de manera continua. Por último, para poder visualizar el vídeo, en un principio había dos posibles bloque a utilizar: “jit.window“ y “jit.pwindow”. La diferencia entre estas dos ventanas radica en que “jit.window”genera una ventana nueva al entorno de desarrollo y “jit.pwindow”genera una ventana embebida en el entorno. Visualizan el contenido de la misma manera y salvo esos dos detalles mencionados anteriormente, tienen las mismas características. Según los casos, se puede utilizar una u otra. El resultado fue el siguiente (fig. 3.22):

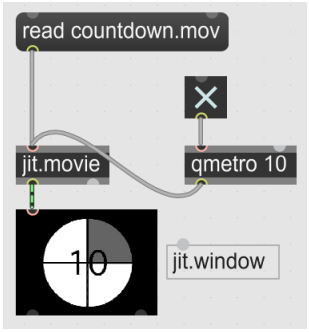

Figura 3.22: Reproductor básico de vídeo

A partir de este módulo se hicieron más pruebas con análisis y edición de vídeo, pero en cuanto se cogió un poco de soltura sobre el terreno, se pasó rápidamente a la generación de objetos con el fin de avanzar en el proyecto.

3.2.2. Primeros módulos con OpenGL

En *Jitter*, la generación de gráficos se realiza a partir de librerías de OpenGL. OpenGL (fig. 3.23) es principalmente considerado una interfaz de programación de aplicaciones (API) que proporciona un gran conjunto de funciones que se pueden usar para manipular gráficos 2D, 3D e imágenes. Sin embargo, OpenGL por sí solo no es un API, sino una especificación, desarrollada y mantenida por el Grupo Khronos, empresa desarrolladora.

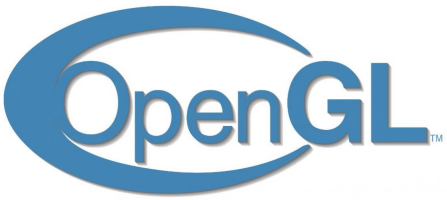

Figura 3.23: Logo de OpenGL.

Una especificación de un lenguaje de programación es la documentación que define un lenguaje, con el fin de que los usuarios y los programas implementadores puedan acordar qué significan los programas en ese lenguaje. OpenGL especifica exactamente cómo debería ser el resultado de cada función y cómo debería funcionar. Los desarrolladores son los que implementan estas especificaciones, con las que se determinan las funciones del lenguaje.

Actualmente, son los fabricantes de GPUs y tarjetas gráficas los que se encargan del desarrollo de OpenGL. Cada tarjeta gráfica disponible en el mercado está configurada con una versión específica de OpenGL, desarrollada para esa serie de tarjetas gráficas.

Antiguamente, OpenGL desarrollaba gráficos en modo inmediato (también conocido como *fixed function pipeline*) [31]. Este método era muy fácil de utilizar, pero mantenía la mayor parte de la funcionalidad de OpenGL oculta en las librerías. Los desarrolladores se dieron cuenta del potencial que tenía y al no tener completo control sobre cómo se desarrollaban los programas que generaban, exigieron a los fabricantes y desarrolladores de GPUs más flexibilidad para las funciones de OpenGL. Sus demandas fueron satisfechas, y los desarrolladores obtuvieron más control sobre sus gráficos. El modo inmediato era muy sencillo para aprender a manejar OpenGL, pero era extremadamente ineficiente. Por ello, en la versión 3.2 de OpenGL, este modo de desarrollo fue descartado. Actualmente se utiliza el desarrollo en modo *core-profile*, el cual es una versión que eliminó todo rastro de funcionalidad obsoleta de OpenGL.

Además de trabajar con datos de vídeo, *Jitter* también proporciona una interfaz para OpenGL. En lugar de procesar vídeo directamente, OpenGL renderiza imágenes de una escena que contiene objetos, texturas e iluminación. Los objetos que utilizan OpenGL son aquellos que contienen en su nombre el prefijo ´´.gl.”, como jit.gl.world [32] y jit.gl.gridshape [33].

A la hora de empezar con la generación de gráficos, se comenzó con la generación de objetos tridimensionales. A la hora de trabajar con OpenGL, *Jitter* hace que generar objetos en 3D sea más sencillo que generar gráficos en 2D [35].

El primer paso es definir el espacio en el que se van a generar los objetos. Para ello se utiliza el bloque jit.world. Este objeto encapsula la funcionalidad de diversos objetos de *Jitter* con OpenGL:

jit.window, jit.gl.render, jit.gl.node, jit.gl.cornerpin y jit.phys.world. Define el entorno vacío: sin objetos, sin iluminación y sin fuerzas físicas. Genera un espacio en el que solo se puede apreciar un color gris sin fondo o dimensiones. Con los atributos de este bloque se consigue modificar la posición, el tamaño, la visualización de la pantalla, el color de fondo (predeterminado: gris) y muchos parámetros más, pero todos en relación con la pantalla y la visualización de esta.

El siguiente paso es generar un objeto tridimensional para que aparezca en el entorno. Para ello se utilizó el objeto jit.gl.gridshape, que crea una forma sencilla predefinida en el programa (esfera, toroide, cilindro, cilindro abierto, cubo, cubo abierto, plano o círculo) dispuesta en una cuadrícula conectada o malla (fig. 3.24). Estas formas están unidos al entorno generado por el nombre: el objeto jit.world tiene un nombre que o bien define el desarrollador o bien lo define la máquina, y el objeto jit.gl.gridshape crea la forma en el mundo al que este llame, y son los parametros de jit.gl.gridshape los que definen su posición, orientación, tamaño, color, forma, dimensiones de mallado e iluminación por defecto, entre muchos otros parámetros. Un objeto que es muy interesante utilizar con jit.gl.gridshape es jit.gl.handle [35], el cual permite la modificación de la posición y de la orientación del objeto con el ratón, con el fin de examinar la figura de una manera inmediata.

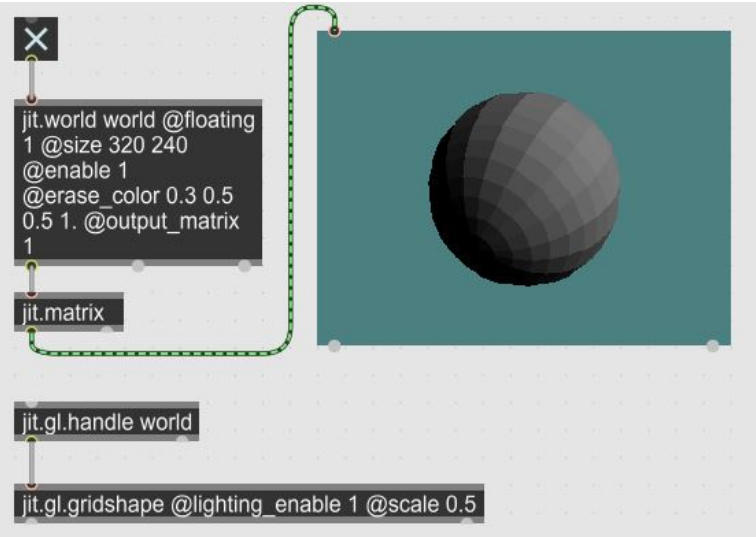

Figura 3.24: jit.world y jit.gl.gridshape

Se ve en la figura 3.24 que la esfera tiene una renderización bastante pobre, quedando un resultado poligonal. Esto se puede solucionar aumentando la resolución de polígonos y con el atributo *smooth_shading* de jit.gl.gridshape. De este modo se puede conseguir que la esfera luzca como tal, a pesar de seguir estando formada por polígonos. Otro objeto interesante es el objeto jit.gl.camera [36], el cual permite la modificación de la posición y orientación de la cámara (fig. 3.25).

A estos objetos se les puede aplicar texturas. Con el objeto jit.gl.texture [37] se pueden generar texturas a partir de matrices. Esto significa que en un objeto se puede plasmar una imagen o incluso un vídeo. En el apartado de análisis de audio se generaron texturas a partir del audio convertido en vídeo con el fin de poder plasmarlo en estos objetos (fig. 3.26).

También se puede recurrir a texturas realistas para las superficies de las figuras. Un objeto muy útil para esto es el objeto jit.gl.material [38], el cual permite la aplicación de texturas realistas a las figuras, e incluso importar texturas diseñadas con otros o para otros programas(fig. 3.27).

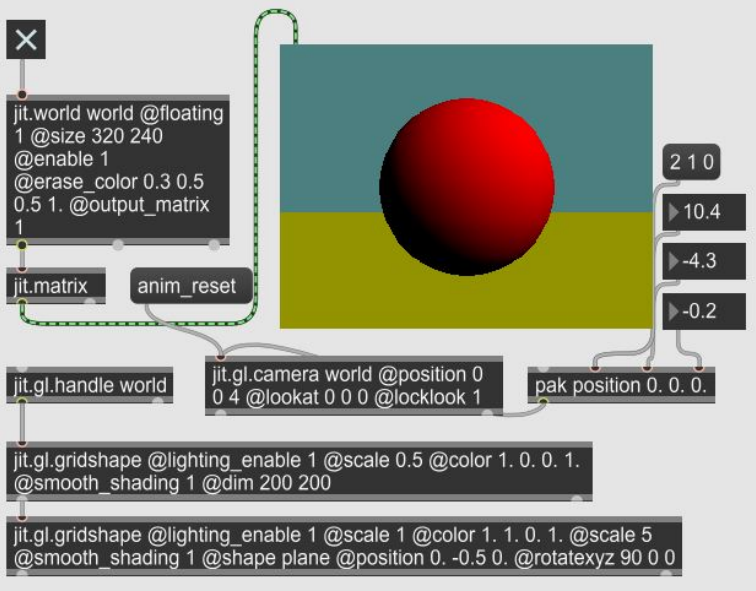

Figura 3.25: jit.world, jit.gl.gridshape y jit.gl.camera

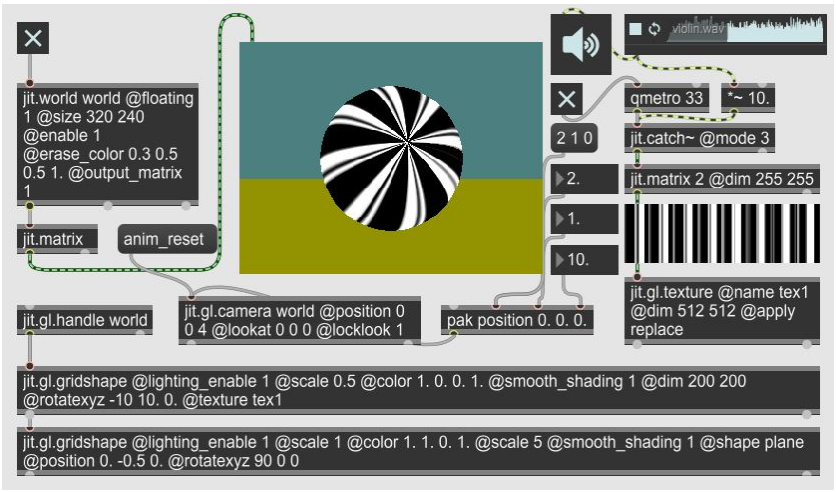

Figura 3.26: jit.gl.texture y adquisición de audio

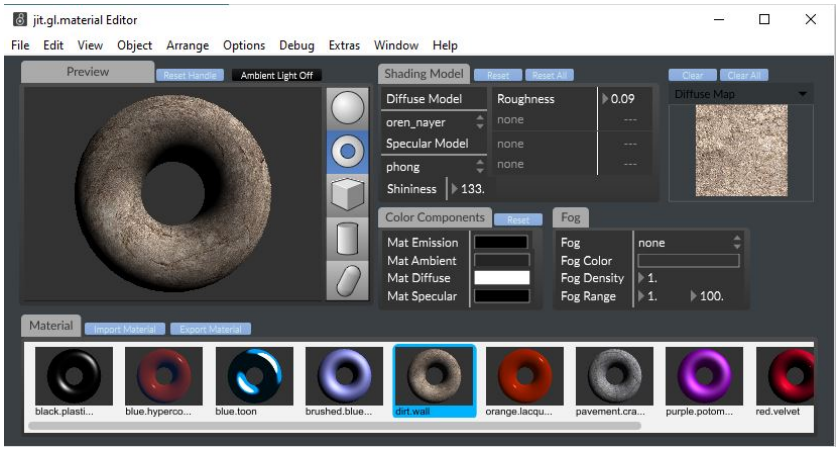

Figura 3.27: jit.gl.material

Las fuentes de luz predeterminadas son fuentes direccionales: focos situados en el infinito que generan haces de luz paralelos. Con el objeto jit.gl.light se pueden cambiar los parámetros de la luz, como cambiar la posición de la fuente a una más próxima o cambiar el enfoque para que no sea omnidireccional y que imite el comportamiento de una linterna o una lámpara. En la figura 3.28 se ha forzado la posición de la luz a la posición de una pequeña esfera que orbita alrededor de la esfera original.

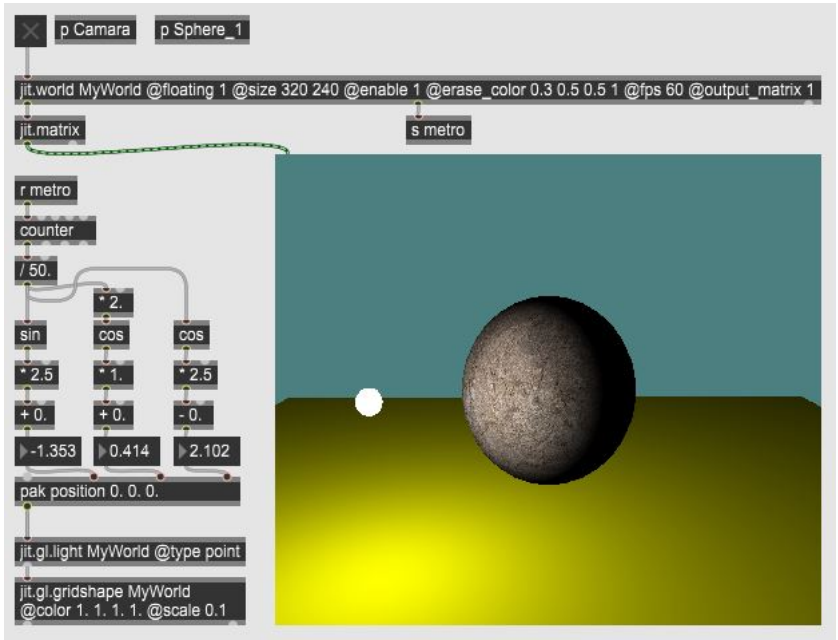

Figura 3.28: jit.gl.light

Por medio de mensajes se pueden modificar todos los parámetros de cualquier objeto físico, hasta con audio, si se transforma adecuadamente de matriz de *Jitter* a matriz de datos, por medio del objeto snapshot~, el cual transforma los valores que recibe de una señal de audio en datos en coma flotante.

En la figura 3.29 se utilizó jit.gen, para representar el espectrograma en tres dimensiones (espaciales). Para ello hubo que generar un plano de coordenadas normalizadas en el plano espacial XZ y alterar las dimensiones del eje 0Y con los valores obtenidos en la matriz de datos del espectrograma de dos dimensiones (fig. 3.29).

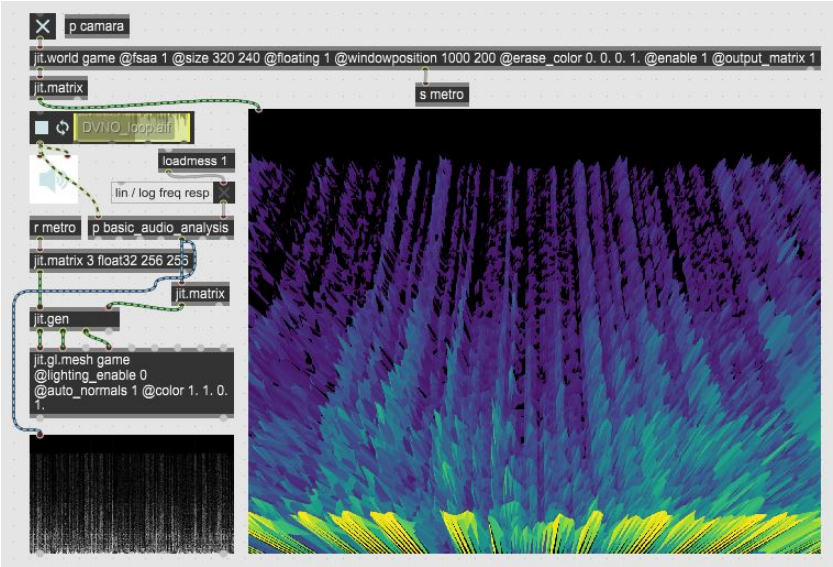

Figura 3.29: Espectrograma 3D (Primera versión)

El último módulo que se realizó siguiendo esta metodología fue uno en el que se creaba una cantidad masiva de objetos, para que reaccionaran a la música y entre todos ello crearan ondas con su movimiento. Para crear una gran cantidad de objetos, se utilizó en objeto jit.gl.multiple [39], el cual puede realizar múltiples copias de un objeto y organizarlas en el espacio como se desee (fig. 3.30).

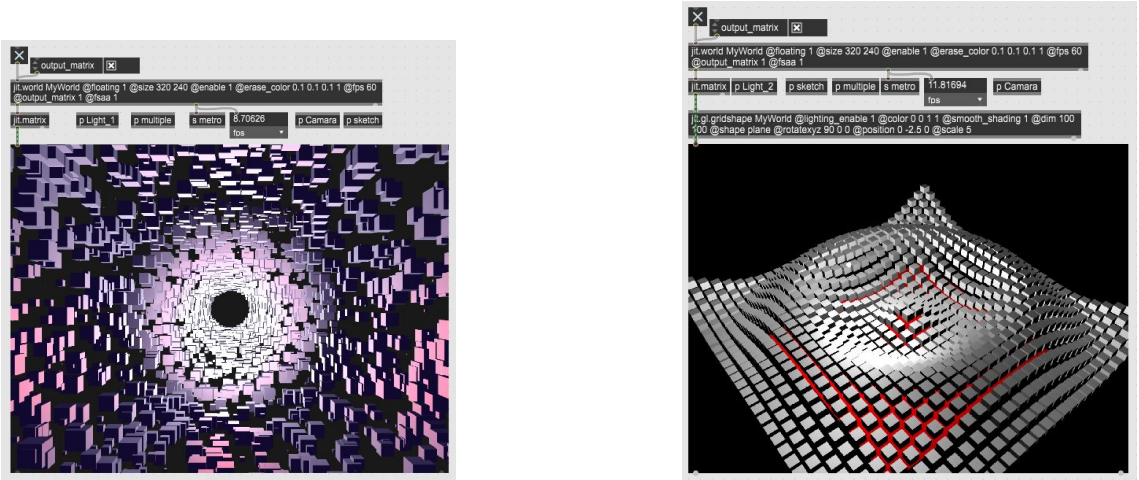

Figura 3.30: Módulos realizados con jit.gl.multiple

Esta metodología para realizar los módulos era prometedora, ya que se obtenían resultados de muy buena calidad en muy poco tiempo, además de que se podían utilizar y modificar objetos

virtuales, creados a partir de modelado virtual por otros programas (p.e. Blender). Sin embargo, no se llegaron a dar debido a que el proyecto requeriría la creación de muchos entornos diferentes, y esto suponía un coste de recursos para la CPU que no merecía la pena. Al final, fue necesario buscar una manera con la que, a partir de un solo entorno definido, se pudiera cambiar de manera sencilla los gráficos a generar.

3.2.3. Shaders

La GPU [40] (Unidad de Procesamiento Gráfico) es un microprocesador suplementario de la CPU dedicado al procesamiento de gráficos u operaciones de coma flotante, con el fin de aligerar la carga de trabajo en aplicaciones como los videojuegos o aplicaciones 3D. De esta forma, mientras gran parte de lo relacionado con los gráficos se procesa en la GPU, la unidad central de procesamiento (CPU) puede dedicarse a otro tipo de cálculos (como la inteligencia artificial o los cálculos mecánicos en el caso de los videojuegos) [41].

Una GPU está altamente segmentada, lo que indica que posee gran cantidad de unidades funcionales. Estas unidades funcionales se pueden dividir principalmente en dos: aquellas que procesan vértices, y aquellas que procesan píxeles. Por tanto, se establecen el vértice y el píxel como las principales unidades que maneja la GPU.

Muchas aplicaciones gráficas conllevan un alto grado de paralelismo inherente, al ser sus unidades fundamentales de cálculo (vértices y píxeles) completamente independientes. Por tanto, es una buena estrategia usar la fuerza bruta en las GPU para completar más cálculos en el mismo tiempo.

Como se comentaba en el apartado sobre OpenGL (vease apartado 3.2.2), los programadores de gráficos les pidieron a los desarrolladores de tarjetas gráficas que les permitieran el acceso a ciertas partes del procesado de gráficos para hacerlo más eficiente. De este acuerdo surgieron los *shaders*.

Un *shader* [42] es un programa diseñados para ejecutarse en alguna etapa del procesado de gráficos. Estos pueden ser configurados por el usuario. También se pueden usar en una forma un poco más limitada para el cálculo general en GPU.

El *rendering pipeline*[43] es el proceso de generación de gráficos, por el cual se transforman las coordenadas 3D del entorno virtual simulado a coordenadas 2D representantes de la pantalla. Este proceso comienza cuando se realiza una operación de renderizado. Estas operaciones requieren como datos de entrada un objeto que defina claramente la posición de los vértices de los polígonos que conforman la figura y las conexiones entre polígonos, llamados *vertex array objects* (VAO). Una vez iniciado, el rendering pipeline opera en el siguiente orden (fig. 3.31):

- Procesado de vértices: Cada vértice definido en el VAO es procesado por un *shader* denominado *vertex shader*, el cual se encarga de generar los vértices en el espacio, formando así malla básica que conforma el objeto. De manera opcional se puede configurar la teselación del objeto y el *shader* de geometría. El fin de estos dos *shaders* es la de dividir los triángulos definidos en el VAO en figuras primitivas (triángulos) más pequeñas, con el fin de alterar la definición de estos objetos.

- Post-procesado de vértices: Tras el procesado de vértices, se someten los resultados generados por el proceso anterior a varias operaciones matemáticas con el fin de adaptar los datos

generados para las etapas posteriores.

- Ensamblaje primigenio: Es el proceso en el que se componen las figuras primitivas a partir de los datos ofrecidos por el VAO. Durante este proceso, algunas figuras primitivas pueden ser descartadas para las siguientes etapas si no están de cara a la pantalla, ya que estas corren el riesgo de no ser visualizadas nunca.
- Rasterización: Es el proceso por el cual las figuras primitivas generadas en el proceso anterior se transforman en un conjunto de píxeles con el fin de ser desplegados en la pantalla. Estos píxeles se guardan en fragmentos(*fragments*), que son conjuntos de estados que guardan la posición del píxel en el espacio de la pantalla.
- Procesado de fragmentos: Los fragmentos generados por la rasterización son procesados por un *fragment shader*. La salida de este *shader* es color del fragmento analizado y un valor de profundidad para cada píxel especificado en el fragmento. La aplicación del *fragment shader* es opcional. Sin su aplicación, los valores de profundidad que posee cada píxel no se ve alterado, pero los colores que puede tener un fragmento no estarían definidos. La representación de fragmentos sin *fragment shader* solo es útil en el estudio de la profundidad predeterminada de estos.
- Procesado por muestra: En este último proceso se trata la información generada por el *fragment shader*, generando datos que son escritos en un *buffer* y que se envían a la pantalla para poder mostrar el resultado al usuario.

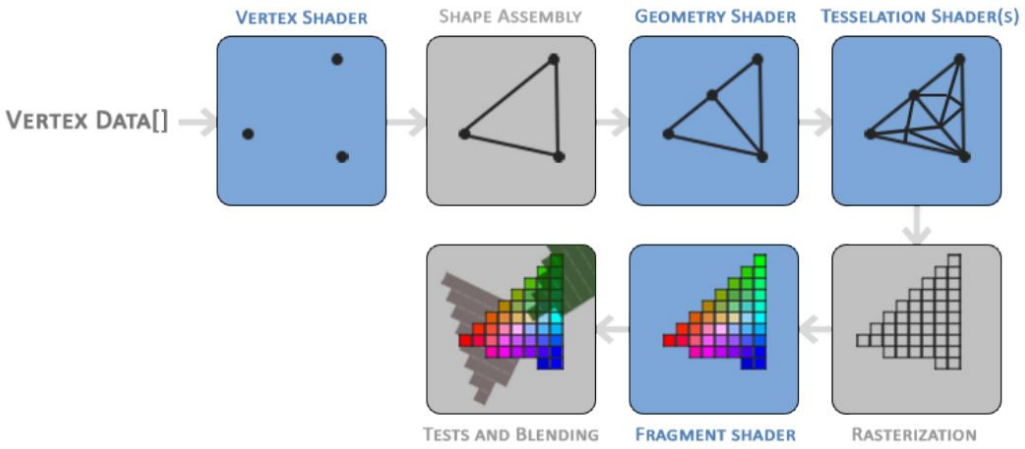

Figura 3.31: Rendering pipeline.

En el anterior intento de realizar los gráficos para el módulo, se estaban realizando todos los pasos del rendering pipeline: se cargaba en un espacio tridimensional un objeto (fases de *vertex shader*, post-procesado y ensamblaje primigenio), y luego se le ponía una textura (fase de procesado de fragmentos) y por último se mostraba por pantalla (fase de operaciones por muestra). Para agilizar el proceso se decidió no utilizar VAOs y emplear solo el proceso de *fragment shader* para aplicarlos directamente los gráficos sobre la pantalla, ya que de este se pueden obtener colores para cada uno de los píxeles que procesa. Lo único que se necesita para ello es definir un plano que coincida con las dimensiones de la pantalla y aplicar el procesado de gráficos sobre este. De esta manera se evitará la creación de objetos tridimensionales y el entorno siempre será el mismo.

Los objetos jit.gl.pix y jit.gl.slab permiten la creación de *shaders* y la aplicación de estos a los objetos 3D y 2D que se deseen. La única diferencia entre ambos es el lenguaje de programación. El bloque jit.gl.slab utiliza el lenguaje original de los *shaders* de OpenGL (GLSL) mientras que jit.gl.pix utiliza el lenguaje nativo de MAX, GEN, el cual acaba siendo traducido a GLSL para poder funcionar en la GPU.

Los procesos para la generación de gráficos en MAX por medio del diseño de *fragment shaders* se describen en los apartados siguientes.

3.2.4. Definición del entorno

Para poder empezar, hay que definir un entorno virtual para representar los *fragment shaders* que se vayan a realizar. Para este caso se emplea el objeto jit.gl.render, el cual tiene la función de renderizar objetos de OpenGL y hacerlos visibles, en este caso, los gráficos generados por el *fragment shader*.

El *shader* está contenido en el bloque jit.gl.pix, el cual funciona a la misma frecuencia que jit.gl.render. El resultado sale con formato de matriz de vídeo, con lo que se puede tomar como salida del módulo sin ningún problema. El entorno definido se puede ver en la siguiente figura (fig. 3.32):

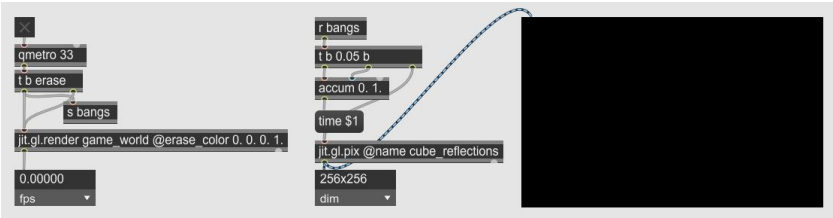

Figura 3.32: Entorno de representación de fragment shaders

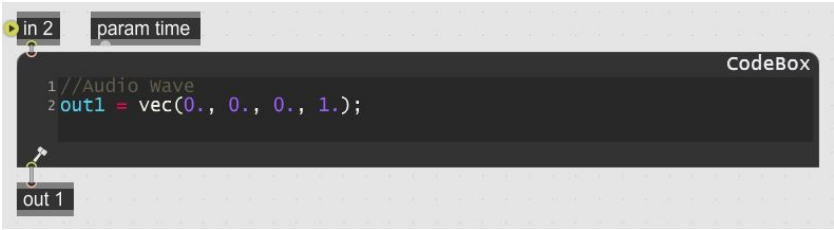

Figura 3.33: Entorno de representación de *fragment shaders*.

El *fragment shader*, dentro del bloque jit.gl.pix, se plantea tal y como se muestra en la figura 3.33. Se utiliza el bloque codebox, el cual permite la inserción de código escrito a mano en el objeto. Aquí es donde se programa el contenido del *fragment shader*. Dentro del bloque codebox se escribe el código del *fragment shader*, que asignará un color a cada píxel de la pantalla. Esta información vendrá codificada como una matriz de *Jitter* donde están codificados los colores según el código RGBA. Para tomar otros datos, como la señal de audio u otros parámetros, tan solo hay que definir el parámetro dentro de jit.gl.pix y conectarlo a la entrada principal del bloque. Los parámetros de entrada son siempre el tiempo y el audio procesado (audio, fft, sonograma y ondas

de ritmo) en forma de textura o de matriz de datos. Para evitar complicaciones, el audio tiene su propia entrada al bloque.

3.2.5. Gráficos 2D

Para empezar a generar gráficos a partir de *fragment shaders*, se comenzará con la implementación de gráficos en 2D, ya que estos son más sencillos de generar que los gráficos en 3D. Se explicará uno por uno lo que se ha hecho en cada uno de ellos.

3.2.5.1. Audio Wave: Representación de la onda de audio

El primero módulo a realizar un módulo de visualización de la onda de audio. El objetivo es captar la matriz del osciloscopio y representarla como una onda en el dominio temporal.

En primer lugar, para este módulo y para todos los que siguen, es definir la pantalla y normalizar las coordenadas. La normalización de la pantalla es el resultado de dividir la posición del píxel entre la resolución de la pantalla o la ventana en la que se esté visualizando. Para ello, el comando "norm"de MAX permite la normalización de las coordenadas de la pantalla, haciendo que se hallen entre los valores 0 y 1 para ambos ejes. La figura 3.34 muestra una visualización de la pantalla donde se representa el valor normalizado de los ejes 0X y 0Y con los colores rojo y verde, respectivamente.

Figura 3.34: Coordenadas normalizadas de la pantalla

El siguiente punto es muestrear correctamente la matriz que representa el audio de entrada. Para ello se utiliza el comando sample, el cual muestrea una matriz de vídeo en las coordenadas especificadas. Para este caso, se muestrea la matriz completa en la posición completa de la pantalla. De esta manera se podrá visualizar en toda la pantalla. En la figura 3.35 se puede observar el resultado del muestreo del audio.

Por último, hay que asociar la amplitud de la onda con la posición en la pantalla. Para ello, se ha aplicado esta ecuación a la salida del *shader* (3.13):

$$col = \frac{(R', G', B')}{|wave - uv.y|} \tag{3.13}$$

Figura 3.35: Resultado de sample

Siendo col el color de los píxeles de salida, $wave$ la amplitud de la onda de audio, el vector (R', G', B') controla el color que obtiene la onda y $uv.y$ el eje 0Y de las coordenadas discretas de la pantalla. Cuando $uv.y$ y $wave$ tienen valores similares, el denominador comienza a multiplicarse, mejorando su visibilidad. Cuanto mayor sean las magnitudes del vector (R', G', B'), antes comenzarán a verse los valores similares de cada franja de colores. Cuando $uv.y$ y $wave$ coinciden, el resultado se indetermina, dando el valor máximo a los píxeles de salida. De esta manera se obtiene la representación gráfica de esta (fig. 3.36).

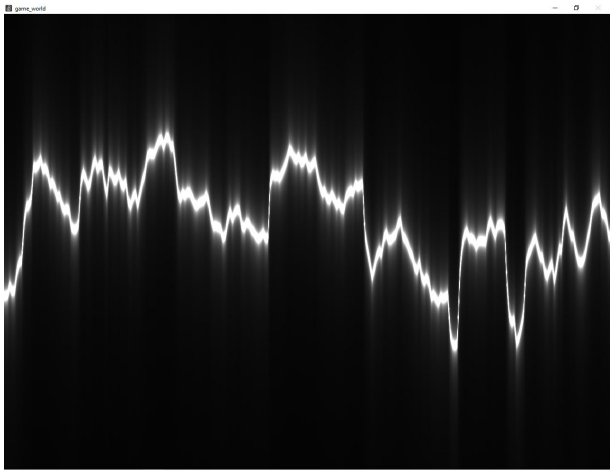

Figura 3.36: Onda resultante.

3.2.5.2. 2D LED Spectrum: Espectrograma amplitud/frecuencia

El siguiente módulo realizado es un módulo para la representación del espectro de frecuencias amplitud-frecuencia, pero discretizando y redondeando los valores de estos para dar la impresión de un *vu meter* digital (fig 3.37).

El módulo comienza con la normalización de la pantalla. Los LEDs que aparecen en el vu

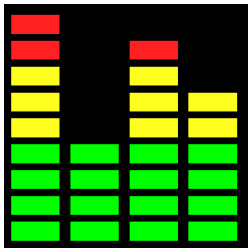

Figura 3.37: Vumeter digital

meter limitan la resolución de esta y del espectro de frecuencias. La pantalla se divide en filas y columnas para formar la matriz de LEDs. Tras definir esta matriz, se discretizan las coordenadas normalizadas de la pantalla siguiendo las funciones (3.14) y (3.15):

$$p.x = \frac{floor(uv.x * bands)}{bands} \tag{3.14}$$

$$p.y = \frac{floor(uv.y * columns)}{columns} \tag{3.15}$$

Siendo $p.x$ y $p.y$ las coordenadas discretas de la pantalla, uv las coordenadas normalizadas, y $bands$ y $columns$, las filas y las columnas de la matriz de LEDs, respectivamente. La función floor redondea a la baja el valor de la multiplicación, de manera que los resultados de esa función son números enteros que están entre 0 y el número de filas/columnas menos uno. Tras dividir entre el número de filas/columnas, el rango vuelve a estar entre 0 y 1, pero los valores tienen menor resolución que antes (fig. 3.38).

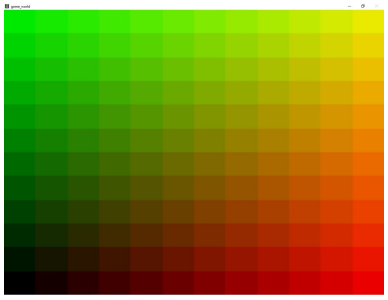

Figura 3.38: Coordenadas normalizadas discretizadas.

El color de los LEDs viene dado por un degradado que obedece la siguiente función (3.16):

$$back(uv.y) = floor\left(\frac{1{,}5 * (uv.y * 2{,}2 - uv.y * 2))}{1{,}5}\right) \tag{3.16}$$

Siendo $back$ el color de los LEDs según su posición en la pantalla. La función queda como un degradado vertical que va del verde al rojo, pasando por el amarillo. Para darle un color definido a cada LED, se decide discretizar los colores, de manera que los LEDs solo puedan pasar del verde al amarillo, y del amarillo al rojo. Para ello se añadió la función *floor*, así se puede dividir la función en tres bandas, el resultado de aplicar dicha función no fue del todo satisfactorio: los colores se

redujeron a tres, pero el que debería haber sido amarillo era de un tono marrón. Para arreglarlo, se sustituyó el color marrón por amarillo de forma manual con una estructura *if/else*. De esta manera se consiguieron los colores de los LEDs. La evolución de estos colores se muestra en la figura 3.39.

Figura 3.39: Evolución de los colores de los LEDs

Para poder plasmar el espectro en la pantalla sin alterar la resolución, hay que muestrear cada valor de la señal con las nuevas coordenadas de posición de la pantalla definidas en el vector p. El resultado del muestreo a la resolución adecuada se puede observar en la figura 3.40.

Figura 3.40: FFT discretizada con la resolución de la pantalla

En este punto, los valores de la FFT han de ser transformados de magnitud a altura en la pantalla. Para ello se compara la magnitud de una banda con la altura discretizada, similar a cómo se realizó la onda de audio (vease 3.3.2.5.1). Si la altura discretizada es menor a la magnitud del espectro, devuelve un 1 y, en caso contrario, un 0. De esta manera se obtiene una huella indicando qué LEDs están encendidos y cuáles no (fig. 3.41).

Figura 3.41: FFT discretizada con la resolución de la pantalla

Una vez generada la máscara y el fondo de LEDs el siguiente paso es definir el espacio entre

cada uno de los LEDs, ya que no hay una separación entre ellos, hay definir los bordes de cada celda, para saber donde limitan unos con otros (3.17):

$$d(uv) = ((uv - p) * vec(bands, columns) - floor((uv - p) * vec(bands, columns))) - 0{,}5 \tag{3.17}$$

Siendo $d(uv)$ la función que define los límites derecho y superior de cada celda. Cuando uv coincide con p, la diferencia es 0 y el color en la pantalla es negro, pero cuando difieren mucho, la diferencia comienza a ser notoria. Los bordes de cada celda se obtienen con la siguiente ecuación (3.18):

$$led(uv) = smoothstep(0{,}5, 0{,}35, abs(d(uv).x)) * smoothstep(0{,}5, 0{,}35, abs(d(uv).y)); \tag{3.18}$$

Siendo $led(uv)$ los bordes de cada celda. La función *smoothstep* realiza la interpolación entre 2 términos para que la transición entre LEDs no sea una línea negra solida. Se hace esta interpolación para los bordes verticales y para los horizontales (fig. 3.42).

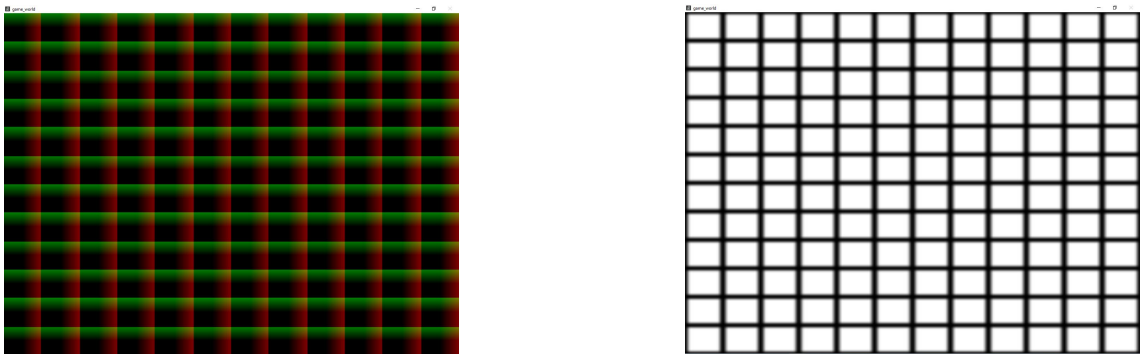

Figura 3.42: Búsqueda de los límites de las celdas(derecha) y generación de los bordes de los LEDs (izquierda)

Una vez se han obtenido los bordes, los colores de los LEDs y el espectro adaptado a la nueva resolución, tan solo hay que unir todo para obtener el resultado final (fig. 3.43).

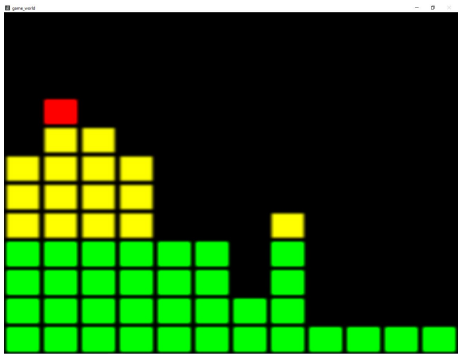

Figura 3.43: Espectro 2D

3.2.5.3. Espectrograma: Aplicación de colores

Con el siguiente módulo se dio color al espectrograma. Debido a que la escala de colores va del blanco al negro el usuario que busque utilizarlo a modo de analizador de espectros profesional, puede perder detalles de interés debido a esta reducida gama de colores.

Los mapas de colores [44] son muy útiles a la hora de resaltar conjuntos de datos que, a simple vista, son difíciles de apreciar en conjunto en una tabla de datos numéricos. Se utilizan para representar mapas de calor, planos con altitudes y espectros de frecuencias, entre otros usos. Estos mapas de colores tiene que cumplir ciertos requisitos:

- Ha de tener una amplia gama de colores bien diferenciados entre sí, para que las diferencias sean fáciles de ver.
- Que la progresión de colores sea secuencial. Por lo general no se sabe que se va a querer representar con el mapa de colores, con lo que lo mejor es diseñarla de manera neutra.
- Que se perciban de manera uniforme, lo que significa que los valores cercanos entre sí han de tener colores de apariencia similar y los valores alejados entre sí, colores de apariencia más diferente.
- Ha de ser útil para personas con daltonismo, de modo que las propiedades anteriores sean válidas para ellas, así como en la impresión en escala de grises.
- Ha de ser agradable a la vista, para fomentar su uso.

La plataforma *matplotlib* [45] es una libreria de python [46] para la generación de mapas y gráficos. Posee ciertos mapas de colores que resultan muy interesantes para utilizar en el espectrograma, ya que cumplen todos los requisitos de un buen mapa de colores con propósitos científicos (fig. 3.44).

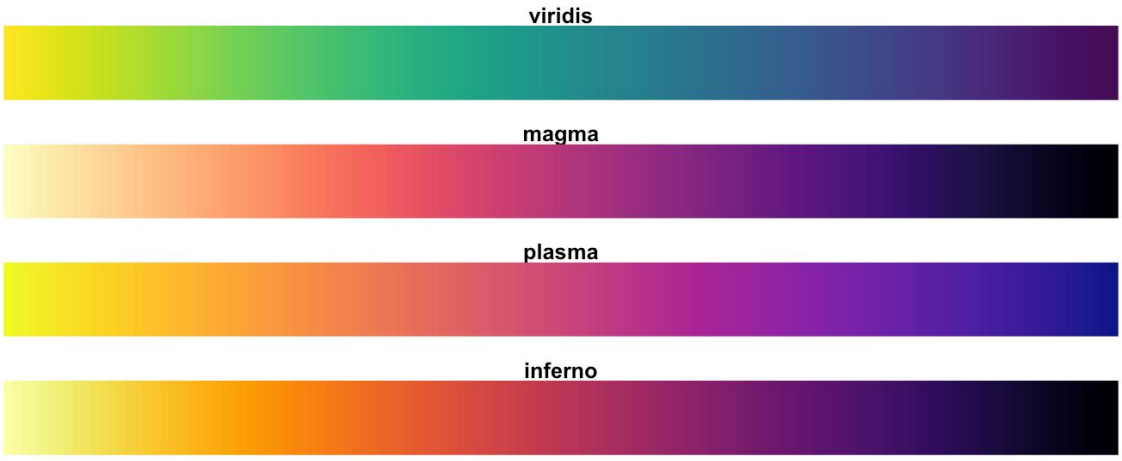

Figura 3.44: Escalas de colores de matplotlib

La principal característica que poseen estos mapas es que los datos no pierden relevancia ante cambios drásticos del color de estos por factores externos, como por ejemplo una impresión en blanco y negro o el daltonismo del usuario. En la figura 3.45 se puede observar que, frente a otras

escalas de colores muy utilizadas, los mapas *viridis* y magma consiguen mantener las propiedades de uniformidad ante cambios enérgicos en la percepción de estos.

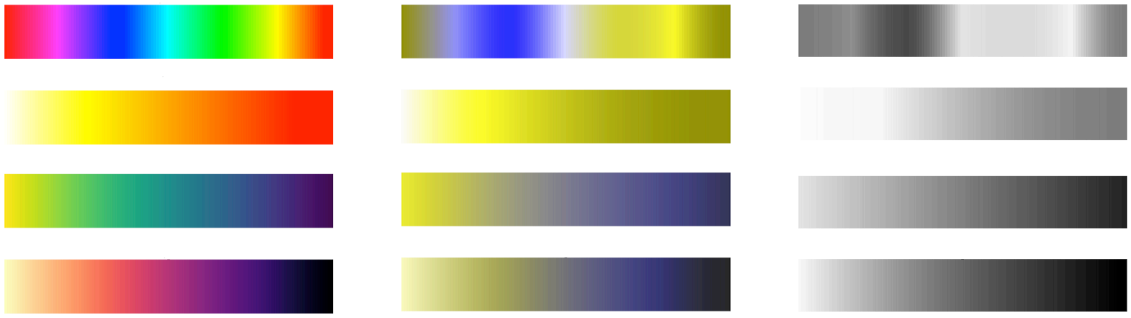

Figura 3.45: Comparación de los mapas de colores viridis y magma con otros mapas utilizados

En la figura 3.45, se puede observar que el mapa arcoiris (primero) no es uniforme y que solo se puede conocer su linealidad cuando están representados todos los colores. En cuanto al mapa de calor (segundo), en cierto momento no se puede apreciar el crecimiento de los datos. En cambio, los mapas viridis (tercero) y magma (cuarto) mantienen la linealidad en todo momento y se puede observar el crecimiento de los datos en cualquiera de los casos.

Para crear el módulo, lo primero que hay que hacer es muestrear la señal de entrada, esta será el sonograma completo (frecuencia-amplitud-tiempo) que se realizó durante la etapa de procesado de audio. Después, hay que aplicarle la escala de colores al sonograma. Los parámetros para la generación de estos mapas de colores están al alcance del dominio público en el repositorio de la *Berkeley Institute for Data Science* (BIDS) [45].

Los resultados obtenidos han sido sonogramas más coloridos, en los que se puede distinguir mejor en qué momento las frecuencias tienen más fuerza en el sonido a analizar en cualquier momento y bajo cualquier condición (fig. 3.46).

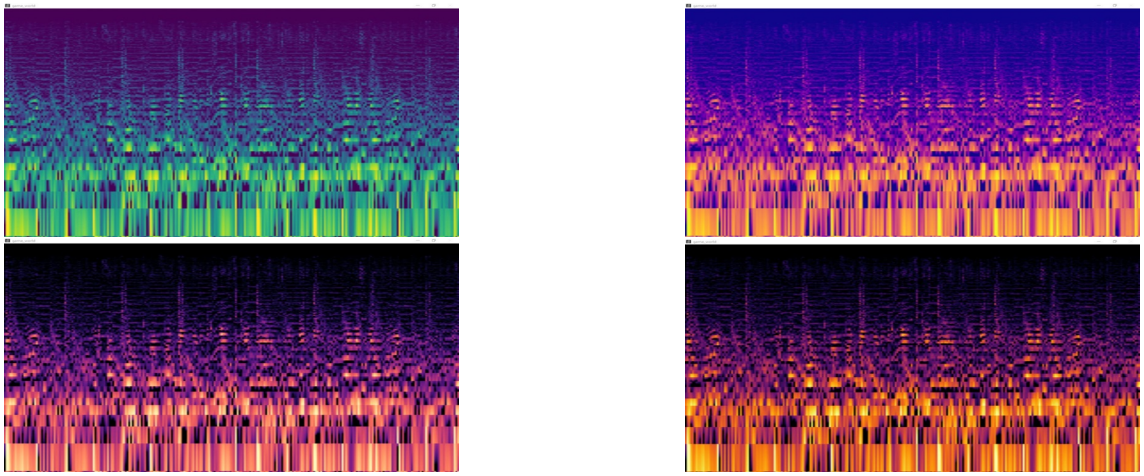

Figura 3.46: Resultados de la aplicación de los mapas de colores de la librería matplotlib al sonograma.

3.2.5.4. Complex Analysis: Representación del audio en planos complejos

Hasta ahora se ha representado el audio sobre un plano 2D ortogonal, en el que los ejes 0X y 0Y eran perfectamente ortogonales, dando lugar a representaciones cartesianas. Con el análisis de planos complejos [46], se obtiene la distorsión del plano y se consiguen patrones curvos, los cuales aportan una estética adecuada al objetivo de visualización atractiva de audio del presente proyecto.

El primer paso en la creación de estos módulos es normalizar el plano de visualización, pero a diferencia de los módulos anteriores, en este caso los valores de la pantalla han de encontrarse entre -1 y 1. El comando *snorm* de MAX normaliza las coordenadas entre -1 y 1 en cada eje.

Para distorsionar el espacio, se utilizan operaciones con funciones trigonométricas hiperbólicas. Estas operaciones afectan a los ejes 0X y 0Y del espacio, los cuales toman los valores impuestos por estas funciones, en lugar de evolucionar de manera lineal. Esto produce que las coordenadas en el espacio se alteren, y no sigan un patrón cartesiano. Por esta razón, es necesario ampliar el rango de valores de los ejes de $(-1, 1)$ a $(-\pi, \pi)$, con el fin de que las operaciones trigonométricas se puedan aplicar a todo su rango hábil. Las fórmulas más básicas empleadas para estas transformaciones se encuentran en la tabla 3.2. A partir de estas ecuaciones, y de combinaciones de estas, se han obtenido los patrones que han formado parte del resultado final de estos módulos.

	$0X$	$0Y$
$sinz(uv)$	$sin(uv_x) * cosh(uv_y)$	$cos(uv_x) * sinh(uv_y)$
$cosz(uv)$	$cos(uv_x) * cosh(uv_y)$	$-sin(uv_x) * sinh(uv_y)$
$tanz(uv)$	$sin(uv_x) * cos(uv_x)$	$\frac{(sinh(uv.y) * cosh(uv_y)}{(cosx * cosx + sinhy * sinhy)}$
$logz(uv)$	$log(\sqrt{uv \cdot uv})$	$atan(uv_y)$
$sqrtz(uv)$	$\frac{uv_x + \sqrt{uv \cdot uv}}{\sqrt{2(uv_x + \sqrt{uv \cdot uv})}}$	$\frac{uv_y}{\sqrt{2(uv_x + \sqrt{uv \cdot uv})}}$
$exp2z(uv)$	$(uv_x)^2 - (uv_y)^2$	$2 * uv_x * uv_y$
$epowz(uv)$	$cos(uv_y)$	$sin(uv_y) * e^{uv_x}$
$invz(uv)$	$\frac{uv.x}{uv \cdot uv}.x$	$\frac{uv.y}{uv \cdot uv}$

Tabla 3.1: Ecuaciones para análisis de planos complejos.

La figura 3.47 muestra ejemplos de coordenadas distorsionadas por estas ecuaciones, siendo la figura de la esquina superior izquierda el espacio normalizado original. Las coordenadas de 0X y 0Y vienen representadas por los colores rojo y verde, respectivamente. Tras la aplicación de la operaciones de análisis complejo, los colores verde y rojo se reparten por toda la pantalla siguiendo patrones circulares, fractales y otras formas geométricas curvas. Estos nuevos mapas indican con su distribución de colores, de qué manera se va a plasmar el audio en la pantalla. Con el espacio original se plasma tal y como entra, como una imagen ortogonal, pero con los nuevos espacios, adopta figuras y patrones curvos que dan lugar a ilusiones psicodélicas muy visuales.

Para poder evitar que los nuevos espacios sean estáticos, se les aplica una matriz de rotación con respecto al eje 0Z marcada por el tiempo, la cual hace girar el espacio original y modifica las posiciones de las coordenadas espaciales alteradas por las ecuaciones complejas (3.20).

$$R(uv, t) = \begin{bmatrix} cos(uv) & sin(uv) \\ -sin(uv) & cos(uv) \end{bmatrix} * \frac{t}{2} \tag{3.19}$$

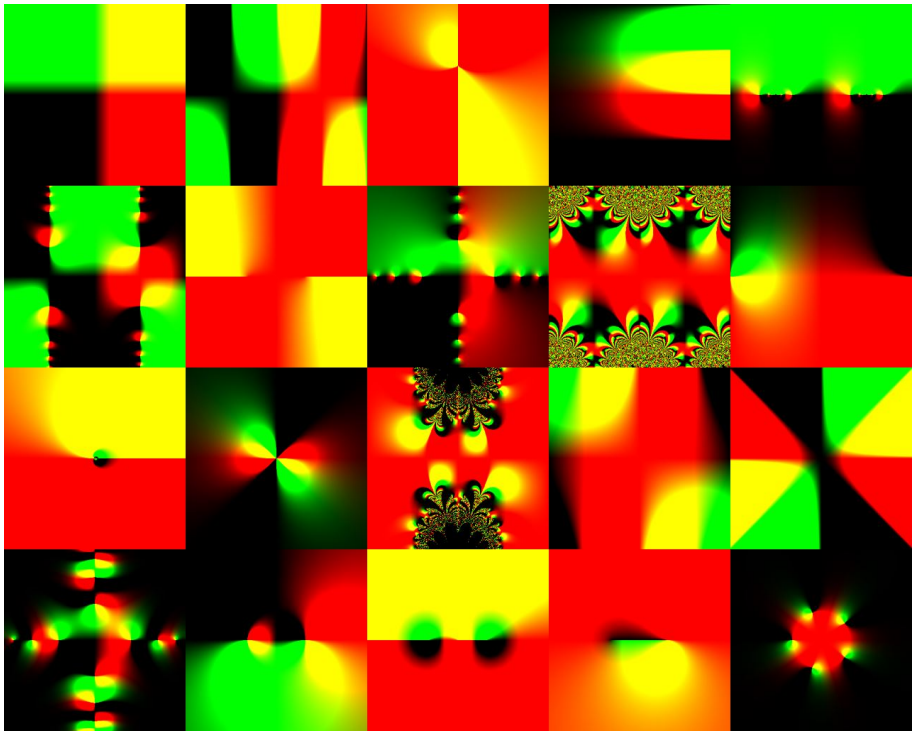

Figura 3.47: Representación de las coordenadas de los planos complejos.

Después se procede al muestreado de la señal de audio. Para estos casos, se utilizará la señal obtenida de la transformada de Fourier, ya que ofrece una respuesta reactiva más armoniosa que la señal de audio. Al representar el espectro de frecuencias, reacciona con fuerza con las frecuencias graves, lo que genera la sensación de que sigue el ritmo y los golpes de la pieza. En cambio, la visualización de la señal de audio es bastante aleatoria, lo que genera mucho ruido visual en los módulos y no es agradable a la vista. Con el fin de realizar patrones cuadrados, la señal se muestrea dos veces, una por cada eje, tras muestrearla se plasma en la pantalla con las coordenadas distorsionadas (fig. 3.48).

Al plasmarlo, se obtiene una imagen en blanco y negro ya que el espectro de frecuencias se obtiene sin coloración. Para poder obtener una gama de colores amplia a partir de estos valores, se supone primero una escala de colores para organizar estos valores y luego se traduce a la escala RGB. Esta es la escala HSV (fig 3.49), la cual define un color según los siguientes términos:

- Matiz (H): Se representa como un grado de ángulo cuyos valores posibles van de 0 a 360° . Para cada valor se corresponde un color (p.e.: 0, rojo; 60, amarillo; 120, verde).
- Saturación (S): Se representa como la distancia al eje de brillo negro-blanco.
- Valor (V): Representa la altura en el eje blanco-negro.

Para estos módulos, el matiz viene definido por el espectro, la saturación está fijada al máximo, y el valor viene dado por la posición de los bordes que limitan la imagen del espectro. Para traducir esta escala a la escala RGB, hay que seguir las siguientes ecuaciones (3.21), (3.22), (3.23), (3.24), (3.25):

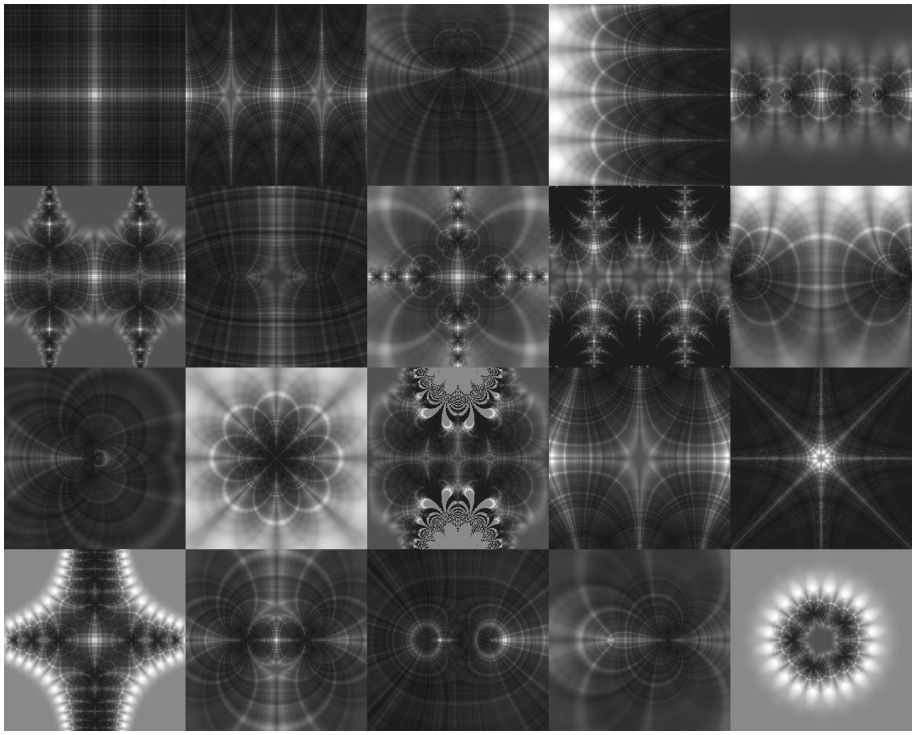

Figura 3.48: Planos complejos con muestra del espectro de frecuencias.

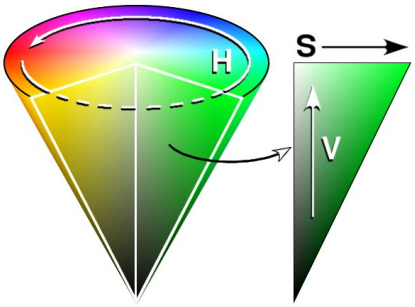

Figura 3.49: Modelo de color HSV.

$$C = V \cdot S \tag{3.20}$$

$$X = C \cdot (1 - |\frac{H}{60°} * \sqrt{2^2 + 1^2}|) \tag{3.21}$$

$$m = V - C \tag{3.22}$$

$$(R', G', B') = \begin{cases} (C, X, 0), & \text{if } 0 \leq H \geq 60 \\ (X, C, 0), & \text{if } 60 \leq H \geq 120 \\ (0, C, X), & \text{if } 120 \leq H \geq 180 \\ (0, X, C), & \text{if } 180 \leq H \geq 240 \\ (X, 0, C), & \text{if } 240 \leq H \geq 300 \\ (C, 0, X), & \text{if } 300 \leq H \geq 360 \end{cases} \tag{3.23}$$

$$(R, G, B) = ((R' + m) * 255, (G' + m) * 255, (B' + m) * 255) \tag{3.24}$$

Una vez definida la transformación de HSV a RGB, el resultado es un colorido patrón geométrico basado en el enélisis de frecuencias, dando por concluido el desarrollo de este módulo. En la figura 3.50 se muestra un ejemplo de los módulos creados los módulos creados. A pesar de haber veinte, se pueden crear muchos más y más complejos por medio de la superposición y combinación de las ecuaciones definidas en la tabla 3.2. Por ello, se anima en futuros trabajos la modificación del código de los módulos con el fin de crear patrones visuales más interesantes.

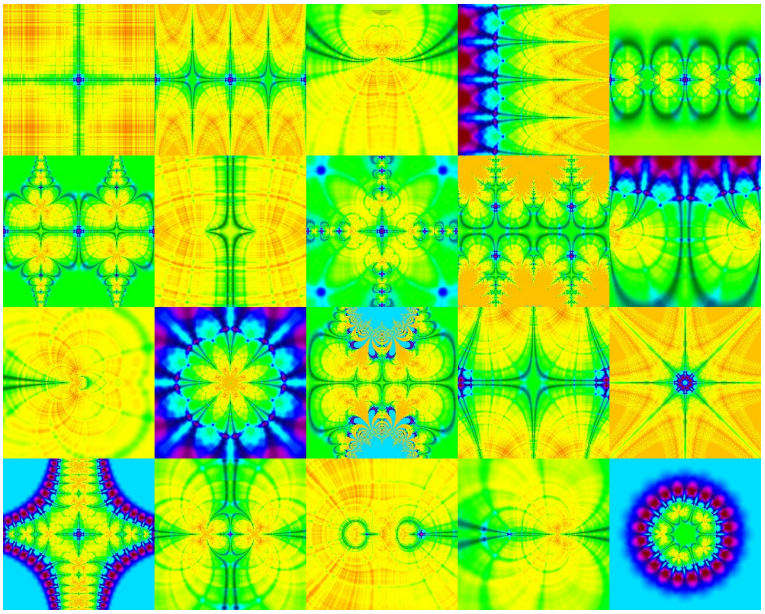

Figura 3.50: Módulos de planos complejos con coloración RGB.

3.2.5.5. Otros módulos 2D: Coordenadas polares

Los restantes módulos 2D tienen la peculiaridad de que se representan con coordenadas polares (fig. 3.50), estas se caracterizan por representar puntos en un plano bidimensional en función de

su distancia con respecto al origen del sistema y al ángulo que forma el segmento que hay entre el punto y el origen, con el eje polar que también pasa por el origen. Esto produce que el espacio de representación quede circular y centrado en mitad de la pantalla, dando lugar a patrones circulares con una base sencilla.

Para transformar las coordenadas normalizadas de la pantalla a coordenadas polares se aplican sobre ellas las siguientes ecuaciones (3.26) y (3.27):

$$\theta = \frac{-atan(uv_y, uv_x) + \pi}{\pi} \tag{3.25}$$

$$r = \sqrt{uv_x)^2 + uv_y)^2} \tag{3.26}$$

Siendo θ el ángulo del punto con respecto al eje polar y r el módulo del punto (fig. 3.51).

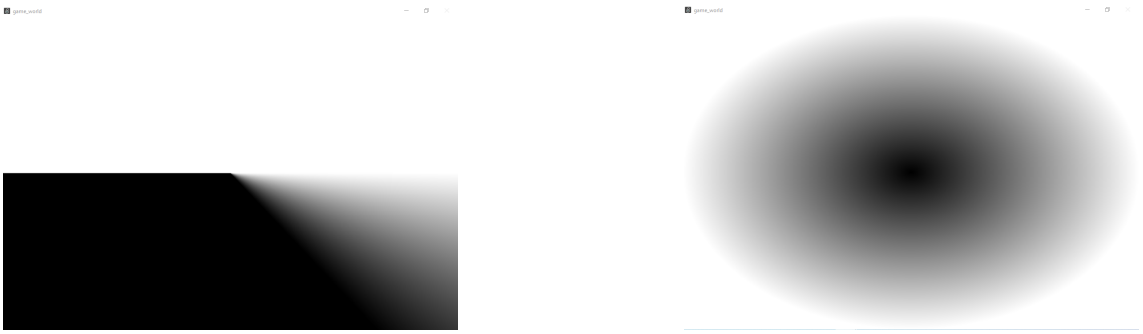

Figura 3.51: Coordenadas polares de la pantalla: θ (derecha) y r (izquierda).

El módulo llamado *audio tunnel* se limita a mostrar el espectro de frecuencias en blanco y negro en la pantalla con las coordenadas polares absolutas, para evitar que ningún hueco en la pantalla quede oscurecido. En cambio, con el módulo llamado *lotus* se experimenta un poco con el color, la adquisición de las muestras de audio y las formas, obteniendo una serie de espirales simétricas que cambian constantemente de color (fig. 3.52).

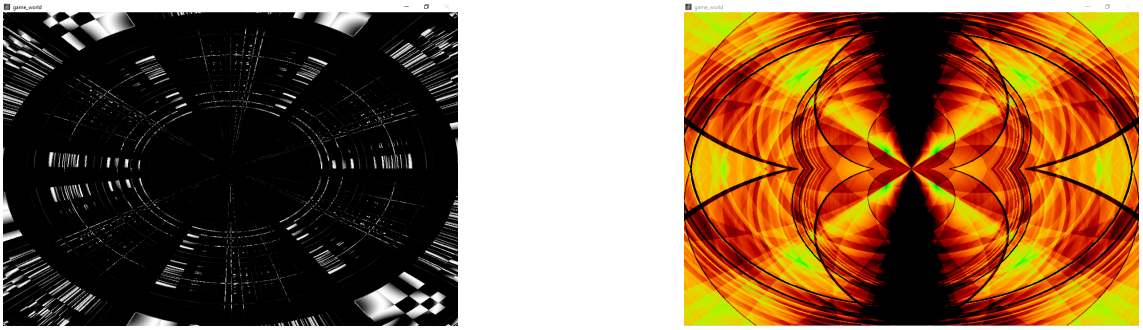

Figura 3.52: *Audio tunnel* (derecha) y *lotus* (izquierda).

3.2.6. Gráficos 3D

Durante este trabajo, la primera aproximación a los gráficos 3D fue con bloques propios de MAX, con los que se generaba el espacio tridimensional, se llenaba con figuras tridimensionales

y planos, se texturizaba y, por último, se ajustaba la ventana para producir la señal de vídeo. Como se explicó con anterioridad, el coste computacional para realizar varios de estos módulos y mantenerlos funcionando era demasiado elevado, con lo que se decidió crear un entorno constante y utilizar el proceso de *fragment shader* para crear los gráficos a mostrar. Durante el proceso de *fragment shader*, se le aplican texturas a las superficies de los objetos de la escena, en este caso, al plano generado que coincide con las dimensiones de la pantalla. Esto implica un ahorro de recursos para la CPU, pero conlleva a la renuncia a un entorno 3D. Esto no es un impedimento para poder generar gráficos que den la ilusión de tener dimensión de profundidad, tan solo hay que comprender cómo generar sobre una imagen bidimensional una imagen que dé la apariencia de tener una tercera dimensión.

3.2.6.1. Ray Marching

Para poder realizar los gráficos tridimensionales, se ha recurrido a una técnica de generación de gráficos llamada *ray marching* [49]. Este algoritmo, usado en combinación con unas funciones de distancia con signo, puede crear gráficos aparentemente tridimensionales en tiempo real.

El ray marching es un algoritmo para la síntesis de imágenes tridimensionales. Al igual que el *ray tracing* [50], el ray marching está basado en el algoritmo de determinación de superficies denominado *ray casting*.

El algoritmo de ray casting determinan las superficies visibles de la escena virtual trazando rayos desde la posición del usuario observador hacia la escena a través de la pantalla. La idea del trazado de rayos no es una idea moderna, se le atribuye a Alberto Durero (1471-1528) la creación de la técnica de dibujo por trazado de rayos (fig. 3.53), y fue en 1982 cuando se acuñó el término ray casting en el contexto de los gráficos por ordenador [51].

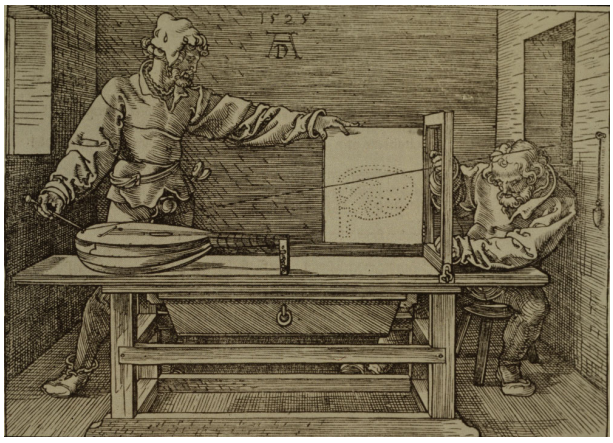

Figura 3.53: Hombre dibujando un laúd con una técnica similar al ray casting (grabado en madera de Alberto Durero, 1525).

Estos tres algoritmos parten de la misma base: un rayo que sale del observador, atravesando la pantalla, que choca con un objeto en el entorno y devuelve color y/o distancia del objeto. Para poder comenzar a desarrollar el algoritmo, es necesario empezar por la idea de una cámara, la cual simboliza la posición del observador ante el entorno a visualizar. Los datos de entrada de la cámara han de ser los siguientes:

- Coordenadas normalizadas de la pantalla (uv).
- Posición de la cámara en el espacio.
- Objetivo: Punto del espacio al que la cámara está mirando (objetivo).
- Distancia del usuario a la pantalla (zoom).

La cámara, a partir de estos datos, devuelve un rayo de visión, que parte desde la cámara, pasa por un píxel de la pantalla y llega hasta la intersección con un objeto o hasta el límite de longitud impuesto. Hay un rayo por cada píxel en la pantalla. Los datos de estos rayos entran en el proceso denominado *ray intersection*, el cual intersecta cada rayo con los objetos del entorno. Este proceso devuelve un valor de distancia que índica la posición a la que se encuentra el objeto de la cámara. Si se han definido las composiciones de los materiales y la iluminación del entorno, se obtiene un color para el píxel que se estaba analizando. En la figura 3.54 se puede observar el funcionamiento de este algoritmo.

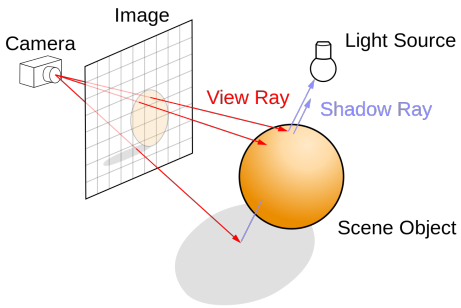

Figura 3.54: *Ray casting*.

La diferencia entre el ray tracing y el ray marching radica en cómo se define la escena. En el ray tracing, la escena generalmente se define en términos de geometría explícita: figuras geométricas conformadas por polígonos y figuras primitivas (VAOs, vease 3.2.3). Para encontrar la intersección entre el rayo de vista y la escena, se realizan una serie de pruebas de intersección geométrica("¿dónde está exactamente la intersección del rayo con la figura?"). En el ray marching toda la escena se define en términos de una función de distancia con signo [52]. Para encontrar la intersección entre el rayo y la escena, se comienza en la cámara y se mueve a lo largo del rayo de vista. En cada punto que recorre se evalúa si está dentro de la superficie de la escena. Si es así, el rayo ha golpeado algo, si no lo es, sigue avanzando y se repite la pregunta hasta llegar al límite impuesto o golpear una superficie.

Los pasos que se avanzan a lo largo del rayo no son constantes: con el fin de hacer el algoritmo más veloz y preciso, al principio se avanza una distancia de seguridad, con la que es seguro que no se salta ningún elemento de la escena. Las distancias para los siguientes pasos se obtienen a partir de las ecuaciones de distancia con signo [51, 52], a estas se les pregunta a qué distancia está el punto de la superficie de un objeto del entorno del rayo. Alrededor del punto en el rayo, se traza una esfera con un radio igual a esta distancia, y el siguiente paso será en la intersección del rayo con la esfera trazada. En cada paso se hará la misma pregunta hasta que el radio de la esfera sea lo suficientemente pequeño como para considerar que se ha golpeado el objeto o hasta que se considere que el rayo ha avanzado lo suficiente y que no hay nada de interés más adelante (fig. 3.54).

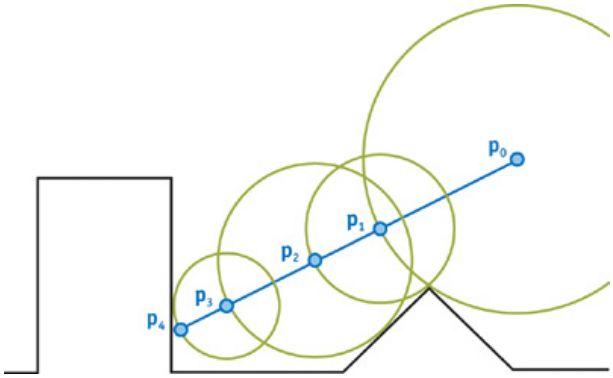

Figura 3.55: Algoritmo de *Sphere Tracing*.

En la figura 3.54, p_0 es la posición de la cámara y la línea azul es el rayo de visión. Siendo p_n los pasos que se dan a lo largo del rayo hasta alcanzar la superficie del objeto, se puede observar que cada uno coincide con la intersección del rayo y de la esfera trazada alrededor de cada uno que simbolizaba la distancia mínima que hay entre el rayo y el entorno. Tal y cómo se formulaba, el rayo avanza a pasos marcados por su distancia a los objetos, y una vez alcanzada la superficie máxima o la distancia máxima permitida, devuelve la distancia a la que se encuentra, el punto, de la cámara, si es que ha llegado a este.

El entorno viene definido por funciones de distancia con signo. Estas funciones devuelven la distancia más corta entre un punto x de un conjunto S, y devuelve un signo del valor de retorno si el punto está dentro de esa superficie o fuera (fig. 3.56). Con estas funciones se pueden definir todo tipo de figuras básicas, con las que se pueden realizar cualquier entorno.

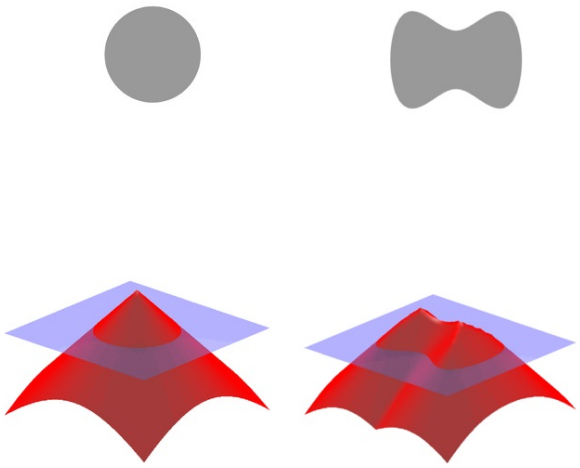

Figura 3.56: Formas planas (gris) y sus funciones de distancia con signo(rojo).

3.2.6.2. Primeros pasos

Para comenzar a realizar cualquier proyecto de *ray marching*, hay que empezar por la cámara. La cámara es un punto en el espacio que está enfocando un punto concreto del entorno, con lo que hay que definir su posición en el espacio y la dirección a la que apunta. Estas variables van a adoptar el nombre de r_0 y L_p (*Ray Origin* y *Lookup Point*, respectivamente). Para que la cámara funcione bien, es necesario que cumpla unos requisitos mínimos:

- r_0 ha de encontrarse siempre a una distancia definida de la pantalla, y esta ha de estar siempre en posición perpendicular a r_0. Si esto no se cumple, el movimiento de la cámara puede producir efectos de deformación de la imagen y zoom indeseados, además de poder perder la visibilidad del entorno.
- La posición de la pantalla ha de seguir la posición de r_0 en todo momento. Si en algún momento, r_0 y L_p se encuentran sin tener entre ambos la pantalla, la escena no será visible.

A partir de estas dos variables se obtienen los puntos de intersección con la pantalla (i_p) y el vector director del rayo(r_d). El vector director se define por la dirección que se toma desde r_0 hasta i_p, siendo i_p cada píxel de la pantalla. Se puede obtener con las siguientes fórmulas (3.28, 3.29):

$$\overrightarrow{r_d} = i_p - r_0 \tag{3.27}$$

$$i_p = c + uv_x * \overrightarrow{R} + uv_y * \overrightarrow{U} \tag{3.28}$$

Siendo uv las coordenadas normalizadas del píxel dentro de la pantalla y c el centro de esta en el espacio XYZ. La pantalla está sumergida en un espacio tridimensional en el que el centro es el observador y la pantalla gira alrededor de él en función de dónde se encuentre el punto al que esté mirando. Para poder trabajar correctamente con su posición y orientación, hay que definirla como un plano dentro de este espacio, al ser un plano 2D carece de profundidad propia, pero al estar en un entorno 3D, posee coordenadas tridimensionales que la definen dentro del espacio. Para poder situarla bien, se han definido tres vectores que representan las direcciones de la pantalla:

- $\overrightarrow{F}$: indica la dirección que sigue el vector normal de la pantalla.
- $\overrightarrow{R}$: indica la dirección que adopta el eje $0U$ de la pantalla en el espacio XYZ
- $\overrightarrow{U}$: indica la dirección que adopta el eje $0V$ de la pantalla en el espacio XYZ.

Estos vectores se pueden calcular aplicando las siguientes fórmulas (3.30), (3.31) y (3.32):

$$\overrightarrow{F} = \frac{L_p - r_0}{|L_p - r_0|} \tag{3.29}$$

$$\overrightarrow{R} = \overrightarrow{0Y} \times \overrightarrow{F} \tag{3.30}$$

$$\overrightarrow{U} = \overrightarrow{F} \times \overrightarrow{R} \tag{3.31}$$

De esta manera se obtienen tres vectores perpendiculares entre sí, que indican la orientación de la pantalla en el espacio. Aplicando las ecuaciones (3.31) y (3.32) a la ecuación (3.29) se obtiene

```
//Camera Setup.
ro = vec(0., 2., -2.); //Ray Origin/Camera Position
lookat = vec(0., 0., 10.); //Lookat Point
zoom = 1.; //Camera Distance from Screen

f = normalize(lookat - ro); //forward vector
r = normalize(cross(vec(0., 1., 0.), f)); //right vector
u = cross(f, r); //up vector
c = ro + f*zoom; //Screen Center
i = c + uv.x*r + uv.y * u; //intersection point Ray-Screen
rd = normalize(i - ro); //Ray direction/Camera Direction
```

Tabla 3.2: Código de configuración de la cámara para ray marching.

la posición del píxel a renderizar en el espacio, de este modo se puede obtener el color de la escena para ese píxel. A continuación se muestra el código utilizado para la configuración de la cámara.

Una vez definida la cámara, lo siguiente es la definición de la función *ray intersection*. En este proceso se define también el avance del rayo, el cual avanza según el principio del *sphere tracing*, explicado anteriormente.

Para definir los elementos en el entorno, se recurre a la generación procedural de diferentes formas geométricas básicas por medio de funciones de distancia con signo. Para comenzar, se busca visualizar una esfera en mitad de un plano. Para dibujar el plano, tan solo hay que calcular la diferencia entre la altura de la cámara y la altura del plano. Para este ejemplo, la cámara está una unidad por encima del plano $0Y$, con lo que se visualizará este plano. Para poder visualizar la esfera, primero se calcula la distancia entre la esfera y la cámara, y después se le resta a esta distancia el radio de la esfera. Debido a que en el algoritmo no puede distinguir las caras internas de un objeto, hay que asegurar que el dato obtenido es el correspondiente al más cercano a la cámara, por ello se comparan todos los datos obtenidos por los rayos de todas las figuras de la escena y se escoge el más cercano para mostrar. A continuación se muestra el código utilizado para generar este ejemplo (tabla 3.4).

```
//RayMarch
d0 = 0.;
dP = 0.;
p = vec(0., 0., 0.);
for(i = 0.; i <100; i += 1.){
  p = ro + d0*rd;
  dS = length(p - vec(0., 1., 6.)) - 1.;
  dP = p.y;
  d0 += min(dS, dP);
  if(dS<0.001) break;
}
```

Tabla 3.3: Código de ray marching/sphere tracing.

La salida de este algoritmo es la distancia entre el observador y los elementos del entorno que chocan con los vectores r_d. Estos valores van más allá de 1, con lo que la visualización directa de

estos datos da como resultado una pantalla en blanco. Para poder visualizarlos bien, se recomienda dividir el resultado entre un número que asegure que el máximo valor alcanzado sea 1, el cual indicará que no hay nada en el trayecto del rayo correspondiente al píxel. En la figura 3.54 se muestra el resultado del algoritmo.

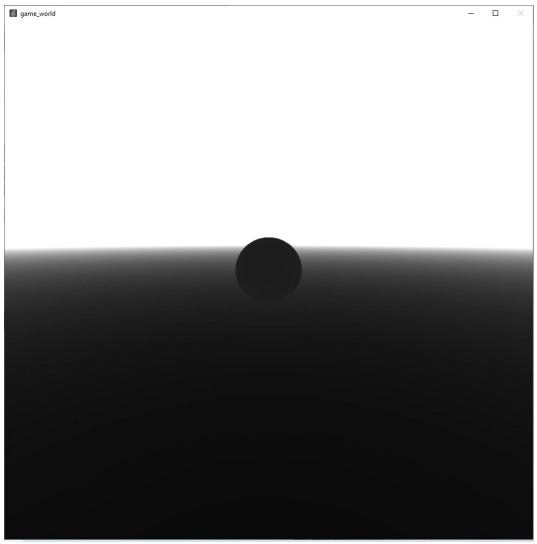

Figura 3.57: Distancias obtenidas por el algoritmo de *ray marching*.

Tras saber la distancia a la que están los objetos, el siguiente paso es darle a la escena una iluminación. Para este ejemplo se realizará una iluminación paralela, la cual consiste en un foco de luz muy alejado, de manera que los rayos que emite son paralelos unos a otros. Es el tipo de iluminación más simple que se puede dar, solo toma en cuenta el ángulo con el que choca con las superficies. Si los rayos de luz colisionan perpendicularmente a la superficie, esta absorbe la máxima cantidad de luz y adquiere el máximo grado de luminosidad. A medida que se va inclinando, esta luminosidad se pierde hasta llegar a estar fuera del contacto con la luz. Para calcular la luminosidad de las superficies es necesario calcular el vector normal de cada punto a renderizar y el vector de luminosidad, el cual es el que va del punto a renderizar al foco de luz. Comparando el ángulo que forman ambos se puede:

- conocer como de expuesta está la superficie a la luz.
- poder darle un valor de luminosidad.

Para poder calcular las componentes normales en cada punto, lo primero es calcular el vector que conecta la posición de la cámara, la dirección del rayo y la distancia de los objetos. Para ello se utilizará esta ecuación (3.33):

$$\overrightarrow{p} = r_0 + d * \overrightarrow{r_d} \tag{3.32}$$

Este vector p tiene la dirección de r_d, el origen en r_0 y la magnitud de d. Después, hay que aplicarle al vector $\overrightarrow{p}$ obtenido la siguiente fórmula (3.34):

$$\overrightarrow{n} = \begin{cases} 0X \rightarrow distance(\overrightarrow{p} + (ds, 0, 0)) - distance(\overrightarrow{p} + (-ds, 0, 0)) \\ 0Y \rightarrow distance(\overrightarrow{p} + (0, ds, 0)) - distance(\overrightarrow{p} + (0, -ds, 0)) \\ 0Z \rightarrow distance(\overrightarrow{p} + (0, 0, ds)) - distance(\overrightarrow{p} + (0, 0, -ds)) \end{cases} \tag{3.33}$$

Siendo $distance()$ la función de ray marching que devuelve la distancia de los objetos en la dirección del vector a utilizar y ds el desvío entre vectores. Esta ecuación calcula la diferencia entre las distancias obtenidas de los vectores desviados, la cual resulta en el valor de la componente normal en cada eje. A continuación se presenta el algoritmo empleado (tabla 3.5).

```
//Get Normal
get_normal(p, s){
  d = 0.0001;
  p = vec(0., 0., 0.);
  return normalize(vec(
  get_dist(p + vec(d, 0., 0.), s) - get_dist(p + vec(-d, 0., 0.), s),
  get_dist(p + vec(0., d, 0.), s) - get_dist(p + vec(0., -d, 0.), s),
  get_dist(p + vec(0., 0., d), s) - get_dist(p + vec(0., 0., -d), s)));
}
```

Tabla 3.4: Código para la obtención de vectores normales de la superficie.

Se puede observar que el plano adopta un color verde uniforme a lo largo de toda su superficie. Esto se debe a que cualquier punto del plano es paralelo al eje $0Y$ representado por el color verde. En cambio, la esfera ha adoptado un rango mayor de colores debido a que su superficie apunta en todas direcciones (fig. 3.58).

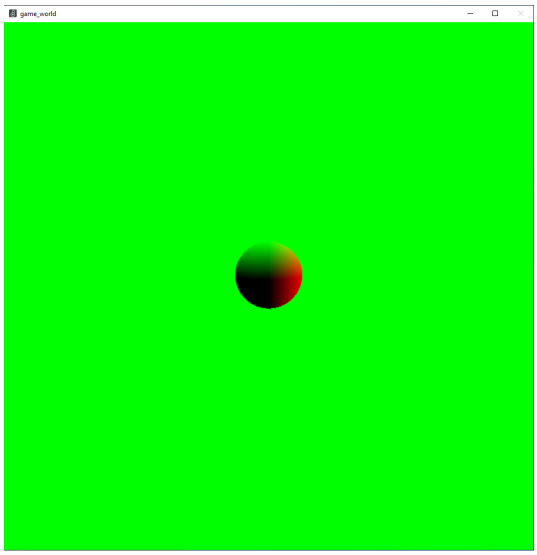

Figura 3.58: Vectores normales obtenidos

Conociendo los vectores normales del entorno, es el momento de calcular la iluminación de la escena. Para ello se ha dado una ubicación espacial a la fuente de luz y se han generado los vectores de luz siguiendo la siguiente ecuación (3.35):

$$\overrightarrow{l} = |l\_pos - \overrightarrow{p}| \tag{3.34}$$

De esta manera se consiguen los vectores que indican la dirección en la que chocará la luz con la superficie de la escena. Para hallar la intensidad de la luz en cada punto de la superficie, se realiza

el producto escalar entre los vectores normales y los vectores de luz, y para calcular las sombras, se realiza un segundo *ray marching*. Este tomará como posición inicial los puntos de las superficies del entorno y avanzará en la dirección de la luz, el cual devolverá la distancia que hay entre la superficie y la luz. En el caso de que el rayo choque con un objeto antes de llegar al foco, se sabrá que a ese punto no debe llegar la luz. El algoritmo empleado ha sido el siguiente (tabla 3.6):

```
//Get Lightning
get_light(p, s, time){
  l_pos = vec(sin(time) + 3., 5., 0. + cos(time));
  l = normalize(l_pos - p);
  n = get_normal(p, s);
  dif = (dot(n, l));
  d0 = RayMarch(p + n*0.2, l, s);
  if(d0<length(l_pos-p)) dif *= 0.1 ;
}
```

Tabla 3.5: Código de iluminación de la escena.

Los resultados del algoritmo se muestran en la figura 3.59.

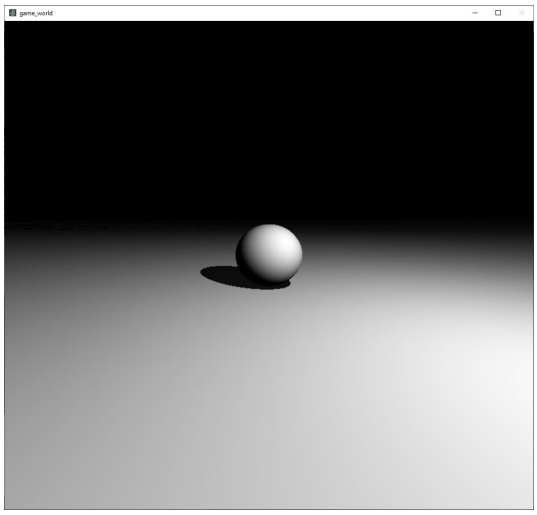

Figura 3.59: Escena iluminada.

El entorno básico ya está creado, Ahora para acabar tan solo hay que definir alguna figura más, con el fin de generar gráficos más variados (fig. 3.60).

Para concluir con esta prueba se ha probado con algunas piezas geométricas simples: cubo, cilindro, cápsula y toroide. Los algoritmos para conseguir estas figuras se consiguieron de Iñigo Quilez [53], experto en generación de gráficos por ordenador, a través de su página web.

3.2.6.3. Twisted Toro: Visión interna de una figura

El objetivo principal de este módulo es visualizar el interior de un toroide, darle una textura y hacer un anillo audio reactivo alrededor de su centro. Esta idea está basada en el proyecto *Torus*,

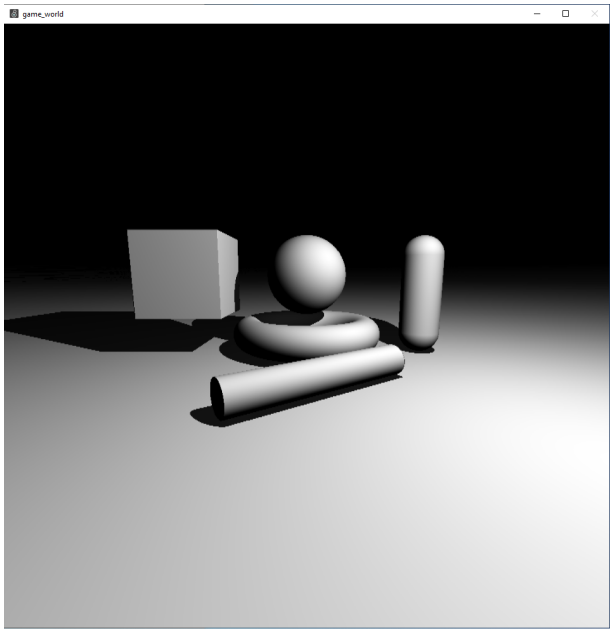

Figura 3.60: Escena iluminada con varias figuras.

de Reinder Nijhoff (fig. 3.61) [54].

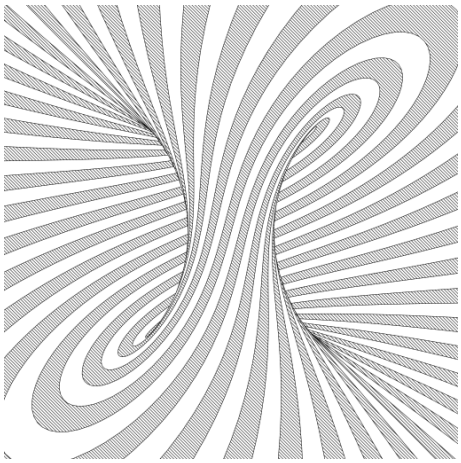

Figura 3.61: *Torus*, de Reinder Nijhoff (2019).

Para comenzar, se realizará un ray marching básico de un toroide, sin iluminación, ya que la intención es ver el interior de la figura. Para poder representar texturas sobre la figura es necesario poder tener un sistema de referencias adaptado a este. Al ser un figura con simetría radial, se utilizarán coordenadas polares, las cuales se obtienen de la siguiente manera (3.36):

$$\overrightarrow{n} = \begin{cases} \theta \rightarrow atan2(p_x, p_z) \\ r \rightarrow atan2(\sqrt{p_x^2 + p_z^2} - 1, p_y) \end{cases} \tag{3.35}$$

Siendo $\overrightarrow{p}$ el vector resultante del *ray marching*, la función $atan2()$ es la arcotangente de dos

parámetros, la cual calcula el ángulo alrededor del centro del toroide (θ) y las distancias del centro del toroide a la superficie del mismo (fig. 3.62).

Figura 3.62: Coordenadas polares del toroide: θ(derecha) y r(izquierda).

Estos valores van de $-\pi$ a π, con lo que si se multiplican los valores de las coordenadas, los valores se repiten a lo largo de las dimensiones del toroide. Luego se pueden mezclar ambos para dar lugar a una espiral alrededor del toroide con está ecuación:

$$col = sin(r * 10 + \theta * 20) \tag{3.36}$$

De esta manera se obtiene el patrón mostrado en la figura 3.63.

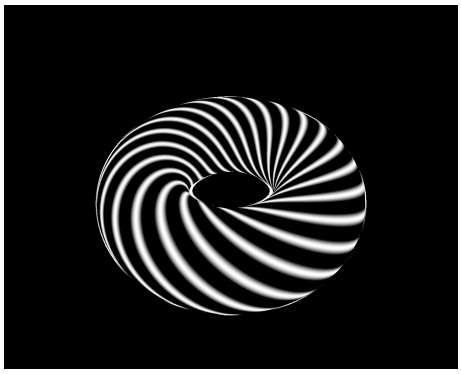

Figura 3.63: Espiral alrededor del toroide.

Para visualizarlo por dentro, hay que desplazar la posición de la cámara a una coordenada interna del toroide. Tal y como está el módulo, dentro de este no se podrá ver nada. Esto se debe a que el algoritmo de *ray marching* solo toma como positivas las distancias externas a la figura. Si la cámara se encuentra dentro de la función, las distancias obtenidas serán negativas. Para poder ver la figura por dentro, las distancias que tiene que devolver el algoritmo han de hacerse hacerse positivas, multiplicándolas por -1 (fig. 3.64).

Siguiendo el principio con el que se ha realizado el patrón en espiral, este se puede retocar un poco para hacer líneas más definidas, añadir más patrones dentro de estas y también para hacerlo variable con el tiempo (fig. 3.65).

Otras variaciones que se han realizado en la figura han sido (fig. 3.66):

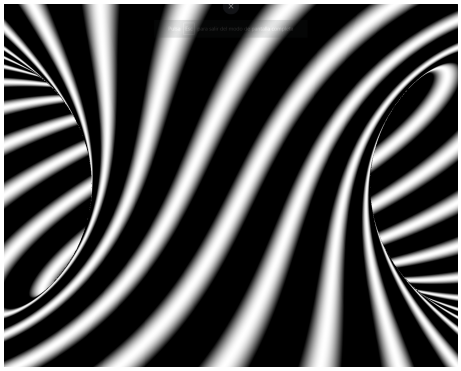

Figura 3.64: Vista interna del toroide.

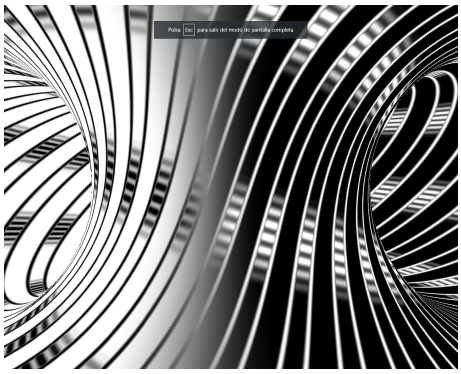

Figura 3.65: Vista interna del toroide con texturas aplicadas.

- La variación del radio hasta hacer desaparecer la columna interna.
- La variación del zoom para modificar la visión del interior del toroide.
- hacer el punto objetivo de la cámara móvil, para poder enfocar a los lados de la figura y girar la cámara.

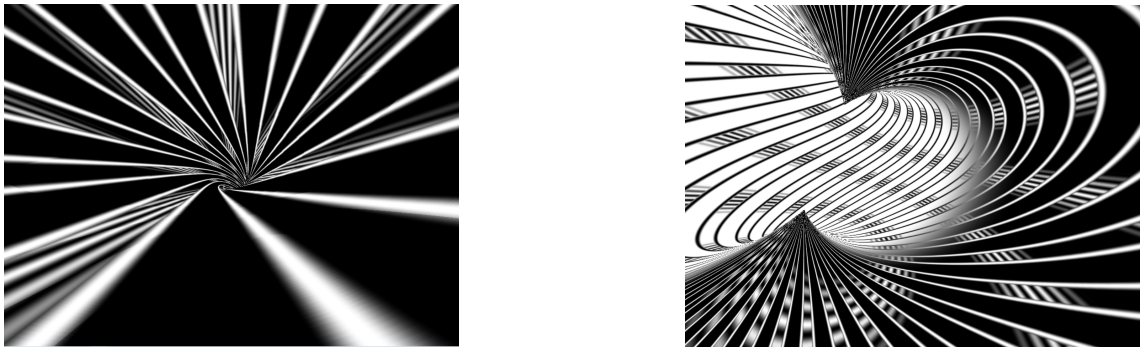

Figura 3.66: Variaciones de las dimensiones del toroide y de las propiedades de la cámara.

Hasta ahora, el módulo se mueve por la fluctuación del tiempo. Para poder hacer este módulo audio reactivo, se pensó en poder visualizar la señal de audio como un halo alrededor del centro del toroide. Para ello hay que volver a la fase del *ray marching*, en la que hay que hacer una función de distancia con signo para el halo de música. Se define el sistema de coordenadas de nuevo, y de nuevo serán coordenadas polares alrededor del centro del toroide. en torno a este se dibujará un nuevo toroide, el cual tendrá un radio interno igual al radio externo, de manera que no se percibirá por ahora. Del audio se muestreará la señal de audio sin procesar. Para que el audio se muestre, se modificará la componente de la altura del toroide, p_y, en función de los datos obtenidos de la señal de audio. Esto deja un halo que reacciona con el audio, pero no se puede ver. Para hacerlo visible, se decidió difuminar un poco los valores en los que se encontraba el halo por medio de los datos obtenidos por el espectro. De esta manera, también reacciona la intensidad con la que se vé y da la impresión que es el halo el que ilumina el toroide. Para no hacerlo todo blanco y negro, el halo oscila entre los colores rojo y azul. El resultado final ha sido el siguiente (fig. 3.67):

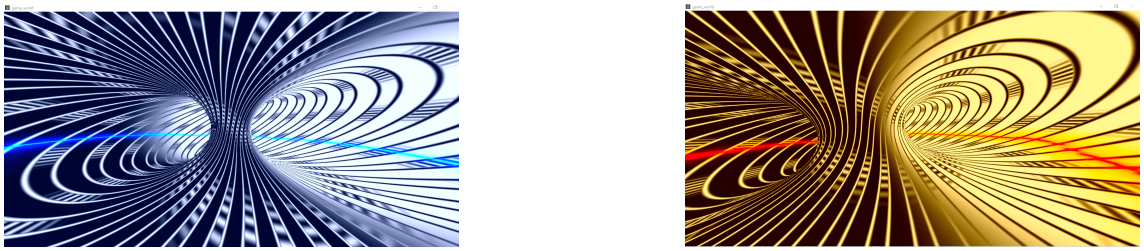

Figura 3.67: Variaciones de las dimensiones del toroide y de las propiedades de la cámara.

3.2.6.4. Spheres: Creación de múltiples objetos y efectos de reflexión

Para este módulo se partió de la idea de hacer un campo capaz de generar objetos hasta donde alcance la resolución de los píxeles de la pantalla [53].

La finalidad es la de poder crear muchos objetos a partir de la definición de uno solo, y poder distribuirlos por todo el espacio. Esto crearía escenarios muy interesantes con un coste computacional muy bajo. Para poder llevar este módulo a cabo, se se le aplicó al vector $\overrightarrow{p}$ la siguiente fórmula:

$$\overrightarrow{p'} = |fract(p) - 0{,}5| \tag{3.37}$$

Siendo $fract()$ una función para aislar los valores fraccionarios de los enteros. Esto produce que la figura se represente de manera repetida en una localidad del espacio. En la figura 3.65 se presenta el resultado del algoritmo con cubos.

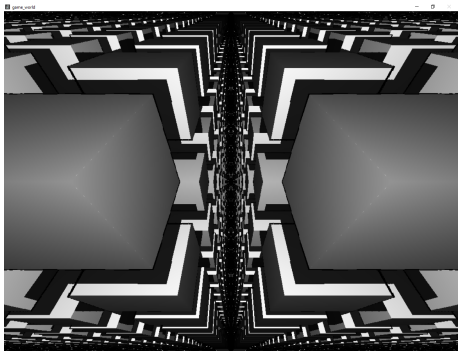

Figura 3.68: Generación de múltiples cubos.

El siguiente paso fue darle un poco de color a la escena, para evitar que se quedara con el color gris. Para poder cambiar el color de las figuras, tan solo hay que multiplicar el valor obtenido del algoritmo por el color que se desee, de esta manera, todas las figuras tendrán un color igual. Para poder diferenciar y dar un color diferente a cada una, se pueden hacer distinciones y procesar el color según la región del espacio, definido por $\overrightarrow{p}$, en la que se encuentren los objetos. Para este módulo se ha realizado un patrón de tablero de ajedrez que alterna los colores gris y morado. El resultado se presenta en la figura 3.69. De esta manera se ha obtenido el patrón con colores, luces y sombras.

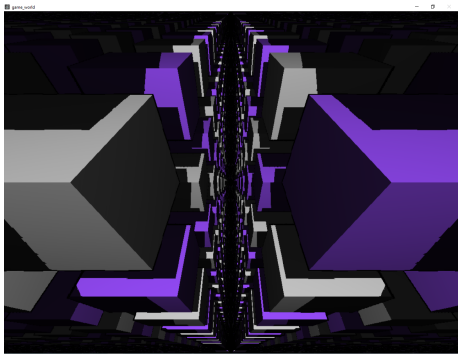

Figura 3.69: Coloración de los cubos por su posición en el espacio.

Otro objetivo de este módulo es darle una textura metálica a las figuras. Para ello hay que hacer un segundo *ray marching*. Con este se calculará el color de los rayos reflejados en las superficies de los objetos. Para este proceso hay que establecer como punto de origen las superficies de los

objetos y como dirección la reflexión del rayo $\overrightarrow{rd}$ sobre la superficie de los objetos. Esto da como resultado la distancia a la que se encuentran los objetos reflejados que llega a ver el usuario (fig. 3.70).

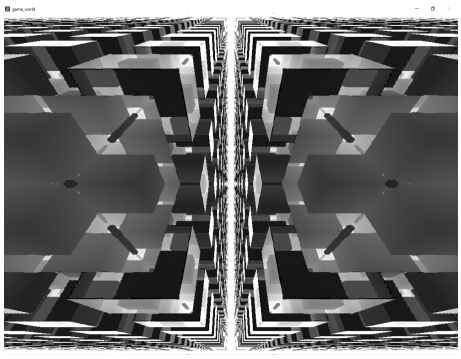

Figura 3.70: Cálculo de reflexiones.

Después de esto, se le aplican los colores y las sombras a los reflejos como en el proceso anterior y se suman los resultados de ambos procesos. El producto final es el siguiente (fig. 3.71):

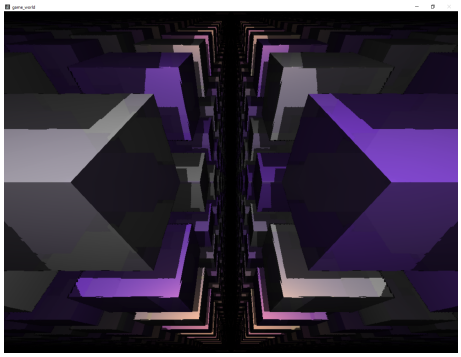

Figura 3.71: Módulo con colores, reflexiones y sombras.

Hay que remarcar que la reflexión en los cubos no es infinita. Solo se reflejan una vez. Para poder realizar más reflexiones habría que repetir este proceso más veces, pero eso supone un mayor cálculo de distancias y un mayor esfuerzo por parte de la CPU.

Una vez completado el algoritmo, se hicieron las siguientes correcciones sobre este:

- Debido a la falta de un filtro antialiasing, las aristas del cubo se veían pixeladas y un poco toscas. Por ello se cambiaron las formas de cubos a esferas.
- Se rebajó la intensidad de las sombras, ya que se perdía mucho color con ellas.
- Se le dio movilidad a la cámara y al punto de enfoque.

El resultado final ha sido el siguiente (fig 3.72).

Para hacerlo audio reactivo, se ha utilizado las señales de la respuesta rítmica para hacer que las esferas cambien de tamaño al ritmo de la música.

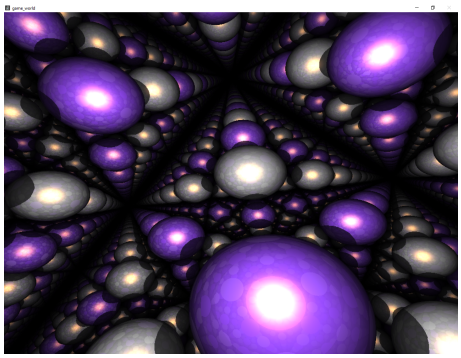

Figura 3.72: Módulo final con esferas.

3.2.6.5. Sonograph 3D: Espectro temporal de frecuencias

Este módulo está basado en el espectrograma de la herramienta de google, *Chrome Music Lab* (fig. 3.73) [55]. Este espectrograma tridimensional trata el espectro como un mapa de altitudes, y genera un plano que varia la altura de los puntos de su superficie correspondientes a los puntos del espectrograma con más intensidad.

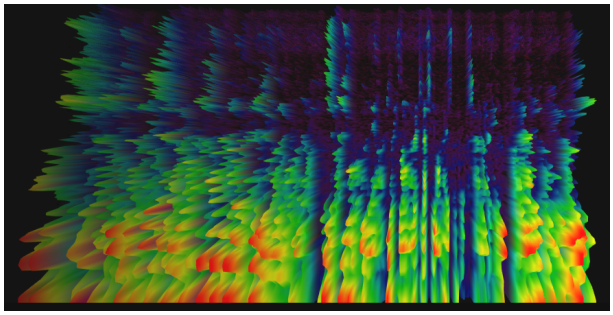

Figura 3.73: Espectrograma de Chrome Music Lab

Este módulo se consiguió realizar con la generación completa de gráficos (fig. 3.29), pero se tuvo que desechar la idea y la metodología por el coste computacional que requería por parte de la CPU (véase punto 3.2.2). En este módulo se planteó la creación de una malla y alterar las dimensiones en su componente $0Y$ en su superficie, pero a la hora de realizar gráficos a partir del *fragment shader* no se pueden cargar objetos. Tampoco se encontró una manera de generar mallas proceduralmente, con lo que para realizar este módulo con *fragment shader* se recurrió a un barrido de puntos en la pantalla.

Los pasos para seguir para la realización de este módulo fueron los siguientes:

1. Definir el espacio en el que se va a representar el espectrograma. Para ello se define un ortoedro a partir de las coordenadas de dos de sus esquinas opuestas y se buscan las intersecciones de rd con los seis lados del ortoedro. Se tendrán en cuenta las distancias más largas, ya que se representarán sobre estas. no se contarán con los lados más próximos a ro, ya que la visualización de estos ocultaría el espectrograma. De este punto interesa saber si rd ha chocado con la figura y la distancia a la que se encuentran los lados más alejados (fig. 3.74).

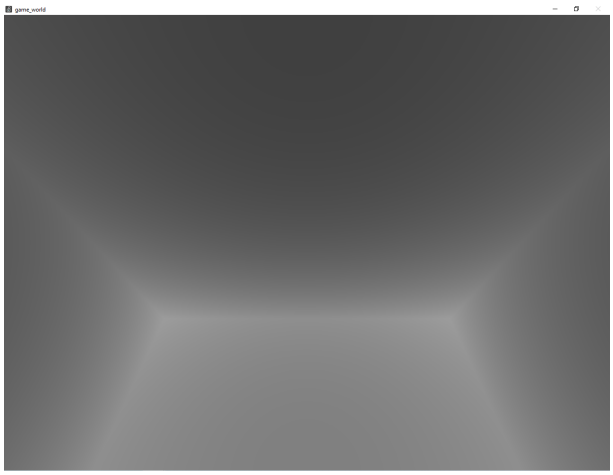

Figura 3.74: Entorno de desarrollo del espectrograma

2. Se procede a la generación del terreno de alturas con *ray marching*, este transforma los valores de intensidad del espectro en valores de altura que se plasman sobre un plano. Dentro del ortoedro calculado anteriormente, se hace una comparación entre los valores de intensidad obtenidos del espectrograma y las coordenadas verticales de la figura. Si estas coinciden, se devuelve una señal booleana indicando que en cierto punto del espacio, la altura del espectrograma y una coordenada del ortoedro han coincidido (fig. 3.75).

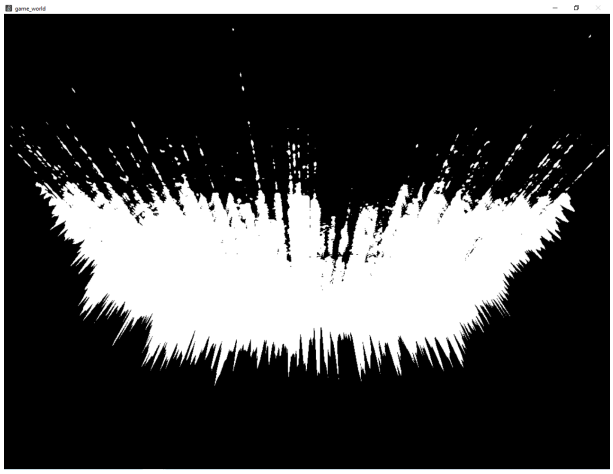

Figura 3.75: Altura del espectrograma.

3 Darle color a las alturas. En este caso se ha evitado recurrir a sombras, ya que podrían ocultar datos sobre el espectro. en lugar de eso se ha decidido utilizar una paleta de colores como las utilizadas en la representación del espectro en 2D (véase apartado 3.2.5.4). Para ello, se ha analizado las coordenadas del plano XZ de cada uno de los puntos destacados en el proceso anterior y se le ha dado el valor de color correspondiente a la intensidad en el mapa de colores (fig. 3.76).

Modificando un poco la posición de la cámara y las dimensiones de la pantalla, se obtiene el

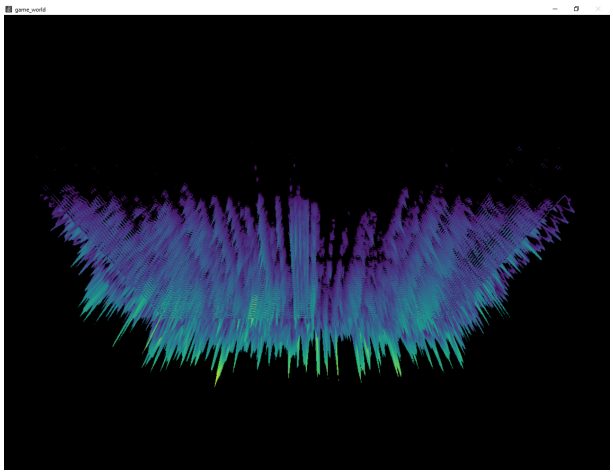

Figura 3.76: Espectrograma con el mapa de colores *viridis*.

resultado final para este módulo (fig. 3.77).

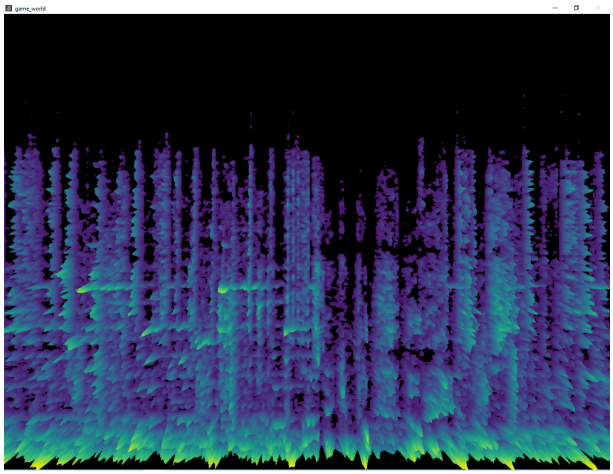

Figura 3.77: Espectrograma 3D

3.3. Implementación del módulo

3.3.1. Módulo de Soundcool

Una vez desarrollados todos los módulos que se querían realizar para este proyecto, era el momento de ponerlos todos juntos. Se partió de una plantilla pre-existente para realizar un módulo de Soundcool compatible con los demás módulos (fig. 3.78).

Este módulo consta de:

- Entrada y salida de audio/vídeo.

- Entrada para sensor kinect.
- puerto de conexión OSC.

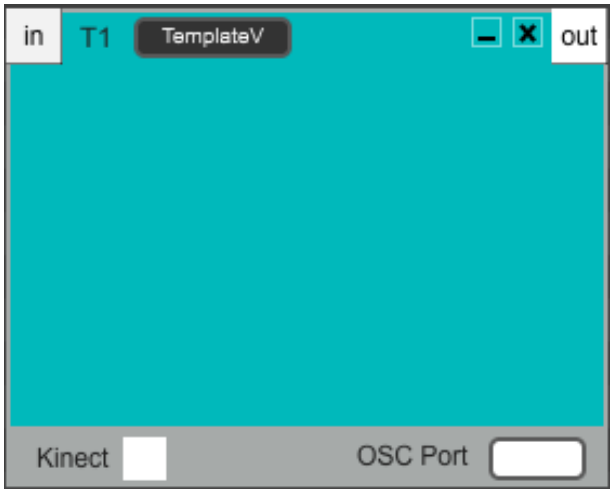

Figura 3.78: Plantilla para módulo de soundcool.

Lo primero que se hizo para adaptar este módulo fue configurar la salida de vídeo y la entrada de audio, ya que estas plantillas no mezclaban conceptos: si procesaban audio, emitían audio, y lo mismo para vídeo.

Debido a que todos los submódulos audioreactivos funcionan con el mismo entorno, lo primero que se hizo fue crear un entorno para este, el que controla dónde se generan los gráficos y la velocidad de renderizado de estos, después se trasladó el submódulo de procesado de audio, para este no hubo muchas complicaciones, ya que se diseñó dentro de un bloque con sus correspondientes señales de entrada y, como ninguna de sus salidas había de ser emitida al exterior, no fue necesario controlar demasiado el flujo de datos de salida del mismo, ya que si no se utiliza, no se ve reflejada en algún aspecto fundamental del módulo.

El proceso comienza a complicarse un poco con la inclusión de los módulos de vídeo, ya que cada uno de ellos requiere de una señal de reloj, la señal de audio con la que trabajen y algún dato extra para añadir expresividad, según el módulo. Para realizar un buen uso del rendimiento de los recursos de la CPU, estos módulos no pueden estar funcionando a la vez. Para poder regular el flujo de datos se utilizaron en conjunto los objetos gate y *preset* de MAX. El objeto gate habilita la entrada de datos si se le da la orden, generalmente con un objeto *toggle* o *bang*. El objeto *preset* guarda los valores de los objetos variables dentro de su alcance: si este se encuentra con *toggles*, *bangs*, *sliders* o *switches*, *preset* guardará los valores de estos en un apartado de memoria y se podrá recurrir a estos siempre que el usuario los seleccione. De esta forma se distribuyen las señales de entrada a los diferentes módulos sin tenerlos a todos generándose al mismo tiempo. A la salida se conectaron todos a la misma salida, ya que los módulos que no están activados no generan señal ni interfieren con el que esté funcionando.

Para que el usuario pueda seleccionar el submódulo que quiera, se recurrió al objeto *umenu*. Este objeto genera una pestaña desplegable en la que aparecen todos los módulos disponibles, el usuario solo tiene que seleccionar el que quiera y este se genera en la pantalla. Este objeto se utilizó

también para poder darle la posibilidad al usuario de seleccionar la resolución de cada módulo para que se adapte mejor a los recursos que tenga.

En algunos módulos se utilizan las señales generadas con ritmo. Para que el usuario pueda escogerlas, se ha habilitado un menú para estas en la interfaz. Aquí puede seleccionar que tipo de onda quiere: rampa, rampa inversa, cuadrada y sinusoidal, también se añadió la posibilidad de hacer cambios de manera manual y de seguir la rítmica de las canciones sin calcular el tempo, siguiendo solo las frecuencias graves de la pista de audio.

Se ha añadido también un barra para aumentar el volumen de la señal de audio. Este control no afecta al sonido que percibe el usuario, solo el que llega a los módulos audioreactivos. Este control amplifica la señal y si se aumenta en exceso, la satura. Esto supone un aumento en la cantidad de armónicos en el espectro de frecuencias, lo cual resulta para los módulos de análisis complejo, en un aumento de los colores mostrados por pantalla. Este control también permite la disminución del volumen.

Por último, se han añadido dos *sliders* que ajustan la magnitud con la que varían las figuras sometidas al ritmo y la memoria del detector de pulsaciones (véase 3.1.4).

Este módulo puede funcionar con otros módulos de la plataforma Soundcool: tan solo hay que conectar una fuente de audio a su entrada y a la salida un módulo de video para poder visualizar o utilizar los resultados obtenidos (fig. 3.77).

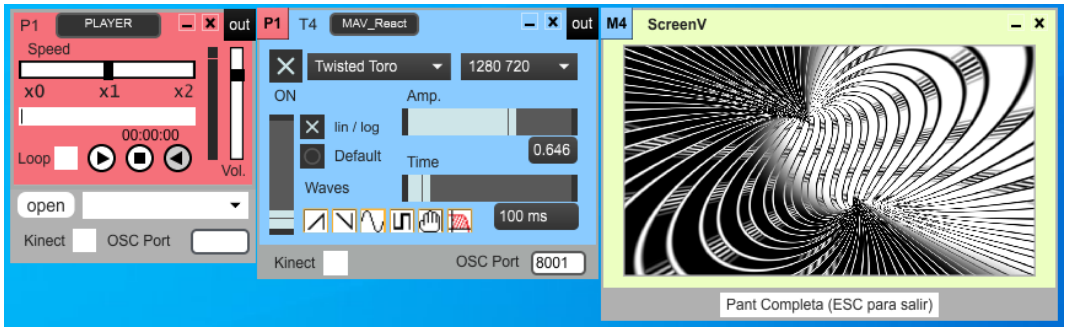

Figura 3.79: Módulo Audio-Reactivo completo con otros módulos de Soundcool.

3.3.2. Interfaz móvil de usuario

Soundcool se caracteriza por tener una interfaz móvil para cada uno de los módulos que posee. Estas interfaces permiten que el usuario pueda controlar el programa desde un dispositivo Android o iOS conectado a la red de internet. Las interfaces se comunican con el programa central por medio del protocolo de comunicación OSC (Open Sound Control), una alternativa mejorada al protocolo de comunicación MIDI. Este sistema fue desarrollado por CNMAT para poder comunicar funcionalidades entre instrumentos, ordenadores y otros dispositivos a través de una red a tiempo real [56].

Los mensajes están compuestos por una secuencia de bytes dividida en tres secciones:

- Dirección: Corresponde a la dirección de los datos. Se organiza igual que un sistema tipo URL, permitiendo la navegación en una estructura jerárquica.

- Tipo de datos: datos enteros (“i”), cadena de caracteres (“s”) o datos en coma flotante (“f”). Este tipo de datos no son utilizados en este trabajo.
- Datos del mensaje: En este trabajo, corresponden a los valores de salida de los objetos de la interfaz, que son los datos de entrada en el PC.

Para el desarrollo de esta interfaz se utiliza la librería UniOSC, basada en el protocolo OSC, sobre el motor de desarrollo Unity. Esta librería contiene los elementos necesarios para establecer conexión mantener una comunicación efectiva entre la interfaz y el módulo.

Debido a la existencia de otras interfaces, para acelerar el desarrollo de esta se han utilizado los bloques creados en versiones anteriores para crear una nueva y añadirla a la aplicación. De esta manera se ha realizado una interfaz funcional para el módulo que cumple con la estética de los demás módulos de Soundcool (fig. 3.78).

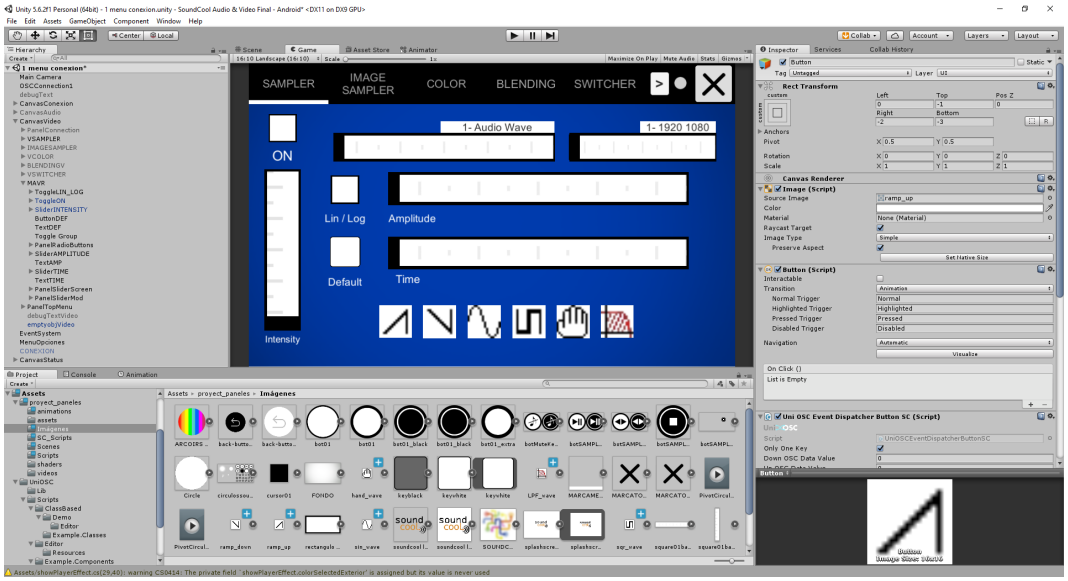

Figura 3.80: Interfaz del módulo en Unity.

La nueva interfaz, que controla el módulo realizado en el presente trabajo, formará parte de la siguiente versión de la aplicación de Soundcool.

Capítulo 4

Conclusiones

ha conseguido el objetivo propuesto para el trabajo, el cual era “el desarrollo del módulo audio reactivo para la plataforma Soundcool, capaz de generar gráficos que representen características del audio de forma visualmente atractiva”. Para alcanzar este objetivo previamente se tuvieron que conseguir los objetivos parciales, para los cuales se han obtenido las siguientes conclusiones:

- Para el primer objetivo, “analizar el audio para adquirir sus propiedades más notables”, se ha conseguido durante el análisis de audio la obtención de la señal de audio, su espectro de frecuencias, su espectro de frecuencias en el tiempo y las pulsaciones por minuto de la pieza que esté sonando, y exportarlo todo a señales de vídeo para su integración en el procesado de gráficos. Todos estos parámetros son clave a la hora de analizar una canción y para la generación coordinada de eventos basados en la música.
- En relación al segundo objetivo de la investigación “Generación de gráficos atendiendo a los valores obtenidos por el objetivo parcial anterior”, se ha logrado generar con éxito gráficos 3D y 2D que representen las características del audio, utilizando la menor cantidad de recursos de la CPU sin perder calidad en el proceso..
- Con respecto al tercer objetivo, “implementación completa del módulo y puesta en marcha dentro de la plataforma”, se ha conseguido implementar un módulo compatible con Soundcool capaz de albergar varios submódulos audio reactivos, con una interfaz de usuario sencilla y una implementación en MAX ordenada y clara, para que sea fácil orientarse dentro del módulo por parte de los nuevos desarrolladores del proyecto.
- Por último, “implementación de una interfaz móvil para el control del módulo”, se ha realizado una interfaz con las mismas herramientas y el mismo aspecto que el resto de los módulos con el fin de mantener el estilo de la aplicación.

Capítulo 5

Líneas futuras de trabajo

- Actualmente, el proyecto Soundcool se está integrando en HTML5 con el fin de alcanzar más sistemas operativos y eliminar las incompatibilidades haciendo del proyecto una aplicación web. Si este módulo quiere ser una parte más del proyecto en un futuro, una línea de trabajo podría ser la implementación del mismo en HTML5.
- Continuar la expansión de módulos audio reactivos. Hay muchas posibilidades en lo que respecta a la expresión sonora y al desarrollo de gráficos. A partir del presente trabajo, desarrolladores futuros del proyecto Soundcool podrán realizar nuevos módulos audio reactivos interesantes facilmente.
- Encontrar una manera de utilizar los módulos creados con las propias herramientas de Jitter rebajando el consumo de recursos de la CPU. Como se dijo anteriormente, este método se descartó por los recursos que consumía, pero la metodología que tiene para generar entornos es muy rápida, sencilla y ofrece resultados con muy buena calidad. Sería interesante aprovecharlos en un futuro.

El presente trabajo forma parte del proyecto Soundcool: Moviles y Herramientas Digitales para la Educación Artística Musical y Audiovisual (16-AC-2016) y todo el trabajo realizado será parte del sistema en la siguiente versión. Con ello se ha cubierto una necesidad importante de visualización atractiva del audio, con aplicaciones tanto educativas como profesionales.

Capítulo 6

Presupuesto

Para calcular el presupuesto del proyecto, se ha tenido en cuenta el salario del ingeniero [57], el coste del material y el precio de la licencia de MAX/MSP.

	Uds.	Precio	Horas	€
Ingeniero	1	11.44€/h	480	5491.2€
PC	1	800€	n/a	800€
Licencia académica MAX/MSP	1	250€	n/a	250€
			TOTAL	6541.2€

Tabla 6.1: Presupuesto.

Bibliografía

[1] Soundcool.
http://soundcool.org

[2] Are Europeans glued to their screens? - eurostat
https://ec.europa.eu/eurostat/web/products-eurostat-news/-/DDN-20180507-1

[3] OpenSound Control (OSC) - Wikipedia
https://es.wikipedia.org/wiki/OpenSound_Control

[4] 'La mare dels peixos', la primera ópera en valenciano que sale al extranjero - El Mundo
https://www.elmundo.es/comunidad-valenciana/alicante/2019/11/20/5dd4348afdddff9eaf8b46be.html

[5] Cycling '74.
https://cycling74.com/

[6] MIDI on the ATtiny
https://mitxela.com/projects/midi_on_the_attiny

[7] Protocolo DMX
https://www.etsist.upm.es/estaticos/ingeniatic/index.php/tecnologias/item/558-protocolo-dmx-digital-multiplex.html

[8] Ableton - MAX for Live
https://www.ableton.com/en/live/max-for-live/

[9] Arduino + MAX/MSP
https://playground.arduino.cc/Interfacing/MaxMSP/

[10] USB (Bus de serie universal)
https://es.ccm.net/contents/407-usb-bus-de-serie-universal

[11] A Functional Listing of all MSP Objects.
https://docs.cycling74.com/max7/vignettes/msp_functiona

[12] MAX 8 Tutorials.
https://www.youtube.com/watch?v=TO8cRfKT624&list=PLVIa8UkRzErsL95NoKH0QFaoLVMFqxbnA

[13] Jitter: What is a Matrix?
https://docs.cycling74.com/max7/tutorials/jitterchapter00a_whatisamatrix

[14] jit.catch~ Reference
https://docs.cycling74.com/max5/refpages/jit-ref/jit.catch~.html

[15] Espectro de frecuencias
https://es.wikipedia.org/wiki/Espectro_de_frecuencias

[16] Transformada de Fourier.
https://es.wikipedia.org/wiki/Transformada_de_Fourier

[17] Series de Fourier
https://es.wikipedia.org/wiki/Serie_de_Fourier

[18] Espectrograma
https://es.wikipedia.org/wiki/Espectrograma

[19] pfft~ Reference
https://docs.cycling74.com/max5/refpages/msp-ref/pfft~.html

[20] Muestreo: Frecuencia de Muestreo de Nyquist
https://www.youtube.com/watch?v=eNZv07rkdIU

[21] jit.gl.slab Reference
https://docs.cycling74.com/max5/refpages/jit-ref/jit.gl.slab.html

[22] Portal:OpenGL Concepts
https://www.khronos.org/opengl/wiki/Portal:OpenGL_Concepts

[23] Learn OpenGL - Joey de Vries
https://learnopengl.com/book/offline%20learnopengl.pdf

[24] The book of Shaders - Patricio Gonzalez Vivo & Jen Lowe
https://thebookofshaders.com/

[25] jit.gl.pix Reference
https://docs.cycling74.com/max7/refpages/jit.gl.pix

[26] Definición de ritmo musical.
https://definicion.de/ritmo-musical/

[27] A. Cipriani & M. Giri. *Electronic Music and Sound Design - Theory and practice with MAX* (Vol. 2). ConTempoNet s.a.s., 2013. ISBN: 978-88-905484-4-4.

[28] beatitude~ - Olivier Pasquet
https://www.opasquet.fr/op-beatitude/

[29] Programming Max: Structuring Interactive Software for Digital Arts
https://www.kadenze.com/courses/programming-max-structuring-interactive-software-for-digital-arts/info

[30] F. Colasanto. *Max/MSP: guía de programación para artistas*. CMMAS, 2010. ISBN: 978-607-00-3163-2.

[31] Rendering Pipeline Overview - Kronos Group
https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

[32] jit.world Reference
https://docs.cycling74.com/max7/refpages/jit.world

[33] jit.gl.gridshape Reference
https://docs.cycling74.com/max7/refpages/jit.gl.gridshape

[34] Amazing MAX Stuff - Federico Foderaro
https://www.federicofoderaro.com/

[35] jit.gl.handle Reference
https://docs.cycling74.com/max5/refpages/jit-ref/jit.gl.handle.html

[36] jit.gl.camera reference
https://docs.cycling74.com/max7/refpages/jit.gl.camera

[37] openGL: Texture
https://www.khronos.org/opengl/wiki/Texture

[38] jit.gl.material Reference
https://docs.cycling74.com/max7/refpages/jit.gl.material

[39] jit.gl.multiple Reference
https://docs.cycling74.com/max7/refpages/jit.gl.multiple

[40] Hardware para novatos: ¿qué es y cómo funciona la GPU?
https://hipertextual.com/archivo/2013/12/hardware-gpu-grafica/

[41] ¿GPU vs. CPU? ¿Qué es la computación por GPU? | NVIDIA
https://www.nvidia.com/es-la/drivers/what-is-gpu-computing/

[42] R.J. Rost. *OpenGL Shading Language* (2nd Edition). Addison Wesley Professional, 2006. ISBN: 978-0-321-33489-3.

[43] Rendering Pipeline Overview - Kronos Group
https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

[44] The viridis color palettes

https://cran.r-project.org/web/packages/viridis/vignettes/intro-to-viridis.html

[45] matplotlib
https://matplotlib.org/

[46] Python
https://www.python.org/

[47] Github - Berkeley Institute for Data Science - colormap
https://github.com/BIDS/colormap/blob/master/colormaps.py

[48] Complex Analysis
https://en.wikipedia.org/wiki/Complex_analysis

[49] GPU Gems. Chapter 8: Per-Pixel Displacement Mapping with Distance Functions - NVIDIA
https://developer.nvidia.com/gpugems/gpugems2/part-i-geometric-complexity/chapter-8-pixel-displacement-mapping-distance-functions

[50] Introduction to Ray Tracing: a Simple Method for Creating 3D Images
https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/implementing-the-raytracing-algorithm?url=3d-basic-rendering/introduction-to-ray-tracing/implementing-the-raytracing-algorithm

[51] Ray tracing (graphics)
https://www.wikiwand.com/en/Ray_tracing_(graphics)

[52] Ray Marching and signed Distance Functions - Jamie Wong
http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/

[53] Distance Functions - Iñigo Quílez
http://iquilezles.org/www/articles/distfunctions/distfunctions.htm

[54] Torus. Created by Reinder Nijhoff 2019
https://turtletoy.net/turtle/90e6288a6b

[55] Chrome Music Lab - Spectrogram
https://musiclab.chromeexperiments.com/Spectrogram/

[56] S. Moncho. *Soundcool: Open Sound Control, Smartphones, tablets y otros interfaces para la creación musical y visual*. Trabajo de fin de grado, UPV, 2018.

[57] Salarios para empleos de Ingeniero/a industrial en España
https://www.indeed.es/salaries/ingeniero-industrial-Salaries

Parte II

Anexo

Apéndice A

Listados adicionales

Se ha creado un repositorio donde se han incluido todas las pruebas que ha realizado el autor para la realización de este trabajo.

https://drive.google.com/drive/folders/1i5prvu8TdJIXkV24vq586mvD-3bMx-hL?usp=sharing

El módulo se integrará al proyecto Soundcool en marzo del 2020. El código fuente estará disponible de manera gratuita, junto al de todo el proyecto, en la página web oficial de Soundcool.

https://soundcool.org/descargas/