



Escuela Técnica Superior de Ingeniería Informática

Ingeniería Informática – Tecnologías Informáticas

## **TRABAJO FIN DE GRADO**

### **Cosmic Whispers - Programación creativa aplicada a la creación de música generativa**

Autor:

Hugo Villanueva Duque

Tutor:

Víctor Jesús Díaz Madrigal

Departamento:

Lenguajes y Sistemas Informáticos

Primera Convocatoria

Curso 2022/2023

# Índice

Resumen	4
Introducción	5
1. Diseño teórico	6
1.1. Sintetizador	6
1.2. Notas musicales	12
1.3. Generadores	13
1.4. Escalas	14
1.5. Theremin	18
2. Proceso de desarrollo	19
3. Tecnologías utilizadas	22
3.1. Javascript	22
3.1.1. P5.js	23
3.1.2. Tone.js	36
3.1.3. dat.GUI	40
3.2. HTML + CSS	41
4. Implementación	42
4.1. Archivo principal	44
4.2. Draw	45
Dibujo y lógica del Theremin	48
Metrónomo	50
4.3. Sound	52
4.4. whiteDots	58
4.5. OctaveGenerators	64
4.6. Background interactivo	68
4.7. Sound Design Panel	72
4.8. Graphic User Interace	79
4.9. Archivos HTML + CSS	85
4.10. Dependencias entre los archivos	87
Futuras implementaciones	88
5. Conclusiones y próximos pasos	89
Bibliografía	90
Conceptos teóricos	90
Documentación	90
Librerías	90

Anexo I: Código completo	91
Index.html	91
soundDesign.html	92
Style.css	93
Main.js	96
Draw.js	97
Sound.js	102
whiteDot.js	108
octaveGenerator.js	112
Retrowave.js	114
soundDesign.js	117
Gui.js	124

## Resumen

El presente Trabajo de Fin de Grado presenta el proyecto Cosmic Whispers, una plataforma de programación creativa que combina librerías de Javascript de animación gráfica (p5) y sintetización musical (Tone.js) para crear un sandbox musical único. El objetivo principal de este proyecto es inspirar a músicos y entusiastas del sonido a explorar nuevas posibilidades y crear patrones musicales innovadores.

Cosmic Whispers permite generar música de forma automática y aleatoria, basándose en algoritmos y reglas de movimiento intuitivas y visibles en pantalla, además de control de variables y escalas para permitir al usuario crear las atmósferas que desee, gracias a la librería dat.gui. Todo esto, además con un diseño estético agradable y artístico para explotar las capacidades de las librerías.

El proyecto es de código abierto y está disponible en:

<https://github.com/hugvildug/Cosmic-Whispers>

## Introducción

En la era digital, la fusión de la tecnología y la creatividad es cada vez más evidente y fundamental en diversas disciplinas. La música, siendo una de las formas más expresivas de arte, no es una excepción a este fenómeno. Este proyecto busca navegar en la intersección de estas dos corrientes, la música y la programación, utilizando las bibliotecas p5.js y Tone.js para crear una plataforma de creación musical interactiva y accesible desde el navegador.

Es importante destacar que, al comenzar este proyecto, no tenía conocimientos previos de las librerías p5.js ni Tone.js. Sin embargo, a través de una dedicada investigación y aprendizaje autodidacta, adquirí las habilidades necesarias para utilizar estas herramientas de manera efectiva y lograr la integración exitosa de ambas en el proyecto.

La inspiración de este proyecto radica en explorar y explotar las capacidades de la programación creativa para transformar la forma en que generamos y experimentamos la música. El objetivo principal es diseñar una herramienta que no solo permita la creación de patrones musicales únicos, sino que también visualice la música de una manera dinámica y estéticamente atractiva. La aplicación permite a los usuarios crear nodos musicales alrededor de generadores orbitales que alteran su comportamiento rítmico en función de su posición, proporcionando una interfaz de usuario intuitiva y una experiencia musical única.

Este proyecto aporta una nueva perspectiva a la relación entre la programación y la música, poniendo el foco en la interactividad, la exploración y la generación de ritmos musicales en tiempo real. Se espera que esta herramienta, además de ser un entorno de experimentación para músicos y artistas, pueda servir como un recurso pedagógico para quienes buscan adentrarse en el fascinante mundo de la programación creativa, ya que demuestra las capacidades de los lenguajes utilizados.

## 1. Diseño teórico

El objetivo principal de Cosmic Whispers consiste ofrecer al usuario un lienzo vacío donde pudiera ir añadiendo notas, representadas mediante círculos móviles. Estos círculos, a los que referiremos como “nodos”, se mueven por el espacio y según su posición con respecto a otros nodos suenan en un momento u otro de una secuencia temporal de 8 compases.

El proyecto Cosmic Whispers surge como una necesidad de explorar las capacidades expresivas de una escala musical, creando las notas y haciendo que se reproduzcan en momentos impredecibles, para crear patrones rítmicos y musicales de manera pseudo-aleatoria. De esta forma, un músico puede evitar caer en sus tendencias repetitivas de composición y recibir inspiración de nuevos patrones.

Para explicar cómo se crea una aplicación de estas características, vamos a explicar cómo funciona la teoría musical detrás de ella.

### 1.1. Sintetizador

Un sintetizador es un instrumento musical electrónico que genera sonidos mediante corrientes eléctricas. Para saber cómo funciona uno, primero hay que entender cómo se genera un sonido.

El sonido se crea cuando una fuente, como una cuerda de guitarra, una membrana de tambor o las cuerdas vocales, se pone en movimiento y comienza a vibrar. Estas vibraciones generan ondas de presión en el medio circundante, ya sea el aire, el agua o un objeto sólido.

Una vibración no es más que la representación de las partículas circundantes moviéndose hacia un lado y hacia el otro. Esto se suele representar mediante una onda.

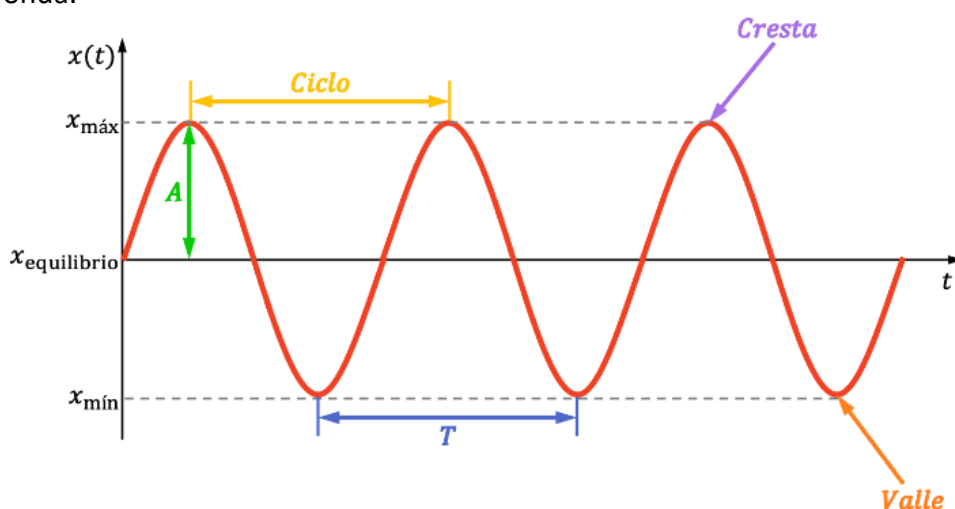


Ilustración 1 - Representación de una onda

Lo que más nos interesa para producir una nota es la frecuencia de la onda, esto es, cuántas veces en un segundo se repite la onda (ciclo). Esta frecuencia la medimos mediante los Hercios (Hz) y es la inversa del período, el tiempo que tarda en repetirse la onda (T).

Según el valor de la frecuencia, distinguimos 3 rangos principales:

- Frecuencias bajas (20 Hz - 250 Hz): Las frecuencias bajas corresponden a sonidos graves. Son percibidas como tonos profundos y de gran resonancia. Por ejemplo, los graves de un bajo eléctrico o el sonido de un bombo de una batería. Estas frecuencias a menudo se sienten más que se escuchan, ya que pueden producir vibraciones físicas en el cuerpo.
- Frecuencias medias (250 Hz - 4000 Hz): Las frecuencias medias son importantes para la inteligibilidad del habla y para resaltar la calidad de los instrumentos melódicos y armónicos. Esta gama abarca las voces humanas, la mayoría de los instrumentos de cuerda y algunos instrumentos de viento y percusión. Las frecuencias medias contribuyen a la claridad y la definición en la música.
- Frecuencias altas (4000 Hz - 20 000 Hz): Las frecuencias altas corresponden a los sonidos agudos. Son percibidas como tonos brillantes y agudos. Ejemplos de sonidos agudos incluyen los platillos de una batería, los instrumentos de cuerda como el violín, y los sonidos de campanas o silbidos. Estas frecuencias añaden brillo y claridad a la música y son importantes para la percepción de los detalles sonoros.

Del mismo modo, cada nota tiene asociada una frecuencia, un tono determinado. Según el sistema de afinación que se elija, la relación entre las notas es la misma, pero se empieza a contar desde una frecuencia u otra. En el sistema de afinación que se utiliza en la música occidental, estandarizada desde el siglo XIX, cada nota tiene una frecuencia asociada partiendo desde el La (A) a 440Hz:

<b>Equal-tempered</b>	
<i>Note</i>	<i>Hz</i>
C	523.25
B	493.88
B <sup>b</sup> /A <sup>#</sup>	466.16
A	440.00
A <sup>b</sup> /G <sup>#</sup>	415.30
G	392.00
G <sup>b</sup> /F <sup>#</sup>	369.99
F	349.23
E	329.62
E <sup>b</sup> /D <sup>#</sup>	311.13
D	293.66
D <sup>b</sup> /C <sup>#</sup>	277.18
C	261.63

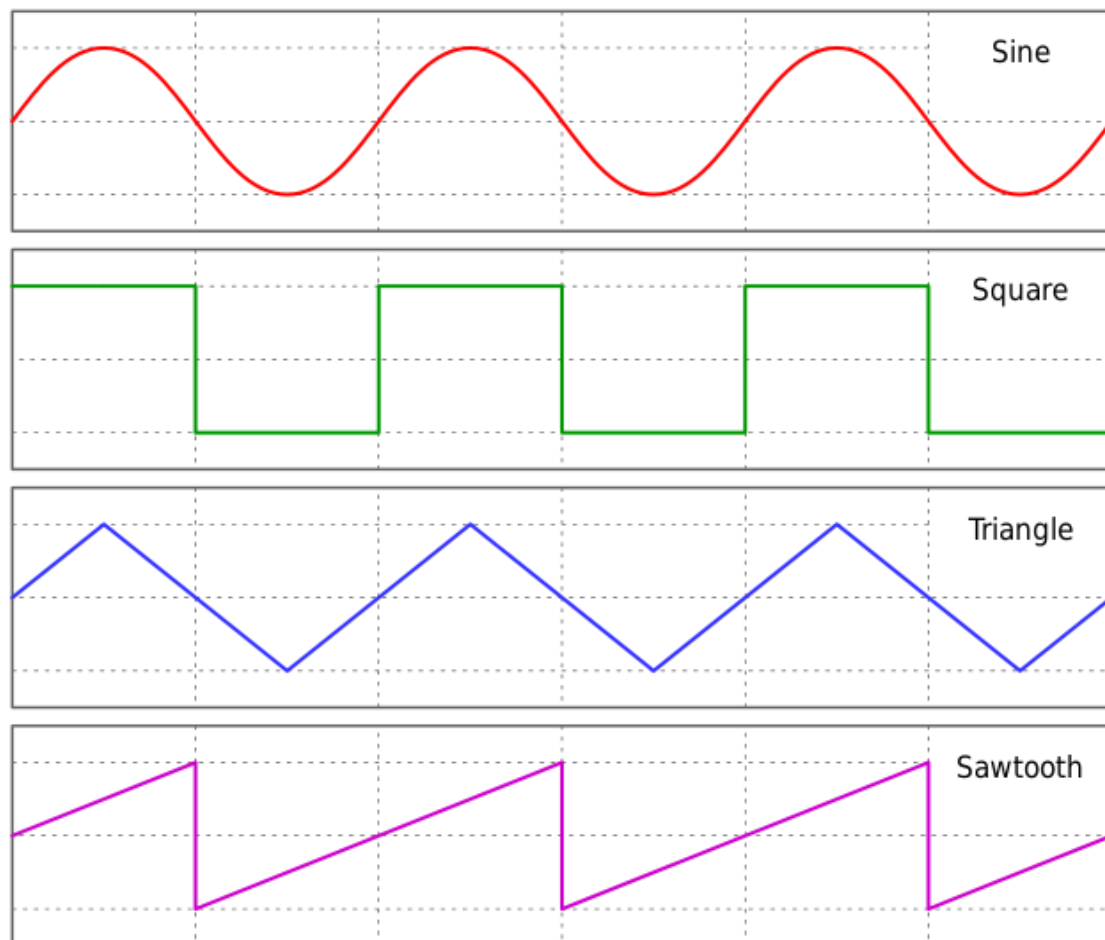
*Ilustración-2: Afinación de las notas según su frecuencia. Sistema occidental*

Una vez sabido esto, entonces si queremos crear un sintetizador, necesitaremos un generador de ondas que pueda oscilar a las frecuencias indicadas por la tabla anterior.

Pero eso es solamente para conseguir el tono del sonido, de la forma más básica posible. Si queremos cambiar el timbre del sonido, ¿cómo lo hacemos?

Una vez que tenemos un generador de ondas capaz de oscilar a diferentes frecuencias, podemos modificar el timbre del sonido combinando varias ondas diferentes. Esto se logra mediante la síntesis de sonido y el uso de diferentes formas de onda.

Las formas de onda básicas utilizadas en la síntesis de sonido son:



*Ilustración 1 Formas de onda básicas utilizadas en sintetizadores*



- Onda senoidal (Sine wave):

La onda senoidal es la forma más pura y simple de onda. Tiene un sonido suave y puro, compuesto por una sola frecuencia sin armónicos adicionales. Es la base para crear otros tipos de ondas y se utiliza comúnmente en sonidos de flautas, cuerdas y sonidos suaves en general.

- Onda triangular (Triangle wave):

La onda triangular tiene una forma triangular y está compuesta por una serie de armónicos impares. Su sonido es más brillante que la onda senoidal, pero aún suave en comparación con otras formas de onda. Se utiliza en instrumentos como pianos y órganos.

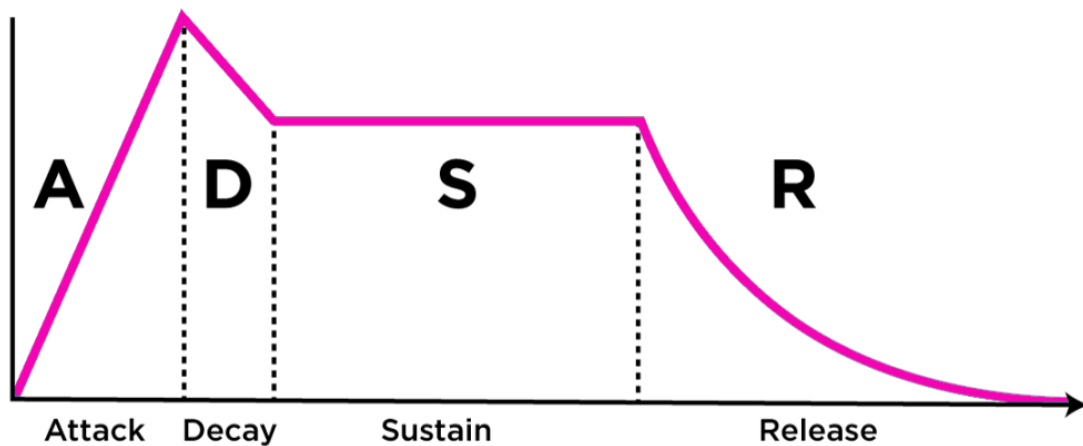
- Onda cuadrada (Square wave):

La onda cuadrada tiene una forma rectangular y está compuesta por armónicos impares y sucesivas caídas abruptas. Tiene un sonido más brillante y rico en armónicos que la onda triangular. Se utiliza en instrumentos como el clavecín, sintetizadores de bajo y sonidos de efectos especiales.

- Onda diente de sierra (Sawtooth wave):

La onda diente de sierra tiene una forma similar a los dientes de una sierra. Contiene todos los armónicos y su sonido es brillante y potente. Se utiliza en instrumentos como guitarras eléctricas, sintetizadores de lead y sonidos de sintetizadores en general.

Junto con las formas de onda, otro elemento importante para modificar el timbre del sonido en un sintetizador es la curva ADSR (Attack, Decay, Sustain, Release).



*Ilustración 2: Curva ADSR de un sonido*

ADSR es un sistema de envolvente que permite controlar cómo se desarrolla el sonido a lo largo del tiempo. Esto quiere decir que funciona como un filtro que se coloca sobre la onda pura seleccionada (sine, square, triangle, sawtooth) para que varíe su volumen durante el tiempo. Cada una de las etapas del ADSR tiene un efecto particular en la forma en que percibimos el sonido. Veamos cada una de ellas:

- **Ataque (Attack):**

El ataque determina cómo se inicia el sonido desde el momento en que se presiona una tecla o se activa una nota. Controla la rapidez con la que el sonido alcanza su nivel máximo de amplitud. Un ataque más rápido crea un sonido más percusivo, mientras que un ataque más lento da lugar a un inicio gradual y suave del sonido.

- **Decaimiento (Decay):**

El decaimiento determina cómo disminuye la amplitud del sonido después del ataque. Controla la velocidad a la que el sonido pasa del nivel máximo al nivel de sostenimiento. Un decaimiento más rápido produce un sonido más corto y puntiagudo, mientras que un decaimiento más lento da lugar a un sonido más prolongado y sostenido.

- **Sostenimiento (Sustain):**

El sostenimiento controla el nivel de amplitud que se mantiene después de la etapa de decaimiento. Permite mantener un nivel constante de sonido mientras se mantenga presionada la tecla o se mantenga activada la nota. El ajuste del nivel de sostenimiento determina la duración y el volumen del sonido sostenido.

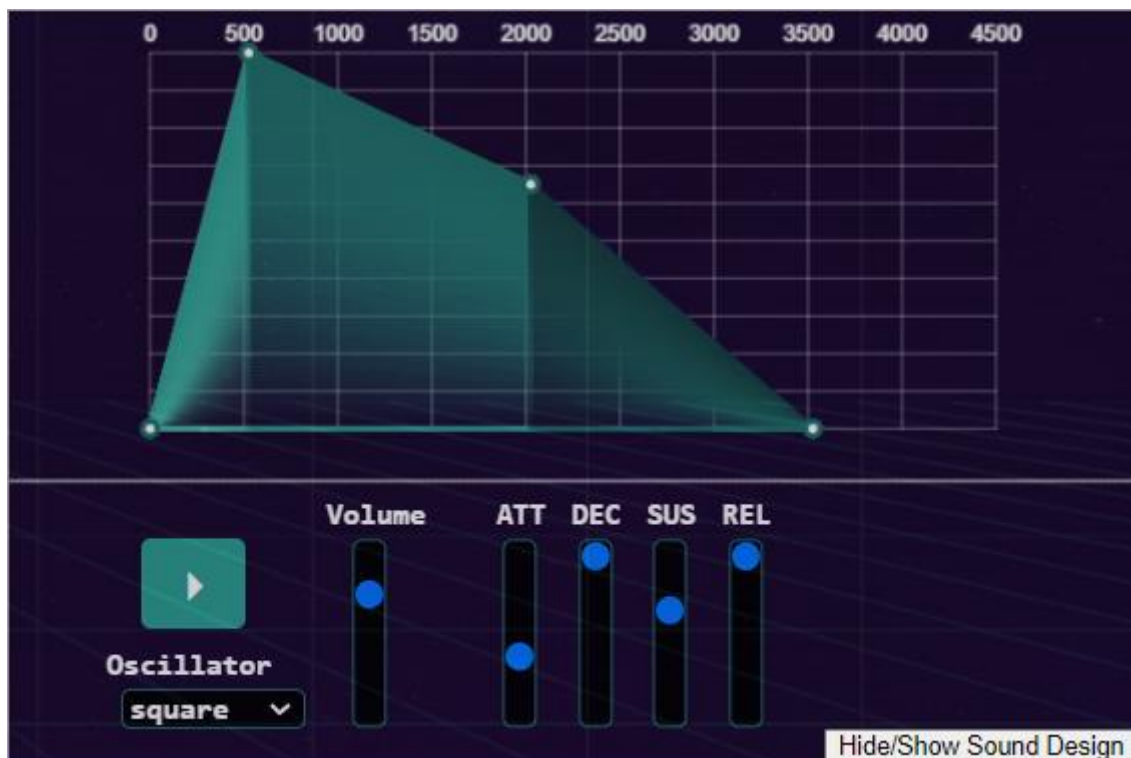
- **Liberación (Release):**

La liberación determina cómo disminuye gradualmente la amplitud del sonido después de soltar la tecla o finalizar la nota. Controla la rapidez con la que el sonido se desvanece después de la liberación. Un tiempo de liberación más rápido produce un

desvanecimiento más abrupto, mientras que un tiempo de liberación más lento da lugar a un desvanecimiento más gradual y prolongado.

El control de estas etapas de envolvente ADSR permite moldear el sonido de una manera más expresiva. Por ejemplo, ajustar un ataque rápido y un decaimiento corto puede crear sonidos percusivos como tambores, mientras que un ataque lento y un decaimiento largo pueden generar sonidos más suaves y envolventes.

Este conocimiento nos va a ser útil para poder diseñar nuestros propios sonidos como queremos que suenen, puesto que en la aplicación se dispone de un panel de control como este para modificar el sonido a gusto del usuario:



*Ilustración 3. Curva ADSR implementada en Cosmic Whispers*

## 1.2. Notas musicales

Ahora que ya sabemos cómo crear una nota, vamos a ver cómo funcionan las notas a nivel teórico musical para crear música. Para ello, primero tenemos que entender cómo funcionan las notas musicales en un instrumento. En un piano, para reproducir una nota, pulsamos su tecla asociada:

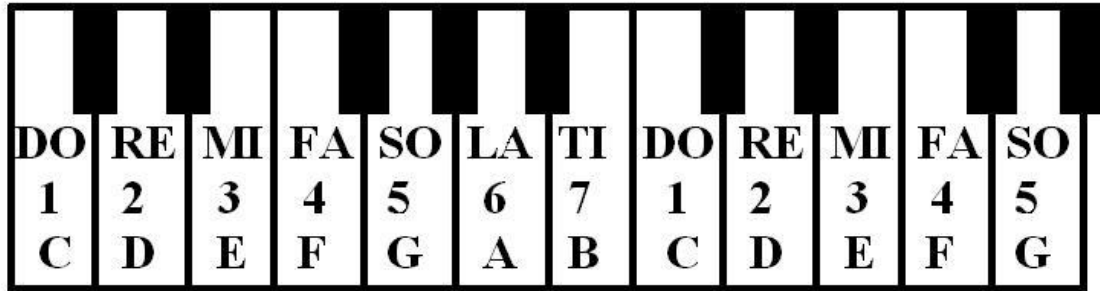


Ilustración 4. Notas musicales en el piano

En Cosmic Whispers, en vez de pulsar una tecla y que suene su sonido, vamos a seleccionar una nota, y creamos su nodo asociado en el lienzo;

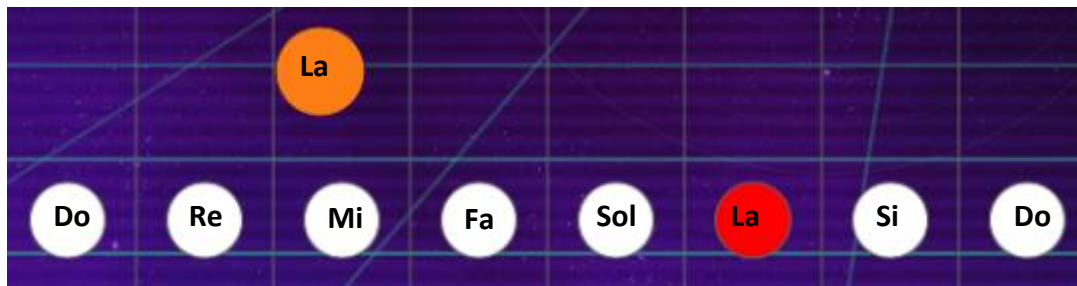


Ilustración 5. Notas musicales en Cosmic Whispers

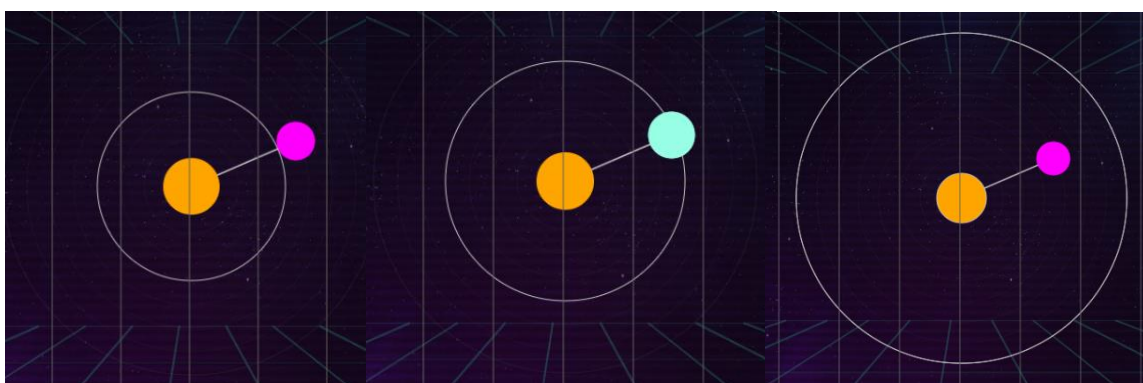
Podemos ir cambiando de nota seleccionada para crear todas las que se deseen. Estas notas creadas serán reproducidas cuando haya generadores para alrededor. Para ello, explicaremos lo que es un generador en el siguiente epígrafe.

### 1.3. Generadores

Si uno se fija en cualquier pieza musical actual, notará que hay patrones que se repiten continuamente, ya sea en la melodía, en el acompañamiento o en la percusión. Esto se debe a que la música actualmente se diseña en software que invita a la reutilización de bloques con los motivos musicales creados por el usuario.

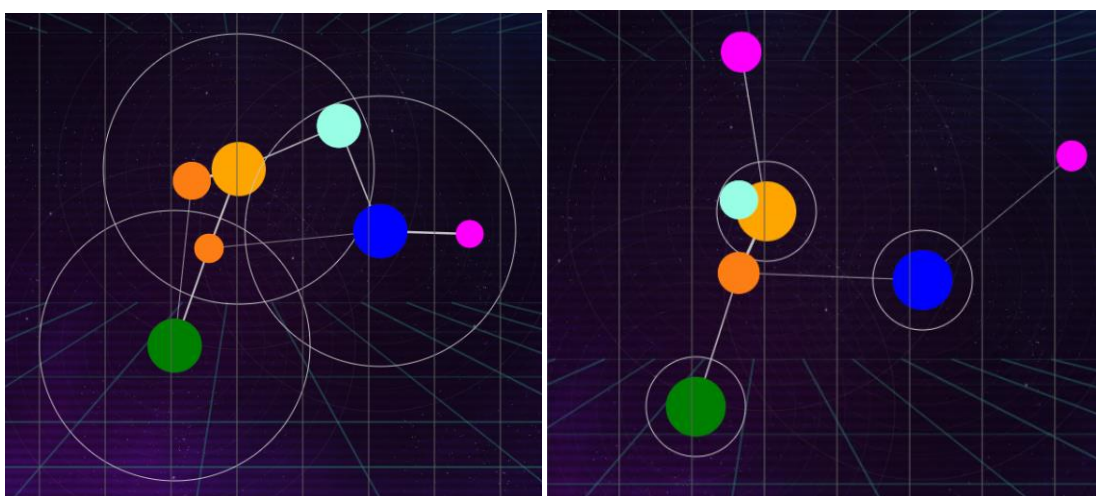
En este proyecto, no se van a repetir bloques musicales, pero en su lugar, se van a crear generadores que se repiten en cada compás (en música, cada unidad de tiempo), y que van a reproducir las notas que se encuentren en su vecindad.

Cada generador tendrá una órbita que va creciendo durante cada compás, y cada vez que la órbita pase por encima de un nodo, este reproducirá su nota asociada.



*Ilustración 6. Generador realizando un ciclo: Antes, durante, y después de reproducir una nota en su órbita*

Esta simple mecánica se vuelve mucho más interesante al añadir múltiples nodos, múltiples generadores y movimiento a los elementos. De esta forma se crean patrones únicos con un efecto visual caótico.



*Ilustración 7. Múltiples generadores, con notas moviéndose alrededor: Antes y después*

Lo que será el resultado que buscábamos. Las notas se irán moviendo por el lienzo, creando nuevas configuraciones espaciales, interactuando con los generadores de maneras únicas y se crearán patrones nuevos de forma ininterrumpida.

De esta manera podemos explorar infinidad de patrones rítmicos aleatorios, así como las combinaciones entre las notas de la escala. Pero podemos añadir otra capa esencial para la creación generativa: Improvisar dentro de una escala.

## 1.4. Escalas

Hasta ahora, solo hemos visto un ejemplo de notas creadas dentro de la escala do-re-mi-fa-sol, pero eso es tan solo la creación dentro de una escala.

Una escala se define como la sucesión de las notas ordenadas de la más grave a la más aguda. Toda pieza musical se mueve dentro de una escala, y esta determina fuertemente el carácter expresivo de la melodía. Las más comúnmente conocidas son la escala mayor, por crear melodías “alegres” y la escala menor, por crear melodías “tristes”. La realidad es que hay cientos de escalas más allá de estas dos, cada una con su propio perfil, y algunas de ellas son idóneas para crear melodías generativas, puesto que contienen menos disonancias entre sí.

Las escalas que se han escogido en este proyecto, se pueden representar fácilmente en una lista numérica, donde un incremento de 1 simboliza un incremento de semitono. Esto, en el piano, significa desplazarse una tecla a la derecha, incluyendo las teclas negras. Partiendo de que la mayor es:

Do	Re	Mi	Fa	Sol	La	Si	Do
0	2	4	5	7	9	11	12

Podemos incrementarla a una escala doble, es decir, con dos octavas, simplemente siguiendo la secuencia numérica de 0 a 12, pero empezando en 12:

Do	Re	Mi	Fa	Sol	La	Si	Do	Re'	Mi'	Fa'	Sol'	La'	Si'	Do'
0	2	4	5	7	9	11	12	14	16	17	19	21	23	24

Aplicando la misma lógica al resto de escalas que vamos a implementar, nos quedaría así:

```
const scales = {
  Major: [0, 2, 4, 5, 7, 9, 11, 12, 14, 16, 17, 19, 21, 23],
  Natural_Minor: [0, 2, 3, 5, 7, 8, 10, 12, 14, 15, 17, 19, 20, 22],
  Harmonic_Minor: [0, 2, 3, 5, 7, 8, 11, 12, 14, 15, 17, 19, 20, 23],
  Melodic_Minor: [0, 2, 3, 5, 7, 9, 11, 12, 14, 15, 17, 19, 21, 23],
  Mixolydian: [0, 2, 4, 5, 7, 9, 10, 12, 14, 16, 17, 19, 21, 22],
  Phrygian: [0, 1, 3, 5, 7, 8, 10, 12, 13, 15, 17, 19, 20, 22],
  Lydian: [0, 2, 4, 6, 7, 9, 11, 12, 14, 16, 18, 19, 21, 23],
  Locrian: [0, 1, 3, 5, 6, 8, 10, 12, 13, 15, 17, 18, 20, 22],
  Mongolian: [0, 2, 5, 7, 9, 12, 14, 17, 19, 21, 24, 26, 29, 31],
  Pentatonic_Major: [0, 2, 4, 7, 9, 12, 14, 16, 19, 21, 24, 26, 28, 31],
  Pentatonic_Minor: [0, 3, 5, 7, 10, 12, 15, 17, 19, 22, 24, 27, 29, 31],
  Overtone: [0, 4, 7, 10, 12, 14, 16, 19, 22, 24, 26, 28, 31, 34, 36, 38,
40, 43, 46, 48, 50],
  Blues: [0, 3, 5, 6, 7, 10, 12, 15, 17, 18, 19, 22, 24, 27],
  Chromatic: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31],
  Whole_Tone: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28,
30],
  Random: getRandomIntervals(),};
```

Cada escala tiene su uso y perfil sonoro asociado, veamos para qué son cada una:

- **Mayor:** La escala mayor es una de las más utilizadas en la música occidental. Tiene un sonido alegre y enérgico, y se asocia con sentimientos de felicidad, alegría y triunfo. Es versátil y se adapta a una amplia variedad de géneros musicales.
- **Natural Minor:** La escala menor natural tiene un sonido más melancólico y triste en comparación con la escala mayor. Es muy común en música clásica, pop, rock y blues. Puede evocar emociones como la melancolía, la introspección y la nostalgia.
- **Menor armónica:** La escala menor armónica tiene un carácter distintivo debido a su séptima aumentada. Tiene un sonido oscuro y misterioso, y se utiliza ampliamente en géneros como el metal, el jazz y la música flamenca. Esta escala puede evocar sentimientos de tensión, pasión y dramatismo.
- **Menor melódica:** La escala menor melódica es similar a la escala menor natural, pero con la sexta y séptima notas ascendiendo medio tono cuando se toca hacia arriba. Esto le da un sonido más exótico y emotivo. Se utiliza en géneros como el jazz, el fusion y la música oriental. Puede transmitir tanto emociones oscuras como luminosas y exóticas.

- **Mixolidio:** La escala mixolidia es una escala mayor con la séptima nota disminuida medio tono. Tiene un sonido festivo, alegre y en ocasiones "bluesy". Se utiliza comúnmente en el jazz, el rock y la música latina. Tiene un carácter juguetón y despreocupado.
- **Frigia:** La escala frigia tiene un carácter exótico y misterioso debido a su segunda nota disminuida medio tono. Se utiliza a menudo en música flamenca, metal y rock progresivo. Puede evocar sentimientos de pasión, tensión y emociones intensas.
- **Lidia:** La escala lidia tiene un sonido brillante y luminoso debido a su cuarta nota aumentada medio tono. Se utiliza en géneros como el jazz y el rock. Puede transmitir una sensación de elevación, grandeza y optimismo.
- **Locrio:** La escala locria tiene un sonido inestable y tenso debido a su quinta y séptima notas disminuidas medio tono. Es una escala poco común y se utiliza principalmente en contextos de jazz y música experimental. Puede evocar sentimientos de intriga, suspenso y tensión extrema.
- **Mongolian:** La escala mongol es una escala pentatónica con algunas adiciones de notas adicionales. Se utiliza en la música tradicional mongola y puede evocar una sensación de exotismo y misticismo. Tiene un sonido melancólico y evocador. Es una de las más agradecidas para música generativa.
- **Pentatónica mayor:** La escala pentatónica mayor es una escala de cinco notas que suena alegre y positiva. Se utiliza ampliamente en la música popular, el blues y el rock. Es conocida por su simplicidad y versatilidad en la improvisación.
- **Pentatónica menor:** La escala pentatónica menor tiene un sonido melancólico y se utiliza en una amplia variedad de géneros musicales, incluyendo el blues, el rock y la música étnica. Es muy expresiva y se presta bien a la improvisación.
- **Overtone:** La escala armónica mayor, también conocida como escala overtone, se basa en los armónicos naturales de un tono fundamental. Tiene un sonido exótico y etéreo, y se utiliza a menudo en la música contemporánea y en la música étnica de diversas culturas. Es capaz de transmitir una amplia gama de emociones, desde lo misterioso hasta lo celestial.
- **Blues:** La escala blues es una escala de seis notas utilizada principalmente en el blues y el jazz. Tiene un sonido característico y expresivo que evoca sentimientos de tristeza, pasión y soul. Es una escala altamente versátil y se puede utilizar para expresar una amplia gama de emociones.



- **Cromática:** La escala cromática consiste en todos los semitonos dentro de una octava. Es una escala muy disonante y se utiliza principalmente para crear tensión armónica y efectos cromáticos en la música. Tiene una cualidad inestable y se utiliza a menudo en géneros experimentales, música contemporánea y jazz.
- **Whole Tone:** La escala de tono completo está compuesta exclusivamente por intervalos de tono completo. Tiene un sonido ambiguo y misterioso debido a la ausencia de semitonos. Se utiliza en música impresionista, jazz y música experimental. Puede evocar una sensación de suspensión y desorientación.
- **Random:** La escala aleatoria se genera con intervalos de manera impredecible y no sigue ninguna estructura establecida. Su uso puede dar lugar a sonidos disonantes, caóticos o altamente experimentales. Se utiliza en la música vanguardista y la improvisación libre para crear efectos de sorpresa y ruptura de expectativas. Cada vez que se seleccione dará una escala diferente.

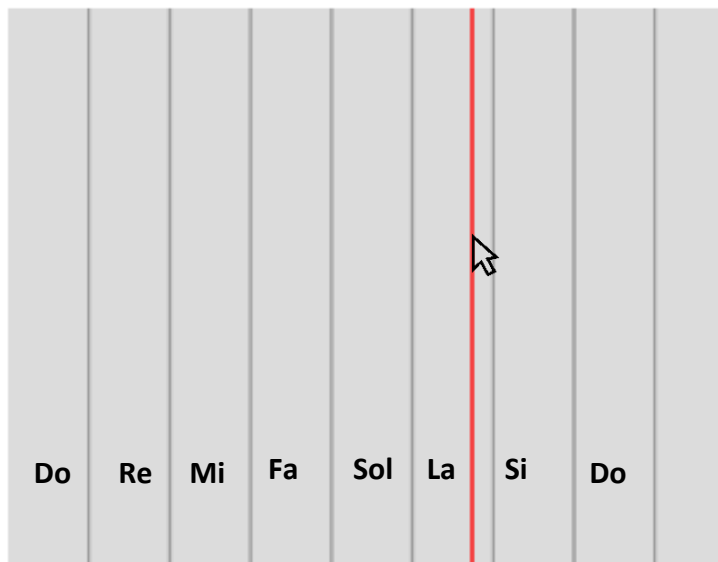
Estas son solo algunas de las escalas que se podrían utilizar. Como se ha visto, es muy fácil añadir escalas nuevas e incluso se podría, en el futuro, implementar la opción al usuario de crear sus propias escalas con facilidad.

## 1.5. Theremin

Además de la capacidad de crear secuencias con los nodos, también se ha añadido un Theremin sobre el cual el usuario tiene control total de las notas. Funciona como un piano sobre el cual se arrastra el ratón para elegir cada nota.

Se trata de un cursor que el usuario puede mover a través de las mismas regiones delimitadas por las notas de la escala, y que si pincha con el ratón en el área donde está el cursor, se emite su sonido correspondiente.

Este theremin está pensado para ser usado a modo de barrido entre las regiones. Se ha creado un mini-proyecto separado para poder ver cómo funciona el theremin por sí solo. La barra roja es el cursor que se mueve por las regiones:



*Ilustración-8 Mini aplicación de Theremin para demostración*

## 2. Proceso de desarrollo

Debido a que el proyecto planteado se mueve en un área no vista antes en la carrera, la programación creativa, era difícil plantear un plan de acción concreto desde el principio, sin antes comprender los aspectos técnicos que iban a aparecer o las posibilidades creativas del proyecto. Es por ello que este proyecto comienza con una larga fase de aprendizaje, investigación y familiarización con la tecnología.

Una vez claro el diseño y el objetivo a perseguir, la metodología planteada en el desarrollo de este proyecto ha sido de naturaleza iterativa, creando una nueva versión en cada iteración con nuevas funcionalidades. Este enfoque se eligió debido a su capacidad para facilitar la adaptación a los cambios y la incorporación progresiva de funcionalidades, permitiendo así una mayor flexibilidad y control en el proceso de desarrollo.

Versión	Implementaciones añadidas
0.1	Puntos blancos que se mueven por el espacio, que crean conexiones al estar cerca
0.2	Los puntos tienen ahora color aleatorio al crearlos
0.3	Cada punto emite una nota al ser creado, dicha nota es asociada aleatoriamente en la creación, usando un sintetizador creado en Tone.js
0.4	Se implementa la lógica de generadores.
1.0	Se implementa la opción de selección de notas en la parte inferior de la pantalla, y la interfaz GUI para control de variables, junto con las escalas. La aplicación ya cumple su cometido principal original
1.1	Se implementa el Theremin. Mejoras estéticas
1.2	Se implementa el Sound Design Panel, y los Offset para los generatos
1.3	Se implementa el fondo animado
1.4	Arreglo de bugs. Mejoras estéticas y organización de los archivos

Durante la ejecución del proyecto, se planeó un proceso de desarrollo estructurado y planificado para garantizar un avance constante y la consecución de los objetivos establecidos. Sin embargo, debido a la coincidencia de exámenes en el periodo intermedio del proyecto, se experimentó una variación en la distribución de las 300 horas previstas distribuidas uniformemente.

Tras un conteo de horas invertidas en cada sesión, se plantea a continuación un Burndown chart donde se representa el descenso de horas restantes planeado en naranja y el descenso real en azul. El desarrollo comenzó el 2 de febrero y terminó el 25 de mayo.

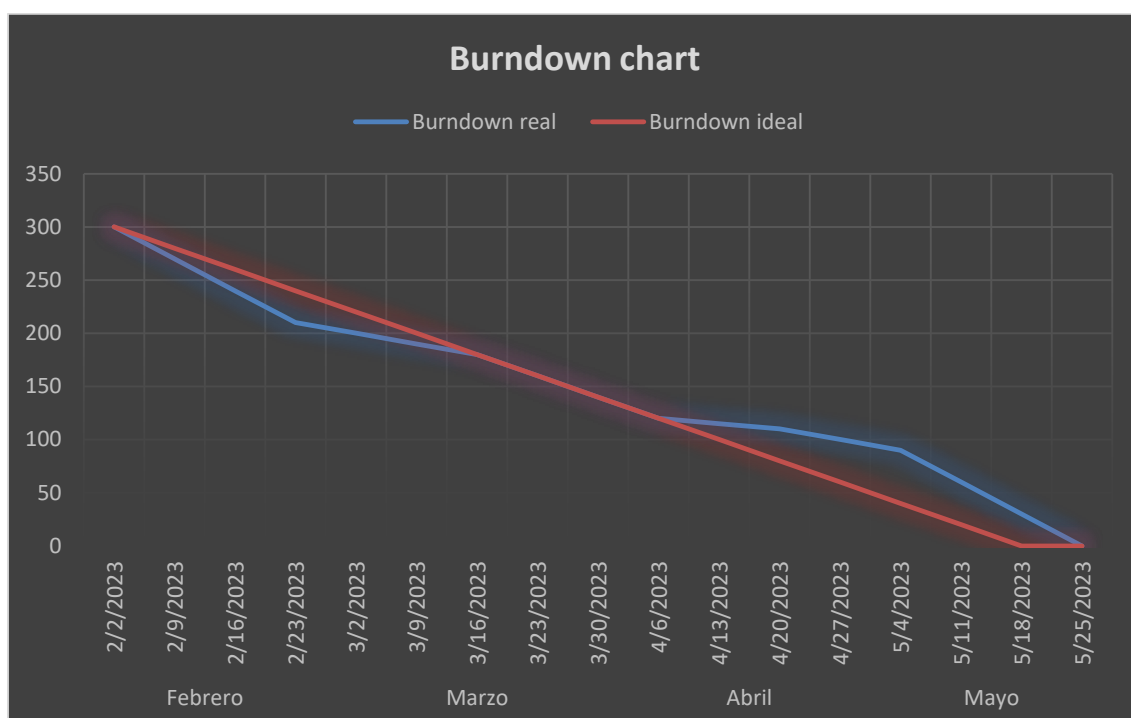


Ilustración 11 - Burndown chart de las horas del proyecto

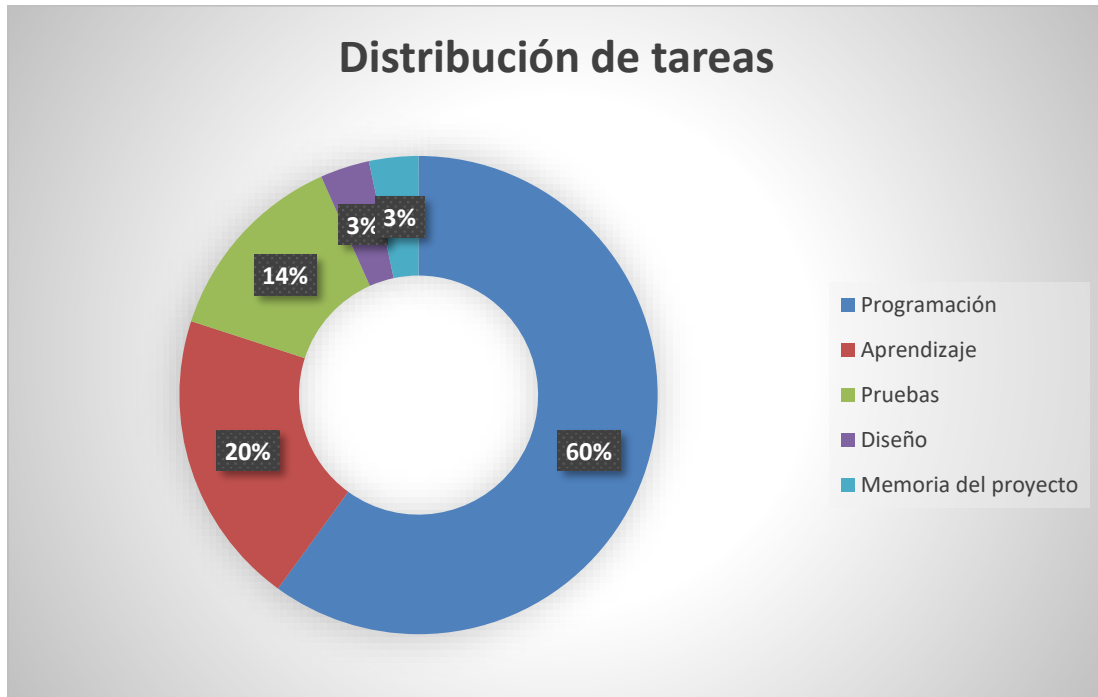
En las etapas iniciales, se dedicó un mayor número de horas al proyecto con el fin de sentar las bases adecuadas y adquirir el conocimiento necesario para el desarrollo eficiente de las tareas posteriores. Estas primeras semanas se destinaron principalmente al aprendizaje, donde se investigaron las tecnologías relevantes, se estudiaron las mejores prácticas y se estableció una sólida comprensión de los requisitos del proyecto.

El primer paso fue establecer una comprensión clara de las posibilidades de los lenguajes utilizados, p5.js y Tone.js. Siendo el primero para generación de gráficos y el segundo para un sintetizador. Ahondaremos en sus capacidades más adelante.

Una vez aprendido lo básico para comenzar a esbozar la idea, comenzó el proceso de diseño. A medida que avanzaba el tiempo y se superaban los obstáculos iniciales, se realizó una transición hacia las fases de programación. Estas etapas requerían una mayor inversión de tiempo y esfuerzo, ya que implicaban la creación de la arquitectura del sistema, la implementación de funcionalidades clave y la resolución de problemas

técnicos específicos. Durante este periodo, el aprendizaje continuaba a la par que las pruebas, por ello se incrementaban por igual. Esto ocupó la mayor parte del tiempo del proyecto.

Finalmente, la memoria se escribió al terminar el proyecto, lo que supuso un incremento de horas en la recta final. Supusieron 10 horas, de baja dificultad, pero alto tedio.



*Ilustración 9 Gráfico de distribución de tareas*

## 3. Tecnologías utilizadas

### 3.1. Javascript

JavaScript es un lenguaje de programación ampliamente utilizado en el desarrollo web. Es especialmente relevante en proyectos interactivos y creativos, ya que permite agregar comportamiento dinámico a las páginas web y manipular elementos HTML y CSS en tiempo real. En este proyecto, JavaScript ha sido la base fundamental para la implementación de la lógica y la interacción de la aplicación.

JavaScript se utiliza para controlar el flujo de la aplicación, responder a eventos del usuario y realizar cálculos y manipulaciones de datos. En este proyecto, se ha utilizado JavaScript para gestionar la lógica de los generadores orbitales y los nodos musicales, así como para sincronizar su comportamiento con la interfaz de usuario.

Además, JavaScript ha sido fundamental para la integración y comunicación con las bibliotecas p5.js y Tone.js. Se han utilizado las API proporcionadas por estas bibliotecas para interactuar con los elementos gráficos y sonoros de la aplicación. JavaScript ha permitido crear instancias de objetos, llamar a métodos, establecer eventos y manipular propiedades para lograr la interacción deseada.

La flexibilidad y versatilidad de JavaScript han sido aprovechadas para implementar la lógica de los generadores orbitales y los nodos musicales, así como para manipular los parámetros de síntesis de Tone.js y los controles de dat.GUI. JavaScript ha permitido que la aplicación responda a las acciones de los usuarios y ajuste dinámicamente la música generada y la visualización de los elementos.

Veamos cómo se han utilizado estas tres librerías:

### 3.1.1. P5.js

p5.js es una biblioteca de JavaScript que facilita la creación de proyectos interactivos, gráficos y animaciones en el navegador. Diseñada para ser accesible y amigable para principiantes, p5.js se basa en los principios de Processing, una plataforma de programación visual para artistas y diseñadores. Con p5.js, los desarrolladores pueden utilizar el lenguaje JavaScript para crear fácilmente elementos visuales y experiencias interactivas en la web.

p5.js se da a conocer en su página web <https://p5js.org/>, donde se nos da la bienvenida explicando el propósito de la librería, y donde se puede acceder a documentación, proyectos, librerías y code playgrounds para conocer mejor la plataforma.

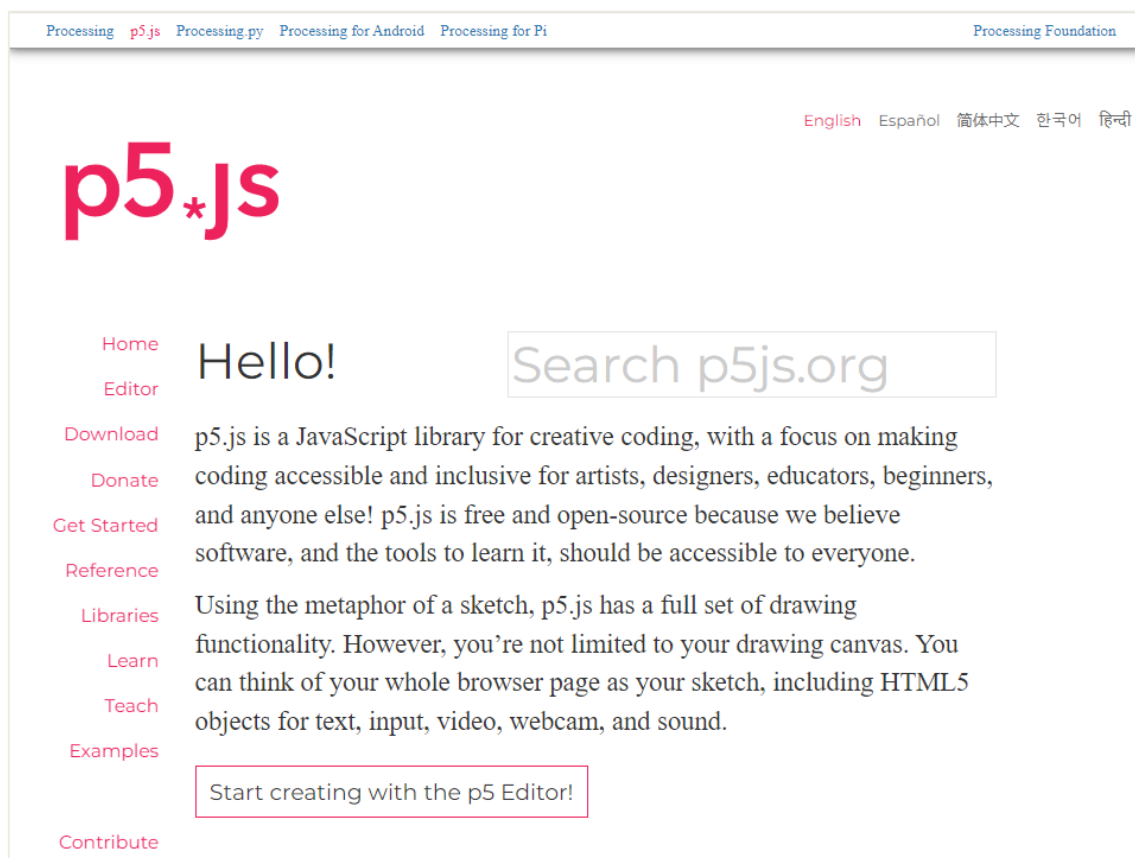


Ilustración 10 Página de inicio de p5.js.org

La biblioteca p5.js proporciona una amplia gama de funciones incorporadas para el dibujo de gráficos, la manipulación de imágenes y la animación. Estas funciones permiten a los desarrolladores crear visualizaciones, juegos, arte generativo y otros tipos de proyectos creativos. Además, p5.js está diseñado para trabajar en conjunto con el Document Object Model (DOM) de HTML, lo que facilita la integración de elementos interactivos en páginas web existentes. Los principales usos que se le han dado han sido:

### 3.1.1.1. Inicialización del canvas:

Prepara el entorno para el dibujo de p5.js. Por defecto, el tamaño del canvas es fijo, por tanto, la aplicación no sería responsiva, pero se puede arreglar redimensionando el canvas con su función correspondiente.

## createCanvas()

### Description

Creates a canvas element in the document and sets its dimensions in pixels. This method should be called only once at the start of `setup()`. Calling `createCanvas` more than once in a sketch will result in very unpredictable behavior. If you want more than one drawing canvas you could use `createGraphics()` (hidden by default but it can be shown).

Important note: in 2D mode (i.e. when `p5.Renderer` is not set) the origin (0,0) is positioned at the top left of the screen. In 3D mode (i.e. when `p5.Renderer` is set to `WEBGL`), the origin is positioned at the center of the canvas. See [this issue](#) for more information.

The system variables `width` and `height` are set by the parameters passed to this function. If `createCanvas()` is not used, the window will be given a default size of 100×100 pixels.

For more ways to position the canvas, see the [positioning the canvas](#) wiki page.

### Examples



```
function setup() {  
  createCanvas(100, 50);  
  background(153);  
  line(0, 0, width, height);  
}
```

edit reset copy



Con la función `resizeCanvas`, podemos hacer la aplicación responsiva, para que redibuje el canvas ajustándose al nuevo tamaño de la ventana.

## `resizeCanvas()`

### Description

Resizes the canvas to given width and height. The canvas will be cleared and draw will be called immediately, allowing the sketch to re-render itself in the resized canvas.

### Examples

---

```
function setup() {  
  createCanvas(windowWidth, windowHeight);  
}  
  
function draw() {  
  background(0, 100, 200);  
}  
  
function windowResized() {  
  resizeCanvas(windowWidth, windowHeight);  
}
```

### 3.1.1.2. Carga de imágenes

Se ha utilizado para cargar el fondo de pantalla, detrás de los gráficos. La imagen está incluida entre los archivos del proyecto. Como no puede aparecer después del resto de elementos, no se precarga, sino que se espera a que la imagen carga para iniciar el resto de la app.

## loadImage()

### Description

Loads an image from a path and creates a `p5.Image` from it.

The image may not be immediately available for rendering. If you want to ensure that the image is ready before doing anything with it, place the `loadImage()` call in `preload()`. You may also supply a callback function to handle the image when it's ready.

The path to the image should be relative to the HTML file that links in your sketch. Loading an image from a URL or other remote location may be blocked due to your browser's built-in security.

You can also pass in a string of a base64 encoded image as an alternative to the file path. Remember to add "data:image/png;base64," in front of the string.

### Examples



```
let img;
function preload() {
  img = loadImage('assets/laDefense.jpg');
}
function setup() {
  image(img, 0, 0);
}
```

edit reset copy

### 3.1.1.3. Gráficos

p5.js proporciona una serie de funciones incorporadas para la creación de gráficos y animaciones. Esta funcionalidad se utilizó ampliamente en este proyecto para representar visualmente los generadores orbitales y los nodos musicales. Se utilizaron métodos de dibujo de p5.js para crear círculos representando los generadores orbitales, y puntos móviles para los nodos musicales que orbitaban alrededor de los generadores. Veamos cada uno de los métodos usados.

- **Ellipse:** Se ha utilizado para generar los círculos que aparecen en toda la aplicación. Se ha elegido utilizar esta función en vez de la del círculo por tener más opciones y funciones asociadas. Se verá con frecuencia en el código, tanto para los botones como para los nodos y generadores.

#### ellipse()

##### Description

Draws an ellipse (oval) to the screen. By default, the first two parameters set the location of the center of the ellipse, and the third and fourth parameters set the shape's width and height. If no height is specified, the value of width is used for both the width and height. If a negative height or width is specified, the absolute value is taken.

An ellipse with equal width and height is a circle. The origin may be changed with the `ellipseMode()` function.

##### Examples



```
ellipse(56, 46, 55, 55);  
describe('white ellipse with black outline in  
middle of a gray canvas');
```

edit reset copy

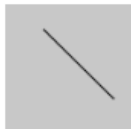
- Line: Se utiliza para dibujar líneas en la aplicación, dados los dos puntos de inicio y final. Principalmente usada para el theremin y el gráfico del diseño ADSR.

## line()

### Description

Draws a line (a direct path between two points) to the screen. If called with only 4 parameters, it will draw a line in 2D with a default width of 1 pixel. This width can be modified by using the `strokeWeight()` function. A line cannot be filled, therefore the `fill()` function will not affect the color of a line. So to color a line, use the `stroke()` function.

### Examples



```
line(30, 20, 85, 75);  
describe(  
  'a 78 pixels long line running from mid-top to  
  bottom-right of canvas'  
);
```

edit reset copy

- Color: Esencial para el relleno de los elementos. Se ha utilizado tanto llamando a los colores por su valor RGB en enteros, como pasándole colores en hexadecimal para tareas de diseño más estático.

## color()

### Description

Creates colors for storing in variables of the color datatype. The parameters are interpreted as RGB or HSB values depending on the current `colorMode()`. The default mode is RGB values from 0 to 255 and, therefore, the function call `color(255, 204, 0)` will return a bright yellow color.

Note that if only one value is provided to `color()`, it will be interpreted as a grayscale value. Add a second value, and it will be used for alpha transparency. When three values are specified, they are interpreted as either RGB or HSB values. Adding a fourth value applies alpha transparency.

If a single string argument is provided, RGB, RGBA and Hex CSS color strings and all named color strings are supported. In this case, an alpha number value as a second argument is not supported, the RGBA form should be used.

### Examples



```
let c = color(255, 204, 0);
fill(c);
noStroke();
rect(30, 20, 55, 55);
describe(`Yellow rect in middle right of canvas,
with 55 pixel width and height.`);
```

edit reset copy

- Stroke: Define el grosor y el color de los bordes de las elipses y las líneas creadas. Igualmente, esencial para definir los aspectos de los elementos.

## stroke()

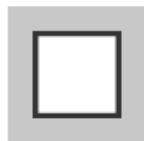
### Description

Sets the color used to draw lines and borders around shapes. This color is either specified in terms of the RGB or HSB color depending on the current `colorMode()` (the default color space is RGB, with each value in the range from 0 to 255). The alpha range by default is also 0 to 255.

If a single string argument is provided, RGB, RGBA and Hex CSS color strings and all named color strings are supported. In this case, an alpha number value as a second argument is not supported, the RGBA form should be used.

A `p5.Color` object can also be provided to set the stroke color.

### Examples



```
// Grayscale integer value
strokeWeight(4);
stroke(51);
rect(20, 20, 60, 60);
describe('White rect at center with dark charcoal
grey outline.');
```

[edit](#) [reset](#) [copy](#)

- Alpha: Define la cantidad de opacidad que tiene un elemento dibujado. Será de gran utilidad para dibujar un fondo de pantalla dinámico con líneas que aparecen y desaparecen en perspectiva.

## alpha()

### Description

Extracts the alpha value from a color or pixel array.

### Examples



```
noStroke();
let c = color(0, 126, 255, 102);
fill(c);
rect(15, 15, 35, 70);
let value = alpha(c); // Sets 'value' to 102
fill(value);
rect(50, 15, 35, 70);
describe('Left half of canvas light blue and right
half light charcoal grey.');
```

edit reset copy

#### 3.1.1.4. Interactividad

p5.js ofrece soporte para interacciones de usuario a través del ratón, el teclado y otros dispositivos de entrada. En este proyecto, se empleó esta funcionalidad para permitir a los usuarios interactuar directamente con la creación y control de los elementos de la aplicación con la pulsación y el movimiento del ratón.

- `MouseClicked` registra cuándo el usuario hace click en el canvas. Enviaremos esta señal a través del código para registrar y provocar eventos

#### `mouseClicked()`

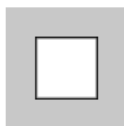
##### Description

The `mouseClicked()` function is called once after a mouse button has been pressed and then released.

Browsers handle clicks differently, so this function is only guaranteed to be run when the left mouse button is clicked. To handle other mouse buttons being pressed or released, see `mousePressed()` or `mouseReleased()`.

Browsers may have different default behaviors attached to various mouse events. To prevent any default behavior for this event, add "return false" to the end of the function.

##### Examples



```
// Click within the image to change
// the value of the rectangle
// after the mouse has been clicked

let value = 0;
function draw() {
  fill(value);
  rect(25, 25, 50, 50);
  describe('black 50-by-50 rect turns white with
mouse click/press.');
```

```
function mouseClicked() {
  if (value === 0) {
    value = 255;
  } else {
    value = 0;
  }
}
```




- **MouseX y MouseY:** Variables del sistema que registran en tiempo real la posición del cursor. Serán de gran utilidad a la hora de crear nuevos elementos en la posición del cursor y para mover el Theremin a donde tiene que estar sonando. Ambas variables funcionan de forma análoga, como se puede comprobar.

## mouseX

### Description

The system variable `mouseX` always contains the current horizontal position of the mouse, relative to (0, 0) of the canvas. The value at the top-left corner is (0, 0) for 2-D and (-width/2, -height/2) for WebGL. If touch is used instead of mouse input, `mouseX` will hold the x value of the most recent touch point.

### Examples



```

// Move the mouse across the canvas
function draw() {
  background(244, 248, 252);
  line(mouseX, 0, mouseX, 100);
  describe('horizontal black line moves left and
right with mouse x-position');
}

```

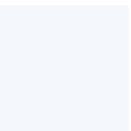
edit reset copy

## mouseY

### Description

The system variable `mouseY` always contains the current vertical position of the mouse, relative to (0, 0) of the canvas. The value at the top-left corner is (0, 0) for 2-D and (-width/2, -height/2) for WebGL. If touch is used instead of mouse input, `mouseY` will hold the y value of the most recent touch point.

### Examples



```

// Move the mouse across the canvas
function draw() {
  background(244, 248, 252);
  line(0, mouseY, 100, mouseY);
  describe('vertical black line moves up and down
with mouse y-position');
}

```

edit reset copy

### 3.1.1.5. Relaciones lógicas entre objetos

Integrar la lógica adecuada entre los elementos es esencial para que todo funcione de acuerdo con el modelo propuesto, respondiendo a las peticiones del usuario e implementando las reglas del diseño, especialmente cuando se quieren hacer interacciones con la librería musical de Tone.

- Map: Funciona como el map tradicional, pero diseñado para objetos de p5.js. Se utiliza para asociar notas a colores, botones a funciones, colores a posiciones, etc.

map()

#### Description

Re-maps a number from one range to another.

In the first example above, the number 25 is converted from a value in the range of 0 to 100 into a value that ranges from the left edge of the window (0) to the right edge (width).

#### Examples



```
function setup() {  
  noStroke();  
}  
  
function draw() {  
  background(204);  
  let x1 = map(mouseX, 0, width, 25, 75);  
  ellipse(x1, 25, 25, 25);  
  //This ellipse is constrained to the 0-100 range  
  //after setting withinBounds to true  
  let x2 = map(mouseX, 0, width, 0, 100, true);  
  ellipse(x2, 75, 25, 25);  
  
  describe(`Two 25×25 white ellipses move with  
mouse x.  
  Bottom has more range from X`);  
}
```

- Lerp: Se utiliza para interpolar valores, ya sea para crear gradientes u obtener valores intercalados. Se puede utilizar en el diseño de gráficos para hacer cualquier transición mucho más uniforme. Se ha usado para transparencias, colores y formas.

## lerp()

### Description

Calculates a number between two numbers at a specific increment. The amt parameter is the amount to interpolate between the two values where 0.0 is equal to the first point, 0.1 is very near the first point, 0.5 is half-way in between, and 1.0 is equal to the second point. If the value of amt is more than 1.0 or less than 0.0, the number will be calculated accordingly in the ratio of the two given numbers. The lerp() function is convenient for creating motion along a straight path and for drawing dotted lines.

### Examples

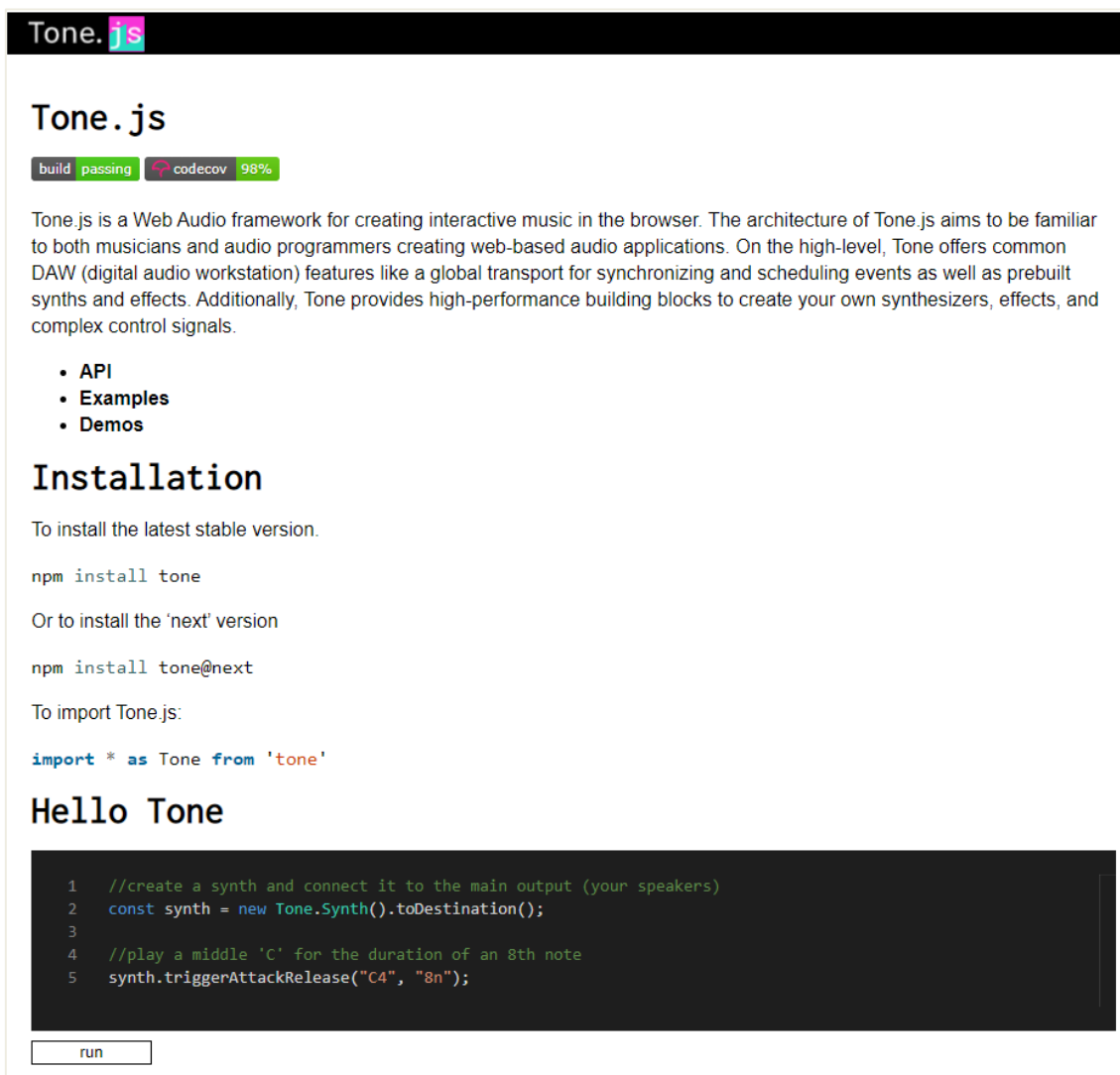


```
function setup() {  
  background(200);  
  let a = 20;  
  let b = 80;  
  let c = lerp(a, b, 0.2);  
  let d = lerp(a, b, 0.5);  
  let e = lerp(a, b, 0.8);  
  
  let y = 50;  
  
  strokeWeight(5);  
  stroke(0); // Draw the original points in black  
  point(a, y);  
  point(b, y);  
  
  stroke(100); // Draw the lerp points in gray  
  point(c, y);  
  point(d, y);  
  point(e, y);  
  
  describe(`5 points horizontally staggered mid-  
  canvas.  
  mid 3 are grey, outer black`);  
}
```

### 3.1.2. Tone.js

Tone.js es una poderosa biblioteca de JavaScript para la creación de música en el navegador. Permite a los desarrolladores generar, sintetizar, organizar y manipular el sonido de una forma flexible y avanzada.

Tone.js también tiene su lugar en la web, visitable en <https://tonejs.github.io/>, los usuarios son recibidos con una bienvenida, explicando el propósito de Tone y se les proporciona acceso directo a tutoriales de instalación y uso de la librería, así como ejemplos y demos. En la página principal se muestra lo básico de todas las funciones necesarias para construir una aplicación musical.



**Tone.js**

build passing codecov 98%

Tone.js is a Web Audio framework for creating interactive music in the browser. The architecture of Tone.js aims to be familiar to both musicians and audio programmers creating web-based audio applications. On the high-level, Tone offers common DAW (digital audio workstation) features like a global transport for synchronizing and scheduling events as well as prebuilt synths and effects. Additionally, Tone provides high-performance building blocks to create your own synthesizers, effects, and complex control signals.

- API
- Examples
- Demos

## Installation

To install the latest stable version.

```
npm install tone
```

Or to install the 'next' version

```
npm install tone@next
```

To import Tone.js:

```
import * as Tone from 'tone'
```

## Hello Tone

```
1 //create a synth and connect it to the main output (your speakers)
2 const synth = new Tone.Synth().toDestination();
3
4 //play a middle 'C' for the duration of an 8th note
5 synth.triggerAttackRelease("C4", "8n");
```

run

Ilustración 14 – Página de bienvenida Tone.js

En el proyecto se han utilizado los siguientes elementos de Tone.js:

- **Inicialización:** Para iniciar el sonido en una aplicación web que usa Tone, es necesario crear una promesa, que se resuelve cuando el usuario interactúa haciendo el primer clic. En nuestra aplicación, se inicia el objeto `Tone.start()` desde el principio y se queda activo durante toda la sesión.

## Starting Audio

**IMPORTANT:** Browsers will not play *any* audio until a user clicks something (like a play button). Run your Tone.js code only after calling `Tone.start()` from a event listener which is triggered by a user action such as "click" or "keydown".

`Tone.start()` returns a promise, the audio will be ready only after that promise is resolved. Scheduling or playing audio before the `AudioContext` is running will result in silence or incorrect scheduling.

```
1 //attach a click listener to a play button
2 document.querySelector('button')?.addEventListener('click', async () => {
3   await Tone.start()
4   console.log('audio is ready')
5 })
```

- **Síntesis de sonido:** Tone.js proporciona una gama de métodos y objetos para generar sonido. El más básico para comenzar es el `Tone.Synth`, que puede reproducir notas recibéndolas como parámetro, explicitadas mediante su nombre ("C4") o su frecuencia como float (261.6). En este proyecto, se utilizaron estas capacidades de sintetizador para producir las notas musicales asociadas a cada nodo.

## Tone.Synth

**Tone.Synth** is a basic synthesizer with a single **oscillator** and an **ADSR envelope**.

**triggerAttack / triggerRelease**

`triggerAttack` starts the note (the amplitude is rising), and `triggerRelease` is when the amplitude is going back to 0 (i.e. **note off**).

```
1 const synth = new Tone.Synth().toDestination();
2 const now = Tone.now()
3 // trigger the attack immediately
4 synth.triggerAttack("C4", now)
5 // wait one second before triggering the release
6 synth.triggerRelease(now + 1)
```

Sin embargo, el sintetizador básico es monofónico, esto quiere decir que solo puede reproducir una nota a la vez. Para resolver esto, se utiliza el sintetizador polifónico, `PolySynth`. Aquí podremos llamar a una instancia del sintetizador por cada nota reproducida, y podremos combinar diferentes sonidos para dar lugar a otros nuevos.

## Instruments

There are numerous synths to choose from including **Tone.FMSynth**, **Tone.AMSynth** and **Tone.NoiseSynth**.

All of these instruments are **monophonic** (single voice) which means that they can only play one note at a time.

To create a **polyphonic** synthesizer, use **Tone.PolySynth**, which accepts a monophonic synth as its first parameter and automatically handles the note allocation so you can pass in multiple notes. The API is similar to the monophonic synths, except `triggerRelease` must be given a note or array of notes.

```
1 const synth = new Tone.PolySynth(Tone.Synth).toDestination();
2 const now = Tone.now();
3 synth.triggerAttack("D4", now);
4 synth.triggerAttack("F4", now + 0.5);
5 synth.triggerAttack("A4", now + 1);
6 synth.triggerAttack("C5", now + 1.5);
7 synth.triggerAttack("E5", now + 2);
8 synth.triggerRelease(["D4", "F4", "A4", "C5", "E5"], now + 4);
```

Además, según el sintetizador elegido, al definir sus propiedades del oscilador mediante el ADSR envelope. Así, podremos obtener un diseño de sonido único. Esto se implementa al modificar las propiedades del sintetizador, tal que así:

```
thereminSynth = new Tone.Synth({
  oscillator: {
    type: "sine", // Use a sawtooth wave for the oscillator
    detune: 10, // Detune the oscillator for a richer sound
  },
  envelope: {
    attack: 0.2, // Increase the attack time for a softer attack
    decay: 0.2, // Decrease the decay time for a sharper decay
    sustain: 0.5, // Decrease the sustain for a shorter sustain
    release: 15, // Increase the release for a longer fade out
  },
  volume: -16,
}).toDestination();
```

- Control de tiempo y ritmo: Las peticiones en javascript no están temporizadas de la forma precisa que requiere un patrón rítmico. Por ello, Tone.js permite un control preciso del tiempo, esencial para cualquier tipo de creación musical. Este aspecto de Tone.js es clave para coordinar el comportamiento rítmico de los nodos musicales en relación con su posición orbital. Se empleó la función Transport de Tone.js para organizar y sincronizar la secuencia rítmica de los nodos. Transport funciona como un metrónomo que cuenta los tiempos definidos en bucle.

## Scheduling

### Transport

**Tone.Transport** is the main timekeeper. Unlike the `AudioContext` clock, it can be started, stopped, looped and adjusted on the fly. You can think of it like the arrangement view in a Digital Audio Workstation or channels in a Tracker.

Multiple events and parts can be arranged and synchronized along the Transport. **Tone.Loop** is a simple way to create a looped callback that can be scheduled to start and stop.

```
1 // create two monophonic synths
2 const synthA = new Tone.FMSynth().toDestination();
3 const synthB = new Tone.AMSynth().toDestination();
4 //play a note every quarter-note
5 const loopA = new Tone.Loop(time => {
6   synthA.triggerAttackRelease("C2", "8n", time);
7 }, "4n").start(0);
8 //play another note every off quarter-note, by starting it "8n"
9 const loopB = new Tone.Loop(time => {
10  synthB.triggerAttackRelease("C4", "8n", time);
11 }, "4n").start("8n");
12 // the loops start when the Transport is started
13 Tone.Transport.start();
14 // ramp up to 800 bpm over 10 seconds
15 Tone.Transport.bpm.rampTo(800, 10);
```

- Efectos y procesamiento de señales: Tone.js incluye una amplia variedad de efectos y herramientas de procesamiento de señales, que permiten manipular el sonido de diversas maneras. En el contexto de este proyecto, se utilizó esta funcionalidad para dar a los usuarios la capacidad de modificar el sonido de los nodos musicales a través de diferentes efectos, como el volumen, el eco, reverb, filtros y más.

## Effects

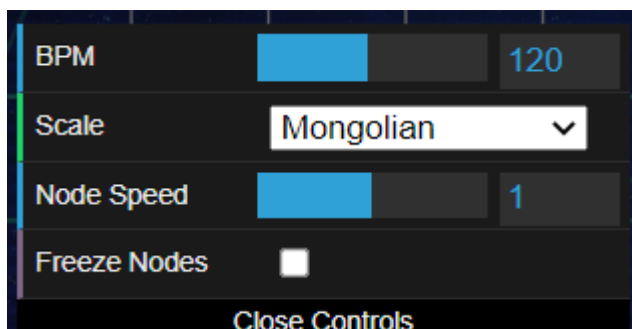
In the above examples, the sources were always connected directly to the **Destination**, but the output of the synth could also be routed through one (or more) effects before going to the speakers.

```
1 const player = new Tone.Player({
2   url: "https://tonejs.github.io/audio/berklee/gurgling_theremin_1.mp3",
3   loop: true,
4   autostart: true,
5 })
6 //create a distortion effect
7 const distortion = new Tone.Distortion(0.4).toDestination();
8 //connect a player to the distortion
9 player.connect(distortion);
```

Veremos en el capítulo de la implementación como se utilizan en detalle todas estas funciones.

### 3.1.3. dat.GUI

dat.GUI es una biblioteca JavaScript que proporciona una interfaz de usuario intuitiva para controlar y ajustar parámetros en tiempo real. Es una librería ampliamente utilizada en proyectos de programación creativa, ya que permite a los usuarios interactuar con los parámetros de forma visual y experimentar con diferentes configuraciones.



A la izquierda se puede observar un ejemplo de cómo se integra dat.GUI en el proyecto. Es una interfaz fácilmente navegable, colapsable y que indica con claridad los parámetros y su valor.

*Ilustración -11 Integración de interfaz dat.gui en el proyecto*

Al utilizar dat.gui.js, se han podido crear controles deslizantes, botones y casillas de verificación que reflejan los parámetros asociados a los sintetizadores, los nodos y generadores. Estos controles proporcionan una forma sencilla de modificar los valores y ver instantáneamente cómo afectan al comportamiento de la música generada. Los usuarios pueden experimentar con diferentes configuraciones y explorar la relación entre los parámetros y el resultado sonoro.

Además, se ha aprovechado la capacidad de dat.gui.js de vincular directamente los controles con las propiedades de los objetos en JavaScript. Esto ha permitido actualizar automáticamente los valores de los parámetros en tiempo real, sin necesidad de recargar la página o reiniciar la aplicación. Esta funcionalidad resulta especialmente valiosa cuando los usuarios desean realizar ajustes finos y observar cómo afectan a la música sin interrupciones.



### 3.2. HTML + CSS

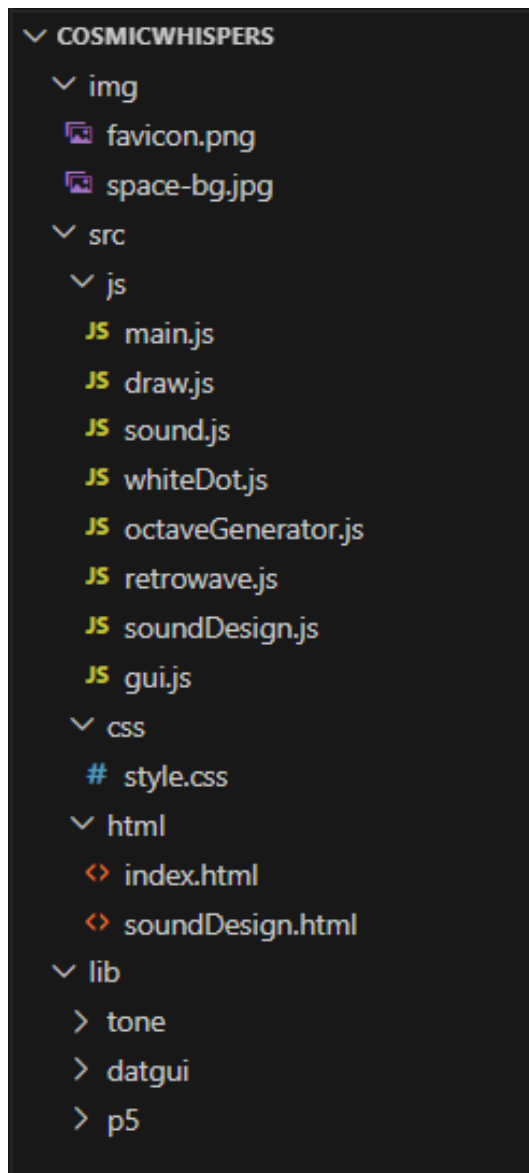
HTML (HyperText Markup Language) y CSS (Cascading Style Sheets) son dos lenguajes fundamentales en el desarrollo web. HTML se encarga de estructurar el contenido de una página web, mientras que CSS se utiliza para definir su presentación y apariencia visual. En Cosmic Whispers, se ha aprovechado las capacidades de HTML y CSS para crear la interfaz de usuario de la aplicación y personalizar su aspecto estético.

HTML ha sido utilizado para estructurar y organizar los elementos de la interfaz de usuario de la aplicación. Se han definido elementos como botones, paneles, contenedores y otros elementos de entrada de datos utilizando las etiquetas y atributos de HTML. Gracias a la semántica proporcionada por HTML, se ha podido crear una estructura lógica y coherente para la interfaz, facilitando la comprensión y navegación por parte de los usuarios.

Por otro lado, CSS ha sido esencial para dar estilo y diseño a la aplicación. Mediante la definición de reglas de estilo, se ha podido personalizar la apariencia visual de los elementos HTML, utilizando propiedades CSS para establecer colores, fuentes, tamaños, márgenes y otros aspectos visuales. Esto ha permitido que la interfaz de usuario sea atractiva y coherente con el tema estético del proyecto, mejorando la experiencia estética de los usuarios.

## 4. Implementación

La aplicación se ha conformado por diferentes componentes en distintos archivos, separados por funcionalidad. Vamos a ver cómo funciona cada uno, para qué se usan y cómo están implementados.

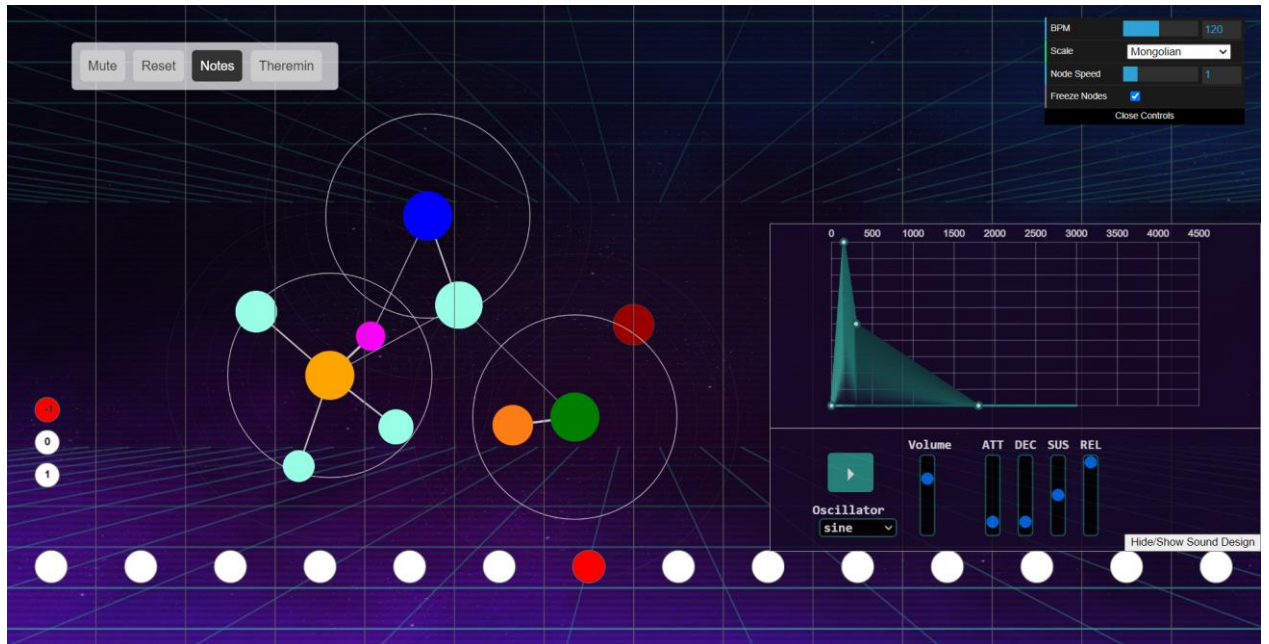


Hay 3 carpetas principales, /img, /src y /lib. Siguiendo el convenio de nombrado de archivos habitual, todo el código fuente del proyecto específico está en la carpeta /src, separado en archivos js, html y css. Estos archivos tienen llamadas a las librerías en la carpeta /lib, donde están las versiones minificadas de Tone, p5 y datgui, y las imágenes en la carpeta /img, que contiene un favicon para usarlo en el navegador, y el space-bg que se llama también en el código para renderizarla en el canvas.

Vamos a explicar cada archivo del /src, pero nos centraremos en el contenido de javascript, que es donde se hace uso de las librerías y los conceptos explicados en conceptos anteriores.

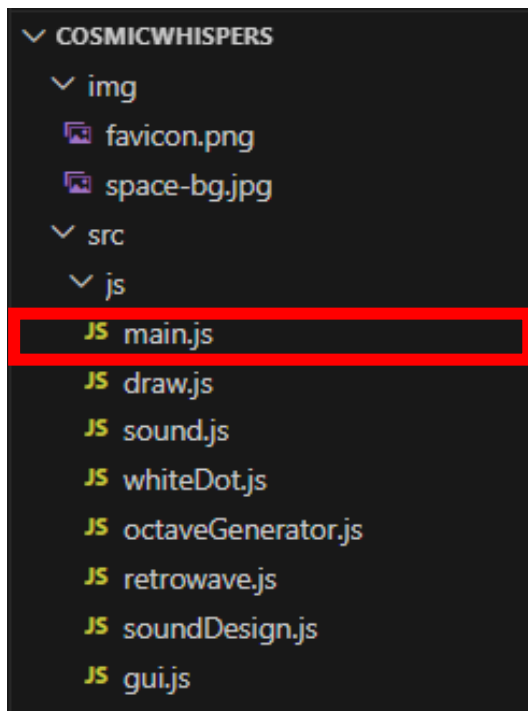
Ilustración 15 – Sistema de archivos de Cosmic Whispers

En la imagen se proporciona una captura de todos los elementos disponibles en la aplicación. Al terminar esta sección, se comprenderá cada elemento, su funcionamiento interno y su propósito.



*Ilustración 16 - Captura de la aplicación usando todos sus elementos*

#### 4.1. Archivo principal



El main.js se ha mantenido lo más pequeño posible para hacer lo más modular posible la aplicación. La función `setup()` configura el canvas de p5, su tasa de refresco a 30 fps, el fondo, el metrónomo de Tone.js.

la barra de herramientas y la interfaz gráfica de usuario (GUI) de la aplicación se llaman en las dos funciones del final, que se encuentran en el archivo `gui.js`

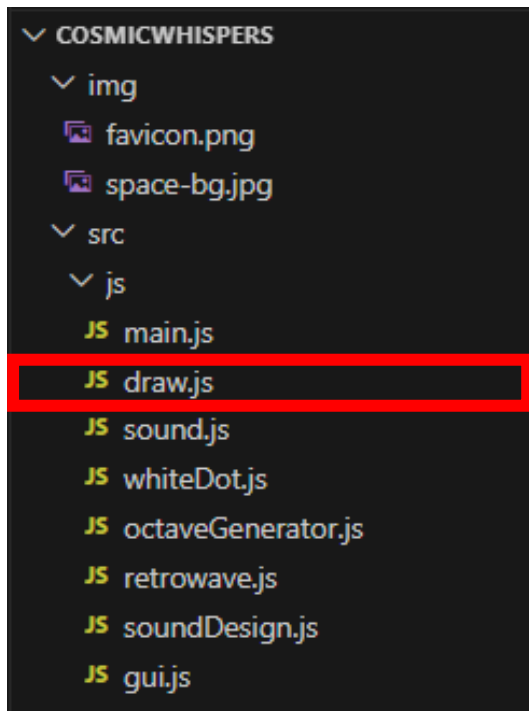
```
function setup() {
  createCanvas(windowWidth, windowHeight);
  setupBackground();
  frameRate(30);

  // set up Tone.js Transport to play a 4-beat pattern
  Tone.Transport.bpm.value = 120;
  Tone.Transport.timeSignature = 4;
  Tone.Transport.loopEnd = "4m";

  Tone.Transport.start();
  Tone.Transport.scheduleRepeat(onBeat, "8n");

  setupToolbar();
  setupGui();
}
```

## 4.2. Draw

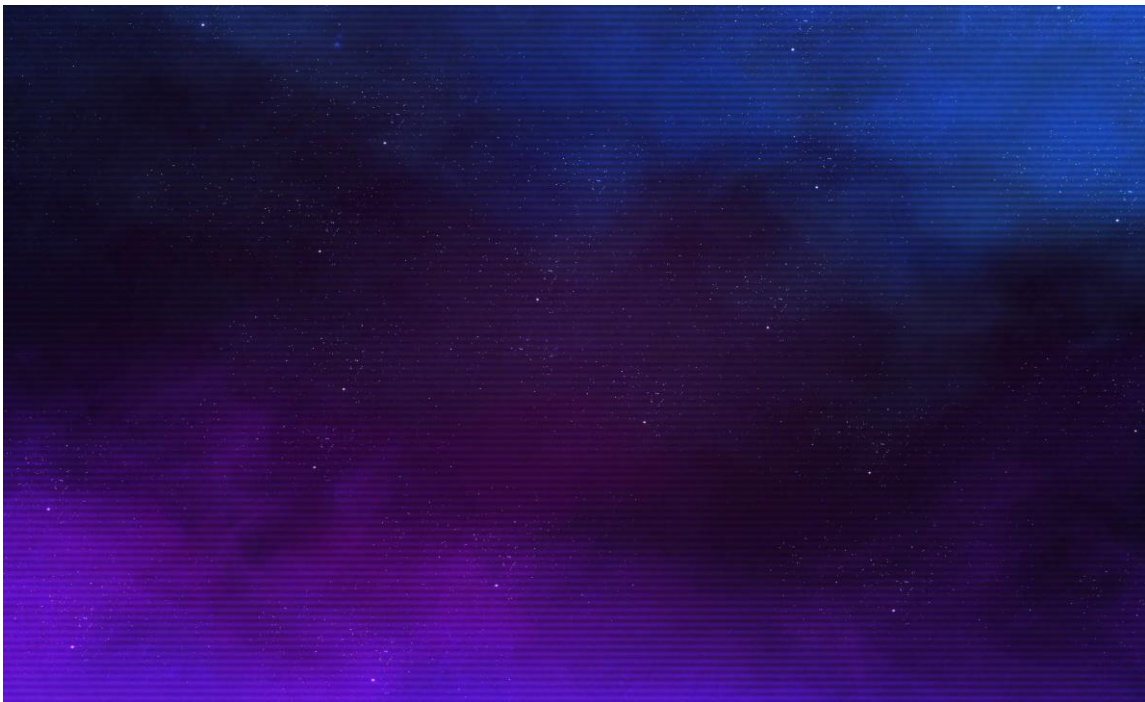


Es uno de los archivos principales. Este código configura la aplicación, maneja la interacción del usuario y dibuja elementos visuales en el lienzo en función del estado de la aplicación y las interacciones del usuario.

Primero, se llama a la función `preload()`, que carga una imagen de fondo utilizando la función `loadImage()` de p5.

```
let backgroundImage;  
function preload() {  
  backgroundImage =  
  loadImage('../space-bg.jpg');  
}
```

Esto carga la imagen siguiente como fondo:



*Ilustración 17 - Fondo usado en Cosmic Whispers*

La función `draw()` es bastante larga y compleja, pues es el engranaje principal del dibujo de gráficos de la aplicación. Se ejecuta en un bucle, 30 veces por segundo como se ha definido en el `framerate`, y se encarga de dibujar el fondo interactivo, crear las instancias de generadores y notas cuando se clicca en la pantalla y actualizar la información que se ve en pantalla.

También maneja la interacción del mouse en los modos "theremin" y "create". Permite eliminar las notas cuando son pinchadas con el ratón. También dibuja las regiones para el Theremin, y actualiza el sintetizador activo y las notas en función del modo de interacción seleccionado.

En la primera sección se llama a la imagen para cargarla, se llama al fondo interactivo (definido en `retrowave.js`), y se llama a la función de dibujo del metrónomo. También se define una regla para que el número de notas no exceda 75, para no colapsar el programa.

```
function draw() {
  image(backgroundImage, 0, 0, width, height);
  drawBackground();

  numDots = whiteDots.length;
  maxDots = 75;

  backgroundSpeed = 10;
  updateBackgroundSpeed();

  background(0, 50);
  drawMetronome();
```

Esta sección de código javascript llama a los métodos de las clases implementadas de los nodos y los generadores para que se dibujen en p5:

```
for (let whiteDot of whiteDots) {
  whiteDot.update();
  whiteDot.display();
  whiteDot.checkDistance(octaveGenerators, distanceThreshold = 200,
lineColor = 200);
}
for (const generator of octaveGenerators) {
  generator.display();
  generator.updateWhiteDotsInRange(whiteDots);
  generator.labelWhiteDots();
}
```

Aquí se implementa la interacción del mouse con las notas cuando el ratón pasa por encima de ellas. Cuando se clican, se eliminan, pero eso se implementa en whiteDot.js, aquí solamente se recoge la señal y se cambia el cursor del mouse.

```
// Mouse Over for interacting with star
let mouseOverDot = false;
for (let whiteDot of whiteDots) {
  whiteDot.display();

  if (whiteDot.isMouseOver()) {
    mouseOverDot = true;
  }
}
if (mouseOverDot) {
  cursor(HAND);
} else {
  cursor(ARROW);
}
```

Con este código se dibuja el menú de notas en la escala en la parte inferior de la pantalla, para que el usuario las pueda seleccionar. Esto interactúa con el archivo gui.js, donde se implementa la lógica de la interfaz.

```
let currentNote;
let isPlaying = false;
let currentScale = scales[settings.scale]
const scaleNotes = currentScale.map(interval => {
  const frequency = rootFrequency * Math.pow(2, interval / 12);
  return Tone.Frequency(frequency).toNote();
});
```

## Dibujo y lógica del Theremin

A continuación, se implementa la sección del Theremin, todavía parte de la función `draw()`. Se trata de un cursor que el usuario puede mover a través de las mismas regiones delimitadas por las notas de la escala, y que si pincha con el ratón en el área donde está el cursor, se emite su sonido correspondiente. Primero se divide el canvas en múltiples secciones, y se dibuja una línea para demarcarlas.

```
const numRegions = scaleNotes.length;

const regionWidth = width / numRegions;
// Draw gridlines for theremin
stroke(100);
for (let i = 1; i < numRegions; i++) {
  line(regionWidth * i, 0, regionWidth * i, height);
}
```

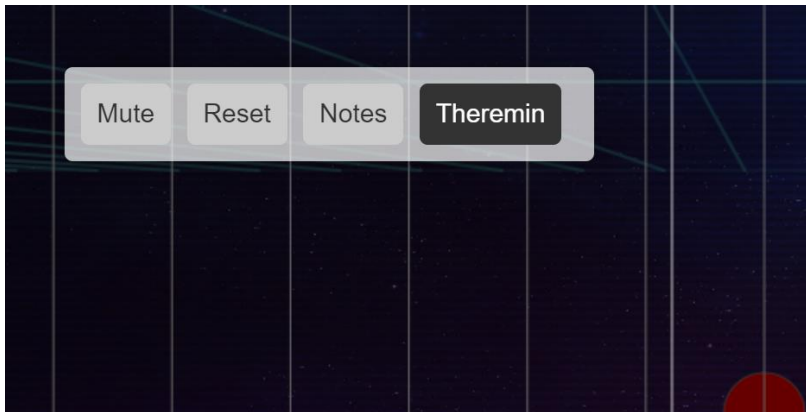


Ilustración 18 – Uso del cursor del Theremin

Además, el eje vertical queda libre para añadir un parámetro de configuración del theremin. En esta implementación, mover el mouse arriba o abajo implica un cambio de volumen, siendo el punto más alto el volumen máximo. Veamos cómo se implementa el theremin:

Primero, se dibuja una línea vertical en la posición del cursor utilizando la función de p5 `line(mouseX, 0, mouseX, height)`. Esta línea es el cursor mencionado anteriormente.

A continuación, se mapea la posición del ratón a una nota en la escala seleccionada basándose en la región en la que se encuentra el cursor. La posición x del ratón se divide por el ancho de la región (`regionWidth`) para determinar la región actual. Esto proporciona un índice para acceder a la nota correspondiente en `scaleNotes`.

```
// Theremin mode
if (interactionMode === "theremin") {

  // Draw a vertical line at the cursor position
  stroke(200);
  line(mouseX, 0, mouseX, height);
}
```



Se utiliza la posición y del ratón para controlar el volumen. Cuanto más alta sea la posición y, más bajo será el volumen. Se utiliza la función de p5 map para mapear la posición y del ratón desde el rango de altura de la ventana a un rango de valores de volumen.

```
// Map the mouse position to a note in the scale based on the region
const regionIndex = floor(mouseX / regionWidth);
// Higher mouse y=> Higher volume
let volume = map(mouseY, height, 0, -48, 0);

const note = scaleNotes[regionIndex];
const frequency = Tone.Frequency(note).toFrequency();

// Update current note based on mouse position
currentNote = frequency;
```

A continuación, se asigna la nota correspondiente a la frecuencia del sintetizador de theremin utilizando `Tone.Frequency(note).toFrequency()`. Esto convierte la nota en su frecuencia correspondiente, utilizando el sintetizador creado en `Tone`. Veremos cómo funciona el sintetizador en el archivo de sonido.

```
if (mouseIsPressed) {
  thereminSynth.triggerAttackRelease(currentNote);
  thereminSynth.volume.value = volume;
  isPlaying = true;
} else {
  thereminSynth.triggerRelease(Tone.now());
  isPlaying = false;
}
```

## Metrónomo

La función `drawMetronome()` dibuja un metrónomo rojo en el centro del lienzo que cambia de color en cada tiempo, de 1 a 4, para indicar la secuencia del beat.

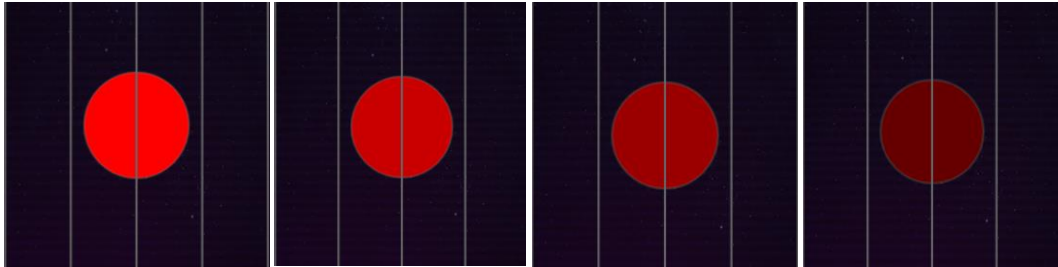


Ilustración 19 - Transición del metrónomo del primer al cuarto tiempo

```
function drawMetronome() {  
  
    let currentPosition = Tone.Transport.position.split(":");  
    let currentBeat = parseInt(currentPosition[1]);  
    beatNum = currentBeat;  
  
    let metronomeSize = 50;  
    switch (beatNum) {  
        case 0:  
            fill(255, 0, 0); //red color for the first beat  
            break;  
        case 1:  
            fill(202, 0, 0);  
            break;  
        case 2:  
            fill(155, 0, 0);  
            break;  
        default:  
            fill(102, 0, 0);  
    }  
  
    ellipse(width / 2, height / 2, metronomeSize);  
}
```

la función `windowResized()` se encarga de redimensionar el lienzo cuando cambia el tamaño de la ventana, haciendo la aplicación responsiva

```
function windowResized() {  
    resizeCanvas(windowWidth, windowHeight);  
}
```

El evento `window.onload` se utiliza para ocultar inicialmente el marco del Sound Design Panel y agregar un evento de clic al botón para mostrar u ocultar el marco.

```
// Add an event listener to the reset button  
const resetButton = document.getElementById('reset-button');
```

```

resetButton.addEventListener('click', function(event) {
  event.stopPropagation(); // prevent event from reaching the canvas
  whiteDots = []; // clear out the whiteDots array
  octaveGenerators = []; // clear out the octaveGenerators array
  Tone.Transport.stop();
  Tone.Transport.position = 0;
  Tone.Transport.start();
});
}

```

El evento `window.onload` adicional para ocultar y mostrar el marco del sintetizador y el evento de clic en el botón de reinicio para borrar los puntos blancos y reiniciar el transporte de `Tone.js`.

```

window.onload = function () {
  let synthFrame = document.getElementById('synth-iframe');
  synthFrame.style.visibility = 'hidden';
  let button = document.getElementById('collapse-btn')
  button.addEventListener('click', function () {
    event.stopPropagation();
    let synthFrame = document.getElementById('synth-iframe');
    if (synthFrame.style.visibility !== 'hidden') {
      synthFrame.style.visibility = 'hidden';
    } else {
      synthFrame.style.visibility = 'visible';
    }
  });
}

```

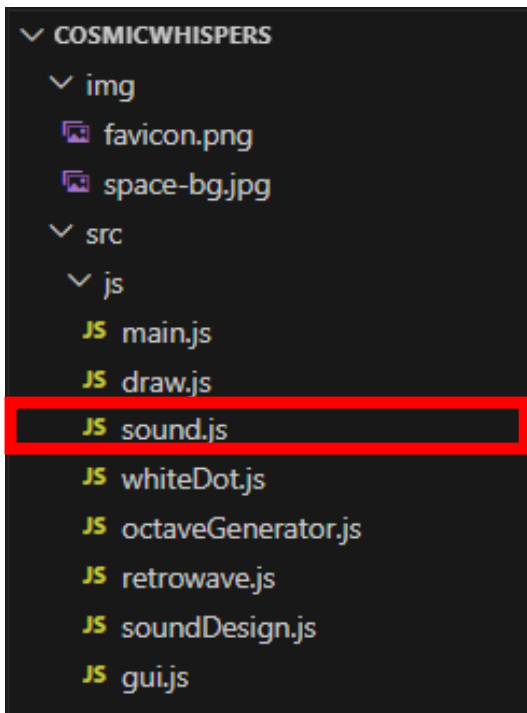
La función `updateBackgroundSpeed` se utiliza en conjunción con el fondo interactivo definido en el archivo `retrowave.js` para acelerar la animación según el número de notas que hay en pantalla.

```

function updateBackgroundSpeed() {
  // Update the speed variable based on the number of dots
  backgroundSpeed = map(numDots, 0, maxDots, 0.1, 10);
}

```

### 4.3. Sound



En este archivo se definen los sintetizadores de Tone.js, sus efectos, la lógica rítmica de la aplicación y las opciones internas relacionadas con el sonido de la aplicación.

Primero, se define el sintetizador para las notas, con los parámetros seleccionados para conseguir un timbre que suena como una Kalimba:



Ilustración 20 - Kalimba siendo tocada

En este código de puro Tone.js, se define un PolySynth a través de la instanciación de un FMSynth, donde podemos definir sus propiedades a través del oscilador, los envelopes ADSR y la modulación para obtener el timbre deseado. En este caso conseguimos que suene como una Kalimba.

```
// Note Synth
let polySynth = new Tone.PolySynth(Tone.FMSynth, {
  "harmonicity": 8,
  "modulationIndex": 2,
  "oscillator": {
    "type": "sine"
  },
},
"envelope": {
  "attack": 0.01,
  "decay": 2,
  "sustain": 0.1,
  "release": 1
},
"modulation": {
  "type": "square"
},
},
"modulationEnvelope": {
  "attack": 0.002,
  "decay": 0.2,
  "sustain": 0,
  "release": 0.2
}
}).toDestination();
```

A este sintetizador le podemos añadir módulos para modificar el sonido y conseguir efectos interesantes. Muchas de sus propiedades se han mantenido por defecto, otras se configuran a base de prueba y error hasta que se consigue un sonido que es agradable para el diseñador de sonido.

- Compresor: Aumenta los sonidos bajos de volumen y reduce los sonidos altos. De esta forma se consigue un sonido más limpio y uniforme.

```
// Compressor to clean the audio peaks
const compressor = new Tone.Compressor({
  attack: 0.003,
  release: 0.25,
  threshold: -24,
  ratio: 4,
  knee: 30,
}).toDestination();
```

- Reverb: Efecto para emular un sonido que se reproduce en un espacio amplio. Se utiliza ampliamente en la música electrónica para darle más cuerpo al sonido y que suene menos sintético. El parámetro “wet” indica cómo de intenso es el reverb.

```
// Reverb
const reverb = new Tone.Reverb({
  decay: 2,
  preDelay: 0.01,
  wet: 0.5
});
```

- Filtros high-pass / low-pass: Se utilizan para filtrar las frecuencias muy agudas o muy graves, ya que pueden crear efectos indeseados sobre los altavoces que distorsionen cómo se percibe el sonido.

```
// Clean low frequencies
const lowpassFilter = new Tone.Filter({
  type: "lowpass",
  frequency: 50,
  Q: 1
});

// Clean high frequencies
const highpassFilter = new Tone.Filter({
  type: "highpass",
  frequency: 1000,
  Q: 100
});
```

- Eco: Hace que la nota se repita después de haber sido reproducida con menos volumen, también conocido como FeedbackDelay. Ayuda mucho a mejorar la estética sonora de la aplicación.

```
// Echo
const echo = new Tone.FeedbackDelay('8n', 0.25).toDestination();
```

Y los conectamos fácilmente al sintetizador:

```
polySynth
  .connect(compressor)
  .connect(reverb)
  .connect(echo)
  .connect(lowpassFilter)
  .connect(highpassFilter)
```

Para el sintetizador del Theremin, lo configuramos con la misma lógica, pero usando un Tone.Synth, más simple:

```
// Theremin Synthesizer
let thereminSynth;
thereminSynth = new Tone.Synth({
  oscillator: {
    type: "sine", // Use a sawtooth wave for the oscillator
    detune: 10, // Detune the oscillator slightly for a richer sound
  },
  envelope: {
    attack: 0.2, // Increase the attack time for a softer attack
    decay: 0.2, // Decrease the decay time for a sharper decay
    sustain: 0.5, // Decrease the sustain level for a shorter sustain
    release: 15, // Increase the release time for a longer fade out
  },
  volume: -16,
}).toDestination();

const thereminEcho = new Tone.FeedbackDelay('8n', 0.25).toDestination();

thereminSynth
  .connect(thereminEcho);
```

Se definen las opciones por defecto del programa, que van a poder ser modificadas en el GUI:

```
// Default settings
const settings = {
  scale: "Mongolian",
  speed: 1,
  lockNodes: true,
  octaveOffset: 0,
};
```

Las escalas disponibles para utilizar. La escala random se define por una lista numérica aleatoria cada vez que es elegida

```
const scales = {
  Major: [0, 2, 4, 5, 7, 9, 11, 12, 14, 16, 17, 19, 21, 23],
  Natural_Minor: [0, 2, 3, 5, 7, 8, 10, 12, 14, 15, 17, 19, 20, 22],
  Harmonic_Minor: [0, 2, 3, 5, 7, 8, 11, 12, 14, 15, 17, 19, 20, 23],
  Melodic_Minor: [0, 2, 3, 5, 7, 9, 11, 12, 14, 15, 17, 19, 21, 23],
  Mixolydian: [0, 2, 4, 5, 7, 9, 10, 12, 14, 16, 17, 19, 21, 22],
  Phrygian: [0, 1, 3, 5, 7, 8, 10, 12, 13, 15, 17, 19, 20, 22],
  Lydian: [0, 2, 4, 6, 7, 9, 11, 12, 14, 16, 18, 19, 21, 23],
  Locrian: [0, 1, 3, 5, 6, 8, 10, 12, 13, 15, 17, 18, 20, 22],
  Mongolian: [0, 2, 5, 7, 9, 12, 14, 17, 19, 21, 24, 26, 29, 31],
  Pentatonic_Major: [0, 2, 4, 7, 9, 12, 14, 16, 19, 21, 24, 26, 28, 31],
  Pentatonic_Minor: [0, 3, 5, 7, 10, 12, 15, 17, 19, 22, 24, 27, 29, 31],
  Overtone: [0, 4, 7, 10, 12, 14, 16, 19, 22, 24, 26, 28, 31, 34, 36, 38,
40, 43, 46, 48, 50],
  Blues: [0, 3, 5, 6, 7, 10, 12, 15, 17, 18, 19, 22, 24, 27],
  Chromatic: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31],
  Whole_Tone: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28,
30],
  Random: getRandomIntervals(),
};

function getRandomIntervals() {
  const numNotes = Math.floor(Math.random() * 10) + 7; // Random number
of notes between 10 and 17
  const intervals = [0]; // The first note is always 0

  for (let i = 1; i < numNotes; i++) {
    const prevInterval = intervals[i - 1];
    const newInterval = prevInterval + Math.floor(Math.random() * 4) + 1;
// Random interval between 1 and 3 semitones
    intervals.push(newInterval);
  }

  return intervals;
}
```

La función `onBeat()` es una de las principales responsables de la lógica de la aplicación. Se ejecuta en cada golpe del metrónomo, y se encarga de hacer sonar a cada nota que está en la región correspondiente del generador para que reproduzca su sonido. Si dos notas iguales suenan a la vez, solo una de ellas suena, para evitar cálculos redundantes en el PolySynth de Tone.js.

```
function onBeat(time) {
  // console.log(currentEighthNote);
  let currentPosition = Tone.Transport.position.split(":");
  let playingNotes = new Set();

  currentEighthNote = getEighthNoteIndex(currentPosition)
  // Loop through each octave generator
  for (const generator of octaveGenerators) {
    // Loop through each white dot in the current generator
    for (const whiteDot of generator.whiteDots) {
      if (whiteDot.beatsToPlay[currentEighthNote]) {
        whiteDot.soundPlayed();
        whiteDot.beatsToPlay = new Array(8).fill(false);

        // Calculate the note with the octave offset
        const noteWithOffset =
Tone.Frequency(whiteDot.scaleNote).transpose(12 *
generator.octaveOffset).toNote();

        // Create a unique key for the playingNotes set
        const noteKey =
`${whiteDot.noteIndex}_${generator.octaveOffset}`;

        // Check if the note is already playing, in that case it wont
play
        if (!playingNotes.has(noteKey)) {
          playingNotes.add(noteKey);

          polySynth.triggerAttackRelease(noteWithOffset, "16n", time,
undefined, () => {
            playingNotes.delete(noteKey);
          });
        }
      }
    }
  }
  // If the beat counter gets stuck, reset the Transport
  if (previousEighthNote === currentEighthNote) {
    resetTransport();
    previousEighthNote = -1;
    return;
  } else {
    previousEighthNote = currentEighthNote;
  }
}
```



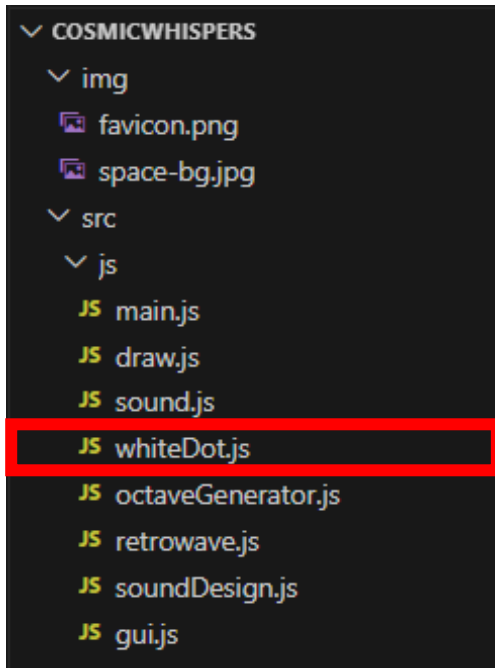
También cuenta con funciones para resetear el metrónomo para evitar el overflow, que se llama cuando el usuario pulsa el botón Reset.

```
function resetTransport() {  
  Tone.Transport.stop();  
  Tone.Transport.position = 0;  
  Tone.Transport.start();  
}
```

Finalmente, un Event Listener que escucha las configuraciones que se realizan en el panel de Sound Design para enviarlas al archivo correspondiente y modificar el sintetizador como se ha indicado.

```
window.addEventListener('message', (event) => {  
  const data = event.data;  
  
  if (data.type === 'updatedSynth') {  
    activeSynth.volume.value = data.options.volume;  
    activeSynth.oscillator.type = data.options.oscillatorType;  
    activeSynth.envelope.attack = data.options.envelope.attack;  
    activeSynth.envelope.decay = data.options.envelope.decay;  
    activeSynth.envelope.sustain = data.options.envelope.sustain;  
    activeSynth.envelope.release = data.options.envelope.release;  
  }  
});  
  
const synthIframe = document.getElementById('synth-iframe');  
  
// Wait for the iframe content to load  
synthIframe.addEventListener('load', () => {  
  // Send the activeSynthData object as a message to the iframe  
  synthIframe.contentWindow.postMessage(activeSynthData, '*');  
});
```

#### 4.4. whiteDots



En este archivo se implementa la clase `whiteDot`. Los `whiteDots` son las notas creadas en el canvas. Aquí se definen sus propiedades y métodos, con las transformaciones subyacentes correspondientes.

Se debe seleccionar una nota de la escala actual pinchando en uno de los círculos de la fila inferior:

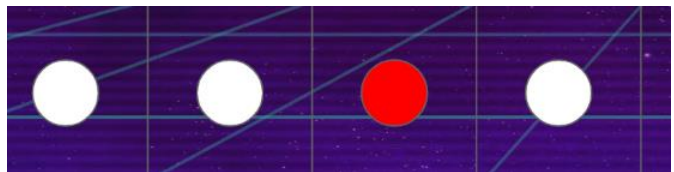


Ilustración 21 – Menú de escalas (detalle)

Al hacer clic en cualquier parte del canvas, se crea un nodo asociado a la nota seleccionada. Cada nota tiene también un color fijo asociado y un tamaño aleatorio dentro de un rango. Para eliminar el nodo, se pincha sobre él.

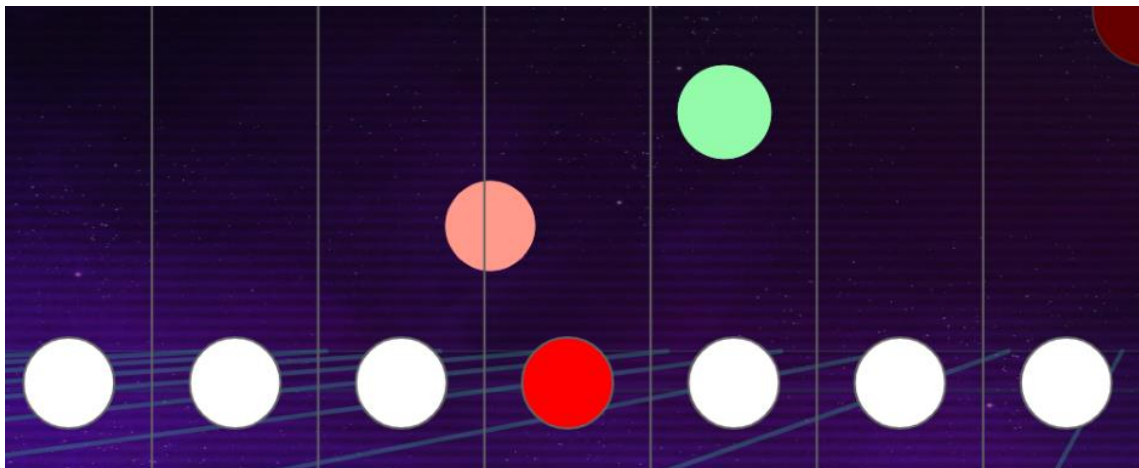


Ilustración 22 – Notas creadas

Además, los nodos se mueven por el espacio, y cuando llegan al extremo de la pantalla, reaparecen en el otro extremo. De esta manera no modifican su velocidad. Se puede controlar su velocidad actual y si se mueven o no en el panel de control, que veremos más adelante.

El constructor de la clase WhiteDot recibe las coordenadas x e y del mouse (que se lee con p5 en draw.js), así como el índice de la nota asociada al punto. El tamaño, la velocidad y el ángulo se inicializan con valores aleatorios. Además, se configuran atributos relacionados con la animación y la reproducción de sonidos.

```
class WhiteDot {
  constructor(x, y, noteIndex) {
    // Position attributes
    this.x = x;
    this.y = y;

    // Original size and size storage
    this.size = random(30, 50);
    this.originalSize = this.size;

    // Animation
    this.animationDuration = 200;
    this.animationStartTime = null;

    // Movement
    this.speed = random(0.2, settings.speed);
    this.angle = random(TWO_PI);

    // Note
    this.playedNote = false;
    this.isPlaying = false;
    this.scale = scales[settings.scale];
    this.noteIndex = noteIndex;
    this.midiNote = root + this.scale[this.noteIndex];
    this.scaleNote = Tone.Frequency(this.midiNote, 'midi');

    // Rhythm
    this.beatsToPlay = new Array(8).fill(false);
    this.creationTime = Tone.Transport.seconds;
```

La función `update()` se encarga de actualizar la posición del punto en cada fotograma de la animación. Si la configuración "lockNodes" está desactivada, el punto se mueve a lo largo de su trayectoria definida por su velocidad y ángulo. Además, se implementa la funcionalidad de envolver la posición del punto alrededor de la pantalla en caso de que salga de los límites.

```
// update function to change position of white dot
update() {
  if (!settings.lockNodes) {
    this.x += this.speed * cos(this.angle);
    this.y += this.speed * sin(this.angle);

    // Wrap the position of the dot around the screen
    if (this.x < 0) {
      this.x = width;
    } else if (this.x > width) {
      this.x = 0;
    }

    if (this.y < 0) {
      this.y = height;
    } else if (this.y > height) {
      this.y = 0;
    }
  }
}
```

La función `calculateTriggerTime()` calcula el tiempo de activación para la reproducción de sonidos del punto en relación con los generadores de octavas. Si la configuración `"showOctaveGenerator"` está activada y existen generadores de octavas, se calcula la distancia mínima entre el punto y los generadores. Luego, se normaliza esta distancia en relación con la distancia máxima posible y se multiplica por el tiempo de un cuarto de nota para obtener el tiempo de activación del punto.

```
calculateTriggerTime(octaveGenerators) {
  if (!settings.showOctaveGenerator || octaveGenerators.length === 0)
return;
  let minDistance = Infinity;
  for (const generator of octaveGenerators) {
    const distance = dist(this.x, this.y, generator.x, generator.y);
    minDistance = min(minDistance, distance);
  }
  const maxDistance = dist(0, 0, width, height);
  const normalizedDistance = minDistance / maxDistance;
    this.triggerTime = Tone.Time("1n").toSeconds() *
normalizedDistance;
}
```

La función `isMouseOver()` verifica si el ratón se encuentra sobre el punto mediante el cálculo de la distancia entre las coordenadas del punto y las coordenadas del ratón, para determinar si se solicita el icono de `MouseOver` y eliminar el nodo si se pincha con el ratón.

```
// Mouse over for interacting
isMouseOver() {
  const distanceToMouse = dist(this.x, this.y, mouseX, mouseY);
  return distanceToMouse < this.size / 2;
}
```

La función `updateNoteIndex()` actualiza el índice de la nota del punto en función de la escala proporcionada como argumento. Esto afecta a la nota MIDI y a la nota musical representada por el punto.

```
updateNoteIndex(scale) {
  this.midiNote = root + scale[this.noteIndex];
  this.scaleNote = Tone.Frequency(this.midiNote, 'midi');
}
```

La función `display()` se encarga de dibujar el punto en la pantalla. El tamaño del punto puede animarse si el punto se está reproduciendo actualmente. En ese caso, el tamaño se incrementa durante un tiempo determinado y luego vuelve a su tamaño original.

```
// display function to draw the white dot
display() {
  // Get the color index based on the note index
  let colorIndex = this.noteIndex % colors.length;
  fill(colors[colorIndex]);

  // Size animation when note played
  let displaySize = this.size;
  if (this.isPlaying) {
    if (this.animationStartTime === null) {
      this.animationStartTime = millis();
    }
    const elapsedTime = millis() - this.animationStartTime;
    if (elapsedTime < this.animationDuration) {
      displaySize *= 1.2;
    } else {
      displaySize = this.originalSize;
      this.isPlaying = false;
      this.animationStartTime = null;
    }
  } else {
    displaySize = this.originalSize;
  }

  noStroke();
  ellipse(this.x, this.y, displaySize);
}
```

La función `soundPlayed()` se llama cuando se reproduce el sonido asociado al punto. Establece el estado de reproducción y el tiempo de inicio de la animación de tamaño.

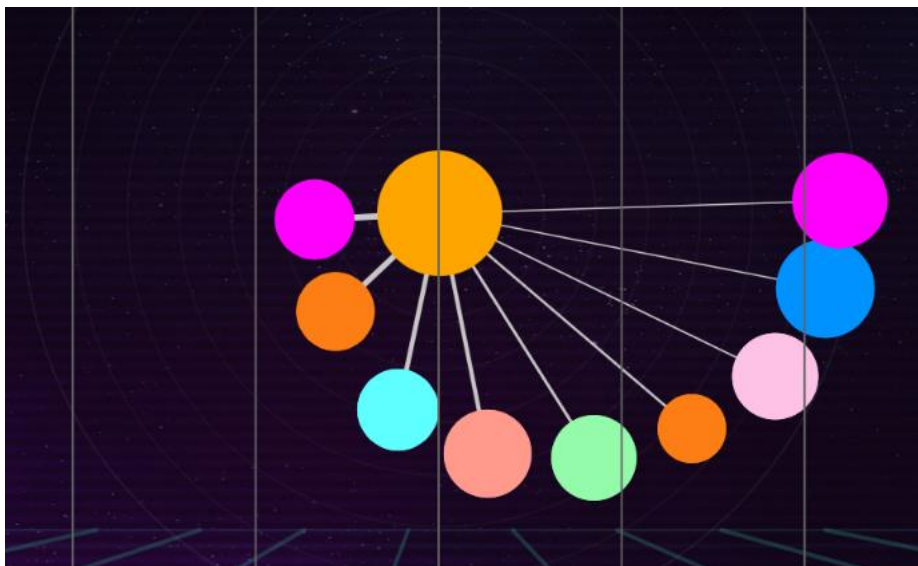
```
// Call this method when the sound is played
soundPlayed() {
  this.isPlaying = true;
  this.animationStartTime = millis();
}
```

La función `checkDistance()` verifica la distancia entre los generadores y los puntos y dibuja líneas si están lo suficientemente cerca. El color y el grosor de las líneas se basan en la distancia entre ellos.

```
checkDistance(generators, distanceThreshold, lineColor) {
  for (let generator of generators) {
    let distance = dist(this.x, this.y, generator.x, generator.y);
    if (distance < distanceThreshold) {
      push(); // Save the current drawing state

      stroke(lineColor);
      // Map distance to stroke weight: maximum weight at 0 distance,
      // minimum weight at the distanceThreshold
      let weight = map(distance, 0, distanceThreshold, 4, 0.5);
      strokeWeight(weight);
      line(this.x, this.y, generator.x, generator.y);
      pop(); // Restore the previous drawing state
    }
  }
}
```

En el ejemplo inferior se muestra cómo se dibujan las líneas de mayor grosor para los puntos más cercanos.



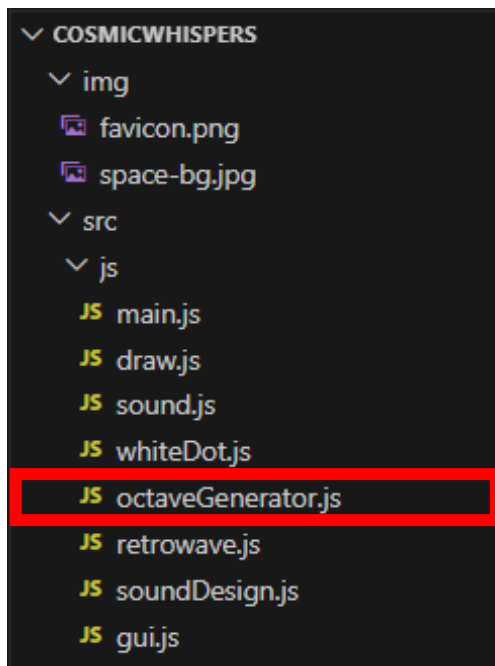
*Ilustración 23 – Líneas de unión a los generadores, variables según la distancia*

La función `updateWhiteDotsScale()` actualiza el índice de la nota de todos los puntos en función de la escala seleccionada en la configuración.

```
function updateWhiteDotsScale() {
  const scale = scales[settings.scale];
  for (let whiteDot of whiteDots) {
    whiteDot.updateNoteIndex(scale);
  }
}
```

```
}
```

## 4.5. OctaveGenerators



Los OctaveGenerators son los generadores de música, que emiten órbitas periódicamente según indicados por el clock definido en Tone.js.

Análogamente a los whiteDots, se han definido como su propia clase, y se instancian en el canvas cuando son llamados por el draw().js

Para crear un octaveGenerator, hay 3 octavas disponibles para elegir: -1, 0 y 1. La idea es que el usuario pueda crear diferentes melodías para el bajo, la armonía y la melodía principal, respectivamente. Se eligen en el panel de círculos de la izquierda, y cada uno tiene un color asociado al crearse. En la imagen inferior se pueden apreciar los 3 generadores distintos, así como el selector de octavas:

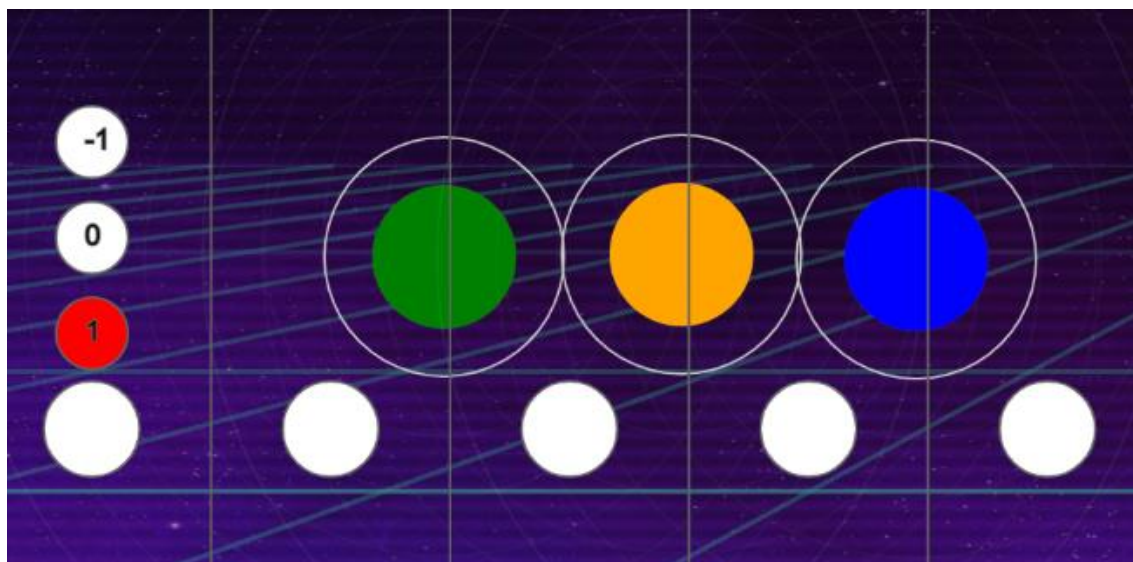


Ilustración 24 – Generadores, con sus diferentes aspectos según la octava que representen

La función de los generadores, es que cuando su órbita blanca pasa por encima de uno de los nodos, hace reproducir su nota correspondiente, en la octava del generador que sea. No hay límite de notas para los generadores.



Puesto que el metrónomo de Tone.js está configurado a 8 beats, las órbitas tienen 8 posiciones, y por tanto los nodos pueden estar en 8 regiones diferentes.

El constructor de la clase OctaveGenerator recibe las coordenadas x e y del generador, de nuevo, indicadas por la lectura del mouseX y mouseY de p5 en la función draw(), así como un desplazamiento de octava opcional. Se inicializan varios atributos, como el tamaño, el rango, el color y el número de anillos del generador.

```
class OctaveGenerator {
  constructor(x, y, octaveOffset = settings.octaveOffset) {
    this.x = x;
    this.y = y;
    this.octaveOffset = octaveOffset;

    this.size = 60;
    this.range = 200;
    this.color = 'orange';
    this.rings = 8;
    this.ringColor = 200;

    this.startingRingSize = 0;
    this.endingRingSize = this.range;

    this.whiteDots = [];
    this.ringWidth = this.range / this.rings;

    if (this.octaveOffset === 0) {
      this.color = 'orange';
    } else if (this.octaveOffset === 1) {
      this.color = 'blue';
    } else if (this.octaveOffset === -1) {
      this.color = 'green';
    }
  }
}
```

El método labelWhiteDots() se utiliza para etiquetar los nodos (WhiteDot) asociados al generador. Recorre todos los nodos y calcula la distancia entre cada uno y el generador. Luego, asigna una etiqueta a los nodos en función del anillo en el que se encuentren. Cada anillo representa un tiempo de reproducción en la animación.

```
labelWhiteDots() {
  for (const whiteDot of this.whiteDots) {
    const distance = dist(this.x, this.y, whiteDot.x, whiteDot.y);
    const ringIndex = Math.floor(distance / this.ringWidth);
    if (ringIndex >= 0 && ringIndex < 8) {
      whiteDot.beatsToPlay[ringIndex] = true;
    }
  }
}
```

El método `isWhiteDotInRange(whiteDot)` verifica si un nodo está dentro del rango del generador. Calcula la distancia entre el generador y el nodo y compara esa distancia con el rango del generador. Devuelve `true` si el nodo está dentro del rango y `false` en caso contrario. La función `dist()` pertenece a p5

```
// Add a method to check if a WhiteDot is within the range
isWhiteDotInRange(whiteDot) {
  const distance = dist(this.x, this.y, whiteDot.x, whiteDot.y);
  return distance <= this.range;
}
```

El método `updateWhiteDotsInRange(whiteDots)` se utiliza para actualizar los puntos blancos dentro del rango del generador. Borra el arreglo de puntos blancos existente en el generador y luego itera sobre el arreglo de puntos blancos proporcionado como entrada. Verifica si cada punto blanco está dentro del rango del generador utilizando el método `isWhiteDotInRange()`. Si el punto blanco está dentro del rango, se agrega al arreglo de puntos blancos del generador.

```
updateWhiteDotsInRange(whiteDots) {
  // Clear the current whiteDots array
  this.whiteDots = [];
  // Iterate through the input whiteDots array
  for (const whiteDot of whiteDots) {
    // Check if the whiteDot is within the range of the generator
    if (this.isWhiteDotInRange(whiteDot)) {
      // Add the whiteDot to the generator's whiteDots array
      this.whiteDots.push(whiteDot);
    }
  }
}
```

El método `drawRings(currentEighthNote)` se encarga de dibujar los anillos alrededor del generador. Itera a través del número de anillos del generador y dibuja cada anillo utilizando la función p5 `ellipse()`. El anillo actual se resalta ajustando la opacidad del dibujo.

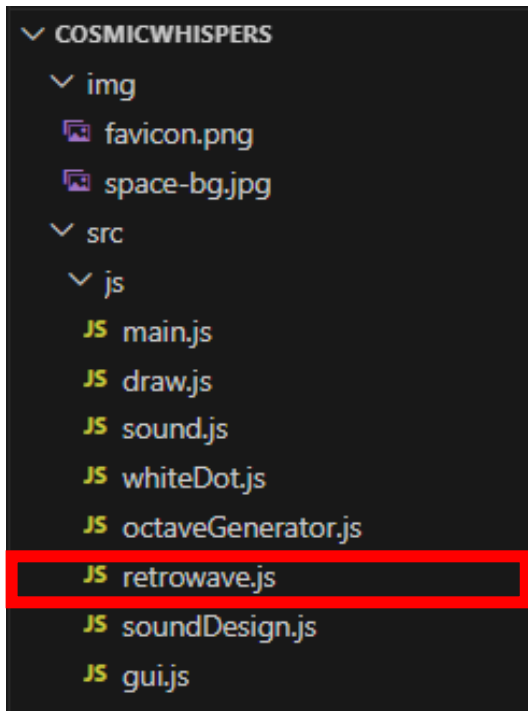
```
drawRings(currentEighthNote) {
  for (let i = 1; i <= this.rings; i++) {
    const opacity = (i === currentEighthNote+1) ? 255 : 10; // Set
    opacity to 255 if it's the current beat, otherwise set it to 50

    stroke(this.ringColor, opacity);
    noFill();
    let ringRadius = i * (this.range/this.rings); // Adjust this value
    to set the distance between rings
    ellipse(this.x, this.y, ringRadius * 2, ringRadius * 2); } }
```

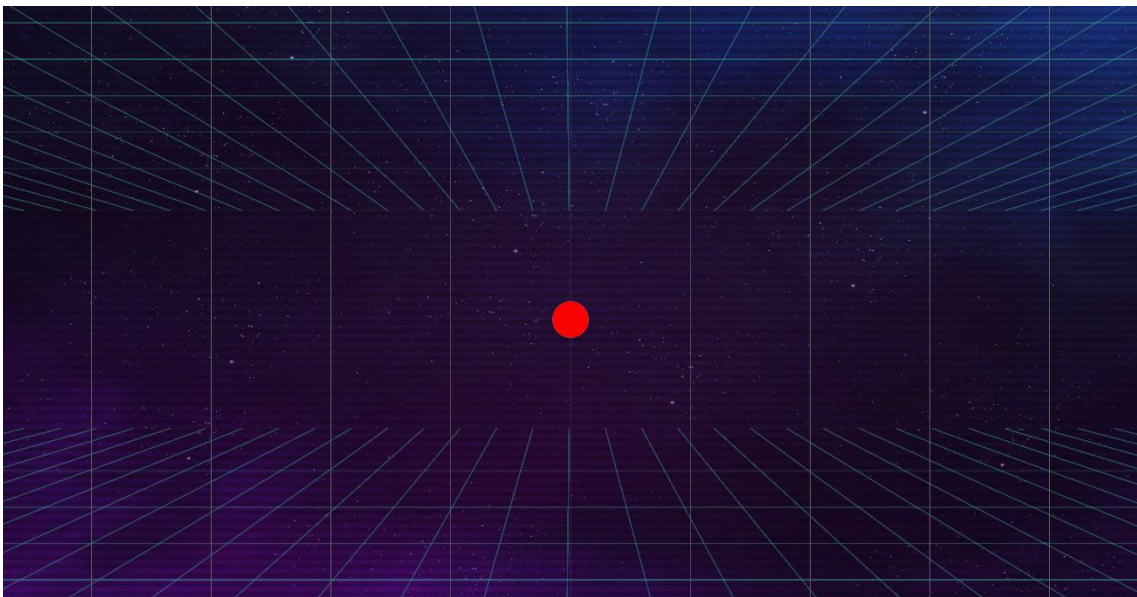
El método `display()` se utiliza para dibujar el generador en la pantalla. Llama al método `drawRings()` para dibujar los anillos alrededor del generador y luego dibuja el círculo principal del generador y el círculo que representa su rango. Se rellena con la función `fill` de p5 con el color definido en sus atributos de clase.

```
// Update the display method to draw the range circle
display() {
  this.drawRings(currentEighthNote);
  fill(this.color);
  ellipse(this.x, this.y, this.size);
  noFill();
  ellipse(this.x, this.y, this.range * 2);
}
}
```

#### 4.6. Background interactivo



Para explotar más las capacidades de p5.js, se ha utilizado como fondo de la aplicación un dibujo interactivo inspirado en la estética retrowave de los años 80. El fondo se conforma de una composición de la animación de la cuadrícula neón y una imagen fija (space\_bg.jpg) que es lo que se aprecia detrás de las líneas. Ambas con opacidad inferior al 100% para que se puedan componer adecuadamente.



*Ilustración 25 - Fondo de la aplicación, compuesto por la imagen fija y el fondo interactivo retrowave*

Además, el fondo crea la ilusión de estar moviéndose cada vez más rápido a medida que se van creando más nodos en pantalla. Esta ilusión de perspectiva se crea mediante la angulación y el gradiente adecuados, que veremos en detalle. Se ha creado un archivo `retrowave.js` con el código exclusivamente para la animación del fondo:

En la parte superior del archivo se definen varias variables que controlan el aspecto y el movimiento del fondo, como el desplazamiento, el espaciado, la perspectiva, la velocidad de fondo, etc. Estas variables se pueden modificar para obtener efectos interesantes en la animación.

```
let gridOffset = 0;

// Space between the two planes
let spaceMargin=150

// Space between each gridline
let spacing = 50;

// Angle to create perspective higher -> more in the ground
let flattenPerspective = 60;

// Animation speed
let backgroundSpeed = 0.1;
```

La función `setupBackground()` se encarga de crear un degradado que se utilizará como fondo del lienzo.

```
function setupBackground() {
  createGradient();
}
```

La función `updateSpeed()` actualiza la velocidad de fondo y la perspectiva en función del valor de un control deslizante (slider) llamado `speedSlider` en la versión original. Veremos que dejar este valor configurable como un slider no afecta a la versión futura si la modificamos por el valor de la cantidad de los nodos, siempre que no lleguemos a crear el objeto de slider.

```
function updateSpeed() {
  backgroundSpeed = speedSlider.value();    flattenPerspective=
speedSlider.value()*100+50
  spaceMargin= (speedSlider.value()*(-15) + 170
}
```

La función `createGradient()` crea el degradado utilizando un objeto p5 `createGraphics` y lo almacena en una variable llamada `gradient`. El degradado se genera aplicando una interpolación lineal entre dos colores en función de la posición vertical en el lienzo. En este caso, crea un degradado entre negro opaco y negro transparente, para hacer desaparecer las líneas cuando llegan al final. Estas funciones de p5 hacen el cálculo de la interpolación por nosotros.

```
function createGradient() {
  gradient = createGraphics(width, height);
  gradient.clear();
  let gHorizon = color(0, 0, 0, 150);
  let gBottom = color(0, 0, 0, 0);

  for (let i = 0; i < height; i++) {
    gradient.stroke(lerpColor(gHorizon, gBottom, i / height));
    gradient.line(0, i, width, i);
  }
}
```

La función `drawBackground()` es la principal función responsable de dibujar el fondo interactivo. Primero se establece el punto de fuga y la línea del horizonte basándose en el tamaño del lienzo y el margen espacial. Luego, se dibujan líneas verticales en perspectiva que convergen hacia el punto de fuga. A continuación, se dibujan líneas horizontales en capas, comenzando desde la línea del horizonte y avanzando hacia abajo, creando un efecto de cuadrícula en perspectiva. Por último, se dibuja una línea de horizonte, que funciona como la línea más alejada. El archivo también contiene un llamado a la función `drawInverted()` y una imagen del degradado para completar el dibujo del fondo.

```
function drawBackground() {
  const vanishingPoint = height/2;
  let horizon=vanishingPoint;
  background(32, 13, 58,50);
  horizon=vanishingPoint+spaceMargin;

  // Draw vertical lines in perspective
  for (let x = -spacing; x <= width + spacing; x += spacing) {
    // Increase the spacing multiply to increase perspective
    let startX = map(x, -spacing, width + spacing, -spacing *
flattenPerspective, width + spacing * flattenPerspective);
    let endX = map(x, -spacing, width + spacing, -spacing, width +
spacing);
    stroke(70, 206, 189, 100);
    strokeWeight(2);
    line(endX, horizon, startX, height);
  }
}
```

```

// Draw horizontal lines
gridOffset = (gridOffset + backgroundSpeed) % spacing;
for (let y = horizon + gridOffset; y <= height; y += spacing) {
  let opacity = map(y, horizon, height, 0, 255);
  stroke(70, 206, 189, opacity);
  strokeWeight(2);
  line(0, y, width, y);
}
// Draw horizon line
stroke(70, 206, 189, 50);
strokeWeight(1);
line(0, horizon, width, horizon);
// Draw the original image
drawInverted();
image(gradient, 0, 0);
}

```

La función `drawInverted()` es similar a `drawBackground()`, pero hace el mismo dibujo al revés, por encima del centro, en perspectiva invertida, para crear un dibujo simétrico.

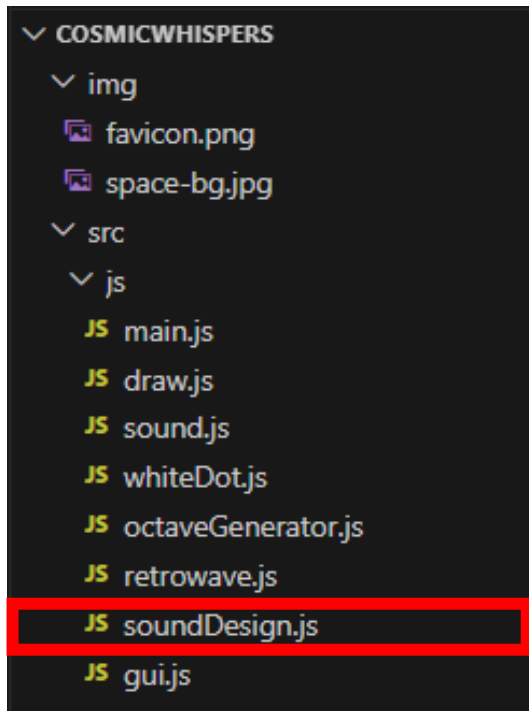
```

function drawInverted() {
  const vanishingPoint = height/2;
  let horizon=vanishingPoint;
  horizon=vanishingPoint-spaceMargin;

  // Draw inverted vertical lines in perspective
  for (let x = -spacing; x <= width + spacing; x += spacing) {
    // Increase the spacing multiply to increase perspective
    let startX = map(x, -spacing, width + spacing, -spacing *
flattenPerspective, width + spacing * flattenPerspective);
    let endX = map(x, -spacing, width + spacing, -spacing, width +
spacing);
    stroke(70, 206, 189, 100);
    strokeWeight(2);
    line(endX, horizon, startX, 0);
  }
  // Draw inverted horizontal lines
  let invertedGridOffset = (gridOffset - backgroundSpeed) % spacing;
  for (let y = horizon - invertedGridOffset; y >= 0; y -= spacing) {
    let opacity = map(y, 0, horizon, 255, 0);
    stroke(70, 206, 189, opacity);
    strokeWeight(2);
    line(0, y, width, y);
  }
  // Draw inverted horizon line
  stroke(70, 206, 189, 50);
  strokeWeight(1);
  line(0, horizon, width, horizon);
}

```

## 4.7. Sound Design Panel



Para explotar las capacidades de Tone.js, se ha creado un iframe HTML con un panel de diseño de sonido, que permite modificar las propiedades del sintetizador del theremin en acción.

Los parámetros a modificar son los mismos que hemos visto al diseñar el synth, pero además ahora también se ha creado manualmente una referencia visual de la curva ADSR y un botón para probar el sonido creado. Además, se ha utilizado un fuerte estilizado CSS para que case con la estética del diseño.

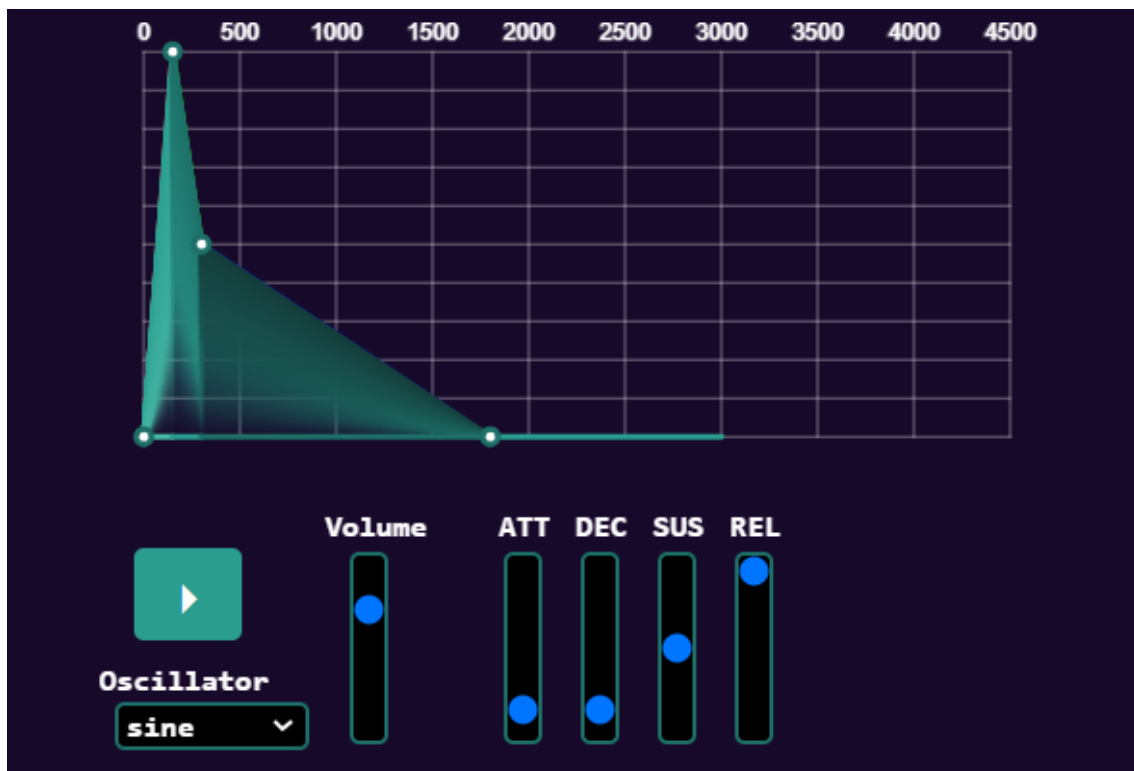


Ilustración 26 – Panel de control de Sound Design



La función `setup()` se encarga de configurar el lienzo, establecer la velocidad de fotogramas y crear los controles deslizantes, etiquetas y botones necesarios para el panel de diseño de sonido. Los controles deslizantes controlan diferentes parámetros del sintetizador, como la frecuencia, el volumen y los valores de la envolvente ADSR (Attack, Decay, Sustain, Release). Este código es muy repetitivo, así que solo es necesario mostrar cómo se configura uno de los sliders:

```
function setup() {
  createCanvas(600,400);
  frameRate(30);

  // Create a Tone.js synth
  testSynth = new Tone.Synth().toDestination();
  wave = new Tone.Waveform(1024);

  // Set the initial position for the controls
  let controlX = width / 10 ;
  let controlY =height - height /4 ;
  let controlSpacing = 40;

  // Create a play button
  playButton = createButton('▶');
  playButton.position(controlX+10, controlY-20);
  playButton.mousePressed(playSound);
  playButton.addClass('play-button');

  // Create labels and sliders for frequency, volume, and oscillator type

  typeLabel = createP('Oscillator');
  typeLabel.position(controlX-140, controlY+80);
  typeLabel.addClass('slider-label');

  typeSelect = createSelect();
  typeSelect.addClass('custom-select');
  typeSelect.option('sine');
  typeSelect.option('square');
  typeSelect.option('triangle');
  typeSelect.option('sawtooth');
  typeSelect.position(controlX, controlY + 60);

  controlX += controlSpacing*2;

  vollabel = createP('Volume');
  vollabel.position(controlX - 50, controlY);
  vollabel.addClass('slider-label');

  volSlider = createSlider(-48, 0, -12, 0.1);
```

```

volSlider.position(controlX, controlY + 20);
volSlider.addClass('custom-slider');

controlX += controlSpacing*2;

[rest of the sliders . . . ]

```

La función `draw()` se utiliza para dibujar el gráfico de la envolvente ADSR y la forma de onda del sintetizador. Llama a las funciones `drawADSRGraph()` y `drawWaveform()` para realizar estas tareas.

```

function draw() {
  background(320);
  drawADSRGraph();
  drawWaveform();
}

```

La función `setupSliders(synth)` se utiliza para configurar los controles deslizantes del panel de diseño de sonido. Recibe un objeto `synth` como argumento, que representa el sintetizador que se utilizará. Dentro de esta función, se definen funciones de devolución de llamada para cada control deslizante que actualizan los parámetros del sintetizador y envían los valores actualizados a través de `window.parent.postMessage()`, para poder enviar al archivo principal los nuevos parámetros del `synth` que se van a usar en el `theremin`.

```

function setupSliders(synth) {
  // Create a function to send the updated synth parameters to the parent
  window
  const sendUpdatedSynth = () => {
    window.parent.postMessage({
      type: 'updatedSynth',
      options: {
        volume: synth.volume.value,
        oscillatorType: synth.oscillator.type,
        envelope: {
          attack: synth.envelope.attack,
          decay: synth.envelope.decay,
          sustain: synth.envelope.sustain,
          release: synth.envelope.release
        }
      }
    }, '*');
  };
}

```

```

    // Update the synth and send the updated parameters when a slider value
    changes
    volSlider.input(() => {
        synth.volume.value = volSlider.value();
        sendUpdatedSynth();
    });

    typeSelect.changed(() => {
        synth.oscillator.type = typeSelect.value();
        sendUpdatedSynth();
    });

    attackSlider.input(() => {
        synth.envelope.attack = attackSlider.value();
        sendUpdatedSynth();
    });
    [ . . . ]

```

La función drawADSRGraph() se encarga de dibujar el gráfico de la envolvente ADSR. Utiliza los valores de los controles deslizantes para mapear los puntos en el gráfico y dibuja líneas y áreas degradadas para representar la envolvente ADSR. Se utilizan bucles for para dibujar todas las líneas y funciones auxiliares para los gradientes con los colores escogidos para las áreas del ADSR.

```

function drawADSRGraph() {
    // Set up the canvas style
    push();
    translate(width / 8, height / 18); // Update the graph position
    background(23,9,41);

    // Draw grid lines and time labels
    stroke(255, 255, 255, 100);
    strokeWeight(1);
    textSize(12);
    fill(255);
    textAlign(CENTER, CENTER);

    // Display grid values in ms
    for (let i = 0; i <= width / 2 + 150; i += 50) {
        line(i, 0, i, height / 2);
        if (i >= 0) {
            text(i * 10, i, -10);
        }
    }

    for (let i = 0; i <= height / 2; i += 20) {
        line(0, i, width / 2 + 150, i);
    }
}

```

```

}

// Map ADSR values to the graph
let attackX = map(attackSlider.value(), 0, 2, 0, width / 4);
let decayX = map(decaySlider.value(), 0, 2, 0, width / 4);
let sustainY = map(sustainSlider.value(), 0, 1, height / 2, 0);
let releaseX = map(releaseSlider.value(), 0, 5, 0, width / 4);

// Draw the ADSR envelope
strokeWeight(3);
stroke(255, 255, 255);
noFill();

// Attack region
stroke(255, 0, 0);
line(0, height / 2, attackX, 0);

// Decay region
stroke(0, 255, 0);
line(attackX, 0, attackX + decayX, sustainY);

// Sustain region
stroke(0, 0, 255);
line(attackX + decayX, sustainY, attackX + decayX + releaseX, height / 2);

// Release region )
stroke('#2A9D8F');
line(attackX + decayX + releaseX, height / 2, width / 2, height / 2);

// Draw gradient areas below each line
drawGradientArea(0, height / 2, attackX, 0, color('#2A9D8F'), color('#48B9A6'));
drawGradientArea(attackX, 0, attackX + decayX, sustainY, color('#1D6F67'), color('#2A9D8F'));
drawGradientArea(attackX + decayX, sustainY, attackX + decayX + releaseX, height / 2, color('#143D3B'), color('#1D6F67'));

// Draw points at vertices
fill(255, 255, 255);
ellipse(0, height / 2, 8);
ellipse(attackX, 0, 8);
ellipse(attackX + decayX, sustainY, 8);
ellipse(attackX + decayX + releaseX, height / 2, 8);

pop();
}

```

La función `drawGradientArea()` se utiliza para dibujar el gradiente de la curva ADSR en la función anterior. Nótese como la función `lerp` es quien crea el degradado entre ambos colores.

```
// Helper function to draw gradient rectangle with a solid line at the bottom
function drawGradientArea(x1, y1, x2, y2, c1, c2) {
  let gradientStep = 0.01;
  noFill();

  for (let i = 0; i <= 1; i += gradientStep) {
    let interColor = lerpColor(c1, c2, i);
    let alphaValue = map(i, 0, 1, 255, 0); // Map the alpha value from 255 to 0
    stroke(interColor._array[0] * 255, interColor._array[1] * 255, interColor._array[2] * 255, alphaValue);
    line(x1, lerp(y1, height / 2, i), x2, lerp(y2, height / 2, i));
  }

  // Draw a solid line at the bottom of the gradient
  stroke(c2);
  line(x1, height / 2, x2, height / 2);
}
```

La función `drawWaveform()` se utiliza para dibujar la forma de onda del sintetizador. Conecta el objeto `wave` al sintetizador y utiliza los valores de la forma de onda para dibujar una forma de onda en el lienzo.

```
function drawWaveform() {
  stroke(255);
  strokeWeight(1);
  noFill();

  // Connect the Waveform object to the synth
  thereminSynth.connect(wave);

  // Update the buffer to get the latest waveform values
  let buffer = wave.getValue();
  push();
  translate(0, height/2);
  beginShape();
  for (let i = 0; i < buffer.length; i++) {
    let x = map(i, 0, buffer.length, 0, width);
    let y = map(buffer[i], -1, 1, height / 4, 0);
    vertex(x, y);
  }
  endShape();
  pop();
}
```

Aquí una captura del panel de diseño siendo usado y probado tras usar el botón de reproducir. La onda se anima acorde con el sonido.

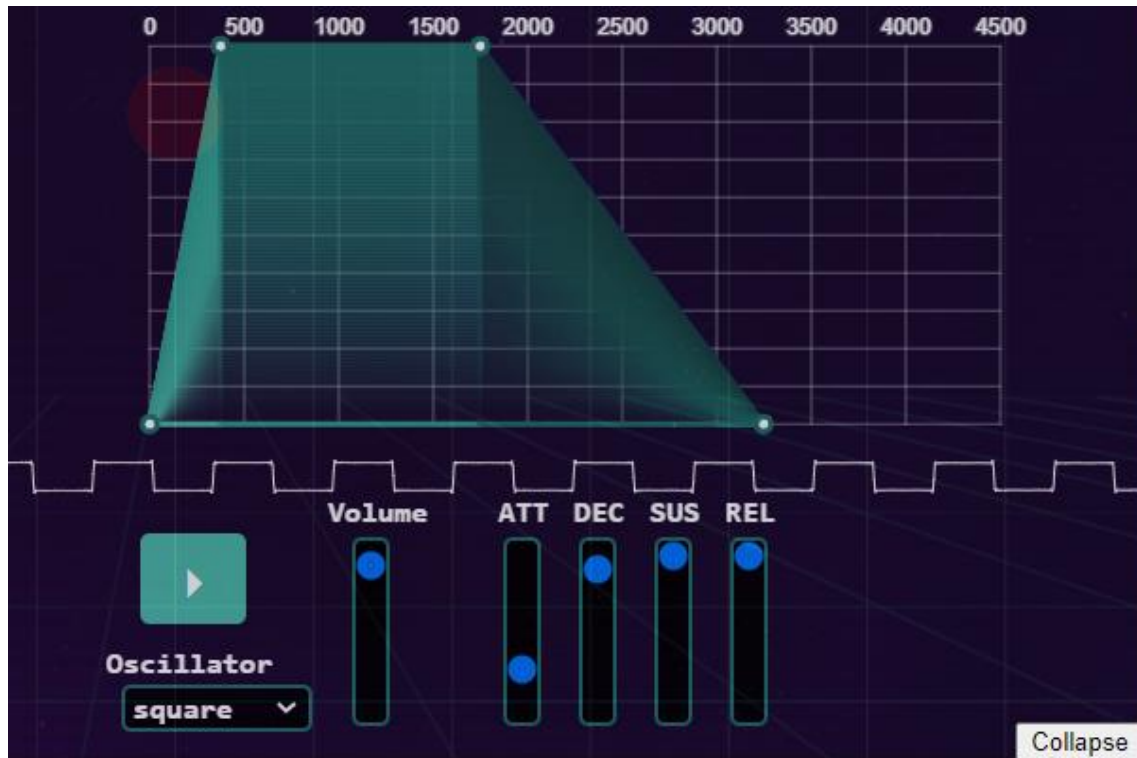
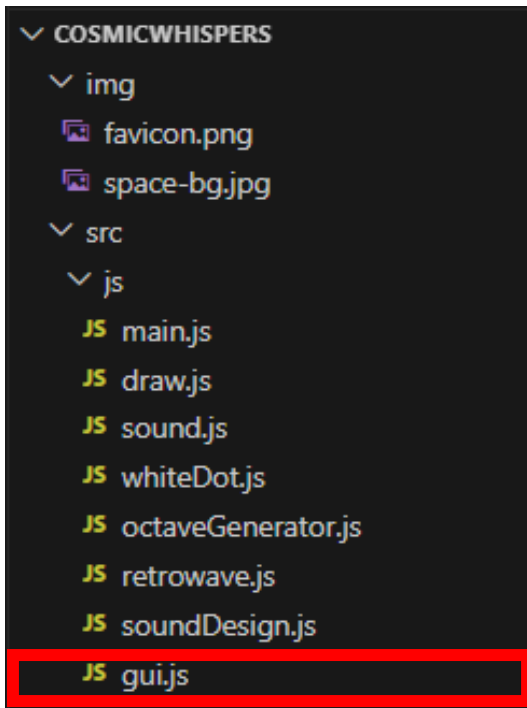


Ilustración 27 – Panel de control de Sound Design reproduciendo la onda recién diseñada

## 4.8. Graphic User Interace



El código utilizado para la creación de interfaz de usuario se ha creado en un módulo aparte para mejor control de las variables. Se utiliza tanto la librería de dat.gui para control de variables en tiempo real como botones en p5.js para la selección de octaveGenerators y Notas.

La función `drawOctaveOffsetToolbar()` dibuja una barra de herramientas para ajustar el desplazamiento de octava del octaveGenerator. Muestra círculos que representan diferentes opciones de desplazamiento de octava, y el círculo seleccionado se resalta en rojo.

```
function drawOctaveOffsetToolbar() {
  const toolbarX = 50;
  const toolbarY = 500;
  const circleSize = 30;
  const circleSpacing = 10;
  const octaveOffsets = [-1, 0, 1];

  for (let i = 0; i < octaveOffsets.length; i++) {
    const circleX = toolbarX;
    const circleY = toolbarY + (circleSize + circleSpacing) * i;

    if (octaveOffsets[i] === settings.octaveOffset) {
      fill(255, 0, 0);
    } else {
      fill(255);
    }

    ellipse(circleX, circleY, circleSize);
    fill(0);
    text(octaveOffsets[i], circleX - 3, circleY + 3);
  }
}
```

La función `handleOctaveOffsetToolbarClick()` maneja los clics en la barra de herramientas de desplazamiento de octava. Actualiza la configuración del desplazamiento de octava en función del círculo seleccionado.

```
function handleOctaveOffsetToolbarClick() {
  const toolbarX = 50;
  const toolbarY = 500;
  const circleSize = 30;
  const circleSpacing = 10;
  const octaveOffsets = [-1, 0, 1];

  for (let i = 0; i < octaveOffsets.length; i++) {
    const circleX = toolbarX;
    const circleY = toolbarY + (circleSize + circleSpacing) * i;

    if (
      mouseX >= circleX - circleSize / 2 &&
      mouseX <= circleX + circleSize / 2 &&
      mouseY >= circleY - circleSize / 2 &&
      mouseY <= circleY + circleSize / 2
    ) {
      settings.octaveOffset = octaveOffsets[i];
      break;
    }
  }
}
```

La función `drawScaleToolbar()` dibuja una barra de herramientas para seleccionar la nota de la escala musical para crear los nodos. Muestra círculos que representan las notas de la escala actual, y la nota seleccionada se resalta en rojo.

```
function drawScaleToolbar() {
  const regionWidth = width / numRegions;
  const toolbarY = height - 100;

  for (let i = 0; i < scales[settings.scale].length; i++) {
    const circleX = (regionWidth * i) + (regionWidth / 2); // Center the
    circle within the region
    const circleY = toolbarY;

    if (i === selectedNoteIndex) {
      fill(255, 0, 0);
    } else {
      fill(255);
    }

    ellipse(circleX, circleY, circleSize);
  }
}
```



La función `handleToolbarClick()` maneja los clics en la barra de herramientas de la escala musical. Actualiza el índice de la nota seleccionada en función del círculo seleccionado.

```
function handleToolbarClick() {

  const regionWidth = width / numRegions;
  const toolbarY = height - 100;

  for (let i = 0; i < scales[settings.scale].length; i++) {
    const circleX = (regionWidth * i) + (regionWidth / 2);
    const circleY = toolbarY;

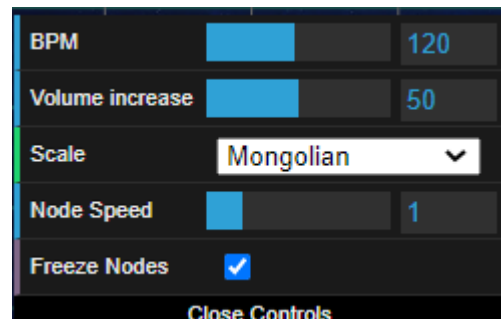
    if (
      mouseX >= circleX - circleSize / 2 &&
      mouseX <= circleX + circleSize / 2 &&
      mouseY >= circleY - circleSize / 2 &&
      mouseY <= circleY + circleSize / 2
    ) {
      selectedNoteIndex = i;
      break;
    }
  }
}
```

La función `setupGui()` configura la interfaz gráfica de usuario utilizando la biblioteca `dat.GUI`. Crea controles deslizantes para ajustar el tempo (bpm), el volumen, la escala, la velocidad de los nodos y su activación de movimiento.

Algunos parámetros son muy sencillos de actualizar, como el bpm o el volumen, ya que conectan directamente con parámetros del master de `Tone.js`, pero otros parámetros más específicos requieren de código adicional. Por ejemplo, cambiar la escala requiere recalcular las frecuencias de todas las notas creadas y los botones de la escala.

```
function setupGui() {
  const gui = new dat.GUI({ autoPlace: false });
  gui.domElement.id = 'gui-container';
  document.body.appendChild(gui.domElement);

  const params = {
    bpm: Tone.Transport.bpm.value,
    volume: 50
```



*Ilustración 28 – Menú de `dat.GUI` para control de variables en tiempo real*

```

};
// BPM
gui.add(params, 'bpm', 10, 240).step(1).name('BPM').onChange((value) => {
  Tone.Transport.bpm.value = value;
});

// Volume
gui.add(params, 'volume', 0, 100).step(1).name('Volume increase').onChange((value) => {
  Tone.Destination.volume.value = value / 10;
});

// Scale option
gui.add(settings, "scale", Object.keys(scales)).name("Scale").onChange(() => {
  if (settings.scale === "Random") {
    scales.Random = getRandomIntervals();
  }
  // Update the current scale
  currentScale = scales[settings.scale];
  updateWhiteDotsScale();

  // Recalculate the notes in the scale and the number of regions
  scaleNotes = currentScale.map(interval => {
    let frequency = rootFrequency * Math.pow(2, interval / 12);
    return Tone.Frequency(frequency).toNote();
  });

  numRegions = scaleNotes.length;

  // Reset the selected note index
  selectedNoteIndex = 0;
});

// Node speed
const speedController = gui.add(settings, "speed", 0.01, 5).step(0.01).name("Node Speed");
speedController.onChange(updateNodeSpeeds);

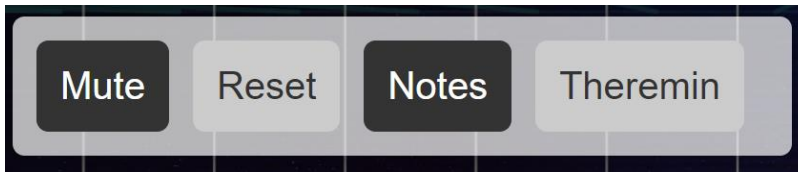
// Freeze nodes
gui.add(settings, "lockNodes").name("Freeze Nodes");
}

```

La función auxiliar `updateNodeSpeeds()` actualiza las velocidades de los nodos (white dots) en función del ajuste de velocidad.

```
function updateNodeSpeeds() {  
  for (let whiteDot of whiteDots) {  
    whiteDot.speed = random(0.5, settings.speed);  
  }  
}
```

La función `setupToolbar()` configura la barra de herramientas de la interfaz de usuario. Agrega event listeners a los botones de la barra de herramientas y maneja los clics para cambiar entre los modos de interacción "Notes" y "Theremin".



*Ilustración 29 – Menú principal de selección de modo y acciones básicas*

```
function setupToolbar() {
  interactionMode = "create";

  let isMuted = false;

  const muteButton = document.getElementById("mute-button");
  muteButton.addEventListener("click", () => {
    isMuted = !isMuted; // Toggle the mute state

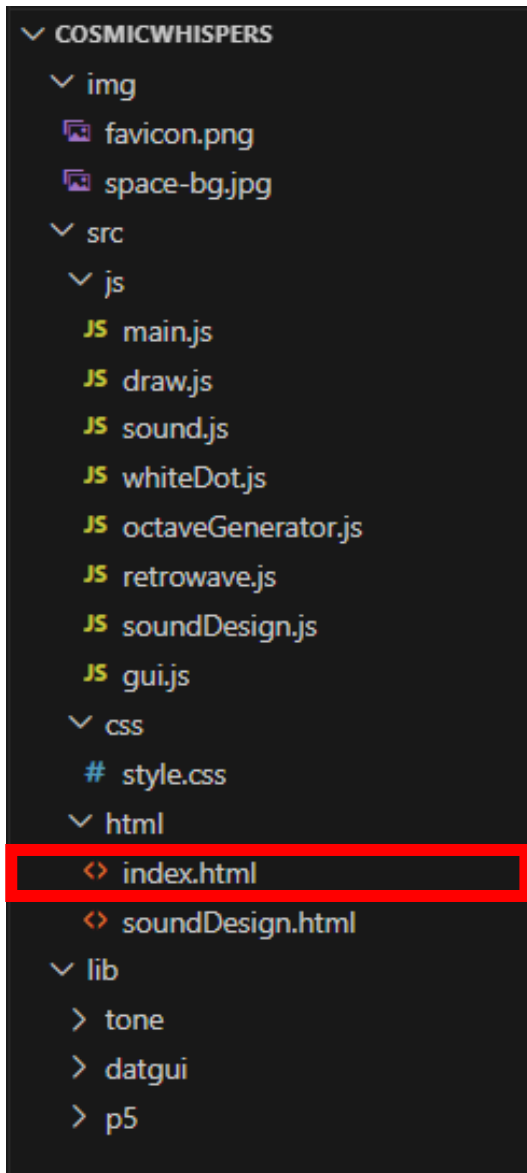
    if (isMuted) {
      muteButton.classList.add("active");
      Tone.Destination.mute = true;
    } else {
      muteButton.classList.remove("active");
      Tone.Destination.mute = false;
    }
  });

  const whiteDotButton = document.getElementById("create-button");
  whiteDotButton.addEventListener("click", () => {
    interactionMode = "create";
    whiteDotButton.classList.add("active");
    thereminButton.classList.remove("active");
  });

  const thereminButton = document.getElementById("theremin-button");
  thereminButton.addEventListener("click", () => {
    interactionMode = "theremin";
    thereminButton.classList.add("active");
    whiteDotButton.classList.remove("active");
  });

  whiteDotButton.classList.add("active");
}
```

## 4.9. Archivos HTML + CSS



El código HTML utilizado ha sido muy básico, pues gran parte del apartado visual se renderiza en p5.js. Sin embargo, es necesario crear la plantilla HTML para incluir en ella las librerías, el favicon, la hoja de estilos y los scripts utilizados.

Adicionalmente, se ha creado el toolbar de selección de modo en HTML y el botón para colapsar el iframe del Sound Design.

El resto del código HTML y CSS usado en el proyecto es absolutamente básico y carece de interés según los temas tratados en el proyecto, pero si aun así se desea consultar, está disponible en el Anexo I: Código completo.

```

<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8">
  <title>Cosmic Whispers</title>
  <link rel="stylesheet" href="../../css/style.css">
  <link rel="icon" type="image/png" href="../../img/favicon.png">

  <script src="../../lib/p5/p5.min.js"></script>
  <script src="../../lib/tone/Tone.min.js"></script>
  <script src="../../lib/datgui/dat.gui.min.js"></script>
</head>

<body>

  <div id="toolbar">
    <button id="mute-button">Mute</button>
    <button id="reset-button">Reset</button>
    <button id="create-button">Notes</button>
    <button id="theremin-button">Theremin</button>

  </div>
  <iframe id="synth-iframe" src="../../html/soundDesign.html" width="600"
height="400" frameborder="0"></iframe>
  <button id="collapse-btn">Hide/Show Sound Design</button>

  <!-- JS scripts here -->
  <script src="../../js/main.js"></script>
  <script src="../../js/draw.js"></script>
  <script src="../../js/sound.js"></script>
  <script src="../../js/soundDesign.js"></script>
  <script src="../../js/retrowave.js"></script>
  <script src="../../js/whiteDot.js"></script>
  <script src="../../js/gui.js"></script>
  <script src="../../js/octaveGenerator.js"></script>
</div>

</body>

</html>

```

#### 4.10. Dependencias entre los archivos

Tras analizar todos los archivos mencionados, podemos sintetizar la dependencia de archivos en un grafo como este:

Donde se puede apreciar cómo draw.js y sound.js son los archivos con mayores interdependencias, pues participan con los componentes principales de la aplicación.

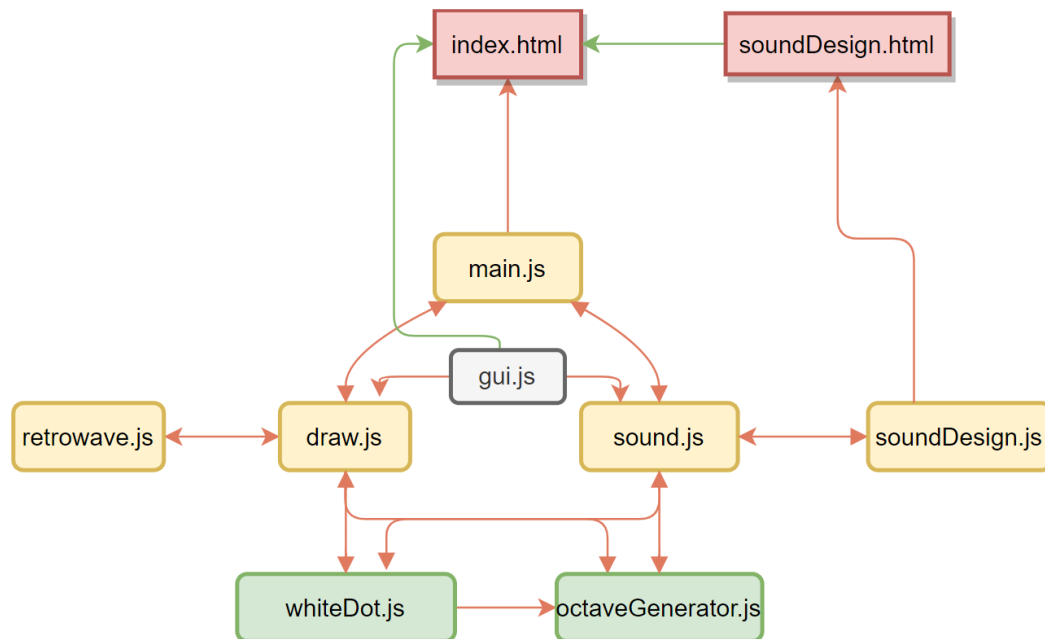


Ilustración 30 – Gráfico de dependencia de archivos

El archivo **gui.js** y **soundDesign.html** están dibujados con dependencias de flechas verdes, pues estos se cargan de manera asíncrona sobre **index.html**, una vez la aplicación ya ha cargado. Recordemos que **soundDesign.html** es un **iframe** que carga su propio código en **index.html**.

## Futuras implementaciones

Este proyecto se podría expandir por muchos frentes. Se muestran algunas sugerencias de implementaciones futuras, ordenadas de menor a mayor complejidad:

- Escalas personalizadas: Permitir al usuario definir sus propias escalas para elegir desde el GUI, indicando los valores de los semitonos que quiere, como se definen actualmente.
- Nuevos instrumentos: Añadir nuevos instrumentos para elegir para los nodos. Actualmente el sintetizador se crea con unos valores de sintetizador concretos, pero se pueden definir una serie de Presets con instrumentos ya creados con el sintetizador para cambiar rápidamente de timbre.
- Percusión: Añadir un kit de percusión, como otro modo aparte, donde se puedan crear nodos con sonidos percusivos asociados, eligiéndolos análogamente al modo “Notes” desde la barra de notas en la escala.
- Samples personalizados: Permitir al usuario utilizar sus propios sonidos, cargándolos desde el ordenador, y modificarlos usando la tecnología actual del Sound Design Panel.
- Grabación: Poder grabar una sesión musical, para revisarla en busca de patrones, ya sea en formato audio, vídeo o MIDI, para poder utilizarla en un software de sonido. Se podría implementar también como plug-in para que permita crearlo directamente sobre el software.
- Compatibilidad móvil: Desarrollar una versión móvil o una aplicación para tabletas, lo que permitiría a los usuarios acceder y utilizar el proyecto en dispositivos portátiles. Esto aumentaría la accesibilidad y la capacidad de llevar el proyecto a cualquier lugar.
- Compatibilidad MIDI: Explorar la posibilidad de integración con controladores externos, como teclados MIDI o superficies de control, para permitir una interacción más táctil y expresiva con los nodos y el sonido resultante.

Estas son solo algunas ideas para expandir aún más las posibilidades del proyecto y brindar a los usuarios más herramientas y opciones para explorar y experimentar con la creación musical. Puesto que es un proyecto de código abierto en GitHub, es bienvenidas cualquier sugerencia.



## 5. Conclusiones y próximos pasos

Este proyecto ha sido una experiencia enriquecedora que ha permitido explorar las capacidades de creación musical a través de la programación. Mediante el uso de sintetizadores y herramientas de programación creativa, se ha logrado desarrollar un sistema interactivo que permite a los usuarios generar música de manera intuitiva y personalizada.

Además, ha sido una gran oportunidad para poner a prueba mis capacidades de autodidacta, teniendo que aprender a usar tres librerías de Javascript totalmente diferentes desde cero. Las librerías han terminado resultando intuitivas y fáciles de utilizar, y dan pie a seguir explorando y aumentando las capacidades del proyecto.

El proyecto ha demostrado el potencial de la programación como una herramienta poderosa para la expresión artística y la creación musical. La capacidad de manipular parámetros como el desplazamiento de octava, la escala musical, el tempo y el volumen, brinda a los usuarios un control completo sobre la música generada.

Este proyecto solo representa la punta del iceberg en cuanto a las posibilidades que ofrece la programación creativa en el campo de la música. Hay un vasto mundo de técnicas, algoritmos y herramientas por descubrir y explorar. La combinación de la música y la programación abre nuevas oportunidades para la experimentación sonora y la creación de composiciones únicas.

Como estudiante de ingeniería informática y músico por vocación, ha sido todo un descubrimiento poder fusionar el conocimiento de ambas para crear una herramienta única, que abre las puertas a nuevos horizontes. Ahora que conozco cómo utilizar estas librerías, esto es solo el principio. Además, todavía no he implementado el uso de inteligencia artificial en la producción (que es mi especialidad), pero cuando ocurra va a ser un cambio de paradigma en la producción considerable también.

# Bibliografía

## Conceptos teóricos

Ingenierizando. (2021). Ciclo de una Onda. Ingenierizando. Recuperado de: <https://www.ingenierizando.com/cinematica/ciclo-de-una-onda/>

Nugent, J. (2020). What is a Synthesizer? Higher Hz. Recuperado de: <https://higherhz.com/what-is-synthesizer/>

Wikipedia. (s.f.). Sonido. Recuperado de: <https://es.wikipedia.org/wiki/Sonido>

Forster, J. (2015). How Many Ways Can a Piano Be Out of Tune? Sterling Piano Tuning. Recuperado de: <https://www.sterlingpianotuning.com/how-many-ways-can-a-piano-be-out-of-tune/>

## Documentación

Mozilla Developer Network. (s.f.). HTML Documentation. Recuperado de: <https://developer.mozilla.org/en-US/docs/Web/HTML>

Mozilla Developer Network. (s.f.). CSS Documentation. Recuperado de: <https://developer.mozilla.org/en-US/docs/Web/CSS>

p5.js. (s.f.). p5.js Reference. Recuperado de: <https://p5js.org/reference/>

Tone.js. (s.f.). Tone.js Documentation. Recuperado de: <https://tonejs.github.io/docs/14.7.77/index.html>

## Librerías

Tone.js: <https://github.com/Tonejs>

p5: <https://github.com/processing/p5.js>

dat.gui: <https://github.com/dataarts/dat.gui>

## Anexo I: Código completo

### Index.html

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8">
  <title>Cosmic Whispers</title>
  <link rel="stylesheet" href="../css/style.css">
  <link rel="icon" type="image/png" href="../../img/favicon.png">

  <script src="../../lib/p5/p5.min.js"></script>
  <script src="../../lib/tone/Tone.min.js"></script>
  <script src="../../lib/datgui/dat.gui.min.js"></script>
</head>

<body>

  <div id="toolbar">
    <button id="mute-button">Mute</button>
    <button id="reset-button">Reset</button>
    <button id="create-button">Notes</button>
    <button id="theremin-button">Theremin</button>

  </div>
  <iframe id="synth-iframe" src="../html/soundDesign.html" width="600"
height="400" frameborder="0"></iframe>
  <button id="collapse-btn">Hide/Show Sound Design</button>

  <!-- JS scripts here -->
  <script src="../js/main.js"></script>
  <script src="../js/draw.js"></script>
  <script src="../js/sound.js"></script>
  <script src="../js/soundDesign.js"></script>
  <script src="../js/retrowave.js"></script>
  <script src="../js/whiteDot.js"></script>
  <script src="../js/gui.js"></script>
  <script src="../js/octaveGenerator.js"></script>
</div>

</body>

</html>
```

## soundDesign.html

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <script src="../../lib/p5/p5.min.js"></script>
  <script src="../../lib/tone/Tone.min.js"></script>
  <script src="../../lib/datgui/dat.gui.min.js"></script>
  <script src="../js/soundDesign.js"></script>

  <link rel="stylesheet" href="../../css/style.css">
</head>

<body>
</body>

</html>
```

## Style.css

```
body {
  margin: 0;
  padding: 0;
  width: 100%;
  height: 100%;
  overflow: hidden;
}

canvas {
  display: block;
}

#gui-container {
  position: absolute;
  top: 20px;
  right: 20px;
}

#toolbar {
  position: absolute;
  transform-origin: top left;
  top: 50px;
  left: 80px;
  background-color: rgba(255, 255, 255, 0.7);
  border-radius: 5px;
  padding: 10px;
  box-shadow: 0px 0px 10px rgba(0, 0, 0, 0.3);

  display: flex;
  flex-direction: row;
  flex-wrap: nowrap;
}

#toolbar button {
  font-size: 16px;
  padding: 10px;
  margin-right: 10px;
  border: none;
  border-radius: 5px;
  background-color: #ccc;
  color: #333;
}

#toolbar button:hover {
  cursor: pointer;
}
```

```

#toolbar button.active {
  background-color: #333;
  color: #fff;
}

#toolbar button:not(.active):hover {
  background-color: #ddd;
}

#synth-iframe {
  position: absolute;
  bottom: 15%;
  right: 0%;
  transition: left 2.5s ease;
  border: 1px solid silver;
  opacity: 80%;
}

#collapse-btn{
  position: absolute;
  bottom: 15%;
  right: 0%;
  transition: left 2.5s ease;
}

.play-button {
  background-color: #2A9D8F;
  color: white;
  font-family: Arial, sans-serif;
  font-size: 24px;
  border-radius: 5px;
  padding: 8px 16px;
  border: none;
  cursor: pointer;
  outline: none;
  transition: background-color 0.3s;
}

.play-button:hover {
  background-color: #48B9A6;
}

.slider-label {
  font-family: monospace;
  font-size: 16px;
}

```

```

font-weight: bold;
color: #FFFFFF;
text-align: center;
position: absolute;
transform: translate(150%,-300%);
}

.custom-slider {
  transform: rotate(270deg);
  width: 100px;
}

.custom-slider::-webkit-slider-thumb {
  -webkit-appearance: none;
  appearance: none;
  width: 5px;
  border-radius: 20%;
  height: 55px;
  background: #04AA6D;
  cursor: pointer;
}

.custom-slider::-webkit-slider-runable-track {
  background-color: #000000;
  border: 2px solid #1D6F67;
  border-radius: 5px;
  position: relative;
}

.custom-select{
  font-family: monospace;
  font-size: 16px;
  font-weight: bold;
  color: #FFFFFF;
  position: absolute;
  background-color: #000000;
  border: 2px solid #1D6F67;
  border-radius: 5px;
  position: relative;
}

```

## Main.js

```
function setup() {  
  createCanvas(windowWidth, windowHeight);  
  setupBackground();  
  
  // set up Tone.js Transport to play a 4-beat pattern  
  Tone.Transport.bpm.value = 120;  
  Tone.Transport.timeSignature = 4;  
  Tone.Transport.loopEnd = "4m";  
  
  Tone.Transport.start();  
  Tone.Transport.scheduleRepeat(onBeat, "8n");  
  
  setupToolbar();  
  frameRate(30);  
  setupGui();  
}
```



## Draw.js

```
window.onload = function () {
  let synthFrame = document.getElementById('synth-iframe');
  synthFrame.style.visibility = 'hidden';
  let button = document.getElementById('collapse-btn')
  button.addEventListener('click', function () {
    event.stopPropagation();
    let synthFrame = document.getElementById('synth-iframe');
    if (synthFrame.style.visibility !== 'hidden') {
      synthFrame.style.visibility = 'hidden';
    } else {
      synthFrame.style.visibility = 'visible';
    }
  });
}

function windowResized() {
  resizeCanvas(windowWidth, windowHeight);
}

let backgroundImage;
function preload() {
  backgroundImage = loadImage('../img/space-bg.jpg');
}

function draw() {
  image(backgroundImage, 0, 0, width, height);
  drawBackground();

  numDots = whiteDots.length;
  maxDots = 75;

  backgroundSpeed = 10;
  updateBackgroundSpeed();

  background(0, 50);
  drawMetronome();

  for (let whiteDot of whiteDots) {
    whiteDot.update();
    whiteDot.display();
    whiteDot.checkDistance(octaveGenerators, distanceThreshold = 200,
lineColor = 200);
  }
  for (const generator of octaveGenerators) {
    generator.display();
    generator.updateWhiteDotsInRange(whiteDots);
    generator.labelWhiteDots();
  }
}
```

```

}

// Mouse Over for interacting with star
let mouseOverDot = false;
for (let whiteDot of whiteDots) {
  whiteDot.display();

  if (whiteDot.isMouseOver()) {
    mouseOverDot = true;
  }
}
if (mouseOverDot) {
  cursor(HAND);
} else {
  cursor(ARROW);
}

let currentNote;
let isPlaying = false;

let currentScale = scales[settings.scale]
const scaleNotes = currentScale.map(interval => {
  const frequency = rootFrequency * Math.pow(2, interval / 12);
  return Tone.Frequency(frequency).toNote();
});
const numRegions = scaleNotes.length;

const regionWidth = width / numRegions;
// Draw gridlines for theremin
stroke(100);
for (let i = 1; i < numRegions; i++) {
  line(regionWidth * i, 0, regionWidth * i, height);
}

// Theremin mode
if (interactionMode === "theremin") {
  activeSynth = thereminSynth;

  // Draw a vertical line at the cursor position
  stroke(200);
  line(mouseX, 0, mouseX, height);

  // Map the mouse position to a note in the Pentatonic scale based
  on the region
  const regionIndex = floor(mouseX / regionWidth);
  // Higher mouse y=> Higher volume

```

```

    let volume = map(mouseY, height, 0, -48, 0);

    const note = scaleNotes[regionIndex];
    const frequency = Tone.Frequency(note).toFrequency();

    // Update current note based on mouse position
    currentNote = frequency;
    // Update synth frequency and trigger attack/release based on
mouse state
    if (mouseIsPressed) {
        thereminSynth.triggerAttackRelease(currentNote);
        thereminSynth.volume.value = volume;
        isPlaying = true;
    } else {
        thereminSynth.triggerRelease(Tone.now());
        isPlaying = false;
    }
}

else { // interactionMode == Create
    activeSynth = polySynth;
    drawOctaveOffsetToolbar();
    drawScaleToolbar();
    thereminSynth.triggerRelease(Tone.now());
}
}

window.onload = function () {
    let synthFrame = document.getElementById('synth-iframe');
    synthFrame.style.visibility = 'hidden';
    let button = document.getElementById('collapse-btn');
    button.addEventListener('click', function () {
        event.stopPropagation();
        let synthFrame = document.getElementById('synth-iframe');
        if (synthFrame.style.visibility !== 'hidden') {
            synthFrame.style.visibility = 'hidden';
        } else {
            synthFrame.style.visibility = 'visible';
        }
    });

    // Add an event listener to the reset button
    const resetButton = document.getElementById('reset-button');
    resetButton.addEventListener('click', function(event) {
        event.stopPropagation(); // prevent event from reaching the canvas
        whiteDots = []; // clear out the whiteDots array
        octaveGenerators = []; // clear out the octaveGenerators array
    });

```

```

    Tone.Transport.stop();
    Tone.Transport.position = 0;
    Tone.Transport.start();
  });
}

function updateBackgroundSpeed() {
  // Update the speed variable based on the number of dots
  backgroundSpeed = map(numDots, 0, maxDots, 0.1, 10);
}

function mouseClicked() {
  if (interactionMode === "create") {
    if (keyIsDown(SHIFT)) {
      const newOctaveGenerator = new OctaveGenerator(mouseX, mouseY);
      octaveGenerators.push(newOctaveGenerator);
    } else {
      // Check if user clicks a dot
      let clickedDotIndex = -1;

      // If toolbar or octave offset toolbar was clicked, return early
      if (handleToolbarClick() || handleOctaveOffsetToolbarClick()) {
        return;
      }

      for (let i = 0; i < whiteDots.length; i++) {
        if (dist(mouseX, mouseY, whiteDots[i].x, whiteDots[i].y) <=
whiteDots[i].size / 2) {
          clickedDotIndex = i;
          break;
        }
      }

      // If user clicks a dot, remove it
      if (clickedDotIndex >= 0) {
        whiteDots.splice(clickedDotIndex, 1);
      } else {
        whiteDots.push(new WhiteDot(mouseX, mouseY,
selectedNoteIndex));
      }
    }
  }
}

```

```

    }
}

// An array of 11 colors to choose from
let colors = ["#ff00ff", "#00ffea", "#5effff", "#FF9A8C", "#94FBAB",
"#FD7E14", "#2D3748", "#FFC2E7", "#0092FF"];

function drawMetronome() {
  let currentPosition = Tone.Transport.position.split(":");
  let currentBeat = parseInt(currentPosition[1]);
  beatNum = currentBeat;

  let metronomeSize = 50;
  switch (beatNum) {
    case 0:
      fill(255, 0, 0); //red color for the first beat
      break;
    case 1:
      fill(202, 0, 0);
      break;
    case 2:
      fill(155, 0, 0);
      break;
    default:
      fill(102, 0, 0);
  }

  ellipse(width / 2, height / 2, metronomeSize);
}

```

## Sound.js

```
const root = 60;

// Note Synth
let polySynth = new Tone.PolySynth(Tone.FMSynth, {

  "harmonicity": 8,
  "modulationIndex": 2,
  "oscillator": {
    "type": "sine"
  },
  "envelope": {
    "attack": 0.01,
    "decay": 2,
    "sustain": 0.1,
    "release": 1
  },
  "modulation": {
    "type": "square"
  },
  "modulationEnvelope": {
    "attack": 0.002,
    "decay": 0.2,
    "sustain": 0,
    "release": 0.2
  }
}).toDestination();

// Additional modules for the synth

// Compressor to clean the audio peaks
const compressor = new Tone.Compressor({
  attack: 0.003,
  release: 0.25,
  threshold: -24,
  ratio: 4,
  knee: 30,
}).toDestination();

// Reverb
const reverb = new Tone.Reverb({
  decay: 2,
  preDelay: 0.01,
  wet: 0.5
});

// Clean low frequencies
const lowpassFilter = new Tone.Filter({
```

```

    type: "lowpass",
    frequency: 50,
    Q: 1
  });

  // Clean high frequencies
  const highpassFilter = new Tone.Filter({
    type: "highpass",
    frequency: 1000,
    Q: 100
  });

  // Echo
  const echo = new Tone.FeedbackDelay('8n', 0.25).toDestination();

  polySynth
    .connect(compressor)
    .connect(reverb)
    .connect(echo)
    .connect(lowpassFilter)
    .connect(highpassFilter);

  // Theremin Synthesizer
  let thereminSynth;
  thereminSynth = new Tone.Synth({
    oscillator: {
      type: "sine", // Use a sawtooth wave for the oscillator
      detune: 10, // Detune the oscillator slightly for a richer sound
    },
    envelope: {
      attack: 0.2, // Increase the attack time for a softer attack
      decay: 0.2, // Decrease the decay time for a sharper decay
      sustain: 0.5, // Decrease the sustain level for a shorter sustain
      release: 15, // Increase the release time for a longer fade out
    },
    volume: -16,
  }).toDestination();

  const thereminEcho = new Tone.FeedbackDelay('8n', 0.25).toDestination();
  // '8n' = 1/8 note delay time, 0.5 = 50% feedback

  thereminSynth
    .connect(thereminEcho);

  let beatNum;
  let currentEighthNote;

  // Default settings

```

```

const settings = {
  scale: "Mongolian",
  speed: 1,
  lockNodes: true,
  octaveOffset: 0,
};

const iframe = document.getElementById('synth-iframe');

const scales = {
  Major: [0, 2, 4, 5, 7, 9, 11, 12, 14, 16, 17, 19, 21, 23],
  Natural_Minor: [0, 2, 3, 5, 7, 8, 10, 12, 14, 15, 17, 19, 20, 22],
  Harmonic_Minor: [0, 2, 3, 5, 7, 8, 11, 12, 14, 15, 17, 19, 20, 23],
  Melodic_Minor: [0, 2, 3, 5, 7, 9, 11, 12, 14, 15, 17, 19, 21, 23],
  Mixolydian: [0, 2, 4, 5, 7, 9, 10, 12, 14, 16, 17, 19, 21, 22],
  Phrygian: [0, 1, 3, 5, 7, 8, 10, 12, 13, 15, 17, 19, 20, 22],
  Lydian: [0, 2, 4, 6, 7, 9, 11, 12, 14, 16, 18, 19, 21, 23],
  Locrian: [0, 1, 3, 5, 6, 8, 10, 12, 13, 15, 17, 18, 20, 22],
  Mongolian: [0, 2, 5, 7, 9, 12, 14, 17, 19, 21, 24, 26, 29, 31],
  Pentatonic_Major: [0, 2, 4, 7, 9, 12, 14, 16, 19, 21, 24, 26, 28, 31],
  Pentatonic_Minor: [0, 3, 5, 7, 10, 12, 15, 17, 19, 22, 24, 27, 29, 31],
  Overtone: [0, 4, 7, 10, 12, 14, 16, 19, 22, 24, 26, 28, 31, 34, 36, 38,
40, 43, 46, 48, 50],
  Blues: [0, 3, 5, 6, 7, 10, 12, 15, 17, 18, 19, 22, 24, 27],
  Chromatic: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31],
  Whole_Tone: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28,
30],
  Random: getRandomIntervals(),
};

function getRandomIntervals() {
  const numNotes = Math.floor(Math.random() * 10) + 7; // Random number
of notes between 10 and 17
  const intervals = [0]; // The first note is always 0

  for (let i = 1; i < numNotes; i++) {
    const prevInterval = intervals[i - 1];
    const newInterval = prevInterval + Math.floor(Math.random() * 4) + 1;
// Random interval between 1 and 3 semitones
    intervals.push(newInterval);
  }

  return intervals;
}

```



```

let isThereminPlaying = false;
let thereminNote = null;

let currentScale = scales[settings.scale]
let rootNote = "C4";
let rootFrequency = Tone.Frequency(rootNote).toFrequency();
let scaleNotes = currentScale.map(interval => {
  let frequency = rootFrequency * Math.pow(2, interval / 12);
  return Tone.Frequency(frequency).toNote();
});
let numRegions = scaleNotes.length;
let activeSynth;
function getEighthNoteIndex(positionArray) {
  const bar = parseInt(positionArray[0]) * 8;
  const beat = parseInt(positionArray[1]) * 2;
  const sixteenthNote = parseInt(positionArray[2]) >= 2 ? 1 : 0;
  return (bar + beat + sixteenthNote) % 8;
}

let beatCount = 0;
let previousEighthNote = -1;

function onBeat(time) {
  console.log(currentEighthNote);
  let currentPosition = Tone.Transport.position.split(":");
  let playingNotes = new Set();

  currentEighthNote = getEighthNoteIndex(currentPosition)
  // Loop through each octave generator
  for (const generator of octaveGenerators) {
    // Loop through each white dot in the current generator
    for (const whiteDot of generator.whiteDots) {
      if (whiteDot.beatsToPlay[currentEighthNote]) {
        whiteDot.soundPlayed();
        whiteDot.beatsToPlay = new Array(8).fill(false);

        // Calculate the note with the octave offset
        const noteWithOffset =
Tone.Frequency(whiteDot.scaleNote).transpose(12 *
generator.octaveOffset).toNote();

        // Create a unique key for the playingNotes set
        const noteKey =
`${whiteDot.noteIndex}_${generator.octaveOffset}`;

        // Check if the note is already playing, in that case it wont
play
        if (!playingNotes.has(noteKey)) {
          playingNotes.add(noteKey);

```

```

        polySynth.triggerAttackRelease(noteWithOffset, "16n", time,
undefined, () => {
            playingNotes.delete(noteKey);
        });
    }
}
}
}

// If the beat counter gets stuck, reset the Transport
if (previousEighthNote === currentEighthNote) {
    resetTransport();
    previousEighthNote = -1;
    return;
} else {
    previousEighthNote = currentEighthNote;
}
}

function resetTransport() {
    Tone.Transport.stop();
    Tone.Transport.position = 0;
    Tone.Transport.start();
}

// Get a reference to the iframe
const activeSynthData = {
    type: 'thereminSynth',
    options: thereminSynth.get()
};

window.addEventListener('message', (event) => {
    const data = event.data;

    if (data.type === 'updatedSynth') {
        activeSynth.volume.value = data.options.volume;
        activeSynth.oscillator.type = data.options.oscillatorType;
        activeSynth.envelope.attack = data.options.envelope.attack;
        activeSynth.envelope.decay = data.options.envelope.decay;
        activeSynth.envelope.sustain = data.options.envelope.sustain;
        activeSynth.envelope.release = data.options.envelope.release;
    }
});

```

```
const synthIframe = document.getElementById('synth-iframe');

// Wait for the iframe content to load
synthIframe.addEventListener('load', () => {
  // Send the activeSynthData object as a message to the iframe
  synthIframe.contentWindow.postMessage(activeSynthData, '*');
});

// array to store white dots
let whiteDots = [];
let octaveGenerators = [];
```

## whiteDot.js

```
// WhiteDot class
class WhiteDot {
  constructor(x, y, noteIndex) {
    // Position attributes
    this.x = x;
    this.y = y;

    // Original size and size storage
    this.size = random(30, 50);
    this.originalSize = this.size;

    // Animation
    this.animationDuration = 200;
    this.animationStartTime = null;

    // Movement
    this.speed = random(0.2, settings.speed);
    this.angle = random(TWO_PI);

    // Note
    this.playedNote = false;
    this.isPlaying = false;
    this.scale = scales[settings.scale];
    this.noteIndex = noteIndex;
    this.midiNote = root + this.scale[this.noteIndex];
    this.scaleNote = Tone.Frequency(this.midiNote, 'midi');

    // Color
    this.originalColor = colors[this.noteIndex % colors.length];
    this.brightColor = "#99ffe4";
    this.currentColor = this.originalColor;

    // Rhythm
    this.beatsToPlay = new Array(8).fill(false);

    this.creationTime = Tone.Transport.seconds;

    const now = Tone.now()

    // Play created note
    polySynth.triggerAttackRelease(this.scaleNote, "0.5n");
    this.calculateTriggerTime(octaveGenerators); // Add this line to
    set triggerTime
  }
}
```

```

}

// update function to change position of white dot
update() {
  if (!settings.lockNodes) {
    this.x += this.speed * cos(this.angle);
    this.y += this.speed * sin(this.angle);

    // Wrap the position of the dot around the screen
    if (this.x < 0) {
      this.x = width;
    } else if (this.x > width) {
      this.x = 0;
    }

    if (this.y < 0) {
      this.y = height;
    } else if (this.y > height) {
      this.y = 0;
    }
  }
}

// Mouse over for interacting
isMouseOver() {
  const distanceToMouse = dist(this.x, this.y, mouseX, mouseY);
  return distanceToMouse < this.size / 2;
}

calculateTriggerTime(octaveGenerators) {
  if (!settings.showOctaveGenerator || octaveGenerators.length === 0)
return;
  let minDistance = Infinity;
  for (const generator of octaveGenerators) {
    const distance = dist(this.x, this.y, generator.x, generator.y);
    minDistance = min(minDistance, distance);
  }
  const maxDistance = dist(0, 0, width, height);
  const normalizedDistance = minDistance / maxDistance;
  this.triggerTime = Tone.Time("1n").toSeconds() *
normalizedDistance;
}

updateNoteIndex(scale) {
  this.midiNote = root + scale[this.noteIndex];
  this.scaleNote = Tone.Frequency(this.midiNote, 'midi');
}

```

```

// display function to draw the white dot
display() {
  fill(this.originalColor);

  // Size animation when note played
  let displaySize = this.size;
  if (this.isPlaying) {
    if (this.animationStartTime === null) {
      this.animationStartTime = millis();
    }
    const elapsedTime = millis() - this.animationStartTime;
    if (elapsedTime < this.animationDuration) {
      displaySize *= 1.2;

      this.currentColor = this.brightColor;
      fill(this.currentColor);
    } else {
      displaySize = this.originalSize;
      this.isPlaying = false;
      this.animationStartTime = null;

      // Reset the color
      this.currentColor = this.originalColor;
    }
  } else {
    displaySize = this.originalSize;
    this.isPlaying = false;
    this.animationStartTime = null;
  }

  fill(this.currentColor);

  noStroke();
  ellipse(this.x, this.y, displaySize);
}

// Call this method when the sound is played
soundPlayed() {
  this.isPlaying = true;
  this.animationStartTime = millis();
}

// check distance between generators and dots and draw lines if
they're close

```

```

checkDistance(generators, distanceThreshold, lineColor) {
  for (let generator of generators) {
    let distance = dist(this.x, this.y, generator.x, generator.y);
    if (distance < distanceThreshold) {
      push(); // Save the current drawing state

      stroke(lineColor);
      // Map distance to stroke weight: maximum weight at 0 distance,
minimum weight at the distanceThreshold
      let weight = map(distance, 0, distanceThreshold, 4, 0.5);
      strokeWeight(weight);
      line(this.x, this.y, generator.x, generator.y);
      pop(); // Restore the previous drawing state
    }
  }
}

function updateWhiteDotsScale() {
  const scale = scales[settings.scale];
  for (let whiteDot of whiteDots) {
    whiteDot.updateNoteIndex(scale);
  }
}

```

## octaveGenerator.js

```
class OctaveGenerator {
  constructor(x, y, octaveOffset = settings.octaveOffset) {
    this.x = x;
    this.y = y;
    this.octaveOffset = octaveOffset;

    this.size = 60;
    this.range = 200;
    this.color = 'orange';
    this.rings = 8;
    this.ringColor = 200;

    this.startingRingSize = 0;
    this.endingRingSize = this.range;

    this.whiteDots = [];
    this.ringWidth = this.range / this.rings;

    if (this.octaveOffset === 0) {
      this.color = 'orange';
    } else if (this.octaveOffset === 1) {
      this.color = 'blue';
    } else if (this.octaveOffset === -1) {
      this.color = 'green';
    }
  }

  labelWhiteDots() {
    for (const whiteDot of this.whiteDots) {
      const distance = dist(this.x, this.y, whiteDot.x, whiteDot.y);
      const ringIndex = Math.floor(distance / this.ringWidth);
      if (ringIndex >= 0 && ringIndex < 8) {
        whiteDot.beatsToPlay[ringIndex] = true;
      }
    }
  }

  // Setter method for the range
  setRange(value) {
    this.range = value;
    this.ringWidth = this.range / this.rings; // If you want the
    ringWidth to update when range changes
  }
}
```



```

}

// Add a method to check if a WhiteDot is within the range
isWhiteDotInRange(whiteDot) {
  const distance = dist(this.x, this.y, whiteDot.x, whiteDot.y);
  return distance <= this.range;
}

updateWhiteDotsInRange(whiteDots) {
  this.whiteDots = [];
  for (const whiteDot of whiteDots) {
    if (this.isWhiteDotInRange(whiteDot)) {
      this.whiteDots.push(whiteDot);
    }
  }
}

drawRings(currentEighthNote) {
  for (let i = 1; i <= this.rings; i++) {
    const opacity = (i === currentEighthNote+1) ? 255 : 10; // Set
opacity to 255 if it's the current beat, otherwise set it to 50

    stroke(this.ringColor, opacity);
    noFill();
    let ringRadius = i * (this.range/this.rings); // Adjust this value
to set the distance between rings
    ellipse(this.x, this.y, ringRadius * 2, ringRadius * 2);
  }
}

display() {
  this.drawRings(currentEighthNote);
  fill(this.color);
  ellipse(this.x, this.y, this.size);
  noFill();
  ellipse(this.x, this.y, this.range * 2);
}
}

```

## Retrowave.js

```
let gridOffset = 0;

// Space between the two planes
let spaceMargin=150

// Space between each gridline
let spacing = 50;

// Angle to create perspective higher -> more in the ground
let flattenPerspective = 60;

// Animation speed
let backgroundSpeed = 0.1;

function setupBackground() {
  createGradient();
}

function updateSpeed() {
  backgroundSpeed = speedSlider.value(); // Update the speed variable
  with the slider value
  flattenPerspective= speedSlider.value()*100+50
  spaceMargin= (speedSlider.value()*(-15) + 170
}

function createGradient() {
  gradient = createGraphics(width, height);
  gradient.clear();
  let gHorizon = color(0, 0, 0, 150);
  let gBottom = color(0, 0, 0, 0);

  for (let i = 0; i < height; i++) {
    gradient.stroke(lerpColor(gHorizon, gBottom, i / height));
    gradient.line(0, i, width, i);
  }
}

function drawBackground() {
  const vanishingPoint = height/2;
  let horizon=vanishingPoint;
  background(32, 13, 58,50);
  horizon=vanishingPoint+spaceMargin;

  // Draw vertical lines in perspective
  for (let x = -spacing; x <= width + spacing; x += spacing) {
```

```

    // Increase the spacing multiply to increase perspective
    let startX = map(x, -spacing, width + spacing, -spacing *
flattenPerspective, width + spacing * flattenPerspective);
    let endX = map(x, -spacing, width + spacing, -spacing, width +
spacing);
    stroke(70, 206, 189, 100);
    strokeWeight(2);
    line(endX, horizon, startX, height);
  }

  // Draw horizontal lines
  gridOffset = (gridOffset + backgroundSpeed) % spacing;
  for (let y = horizon + gridOffset; y <= height; y += spacing) {
    let opacity = map(y, horizon, height, 0, 255);
    stroke(70, 206, 189, opacity);
    strokeWeight(2);
    line(0, y, width, y);
  }

  // Draw horizon line
  stroke(70, 206, 189, 50);
  strokeWeight(1);
  line(0, horizon, width, horizon);

  // Draw the original image
  drawInverted();
  image(gradient, 0, 0);
}

function drawInverted() {
  const vanishingPoint = height/2;
  let horizon=vanishingPoint;
  horizon=vanishingPoint-spaceMargin;

  // Draw inverted vertical lines in perspective
  for (let x = -spacing; x <= width + spacing; x += spacing) {
    // Increase the spacing multiply to increase perspective
    let startX = map(x, -spacing, width + spacing, -spacing *
flattenPerspective, width + spacing * flattenPerspective);
    let endX = map(x, -spacing, width + spacing, -spacing, width +
spacing);
    stroke(70, 206, 189, 100);
    strokeWeight(2);
    line(endX, horizon, startX, 0);
  }
}

```

```
// Draw inverted horizontal lines
let invertedGridOffset = (gridOffset - backgroundSpeed) % spacing;
for (let y = horizon - invertedGridOffset; y >= 0; y -= spacing) {
  let opacity = map(y, 0, horizon, 255, 0);
  stroke(70, 206, 189, opacity);
  strokeWeight(2);
  line(0, y, width, y);
}

// Draw inverted horizon line
stroke(70, 206, 189, 50);
strokeWeight(1);
line(0, horizon, width, horizon);
}
```

## soundDesign.js

```
let thereminSynth;
let freqSlider, volSlider, typeSelect;
let freqLabel, volLabel, typeLabel;
// Option to load sample
// More sound design options.

function setup() {
  createCanvas(600,400);
  frameRate(30);

  // Create a Tone.js synth
  testSynth = new Tone.Synth().toDestination();
  wave = new Tone.Waveform(1024);

  // Set the initial position for the controls
  let controlX = width / 10 ;
  let controlY =height - height /4 ;
  let controlSpacing = 40;

  // Create a play button
  playButton = createButton('▶');
  playButton.position(controlX+10, controlY-20);
  playButton.mousePressed(playSound);
  playButton.addClass('play-button');

  // Create labels and sliders for frequency, volume, and oscillator type

  typeLabel = createP('Oscillator');
  typeLabel.position(controlX-140, controlY+80);
  typeLabel.addClass('slider-label');

  typeSelect = createSelect();
  typeSelect.addClass('custom-select');
  typeSelect.option('sine');
  typeSelect.option('square');
  typeSelect.option('triangle');
  typeSelect.option('sawtooth');
  typeSelect.position(controlX, controlY + 60);

  controlX += controlSpacing*2;

  volLabel = createP('Volume');
  volLabel.position(controlX - 50, controlY);
  volLabel.addClass('slider-label');

  volSlider = createSlider(-48, 0, -12, 0.1);
```

```

volSlider.position(controlX, controlY + 20);
volSlider.addClass('custom-slider');

controlX += controlSpacing*2;

// Create labels and sliders for Attack, Decay, Sustain, and Release
attackLabel = createP('ATT');
attackLabel.position(controlX, controlY);
attackLabel.addClass('slider-label');

attackSlider = createSlider(0, 2, 0.2, 0.1);
attackSlider.position(controlX, controlY + 20);
attackSlider.addClass('custom-slider');

controlX += controlSpacing;

decayLabel = createP('DEC');
decayLabel.position(controlX, controlY);
decayLabel.addClass('slider-label');

decayLabel.position(controlX, controlY);
decaySlider = createSlider(0, 2, 0.2, 0.01);
decaySlider.position(controlX, controlY + 20);
decaySlider.addClass('custom-slider');

controlX += controlSpacing;

sustainLabel = createP('SUS');
sustainLabel.position(controlX, controlY);
sustainLabel.addClass('slider-label');

sustainSlider = createSlider(0, 1, 0.5, 0.01);
sustainSlider.position(controlX, controlY + 20);
sustainSlider.addClass('custom-slider');

controlX += controlSpacing;

releaseLabel = createP('REL');
releaseLabel.position(controlX, controlY);
releaseLabel.addClass('slider-label');

releaseSlider = createSlider(0, 5, 5, 0.01);
releaseSlider.position(controlX, controlY + 20);
releaseSlider.addClass('custom-slider');
}

```

```

window.addEventListener('message', (event) => {
  const data = event.data;

  if (data.type === 'thereminSynth') {
    thereminSynth = new Tone.Synth(data.options);
    thereminSynth.toDestination(); // Connect the synth to the audio
    output
    // Call your function with the reconstructed thereminSynth
    setupSliders(thereminSynth);
  }
});

function draw() {
  background(320);
  drawADSRGraph();
  drawWaveform();
}

function setupSliders(synth) {
  // Create a function to send the updated synth parameters to the parent
  window
  const sendUpdatedSynth = () => {
    window.parent.postMessage({
      type: 'updatedSynth',
      options: {
        volume: synth.volume.value,
        oscillatorType: synth.oscillator.type,
        envelope: {
          attack: synth.envelope.attack,
          decay: synth.envelope.decay,
          sustain: synth.envelope.sustain,
          release: synth.envelope.release
        }
      }
    }, '*');
  };

  // Update the synth and send the updated parameters when a slider value
  changes
  volSlider.input(() => {
    synth.volume.value = volSlider.value();
    sendUpdatedSynth();
  });
}

```

```

typeSelect.changed(() => {
  synth.oscillator.type = typeSelect.value();
  sendUpdatedSynth();
});

attackSlider.input(() => {
  synth.envelope.attack = attackSlider.value();
  sendUpdatedSynth();
});

decaySlider.input(() => {
  synth.envelope.decay = decaySlider.value();
  sendUpdatedSynth();
});

sustainSlider.input(() => {
  synth.envelope.sustain = sustainSlider.value();
  sendUpdatedSynth();
});

releaseSlider.input(() => {
  synth.envelope.release = releaseSlider.value();
  sendUpdatedSynth();
});
}

function drawADSRGraph() {
  // Set up the canvas style
  push();
  translate(width / 8, height / 18); // Update the graph position
  background(23,9,41);

  // Draw grid lines and time labels
  stroke(255, 255, 255, 100);
  strokeWeight(1);
  textSize(12);
  fill(255);
  textAlign(CENTER, CENTER);

  // Display grid values in ms
  for (let i = 0; i <= width / 2 + 150; i += 50) {
    line(i, 0, i, height / 2);
    if (i >= 0) {
      text(i * 10, i, -10);
    }
  }
}

```



```

for (let i = 0; i <= height / 2; i += 20) {
  line(0, i, width / 2 + 150, i);
}

// Map ADSR values to the graph
let attackX = map(attackSlider.value(), 0, 2, 0, width / 4);
let decayX = map(decaySlider.value(), 0, 2, 0, width / 4);
let sustainY = map(sustainSlider.value(), 0, 1, height / 2, 0);
let releaseX = map(releaseSlider.value(), 0, 5, 0, width / 4);

// Draw the ADSR envelope with a futuristic style
strokeWeight(3);
stroke(255, 255, 255);
noFill();

// Attack region (red)
stroke(255, 0, 0);
line(0, height / 2, attackX, 0);

// Decay region (green)
stroke(0, 255, 0);
line(attackX, 0, attackX + decayX, sustainY);

// Sustain region (blue)
stroke(0, 0, 255);
line(attackX + decayX, sustainY, attackX + decayX + releaseX, height /
2);

// Release region )
stroke('#2A9D8F');
line(attackX + decayX + releaseX, height / 2, width / 2, height / 2);

// Draw gradient areas below each line
drawGradientArea(0, height / 2, attackX, 0, color('#2A9D8F'),
color('#48B9A6'));
drawGradientArea(attackX, 0, attackX + decayX, sustainY,
color('#1D6F67'), color('#2A9D8F'));
drawGradientArea(attackX + decayX, sustainY, attackX + decayX +
releaseX, height / 2, color('#143D3B'), color('#1D6F67'));

// Draw points at vertices
fill(255, 255, 255);
ellipse(0, height / 2, 8);
ellipse(attackX, 0, 8);
ellipse(attackX + decayX, sustainY, 8);
ellipse(attackX + decayX + releaseX, height / 2, 8);

// Reset canvas style

```

```

    pop();
}

// Helper function to draw gradient rectangle with a solid line at the
// bottom
function drawGradientArea(x1, y1, x2, y2, c1, c2) {
    let gradientStep = 0.01;
    noFill();

    for (let i = 0; i <= 1; i += gradientStep) {
        let interColor = lerpColor(c1, c2, i);
        let alphaValue = map(i, 0, 1, 255, 0); // Map the alpha value from
        255 to 0
        stroke(interColor._array[0] * 255, interColor._array[1] * 255,
        interColor._array[2] * 255, alphaValue);
        line(x1, lerp(y1, height / 2, i), x2, lerp(y2, height / 2, i));
    }

    // Draw a solid line at the bottom of the gradient
    stroke(c2);
    line(x1, height / 2, x2, height / 2);
}

function drawWaveform() {
    stroke(255);
    strokeWeight(1);
    noFill();

    // Connect the Waveform object to the synth
    thereminSynth.connect(wave);

    // Update the buffer to get the latest waveform values
    let buffer = wave.getValue();
    push();
    translate(0, height/2);
    beginShape();
    for (let i = 0; i < buffer.length; i++) {
        let x = map(i, 0, buffer.length, 0, width);
        let y = map(buffer[i], -1, 1, height / 4, 0);
        vertex(x, y);
    }
    endShape();
    pop();
}

// Callback function to play the sound
function playSound() {

```

```
    thereminSynth.triggerAttackRelease(thereminSynth.frequency.value,  
    '8n');  
}
```

```
// GUI SETTINGS

let selectedNoteIndex = 0;

const toolbarX = 10;
const circleSize = 40;

function drawOctaveOffsetToolbar() {
  const toolbarX = 50;
  const toolbarY = 500;
  const circleSize = 30;
  const circleSpacing = 10;
  const octaveOffsets = [-1, 0, 1];

  for (let i = 0; i < octaveOffsets.length; i++) {
    const circleX = toolbarX;
    const circleY = toolbarY + (circleSize + circleSpacing) * i;

    if (octaveOffsets[i] === settings.octaveOffset) {
      fill(255, 0, 0);
    } else {
      fill(255);
    }

    ellipse(circleX, circleY, circleSize);
    fill(0);
    text(octaveOffsets[i], circleX - 3, circleY + 3);
  }
}

function handleOctaveOffsetToolbarClick() {
  const toolbarX = 50;
  const toolbarY = 500;
  const circleSize = 30;
  const circleSpacing = 10;
  const octaveOffsets = [-1, 0, 1];

  for (let i = 0; i < octaveOffsets.length; i++) {
    const circleX = toolbarX;
    const circleY = toolbarY + (circleSize + circleSpacing) * i;

    if (
      mouseX >= circleX - circleSize / 2 &&
      mouseX <= circleX + circleSize / 2 &&
      mouseY >= circleY - circleSize / 2 &&

```

```

        mouseY <= circleY + circleSize / 2
    ) {
        settings.octaveOffset = octaveOffsets[i];
        return true; // Toolbar was clicked, return true
    }
}
return false; // Toolbar was not clicked, return false
}

function drawScaleToolbar() {
    const regionWidth = width / numRegions;
    const toolbarY = height - 100;

    for (let i = 0; i < scales[settings.scale].length; i++) {
        const circleX = (regionWidth * i) + (regionWidth / 2); // Center the
        circle within the region
        const circleY = toolbarY;

        if (i === selectedNoteIndex) {
            fill(255, 0, 0);
        } else {
            fill(255);
        }

        ellipse(circleX, circleY, circleSize);
    }
}
let toolbarClicked = false;

function handleToolbarClick() {
    const regionWidth = width / numRegions;
    const toolbarY = height - 100;

    for (let i = 0; i < scales[settings.scale].length; i++) {
        const circleX = (regionWidth * i) + (regionWidth / 2);
        const circleY = toolbarY;

        if (
            mouseX >= circleX - circleSize / 2 &&
            mouseX <= circleX + circleSize / 2 &&
            mouseY >= circleY - circleSize / 2 &&
            mouseY <= circleY + circleSize / 2
        ) {
            selectedNoteIndex = i;
            return true; // Toolbar was clicked, return true
        }
    }
}

```

```

    return false; // Toolbar was not clicked, return false
}

function updateNodeSpeeds() {
  for (let whiteDot of whiteDots) {
    whiteDot.speed = random(settings.speed-1, settings.speed);
  }
}

function setupGui() {
  const gui = new dat.GUI({ autoPlace: false });
  gui.domElement.id = 'gui-container';
  document.body.appendChild(gui.domElement);

  const params = {
    bpm: Tone.Transport.bpm.value,
    volume: 50
  };

  // BPM
  gui.add(params, 'bpm', 10, 240).step(1).name('BPM').onChange((value) => {
    Tone.Transport.bpm.value = value;
  });

  // Scale option
  gui.add(settings, "scale",
Object.keys(scales).name("Scale").onChange(() => {
    if (settings.scale === "Random") {
      scales.Random = getRandomIntervals();
    }
    // Update the current scale
    currentScale = scales[settings.scale];
    updateWhiteDotsScale();

    // Recalculate the notes in the scale and the number of regions
    scaleNotes = currentScale.map(interval => {
      let frequency = rootFrequency * Math.pow(2, interval / 12);
      return Tone.Frequency(frequency).toNote();
    });

    numRegions = scaleNotes.length;

```

```

    // Reset the selected note index
    selectedNoteIndex = 0;
  });

  // Node speed
  const speedController = gui.add(settings, "speed", 0.01,
5).step(0.01).name("Node Speed");
  speedController.onChange(updateNodeSpeeds);

  // Freeze nodes
  gui.add(settings, "lockNodes").name("Freeze Nodes");
}

function setupToolbar() {
  interactionMode = "create";

  let isMuted = false;

  const muteButton = document.getElementById("mute-button");
  muteButton.addEventListener("click", (event) => {
    event.stopPropagation(); // prevent event from reaching the canvas
    isMuted = !isMuted; // Toggle the mute state

    if (isMuted) {
      muteButton.classList.add("active");
      Tone.Destination.mute = true;
    } else {
      muteButton.classList.remove("active");
      Tone.Destination.mute = false;
    }
  });

  const whiteDotButton = document.getElementById("create-button");
  whiteDotButton.addEventListener("click", (event) => {
    event.stopPropagation();
    interactionMode = "create";
    whiteDotButton.classList.add("active");
    thereminButton.classList.remove("active");
  });

  const thereminButton = document.getElementById("theremin-button");
  thereminButton.addEventListener("click", (event) => {
    event.stopPropagation();
    interactionMode = "theremin";
    thereminButton.classList.add("active");
    whiteDotButton.classList.remove("active");
  });
}

```

```
whiteDotButton.classList.add("active");  
}
```