



TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN



Desenvolvemento dun sintetizador de son modular

Estudante: Juan Villaverde Rodríguez

Dirección: José Antonio García Naya

A Coruña, novembro de 2024.

A meus pais, que tanto fixeron por min.

Agradecementos

Grazas aos meus pais, Marcos e Beatriz, polo apoio, cariño e sacrificio durante todos estes anos. Sen vós, nunca tería chegado ata aquí. Grazas tamén aos meus amigos polo seu ánimo dende que comecei o proxecto. Finalmente, quero agradecer ao meu titor, José Antonio García Naya, pola súa dedicación e axuda durante estes meses de traballo.

Resumo

Este traballo céntrase no desenvolvemento dun sintetizador de son modular gratuíto para produción musical, motivado pola escaseza de alternativas similares accesibles económicamente. O obxectivo principal do proxecto é a creación dun sintetizador modular que sexa flexible e personalizable. O nome deste sintetizador é *Ocnet*, inspirado no meu nome artístico, *TeNco*.

Ademais, para garantir a súa utilidade, o sintetizador debe ser capaz de soportar cargas de traballo de moderadas a altas, polo que se prioriza a eficiencia dos algoritmos que impulsan cada módulo, buscando a súa máxima optimización. Tamén se quere asegurar que a ferramenta sexa compatible coas principais DAW (*digital audio workstation*) e sistemas operativos, evitando así limitacións de accesibilidade para os usuarios.

Outro aspecto fundamental é a documentación, que inclúe tanto a redacción desta memoria como unha documentación clara e ben estruturada do código para facilitar o seu uso e mantemento futuro. O desenvolvemento do sintetizador segue o framework JUCE e unha metodoloxía iterativa e incremental.

Abstract

This work focuses on the development of a free modular synthesizer for music production, driven by the lack of economically accessible alternatives. The main objective of the project is to create a modular synthesizer that is flexible and customizable. The name of this synthesizer will be *Ocnet*, inspired by my artist name, *TeNco*.

Furthermore, to ensure its usefulness, the synthesizer must be able to handle moderate to high workloads, so optimizing the efficiency of the algorithms powering each module will be a priority. It will also be important to ensure that the tool is compatible with major DAWs (*digital audio workstations*) and operating systems, avoiding accessibility limitations for users.

Another key aspect is the documentation, which includes both the writing of this report as well as clear and well structured comments within the code to facilitate its use and maintenance. The development of the synthesizer follows the JUCE framework and an iterative, incremental methodology.

Palabras clave:

- Sintetizador modular
- Producción musical
- Diseño sonoro
- Música
- Oscilador
- Modulación
- Efecto
- Eficiencia algorítmica
- Tempo real
- Transformada de Fourier

Keywords:

- Modular synthesizer
- Music production
- Sound design
- Music
- Oscillator
- Modulation
- Effect
- Algorithmic efficiency
- Real-time
- Fourier Transform

Índice Xeral

1	Introdución	1
1.1	Motivación	1
1.2	Introdución aos sintetizadores	1
1.3	Obxectivos	3
2	Fundamentos Teóricos	4
2.1	Que é o son?	4
2.2	Transformada de Fourier	6
2.3	Aliasing	7
3	Xestión do Proxecto	8
3.1	Metodoloxía	8
3.2	Planificación do traballo	9
3.3	Seguimento	10
3.4	Custo do proxecto	10
4	Análise	13
4.1	Mercado	13
4.2	Requisitos	14
4.3	Framework e outras ferramentas	16
5	Deseño	19
5.1	Arquitectura	19
5.1.1	Modelo	20
5.1.2	Vista	22
5.1.3	Controlador	25
5.2	Deseño da interface e interacción	25
5.2.1	Engadir un módulo	26

5.2.2	Mover un módulo	27
5.2.3	Eliminar un módulo	27
5.2.4	Silenciar un módulo	27
5.2.5	Tocar notas	28
5.2.6	Conectar unha modulación a un parámetro	28
5.2.7	Establecer valor de modulación e eliminación	29
5.3	Parámetros	30
6	Implementación	31
6.1	Introducción ao procesamento de son	31
6.2	Sistema de parámetros	32
6.3	Osciladores	34
6.3.1	Oscilador de táboas de onda	34
6.3.2	Sampler	44
6.4	Moduladores	46
6.4.1	Envolvente	46
6.4.2	Oscilador de baixa frecuencia (LFO)	47
6.4.3	Aleatorizador	48
6.4.4	Macro	49
6.5	Efectos	50
6.5.1	Distorsión	50
6.5.2	Filtro	52
6.5.3	Ecualizador	55
6.5.4	Reverberación	57
6.5.5	Delay	60
6.6	Parámetros xerais	61
6.6.1	Número de voces	61
6.6.2	Legato	62
6.7	Visualización do resultado final	64
7	Probas	66
7.1	JUCE Plugin Host	66
7.2	Probas software	66
7.2.1	Probas de validación	66
7.2.2	Probas de verificación	67
7.2.3	Probas de compatibilidade	67
7.2.4	Probas de usabilidade e calidad	67
7.3	Probas automatizadas	68

7.4	Detección de fugas de memoria	70
7.5	Probas de rendemento	71
7.6	Probas de son	71
8	Conclusións	73
9	Liñas Futuras	76
A	Algoritmos, códigos, scripts e saídas	78
A.1	Algoritmo de selección de voces	78
A.2	Algoritmo da FFT	83
A.3	Script de instalación	85
A.4	Saída de PluginVal	85
A.5	Detección de fugas de memoria	90
Relación de Acrónimos		92
Glosario		93
Bibliografía		94

Índice de Figuras

1.1	Dous sintetizadores.	2
1.2	Dous sintetizadores virtuais.	2
1.3	Dous dos digital audio workstation (DAW) más utilizados na actualidade.	3
2.1	Comparación de ondas con distinta frecuencia.	4
2.2	Comparación de ondas con distinta amplitud.	5
2.3	Comparación de dúas ondas con distinta forma.	5
2.4	Visualización da Transformada de Fourier (imaxé extraída de [1, Fig. 1]).	6
2.5	Frecuencias pregándose na frecuencia de Nyquist.	7
3.1	Diagrama da metodoloxía baseada no modelo iterativo incremental.	9
5.1	Diagrama MVC.	20
5.2	Diagrama xeral de clases do modelo.	22
5.3	Orde de procesamento dos procesadores de son. Cada cadrado corresponde a unha instancia dese procesador.	22
5.4	Diagrama GUI.	23
5.5	Diagrama GUI.	23
5.6	Diagramas de clases que especifican a conexión entre un parámetro e súa parte visual.	24
5.7	Diagrama de secuencia para engadir un módulo de distorsión.	25
5.8	Exemplo da interface con algúns módulos engadidos.	26
5.9	Exemplo da interface para engadir un Sampler.	26
5.10	Exemplo da interface para mover un módulo.	27
5.11	Exemplo da interface para silenciar un módulo.	28
5.12	Piano virtual.	28
5.13	Exemplo da interface para conectar unha modulación a un parámetro.	29
5.14	Exemplos de como modificar ou eliminar unha modulación.	29

6.1	Opcións que proporciona FL Studio [2] para o tamaño do <i>buffer</i> de son.	32
6.2	Exemplo dunha árbore de parámetros.	33
6.3	Ciclos das ondas que produce o sintetizador.	34
6.4	Simetría conxugada para unha onda cadrada.	36
6.5	Comparación entre dous factores de redución.	38
6.6	Exemplos de comportamento dun vectorscopio.	41
6.7	Representacións das leis de paneo circular e triangular.	41
6.8	Síntese FM.	42
6.9	Módulo do oscilador.	44
6.10	Pasos para instalar o VST.	45
6.11	Módulo do sampler.	45
6.12	Módulo da envolvente.	47
6.13	Módulo do LFO.	48
6.14	Función soft noise.	48
6.15	Función Sample and Hold.	49
6.16	Módulo do aleatorizador.	49
6.17	Módulo da macro.	50
6.18	Soft Clipping.	51
6.19	Hard Clipping	51
6.20	Módulo da distorsión.	52
6.21	Diagrama de bloques do filtro <i>IIR all-pass</i>	54
6.22	Filtro <i>all-pass</i> . Non modifica as frecuencias da sinal.	54
6.23	Diagrama de bloques do filtro.	55
6.24	Módulo do filtro.	55
6.25	Ecualizador de FL Studio que simula distintas configuracións das tres bandas permitidas por este sintetizador.	56
6.26	Comparativa da resposta en frecuencia dun filtro Butterworth fronte á dun Chebyshev.	56
6.28	Diagrama mostrando o comportamento das reflexións temperás, tardías e son directo.	57
6.27	Módulo do ecualizador.	57
6.29	Orde das operacións de difusión.	58
6.30	Delay multicanle.	58
6.31	Matriz de hadamard.	59
6.32	Inversor da polaridade.	59
6.33	Etapa de realimentación.	60
6.34	Diagrama de bloques completo da reverberación.	60

6.35	Módulo da reverberación	60
6.36	Diagrama de bloques do efecto de <i>delay</i>	61
6.37	Módulo de <i>delay</i>	61
6.38	Legato rápido actuando sobre o sintetizador en modo monofónico.	63
6.39	Legato lento actuando sobre o sintetizador en modo monofónico.	63
6.40	Legato rápido actuando sobre o sintetizador en modo polifónico.	63
6.41	Visualización da sección de osciladores.	65
6.42	Visualización da sección de efectos.	65
7.1	JUCE Plugin Host executándose directamente tras compilar o proxecto. A conexión vermella representa o sinal MIDI de entrada e as liñas de cor verde representan a saída estéreo do sintetizador. O nodo central é o <i>plugin</i> a probar, que se abre ao facer dobre clic enriba.	67
7.2	Interface gráfica da aplicación PluginVal co plugin que queremos probar seleccionado. Na barra de abajo pódese seleccionar o nivel de esixencia desexado.	69
7.3	Perfil dunha parte do algoritmo de reverberación.	71
7.4	vectorscopio, osciloscopio e analizador de espectro.	72

Índice de Táboas

3.1 Datas e horas de dedicación a cada fase.	10
3.2 Custo total do proxecto sen IVE nin custos indirectos.	12

Capítulo 1

Introducción

1.1 Motivación

A música está en constante evolución, impulsada pola aparición de novo software de producción musical que lles ofrece aos produtores a posibilidade de experimentar con sons únicos e explorar novas texturas sonoras. Na actualidade, existen moitas opcións de software, tanto de pagamento como gratuítas. Non obstante, as opcións gratuítas adoitan estar limitadas en funcionalidade ou, no caso dos sintetizadores modulares, son praticamente inexistentes.

A idea de unir a miña paixón pola producción musical e a informática xurdiu durante o desenvolvemento da segunda metade da materia optativa de *Dispositivos Hardware e Interfaces*. Neste contexto, decidín empregar o meu traballo de fin de grao para fusionar estes dous campos. Tras investigar o panorama actual do mercado de *plugins* de son que se empregan na producción musical, deime conta de que existe unha carencia de opcións gratuítas para sintetizadores modulares. Foi aí onde atopei unha oportunidade e, por iso, decidín desenvolver un sintetizador modular gratuíto que cubra esta necesidade.

Aos produtores musicais encántalles contar cos mellores *plugins*, especialmente aqueles que teñen calidades únicas. Esta proposta busca precisamente proporcionar unha ferramenta que non só sexa accesible, senón que permita aos usuarios personalizala e modificala segundo as súas necesidades creativas, ofrecendo así unha maior liberdade para explorar novas sonoridades. Este traballo busca achegar unha nova metodoloxía na que os produtores poidan experimentar con novas formas de son.

1.2 Introdución aos sintetizadores

Un sintetizador é un instrumento musical electrónico que permite xerar e modificar sinais de son en tempo real. Mediante varios procesos de síntese, un sintetizador é capaz de producir unha ampla gama de sons, dende imitaciós de instrumentos acústicos ata timbres totalmente

sintéticos.

O son é xerado mediante sinais eléctricos, que se transforman en ondas sonoras audibles. Estes sinais pódense modificar en tempo real a través de filtros, envolventes e outros parámetros. Polo xeral, os sintetizadores adoitan controlarse mediante un teclado, como un piano.



Figura 1.1: Dous sintetizadores.

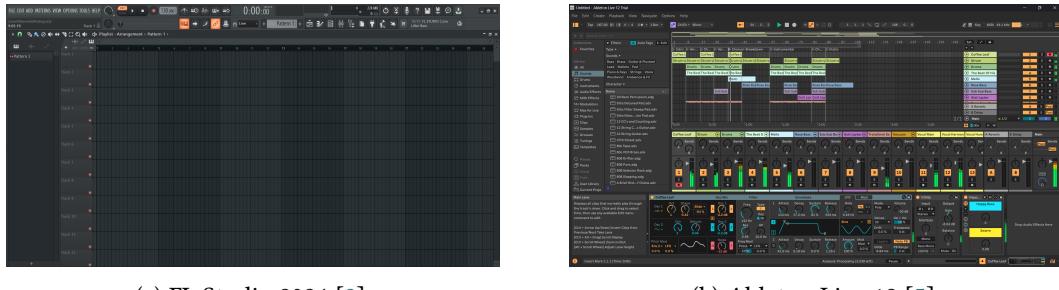
Un sintetizador virtual é unha versión software dun sintetizador tradicional. A diferenza dos sintetizadores físicos, que xeralmente empregan hardware especializado para a xeración de son, os de tipo virtual xéranlo a través de algoritmos software. Unha vantaxe do sintetizador virtual é a capacidade de gardar a configuración do sintetizador en arquivos, permitindo deste xeito restablecer o estado dos parámetros do sintetizador á configuración que permitía xerar un son concreto. Dous sintetizadores virtuais son Vital [3] e Phaseplant [4], que se poden ver na figura 1.2.



Figura 1.2: Dous sintetizadores virtuais.

Os sintetizadores virtuais adoitan integrarse dentro de estudos virtuais, tamén coñecidos como **DAW**. Para conectar o sintetizador cun **DAW**, faise uso dunha interface estándar. A máis común é a propia de Steinberg, chamada **Visual Studio Technology (VST)**. Mediante

unha interface, pódese crear un único software válido para todos os DAW. Exemplos de DAW son FL Studio [2] e Ableton Live [5], que se poden ver na figura 1.3.



(a) FL Studio 2024 [2].

(b) Ableton Live 12 [5].

Figura 1.3: Dous dos DAW más utilizados na actualidade.

1.3 Obxectivos

O principal obxectivo deste traballo é a creación dun sintetizador modular que sexa gratuíto, funcional, útil e robusto fronte a fallos. Un sintetizador modular é un tipo de sintetizador que permite aos usuarios conectar módulos individuais de maneira libre e flexible. Un módulo pode ser un oscilador, un efecto ou un modulador de parámetros. Este sintetizador poñerá a disposición dos usuarios distintos tipos de síntese, como a aditiva, subtractiva ou de **frequency modulation (FM)**, e permitirá crear sons orixinais cunha calidade e facilidade aceptables. A continuación, descríbense con máis detalle tanto este como outros obxectivos:

- **Creación dun sintetizador modular:** Este é o obxectivo principal do proxecto.
- **Eficiencia dos algoritmos:** Ademais da modularidade, para que o sintetizador sexa útil debe ser capaz de soportar cargas de trabalho de moderadas a altas, polo que os algoritmos detrás de cada módulo deben estar optimizados ao máximo posible.
- **Compatibilidade con varios DAW e equipos:** Para non limitar a accesibilidade deste software aos usuarios, é imprescindible que sexa capaz de executarse nos principais DAW e sistemas operativos.
- **Documentación software apropiada:** Non menos importante é a documentación, que inclúe a redacción desta memoria, así como os comentarios e a claridade dentro do código.

Capítulo 2

Fundamentos Teóricos

2.1 Que é o son?

O son é unha vibración que se propaga a través dun medio material (aire, auga ou un sólido), e que é percibida polo noso sistema auditivo. Estas vibracíons producen cambios de presión no medio, cambios que poden representarse matematicamente mediante funcións ou sinais. O son pode ser dixital ou analóxico, a diferenza principal entre os dous é que o primeiro tipo represéntase mediante sinais continuos, mentres que o segundo faino por medio de sinais discretos. Existen tres características intrínsecas do son que nos permiten diferenciar uns doutros:

- Ton: O ton depende da frecuencia de oscilación do sinal e permite diferenciar entre sons graves (menor frecuencia) ou agudos (maior frecuencia). Móstrase un exemplo de onda sinusoidal con distinta frecuencia na figura 2.1. A unidade de medida da frecuencia é o hercio (Hz).

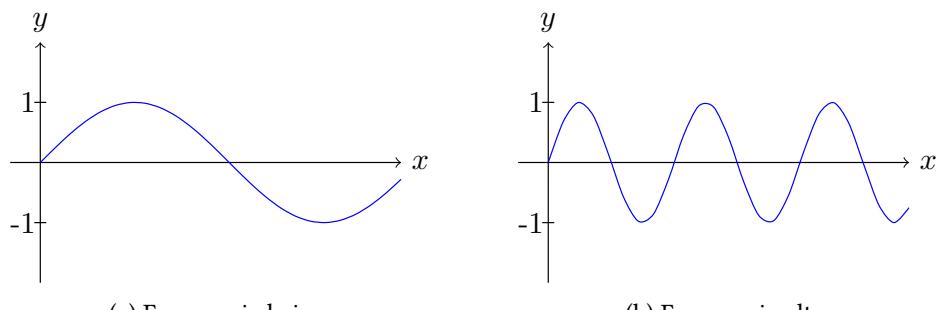


Figura 2.1: Comparación de ondas con distinta frecuencia.

- Amplitude: Determina o volume e a súa unidade de medida é o decibelio (dB), pero medido con respecto á presión sonora de 20 micropascais, o que tamén se denota como

dB_{SPL} [6]. Na figura 2.2 móstranse dúas ondas con distinta amplitude.

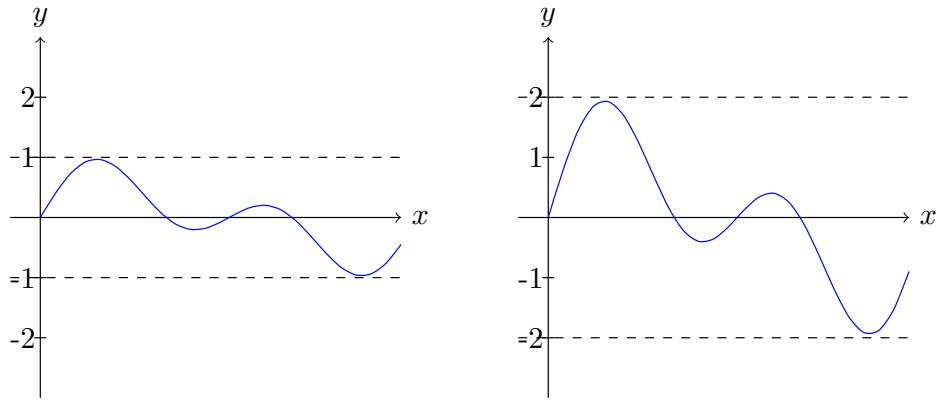


Figura 2.2: Comparación de ondas con distinta amplitud.

- Timbre: Directamente relacionado coa forma da onda, é a calidade que permite diferenciar, por exemplo, o son dunha guitarra do dun piano (ver figura 2.3).

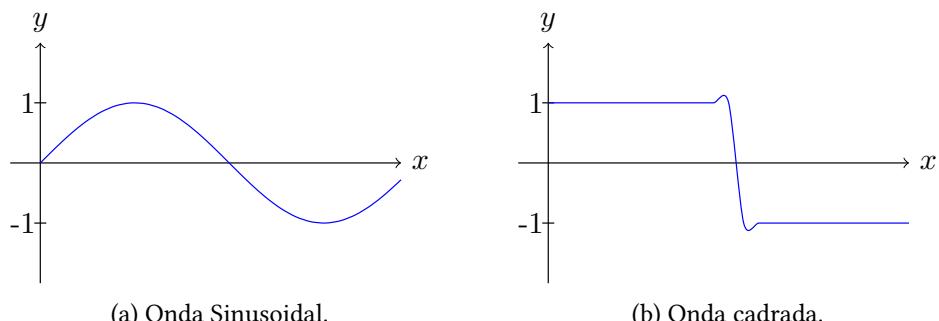


Figura 2.3: Comparación de dúas ondas con distinta forma.

O son co que se traballa nunha computadora é dixital, mentres que o que reproducen os altofalantes ou graban os micrófonos é analóxico. Isto implica un proceso de conversión en ambos sentidos. Da conversión do son analóxico a dixital encárganse os dispositivos coñecidos como [analog-to-digital converter \(ADC\)](#) e da dixital a analóxica, os [digital-to-analog converter \(DAC\)](#). Hai dous parámetros principais á hora de converter un sinal analóxico a dixital:

- Frecuencia de mostraxe: cantidade de mostras que se toman do sinal orixinal durante un segundo. Exprésase en hercios (Hz). Valores normais son 44100 Hz ou 48000 Hz.
- Profundidade de bit: nivel de cuantificación disponible para cada mostra. Cantos máis

bits dispoñibles para cada mostra, máis precisa será a súa representación. Os valores más comúns na actualidade son de 24 ou 32 bits por mostra.

2.2 Transformada de Fourier

O son máis puro posible é representado por un sinal sinusoidal. A combinación destes sons puros en distintas frecuencias, amplitudes e fases permite obter calquera outro son. É dicir, todos os sons poden ser descompuestos noutros más puros, representados por sinais sunusoidais. É mediante a descomposición en Series de Fourier que podemos representar un sinal complexo polas súas compoñentes más simples. No caso das notas musicais, as frecuencias destes sinais sinusoidais son múltiplos da frecuencia fundamental desa nota e denomináse-lles harmónicos. Por exemplo, chamámoslle LA á nota cuxa frecuencia fundamental (primeiro harmónico) é igual a 440 Hz.

- 1º harmónico (fundamental): 440 Hz
- 2º harmónico: $440 \text{ Hz} \times 2 = 880 \text{ Hz}$
- 3º harmónico: $440 \text{ Hz} \times 3 = 1.320 \text{ Hz}$
- 4º harmónico: $440 \text{ Hz} \times 4 = 1.760 \text{ Hz}$

Deste xeito, un sinal pode ser representado tanto no dominio do tempo ($x(t)$) como no da frecuencia ($X(F)$), segundo se ilustra na figura 2.4. Mediante o par de Transformadas de Fourier directa e inversa (2.1) podemos movernos entre os dous dominios.

$$X(F) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi F t} dt \Leftrightarrow x(t) = \int_{-\infty}^{\infty} X(F)e^{j2\pi F t} dF \quad (2.1)$$

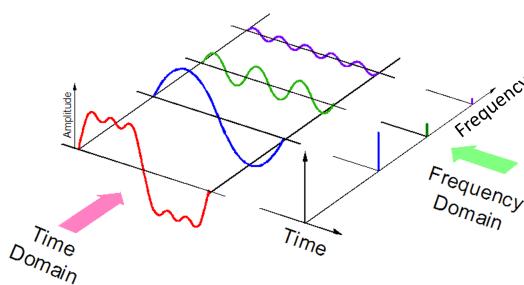


Figura 2.4: Visualización da Transformada de Fourier (imaxe extraída de [1, Fig. 1]).

A serie de Fourier dunha función periódica no tempo dá lugar a un espectro discreto de frecuencias e representa as compoñentes en forma complexa. Para sinais periódicos, a

descomposición en frecuencias realizase mediante a serie de Fourier, que é esencialmente unha versión discreta da Transformada de Fourier aplicada a funcións periódicas. Estas ideas serán tratadas a fondo máis adiante para explicar a implementación dos osciladores.

2.3 Aliasing

O aliasing é un fenómeno non desexado que se produce cando, no proceso de converter un sinal continuo a dixital, utilízase unha frecuencia de mostraxe insuficiente. Isto provoca que o sinal orixinal non se poida representar correctamente, dando lugar a distorsións e erros na súa reconstrucción. Para evitar o aliasing, faise uso do criterio de Nyquist-Shannon, que establece que a frecuencia de mostraxe debe ser, como mínimo, o dobre da frecuencia máxima presente no sinal orixinal.

Na figura 2.5 obsérvase o efecto do aliasing nos harmónicos superiores, que ao exceder a frecuencia de Nyquist, reflíctense cara ao espectro audible.

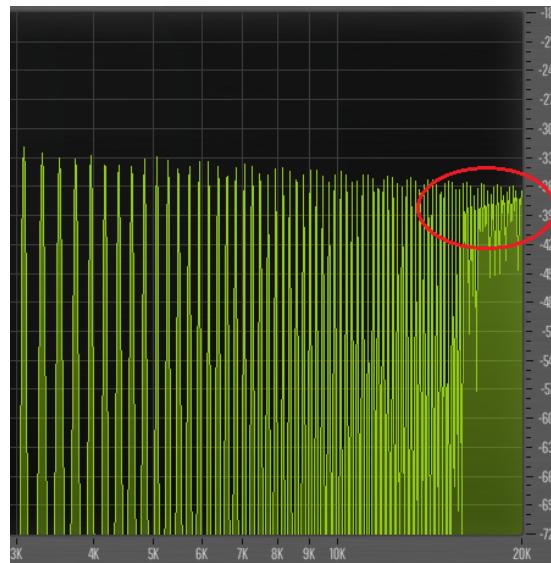


Figura 2.5: Frecuencias pregándose na frecuencia de Nyquist.

Capítulo 3

Xestión do Proxecto

3.1 Metodoloxía

Para o correcto e eficaz desenvolvemento deste proxecto, optouse por seguir a metodoloxía baseada no modelo iterativo incremental (ver figura 3.1). A base deste modelo é obter produtos funcionais (incrementos) en cada ciclo de desenvolvemento (iteración). Cada unha das iteracións pode verse como o desenvolvemento dunha versión reducida do proxecto que segue a metodoloxía en cascada, podendo dividirse cada unha delas en catro fases ben definidas.

- **Análise:** Ao principio de cada iteración definense os requisitos que debe cumplir o software resultante. Diferéncianse os requisitos funcionais, que responden á pregunta “Que debe facer?”, dos requisitos non funcionais (rendemento, seguridade, estética, etc.).
- **Deseño:** Planíficase como se vai implementar a nova funcionalidade. Identifícanse os compoñentes necesarios e como se conectarán entre si, facilitando a implementación e a comprensión do código.
- **Implementación:** Nesta fase lévase a cabo a codificación da funcionalidade, seguindo o deseño planificado.
- **Probas:** Verificase que o incremento cumple cos requisitos definidos na fase de análise (probas de validación) e que funciona correctamente (probas de verificación).

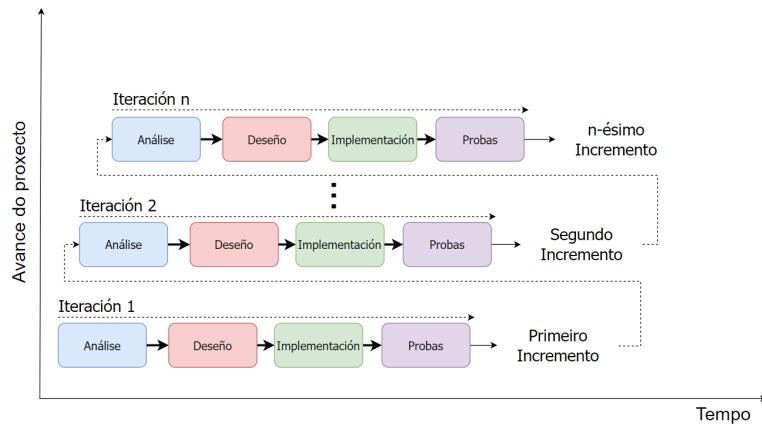


Figura 3.1: Diagrama da metodoloxía baseada no modelo iterativo incremental.

As vantaxes deste método de traballo son varias, entre as que destacan que cada incremento introduce unha parte nova ou mellorada do software de forma funcional, permitindo unha evolución continua do produto. Ademais, facilita a corrección de erros non detectados en iteracións anteriores, permitindo que estes se solucionen sen afectar ao avance global. Finalmente, promove a creación de módulos desacoplados entre si, o que simplifica o mantemento e a futura extensión do software.

3.2 Planificación do traballo

A planificación do proxecto implica establecer unha data de inicio e unha de entrega, ademas de definir a duración e o número de fases intermedias entre ambas. A data de inicio pode establecerse no momento en que xorde a idea do proxecto, mentres que a data de finalización corresponde ao día da entrega final. Para garantir unha execución organizada, o proxecto divídese en catro fases principais:

1. **Fase 1: Análise de requisitos e valoración inicial.** Nesta primeira fase defínense os requisitos do proxecto, tanto funcionais como non funcionais. Ademais, realiza unha valoración da complexidade do proxecto, avaliando se é factible dentro dos recursos dispoñibles (tempo, coñecemento, orzamento e equipo) e determinando que tecnoloxías serán necesarias para o seu desenvolvemento.
2. **Fase 2: Deseño de alto nivel.** Nesta fase decídense a metodoloxía que se vai a empregar e elabórase un deseño xeral da arquitectura do sistema, no que se mostre a alto nivel como interactúan os compoñentes entre si. Unha vez definidos os requisitos e a arquitectura, procede a fase de desenvolvemento na que, seguindo a metodoloxía antes

explicada, se obterá mediante iteracións e incrementos o software final que cumpla os requisitos establecidos na primeira fase.

3. **Fase 3: Desenvolvemento do software.** Unha vez definidos os requisitos e a arquitectura, procede a fase de desenvolvemento na que, seguindo a metodoloxía antes explicada, se obterá mediante iteracións e incrementos o software final que cumpla os requisitos establecidos na primeira fase.
4. **Fase 4: Desenvolvemento da memoria e entrega.** Nesta última fase procédese á elaboración da memoria do proxecto que documenta todos os aspectos do desenvolvemento. Finalmente, realizase a entrega do software final xunto co resto da documentación requirida.

3.3 Seguimento

A táboa 3.1 detalla as horas dedicadas a cada fase por parte do desenvolvedor. Consideréranse unicamente os días laborais (luns a venres).

Fase	Inicio	Fin	Media diaria (h)	Tempo (h)
1	17/05/2024	04/06/2024	1	13
2	04/06/2024	17/06/2024	2	18
3	17/06/2024	01/10/2024	2	152
4	01/10/2024	15/11/2024	4	132
Total	17/05/2024	15/11/2024	2,25	315

Táboa 3.1: Datas e horas de dedicación a cada fase.

3.4 Custo do proxecto

Para obter de maneira efectiva o custo total do proxecto debemos calcular, por un lado, os custos asociados ao equipo humano. No caso deste traballo fin de grao o equipo está formado polo estudiante, co rol de desenvolvedor do proxecto e a dirección do proxecto.

Baseándonos na información obtida de Glassdoor [7] e Jobted [8], e apoiándonos na calculadora de salarios da UDC, podemos establecer un custo anual que se distribúe en 21.600,00 € de salario bruto máis 3.600,00 € de pagas extra e 8.361,36 € de cuota patronal, o que da lugar a un custo total de 33.561,36 € anuais¹. Se supoñemos 1.720 h de traballo efectivo

¹ Neste caso supoñemos que o custo de contratación corresponde a un traballador do plantel, polo que non se ten en conta o custo asociado á indemnización pola finalización do contrato, que neste caso é igual a 859,81 €.

anuais (xa descontadas as vacacións), entón o custo por hora do desenvolvedor é igual a $33.561,36\text{€}/1.720\text{ h} = 19,51\text{€}/\text{h}$. Segundo a táboa 3.1, o desenvolvedor dedica 315 h ao proxecto, que a un custo de $19,51\text{€}/\text{h}$ resulta en $6.146,41\text{€}$.

Con respecto ao custo asociado ao director do proxecto, tomando novamente a información dispoñible en Glassdoor [7] e Jobted [8], establecemos un custo anual que se distribúe en 31.200€ de salario bruto máis $5.199,96\text{€}$ de pagas extra e $11.640,72\text{€}$ de cuota patronal, o que da lugar a un custo total de $48.040,68\text{€}$ anuais, que equivale a $48.040,68\text{€}/1.720\text{ h} = 27,93\text{€}/\text{h}$. Dado que a dirección do proxecto ten unha dedicación total de 30 h a un custo de $27,93\text{€}/\text{h}$ resulta en $837,92\text{€}$.

Por outra banda, debemos ter en conta tamén os custos asociados ás ferramentas software e hardware. No caso do software, evitouse empregar ferramentas que precisasen o pago dunha licenza, aproveitando no seu lugar as probas ou versións gratuítas dispoñibles. Sen embargo, hay que incluír o custo adicional de 230€ pola licenza de *FL Studio Producer Edition*, necesaria para poder engadir *plugins* externos, posto que a versión gratuita non o permite. Tamén se debe ter en conta que, en determinadas situacóns, é necesario incluír o custo asociado á **débeda tecnolóxica**, aínda que para este proxecto non se tivo en conta.

No caso do hardware, aínda que non foi necesario investir nun novo equipo (o desenvolvedor xa contaba cun portátil como ferramenta de traballo), debemos calcular un custo polo seu uso. Podemos considerar que a vida útil dun portátil é de 3 anos, polo que cada ano o custo de uso equivale a un terzo do seu valor de adquisición. Supoñemos un custo de adquisición dun portátil para un desenvolvedor de 1.500€ máis IVE. Dado que 315 h de traballo supoñen o 18,3 % das 1.720 h de traballo efectivas anuais, o custo asociado a este equipamento é igual a $(1.500\text{€}/3) \cdot 0,183\% = 91,57\text{€}$.

Polo tanto, o custo total de persoal do proxecto, sen ter en conta indemnizacións de fin de contrato, ascende a $6.146,41\text{€} + 837,92\text{€} = 6.984,33\text{€}$, sumados aos $91,57\text{€}$ de custo hardware e aos 230€ pola licenza de FL Studio, producen o custo total do proxecto, que é igual a $7.305,90\text{€}$, segundo se reflicte na táboa 3.2. O custo non inclúe a marxe comercial, outros custos indirectos de uso das instalacións (electricidade, auga, alugueres etc.) nin o IVE.

Recurso	Custo
Desenvolvedor novel	6.146,41 €
Dirección de proxecto	837,92 €
Equipamento informático	91,57 €
Licenzas software	230,00 €
Total	7.305,90 €

Táboa 3.2: Custo total do proxecto sen IVE nin custos indirectos.

Capítulo 4

Análise

Este capítulo corresponde á primeira fase descrita no capítulo 3 e describe os requisitos de software funcionais e non funcionais que deberá cumplir o produto final, decididos unha vez analizada a situación actual do mercado de software de son. Ao final indícanse as tecnoloxías escollidas para a realización do traballo.

4.1 Mercado

Para decidir o camiño que debe seguir o desenvolvemento deste sintetizador e facerse un oco no mercado, é necesario responder a unha serie de preguntas clave.

1. Que características buscan os usuarios? Isto inclúe explorar elementos como a variedade de sons preestablecidos, a capacidade de modulación, os efectos integrados e as opcións de personalización do sintetizador.
2. A que público irá dirixido? Debemos diferenciar entre produtores profesionais e aficionados á produción musical. Os primeiros están dispostos a investir grandes cantidades en sintetizadores con calidades únicas que cumpran uns estándares de calidad de son elevados. Os segundos non son tan exixentes na calidade de son e buscan un sintetizador accesible economicamente e versátil que lles permita, cunha mesma peza de software, acadar a maior cantidade de sons posibles. Unha característica común a ambos é o desexo por unha alta personalización.
3. Cales son as tendencias do mercado? A pesar do avance dixital, mantense un interese crecente polos sintetizadores analóxicos, grazas ao seu carácter e calidez sonora. O son *vintage* é especialmente atractivo para certos xéneros de música electrónica e experimental e a maioría dos sintetizadores dixitais proporcionan maneiras de emular esa calidade sonora orgánica. Ademais, obsérvase unha tendencia á modularidade, na

que moitos músicos prefieren sintetizadores que permitan engadir ou combinar módulos para personalizar ainda máis o son. Outras tendencias notables son o incremento de prezos e a incorporación de intelixencia artificial para a síntese de sons únicos de maneira instantánea.

4. Cales son os principais competidores no mercado? Actualmente, os sintetizadores dixitais más populares entre os produtores son Vital [3], Phaseplant [4] e Serum [9], cada un deles con características que os fan únicos. Analizar as súas funcións diferenciadoras pode inspirar e guiar a incorporación de novas características no noso sintetizador.
5. Cal é a importancia da compatibilidade con outros produtos de software de producción musical? A compatibilidade con DAW populares como Ableton Live [5], Logic Pro [10] ou FL Studio[2] é un factor crucial. Algúns usuarios prefieren un DAW en particular, mentres que outros optan por diferentes opcións. Incluso hai quen utiliza varios DAW.

4.2 Requisitos

Tras a análise do mercado, podemos establecer as características que formarán parte deste sintetizador. Optouse por un deseño modular que proporcionará unha alta capacidade de personalización, permitindo así a adaptación do sintetizador a diversos xéneros musicais e preferencias dos produtores.

Podemos subdividir o noso sintetizador en tres seccións fundamentais:

- A sección de osciladores, encargada de producir o son base
- A sección de efectos, que aplica modificacións ao son producido polos osciladores
- A sección de moduladores, que permite a automatización dos parámetros dos osciladores e dos efectos, aumentando a flexibilidade na producción de sons e posibilitando sons máis “orgánicos” ou “vintage”

Dentro de cada unha destas seccións inclúense módulos, que serán explicados en detalle en capítulos posteriores. Dentro da sección de osciladores o usuario poderá engadir:

- Oscilador de táboas de onda: un tipo especial de oscilador caracterizado por precalcular un ciclo da onda que se desexa reproducir. O oscilador debe proporcionar aos usuarios:
 - Síntese de FM: Un tipo de síntese que permite xerar sons metálicos.
 - Unísono: Permite a xeración de varias voces simultáneas, xerando un efecto de coro.
 - Control de volume.

- Control de panorama.
- Selector de oitava.
- Formas de onda básicas: Como a cadrada, sinusoidal e a de serra.
- Establecer a liña de efectos que actuará sobre o sinal de saída.
- *Sampler*: Un módulo que permite cargar archivos de son e reproducilos, ofrecendo tamén a posibilidade de xerar ruído branco.

En canto a efectos, o usuario poderá engadir:

- Distorsión: Engade textura ao son ao modificar a forma da onda, producindo un efecto de distorsión harmónica.
- Filtro: Permite eliminar certas frecuencias do son.
- Ecualizador: Axusta o balance de frecuencias, permitindo mellorar a claridade do son ou adaptalo ao contexto musical.
- Reverberación: Reproduce o efecto de espazo no son, simulando a resonancia dun espazo pechado como unha sala ou un auditorio.
- Delay: Crea ecos ou repeticións do son, engadindo ritmo e profundidade ao sinal.

En canto a moduladores:

- *Envelope*: Controla o desenvolvemento do son ao longo do tempo. Permite dar forma á evolución do son, definindo se é máis suave ou brusco.
- *low-frequency oscillator (LFO)*: Un oscilador de baixa frecuencia que permite a modulación de parámetros de forma periódica, creando efectos de vibrato, trémolo ou variacións no ton. É moi útil para crear movemento e dinamismo no son, facéndoo máis orgánico.
- Macro: Permite controlar varios parámetros simultaneamente cun único control.
- Aleatorizador: Xera cambios aleatorios nos parámetros, engadindo variabilidade.

Finalmente, fóra destas seccións, o sintetizador debe permitir o control sobre:

- Número de voces: Debe permitir establecer un comportamento monofónico (1 voz) ou polifónico (varias voces). O número de voces está directamente relacionado co número de notas que se poden pulsar simultaneamente.

- *Legato*: Esta técnica musical permite tocar notas de forma continua, sen interrupción entre elas, xerando unha transición suave. O usuario poderá activar ou desactivar esta función, e o sintetizador debe soportar tanto *legato* monofónico como polifónico.
- Calidade do son.

Ademais, para garantir a calidade e usabilidade do sintetizador, deben cumplirse os seguintes requisitos non funcionais:

- **Rendemento**: O sintetizador debe operar de forma eficiente, minimizando o uso de recursos do sistema e mantendo un tempo de resposta adecuado para a interacción en tempo real.
- **Usabilidade**: A interface debe ser intuitiva, de fácil navegación e accesible para usuarios de diferentes niveis de experiencia.
- **Atractivo visual aceptable**: A presentación visual debe ser clara e agradable, axudando a que os usuarios se sintan cómodos e inspirados ao usar o sintetizador.
- **Intuitivo**: O funcionamento do sintetizador debe ser natural e coherente, permitindo que os usuarios entanden rapidamente como configurar e manipular os diferentes módulos e efectos.

4.3 Framework e outras ferramentas

Existen varias opcións de *frameworks* e ferramentas para o desenvolvemento de sintetizadores de son, cada un con características específicas. Algunhas das opcións más populares actualmente son IPlug2[11], JUCE[12], Audiokit[13] e Dplug[14]

- **JUCE**: de código aberto, amplio soporte multiplataforma e de formatos de plugin. Debido á gran cantidade de características, resulta pesado e pode chegar a ter unha curva de aprendizaxe pronunciada. Emprega a linguaxe de programación C++. Conta cunha ampla documentación de moi boa calidade e cunha gran comunidade activa de desenvolvedores. Ten unha opción gratuita baixo a licenza de GPLv3 e outras dúas opcións de pago que permiten a distribución do software con código pechado.
- **IPlug2**: gratuito e de código aberto, con soporte multiplataforma, centrado en ser lixeiro e fácil de usar, pero cunha menor comunidade que JUCE. Non conta con demasiadas características, pero non está mal. Documentación escasa e de mala calidade. Gran comunidade.

- **AudioKit**: de código aberto, gratuito e fácil de usar, pero limitado a dispositivos de Apple (iOS e MacOS).
- **DPlug**: de código aberto, lixeiro, deseñado para facilitar o mantemento do código e non tanto para o desenvolvemento de novas características. Emprega a linguaxe de programación *D* e permite desenvolver *plugins* para varios formatos (VST2, VST3, AAX, CLAP etc.). A documentación é escasa e a comunidade pequena. Recibe actualizacóns constantes.

Tras analizar as calidades e diferenzas entre estes *frameworks*, finalmente optouse por JUCE [12] principalmente pola cantidade de documentación de calidad e a ampla comunidade moi activa que ten, o que facilitou e axilizou en gran medida o desenvolvemento deste sintetizador, pero tamén polo tamaño do propio *framework*, o soporte multiplataforma e as constantes actualizacóns que recibe. A súa principal limitación é que require coñecementos sólidos de C++, o que pode supoñer unha curva de aprendizaxe pronunciada para desenvolvedores sen experiencia previa nesta linguaxe.

En canto á licenza, JUCE ofrece diferentes modalidades dependendo dos ingresos anuais do proxecto ou entidade que o empregue. A versión *Starter* é gratuita, pero limita o uso a proxectos cuxos ingresos anuais sexan inferiores a 20.000 USD, e require que o proxecto estea baixo a licenza AGPLv3 [15], que obriga a liberar o código fonte. Para proxectos con ingresos anuais entre 20.000 USD e 300.000 USD é necesario adquirir unha licenza *Indie*, mentres que para proxectos sen límite de ingresos é obrigatorio dispoñer dunha licenza *Pro*. Estas licenzas permiten a distribución de software de código pechado mais, polo momento, non se ten pensado distribuír o sintetizador, e menos comercialmente, polo que se traballará baixo a licenza AGPLv3 [15].

A continuación, lístanse outras ferramentas software empregadas para o desenvolvemento deste traballo, incluíndo esta memoria:

- **Visual Studio 2022** [16]: integrated development environment (*IDE*) empregado para o desenvolvemento do proxecto.
- **Projucer**: *IDE* deseñado para crear e xestionar proxectos de JUCE (está incluído en JUCE). Permite a xeración de proxectos para *IDE* de terceiros como Visual Studio.
- **FL Studio 24** [2]: *DAW* empregado para probar o sintetizador nun contexto real. O sintetizador engádese a FL Studio como un *plugin* en formato VST e, aproveitándose de *plugins* de terceiros incluídos no *DAW*, realizanse medicións sobre o son, analizando o espectro de frecuencias, a imaxe estéreo, o volume e máis.
- **GitHub e Git** [17]: Para almacenar e xestionar o código.

- **Geogebra** [18]: Para visualizar e comprender as funcións matemáticas coas que se traballa.
- **Overleaf** [19]: Para a realización desta memoria.
- **Python** [20]: Para xerar gran parte das figuras que se mostran nesta memoria.
- **Balsamiq** [21]: Para a maquetación inicial da interface gráfica.
- **Draw.io** [22]: Para a creación de diagramas UML.
- **Discord** [23]: Utilizado para a comunicación con outros desenvolvedores de son e para a realización de reunións telemáticas con produtores musicais durante a fase de probas.
- **Teams**: Empregado para reunións telemáticas co titor e para a comunicación directa.

Capítulo 5

Deseño

En correspondencia coa segunda fase establecida na planificación do proxecto, este capítulo detalla o deseño e a arquitectura seguidos durante o desenvolvemento. Debido á natureza do modelo iterativo incremental, cada iteración inclúe a súa propia fase de deseño. Así, este capítulo combina o deseño inicial de alto nivel coas fases de deseño específicas de cada iteración realizadas ao longo do proxecto.

Antes de continuar, debemos aclarar que todas as clases e métodos que comecen polo prefijo `juce::` son propias do framework de JUCE [12].

5.1 Arquitectura

Para a arquitectura do sintetizador séguese unha adaptación ao modelo denominado [Model-View-Controller \(MVC\)](#), que divide unha aplicación en tres partes claramente diferenciadas. Por un lado, está o modelo, que se encarga de gardar e transformar os datos. Por outro lado, atópase a vista, que proporciona unha interfaz coa que o usuario pode interactuar. Finalmente, temos o Controlador, que actúa como intermediario entre a vista e o modelo, que xestiona o comportamento da vista e determina que accións debe executar o modelo en resposta ás interaccións do usuario. A vista comunica ao controlador as accións que realiza o usuario (como pulsar un botón, tocar unha nota ou mover un slider). O controlador, á súa vez, decide como procesar esa acción, para o que indica ao modelo que realice unha determinada operación e solicítalle á vista que se actualice segundo sexa necesario.

Debido a como está deseñado JUCE [12], non se pode aplicar de maneira directa o devandito patrón. Porén, segue sendo útil para a organización e mantemento do código. Ao crear un proxecto con Projucer, créanse automáticamente dúas clases fundamentais sobre as que se vai construír este modelo:

- `juce::AudioProcessor`: onde se realiza o procesamento do son.
- `juce::AudioEditor`: encargada da parte visual da aplicación.

Deste xeito, sobre a clase `juce::AudioProcessor` implementaremos o modelo da aplicación e transformamos `AudioEditor` no controlador da mesma. Para iso, facemos que esta última herde dunha nova clase chamada `Controller`, a cal está composta pola clase `GUI`, que conterá toda a parte visual do plugin.

Será mediante a clase `EventHandler`, creada manualmente, que a vista se comunique co controlador, indicando que acción deseja realizar o usuario. O controlador implementa os métodos declarados en `EventHandler` e os componentes da interfaz chamarán aos ditos métodos.

O diagrama [unified modeling language \(UML\)](#) da figura 5.1 mostra, conceptualmente, como se relacionan as clases entre si. A cor vermella fai referencia ás clases propias do modelo, a cor verde corresponde ás clases do controlador e as clases azuis pertencen á vista.

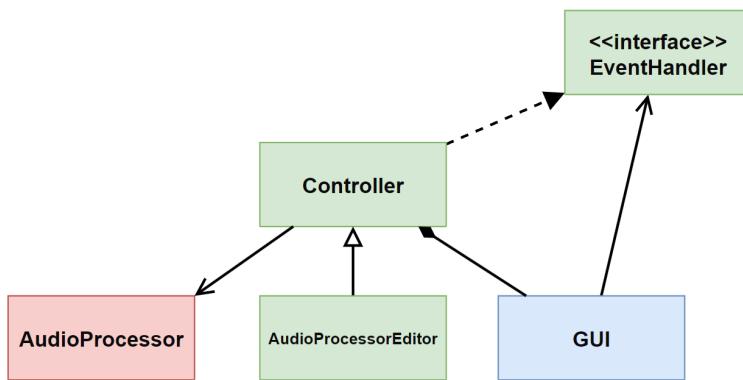


Figura 5.1: Diagrama MVC.

É común que no modelo [MVC](#) a vista sexa executada nun fío á parte, de xeito que a aplicación non se colgue nos momentos de cálculos intensivos por parte do modelo. JUCE proporciona dous fíos principais que separan modelo e vista:

- **Message Thread:** é o fío principal, encargado de xestionar a [graphical user interface \(GUI\)](#) e a cola de mensaxes (eventos, mensaxes doutros fíos, repitido etc).
- **Audio Thread:** é o encargado de xestionar o procesamento de son. Calquera retraso neste fío pode provocar interrupcións ou problemas no son.

5.1.1 Modelo

Como se comentou, no modelo realiza o procesamento de son. A clase principal é `juce::AudioProcessor`. O seu método principal, `juce::ProcessBlock(AudioBuffer& buffer)`, é executado en bucle polo `Audio Thread`, e é onde se debe implementar a lóxica de procesamento de son.

AudioProcessor inclúe dúas estruturas clave: un struct ProcessorInfo, que facilita a comunicación co DAW, e a clase ParameterHandler, que almacena e xestiona os parámetros. As dúas estructuras deben ser creadas manualmente polo programador.

AudioProcessor herda de OcnetSynthesiser, que á súa vez herda de juce::Synthesiser, unha clase de JUCE deseñada para crear sintetizadores polifónicos. Isto permite ter varias “vozes”, onde cada voz é unha instancia independente que xera o son para unha nota específica. Así, cada vez que se toca unha nota no sintetizador, asignaselle unha voz para reproducila. O número é fixo e limitado. A asignación dunha voz faise por medio dun algoritmo, que ha de ser modificado, como se explicará na implementación.

As voces están representadas pola clase SynthVoice, que herda e sobrescribe métodos de juce::SynthesiserVoice. Para que estas voces poidan producir son, é necesario ter polo menos unha instancia de juce::SynthesiserSound.

juce::SynthesiserSound é unha clase en JUCE que representa o tipo de son que un sintetizador pode producir e permite especificar parámetros como o rango de notas ou se unha voz determinada pode ou non tocar ese son. Neste sintetizador non faremos uso das súas vantaxes, pero segue sendo necesario polo menos unha instancia desta clase.

Chamámoslle procesador de son (Processor) á parte do módulo que forma parte do modelo da aplicación, é dicir, á sección do módulo encargada de xerar ou modificar o son. A orde de procesamento destes procesadores de son é crucial para o correcto funcionamiento do sistema: primeiro deben executarse os moduladores, despois os osciladores e, finalmente, os efectos. Cando se engade un oscilador ou modulador, engádese realmente unha instancia dese procesador por cada voz do sintetizador. Isto débese á independencia entre voces mencionada anteriormente, onde cada voz xestiona os seus propios osciladores e moduladores. No caso dos efectos, non se aplica esta regra, xa que deben aplicarse sobre a suma final de todas as voces, polo que unha única instancia xestionada por AudioProcessor é suficiente. Na figura 5.3 móstrase visualmente esta orde de procesamento.

Ademais, as voces procésanse secuencialmente, e a elección da orde de procesamento depende do algoritmo de selección de voces. JUCE proporciona un algoritmo básico para seleccionar esta orde, mais este debe ser adaptado para considerar a futura inclusión de legato.

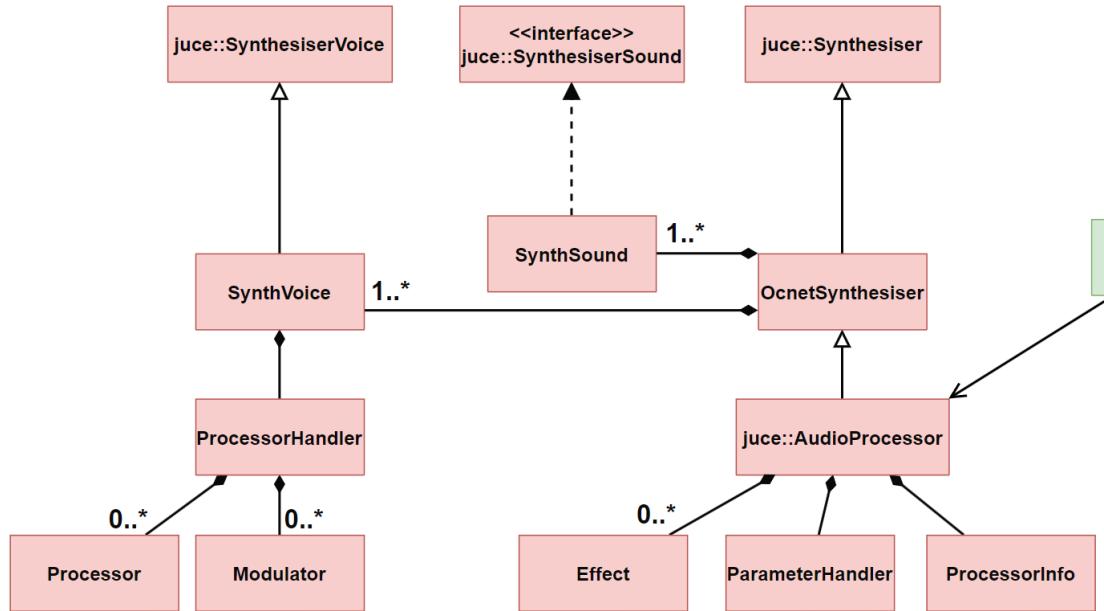


Figura 5.2: Diagrama xeral de clases do modelo.

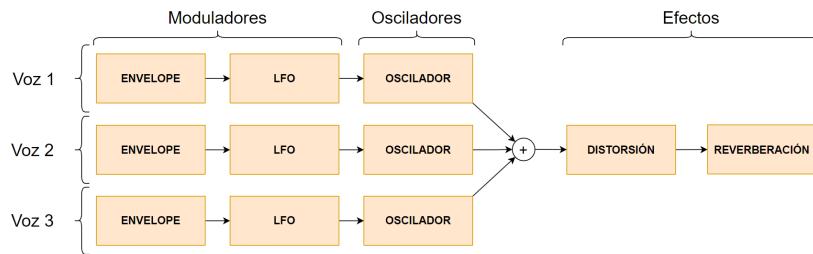


Figura 5.3: Orde de procesamento dos procesadores de son. Cada cadrado corresponde a unha instancia dese procesador.

5.1.2 Vista

A arquitectura da vista segue o principio de composición sobre herdanza, o que permite desacoplar as distintas partes da interface gráfica (ver figura 5.4). A clase GUI está composta por diversas seccións (Section), e estas seccións, á súa vez, poden estar compostas por ningunha ou varias subseccións (Subsection). Tanto as subseccións como as seccións manteñen unha referencia a EventHandler, que se encarga de xestionar os eventos da interface. As subseccións seguen a seguinte xerarquía de herdanza (ver figura 5.5): cada subsección representa a parte visual dun módulo e está composta por controis manexables polo usuario. Estes controis son tres:

- Slider: Un slider é un control deslizante que permite establecer un valor dentro dun

rango determinado. Visualmente, pode adoptar dúas formas distintas, definidas polas clases Knob1 e Knob2.

- ComboBox: Un combo box é un menú despregable que permite seleccionar unha opción entre varias dispoñibles.
- Button: Un botón pode ser de dous tipos: un botón tipo *toggle* (permite activar e desactivar unha función) ou un botón normal (executa unha acción ao ser premedo).

Por simplicidade, o diagrama da figura 5.5 non amosa todos os posibles tipos de subseccións nin todos os tipos de elementos que componen cada unha.

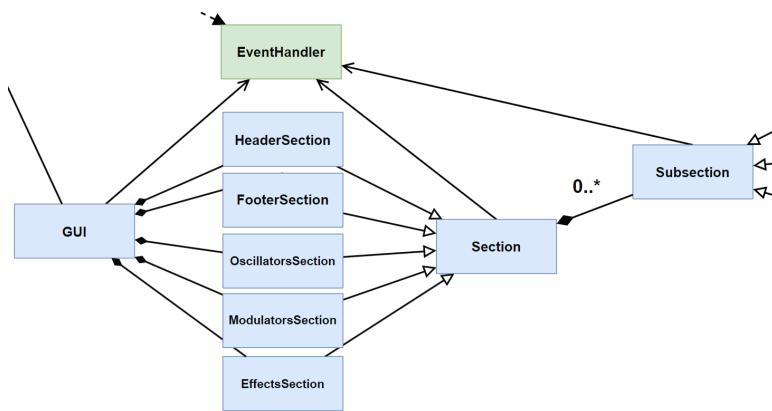


Figura 5.4: Diagrama GUI.

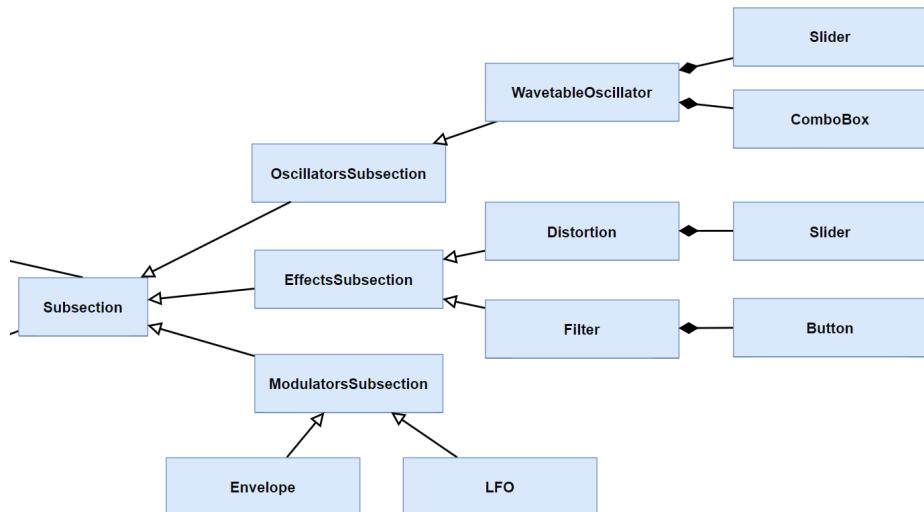
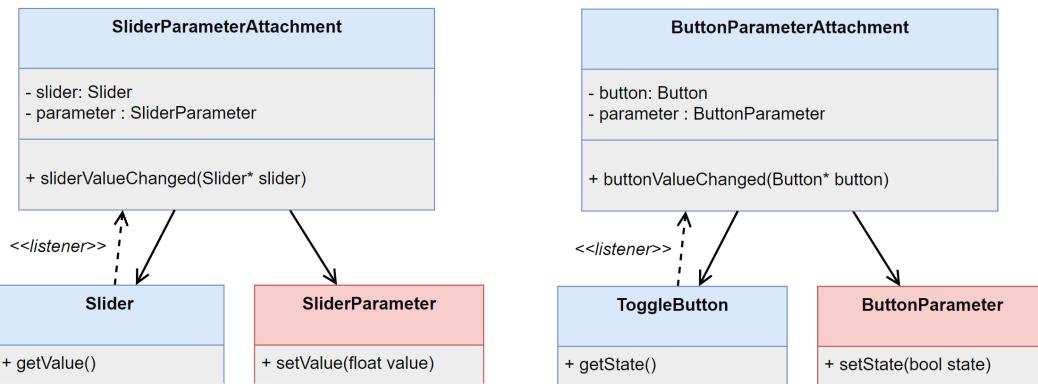


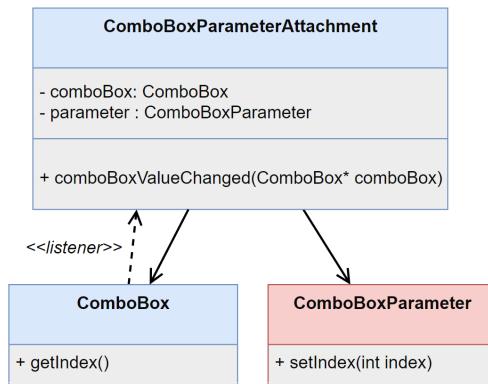
Figura 5.5: Diagrama GUI.

A comunicación entre un slider e o respectivo parámetro faise mediante unha clase intermedia chamada `SliderParameterAttachment` (ver figura 5.6a). Segundo o patrón Observer, cada vez que o usuario modifica o valor do slider, este notifica o cambio a `SliderParameterAttachment`, que actualiza o valor do parámetro co novo valor do slider. Os botóns e caixas de selección seguén o mesmo principio cos seus parámetros e Attachments análogos (ver figuras 5.6b y 5.6c). Deste xeito, EventHandler queda reservado para operacións más complexas, como eliminar un módulo ou conectar unha modulación a un parámetro.



(a) Diagrama de SliderParameterAttachment.

(b) Diagrama de ButtonParameterAttachment.



(c) Diagrama de ComboBoxParameterAttachment.

Figura 5.6: Diagramas de clases que especifican a conexión entre un parámetro e súa parte visual.

5.1.3 Controlador

Agás a actualización de parámetros que, como xa se comentou, lévase a cabo mediante o patrón Observador, o resto de operacións segue o modelo MVC. O controlador implementa os métodos definidos en EventHandler e a Vista chama a eses métodos. Por exemplo, para engadir un módulo de distorsión o diagrama de secuencia sería o da figura 5.7.

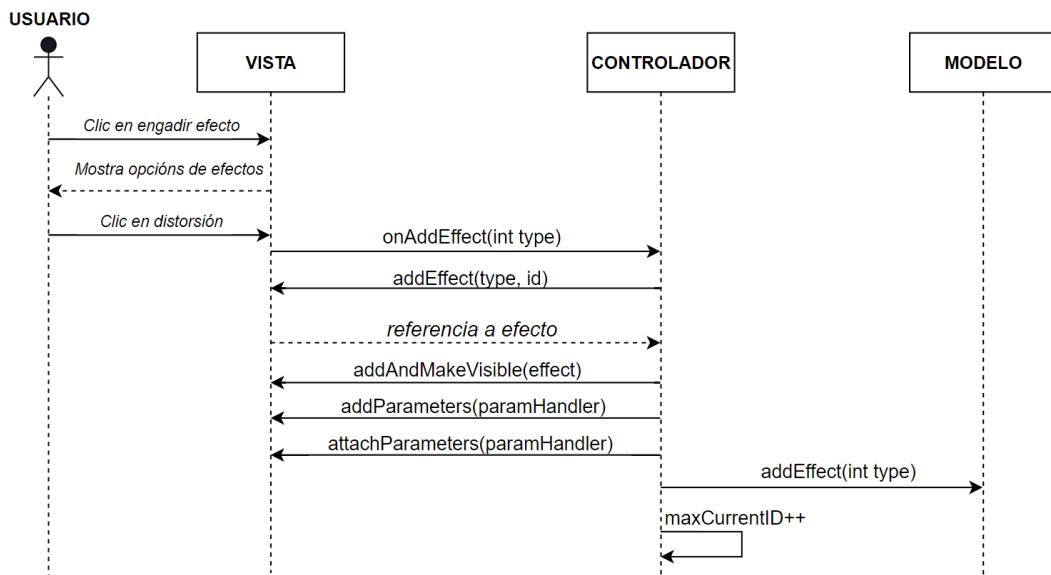


Figura 5.7: Diagrama de secuencia para engadir un módulo de distorsión.

Outras operacións manexadas polo controlador son a creación e conexións das modulacións a os parámetro, a eliminación dun módulo ou o restablecemento do estado da vista a partir da árbore de parámetros.

5.2 Deseño da interface e interacción

Debemos tamén planificar de antemán o deseño que vai ter a interface gráfica e como o usuario vai interactuar con ela. Tendo en conta os requisitos non funcionais, debemos colocar os elementos de tal xeito que sexan facilmente accesibles nuns poucos clics. Tamén debe de ser intuitiva para que os novos usuarios se adapten rapidamente, de xeito que debaixo de cada control haberá unha etiqueta indicando que fai. Por último e non menos importante, debemos facela visualmente agradable, promovendo a inspiración do produtor. Debemos ter en conta no deseño que este sexa adaptable a distintos tamaños de pantalla ou gustos persoais. O deseño dunha interface gráfica coas devanditas características proporcionará aos usuarios un marco de traballo no que a obtención de novos sons sexa unha tarefa agradable, cómoda, dinámica e sinxela.

Para a maquetación inicial, utilizouse a ferramenta Balsamiq [21]. O sintetizador está dividido en dúas partes (ver figura 5.8), a parte dereita corresponde á sección de moduladores, mentres que a esquerda ubica a sección de osciladores e efectos no mesmo espazo. Para intercambiar entre a vista de efectos e de osciladores utilizanse os botóns da parte superior. A maioría de módulos presentan gráficos que varían en función do valor dos parámetros, o que permite un recoñecemento máis visual de como está configurado o módulo.

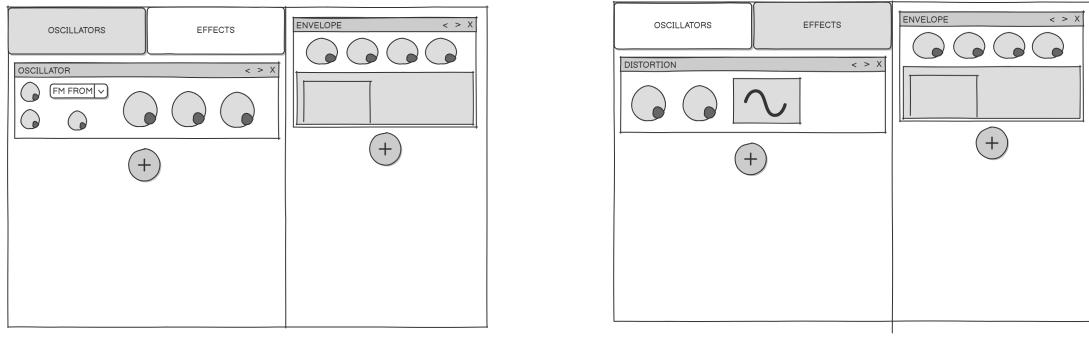


Figura 5.8: Exemplo da interface con algúns módulos engadidos.

5.2.1 Engadir un módulo

Para engadir módulos (osciladores, moduladores ou efectos), o usuario deberá facer clic no botón “+” dentro da respectiva sección. Aparecerá un despregable que mostrará as distintas opcións. Ao facer clic nunha delas, engadirase o módulo dentro da correspondente sección. De sobrepassar os límites verticais da sección, poderase facer *scroll* vertical. Por exemplo, na figura 5.9 móstrase o proceso para engadir un Sampler á sección de osciladores.

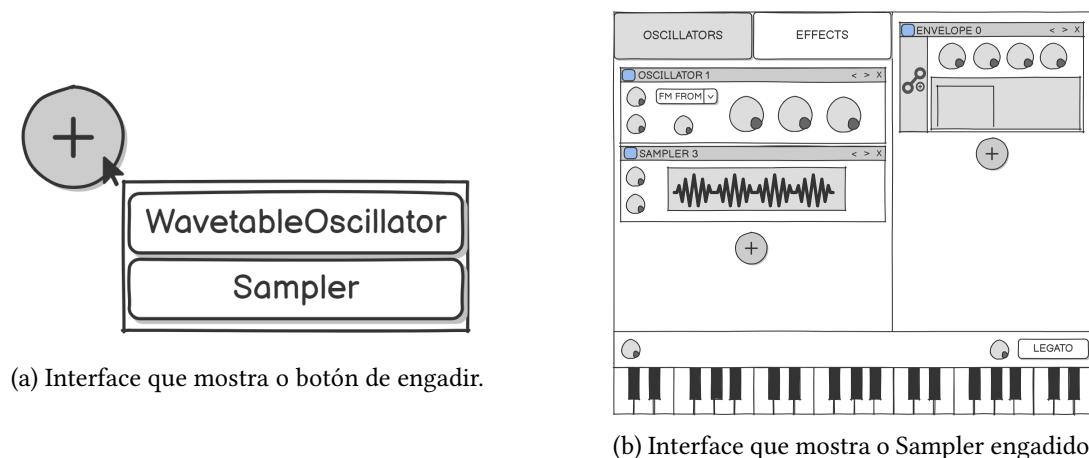


Figura 5.9: Exemplo da interface para engadir un Sampler.

5.2.2 Mover un módulo

Pódese modificar a orde de todos os módulos, cara arriba ou cara abaxo, ao facer clic nas iconas “<” e “>” (ver figura 5.10). Mover un módulo de sitio non afecta á orde de procesamento, a menos que sexa un efecto. Isto permite unha mellor organización e, no caso dos efectos, conseguir distintos resultados ao aplicalos en distinta orde.

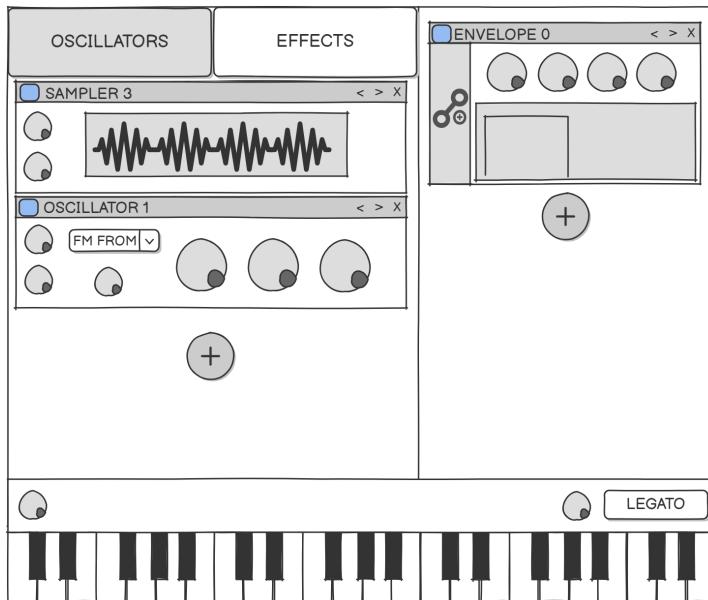


Figura 5.10: Exemplo da interface para mover un módulo.

5.2.3 Eliminar un módulo

Todos os módulos, excepto a envolvente principal (Módulo con ID 0), poderán ser eliminados mediante un simple clic no botón marcado cun “x”. A envolvente principal explicarase máis adiante na sección de envolventes do capítulo de implementación.

5.2.4 Silenciar un módulo

Todos os módulos, excepto a envolvente principal, poden ser silenciados ou desactivados facendo clic no botón iluminado (ver figura 5.11). Ao silenciar ou desilenciar un módulo, a apariencia deste cambiará para reflectir o cambio.

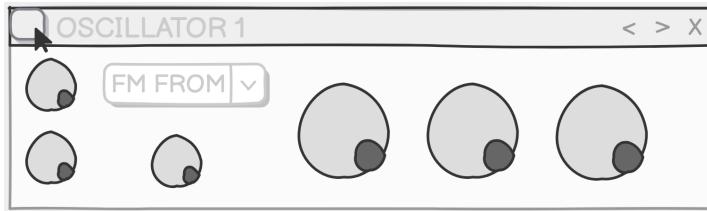


Figura 5.11: Exemplo da interface para silenciar un módulo.

5.2.5 Tocar notas

Para tocar notas empréganse as teclas do teclado. Existen, como máximo, dúas distribucións de teclas: a propia de JUCE e a do DAW no que se está executando o VST. Para usar a distribución de JUCE, primeiro débese facer clic nunha das notas do piano virtual (ver figura 5.12). O mapeo das notas está establecido de tal forma que se asemelle o máis posible a un piano. A fila de teclas "A-L" toca as notas brancas, e a fila de teclas da "Q-P" permite tocar as notas negras. Para usar a distribución establecida polo DAW, débese facer clic en calquera outro lugar da ventá. As notas presionadas aparecerán destacadas en cor amarela.

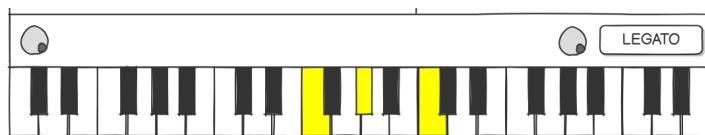


Figura 5.12: Piano virtual.

5.2.6 Conectar unha modulación a un parámetro

Para conectar unha modulación a un parámetro, o usuario debe arrastrar e soltar a modulación sobre o parámetro desexado (ver figura 5.13). Os parámetros que poidan ser modulados por ese modulador aparecerán resaltados cun recadro laranxa. Un modulador non pode modular dúas veces o mesmo parámetro e pode modular como máximo seis. As modulacións só se poden aplicar a parámetros de tipo deslizador, é dicir, non se pode aplicar unha modulación a unha caixa de selección ou a un botón.

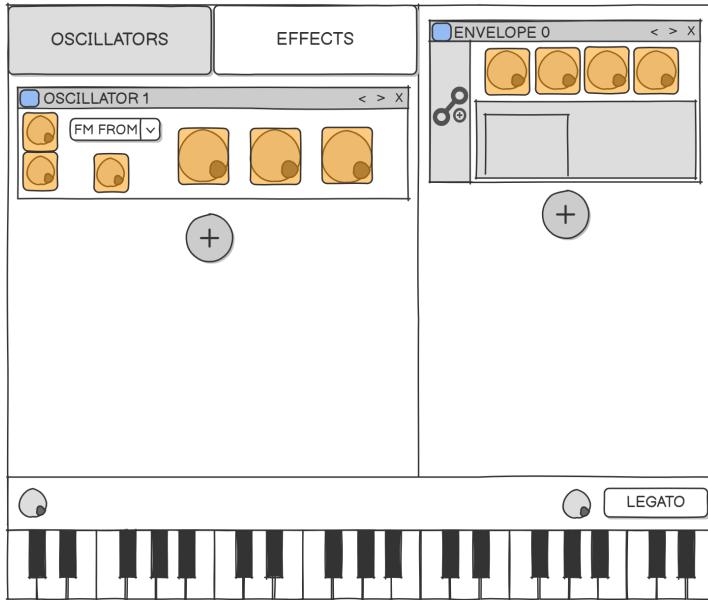


Figura 5.13: Exemplo da interface para conectar unha modulación a un parámetro.

5.2.7 Establecer valor de modulación e eliminación

Unha vez conectado o modulador co parámetro aparecerá unha pequena icona azul na zona de modulacións (ver figura 5.14a). Esta icona compórtase como un deslizador, e establece a cantidad ou o rango de modulación. Pasando o ratón por encima mostra o parámetro que está modulando e mais a cantidad de modulación que se lle está aplicando. Facendo clic derecho sobre él, aparece unha ventá que permite establecer mediante teclado un novo valor ou eliminar a modulación (ver figura 5.14b).

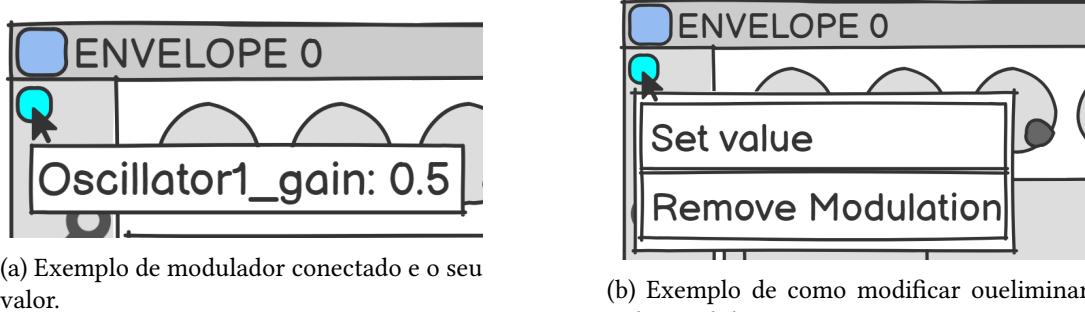


Figura 5.14: Exemplos de como modificar ou eliminar unha modulación.

5.3 Parámetros

Os parámetros son os distintos controis do sintetizador cos que o usuario pode interactuar como, por exemplo, o control de ganancia dun oscilador. JUCE proporciona unha metodoloxía interna para a creación e xestión dos parámetros que consiste en declarar todos os parámetros que se foran a usar ao instanciar o plugin. Porén, o deseño deste sintetizador non admite o dito modelo, pois o número de parámetros depende de cuntos módulos decide engadir o usuario. É por iso que se debe proporcionar un novo sistema de creación e xestión de parámetros que manexe o estado dos parámetros, permitindo engadir parámetros en tempo real. Este novo sistema tamén debe permitir a reconstrucción da interface gráfica, pois esta é destruída unha vez se pecha a ventá do plugin.

O inconveniente de crear e eliminar parámetros baixo demanda é que ningún host vai ser capaz de recoñecelos, polo que a automatización de parámetros por parte do host non vai ser posible. Esta carencia pode ser resolta mediante as *macros*, un tipo especial de modulador consistentes nun único desprazador. Podemos seguir a metodoloxía de creación de parámetros de JUCE para conectar o dito desprazador cun parámetro de JUCE, de xeito que a macro poida ser recoñecida polo DAW e automatizada. Dado que é un modulador, poderemos internamente conectar a macro con calquera parámetro do sintetizador. Deste xeito, o DAW modula a macro, mentres que esta modula os parámetros.

Capítulo 6

Implementación

Neste capítulo explícanse os detalles técnicos más relevantes sobre a codificación e integración dos distintos módulos no sistema. O enfoque principal é a implementación dos procesadores de son e non tanto na parte visual.

6.1 Introducción ao procesamento de son

Os plugins e o DAW no que se aloxa están en constante comunicación. O DAW envía información aos plugins, como a posición actual do *playhead* ou eventos *musical instrument digital interface (MIDI)*, como a pulsación dunha nota. O plugin tamén se comunica co DAW, enviándolle a saída de son. Esta comunicación realiza en bloques: hai un bloque dedicado aos eventos *MIDI* (o *MIDI buffer*) e outro para as mostras de son (o *audio buffer*).

Por regra xeral, o usuario pode elixir o tamaño máximo que pode ter o bloque de son. A maior tamaño de bloque, menor será o consumo de *central processing unit (CPU)*, pero maior será a *latencia*. As opcións de tamaño de bloque adoitan ser potencias de dous: 256, 512, 1024, 2048 etc. (ver figura 6.1). Porén, o *DAW* establecerá un tamaño do *buffer* diferente entre iteracións dentro do máximo establecido. É moi importante ter este último punto en conta á hora de programar os algoritmos de procesamento para evitar artefactos e accesos a memoria incorrectos.

```
1 juce::ProcessBlock(juce::AudioBuffer<float> & buffer);
```

Listaxe 6.1: Referencia en JUCE ao buffer de son.

O framework JUCE [12] proporciona unha referencia a este buffer, e cada un dos procesadores do sintetizador implementa o método da listaxe 6.1 que realiza o procesamiento correspondente. O procesamento dos módulos é secuencial e ordenado: primeiro procédese cos moduladores, logo cos osciladores e, finalmente, cos efectos.

É fundamental que o código dentro deste método sexa o máis rápido posible, pois é onde se

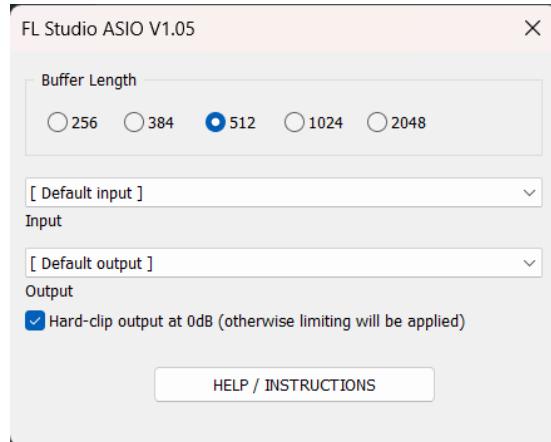


Figura 6.1: Opcións que proporciona FL Studio [2] para o tamaño do *buffer* de son.

realiza todo o procesamento dos sinais. A principal técnica de optimización consiste en sacar todos os cálculos repetitivos e constantes fóra do bucle entre chamadas, para que se realicen unha única vez durante a inicialización. É dicir, en facer a maior cantidade de precalculacións posibles.

```
1 juce::PrepareToPlay(juce::ProcessSpec spec);
```

Listaxe 6.2: Especificación do son.

Outro método moi importante é o da listaxe 6.2, dispoñible en todos os procesadores de son. Este método chámase ao engadir un módulo e permite o paso de tres parámetros importantes para procesar o son correctamente: a frecuencia de mostraxe (ou *sample rate*), o tamaño máximo de bloque e o número de canles dispoñibles.

6.2 Sistema de parámetros

Como se comentou no capítulo 5, é necesario un novo sistema de creación e xestión de parámetros. As instancias dos parámetros gárdanse na clase `ParameterHandler`, clase que incorpora métodos para acceder a eles, eliminarlos e actualizalos. Os parámetros gárdanse de dúas formas. Por un lado, a instancia do parámetro gárdase nun *hashMap*, onde a chave é o identificador do parámetro e o valor contén a instancia. Por outro, o estado destes parámetros gárdase nunha árbore de datos, cuxa implementación é proporcionada por JUCE mediante a clase `ValueTree`. Esta árbore pode considerarse como a base de datos do sintetizador, de acceso rápido tanto para lectura como escritura.

Para diferenciar un parámetro doutro débese seguir unha convención no formato do seu ID. Así, un ID xenérico segue o formato “`ProcessorSubtype_ProcessorID_ParameterName`”.

Por exemplo, para identificar o parámetro de ganancia entre dous osciladores quedaría: “WavetableOscillator_1_volume” e “WavetableOscillator_2_volume”. Unha representación do estado desa árbore pode verse na figura 6.2.

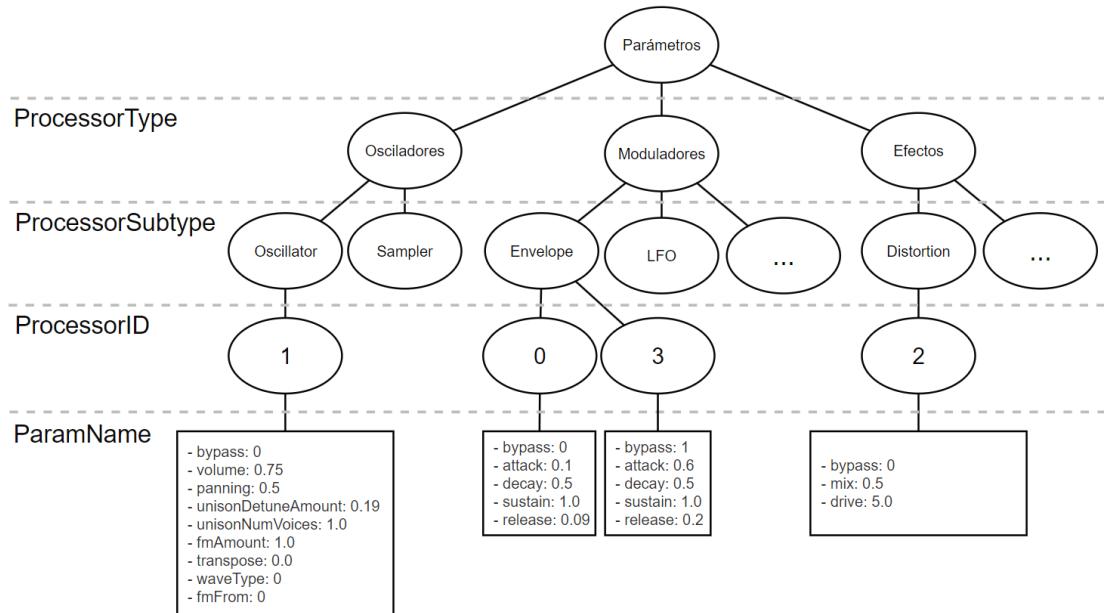


Figura 6.2: Exemplo dunha árbore de parámetros.

Para a reconstrucción do estado do plugin, cando se cerra a ventá, JUCE emprega o método da listaxe 6.3 de `AudioProcessor`. Este método debemos modificalo para obter o [extensible markup language \(XML\)](#) a partir do nodo raíz da árbore de parámetros para despois gardalo en formato binario nun bloque de memoria interno de JUCE.

```

1 void getStateInformation(juce::MemoryBlock& destData) {
2     auto xml = parameterHandler.getRootTree().createXml();
3     copyXmlToBinary(*xml, destData);
4 }
```

Listaxe 6.3: Método para obter o estado do plugin.

A árbore recupérase mediante o método da listaxe 6.4, onde se volve a converter o bloque de memoria de binario a [XML](#), pasándolle á árbore.

```

1 void OcnetAudioProcessor::setStateInformation (const void*
data, int sizeInBytes) {
2     std::unique_ptr<juce::XmlElement>
xmlState(getXmlFromBinary(data, sizeInBytes));
3
4     if (xmlState.get() != nullptr)
  
```

```

5     if
6         (xmlState->hasTagName(parameterHandler.getRootTree().getType()))
7             parameterHandler.getRootTree().fromXml(*xmlState);
}

```

Listaxe 6.4: Método para establecer o estado do plugin.

Finalmente, dende o controlador reconstrúese a interface gráfica ao ler a árbore mediante o método da listaxe 6.5.

```

1 void initialiseGUIFromTree(juce::ValueTree tree);

```

Listaxe 6.5: Método para inicializar a interface gráfica a partir da árbore.

A continuación, describese en detalle a implementación de cada un dos módulos.

6.3 Osciladores

Un oscilador é un procesador de son encargado de xerar o son base, polo que a calidade do son dun sintetizador depende en gran parte da implementación dos seus osciladores. É nesta parte onde debemos ter especial coidado co aliasing.

6.3.1 Oscilador de táboas de onda

Un oscilador de táboas de onda (*wavetable oscillator*) é un tipo de oscilador que precalcula, na súa inicialización, un ciclo da onda que vai reproducir. A ese ciclo denominánselle *wavetable*. No momento de procesar un bloque de son, recorre o devandito ciclo mediante un incremento de fase distinto para cada frecuencia.

O oscilador ten tres opcións de onda (ver figura 6.3): serra, cadrada e sinusoidal.

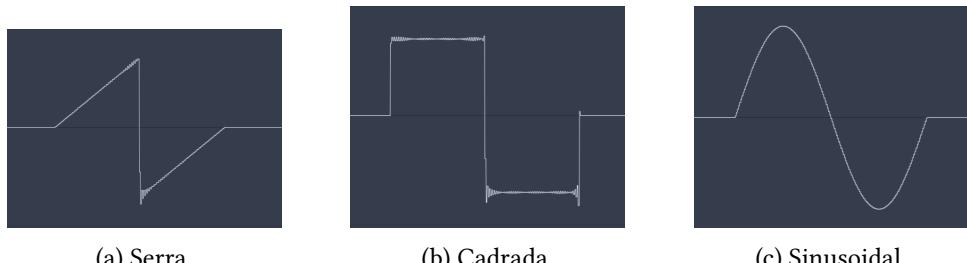


Figura 6.3: Ciclos das ondas que produce o sintetizador.

No capítulo 2 explícanse a transformada de Fourier e como se pode descompoñer un sinal periódico no tempo mediante as series de Fourier. Neste apartado detállase o emprego desta serie para obter as formas de onda mencionadas.

O proceso xeral para crear unha onda é o seguinte. As ondas inicialízanse no constructor da clase. Créanse a partir do seu espectro ao engadir os harmónicos propios de cada unha e despois aplícase a [fast Fourier transform \(FFT\)](#) para obter as mostras dun ciclo da mesma. Para aplicar a FFT é necesario traballar coa forma complexa da transformada de Fourier, polo que cada harmónico está composto por unha parte imaxinaria e unha real. A parte imaxinaria de cada harmónico débese igualar a 0 para que todos os harmónicos teñan a mesma fase e non alteren a forma de onda final ao aplicar a transformada inversa de Fourier. En [\(6.1\)](#) pódese ver a serie de Fourier en forma complexa.

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{in\omega_0 t} \quad (6.1)$$

sendo $c_n = a_n + i b_n$ o coeficiente complexo, onde a_n é a súa parte real, b_n é a súa parte imaxinaria e o termo $\omega_0 = 2\pi f_0$ representa a frecuencia angular en rad/s. A fase Φ_n e a amplitude A_n de cada harmónico (n) calcúlanse a partir de $c_n = A_n e^{i\Phi}$

- Amplitude: $A_n = \sqrt{a_n^2 + b_n^2}$
- Fase: $\Phi_n = \arctan\left(\frac{b_n}{a_n}\right)$

Por exemplo, para xerar unha onda cadrada (ver figura [6.4](#) e listaxe [6.6](#)), debemos conseguir unha distribución harmónica tal que a parte real dos harmónicos pares sexa igual a 0 e nos impares a amplitude decreza segundo o factor $\frac{1}{x}$, onde x é o número de harmónico. Dado que traballamos con números complexos, debemos ter en conta a propiedade de simetría conxugada $c_{-n} = \overline{c_n}$, segundo a cal, ao crear o espectro de cada onda, debemos engadir harmónicos tanto nas frecuencias negativas como nas positivas. Isto asegura que o sinal reconstruído sexa real no dominio do tempo.

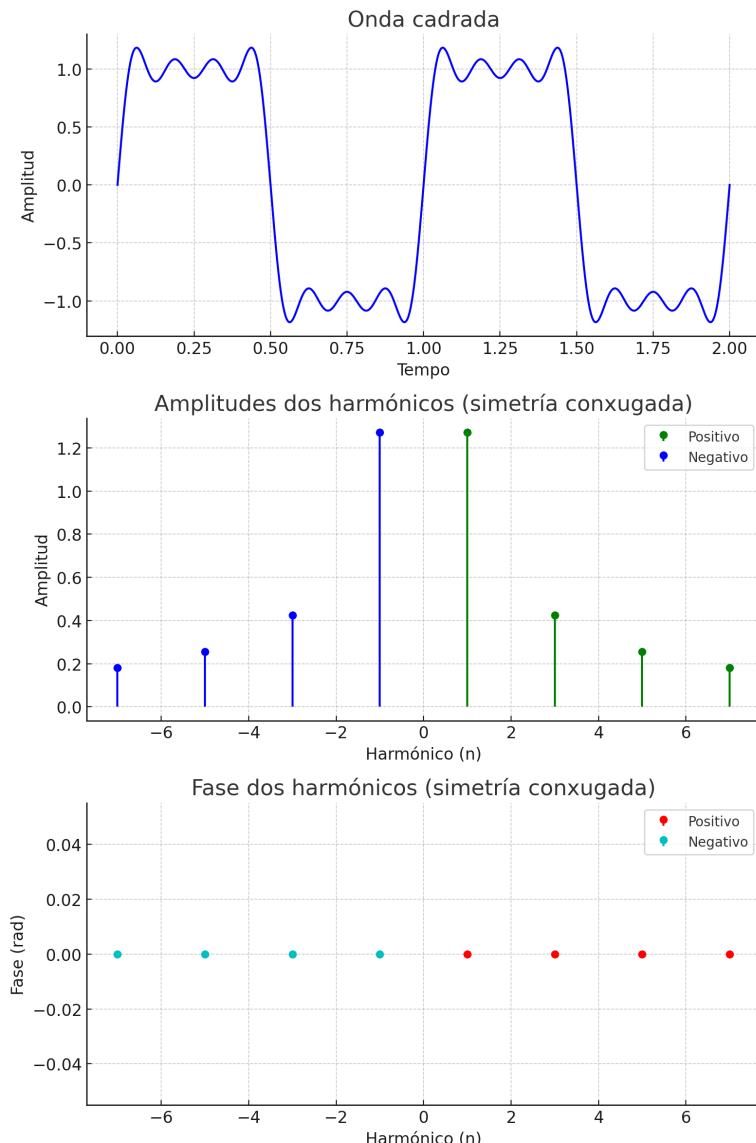


Figura 6.4: Simetría conxugada para unha onda cadrada.

```

1 // Parte imaxinaria
2 for (int idx = 0; idx < tableSize; idx++) {
3     freqWaveIm[idx] = 0.0;
4 }
5
6 // Parte real
7 void square(std::unique_ptr<double[]>& freqWaveRe, int
8 tableSize) {
9     freqWaveRe[0] = freqWaveRe[tableSize >> 1] = 0.0;
10    for (int idx = 1; idx < (tableSize >> 1); idx++) {
11        freqWaveRe[idx] = freqWaveRe[tableSize - idx];
12    }
13 }
```

```

10     if (idx % 2 != 0) { // Só índices impares
11         // Espectro da onda cadrada
12         freqWaveRe[idx] = 1.0 / idx;
13         // Espello
14         freqWaveRe[tableSize - idx] = -freqWaveRe[idx];
15     }
16     else { // Índices pares son 0
17         freqWaveRe[idx] = 0.0;
18         freqWaveRe[tableSize - idx] = 0.0;
19     }
20 }
21 }
```

Listaxe 6.6: Cálculo do espectro da onda cadrada.

De empregar unha única *wavetable* no oscilador, ao reproducila a frecuencias altas podería producirse *aliasing* ou solapamento, pois a forma de onda incorporaría harmónicos en frecuencias más altas que a de Nyquist. O que se fai é utilizar varias *waveTables*, cada unha optimizada para un rango de frecuencias, cunha cantidade reducida de harmónicos, limitando os más altos que poideran producir aliasing.

A cantidade de harmónicos que contén a primeira *wavetable* é igual á metade da cantidade de mostras establecida. A maior cantidade de mostras, mellor calidade, pero maior espazo ocupan en memoria. Como o espazo non adoita ser un problema, estableceuse un tamaño fixo de 4096 mostras por ciclo.

A primeira táboa que se xera ten todos os harmónicos da onda orixinal (xerada a 1 Hz) e utilizase para as frecuencias baixas. A medida que a frecuencia aumenta, elimináñanse progresivamente os harmónicos más altos. O número de táboas de onda resultante segue aproximadamente a seguinte fórmula, onde f é o factor de redución e N o número de mostras.

$$\text{Número de táboas} \approx \left\lfloor \log_f \left(\frac{N}{2} \right) \right\rfloor \quad (6.2)$$

Un factor de redución moi baixo pode chegar a xerar unha táboa por cada nota, e un factor de redución de 2 (máximo) xera unha táboa por cada oitava. Máis táboas proporcionan maior precisión ao eliminar o aliasing, mais implican unha maior carga á hora de buscar a táboa óptima para a frecuencia que se quere reproducir (cada vez que se pulsa unha nota). Polo tanto, optouse por un factor de redución de 1.5, é dicir, cada 6 notas.

A figura 6.5 compara o `highend` dunha onda de serra na transición entre táboas para dous factores de redución distintos. Co factor 2 a transición entre táboas é moi brusca, producíndose gran cantidade de aliasing na última nota dunha das táboas e un gran corte na primeira nota da seguinte táboa. Co factor 1.5 o cambio é moito más suave, imperceptible auditivamente.

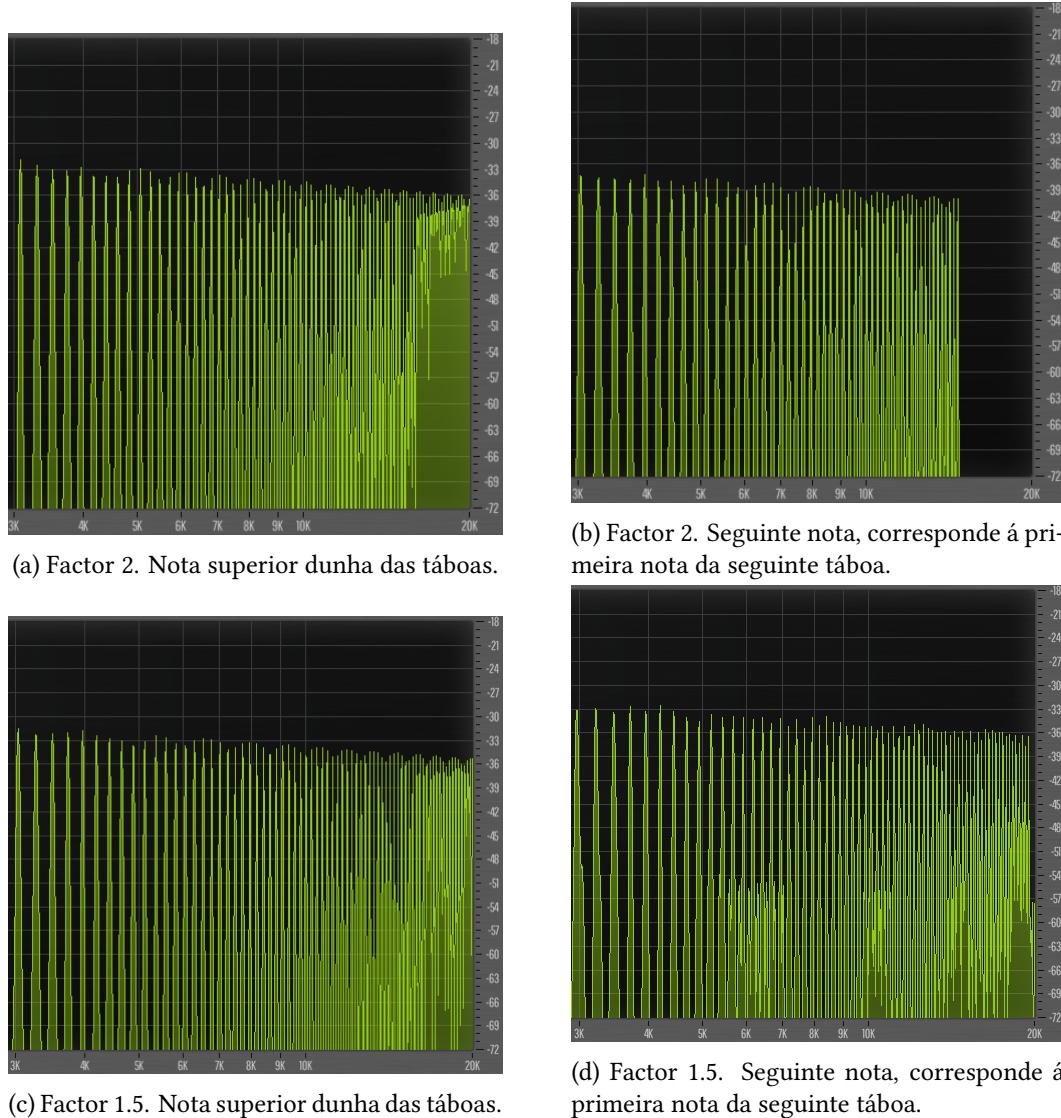


Figura 6.5: Comparación entre dous factores de redución.

As táboas de ondas xeradas gárdanse nun struct con dous campos:

- `double topFreq`: Garda a frecuencia máxima que pode representar a táboa, igual á frecuencia do seu último harmónico.
- `AudioSampleBuffer wavetable`: O vector de mostras desa táboa.

Cada vez que se pulsa unha nota, a partir da súa frecuencia obtense o incremento de fase necesario para reproducila a esa frecuencia (ver listaxe 6.7).

```

1 auto tableSizeOverSampleRate = (float)tableSize / sampleRate;
2 tableDelta = frequency * tableSizeOverSampleRate;
```

Listaxe 6.7: Cálculo do incremento de fase para reproducir unha nota a unha frecuencia determinada.

Tamén se obtén unha referencia á primeira táboa que poida representar esa frecuencia libre de aliasing (ver listaxe 6.8).

```

1 int waveTableIdx = 0;
2 float frequencyOverSamplerate = frequency / sampleRate;
3
4 while ((frequencyOverSamplerate >=
5 (*tables)[waveTableIdx].topFreq) && (waveTableIdx <
6 (numWavetables - 1))) {
7     ++waveTableIdx;
8 }
9 wavetable = &(*tables)[waveTableIdx];

```

Listaxe 6.8: Cálculo da táboa de onda óptima para reproducir unha nota a unha frecuencia determinada.

Finalmente, comentar que a implementación da [FFT](#) está baseada na implementación orixinal de Cooley-Tukey [24] adaptada posteriormente por Ken Steiglitz a partir do traballo descrito en [25]. Esta implementación pódese atopar no apéndice listado A.2

Ganancia

A ganancia é a amplificación ou atenuación que se lle aplica a un sinal antes de que chegue a etapa final de saída (volume).

A ganancia pódese medir en unidades lineais ou logarítmicas. A forma logarítmica é máis común e exprésase en decibelios (dB), que describen a relación entre dous niveis do sinal. É unha medida adaptada á percepción humana dos cambios de volume. Por exemplo, un cambio de 10 dB percíbese aproximadamente como o dobre ou a metade do volume. Para converter un valor de ganancia en escala lineal a escala logarítmica emprégase a ecuación(6.3)

$$G_{\text{dB}} = 20 \log_{10}(G_{\text{lin}}) \quad (6.3)$$

e a fórmula inversa, para convertir a escala logarítmica en lineal

$$G_{\text{lin}} = 10^{\frac{G_{\text{dB}}}{20}} \quad (6.4)$$

En edición de son, ao utilizar a escala logarítmica tómase como referencia o valor 0 como o valor máximo que pode ter o sinal, e $-\infty$ como o valor mínimo.

En canto á implementación (ver listaxe 6.9), o control de ganancia toma valores entre 0 e 1, pero aplícase un factor de *skew* logarítmico que proporciona maior control nos valores más baixos.

```

1 leftChannelBuffer[sample] += currentSample * gainLeft;
2 rightChannelBuffer[sample] += currentSample * gainRight;
```

Listaxe 6.9: Aplicación da ganancia.

Panorama

O panorama refírese á ubicación espacial dun son. Permite simular que un son provén de distintas ubicacións. O efecto prodúcese basicamente baixando a ganancia dunha das canles. Dise que un son está totalmente paneado á dereita cando únicamente se escucha polo lado derecho dos altofalantes ou auriculares e totalmente paneado a esquerda cando soamente se escucha polo esquerdo.

Para que un son se perciba igual de forte sen importar en que lugar do panorama se sitúe, debe seguirse unha lei de paneo, que se refire ao axuste do volume dun sinal cando se move dunha canle a outra. As dúas leis de paneo principais son a circular e a triangular, cada unha cunha forma diferente de axustar o volume.

Para medir o panorama emprégase un instrumento de medida denominado *vectorscopio* (ver figura 6.6), que recibe como entrada o son estéreo e dibuxa nun gráfico bidimensional a dirección do son. Canto máis lonxe do centro chegue o sinal (representado como unha liña branca), maior será a súa amplitude.

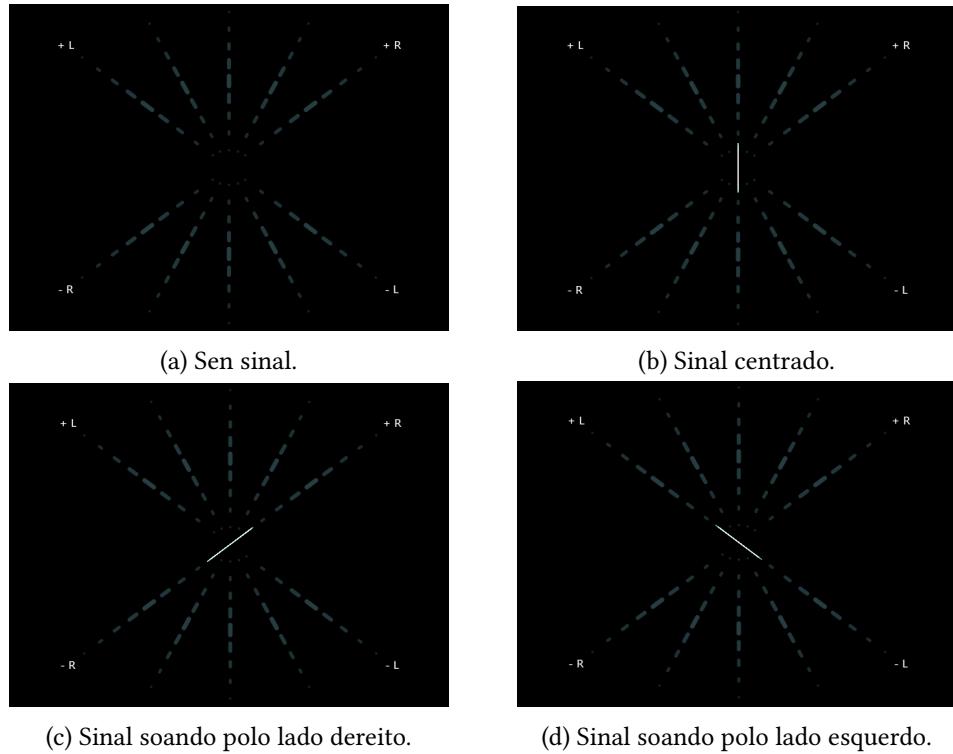


Figura 6.6: Exemplos de comportamento dun vectorscopio.

Na lei de paneo circular (ver figura 6.7a), a redución de volume básease en como percibimos os sons no mundo real. Cando o sinal está no centro do campo estéreo redúcese 3 dB en cada canle (redúcese o volume á metade), polo que ao combinar a enerxía de ámbalas dúas canles non se percibe como máis forte que si estivera soando nun só. Na lei de paneo triangular (ver figura 6.7b), a redución de volume faise de forma lineal, polo que a implementación é máis sinxela. O sinal redúcese en 6 dB por canle cando está no centro.

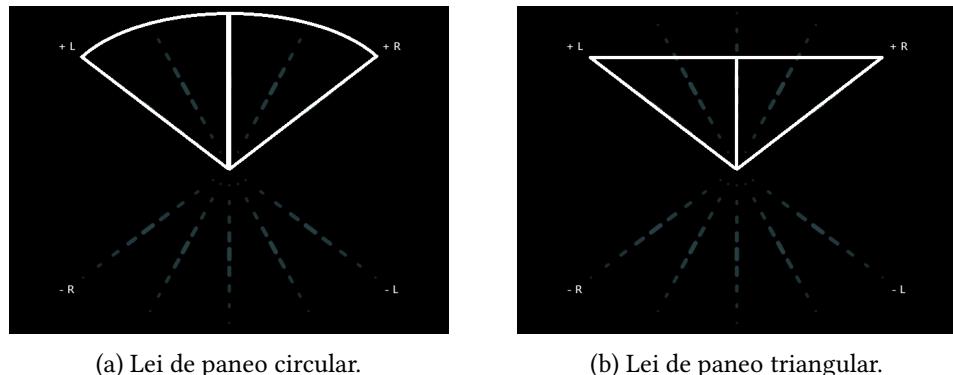


Figura 6.7: Representacións das leis de paneo circular e triangular.

A opción que se implementou é a circular. Aínda que é máis complexa de implementar e tamén computacionalmente más cara, proporciona unha sensación máis natural e cercana ao mundo real. Impleméntase pasando o valor do parámetro de panning, entre 0 e 1, a grados, segundo a listaxe 6.10.

```

1 const auto globalPanAngle = panning * halfPi;
2 const auto globalPanningLeft = std::cos(globalPanAngle);
3 const auto globalPanningRight = std::sin(globalPanAngle);
```

Listaxe 6.10: Cálculo do panning circular.

Síntese FM

A síntese FM é unha técnica que permite xerar sons complexos, metálicos ou mesmo replicar instrumentos reais, a partir doutros simples. Emprega dous osciladores, un deles xera o sinal da portadora e o outro o modulador. O sinal da portadora modula a súa frecuencia en función da amplitude do modulador. Os mellores resultados obtéñense cando o sinal modulador é unha onda sinusoidal (ver figura 6.8).

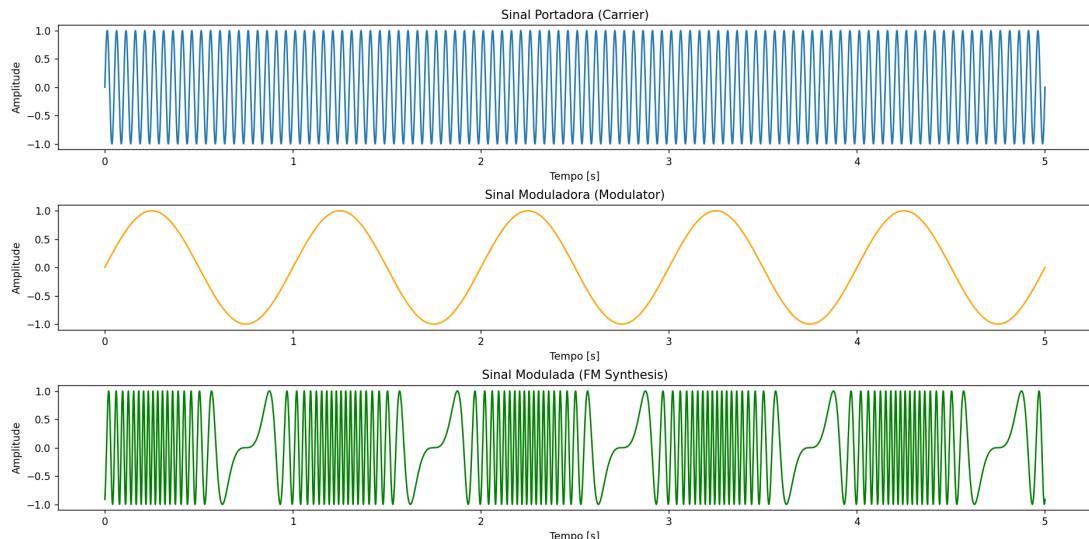


Figura 6.8: Síntese FM.

Un parámetro fundamental que o usuario pode e debe controlar é o índice de modulación, que establece unha relación entre a amplitude da onda moduladora e a frecuencia da onda portadora. A síntese FM entre dous sinais adoita interpretarse como unha modulación de fase segundo (6.5).

$$y(t) = A_c \sin(\omega_c t + I \sin(\omega_m t)) \quad (6.5)$$

onde I é o índice de modulación e ω_c e ω_m son as frecuencias angulares da onda portadora e moduladora, respectivamente.

Dado que o algoritmo para recorrer a wavetable non acepta incrementos de fase negativos, (6.5) modifícase para realizar a modulación directamente sobre a frecuencia da portadora tendo en conta que $\omega=2\pi f$

$$y(t) = A_c \sin(2\pi(f_c + I \sin(\omega_m t))t) \quad (6.6)$$

O inconveniente deste método é que debemos tratar a saída da moduladora en unidade de cents, pois a diferencia en hercios entre dúas notas na primeira oitava é menor que a diferencia entre as mesmas notas noutra oitava do piano. Un cent é a menor unidade empregada para medir intervalos musicais. É unha unidade logarítmica que equivale á centesima parte dun semitono (distancia entre dúas notas). É dicir, un semitono son 100 cents e unha oitava son 1200 cents.

Para obter a frecuencia relativa a unha nota máis certa cantidade de cents, emprégase

$$\text{frecuencia relativa} = f_{\text{actual}} \cdot 2^{\frac{\text{cents}}{1200}} \quad (6.7)$$

Para aplicar modulación FM, o usuario debe agregar dous osciladores, elixir aquel que quere que produza o sinal de portadora e seleccionar no despregable o oscilador que actuará como modulador. O normal é baixar a ganancia do modulador a 0, para que únicamente se escute a modulación FM.

Unísono

O unísono é unha técnina na que varias voces lixeiramente desafinadas soan ao mesmo tempo para producir un son más cheo e rico. A desafinación adoita ser menor a un semitono sobre a nota base. A desafinación é acompañada cun lixeiro e sutil paneo cara ós lados. Mediante o unísono créanse sons icónicos como o *Reese Bass* ou o *Supersaw*.

Debemos diferenciar as voces do sintetizador das voces do unísono. Establecer tres voces de unísono é igual a engadir tres osciladores con unísono 1, lixeiramente desfasados e paneados. É dicir, as voces procésanse secuencialmente sumándose ao buffer de son.

O usuario ten control sobre cantas voces de unísono debe reproducir o oscilador e sobre a

cantidadade de desafinación (*detune*) entre elas. A desafinación faiuse mediante cents polo mesmo motivo descrito no apartado anterior. O número de voces máximas é de 8 por oscilador e a cantidadade máxima de desafinación é de 2 semitonos (200 cents).

Para mellorar a eficiencia, a cantidadade máxima de desafinación e separación para cada voz precalcúlase no constructor da clase. Tamén se inicializan os desfasos para cada nota mediante unha función de aleatorización. O desfase é necesario para evitar que nos primeiros segundos logo de pulsar unha nota se produza un efecto láser, provocado pola cancelación de fases entre as voces.

Transposición

A transposición consiste en mover as notas arriba ou abaixo na escala. A transposición mídese en semitonos, por exemplo, se transportamos un oscilador unha cantidadade de 1 semitono, ao pulsar unha nota DO, soará unha nota DO#, e se transportamos 12 semitonos soará DO unha oitava más arriba.

Implementar a transposición é tan sixelo como sumarlle ao número de nota pulsada a cantidadade de *transpose*. O método da listaxe 6.11 devolve a frecuencia, expresada en hercios, da nota pasada por parámetro. A cantidadade máxima de transposición vai de -48 a 48 notas. É unha operación sinxela que permite combinar varios osciladores para producir acordes ou outros intervalos, obtendo sons más ricos e complexos harmónicamente.

```
1     getMidiNoteInHertz(midiNoteNumber + transpose);
```

Listaxe 6.11: Transposición.

Resultado final do oscilador

Con todo isto, a subsección do oscilador vese como na figura 6.9.

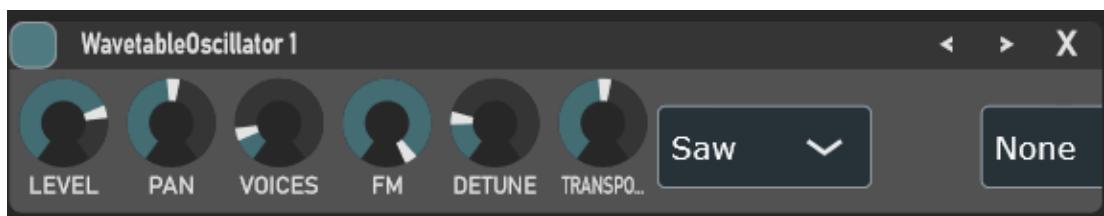


Figura 6.9: Módulo do oscilador.

6.3.2 Sampler

Un *ampler* é un xerador que permite reproducir un arquivo de son previamente seleccionado. Estes arquivos de son deben estar localizados nunha ruta específica para que poidan

ser cargados correctamente dentro do sampler. Moitos sintetizadores inclúen, de serie, un conxunto de mostras de son predefinidas, listas para o seu uso. Desta forma, o usuario dispón desde o principio dun pequeno repertorio de opcións. Estas mostras adoitan incluír ruído branco, atmósferas sonoras e mostras de percusión como tambores ou pratos.

Para implementar isto, optouse por crear un instalador para o sintetizador que configura automaticamente un directorio nunha ruta coñecida. Este directorio incluirá exemplos de son que poderán ser cargados polo sampler. Ademais, se o usuario o desexa, poderá engadir máis arquivos a este directorio para ampliar o repertorio de sons dispoñibles.

O instalador foi creado empregando a ferramenta gratuita Inno Setup [26], exclusiva para Windows, que xera instaladores no formato .exe. Para crear o instalador é necesario escribir un script que especifique os arquivos que se van incluír, a súa localización no sistema e información adicional como a creación de subdirectorios ou a definición do nome da aplicación. O instalador (ver figura 6.10) dará a opción de elixir unha ruta distinta para gardar o VST, mais obligatoriamente establece a ruta para o directorio de arquivos na carpeta de Documentos do equipo.

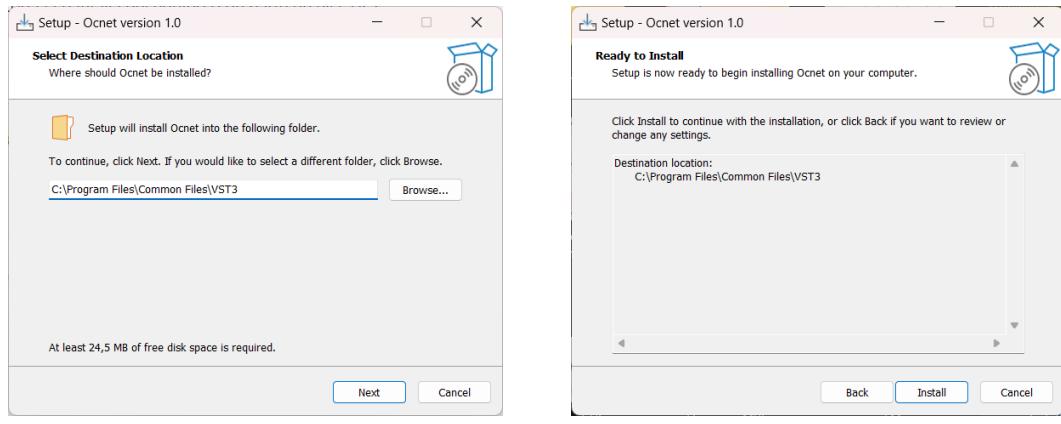


Figura 6.10: Pasos para instalar o VST.

Resultado final do sampler



Figura 6.11: Módulo do sampler.

6.4 Moduladores

Un modulador é o procesador de son encargado de xerar as modulacións dos parámetros do sintetizador. Os moduladores son os primeiros en procesarse, pois as modulacións son gardadas nun buffer que será lido polos demás procesadores. O tamaño dese buffer é o mesmo que o do buffer de son.

Cada modulación modula un só parámetro e ten asociado un ID co seguinte formato: “modulatorType_modulatorID_modulationAmount_parameterID”. Por exemplo, se un modulador con ID 1 de tipo Envelope está modulando un parámetro con ID “Distortion_2_drive”, modulationID quedaría: “Envelope_1_modulationAmount_Distortion_2_drive”.

Mediante o método `updateParameters()` os procesadores obteñen para cada parámetro, unha única vez antes de procesar cada bloque, tanto o novo valor como o buffer de modulacións. Posteriormente, no `processBlock()` por cada mostra, engádeselle ao valor actual do parámetro o valor gardado no índice correspondente do buffer de modulacións para obter o valor final modulado para ese parámetro. Dado que o valor de modulación oscila entre -1 e 1 incluídos, pode ocorrer que o valor final da suma sexa un valor por debaixo de 0 ou maior a 1, polo que a suma debe de ser normalizada.

6.4.1 Envolvente

A envolvente fai referencia ao comportamento da amplitude dun son no tempo. Mais un modulador de envolvente realmente pode ser aplicado a calquera outro parámetro, como un filtro ou un panorama. Consta de catro parámetros fudamentais: ataque, caída, sostemento e liberación. É por iso que moitas veces a este tipo de modulador se lle denomina `attack, decay, sustain, release (ADSR)`.

Para implementar a envolvente, fixose uso da clase `juce::ADSR` que proporciona JUCE. Para usala, debe chamarse a `getNextSample()` ata encher o buffer de modulación. Os parámetros `ADSR` actualízanse unha vez por bloque mediante `setParameters(Parameters)`.

Este sintetizador inclúe por defecto unha envolvente mestra, que non pode ser eliminada, e que aplica unha modulación de volume despois de procesar os osciladores e antes dos efectos (ver listaxe 6.12).

```

1 ProcessorHandler::processBlock(outputBuffer) {
2     procesarOsciladores(outputBuffer);
3     mainEnvelope->processBlock(outputBuffer);
4     procesarEfectos(outputBuffer);
5 }
```

Listaxe 6.12: Modulador de envolvente mestra.

Resultado final da envolvente



Figura 6.12: Módulo da envolvente.

6.4.2 Oscilador de baixa frecuencia (LFO)

Un **LFO**, ou oscilador de baixa frecuencia, é un tipo especial de oscilador que traballa a unha frecuencia normalmente por debaixo dos 20 Hz. Este modulador utilízase para crear modulacións vibratorias. Por exemplo, se conectamos o **LFO** a un parámetro de volume, podemos obter un efecto de trémolo, no que o volume oscila periodicamente.

Os **LFO** adoitan xerar formas de onda sinxelas, como a onda sinusoidal, cadrada ou triangular. Neste sintetizador, por simplicidade, só se implementa unha onda sinusoidal. A diferenza do oscilador principal, que produce o son a través de táboas de onda, o **LFO** segue unha función sinusoidal a unha frecuencia que é determinada polo usuario.

O usuario pode establecer a velocidade de oscilación do **LFO** en seis modos diferentes:

- Modo libre: Neste modo, o usuario controla a velocidade do **LFO** mediante un *knob* de *ratio*, que permite axustar a frecuencia entre 0 Hz e 20 Hz.
- Modos sincronizados co **beats per minute (BPM)**: Os outros cinco modos sincronizan o **LFO** co **BPM** establecido no **DAW**. Nestes modos, a velocidade do **LFO** segue figuras rítmicas musicais:
 - 2º Modo: Redonda
 - 3º Modo: Branca
 - 4º Modo: Negra
 - 5º Modo: Corchea
 - 6º Modo: Semicorchea

Estes modos permiten que o **LFO** oscile a unha frecuencia que está sincronizada co tempo do **DAW**, producindo modulacións ao ritmo deste.

Resultado final do LFO



Figura 6.13: Módulo do LFO.

6.4.3 Aleatorizador

Como indica o seu nome, é un modulador que establece o valor mediante unha función de aleatorización. Do mesmo xeito que ocorre co [LFO](#), o usuario pode establecer a frecuencia de oscilación mediante un *knob*. Dáse a escoller entre dúas funcións xeradoras de ruído.

Soft Noise

Este tipo de ruído xera valores pseudoaleatorios que, ao mesmo tempo, manteñen unha coherencia entre si, producindo variacións suaves e continuas (ver figura 6.14). Esta variación gradual, en vez de ser brusca, permite simular sons más naturais. O proceso impleméntase en tres pasos para cada mostra. Primeiro, xérase un valor aleatorio entre -1 e 1 . En segundo lugar, obtense un valor intermedio entre o valor anterior e o actual mediante interpolación cosenoidal, utilizando a fase como factor de interpolación. Finalmente, increménțase a fase para calcular o próximo valor.

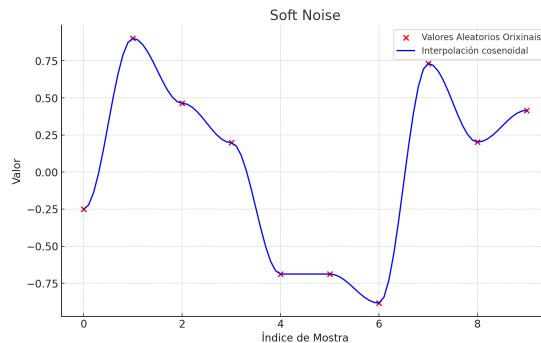


Figura 6.14: Función soft noise.

Sample & Hold

Ao contrario que o *Perlin Noise*, este ruído caracterízase por saltos bruscos e discontinuos do sinal modulador. Se o anterior ruído permitía simular variacións naturais, este simula sons artificiais e dixitais (ver figura 6.15). Impleméntase de forma moi sinxela facendo que se obteña un novo valor aleatorio entre -1 e 1 cada n mostras (Intervalo *Hold*).

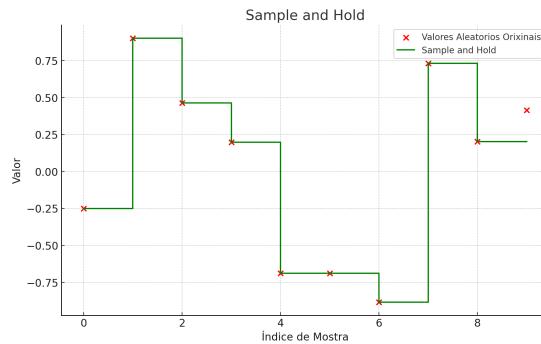


Figura 6.15: Función Sample and Hold.

Resultado final do aleatorizador

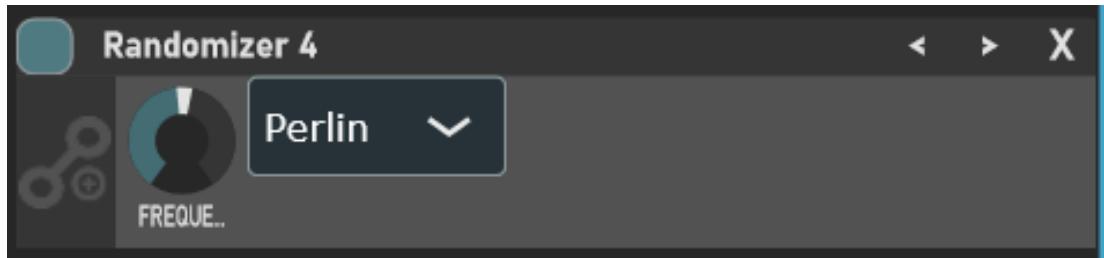


Figura 6.16: Módulo do aleatorizador.

6.4.4 Macro

A macro é un tipo especial de modulador moi sinxelo pero de grande utilidade. Consta dun único *knob*, que permite ao usuario modular varios parámetros simultaneamente. É dicir, en vez de seguir unha función ou algoritmo como os demais moduladores, é o usuario quen determina o valor de modulación en cada momento. Ademais, como foi explicado no capítulo de deseño, neste sintetizador en particular ten unha función especial, pois é a única forma de que o *DAW* poida acceder aos parámetros do sintetizador, permitindo a automatización de parámetros dende o *DAW*. Para que o parámetro da macro sexa recoñecido polos *DAW*, este foi implementado seguindo a metodoloxía de JUCE, pensada para declarar na inicialización do plugin unha cantidade fixa de parámetros. É por isto que o número de macros que se

poden engadir está limitado actualmente a 12. A forma de JUCE de crear parámetros é mediante a clase *AudioProcessorValueTreeState*, que debe instanciarse xunto con *PluginProcessor*, pasándolle como parámetro unha instancia de *ParameterLayout*, que contén as instances de parámetros das macros.

Resultado final da macro



Figura 6.17: Módulo da macro.

6.5 Efectos

Os efectos son os últimos na cadea de procesamento. Engaden calidades novas sobre o son base antes de emitilo polos altofalantes.

6.5.1 Distorsión

O efecto de distorsión consiste en modificar a forma de onda do sinal orixinal, de modo que se xeren novas frecuencias, normalmente harmónicas, que non estaban presentes. Isto produce un son máis agresivo, saturado ou “sucio”, o que incrementa tamén a potencia media da sinal. O proceso de distorsión pode realizarse de varias maneiras, mais xeralmente implica modificar a amplitude do sinal de maneira non lineal. Existen varios tipos de distorsión, como o *wave shaping*, *overdrive*, saturación, *fuzz* e *clipping*. Para este sintetizador, seguindo os requisitos, implementouse a distorsión de tipo *clipping*. Este é o tipo máis básico de distorsión e ocorre cando a amplitude do sinal supera un límite predefinido, coñecido como *threshold*, e as partes do sinal que superan ese umbral son “recortadas”, producíndose harmónicos. Implementáronse dous tipos de *clipping*, dependendo da súa agresividade.

Soft Clipping

O sinal recórtase de forma gradual a medida que se achega ao umbral. Isto conséguese mapeando a amplitude do sinal de entrada a través dunha función como a da figura 6.18. Onde

A é a amplitud do sinal de entrada e d é o parámetro *drive* que permite controlar con que intensidade se aplica a distorsión. Produce sons cálidos.

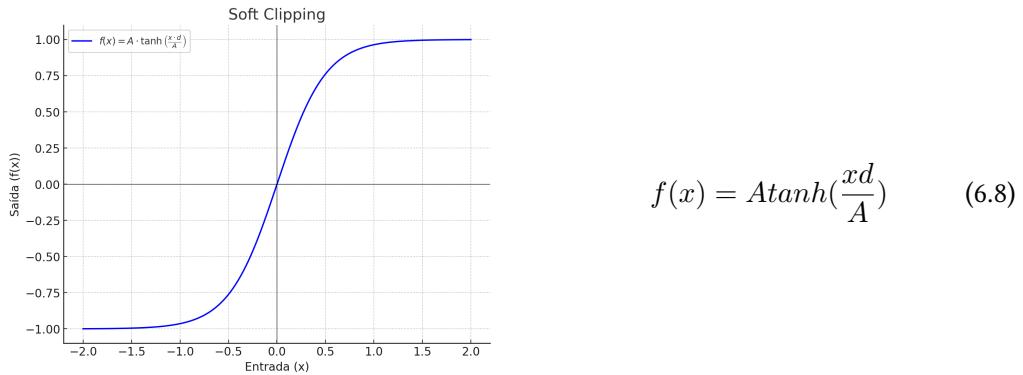


Figura 6.18: Soft Clipping.

Hard Clipping

Como indica o seu nome, é unha versión máis agresiva de *clipping*. En vez de mapear a amplitud do sinal de forma gradual, a medida que se achega ao umbral, esta é recortada de forma abrupta unha vez é superado. Segue a función da figura 6.19 definida en (6.9).

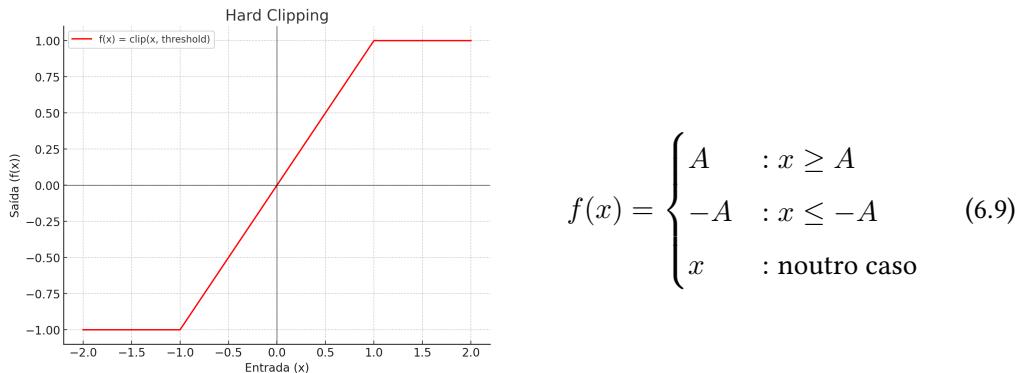


Figura 6.19: Hard Clipping

Como se comentou, no momento en que o sinal se recorta ao sobrepasar o umbral establecido, prodúcense harmónicos. Isto pode converterse nun problema, xa que moitos destes harmónicos aparecen por enriba da frecuencia de Nyquist, o que dá lugar a problemas de aliasing. Para mitigar este problema, aplícase a técnica de sobremostraxe do sinal orixinal ou, en inglés, *oversampling*. O *oversampling* permite procesar un bloque de sinal a unha frecuencia de mostraxe maior que a orixinal, o que facilita a aplicación dun filtro paso baixo que elimina as frecuencias por enriba da frecuencia de Nyquist antes de retornar á frecuencia de mostraxe orixinal.

Esta técnica é custosa en termos de complexidade computacional, polo que o seu uso debe ser controlado. Neste caso, empregouse a clase `Oversampling`, proporcionada por JUCE. Para utilizala é necesario especificar tanto a frecuencia de mostraxe orixinal como o factor de sobremostraxe (que pode ser 2, 4, 8 ou 16 veces a frecuencia orixinal). A maior factor de sobremostraxe, menos artefactos aparecerán no espectro audible, mentres que a complexidade computacional aumenta de forma exponencial. Neste caso, optouse por un factor de 4. Así, de forma simplificada, o método `ProcessBlock` da distorsión vese segundo a listaxe 6.13.

```

1 void processBlock(AudioBuffer<float>& buffer) {
2     // Inicializar bloque de son
3     sonBlock<float> block(buffer), upSampledBlock(buffer);
4
5     // Aumentar a frecuencia de mostraxe N veces
6     upSampledBlock = oversampler.processSamplesUp(block);
7
8     // Procesar bloque sobremostreado
9     processUpsampledBlock(upSampledBlock);
10
11    // Recuperar a frecuencia de mostreo (Aplica un filtro paso
12    // baixo na frecuencia de Nyquist antes)
13    oversampler.processSamplesDown(block);
14 }
```

Listaxe 6.13: Método `ProcessBlock` da distorsión

Resultado final da distorsión.

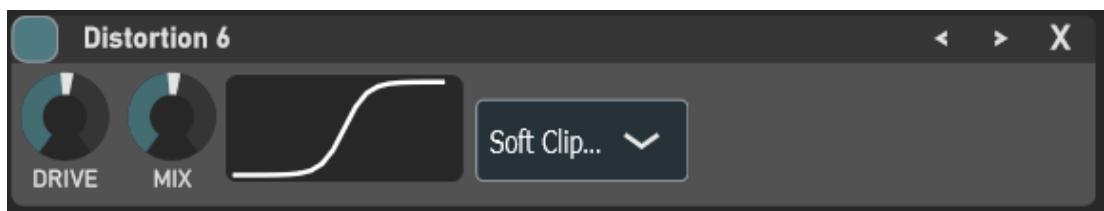


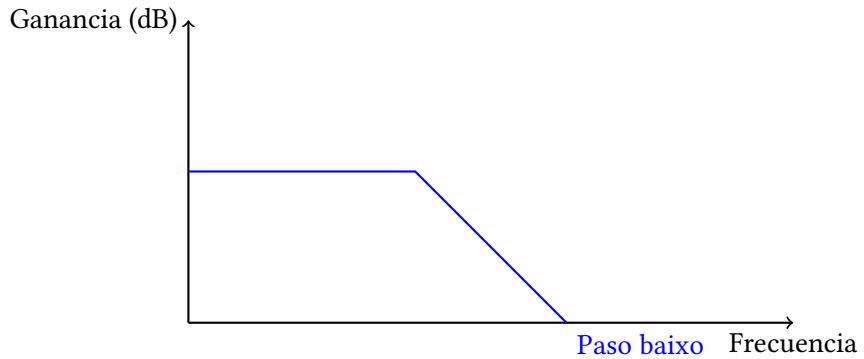
Figura 6.20: Módulo da distorsión.

6.5.2 Filtro

O efecto de tipo filtro permite eliminar as frecuencias dun sinal que estean por enriba ou por debaixo da frecuencia de corte marcada polo usuario. Os dous tipos principais de filtros utilizados para cortar frecuencias son o filtro paso baixo e o filtro paso alto.

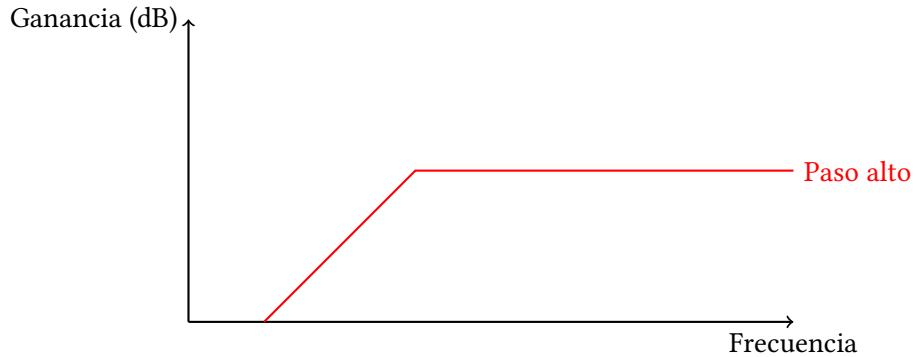
Filtro paso baixo

Elimina as frecuencias que están por enriba da frecuencia de corte, permitindo o paso daquelas que están por debaixo. Psicoacústicamente, fai que un son se perciba como máis apagado, escuro e suave.



Filtro paso alto

Este filtro elimina as frecuencias que están por debaixo da frecuencia de corte e permite o paso daquelas que están por enriba. Psicoacústicamente, fai que un son se perciba como máis lixeiro.



Podemos implementar un filtro de dúas maneiras: como un filtro de resposta ao impulso finita, [finite impulse response \(FIR\)](#), ou como un filtro con resposta ao impulso infinita [infinite impulse response \(IIR\)](#). Neste caso optouse polo [IIR](#), que precisa dunha estrutura de realimentación que toma os valores de saída anteriores para incluílos na entrada da próxima iteración. Por iso se coñencen como filtros de resposta infinita xa que, debido á realimentación, a saída teóricamente perdura para sempre. Como efecto secundario, ao aplicar un filtro tamén estamos modificando a fase das frecuencias. Mientras que esta modificación nos filtros [FIR](#) é mínima ou lineal, o que significa que todas as fases son desprazadas a mesma cantidade de tempo, nun filtro [IIR](#) a modificación é distinta para cada frecuencia.

Aproveitándonos disto, construímos o filtro mediante a combinación do sinal orixinal cunha versión modificada do mesmo que contén certas frecuencias desfasadas, o que se logra mediante un filtro de resposta infinita de tipo *all-pass*. Este tipo de filtro non afecta a amplitude das frecuencias (déixaas pasar todas), senón que, aproveitándose da característica antes descrita dun filtro IIR, modifica a fase de cada unha en distinta cantidade. O desprazamento de fase que xera o filtro *all-pass* produce cancelacións de certas frecuencias ao combinalas co sinal orixinal. A figura 6.21 mostra o diagrama de bloques do filtro IIR *all-pass* de primeira orde que se deseñou, mentres que a figura 6.22 mostra o súa resposta en frecuencia.

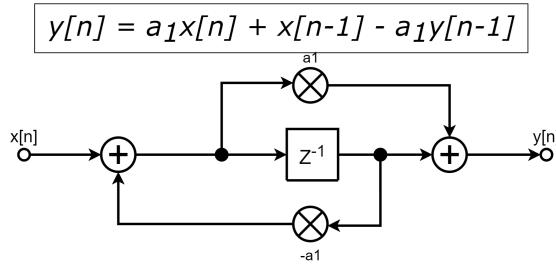


Figura 6.21: Diagrama de bloques do filtro *IIR all-pass*.



Figura 6.22: Filtro *all-pass*. Non modifica as frecuencias da sinal.

O coeficiente a_1 do filtro da figura 6.21 controla a cantidad de desfase aplicada ás frecuencias e o seu cálculo depende da frecuencia de corte f_c e da frecuencia de mostraxe f_s

$$a_1 = \frac{\tan(\pi f_c/f_s) - 1}{\tan(\pi f_c/f_s) + 1} \quad (6.10)$$

A pendente de corte do filtro é maior canto maior sexa a súa orde, que depende do número de coeficientes empregados no mesmo. Para un filtro de primeira orde como este, a pendente de corte é de 6 dB por oitava; para un de segunda orde é de aproximadamente 12 dB por oitava. Por simplicidade, este filtro permite cambiar o módulo do ecualizador, pero non a súa pendente de corte.

Para aplicar un filtro paso alto invertimos o signo da saída do filtro *all-pass* e, para aplicar un paso baixo, deixámola como está. Ademais, debemos multiplicar a saída por 1/2 para evitar que a suma de dúas frecuencias en fase supere o umbral de 0 dB (ver figura 6.23).

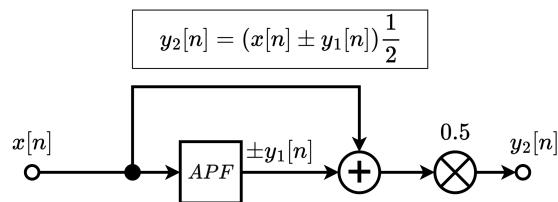


Figura 6.23: Diagrama de bloques do filtro.

Resultado final do filtro.

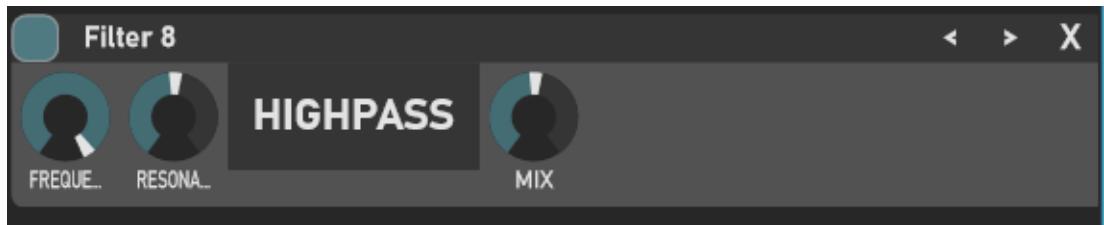
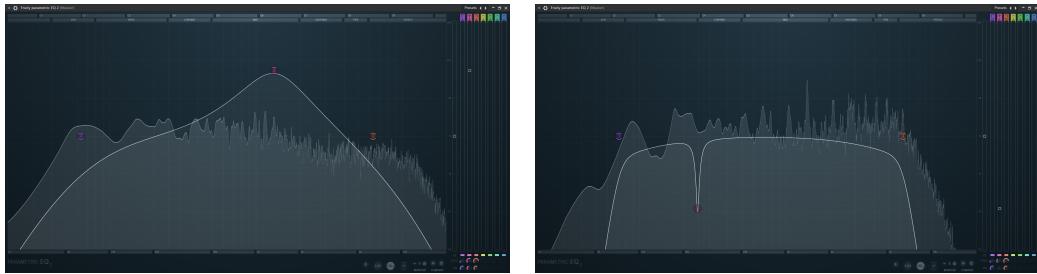


Figura 6.24: Módulo do filtro.

6.5.3 Ecualizador

Un ecualizador permite axustar o nivel de amplitud de distintas bandas de frecuencia do son (ver figura 6.25). É útil para aumentar a claridade en zonas do espectro audible ou para eliminar ou reducir frecuencias problemáticas. Para este sintetizador, implementouse un ecualizador sinxelo que actúa nas tres bandas de frecuencia principais: baixa, media e alta. A banda baixa e a alta utilizan filtros paso alto e paso baixo, respectivamente, o que permiteo controlar a pendente dos mesmos en catro valores: 12, 24, 36 ou 48 dB por oitava. A banda media utiliza un filtro de campá que permite realzar ou reducir a amplitude nun rango específico de frecuencias. O filtro de campá tamén permite modificar o factor Q ou pico de resonancia que controla o ancho da área afectada pola ganancia aplicada.



(a) Filtros paso baixo e alto con **pendente suave** e filtro de campá que **realza** frecuencias cun **factor Q baixo**.

(b) Filtros paso baixo e alto con **pendente pronunciada** e filtro de campá que **atenúa** frecuencias cun **factor Q elevado**.

Figura 6.25: Ecualizador de FL Studio que simula distintas configuracións das tres bandas permitidas por este sintetizador.

O ecualizador foi implementado coas clases `Filter` e `ProcessorChain` de JUCE. Para o filtro, escolleuse a versión `IIR`. O motivo de elixir a implementación de JUCE en lugar da propia é que esta última non permite establecer diferentes pendentes. A clase `ProcessorChain` facilita o procesamento secuencial dos filtros mediante o método `Process()` (ver listaxe 6.14).

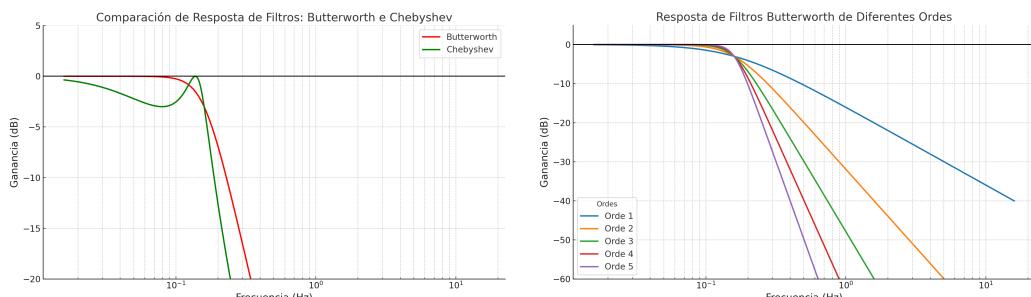
```

1  using Filter = Filter<float>;
2  using CutFilter = ProcessorChain<Filter, Filter, Filter, Filter>;
3  using MonoChain = ProcessorChain<CutFilter, Filter, CutFilter>;

```

Listaxe 6.14: Definición da clase `ProcessorChain`.

A cada canle (esquerda e dereita) correspondele unha instancia de `MonoChain`, é dicir, unha liña de procesamento independente. Antes de procesar cada `MonoChain` actualízanse os coeficientes dos filtros internos. Os coeficientes do filtro paso baixo e paso alto actualízanse segundo o método *Butterworth*, que permite obter unha resposta en frecuencia más plana na banda de paso en comparación con outros métodos como *Chebyshev* (ver figura 6.26).



(a) Comparativa da resposta en frecuencia dun filtro Butterworth fronte á dun Chebyshev.

(b) Comparativa de pendente en distintas ordes de Butterworth.

Figura 6.26: Comparativa da resposta en frecuencia dun filtro Butterworth fronte á dun Chebyshev.

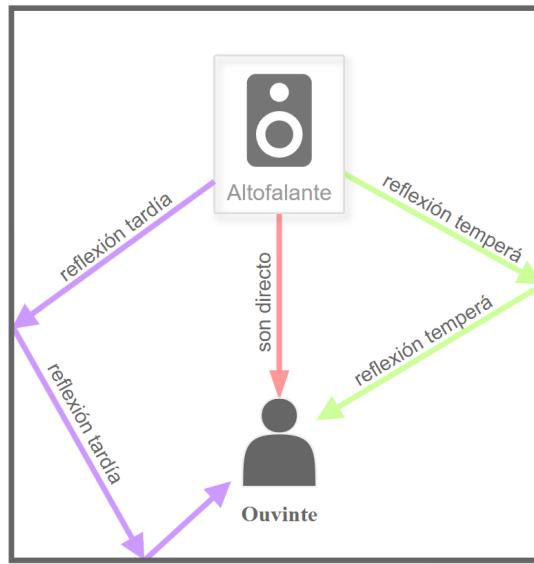


Figura 6.28: Diagrama mostrando o comportamento das reflexións temperás, tardías e son directo.

Resultado final do ecualizador.

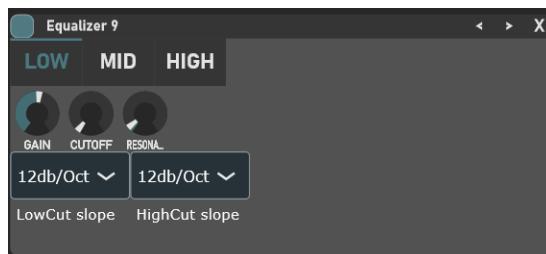


Figura 6.27: Módulo do ecualizador.

6.5.4 Reverberación

Un efecto de reverberación simula o comportamento do son dentro dun espazo físico, como unha sala, teatro ou cova. A reverberación ocorre cando o son é reflectido nas paredes dese espazo e chega ao ouvinte en múltiples momentos, direccións e intensidades, xerando unha sensación espacial e de profundidade (ver figura 6.28).

O algoritmo de reverberación descrito e implementado baséase no traballo realizado por Signal Smith, [27], adaptando e reorganizando o código recollido en [28] para lograr unha mellor eficiencia. Tamén se empregou a súa libraría [29] para facer uso dos módulos necesarios para a implementación.

Unha complicación á hora de deseñar algoritmos de reverberación é evitar as resonancias

producidas debido á realimentación. Para isto, divídese o algoritmo de reverberación en dúas etapas: a etapa de difusión e a etapa de realimentación.

Etapa de difusión

Encárgase de difuminar, suavizar ou aumentar a densidade do sinal de entrada, de tal forma que ao estendelo no tempo se reduza a posibilidade de resonancias. Simula a **cantidadade** de reflexións do son nunha sala. O algoritmo de reverberación considera N canles, polo que o primeiro paso é dividir as dúas canles de entrada (esquerda e dereita) en N . A división faise copiando a canle esquerda nas primeiras $\frac{N}{2}$ canles e a canle dereita no resto. Isto implica que as operacións internas serán multicanle, o que favorece unha reverberación máis densa e, por suposto, más cara computacionalmente.

O segundo paso é aplicar as operacións de difusión. Para conseguir unha difusión aínda maior, estas operacións realizanse en varios pasos, repetíndose tres veces (ver figura 6.29). En cada paso, aplícanse as seguintes operacións nesta orde: delay multicanle, matriz de Hadamard, inversión de polaridade e intercambio de canles (ver figura 6.29). Cada iteración deste proceso incrementa a densidade do sinal, achegando unha difusión máis complexa.



Figura 6.29: Orde das operacións de difusión.

- Delay multicanle: Aplica un retardo diferente ás mostras por cada canle.



Figura 6.30: Delay multicanle.

- Matriz de Hadamard: Distribúe a **enerxía** do sinal entrante entre as N canles, é dicir, cada canle de saída é unha combinación lineal das canles de entrada. A figura 6.31 mostra como se reparte a enerxía da mostra de entrada na segunda canle entre as catro canles de saída.



Figura 6.31: Matriz de hadamard.

Unha matriz de Hadamard é unha matriz cadrada de orde N cuxas entradas poden ser $+1$ ou -1 e cuxas filas son ortogonais entre sí, o que significa que as entradas de cada par de filas corresponden coa metade de cada par de columnas, mentres que na outra metade as entradas son opostas.

$$H_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \quad (6.11)$$

- Inversión de polaridade: Inverte a polaridade das mostras de entrada de forma aleatoria. É dicir, cambia o signo para algunas mostras.



Figura 6.32: Inversor da polaridade.

Etapa de realimentación

Estende o sinal no tempo, creando a cola da reverberación. Isto simula o **tempo** que tardan as reflexións nunha sala en desaparecer, determinando a duración do efecto. A etapa recibe o sinal de entrada, dividido en N canais, procedente da fase de difusión, e aplícase un retardo a cada canle. Denomínase etapa de realimentación porque a saída dese retardo engádese á entrada na seguinte iteración. Na figura 6.33 móstrase o diagrama de bloques desta etapa, onde todas as operacións son multicanle. A matriz situada antes de sumar o sinal de saída co de entrada proporciona unha mestura sutil entre os canais, similar á matriz de Hadamard. Esta mestura debe ser suave, xa que, doutro modo, podería alterar a tonalidade da reverberación con colas longas.

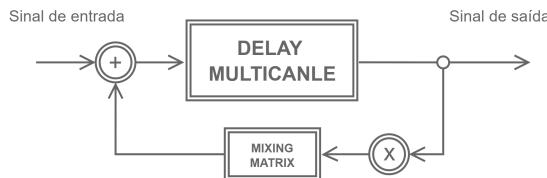


Figura 6.33: Etapa de realimentación.

Despois da etapa de realimentación, as N canles combínanse de novo nas canles esquerda e dereita.

Así, o diagrama de bloques da reverberación quedaría como na figura 6.34 e o módulo veríase como na figura 6.35.

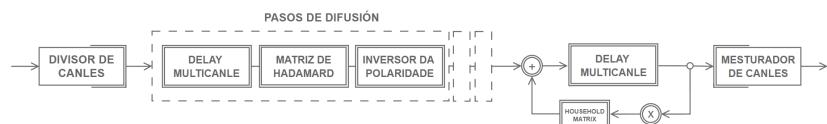


Figura 6.34: Diagrama de bloques completo da reverberación.

Resultado final da reverberación.

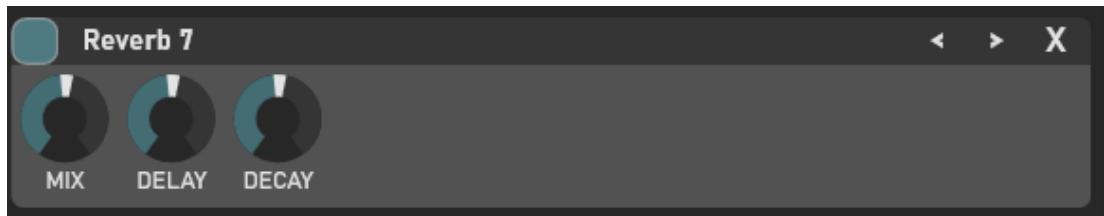


Figura 6.35: Módulo da reverberación.

6.5.5 Delay

O efecto retardo ou *delay* ten unha base similar ao da reverberación. Ámbolos dous efectos utilizan ecos para simular unha sensación de espazo. Porén, o efecto de *delay* caracterízase por producir repeticións claramente separadas e distinguibles no tempo, ao contrario da reverberación, onde os ecos mestúranse para producir unha gran densidade. Os dous parámetros principais dun efecto de *delay* son:

- **Tempo:** Permite controlar a separación entre repeticións. Para que as repeticións sexan distinguibles entre si, o intervalo entre ecos debe ser maior a 30 ms. Este parámetro está determinado polo valor de n , que indica o número de mostras de atraso aplicado ao sinal.

- **Caída:** Tamén coñecido como *feedback*, indica a velocidade coa que cae o volume dos ecos. É dicir, marca a duración do efecto, controlando a intensidade coa que o sinal atrasado se retroalimenta no sistema.

O efecto de *delay* foi implementado mediante un simple *buffer* ou cola circular que almacena os valores das mostras de son atrasadas, permitindo que se reutilicen para producir os ecos. Na implementación, o parámetro de tempo determina o número de mostras almacenadas no *buffer*, mentres que a caída controla a intensidade do sinal que se reintroduce no sistema.

A figura 6.36 mostra o diagrama de bloques deste efecto. O bloque z^{-n} representa o retardo do sinal, que é controlado polo parámetro de tempo. A ganancia de retroalimentación regula a caída do sinal, mentres que o control de mestura (*mix*) axusta a proporción entre o sinal orixinal e o procesado.

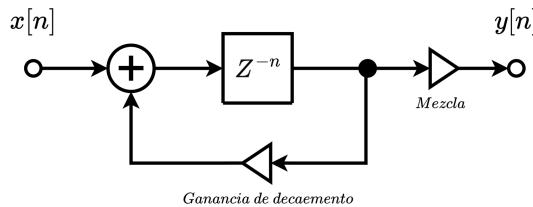


Figura 6.36: Diagrama de bloques do efecto de *delay*

Resultado final do *delay*



Figura 6.37: Módulo de *delay*.

6.6 Parámetros xerais

Existen uns parámetros que non forman parte de ningunha destas seccións comentadas. Estes parámetros forman parte de configuración xeral do sintetizador.

6.6.1 Número de voces

O sintetizador soporta ata un máximo de 8 voces simultáneas. Como se dixo no deseño, unha “voz” refírese a unha instancia individual do son xerado polo sintetizador, composta po-

los osciladores e moduladores necesarios para producir unha nota. Cada vez que se toca unha nova nota en modo polifónico, utilízase unha voz diferente para que as notas poidan soar ao mesmo tempo, mantendo a súa independencia.

Decidiuse establecer o límite en 8 voces porque, por un lado, en moi poucas ocasións se necesitan 8 notas ou máis soando ao mesmo tempo e, por outro, por razóns de eficiencia. Como se explicou no deseño, engadir unha nova voz implica cargar e procesar máis instancias de osciladores e moduladores.

O usuario pode establecer o número de voces que desexe mediante un parámetro de tipo slider. Cando selecciona unha soa voz, dise que o sintetizador está en modo monofónico. Cando hai máis dunha voz, está en modo polifónico.

En realidade, o sintetizador sempre ten 8 instancias de voces cargadas, cada unha cos seus respectivos osciladores e moduladores. O que ocorre cando o usuario establece un número menor a 8 voces é que algunas delas quedan deshabilitadas, de forma que non poden ser escollidas polo algoritmo de selección de voces.

O “algoritmo de selección de voces” é o encargado de decidir que voz utilizar cada vez que se toca unha nova nota. Cando se preme unha tecla, o algoritmo busca unha voz disponible, é dicir, unha que estea desactivada ou que non estea a reproducir outra nota en ese momento. Este algoritmo foi modificado, partindo da base implementada por JUCE, para comprobar se cada voz está activa ou non, de xeito que só as voces habilitadas se poidan escoller para reproducir novas notas. Así, o sintetizador usa as voces disponibles sen desperdiciar recursos en voces que están desactivadas polo usuario. Se se chegan a pulsar máis de 8 notas a vez, o algoritmo remplazará a voz máis vella para que reproduza a nova nota. A implementación completa deste algoritmo pódese ver no apéndice (listado A.1).

6.6.2 Legato

Como se mencionou na fase de análise, o legato é unha técnica musical na que as notas que se solapan xeran unha transición suave e continua entre elas, sen interrupcións. Esta transición conséguese incrementando progresivamente a frecuencia da nota ata acadar a frecuencia da seguinte. Por exemplo, se tocamos unha nota LA (440,0 Hz) e, mentres a mantemos presionada, engadimos unha nota DO (523,251 Hz), o sintetizador realiza unha transición fluída desde os 440 Hz ata os 523,251 Hz, creando un efecto de unión entre ambas notas. A figura 6.38 ilustra este efecto de legato mediante o piano virtual de FL Studio, orientado verticalmente. O eixe horizontal representa o tempo transcorrido. A liña laranxa, engadida mediante edición da imaxe, representa a frecuencia que se está a reproducir en cada momento. O período de transición entre unha nota e outra pódese acortar ou alongar mediante un parámetro; a figura 6.39 mostra un alongamento do período de transición entre notas.

No caso de ter a polifonía activada, ao tocar unha segunda nota crearase unha nova voz

que partirá desde a frecuencia reproducida pola voz anterior. A figura 6.40 mostra o legato en modo polifónico, onde a cor vermella representa a frecuencia da segunda voz.

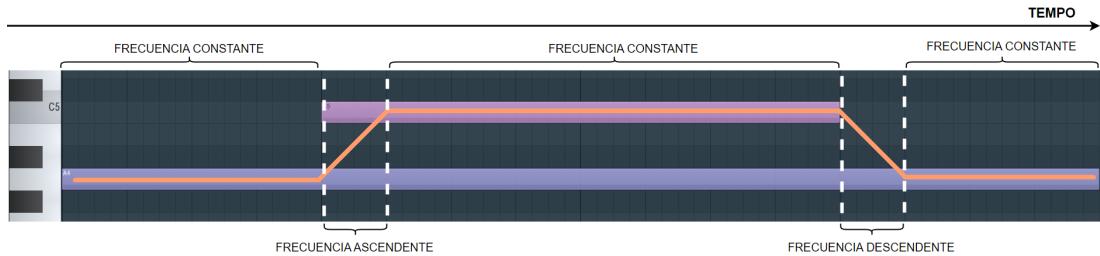


Figura 6.38: Legato rápido actuando sobre o sintetizador en modo monofónico.



Figura 6.39: Legato lento actuando sobre o sintetizador en modo monofónico.

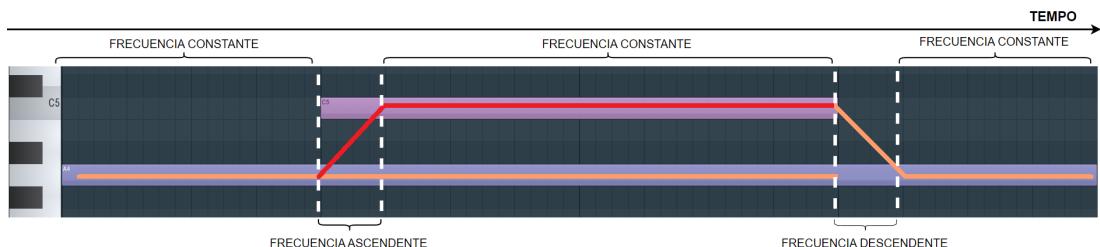


Figura 6.40: Legato rápido actuando sobre o sintetizador en modo polifónico.

Modificación do algoritmo de selección de voces

Para que o sintetizador poida soportar o efecto de legato, foi necesario modificar novamente o algoritmo de selección de voces. Esta modificación é esencial para permitir unha transición de descenso cara á nota anterior cando se solta unha nota. A modificación consiste en engadir, sobre o algoritmo base de JUCE, unha lista das notas que foron presionadas e que ainda non foron soltadas. Esta lista permite ao sintetizador coñecer en todo momento cales son as notas activas e cal foi a última presionada.

Funcionamento do novo algoritmo

- **Engadir unha nota á lista:** Cando se presiona unha nova nota, esta engádese á lista de notas activas (notas que están sendo sostidas).
- **Eliminar unha nota da lista:** Cando unha nota é soltada, elimínase da lista de notas activas.
- **Transición á última nota activa:** Se, ao soltar unha nota, aínda quedan outras notas na lista, o sintetizador realiza unha transición suave (legato) cara á frecuencia da última nota activa.

Por exemplo, supoñendo que temos dúas notas A e B activas simultaneamente:

- A nota **A** está a reproducir un **LA** de 440,0 Hz.
- A nota **B** está a reproducir un **DO** de 523,251 Hz.

Se a nota **B** deixa de estar presionada, entón:

1. A voz que estaba asignada a **B** deixa de procesarse.
2. A voz de **A** entra nun estado de transición, baixando suavemente dende os 523,251 Hz ata os 440,0 Hz, que era a frecuencia orixinal de **A**.

Pode verse visualmente na figura 6.40, durante a transición de retorno de DO a LA.

6.7 Visualización do resultado final

As seguintes figuras mostran o resultado final da interface gráfica unha vez todos os módulos foron implementados. A figura 6.41 mostra a vista de osciladores e a figura 6.42 mostra a vista de efectos. A estructura xeral da interface segue o planificado na fase de deseño.

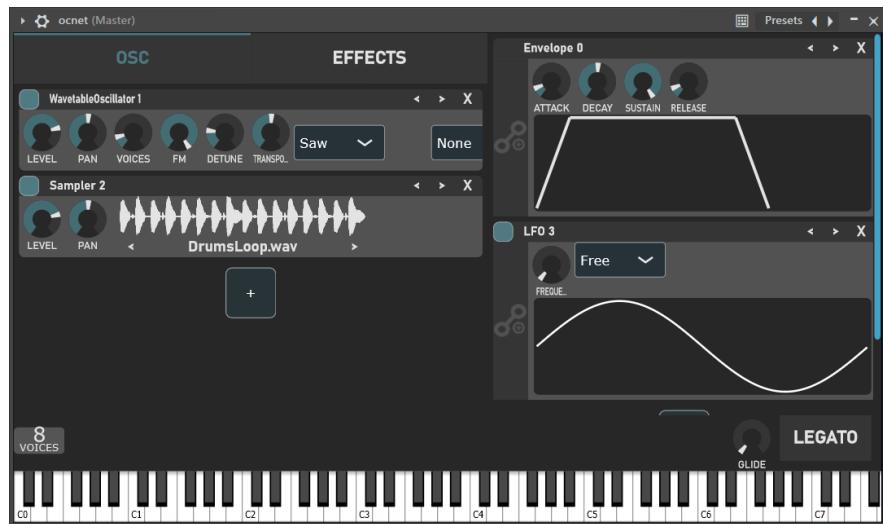


Figura 6.41: Visualización da sección de osciladores.



Figura 6.42: Visualización da sección de efectos.

Capítulo 7

Probas

Neste capítulo describense as diferentes probas realizadas durante e despois do desenvolvemento do sintetizador. Inclúense probas de software, de compatibilidade con diferentes [DAW](#), equipamentos e sistemas operativos, así como avaliaciós da calidade final, do deseño, da estética e, naturalmente, do son xerado.

7.1 JUCE Plugin Host

JUCE Plugin Host é unha aplicación integrada na biblioteca JUCE [12] que permite cargar e probar *plugins* de son nun contorno dedicado, sen precisar exportalos a un [DAW](#). Dende o comezo do proxecto, JUCE Plugin Host foi unha ferramenta esencial, xa que permitiu verificar o funcionamento dos módulos e realizar probas de xeito inmediato tras cada compilación. Isto simplificou e acelerou o proceso de desenvolvemento, ao eliminar a necesidade de exportar e cargar o *plugin* en [DAW](#) externos cada vez que se recompilaba o código. A figura 7.1 mostra a interface gráfica desta aplicación.

7.2 Probas software

As probas de software realizáronse de xeito continuo durante o desenvolvemento dos módulos do sintetizador.

7.2.1 Probas de validación

As probas de validación verifican se un software satisfai as necesidades e expectativas do usuario final. Para comprobar este tipo de probas nun contorno real, realizaríanse reunións co cliente. Non obstante, neste caso os requisitos foron establecidos polo propio desenvolvedor, polo que se limitan á revisión periódica dos requisitos completados e os pendentes de desenvolver.

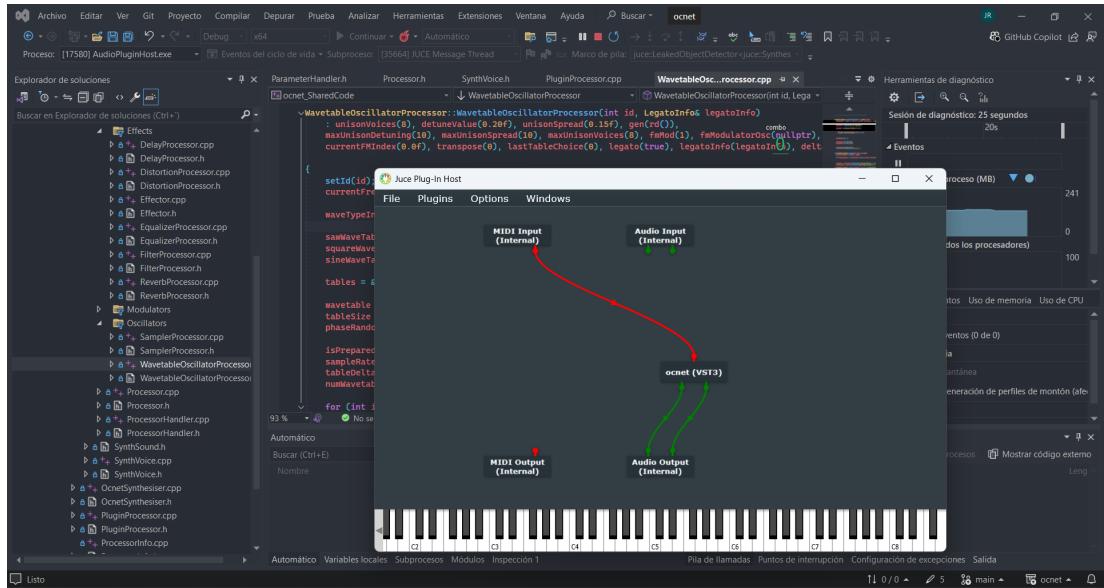


Figura 7.1: JUCE Plugin Host executándose directamente tras compilar o proxecto. A conexión vermella representa o sinal MIDI de entrada e as liñas de cor verde representan a saída estéreo do sintetizador. O nodo central é o *plugin* a probar, que se abre ao facer dobre clic enriba.

7.2.2 Probas de verificación

As probas de verificación deben responder á pregunta: "Estamos a construír o software correctamente?" O obxectivo principal destas probas foi comprobar o funcionamento axeitado dos módulos, parámetros e modulacións. Outro aspecto fundamental foi verificar o rendemento e o consumo de recursos dos algoritmos. Así mesmo, comprobouse a integración de cada módulo co resto do sistema.

7.2.3 Probas de compatibilidade

Para realizar estas probas contouse coa colaboración dun produtor musical que instalou o software no seu propio equipamento e verificou o seu funcionamento. Este produtor empregaba unha versión algo máis antiga de FL Studio en comparación coa utilizada polo desenvolvedor, así como unha copia de Ableton. Logo de confirmar a correcta instalación e funcionamento do software noutro equipo e en diferentes contornos de traballo, considerouse superada a proba de compatibilidade.

7.2.4 Probas de usabilidade e calidad

Do mesmo xeito que coas probas de compatibilidade, foi preciso contar con persoas alleas ao proxecto. Para esta proba participou o mesmo produtor musical das probas anteriores.

Realizouse unha reunión telemática na que se lle pediu que utilizase o sintetizador. Durante a proba, o avaliador debe observar e documentar como interactúa o produtor co sintetizador e cal é o seu proceso de pensamento. A proba dura aproximadamente 10 minutos e, ao rematar, formúlanselle ao produtor as seguintes preguntas:

- Resultouche doado utilizar o *plugin*?
- Atopaches algo confuso ou pouco intuitivo?
- Consideras suficiente a calidade do son?
- Paréceche excesivo o consumo de [CPU](#)?
- En que xéneros ou proxectos musicais empregarías este sintetizador?
- Que melloras suxerirías para facelo máis útil ou accesible?
- Que che parece a estética do sintetizador?

7.3 Probas automatizadas

Ademais das probas manuais mencionadas anteriormente, tamén se realizaron probas automáticas de forma continuada mediante a aplicación PluginVal [30], desenvolvida por Tracktion, que permite comprobar e verificar a estabilidade entre un *plugin* e un [DAW](#) calquera de maneira sinxela e xeral, xa que testa as accións básicas que todo [DAW](#) realiza sobre un *plugin*. Entre as comprobacións realizadas inclúense accións como abrir e pechar o *plugin*, recargar parámetros e modular os parámetros entre o [DAW](#) e o *plugin*.

PluginVal (ver figura 7.2) permite establecer un nivel de esixencia entre 1 e 10. Canto maior é o nivel, maior é o número de tests realizados, cubrindo un maior número de casos potenciais de fallo. Considérase que un *plugin* é estable cando supera, polo menos, os tests do nivel 5.

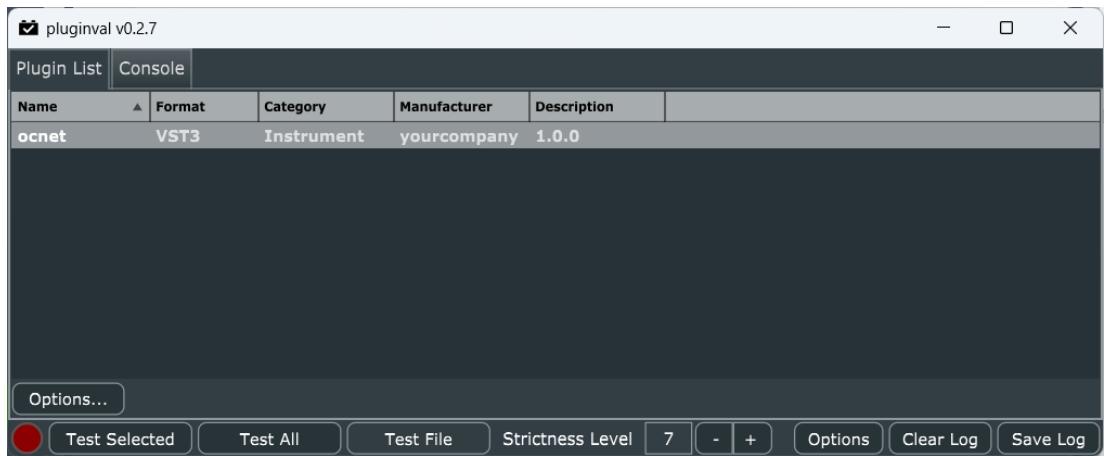


Figura 7.2: Interface gráfica da aplicación PluginVal co plugin que queremos probar seleccionado. Na barra de abaixo pódese seleccionar o nivel de esixencia desexado.

Para usar PluginVal simplemente hai que seleccionar o *plugin* en formato .vst3 que queríamos probar e establecer o nivel de esixencia, como mostra a figura 7.2. Este *plugin* chega a superar o máximo nivel, é dicir, o nivel 10. Entre as comprobacións realizadas están:

- **Escaneo de tipos coñecidos:** Comprobación dos tipos compatibles do *plugin* (VST3, VST2, AU etc.).
- **Información do *plugin*:** Verificación dos detalles do *plugin*: nome, precisión dobre, latencia e lonxitude de cola.
- **Apertura do *plugin* (en frío):** Medición do tempo necesario para abrir o *plugin* sen carga previa.
- **Apertura do *plugin* (en quente):** Medición do tempo para abrir o *plugin* despois de ter sido aberto previamente.
- **Apertura do editor durante o procesamento:** Verificación da capacidade do editor para abrirse mentres o *plugin* está procesando son.
- **Procesamento de son:** Comprobación do rendemento con distintas taxas de mostraxe (44100, 48000 e 96.000 Hz) e tamaños de bloque.
- **Estado do *plugin*:** Verificación do estado interno do *plugin*.
- **Restauración do estado do *plugin*:** Comprobación da restauración correcta do estado do *plugin*.
- **Automatización:** Probas de automatización de parámetros con diferentes configuracións de taxas de mostraxe e tamaños de bloque.

- **Parámetros do plugin:** Verificación do funcionamento correcto dos parámetros do *plugin*.
- **Estado do fío en segundo plano:** Verificación do funcionamento de fíos en segundo plano.
- **Seguridade dos fíos de parámetros:** Comprobación da seguridade dos fíos dos parámetros.
- **Estrés do editor:** Probas de apertura do editor con procesamento liberado e con diferentes tamaños de bloque e taxas de mostraxe.
- **Asignación de memoria durante o proceso:** Comprobación da asignación de memoria durante o procesamento con diferentes taxas de mostraxe e tamaños de bloque.
- **Proceso con un tamaño de bloque maior ao preparado:** Comprobación da capacidade do *plugin* para manexar bloques de tamaño maior ao que está preparado.
- **Fuzzing de parámetros:** Comprobación da robustez do *plugin* ao cambiar aleatoriamente os parámetros.

A saída de PluginVal pode verse na listaxe A.4 do apéndice A.

7.4 Detección de fugas de memoria

Para a detección de posibles fugas de memoria empregouse a macro propia de JUCE JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR. O uso desta macro é moi sinxelo: simplemente debe colocarse no ficheiro de cabecera (.h) de cada clase na que se desexen comprobar as fugas de memoria. Case tódalas clases deste proxecto levan incluída esta macro. Cada vez que compilamos e executamos o proxecto, JUCE detecta as posibles fugas de memoria. Ao pechar o Plugin Host, saltaría unha excepción e imprimiríase un *log* na consola cos detalles das fugas de memoria, se os houbese. Neste caso, todas as fugas de memoria detectadas ao longo do proxecto foron corrixidas. A saída de JUCE, en caso de que se producira unha fuga de memoria, sería como na listaxe 7.1

```
1 *** Leaked objects detected: 1 instance(s) of class <NomeDaClase>
2 *** Leaked object: <dirección_da_fuga_de_memoria>
```

Listaxe 7.1: Exemplo da saída de JUCE en caso de fuga de memoria.

Por outra banda, o depurador de Visual Studio tamén foi de gran axuda para a detección de fugas de memoria. Porén, existe un *bug* rexistrado que provoca que Visual Studio sempre detecte fugas de memoria, mais son alleas ao traballo realizado. Este *bug* foi reportado varias

vezes, un exemplo pode verse en [31], e a saída correspondente móstrase na listaxe A.5 do apéndice A.

7.5 Probas de rendemento

Para detectar as funcións e seccións de código que máis tempo de CPU consumían fixose uso do xerador de perfiles de Visual Studio. Esta ferramenta foi moi útil para realizar comparacións de rendemento entre dúas versións do mesmo algoritmo, permitindo visualizar mediante porcentaxes e cores que versión resultaba máis eficiente. A figura 7.3 mostra a facilidade coa que se pode identificar que partes do código son más custosas (neste caso a lectura e máis a escritura).

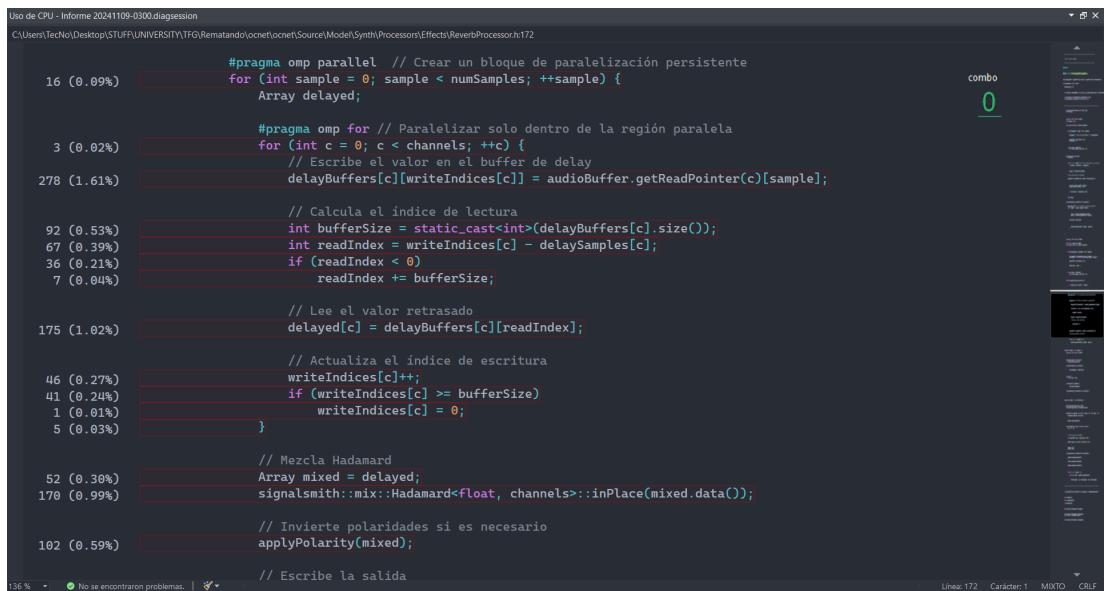


Figura 7.3: Perfil dunha parte do algoritmo de reverberación.

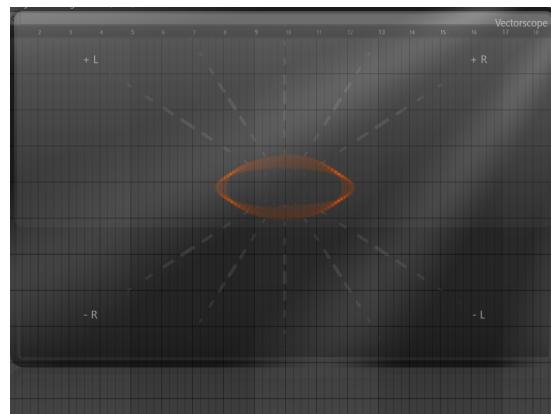
7.6 Probas de son

Este tipo de probas realizáronse mediante programas e *plugins* de medición de son. Os principais tipos de ferramentas empregadas foron analizadores de espectro, vectorscopios e osciloscopios. Principalmente, fixose uso de Wave Candy [32], un *plugin* nativo de FL Studio que incorpora un osciloscopio, un vectorscopio e un spectrograma (ver figuras 7.4a e 7.4b). Tamén se empregou o *plugin* Span [33], de Voxengo, para analizar o espectro de frecuencias (ver figura 7.4c).

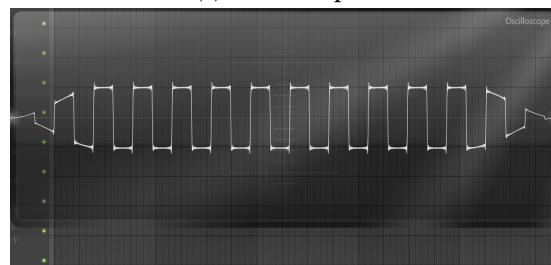
Estas probas realizáronse tras a implementación de cada módulo para garantir un son de calidade e corrixir errores. Entre os problemas que se puideron detectar mediante o uso destas

ferramentas atopanse:

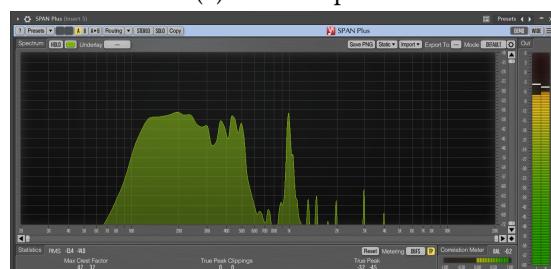
- A detección e corrección do *aliasing* presente nas formas de onda xeradas polo oscilador, que se visualizou grazas a Span.
- Un lixeiro desprazamento do son cara á dereita durante a implementación do paneo, detectado coa axuda do vectorscopio.
- A xeración incorrecta da forma de onda, identificada grazas ao osciloscopio.



(a) Vectorscopio.



(b) Osciloscopio.



(c) Análise de espectro (Span).

Figura 7.4: vectorscopio, osciloscopio e analizador de espectro.

Capítulo 8

Conclusións

Finalizado o proxecto, podemos concluír que, en termos xerais, se cumpriron os obxectivos establecidos. Obtívose un sintetizador totalmente funcional, modular e útil que pode ser empregado tanto por principiantes como por profesionais do ámbito musical para sintetizar sons que poden utilizar nas súas composicións. Ademais, en case todos os módulos conseguiuse cumplir o obxectivo secundario de implementar un algoritmo eficiente que non consumise demasiados recursos do computador, como o tempo de CPU ou memoria. O único requisito que non se conseguiu cumplir foi a incorporación de varias liñas de efectos en paralelo. A decisión de non implementar esta funcionalidade baseouse no feito de que complicaría significativamente a usabilidade do sistema, engadindo unha complexidade innecesaria para unha característica que non é fundamental nin crítica para o funcionamento do sintetizador.

Este proxecto proporciona unha ferramenta de gran valor para a creación musical, permitindo aos usuarios explorar novas posibilidades sonoras grazas á súa modularidade e funcionalidade. O sintetizador desenvolvido non só facilita a aprendizaxe de conceptos fundamentais de síntese de son para principiantes, senón que tamén ofrece opcións avanzadas que poden ser utilizadas por profesionais para enriquecer as súas composicións. Deste xeito, contribúe ao crecemento da creatividade musical e á innovación dentro da comunidade de artistas e produtores musicais.

Traballar neste proxecto foi unha oportunidade para aprender a linguaxe de programación C++, o *framework* JUCE, as bases dos algoritmos de procesamento do sinal e a aplicación de técnicas para o seu procesamento eficiente. Ás veces, isto fai necesario o sacrificio da abstracción ou da modularidade do código en favor dun maior rendemento. Tamén se pon de manifesto a importancia dunha boa planificación e deseño previos ao inicio das tarefas de desenvolvemento, sobre todo nun proxecto tan grande como este. Dispoñer dunha boa organización ao longo do desenvolvemento do proxecto resulta clave para a súa consecución exitosa.

A principal dificultade atopada durante o desenvolvemento foi lograr unha alta capacida-

de de adaptación. Isto debeuse á necesidade de axustarse a cambios de enfoque e mentalidade para implementar algoritmos con bases e fundamentos completamente distintos. Unha das tarefas más complexas foi desenvolver un oscilador libre de *aliasing*. Este proceso levou aproximadamente un mes, atrasando o proxecto non só pola súa propia complexidade, senón tamén porque foi unha das primeiras tarefas realizadas. A falta de experiencia inicial dificultou a comprensión dos conceptos necesarios para a súa implementación. Outra tarefa especialmente complicada foi a creación do novo sistema de parámetros. Foi preciso deseñar un sistema desde cero, de gran tamaño, que permitise soportar actualizacións en tempo real tanto por parte do usuario como a través de modulacións, ou mesmo de ambas ao mesmo tempo. Por último, o deseño dunha interface atractiva tamén supuxo un reto, debido á inexperiencia do desenvolvedor no ámbito do deseño de interfaces gráficas de usuario.

Durante o desenvolvemento do proxecto tamén foron necesarias as competencias adquiridas na titulación. Sirva como exemplo que, para poder desenvolver algoritmos eficientes, foi necesario emplegar técnicas como a paralelización dos cálculos (*Concorrencia e Paralelismo*) ou a optimización do proceso de compilación e o desenvolvemento de bucles (*Estrutura de Computadores*). Tamén foron fundamentais as competencias da materia *Dispositivos Hardware e Interfaces*, especialmente os aspectos relacionados cos sinais de son dixitais. De cara á planificación, xestión e o desenvolvemento do proxecto resultaron chave as competencias das materias *Xestión de Proxectos e Proceso Software*. Por último, a implementación de algoritmos eficientes a partir de fundamentos matemáticos está estreitamente relacionada coa mención en Computación.

Actualmente, este proxecto atópase baixo a licenza AGPLv3. Isto débese a que JUCE establece que, para poder emplegar o seu framework na súa versión gratuíta, é necesario que o proxecto estea baixo esta licenza. É importante ter en conta que a licenza AGPLv3 permite a distribución e incluso a comercialización do software, pero obriga a que o código fonte do proxecto sexa accesible para os usuarios baixo os mesmos termos da licenza AGPLv3. De cara ao futuro mantemento do software, é probable que sexa necesario incorporar un sistema de monetización. Dado que o software será gratuíto, este sistema podería basearse na venda de funcionalidades especiais, temas visuais ou paquetes de sons adicionais para o *sampler*. Porén, no caso de que os beneficios anuais superen os 20.000 USD, será obligatorio actualizar a unha licenza de pago, específica de JUCE, que permita a distribución de software de código pechado sen as restricións da licenza AGPLv3.

Traballar neste proxecto supuxo un reforzo no meu interese como estudiante e na miña motivación á hora de experimentar e traballar en proxectos relacionados co procesamento de sinais de son. En definitiva, este proxecto non só cumpliu cos seus obxectivos técnicos, senón que tamén serviu como unha valiosa experiencia de aprendizaxe e crecemento profesional. A combinación de coñecementos técnicos, xestión de proxectos e creatividade musical fixo deste

CAPÍTULO 8. CONCLUSIÓNS

un proxecto especialmente enriquecedor, sentando as bases para futuros desenvolvimentos no campo do son dixital.

Capítulo 9

Liñas Futuras

Neste capítulo detállanse posibles melloras e funcionalidades que non estaban incluídas nos obxectivos iniciais do proxecto ou que, por falta de tempo, non se puideron implementar.

- Engadir un parámetro para controlar a concentración da potencia das voces do unison. Actualmente, a distribución é uniforme, pero sería beneficioso permitir axustar esta concentración. Ademais, mellorar o algoritmo de incorporación de voces unison para optimizar o rendemento e a calidade sonora.
- Implementar a funcionalidade de varias liñas de efectos. Esta opción era un requisito inicial, pero non foi posible levala a cabo debido a limitacións de tempo e ao nivel de coñecemento requirido.
- Mellorar a eficiencia dos algoritmos de procesamento de son, especialmente os de reverberación e distorsión.
- Refinar a aparencia da interface gráfica e solucionar erros visuais.
- Engadir opcións de configuración avanzadas, como a posibilidade de elixir entre distintos algoritmos de roubo de voces, axustar a calidade do son ou aplicar temas visuais.
- Crear unha páxina web desde a que descargar o plugin. Esta páxina incluiría información sobre as funcionalidades, instrucións de instalación e uso, e unha sección de soporte para resolver posibles dúbidas.
- Aplicar técnicas SIMD (Single Instruction, Multiple Data) para optimizar o rendemento en operacións paralelas. Estas técnicas permiten executar varias operacións de forma simultánea, o que pode mellorar significativamente a eficiencia do sintetizador.
- Explorar o uso de intelixencia artificial para ofrecer funcionalidades avanzadas, como a xeración de *presets* automáticos baseados en parámetros de usuario, a análise de son para recomendación de efectos ou a mellora automática da calidade do son.

Apéndices

Apéndice A

Algoritmos, códigos, scripts e saídas

A seguinte ligazón [Repositorio de GitHub](#) permite acceder ao repositorio en GitHub do proxecto [17].

A.1 Algoritmo de selección de voces

Este é o algoritmo completo empregado para a selección de voces. Parte da base implementada por JUCE. Os fragmentos de código en vermello son aquelas partes que foron engadidas para adaptalo ao contexto deste sintetizador.

```
1 juce::SynthesiserVoice*
2     OcnctSynthesiser::findVoiceToSteal(juce::SynthesiserSound*
3         soundToPlay, int, int midiNoteNumber) const
4     {
5         // This voice-stealing algorithm applies the following
6         // heuristics:
7         // - Re-use the oldest notes first
8         // - Protect the lowest & topmost notes, even if sustained, but
9         // not if they've been released.
10
11         // apparently you are trying to render audio without having any
12         // voices...
13         jassert(!voices.isEmpty());
14
15         // These are the voices we want to protect (ie: only steal if
16         // unavoidable)
17         juce::SynthesiserVoice* low = nullptr; // Lowest sounding note,
18         // might be sustained, but NOT in release phase
19         juce::SynthesiserVoice* top = nullptr; // Highest sounding
20         // note, might be sustained, but NOT in release phase
21
22         // All major OSes use double-locking so this will be lock- and
```

```
wait-free as long as the lock is not
15 // contended. This is always the case if you do not call
  findVoiceToSteal on multiple threads at
16 // the same time.
17 const juce::ScopedLock sl(stealLock);

18 // this is a list of voices we can steal, sorted by how long
19 they've been running
20 usableVoicesToStealArray.clear();

21 for (auto* voice : voices)
22 {
23     if (auto synthVoice = dynamic_cast<SynthVoice*>(voice)) {
24         if (voice->canPlaySound(soundToPlay) &&
25             synthVoice->isEnabled())
26         {
27             jassert(voice->isVoiceActive()); // We wouldn't be
here otherwise

28         usableVoicesToStealArray.add(voice);

29
30             // NB: Using a functor rather than a lambda here
due to scare-stories about
31             // compilers generating code containing heap
allocations..
32             struct Sorter
33             {
34                 bool operator() (const juce::SynthesiserVoice*
35 a, const juce::SynthesiserVoice* b) const noexcept { return
36 a->wasStartedBefore(*b); }
37             };
38
39             std::sort(usableVoicesToStealArray.begin(),
40 usableVoicesToStealArray.end(), Sorter());
41
42             if (!voice->isPlayingButReleased()) // Don't
protect released notes
43             {
44                 auto note = voice->getCurrentlyPlayingNote();
45
46                 if (low == nullptr || note <
47 low->getCurrentlyPlayingNote())
48                     low = voice;

49
50                 if (top == nullptr || note >
51 top->getCurrentlyPlayingNote())
52                     top = voice;
53             }
54         }
55     }
56 }
```

```

48         top = voice;
49     }
50     }
51   }
52 }
53
54 // Eliminate pathological cases (ie: only 1 note playing): we
55 // always give precedence to the lowest note(s)
56 if (top == low)
57     top = nullptr;
58
59 // The oldest note that's playing with the target pitch is
60 // ideal..
61 for (auto* voice : usableVoicesToStealArray)
62     if (voice->getCurrentlyPlayingNote() == midiNoteNumber)
63         return voice;
64
65 // Oldest voice that has been released (no finger on it and not
66 // held by sustain pedal)
67 for (auto* voice : usableVoicesToStealArray)
68     if (voice != low && voice != top &&
69 voice->isPlayingButReleased())
70     return voice;
71
72 // Oldest voice that doesn't have a finger on it:
73 for (auto* voice : usableVoicesToStealArray)
74     if (voice != low && voice != top && !voice->isKeyDown())
75         return voice;
76
77 // Oldest voice that isn't protected
78 for (auto* voice : usableVoicesToStealArray)
79     if (voice != low && voice != top)
80         return voice;
81
82 // We've only got "protected" voices now: lowest note takes
83 // priority
84 jassert(low != nullptr);
85
86 // Duophonic synth: give priority to the bass note:
87 if (top != nullptr)
88     return top;
89
90 return low;
91 }
92
93 juce::SynthesiserVoice*

```

```

OcnetSynthesiser::findFreeVoice(juce::SynthesiserSound*
soundToPlay, int midiChannel, int midiNoteNumber, bool
stealIfNoneAvailable) const
{
    const juce::ScopedLock s1(lock);

    for (auto* voice : voices) {
        if (auto synthVoice = dynamic_cast<SynthVoice*>(voice)) {
            if (!synthVoice->isVoiceActive()) &&
synthVoice->canPlaySound(soundToPlay) && synthVoice->isEnabled())
                return synthVoice;
        }
    }

    if (stealIfNoneAvailable)
        return findVoiceToSteal(soundToPlay, midiChannel,
midiNoteNumber);

    return nullptr;
}

void OcnetSynthesiser::noteOff(int midiChannel, int midiNoteNumber,
float velocity, bool allowTailOff)
{
    const juce::ScopedLock s1(lock);

    for (auto* voice : voices)
    {
        if (voice->getCurrentlyPlayingNote() == midiNoteNumber
&& voice->isPlayingChannel(midiChannel))
        {
            if (auto sound = voice->getCurrentlyPlayingSound())
            {
                if (sound->appliesToNote(midiNoteNumber)
&& sound->appliesToChannel(midiChannel))
                {
                    //jassert(!voice->keyIsDown ||

voice->isSustainPedalDown() == sustainPedalsDown[midiChannel]);

                    voice->setKeyDown(false);

                    if (!(voice->isSustainPedalDown() ||
voice->isSostenutoPedalDown())))
{
                        if
(processorInfo.legatoInfo.legatoIsActive) {
notesThatDidntEnd.remove(

```

```

126
127     notesThatDidntEnd.indexOf(voice->getCurrentlyPlayingNote())));
128         }
129         stopVoice(voice, velocity, allowTailOff);
130     }
131
132         // Si hay notas que quedan pendientes, cambia
133         // la voz a reproducir la ultima de esas voces
134         if (processorInfo.legatoInfo.legatoIsActive) {
135             if (!notesThatDidntEnd.isEmpty()) {
136                 int newerNote =
137                     notesThatDidntEnd.getLast();
138
139                     notesThatDidntEnd.removeLast();
140                     noteOn(midiChannel, newerNote, 1.0f);
141
142             }
143         }
144     }
145     else {
146         if (processorInfo.legatoInfo.legatoIsActive) {
147             notesThatDidntEnd.remove(
148                 notesThatDidntEnd.indexOf(midiNoteNumber));
149         }
150     }
151
152 }
153 }
154 }
155
156 void OcnetSynthesiser::noteOn(const int midiChannel, const int
157 midiNoteNumber, const float velocity)
158 {
159     const juce::ScopedLock sl(lock);
160
161     for (auto* sound : sounds)
162     {
163         if (sound->appliesToNote(midiNoteNumber) &&
164             sound->appliesToChannel(midiChannel))
165         {

```

```

165         if (processorInfo.legatoInfo.legatoIsActive) {
166             notesThatDidntEnd.add(midiNoteNumber);
167         }
168
169         // If hitting a note that's still ringing, stop it
170         // first (it could be
171         // still playing because of the sustain or sostenuto
172         // pedal).
173         for (auto* voice : voices)
174             if (voice->getCurrentlyPlayingNote() ==
175                 midiNoteNumber && voice->isPlayingChannel(midiChannel)) {
176                 notesThatDidntEnd.remove(
177                     notesThatDidntEnd.indexOf(midiNoteNumber));
178                 stopVoice(voice, 1.0f, true);
179             }
180
181     }
182 }
```

Listaxe A.1: Algoritmo de selección de voces.

A.2 Algoritmo da FFT

```

1 void fft(int N, std::unique_ptr<double[]>& ar,
2         std::unique_ptr<double[]>& ai)
3 /*
4  * in-place complex fft
5  *
6  * After Cooley, Lewis, and Welch; from Rabiner & Gold (1975)
7  *
8  * program adapted from FORTRAN
9  * by K. Steiglitz (ken@princeton.edu)
10 * Computer Science Dept.
11 * Princeton University 08544 */
12 {
13     int i, j, k, L; /* indexes */
14     int M, TEMP, LE, LE1, ip; /* M = log N */
15     int NV2, NM1;
16     double t; /* temp */
17     double Ur, Ui, Wr, Wi, Tr, Ti;
18     double Ur_old;
```

```

18
19 // if ((N > 1) && !(N & (N - 1))) // make sure we have a
20 // power of 2
21
22 NV2 = N >> 1;
23 NM1 = N - 1;
24 TEMP = N; /* get M = log N */
25 M = 0;
26 while (TEMP >= 1) ++M;
27
28 /* shuffle */
29 j = 1;
30 for (i = 1; i <= NM1; i++) {
31     if (i < j) { /* swap a[i] and a[j] */
32         t = ar[j - 1];
33         ar[j - 1] = ar[i - 1];
34         ar[i - 1] = t;
35         t = ai[j - 1];
36         ai[j - 1] = ai[i - 1];
37         ai[i - 1] = t;
38     }
39
40     k = NV2; /* bit-reversed counter */
41     while (k < j) {
42         j -= k;
43         k /= 2;
44     }
45
46     j += k;
47 }
48
49 LE = 1.;
50 for (L = 1; L <= M; L++) { // stage L
51     LE1 = LE; // (LE1 = LE/2)
52     LE *= 2; // (LE = 2^L)
53     Ur = 1.0;
54     Ui = 0.;
55     Wr = cos(M_PI / (float)LE1);
56     Wi = -sin(M_PI / (float)LE1); // Cooley, Lewis, and Welch
57     have "+" here
58     for (j = 1; j <= LE1; j++) {
59         for (i = j; i <= N; i += LE) { // butterfly
60             ip = i + LE1;
61             Tr = ar[ip - 1] * Ur - ai[ip - 1] * Ui;
62             Ti = ar[ip - 1] * Ui + ai[ip - 1] * Ur;
63             ar[ip - 1] = ar[i - 1] - Tr;

```

```

62         ai[ip - 1] = ai[i - 1] - Ti;
63         ar[i - 1] = ar[i - 1] + Tr;
64         ai[i - 1] = ai[i - 1] + Ti;
65     }
66     Ur_old = Ur;
67     Ur = Ur_old * Wr - Ui * Wi;
68     Ui = Ur_old * Wi + Ui * Wr;
69 }
70 }
71 }
```

Listaxe A.2: Algoritmo da FFT.

A.3 Script de instalación

```

1 [Setup]
2 AppId={{156ECD98-BFA3-4A04-94CE-8F9EB4920824}
3 Uninstallable=no
4 AppName=Ocnet
5 AppVersion=1.0
6 DefaultDirName={commoncf}\VST3
7 DefaultGroupName=Ocnet
8 OutputDir=..\\
9 ArchitecturesInstallIn64BitMode=x64
10 OutputBaseFilename=Ocnet_Installer
11 DirExistsWarning=no
12
13 [Files]
14 Source: ".\Ocnet.vst3"; DestDir: "{commoncf}\VST3"; Flags:
15 ignoreversion
16 Source: ".\Ocnet\*"; DestDir: "{userdocs}\Ocnet"; Flags:
17 ignoreversion createallsubdirs recursesubdirs
```

Listaxe A.3: Script de instalación.

A.4 Saída de PluginVal

Este é o resultado que ofrece PluginVal ao testar o plugin cun nivel de esixencia de 10.

```

1 pluginval v0.2.7 - JUCE v5.4.7
2 Started validating: C:\Program Files\Common Files\VST3\ocnet.vst3
3 Random seed: 0x3bb0d3a
4 Validation started: 9 Nov 2024 12:00:15am
5
```

```

6 Strictness level: 10
7 -----
8 Starting test: pluginval / Scan for known types: C:\Program
   Files\Common Files\VST3\ocnet.vst3...
9 Num types found: 1
10
11 Testing plugin: VST3-ocnet-12c38bea-3125a797
12 All tests completed successfully
13 -----
14 Starting test: pluginval / Open plugin (cold)...
15
16 Time taken to open plugin (cold): 375 ms
17 All tests completed successfully
18 -----
19 Starting test: pluginval / Open plugin (warm)...
20
21 Time taken to open plugin (warm): 179 ms
22 Running tests 1 times
23 All tests completed successfully
24 -----
25 Starting test: pluginval / Plugin info...
26
27 Plugin name: ocnet
28 Alternative names: ocnet
29 SupportsDoublePrecision: no
30 Reported latency: 0
31 Reported taillength: 0
32
33 Time taken to run test: 0
34 All tests completed successfully
35 -----
36 Starting test: pluginval / Plugin programs...
37 Num programs: 0
38 All program names checked
39
40 Time taken to run test: 0
41 All tests completed successfully
42 -----
43 Starting test: pluginval / Editor...
44
45 Time taken to open editor (cold): 270 ms
46 Time taken to open editor (warm): 242 ms
47
48 Time taken to run test: 645 ms
49 All tests completed successfully
50 -----

```

```

51 Starting test: pluginval / Open editor whilst processing...
52
53 Time taken to run test: 544 ms
54 All tests completed successfully
55 -----
56 Starting test: pluginval / Audio processing...
57 Testing with sample rate [44100] and block size [64]
58 Testing with sample rate [44100] and block size [128]
59 Testing with sample rate [44100] and block size [256]
60 Testing with sample rate [44100] and block size [512]
61 Testing with sample rate [44100] and block size [1024]
62 Testing with sample rate [48000] and block size [64]
63 Testing with sample rate [48000] and block size [128]
64 Testing with sample rate [48000] and block size [256]
65 Testing with sample rate [48000] and block size [512]
66 Testing with sample rate [48000] and block size [1024]
67 Testing with sample rate [96000] and block size [64]
68 Testing with sample rate [96000] and block size [128]
69 Testing with sample rate [96000] and block size [256]
70 Testing with sample rate [96000] and block size [512]
71 Testing with sample rate [96000] and block size [1024]
72
73 Time taken to run test: 146 ms
74 All tests completed successfully
75 -----
76 Starting test: pluginval / Plugin state...
77
78 Time taken to run test: 42 ms
79 All tests completed successfully
80 -----
81 Starting test: pluginval / Plugin state restoration...
82
83 Time taken to run test: 47 ms
84 All tests completed successfully
85 -----
86 Starting test: pluginval / Automation...
87 Testing with sample rate [44100] and block size [64] and sub-block
     size [32]
88 Testing with sample rate [44100] and block size [128] and sub-block
     size [32]
89 Testing with sample rate [44100] and block size [256] and sub-block
     size [32]
90 Testing with sample rate [44100] and block size [512] and sub-block
     size [32]
91 Testing with sample rate [44100] and block size [1024] and
     sub-block size [32]

```

```

92| Testing with sample rate [48000] and block size [64] and sub-block
93|   size [32]
94| Testing with sample rate [48000] and block size [128] and sub-block
95|   size [32]
96| Testing with sample rate [48000] and block size [256] and sub-block
97|   size [32]
98| Testing with sample rate [48000] and block size [512] and sub-block
99|   size [32]
100| Testing with sample rate [48000] and block size [1024] and
101|   sub-block size [32]
102|
103| Time taken to run test: 154 ms
104| All tests completed successfully
105| -----
106| Starting test: pluginval / Editor Automation...
107|
108| Time taken to run test: 18 secs
109| All tests completed successfully
110| -----
111| Starting test: pluginval / Automatable Parameters...
112|
113| Time taken to run test: 10 ms
114| All tests completed successfully
115| -----
116| Starting test: pluginval / Parameters...
117|
118| Time taken to run test: 9 ms
119| All tests completed successfully
120| -----
121| Starting test: pluginval / Background thread state...
122|
123| Time taken to run test: 2 secs
124| All tests completed successfully
125| -----
126| Starting test: pluginval / Parameter thread safety...
127|

```

```

128 Time taken to run test: 1 sec
129 All tests completed successfully
130 -----
131 Starting test: pluginval / Basic bus...
132 All tests completed successfully
133 -----
134 Starting test: pluginval / Listing available buses...
135 Inputs:
136   Named layouts: None
137   Discrete layouts: None
138 Outputs:
139   Named layouts: Stereo
140   Discrete layouts: None
141 Main bus num input channels: 0
142 Main bus num output channels: 2
143 All tests completed successfully
144 -----
145 Starting test: pluginval / Enabling all buses...
146 All tests completed successfully
147 -----
148 Starting test: pluginval / Disabling non-main busses...
149 All tests completed successfully
150 -----
151 Starting test: pluginval / Restoring default layout...
152 Main bus num input channels: 0
153 Main bus num output channels: 2
154
155 Time taken to run test: 5 ms
156 All tests completed successfully
157 -----
158 Starting test: pluginval / Editor stress...
159 Testing opening Editor with released processing
160 Testing opening Editor with 0 sample rate and block size
161
162 Time taken to run test: 817 ms
163 All tests completed successfully
164 -----
165 Starting test: pluginval / Editor DPI Awareness...
166 Testing opening Editor with DPI Awareness disabled
167
168 Time taken to run test: 402 ms
169 All tests completed successfully
170 -----
171 Starting test: pluginval / Allocations during process...
172 Testing with sample rate [44100] and block size [64]
173 Testing with sample rate [44100] and block size [128]

```

```

174 Testing with sample rate [44100] and block size [256]
175 Testing with sample rate [44100] and block size [512]
176 Testing with sample rate [44100] and block size [1024]
177 Testing with sample rate [48000] and block size [64]
178 Testing with sample rate [48000] and block size [128]
179 Testing with sample rate [48000] and block size [256]
180 Testing with sample rate [48000] and block size [512]
181 Testing with sample rate [48000] and block size [1024]
182 Testing with sample rate [96000] and block size [64]
183 Testing with sample rate [96000] and block size [128]
184 Testing with sample rate [96000] and block size [256]
185 Testing with sample rate [96000] and block size [512]
186 Testing with sample rate [96000] and block size [1024]
187
188 Time taken to run test: 100 ms
189 All tests completed successfully
190 -----
191 Starting test: pluginval / Process called with a larger than
   prepared block size...
192 INFO: Skipping test for plugin format
193
194 Time taken to run test: 0
195 All tests completed successfully
196 -----
197 Starting test: pluginval / Fuzz parameters...
198
199 Time taken to run test: 13 ms
200
201 Time taken to run all tests: 26 secs
202 All tests completed successfully
203
204 Finished validating: C:\Program Files\Common Files\VST3\ocnet.vst3
205 ALL TESTS PASSED

```

Listaxe A.4: Saída de PluginVal.

A.5 Detección de fugas de memoria

```

1 Dumping objects ->
2 {372833} normal block at 0x000002452329F210, 3 bytes long.
3 Data: <en > 65 6E 00
4 {372832} normal block at 0x00000245236DDA00, 16 bytes long.
5 Data: < i E      #E    > 90 00 69 1C 45 02 00 00 10 F2 29 23 45 02
   00 00
6 {145122} normal block at 0x0000024523370360, 6 bytes long.

```

```

7 Data: <en-gb > 65 6E 2D 67 62 00
8 {145121} normal block at 0x000002452336F730, 16 bytes long.
9 Data: < A E ` 7#E > C0 41 0E 1D 45 02 00 00 60 03 37 23 45 02
   00 00
10 {141291} normal block at 0x000002451CFD58B0, 2 bytes long.
11 Data: <c > 63 00
12 {141290} normal block at 0x000002451D0E41C0, 16 bytes long.
13 Data: < X E > 00 00 00 00 00 00 00 00 B0 58 FD 1C 45 02
   00 00
14 {141287} normal block at 0x000002451C6873C0, 216 bytes long.
15 Data: < > 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00
16 {141267} normal block at 0x000002451CF59790, 184 bytes long.
17 Data: < > 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00
18 {37619} normal block at 0x000002451C8940A0, 2 bytes long.
19 Data: <c > 63 00
20 {37618} normal block at 0x000002451C690090, 16 bytes long.
21 Data: <p i E @ E > 70 02 69 1C 45 02 00 00 A0 40 89 1C 45 02
   00 00
22 {28925} normal block at 0x000002451C690900, 6 bytes long.
23 Data: <en-gb > 65 6E 2D 67 62 00
24 {28924} normal block at 0x000002451C690270, 16 bytes long.
25 Data: < i E > 00 00 00 00 00 00 00 00 00 09 69 1C 45 02
   00 00
26 {28923} normal block at 0x000002451C687BA0, 216 bytes long.
27 Data: < > 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00
28 {28871} normal block at 0x000002451CF5AE90, 184 bytes long.
29 Data: < > 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00
30 Object dump complete.

```

Listaxe A.5: Fugas de memoria detectadas polo de depurador de Visual Studio. Todas elas proceden dun *bug* da libraria de JUCE.

Relación de Acrónimos

ADC analog-to-digital converter 5

ADSR attack, decay, sustain, release 46

BPM beats per minute 47

CPU central processing unit 31, 68

DAC digital-to-analog converter 5

DAW digital audio workstation iv, 2, 3, 14, 17, 21, 31, 47, 49, 66, 68

FFT fast Fourier transform 35, 39

FIR finite impulse response 53

FM frequency modulation 3, 14, 42, 43

GUI graphical user interface 20

IDE integrated development environment 17

IIR infinite impulse response 53, 54, 56

LFO low-frequency oscillator 15, 47, 48

MIDI musical instrument digital interface 31

MVC Model-View-Controller 19, 20

UML unified modeling language 20

VST Visual Studio Technology 2, 28, 45

XML extensible markup language 33

Glosario

enerxía A enerxía dunha sinal é definida como a suma dos cadrados das súas amplitudes:

$$\text{Enerxía} = \sum_{i=1}^N \|x_i\|^2 \quad 58$$

highend Frecuencias pertencentes á parte máis alta do espectro, tipicamente superiores a 10 kHz. [37](#)

latencia Tempo que tarda en procesarse un bloque de audio. [31](#)

playhead Marcador que indica a posición actual de reproducción. [31](#)

Bibliografía

- [1] R. Setiawan and J. A. R. Hakim, “Embedded FFT with dsPIC30F4013,” *JAREE-Journal on Advanced Research in Electrical Engineering*, vol. 1, no. 1, pp. 14–20, 2017. [En línea]. Disponible en: <http://jaree.its.ac.id/index.php/jaree/article/viewFile/12/3>
- [2] Image-Line, “FL Studio,” 2024, consultado o 14 de novembro de 2024. [En línea]. Disponible en: <https://www.image-line.com/fl-studio/>
- [3] Vital, “Vital,” 2024, consultado o 14 de novembro de 2024. [En línea]. Disponible en: <https://vital.audio/>
- [4] Phaseplant, “Phaseplant,” 2024, consultado o 14 de novembro de 2024. [En línea]. Disponible en: https://kilohearts.com/products/phase_plant
- [5] Ableton, “Ableton,” 2024, consultado o 14 de novembro de 2024. [En línea]. Disponible en: <https://www.ableton.com/>
- [6] Audio Interfacing, “What is dB SPL?” 2023, consultado o 14 de novembro de 2024. [En línea]. Disponible en: <https://audiostreaming.com/what-is-dbspl/>
- [7] Glassdoor, “Glassdoor,” 2024, consultado o 14 de novembro de 2024. [En línea]. Disponible en: <https://www.glassdoor.es/>
- [8] Jobted, “Jobted,” 2024, consultado o 14 de novembro de 2024. [En línea]. Disponible en: <https://www.jobted.es/>
- [9] Serum, “Serum,” 2024, consultado o 14 de novembro de 2024. [En línea]. Disponible en: <https://xferrecords.com/products/serum/>
- [10] Logic, “Logic,” 2024, consultado o 14 de novembro de 2024. [En línea]. Disponible en: <https://www.apple.com/logic-pro/>
- [11] “IPlug2,” consultado o 14 de novembro de 2024. [En línea]. Disponible en: <https://iplug2.github.io/>

- [12] Raw Material Software Limited, “JUCE: Cross-platform audio application and plug-in framework,” 2023, consultado o 14 de novembro de 2024. [En liña]. Dispoñible en: <https://juce.com/>
- [13] “Audiokit,” consultado o 14 de novembro de 2024. [En liña]. Dispoñible en: <https://audiokitpro.com/>
- [14] “Dplug,” consultado o 14 de novembro de 2024. [En liña]. Dispoñible en: <https://dplug.org/>
- [15] F. S. Foundation, “Gnu afferro general public license v3,” consultado o 14 de novembro de 2024. [En liña]. Dispoñible en: <https://www.gnu.org/licenses/agpl-3.0.en.html>
- [16] Microsoft, “Visual studio 2022,” consultado o 14 de novembro de 2024. [En liña]. Dispoñible en: <https://visualstudio.microsoft.com/>
- [17] J. V. Rodríguez, “Repositorio de github do TFG,” consultado o 14 de novembro de 2024. [En liña]. Dispoñible en: <https://github.com/JuanVillaverdeRodriguez/ocnet>
- [18] Geogebra, “Geogebra,” consultado o 14 de novembro de 2024. [En liña]. Dispoñible en: <https://www.geogebra.org/>
- [19] Overleaf, “Overleaf,” consultado o 14 de novembro de 2024. [En liña]. Dispoñible en: <https://www.overleaf.com/>
- [20] Python, “Python,” consultado o 14 de novembro de 2024. [En liña]. Dispoñible en: <https://www.python.org/>
- [21] Balsamiq, “Balsamiq,” 2024, consultado o 14 de novembro de 2024. [En liña]. Dispoñible en: <https://balsamiq.com/>
- [22] Draw.io, “Draw.io,” consultado o 14 de novembro de 2024. [En liña]. Dispoñible en: <https://www.draw.io/>
- [23] Discord, “Discord,” consultado o 14 de novembro de 2024. [En liña]. Dispoñible en: <https://discord.com/>
- [24] J. W. Cooley and J. W. Tukey, “An algorithm for the machine calculation of complex fourier series,” *Mathematics of Computation*, vol. 19, pp. 297–301, 1965. [En liña]. Dispoñible en: <https://api.semanticscholar.org/CorpusID:121744946>
- [25] T. Trick, “Theory and application of digital signal processing,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 23, no. 4, pp. 394–395, 1975.

- [26] Inno Setup, “Inno setup,” 2024, consultado o 14 de novembro de 2024. [En liña]. Dispoñible en: <https://jrsoftware.org/isinfo.php>
- [27] S. Audio, “Let’s write a reverb,” 2021, consultado o 14 de novembro de 2024. [En liña]. Dispoñible en: <https://signalsmith-audio.co.uk/writing/2021/lets-write-a-reverb/#putting-it-together-examples>
- [28] ——, “reverb-example-code,” 2021, consultado o 14 de novembro de 2024. [En liña]. Dispoñible en: <https://github.com/Signalsmith-Audio/reverb-example-code/tree/main>
- [29] ——, “DSP,” 2022, consultado o 14 de novembro de 2024. [En liña]. Dispoñible en: <https://github.com/Signalsmith-Audio/dsp>
- [30] Tracktion, “PluginVal,” 2024, consultado o 14 de novembro de 2024. [En liña]. Dispoñible en: <https://www.tracktion.com/>
- [31] consultado o 14 de novembro de 2024. [En liña]. Dispoñible en: <https://github.com/juce-framework/JUCE/issues/1391>
- [32] Didier Dambrin, “Wave candy. fl studio plugin,” consultado o 14 de novembro de 2024. [En liña]. Dispoñible en: <https://www.image-line.com/fl-studio-learning/fl-studio-online-manual/html/plugins/Wave%20Candy.htm>
- [33] A. Vaneev, V. Stolypko, and S. Kane, “Voxengo span,” consultado o 14 de novembro de 2024. [En liña]. Dispoñible en: <https://www.voxengo.com/product/span>