Informe del trabajo FCG

Bruno Weisz

February 2022

1 Idea del trabajo

La propuesta elegida para este trabajo fue realizar un catálogo de visualizaciones musicales sobre una pista de audio cargada por el usuario. Para el desarrollo del mismo se hizo uso de la librería **three.js** y de la **Web Audio API**.

En total se realizaron 5 posibles visualizaciones que serán detalladas más adelante en el informe, con la posibilidad de personalizar desde la UI la resolución y el estilo de las mismas.

1.1 Web Audio API

La **Web Audio API** de javascript provee un sistema versátil que nos permite manipular y estudiar audio desde una aplicación web. Para crear un modelo de audio, debemos crear una instancia de **AudioContext**, que contiene un **audio source** (en este caso un **MediaElementAudioSourceNode** ya que la fuente de audio provendrá de un elemento $< audio >$ del DOM) y un **destino**, en este caso los altavoces por default.

La Web Audio API tiene un modelo de enrutamiento modular, en el cual el stream de audio puede pasar por una serie de **nodos** hallados entre la fuente y el output del audio context. No nos interesa para el alcance de este trabajo modificar el audio, sin embargo se hizo uso de un único nodo intermedio: **AnalyserNode**. Instancias de este nodo proveen una interfaz que nos permite saber en todo momento, dado un stream de audio, el estado de la intensidad de las frecuencias (obtenido a partir de una Fast Fourier Transform) y un analisis del tiempo-dominio. Es importante notar que podemos personalizar el **fftSize**, es decir, la cantidad de intervalos de frecuencia en la cual el AnalyserNode va a separar el stream de audio. Esto nos permite tener un mayor o menor nivel de detalle de las frecuencias, que se verá traducido en las visualizaciones en una mayor o menor resolución de la imagen.

La información de la intensidad de las frecuencias o del estado de la onda será recuperada en forma de instancias de **Uint8Array**, cuyo tamaño (por restricción de la API) deberá ser una potencia de 2. En otras palabras, la intensidad de cada intervalo de frecuencias será un valor entero entre **0** y **255**.

2 Generalidades del modelo

Una idea general del modelo desarrollado es contar con la menor cantidad de variables globales posible, con la idea de obtener un modelo modular y extensible. Probablemente en un futuro decida extender la funcionalidad del modelo, una sección de posibles ideas se encuentra al final del informe. El **canvas** y la variable encargada de almacenar el **animationFrameID** son las únicas variables globales en el modelo.

La ejecución comienza cuando desde 'index.js' le indicamos al módulo **EventHandler** que configure el cargado del archivo de audio, para poder asociarlo luego al stream de audio que vamos a analizar. Una vez que el archivo de audio es cargado, se le indica al **AudioManager** que comienze un AudioContext, usando como input stream el audio que fue previamente cargado. A este Audio Context se le agrega un nodo intermedio de análisis, que será pasado como parámetro junto con el nombre de la visualización default al módulo **Visualization**. Este último cuenta con 3 mensajes principales:

1. **visualize()**: recibe el nombre de una visualización y una instancia de AnalyserNode. Se encarga de llamar a una `visualization setup` para la visualización elegida (tarea a cargo del módulo EventHandler) y llamar a la función encargada de finalmente mostrar la visualización para la pista de audio. Cada visualización está a cargo de su propio módulo, cada uno de los cuales provee un único mensaje **draw()**. También, cada módulo de visualización sabrá manejar un objeto de **settings**, que es pasado como parámetro a la función 'draw' junto con el AnalyserNode. Estos settings contienen información sobre la resolución y el estilo de la visualización.

2. **changeDivission()**: recibe un nuevo valor para la resolución de la visualización que se esta presentando en ese momento, y vuelve a llamar a la función 'draw()' del modulo correspondiente, habiendo modificado los settings de la misma.

3. **changeStyle()**: la idea es similar a la anterior, pero esta vez modificando el estilo en el objeto de configuración actual de la visualización.

Natuarlmente, la explicación previa es un resumen muy superficial del código. Para ver el detalle, los archivos involucrados se encuentran en la carpeta **/src**.

3 Visualizaciones

3.1 Algunas generalidades

Todos los módulos de visualización se dividen en 2 partes:

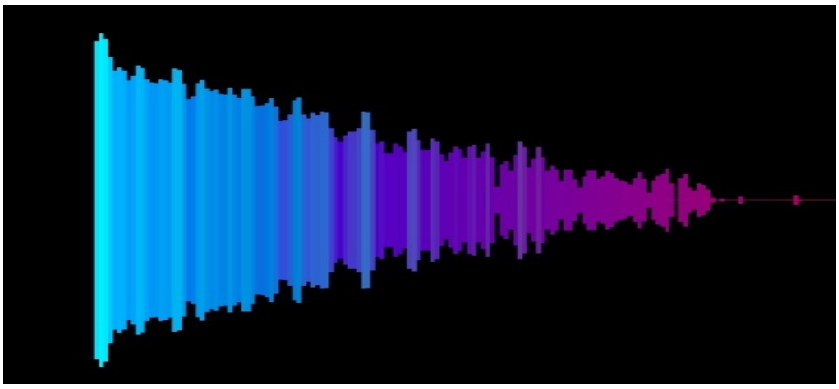
1. **Configuración Inicial:** Se encarga de inicializar y configurar todo lo relacionado con **three.js** para cada visualización en particular. Entre las tareas se encuentra:
   - Crear el **Renderer**.
   - Crear la **Cámara de Perspectiva**.
   - Crear la **Escena**.
   - Crear la **Iluminación**.
   - Establecer la posición de la cámara y de la iluminación. Estas dependen en general de la resolución de la visualización. En todos los casos, la mínima unidad de posicionamiento global en la escena es de 1, por lo tanto si quisiéramos por ejemplo tener 1024 barras de frecuencia, cada una tendría un ancho de 1, y por lo tanto la cámara debe estar más alejada y la fuente de iluminación colocada acordemente.
   - Crear las estructuras necesarias para contener los objetos de la escena que van a ser renderizados en el render loop. Más detalle sobre esto en cada visualización particular.

2. **Render Loop:** En cada ciclo se debe...
   (a) Cada vez que cambia el tamaño de la pantalla, se debe readaptar el tamaño interno y externo del canvas, el aspecto de la camara, el tamaño del renderer y actualizar la matriz de proyección de la cámara.
   (b) Obtener los datos de las intensidades de las frecuencias del analyserNode, enviándole el mensaje **getByteFrequencyData**.
   (c) A partir de los datos de las frecuencias, modificar los **mesh** de la escena que están siendo renderizados en base a estos valores (su posición, color o tamaño según corresponda).
   (d) Enviar el mensaje **render()** al renderer, pasándole la cámara y la escena como parámetros.
   (e) Invocar al siguiente frame de animación usando $window.requestAnimationFrame()$.

3. **Observaciones:** La posición de la luz y de la cámara según la resolución, el escalado de los objetos según la distancia de la camara y el color de los materiales está a cargo del módulo **ThreeUtilities**.

3.2 Visualizaciones específicas

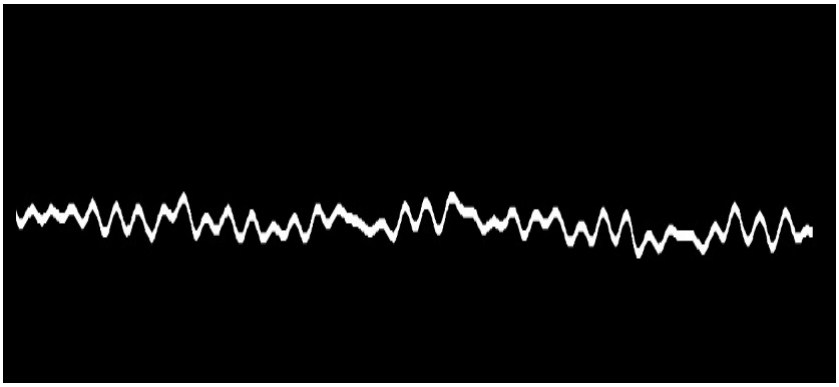
3.2.1 Frequency 2D



La idea de esta visualización es mostrar un gráfico de barras en 2 dimensiones que represente la intensidad de las frecuencias en un instante dado. Para esto se hizo uso del **PlaneGeometry**, una geometría primitiva de three.js, y **MeshBasicMaterial**, un material que no refleja la luz.

La idea general del render loop es mantener un array de **barMesh** (mesh creados a partir de la geometría y el material mencionados), posicionados de forma consecutiva, todos de altura y ancho unitarios. Cada mesh estará asociado a un rango de frecuencias y será escalado acordemente a la intensidad de el rango asociado.

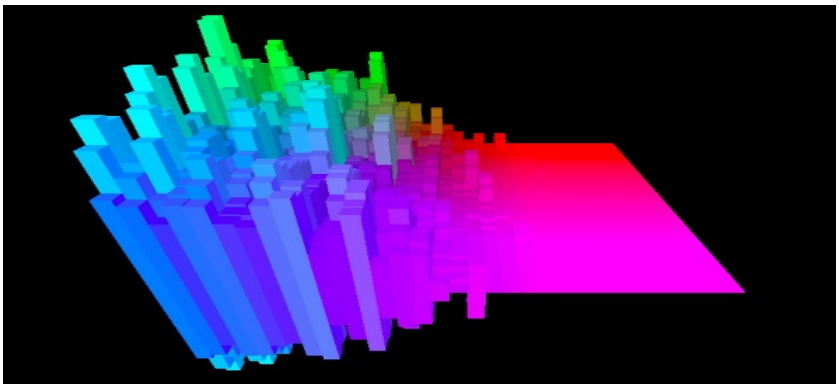
3.2.2 Osciloscope 2D



La idea de esta visualización es mostrar el estado de la onda sonora en un momento dado, también en 2D. Para esto se usó un **BufferGeometry** personalizado, más parecido al trabajo requerido para renderizar elementos usando WebGL sín three.js. Esta geometría requiere ciertos atributos que hay que calcular de forma manual: posición de los vértices, color de los vértices y triplas de índices de vértices que conforman los triángulos de la figura. Lo que buscamos es generar una cinta bidimensional.

Para el render loop, la idea es esencialmente la misma, solo que en lugar de enviar un mensaje al analyserNode para recuperar las frecuencias, se le manda el mensaje **getByteTimeDomainData** y se actualiza la altura de la cinta en la coordenada correspondiente.

3.2.3 Frequency 3D - Rectangulos

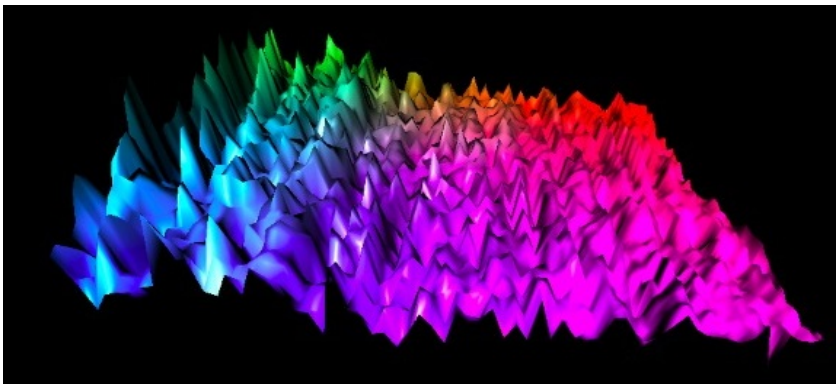


La idea de esta visualización no difiere mucho de la **Frequency 2D**, solamente que se se empaquetan los rangos de frecuencia por columnas para generar un efecto tridimensional, y para esto se hace uso de la primitiva de three.js **BoxGeometry**, con un tamaño inicial de (1,1,1) para todos los rangos de frecuencias.

La idea general del render loop es mantener un array de arrays de **barMesh** (mesh creados a partir de la geometría y el material mencionados), posicionados de forma consecutiva, al igual que en la frecuencia en 2D. En este caso también cada mesh también estará asociado a un rango de frecuencias y será escalado acordemente a la intensidad del rango de frecuencias asociado. El color también se decirirá (dependiendo del estilo seleccionado) a partir de la intensidad del rango.

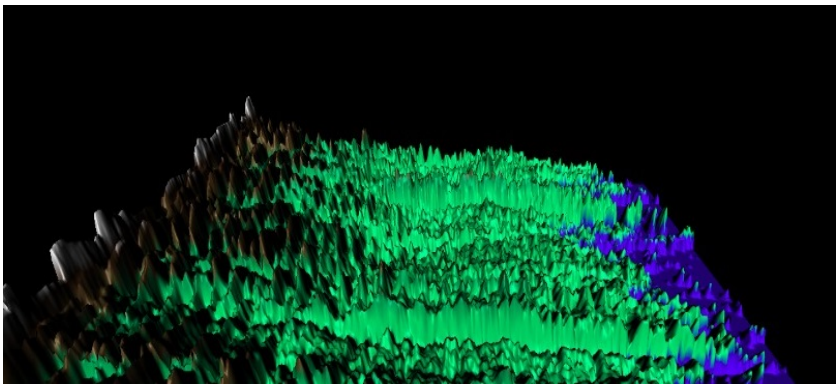
Obs: para visualizar de qué forma están dispuestos los rangos de frecuencias en el cuadrado, seleccione el audio 'test.mp3' de la carpeta de ejemplos.

3.2.4 Frequency 3D - HeightMap



Una de las opciones que más consideré para este trabajo fue la de generación de terrenos usando mapas de alturas (heightmaps), entonces cuando estaba haciendo este otro trabajo me pareció que sería una incorporación interesante generar mapas de alturas pero en lugar de usar generación pseudo-aleatoria, usar la información de las frecuencias del audio. La idea es muy similar a la de **Frequency 3D - rectangulos**, solamente que para hacer esto era necesario usar, al igual que en **Osciloscope 2D** un **BufferGeometry** personalizable, estableciendo manualmente los vertices, colores e índices para generar la malla deseada.

3.2.5 Frequency through time



Finalmente, se me ocurrió que sería interesante otra visualización que combinara **Frequency 2D** con **Frequency 3D - HeightMap**, para generar el efecto de la música siendo creada a medida que suena, pero persistiendo en el tiempo hasta que salga del rango de visión de la cámara, la cual avanza para mantenerse a una distancia constante del la posición en **Z** donde se genera una nueva linea de terreno musical.

Para esto usé una pseudo generación por lotes: cada lote es una mesh de a lo sumo **c** lineas de terreno. Los lotes están representados por instancias de **Badge**, que contienen la información de los vertices, color e indices de la malla tanto como el mesh de three.js en sí. Estos lotes son administrados por un **BadgeManager**, que contiene a todos los lotes, crea nuevos cuando es necesario y los elimina cuando se supera la máxima cantidad de lotes almacenables (para no saturar la memoria con una generación indefinidamente larga).

El módulo de visualización solamente debe crear un badgeManager en la etapa de setup, y en cada etapa enviarle un mensaje **addRow** con la información de las frecuencias, y este se encarga de agregar una linea nueva al mesh en curso, o crear un nuevo mesh para seguir avanzando. Un detalle necesario para evitar cortes en la imagen es que la última linea de cada badge es copiada como la primera linea del badge siguiente, generando una redundancia de vértices que permite su continuidad.

4 Cómo se podría extender este trabajo

La representación visual del sonido es un tema que puede extenderse infinitamente. A lo largo del proyecto se me ocurrieron muchas cosas que se podrían agregar o hacer de forma diferente, por lo tanto quiero dejar sentadas algunas de estas ideas y cuán difícil sería implementarlas dado el modelo planteado.

- Sobre la interfaz de usuario, se me ocurrió tener n canvas visibles con todas las representaciones en pantalla transcurriendo al mismo tiempo (si la GPU lo permite claro), y que al cliquear en una, su canvas ocupe toda la pantalla, mostrando las frecuencias con una mayor resolución. **Dificultad: Media**.
- Sobre nuevas visualizaciones:
    1. **Guitar Hero visualization:** una visualización que imite la dinámica del famoso juego guitar hero, o más en general, que muestre las notas como individuos a medida que aparecen, en lugar de mostrar un continuo de frecuencias donde los armónicos tienen la misma importancia que las notas fundamentales. Esto requeriría usar otra API que analice y permita distinguir las notas fundamentales a partir del archivo de audio. **Dificultad: Media-alta**.
    2. **Guitar Hero Playable:** misma idea que lo anterior, pero dando la posibilidad al usuario de jugar a pegarle a las notas en el momento que ocurre. El problema con esto es que habría que generar la nota antes de que ocurran, para que el usuario tenga tiempo de verla venir, y que suene recién cuando llegue a determinado punto en la pantalla. Esto requeriría agregar un nodo de Delay a continuación del analyserNode (dentro del modelo de la WebAudioAPI). **Dificultad: Alta**.
    3. **Diversificación de instrumentos:** hoy en día existen herramientas creadas con IA que permiten analizar una pista de audio y separarla en varias pistas, cada una con un instrumento distinto. Una de ellas por ejemplo es **demucs** (https://github.com/facebookresearch/demucs). Separar los instrumentos de una una pista de musica abre muchos caminos a la hora de visualizarla, pero la integración de la herramienta al proyecto (usando python) no es trivial. **Dificultad: Alta**.

- Sobre parámetros personalizables de las visualizaciones existentes:

    1. En las escenas 3D con posicionamiento estático, se podría establecer una rotación manual con el mouse o automática. Esto requeriría establecer un nuevo camino entre el EventHandler y los módulos de visualización, camino que permita modificar algún aspecto del módulo sin resetear la visualización. **Dificultad: Media**.
    2. Se pueden agregar más estilos de color de visualización. **Dificultad: muy facil**.
    3. En las escenas 3D se podría establecer una cámara de movimiento libre, para recorrer la escena como se desee. **Dificultad: media**.