UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Superior d'Enginyeries Industrial,
Aeroespacial i Audiovisual de Terrassa

DISSENY I PROGRAMACIÓ D'UN SAMPLER EN FORMAT WEB

TREBALL DE FI D'ESTUDIS

Document:
Memòria

Autor:
Pol Martínez Martín

Director:
Ignasi Esquerra Llucià

Titulació:
Grau en Enginyeria de Sistemes Audiovisuals

Convocatòria:
Gener 2023.

Agraïments

Els meus agraïments van dirigits principalment a totes les persones que han estat al meu costat i m'han recolzat durant aquesta etapa formativa.

En primer lloc, agrair a la meva família i amics, que m'han donat un suport molt necessari.

A la Universitat Politècnica de Catalunya i als professors que he tingut durant tot el grau, en especial també al meu tutor Ignasi Esquerra, per haver acceptat la proposta.

Resum

Aquest projecte es basa en el disseny i programació d'un sampler en format web orientat a ordinador. Un sampler és un instrument musical que utilitza gravacions (samples) per després ser reproduïdes mitjançant, un dispositiu MIDI o un seqüenciador amb l'objectiu de composar música i generar ritmes.

L'usuari pot carregar i utilitzar els seus propis samples, ja sigui amb un sol fitxer d'àudio o gravant des del micròfon de l'ordinador. També pot gravar les composicions aplicant-hi efectes en temps real, podent controlar certs paràmetres. Per altra banda, l'usuari pot descarregar-se en un fitxer d'àudio aquestes gravacions.

L'aplicació permet gravar melodies amb control de pitch mitjançant el teclat de l'ordinador per després també ser seqüenciades junt amb els altres instruments. El *pitch* representa la nota a la que s'està reproduint el sample.

Abstract

This project is based on the design and programming of a sampler in web and computer-oriented format. A sampler is a musical instrument that uses recordings (samples) to be played back later, by a MIDI device or a sequencer for the purpose of composing music and generating rhythms.

The user can load and use their own samples, by uploading audio files or by recording sounds from the computer's microphone. Also, the user can record the compositions applying effects in real time, by controlling its parameters with knobs. On the other hand, the user can download these recordings in audio file format.

The application allows you to record melodies with pitch control using the computer keyboard, to then also be sequenced with the other instruments. The pitch represents the note at which the sample is being played.

Índex

Índex de taules

Índex de figures

Llista d'abreviatures

· **DAW** - *Digital Audio Workstation*
· **VST** - *Virtual Studio Technology*
· **MIDI** - *Musical Instrument Digital Interface*
· **URL** - *Uniform Resource Locator*
· **API** - *Application Programming Interface*
· **AD** - *Analògic-Digital*

1. Introducció

1.1 Objecte

L'objectiu d'aquest projecte és dissenyar i desenvolupar un sampler en format web.

Un sampler és un instrument musical digital i electroacústic que utilitza *samples* per després ser reproduïts mitjançant, per exemple un seqüenciador o un teclat MIDI, amb l'objectiu de composar música i generar ritmes. En l'àmbit de la música, l'anglicisme *sample* (mostra en català) s'utilitza per referir-se a una mostra de so gravada en qualsevol tipus de dispositiu, per posteriorment ser utilitzada com a un instrument musical o per a una altra gravació [20].

Aquest software s'implementarà en format web utilitzant com a llenguatge base JavaScript i com a entorns principals P5.js amb el qual es desenvoluparà principalment tota la interfície gràfica d'usuari, i Tone.js, amb el que s'implementarà tota la part de reproducció, seqüenciació i processament d'àudio en el navegador.

Habitualment, els samplers poden interpretar qualsevol tipus de gravació i la majoria ofereixen possibilitats d'edició que permeten a l'usuari modificar i processar el so aplicant-hi efectes, per tant, un altre dels objectius del projecte és que l'usuari pugui penjar i utilitzar els seus propis samples, que en aquest cas seran fitxers d'àudio o gravacions fetes des del mateix navegador. També podrà aplicar-hi efectes, els quals podrà controlar mitjançant controls (knobs), amb la finalitat de fer l'experiència el més semblant possible a estar utilitzant aquest tipus d'instrument amb hardware real.

Una altra de les funcionalitats constarà d'una part de reproducció dels samples amb control de tonalitat (pitch) a mitjançant el teclat de l'ordinador.

L'usuari també podrà gravar ritmes des del propi navegador aplicant-hi efectes en temps real i podrà descarregar-los en fitxers d'àudio.

1.2 Abast

El desenvolupament de l'aplicació es divideix en diverses parts, sempre comprovant que cada part funcioni correctament un cop finalitzada. Cada fase representa una nova funcionalitat de l'aplicació, partint d'una base que serà un seqüenciador.

1. Seqüenciador bàsic de ritmes amb 8 pistes, 8 compassos i samples per defecte. Funcionament amb botons d'encès i apagat, cada un corresponent a 1 temps.
2. Desenvolupament de la funcionalitat que permet a l'usuari pujar i afegir els seus propis samples en forma de fitxer d'àudio, o en forma de gravació des del propi navegador a les diferents pistes.
3. Desenvolupament de la part d'efectes i dels controls que permeten a l'usuari manipular paràmetres en temps real a cada pista.
4. Desenvolupament de la funcionalitat de reproducció dels samples amb control de pitch.
5. Desenvolupament de la gravadora i de la funcionalitat de descàrrega de les gravacions en format d'àudio al navegador.
6. Anàlisi i proves del producte final.

1.3 Motivació

La principal motivació que m'ha portat a dur a terme aquest projecte és la gran passió que li tinc a la música des de que sóc petit. La música ha format part del meu dia a dia des de que era pràcticament un nadó. En aquelles èpoques mons pares col·leccionaven vinils de tot tipus i me'ls feien escoltar. A mida que van anar passant els anys vaig començar a fixar-me en la música electrònica, un estil que em va fascinar pel fet que permetia als artistes composar i crear sense límits.

Quan era adolescent vaig començar a produir música com a hobby, així doncs, a jugar amb emulacions de sintetitzadors i a elaborar els meus propis ritmes i melodies. Tot això ho feia amb una DAW, i amb les emulacions d'efectes natius del programa. Eines que han potenciat enormement la creativitat de molts artistes al llarg dels últims anys.

Va ser més tard quan em va entrar una gran curiositat per la tecnologia i per saber què havia darrere de tot aquell software que utilitzava. De manera que vaig decidir estudiar enginyeria. Durant el grau va néixer en mi un interès per la programació a mida que anava avançant en les assignatures i per tant, adquirint cada cop més coneixement més sòlid sobre el tema el meu interès va anar creixent. A més, vaig poder realitzar les pràctiques curriculars en una empresa de disseny i desenvolupament web, en aquell període de temps vaig adonar-me que aquell tema també m'atreia.

Moltes de les persones que volen iniciar-se en el món de la producció musical busquen eines intuïtives com samplers, que els permetin aprendre i practicar de manera eficaç, i també posar en funcionament les seves possibles habilitats musicals. Però hi ha inconvenients respecte a tot això, els instruments sobretot en format hardware solen tenir un preu molt elevat.

Per altra banda, la fabricació dels components electrònics que es necessiten per confeccionar qualsevol tipus de hardware d'àudio o en molts altres àmbits de l'electrònica, generen un gran impacte ambiental. La proliferació de productes electrònics ha donat lloc a importants deixalles electròniques als nostres abocadors, per tant, a l'augment de l'ús d'energia i a l'alliberament de materials perillosos al nostre medi ambient. Tot això es deu a que aquests components contenen una sèrie de materials altament contaminants com metalls pesats: mercuri, plom, cadmi, arsènic o antimoni, els quals són susceptibles de causar diversos danys per a la salut i per al medi ambient [12]. Per aquests últims motius, és primordial cercar alternatives que substitueixin aquests equips.

El desenvolupament de noves tecnologies en qualsevol dels seus àmbits, amb la finalitat de causar el menor impacte ambiental i d'evitar la generació de residus, és un fet fonamental a tenir en compte de cara a la nostra pròpia salut i per a la gestió de dels recursos naturals d'una manera sostenible.

La transformació digital ha suposat un abans i un després també en l'àmbit musical, ja que qualsevol artista en qualsevol part del món, pot tenir accés a la experimentació amb aquest tipus d'eines. El desenvolupament d'un codi pot ser utilitzat per milers, inclòs milions de persones sense necessitat d'equipament extra, atès que aquest mateix ja està inclòs dins del propi ordinador, és un avenç essencial de cara a un futur més sostenible.

2 Antecedents

2.1 Tecnologies musicals

2.1.1 Història de la producció musical

La capacitat d'enregistrar sons sempre ha estat un desig que perseguien grans pensadors i que ha donat forma a la història de la producció musical. No obstant això, aquest esforç no va agafar força fins a finals del segle XIX [2].

Thomas Edison va ser el primer en inventar un dispositiu que podia gravar i reproduir so, el fonògraf. Funcionava amb una botzina metàl·lica per la qual una persona parlava, el mateix so es transportava per una membrana i quedava gravat en un cilindre de metall [3]. El funcionament per reproduir el so emmagatzemat era invers, amb una agulla connectada a la botzina.

Figura 1 *Fonògraf*

Més d'una dècada més tard va aparèixer la cinta magnètica com a mitjà de suport alternatiu per enregistraments sonors.

Amb la distribució generalitzada del telèfon es va desenvolupar una tècnica d'enregistrament que va substituir les anteriors basades en el mètode acústico-mecànic: el micròfon de carboni.

Aproximadament al mateix temps, les botzines que s'utilitzaven per reproduir el so gravat van substituir-se per l'altaveu electrodinàmic, el qual tenia una millor relació senyal-soroll junt amb el micròfon i els vinils, permeten una notable millora de la qualitat del so.

La possibilitat d'enregistrar i reproduir so amb el mètode estereofònic es va fer possible en l'any 1958 quan va sortir al mercat. Aquest mètode permetia també reproduir enregistraments mono i es va consolidar com a estàndard de la indústria musical fins al dia d'avui, que és actualment el més utilitzat [2].

La gravadora multipista va suposar un abans i un després. A la dècada del 1950 va ser possible gravar dues pistes en temps real. A principis dels anys 70 van sorgir gravadores de 16 pistes, i poc després el número va augmentar a 24.

Figura 2 *Gravadora multipista Tascam Model 24*

El creixement dels equips va portar a que es poguessin enregistrar molts més sons en una cançó, i per tant, a superposicions freqüencials entre els instruments, fet que va impulsar el desenvolupament de compressors i equalitzadors.

La digitalització va suposar una revolució en la indústria musical, la música podia reproduir-se sense pèrdues. Això va portar al naixement del Compact Disc (CD) o del Digital Audio Tape (DAT). Durant la dècada dels 80 també es van inventar les taules de mescles digitals, que suposaven una gran avantatge perquè permetien allotjar nombrosos canals en un espai petit, ja que no tots necessiten el seu conjunt de controls. Com a resultat es van optimitzar els costos de producció.

Donada l'arribada de les taules de mescles digitals va sorgir la necessitat de transmetre i emmagatzemar l'àudio digital. El mètode PCM es va convertir en el principal per emmagatzemar àudio, amb el qual estan basats alguns formats de fitxer d'àudio com el WAV, utilitzat gairebé en totes les DAWs de l'actualitat.

El format MIDI (Musical Instruments Digital Interface) es va introduir a principis dels anys 80 amb l'objectiu de crear un intercanvi d'informació de control entre instruments electrònics: per controlar els instruments VST i per sincronitzar sintetitzadors analògics amb la DAW [2]. Aquest s'ha consolidat fins avui com a estàndard de la indústria.

Més tard, quan els ordinadors domèstics van tenir prou potència de càlcul van aparèixer els primers programes d'edició per processar àudio de forma digital. Cosa que va fer que apareguessin els actuals *homestudios* .En els habitatges moderns els avenços tecnològics dels últims 150 anys es combinen només en uns quants dispositius. En l'àmbit de la producció musical moderna passa exactament el mateix, el component essencial és un ordinador amb la suficient potència de càlcul per reproduir simultàniament diverses pistes i també els efectes assignats a cada una.

2.1.2 Tecnologies Musicals Actuals

La base dels actuals sistemes de seqüenciació i gravació d'àudio digital constitueix, per un costat, el protocol MIDI, desenvolupat com a llenguatge amb el que poder controlar el so de diferents instruments musicals electrònics; per altre costat, la tecnologia d'àudio digital,

permetent convertir un so analògic en una combinació de codi binari que el representi, amb la possibilitat de manipular-lo i modificar-lo a través d'un programa informàtic.

El desenvolupament de les estacions de treball d'àudio digital (DAW) va ser el primer pas en l'establiment dels ordinadors com a eina principal de creació musical. Aquests programes s'han anat convertint amb el temps en sistemes complets de producció musical, tenint en compte tot el procés que comporta. Incorporen eines que engloben des de l'acció de composar o editar partitures, fins a gravació, síntesi, mescla i masterització [5].

Figura 2.1.5 *Captura de pantalla d'una DAW (Ableton Live 11)*

La virtualització del tractament del so dels últims anys, després de que les gravadores de cinta fossin reemplaçades, ha arribat junt amb els *plug-ins*, que són programes que s'insereixen dins les DAWs d'igual manera a com es faria amb un taula de mescles analògica. Aquests són capaços de reproduir el comportament en la manera de processar el so d'igual manera a com ho faria un equip analògic constituït per compressors, equalitzadors o altres efectes. Tot això ha ampliat les possibilitats del processat i modelat del so, un exemple és la reverberació per convolució, que permet simular el comportament acústic de qualsevol espai. També ho són els *plug-ins* de processat tonal, que permeten retocar l'afinació de la veu [5].

La pròpia taula de mescles també ha sigut substituïda en la pròpia DAW, i a diferència dels equips analògics donen la possibilitat de mesclar desenes de pistes simultàniament podent automatitzar tots els paràmetres en temps real.

2.1.3 Noves eines

El protocol MIDI ha permès als artistes gravar seqüencies de dades per a que siguin reproduïdes no només per sintetitzadors, sinó per qualsevol tipus d'instrument. El sampler

n'és un exemple, que va introduir la possibilitat de tractar qualsevol so com a una nota musical.

El desenvolupament d'instruments virtuals substitueixen una font sonora real, com podria ser un instrument acústic, per una de virtual, la qual neix dins dels mateixos ordinadors mitjançant la síntesi digital del so o la reproducció de mostres. Un exemple molt avançat d'això podria ser la que ha realitzat *Vienna Symphonic Library*, que consta de la simulació de tots els instruments d'una orquestra simfònica. Aquesta llibreria consta d'una base de dades amb cada una de les notes de cada un dels instruments gravades amb tots els estils possibles i amb algorismes que controlen les transicions entre notes per fer-les sonar el més natural possible. En total, la llibreria supera el milió se mostres, fent-la la llibreria més extensa i avançada pel que fa a una orquestra clàssica [6].

La intel·ligència artificial també està tenint un paper important en l'àmbit musical. *Diffusion* és una tècnica d'aprenentatge automàtic per generar imatges. DALL-E 2 i Stable Diffusion són els models més destacats que funcionen substituint gradualment el soroll visual amb la que l'IA creu que hauria de semblar una indicació. Riffusion és el model d'intel·ligència artificial que composa música visualitzant espectrogrames. La idea va sorgir per part de Seth Forsgren i Hayk Martiros i per la seva afició a la música. Es van preguntar si seria possible que Stable Diffusion creés un espectrograma a partir d'una entrada per frase, amb prou fidelitat per convertir-la després en àudio [7]. Els espectrogrames són representacions visuals d'una senyal d'àudio que mostren l'amplitud de les freqüències en temps. El procés té algunes pèrdues, però és el suficientment acurat com per detectar notes i instruments i té l'avantatge de que es pot fer el mateix procés a la inversa. Els creadors d'aquesta idea van generar molts espectrogrames i els van etiquetar amb termes rellevants com "piano jazz" o "guitarra blues" per després ser utilitzats per entrenar el model.

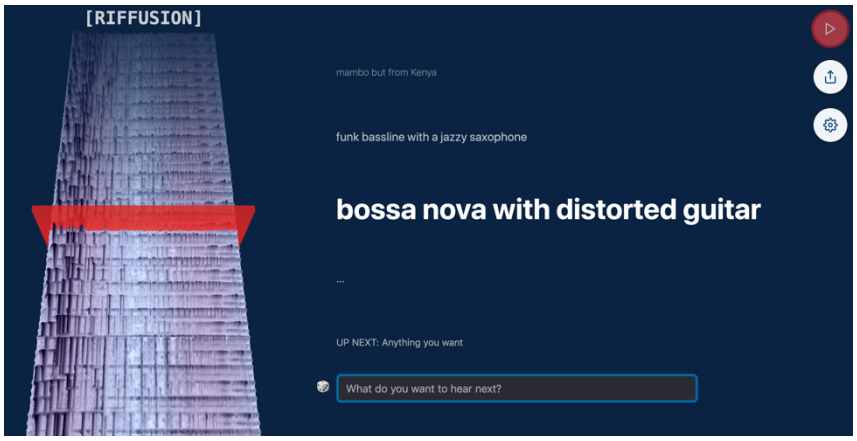

Figura 3 *Captura de pantalla de Riffusion en format web*

A part de les eines i instruments que s'han comentat tant de tipus hardware com software, també es possible crear-ne en format web. L'API d'àudio web proporciona un sistema potent i versàtil per controlar l'àudio web, que permet als desenvolupadors triar fonts d'àudio, aplicar efectes i també ofereix de síntesi i de reproducció d'àudio.

L'API d'àudio web implica la gestió d'operacions d'àudio dins d'un context i està dissenyada per realitzar aquestes operacions amb nodes connectats entre si, que s'uneixen formant una cadena amb entrades i sortides. Normalment comencen per una o més fonts que proporcionen mostres d'àudio, que són matrius amb valors d'intensitat de so en el temps. Les sortides dels nodes poden ser enllaçades a entrades d'altres, les quals poden barrejar i modificar els fluxos de mostres d'àudio. Un cop el so s'ha processat amb l'efecte o efectes desitjats, podem enllaçar les mostres resultants amb l'entrada d'una destinació que per exemple, poden ser uns altaveus o auriculars. Aquest disseny modular proporciona la flexibilitat per crear funcions d'àudio complexes amb efectes dinàmics [9].

Figura 4 *Esquema de fluxe de dades de Web Audio API*

2.2 Els samplers

2.2.1 Història

Els precursors del sampler van ser el *Chamberlin* i el *Mellotron*. El *Chamberlin* va ser inventat als Estats Units durant la dècada del 1950 per Harry Chamberlin, amb l'objectiu de replicar sons d'orquestra per cantar en reunions familiars. A principis de la dècada del 1960 un venedor contractat pel propi Harry Chamberlin, anomenat Bill Fransen va portar la idea al Regne Unit i la va presentar com a pròpia, fent néixer l'interès per part de l'empresa Mellotronics, que va fabricar 2000 unitats amb el nom de *Mellotron*, que copiava el concepte del seu antecessor Chamberlin però incorporant algunes millores tècniques [1]. En els dos casos es tractava d'instruments que reproduïen un enregistrament d'una cinta que corresponia a una nota d'un instrument (violí, orquestra, veu...). Tots dos tenien un principi de funcionament semblant al del sampler però no eren instruments digitals i, a més, no era possible enregistrar-hi nous sons. El *Mellotron* era un teclat capaç de reproduir en tres

Figura 5 *Mellotron*

canals cintes pre-gravades accionades amb les dues mans, amb les quals es podien seleccionar ritmes diferents. Va ser un dels primers teclats elèctrics i pot considerar-se com l'antecedent directe del sampler, donat que utilitzava bucles (*loops*) de cinta per crear nous sons [4].

El primer sampler digital va ser el Computer Music Melodian (1976). Dissenyada per Harry Mendal. És conegut com el primer sampler digital disponible comercialment. Incloïa funcions de conversió de digital a analògic i d'analògic a digital cablejats a mà. Era monofònic, tenia un convertidor AD de 12 bits, i una freqüència de mostreig de 22KHz. També permetia la compatibilitat amb sintetitzadors analògics, podent-los sincronitzar amb el pitch [1].

El primer sampler digital polifònic va ser el Fairlight CMI (Computer Music Instrument), creat per Peter Fogel i Kim Ryrie va ser originalment dissenyat per crear sons modelant la forma d'ona. No obstant això, la seva potència de processament era incapaç d'aconseguir-ho, així que ho van provar amb un so gravat i el resultat va ser exitós.

Figura 6 *Fairlight CMI (1979)*

Publicitat per la companyia japonesa Akai en col·laboració amb Roger Linn, l'Akai MPC60 va ser el primer model d'Akai no muntat en bastidor que es va vendre. També va ser un dels primers samplers a incloure coixinets d'activació sensibles al tacte [1]. Els 16 pads de velocitat podien emmagatzemar 4 bancs de sons, amb una freqüència de mostreig de 40KHz.

Figura 7 *Akai MPC60 (1988)*

A principis dels anys 90, els avenços en la potència de processament i la capacitat de memòria van fer possible l'execució de samplers en format software. Tot i que les samplers en format hardware encara s'utilitzen, els samplers en format software solen incloure's amb DAWs, especialment com a VST o plug-in, fent que es puguin utilitzar juntament amb altres mòduls, efectes de so i emulacions de sintetitzadors.

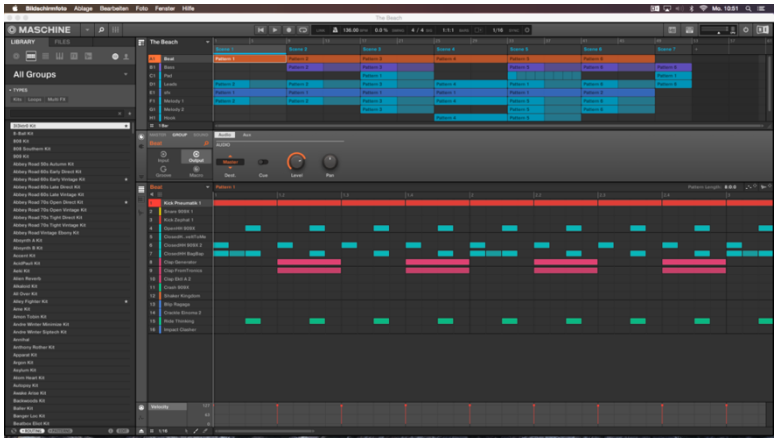

Figura 8 *Native Instruments Maschine*

El desenvolupament del sampler digital va tenir un efecte significatiu en el gènere del Hip-Hop als anys 80, on el mostreig d'altres discos a través de mitjans no digitals, com ara els tocadiscos, era una part crucial en com es creava una cançó d'aquell estil. Molts productors de Hip-Hop van adoptar aquesta nova tecnologia. Els permetia editar les mostres que estaven utilitzant per primera vegada, canviant el to i longitud d'aquesta. El seu principal avantatge era que l'edició es va imprimir en el so, a diferència d'un DJ que hauria d'arrossegar manualment el vinil més lentament o més ràpid per crear el canvi de to i velocitat [1].

El looping també es va fer molt més fàcil amb l'avenç de la tecnologia, ja que es tractava d'establir punts de bucle on quant es reproduïa una mostra, a diferència d'utilitzar un tocadiscos on si volíem aconseguir això s'havia de col·locar el vinil en el punt just i en el moment adequat per obtenir un bucle perfecte.

També va ser possible posar en capes múltiples mostres unes sobre les altres, fent possible el fet de poder formar tot el ritme d'una cançó amb una sola peça de hardware.

2.2.2 Funcionament

La interfície de control d'un sampler modern es basa en un teclat midi que normalment són botons amb forma quadrada sensibles a la velocitat. Cada un d'aquests botons té un *pitch* assignat. Al grup de notes a les qual s'assigna un sample se les sol denominar en anglès *keyzone* i el conjunt de les zones resultants, *keymap* [1]. La interfície sol incloure també un control d'efectes per processar els sons en temps real mitjançant *knobs*. Els *knobs* són dispositius rotatius que s'utilitzen per ajustar manualment l'entrada d'un sistema mecànic/elèctric, de manera que el grau de rotació correspon al número d'entrada del mateix sistema [8]. S'utilitzen en dispositius quotidians com podrien ser radios, estufes... però en l'àmbit de l'electrònica musical normalment solen ser potenciòmetres.

Figura 9 *Esquema de Korg Electribe 2 Sampler*

A la base d'aquest instrument hi ha els samples, enregistraments individuals de qualsevol so gravats amb una taxa i resolució particulars. Hi ha un *pitch* central indica la freqüència actual de la nota enregistrada. Es poden seqüenciar els ritmes mitjançant la definició dels punts on es desitja que sonin. Això es fa activant els botons o desactivant-los en certs punts a les columnes de botons. Cada una com a regla general, representa un temps i cada quatre un compàs.

3 Desenvolupament

Per arribar al resultat final, l'aplicació s'ha desenvolupat en diferents fases, cada respectiva fase representa una nova funcionalitat de l'aplicació. Primerament es comença per una fase de plantejament previ, on es mostren tots els requeriments necessaris per poder dur a terme el projecte i finalment acaba amb una fase d'anàlisi i proves de l'aplicació.

El codi font es pot obtenir a: **https://github.com/polmartmart/TFG**

3.1 Plantejament previ

Abans de començar amb el desenvolupament de l'aplicació es planteja quins són els requeriments bàsics per fer possible el projecte:

· **Ordinador:** Eina imprescindible per al desenvolupament de l'aplicació. Per dur a terme la programació s'utilitza un ordinador amb sistema operatiu Mac.

· **Visual Studio Code:** Editor de codi. Encara que no és imprescindible, s'ha instal·lat l'extensió **Live Server** dins de l'editor, que actualitza la pestanya del navegador cada cop que s'actualitza qualsevol part del codi, amb l'objectiu de fer més còmode el desenvolupament.

· **JavaScript:** Llenguatge de programació en el qual es basa tot el projecte. JavaScript és un llenguatge interpretat amb suport per a la programació orientada a objectes.

· **Entorns/Llibreries:**

  · **Node.js:** Entorn d'execució de JavaScript construït amb V8, el motor de JavaScript de Chrome. En el cas d'aquest projecte Node.js s'utilitza principalment per fer proves fora del navegador mitjançant la línia de comandes.

  · **P5.js:** P5.js és una llibreria JavaScript orientada a la programació creativa inspirat en l'entorn de programació Processing que ofereix la potència i simplicitat de Processing però enfocat a la integració en format web. Ens proporciona funcionalitats de dibuix d'alt nivell i també facilita la interacció de l'usuari amb els elements de l'aplicació. La llibreria P5.Sound ho amplia amb les funcionalitats d'àudio que inclou entrada, reproducció, anàlisis i síntesis d'àudio.

  Per poder utilitzar les respectives llibreries s'han de referenciar al fitxer index.html en una etiqueta <*script*>.

  En concret s'utilitzen dos llibreries dins de P5.js:

    · **p5.min.js:** És la versió comprimida de la pròpia llibreria de p5.js. En desenvolupament web sovint les llibreries es presenten en dues versions. La versió comprimida és essencialment una versió extremadament compacta del mateix codi. Com que no alterem cap part de la llibreria, fem servir aquesta versió.

    · **p5.dom.min.js:** Ens permet afegir elements DOM (Document Object Model) en l'aplicació. Aquests elements inclouen determinats controladors com ara botons, sliders, camps d'entrada, etc. Els quals seran essencials en l'aplicació.

    · **p5.sound:** Llibreria de p5.js basada en l'API d'àudio web que inclou àudio d'entrada, reproducció repetida, anàlisi i síntesi.

  · **Tone.js:** Es tracta d'un entorn basat en l'API d'àudio web per crear música de forma interactiva al navegador. L'arquitectura de Tone.js pretén ser familiar tant per als músics com per als programadors d'àudio que creen aplicacions d'àudio basades en web. Al més alt nivell, la llibreria ofereix funcions comunes de DAW (Digital Audio Workstation)

com per exemple un transport global per sincronitzar i programar esdeveniments musicals. Aquestes últimes són les essencials per la reproducció i la repetició sincronitzada dels samples que es fan servir en l'aplicació.

A més, Tone proporciona blocs de construcció d'alt rendiment per crear sintetitzadors i efectes, els quals també són essencials per al processament de l'àudio de l'aplicació [10].

· **KnobMakerC.js:** Llibreria que dibuixa knobs controlables pel *mouse* de l'usuari. Ens retornen valors en funció de la posició del punter del cercle els quals són assignats als efectes. El knob és un element primordial de l'aplicació perquè l'usuari pugui controlar els paràmetres dels diferents efectes al seu gust.

3.1.1 Funcionament d'una aplicació desenvolupada amb P5.js

Tenint en compte que P5.js és una interpretació de Processing però en format web, el funcionament base de les seves aplicacions és gairebé el mateix.

Primer, es crea una carpeta on es que serà l'espai de treball, on tindrem tots els fitxers necessaris perquè l'aplicació funcioni. Es crea un fitxer en format *js* (JavaScript) que podem anomenar com vulguem, en el nostre cas l'anomenem *sampler.js*. Aquest fitxer, inicialment ha de tenir dues funcions: *draw()* i *setup()*. *Setup()* només s'executa una vegada i ho fa quan s'inicia el programa, *draw()* s'executa contínuament mentre l'aplicació s'està executant.

Com estem treballant en una aplicació web, és imprescindible que la carpeta del projecte inclogui també un fitxer *index.html*. Dins de l'etiqueta <*head*> hi ha la crida a les llibreries que utilitzem mitjançant també l'etiqueta <*script*>. Aquesta última també s'utilitza per referenciar *sampler.js* dins l'etiqueta de <*body*>.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Sampler</title>

    <script src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/0.6.1/p5.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/0.6.1/ad-
dons/p5.dom.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/tone/14.8.9/Tone.js" type="text/
javascript"></script>
    <script language="javascript" type="text/javascript" src="KnobMakerC.js"></script>

    <link rel="stylesheet" type="text/css" href="style.css">
    <link rel="shortcut icon" href="#">
  </head>

  <body>
    <script src="sampler.js"></script>
  </body>
</html>
```

3.2 Desenvolupament del seqüenciador

El primer pas un cop tenim l'essencial és dibuixar una interfície de botons semblant a la d'un sampler amb la qual l'usuari pugui interactuar. L'objectiu és dissenyar una quadrícula, amb la que els quadrats que la integrin representin interruptors d'encès i apagat, i que l'usuari pugui tenir interacció amb aquests botons.

Un cop dibuixada aquesta interfície, es desenvolupa la part de d'àudio. Cada fila de la quadrícula representa un instrument, cada columna de quadrats representa un temps, si el botó està activat en aquell punt, l'instrument carregat en aquella fila sonarà.

3.2.1 Interfície

Des de la funció *draw()* cridem a *drawGrid()*, funció que ens dibuixa la quadrícula per línies horitzontals i verticals. Consisteix en un bucle que va crida a la funció *line(x1, y1, x2, y2)* [11], tal com diu el seu nom, ens dibuixa una línia amb unes coordenades donades en píxels. Tal i com podem observar en la *figura 10*, el sistema de coordenades està invertit.

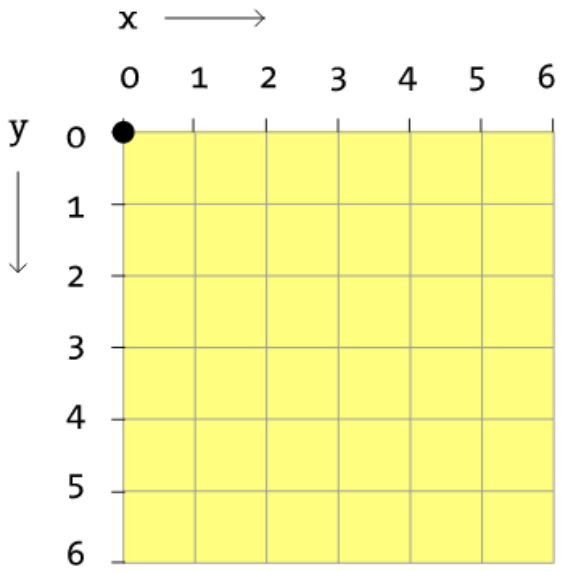

Figura 10 *Sistema coordenat per píxels en una pantalla d'ordinador*

```
function drawGrid() {
  //linies horitzontals
  for(let j=0; j<nTracks+1; j++) {
      stroke(200);
      strokeWeight(3);
      line(origen,j*cellWidth+origen,cellWidth*32+origen,j*cellWidth +origen);
  }

  //linies verticals
  for(let i=0; i<nSteps+1; i++) {
    if (i % 4 === 0 && i != 0 && i != 32) {
      stroke(50);
```

```
      strokeWeight(4);
      line(i*cellWidth+origen, origen, i*cellWidth +origen,cellWidth*8+origen);
    }
    else {
      stroke(200);
      strokeWeight(3);
      line(i*cellWidth+origen, origen, i*cellWidth +origen,cellWidth*8+origen);
    }
  }
}
```

Les variables *cellWidth*, *origen*, *nTracks* i *nSteps* són variables globals declarades al principi del codi amb la sintaxi ***let nom_de_la_variable = valor (opcional)***: *cellWidth* és l'amplitud del costat dels quadrats, *origen* és una variable que representa un origen de coordenades que utilitzo per centrar elements en la pantalla, *nTracks* i *nSteps* són respectivament les pistes, és a dir, els instruments que podran sonar alhora que són 8, i els passos o temps que en total són 32, formant així 8 compassos (4 temps cada compàs).

Utilitzant *stroke()* passant-li un enter com a paràmetre aconseguim assignar un color a la línia que estem dibuixant en un valor en escala de gris. En aquest cas dibuixem les línies que siguin 4 o múltiples de 4 amb un valor d'escala de gris de 50, és a dir, amb un color més negre que blanc per fer més visuals els compassos. Amb *strokeWeight()* [11] indiquem que volem que la línia sigui més gruixuda.

Abans d'haver cridat les funcions anteriors es fa la crida a *createCanvas()* [11] des de *setup()*.

La variable resultant *cnv* simbolitza el llenç en el que dibuixarem i en el qual l'aplicació funciona. Després cridem a *background()* [11] per escollir el color que volem de fons.

```
cnv = createCanvas(window.innerWidth, window.innerHeight);
background("#249da3");
```

Un cop dibuixada la quadrícula procedim a dibuixar els botons interactius. Això ho fem mitjançant la variable *pattern*, que és una matriu d'uns i zeros que representa el patró amb el qual els instruments sonaran o no. Si és un 1 es reproduirà i si és un 0 no ho farà.

```
function drawPattern() {
    // aquest bucle omple d'un color en concret cada casella del pattern que hem
seleccionat
  for(let track=0; track<nTracks; track++){
    for(let step=0; step<nSteps; step++){
      if(pattern[track][step] === 1){
        fill(255 - track*30);
        rect(step*cellWidth+cellWidth/6 + origen, track*cellWidth + cellWidth/6 +
origen, cellWidth*2/3, cellWidth*2/3, 2);
      }
    }
  }
}
```

Com podem observar amb la funció *rect(coordenada x, coordenada y, amplitud, altitud, radi per arrodonir les cantonades)* [11] dibuixem els quadrats dins de la quadrícula si és que

estan activats, aquests canvien de color de blanc a negre progressivament en funció de si pertanyen a pistes diferents, això ho fem per diferenciar-los més fàcilment.

Per fer que els botons siguin activables i desactivables és imprescindible que la següent línia de codi sigui executada en la funció *setup()*:

```
cnv.mousePressed(canvasPressed);
```

El que fem amb això indicar que s'ha de captar cada cop que l'usuari pitja sobre alguna part del llenç de l'aplicació i cridar la funció *canvasPressed()* automàticament un cop ho hagi fet. Un cop cridat *canvasPressed()* s'analitza on l'usuari clica exactament per després activar o desactivar cada botó.

```
function canvasPressed(){
  Tone.start();

  if (mouseY > origen &&  mouseX > origen && mouseX < (origen+cellWidth*32) &&
mouseY < (origen+cellWidth*8)) {
    let row = floor((mouseY-origen)/cellWidth);
    let index = floor((mouseX-origen)/cellWidth);

    pattern[row][index] = +!pattern[row][index];
  }
  return false;
}
```

El que es fa primerament és afegir una condició per analitzar si l'usuari està clicant dins de la quadrícula o no. Si això es compleix trobarem el número de fila i columna amb la relació *floor(mouseX o mouseY - origen / cellWidth)*. *MouseX* i *MouseY* són variables reservades que ens indiquen amb coordenades on es troba el ratolí exactament, i el que fa la funció *floor()* és retornar el número decimal que li passem per paràmetre arrodonit a l'enter proper més baix [11], és a dir, si li passem un 3.78 ens retornarà un 3. Aquesta relació ens indica de forma acurada la posició en la matriu. Quan l'usuari clica en una casella de la quadrícula, la matriu en aquell punt canvia a estat 0 o 1, per tant, s'activa o es desactiva l'instrument en aquell instant. Un cop actualitzada la matriu *pattern* automàticament veiem com també el dibuix s'actualitza degut a que, com s'ha comentat anteriorment, *drawPattern()* es crida des de *draw()*, funció que s'executa de forma constant.

Podem observar el resultat de la interfície en la següent figura:

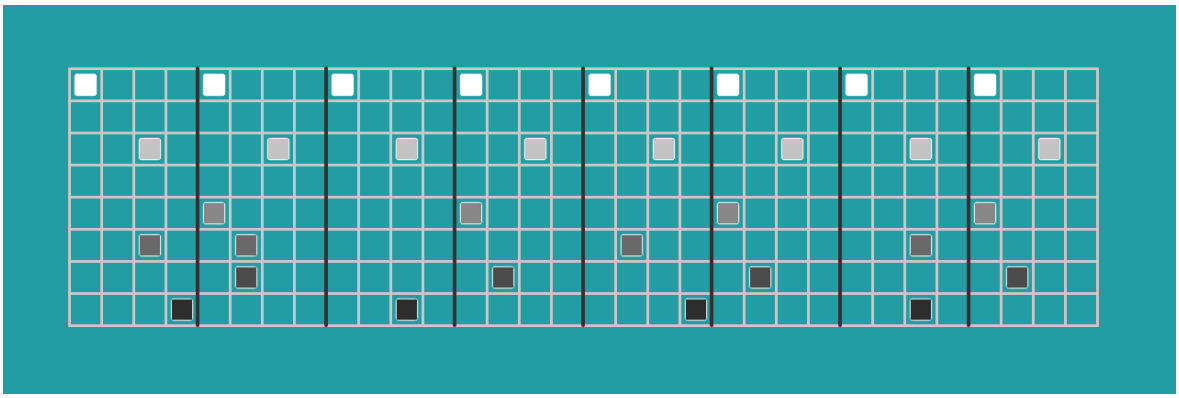

Figura 11 *Interfície gràfica del seqüenciador*

3.2.2 Reproducció i repetició

El primer pas en aquesta fase és l'obtenció dels *samples*, que seran els que l'aplicació carregarà per defecte un cop iniciada. Alguns s'obtenen de forma gratuïta i altres s'obtenen d'una font pròpia, gravant i exportant sons de sintetitzadors en una DAW. Els que s'obtenen de forma gratuïta són percussions provinents de la famosa caixa de ritmes Roland TR-909, s'obtenen a partir de: https://bvker.com/free-909-samples/ i escollim aquests sons degut al seu simbolisme, especialment en els estils House i Techno en les dècades dels 80 i 90.

En una primera fase del projecte, aquesta part es desenvolupa amb la llibreria *p5.sound*, concretament utilitzant l'objecte *p5.Phrase()*, que és un patró que representa esdeveniments musicals al llarg del temps, i *p5.Part()* en el qual afegim els objectes p5.*Phrase* per aconseguir el *playback* però la llibreria presenta un error greu. El so comença a distorsionar-se a mida que deixem sonar un loop durant una estona i sembla que té alguna cosa a veure amb la càrrega de CPU pel tipus de so, el problema es presenta en gairebé tots els navegadors. Es tracta d'un error de la llibreria p5.sound reportat a GitHub (https://github.com/processing/p5.js-sound/issues/494) i que encara no s'ha solucionat.

Un cop localitzat aquest error es decideix traslladar tota la part d'àudio a la llibreria Tone.js, que també disposa de mètodes per seqüenciar múltiples sons.

Utilitzem l'objecte *Tone.Buffer,* utilitzat internament per totes les classes que fan sol·licituds de fitxers d'àudio, com ara *Tone.Player* [10].

La càrrega dels fitxers d'àudio es fa de la següent manera:

```
for (i = 0; i < routes.length; i++) {
  buffer[i] = new Tone.Buffer(routes[i]);
}
```

Routes és un objecte de tipus *array* el qual conté les rutes cap als fitxers d'àudio col·locats prèviament en la carpeta samples de l'aplicació. Com podem observar, carreguem les rutes dins l'objecte *Buffer* per després concatenar aquests objectes en una altra *array* anomenada *buffer*. Cada fitxer es carrega en una posició de l'*array*, per tant, els índex d'aquesta fan d'identificador del propi sample, fet important que més tard s'utilitzarà.

Un cop carregat el buffer d'àudio, declarem l'objecte *Tone.Players,* que és una combinació de múltiples objectes de tipus *Tone.Player* [10], que és un reproductor de fitxers d'àudio amb funcions d'iniciar, repetir i parar.

```
kit = new Tone.Players(
  {"kick" : buffer[indexKick],
   "snare" : buffer[indexSnare],
   "hh" : buffer[indexHh],
   "ohh" : buffer[indexOhh],
   "clap" : buffer[indexClap],
   "perc" : buffer[indexPerc],
   "stab" : buffer[indexStab],
   "vocal" : buffer[indexVocal]
}).toDestination();

Tone.Transport.scheduleRepeat(onStep, "16n");
```

Dins l'objecte *Players* assignem el nom del tipus d'instrument que estem carregant, per exemple "kick" o "snare" (bombo i caixa) al fitxer d'àudio corresponent, això ho fem mitjançant els índex del buffer. Com s'ha comentat anteriorment, cada índex representa un identificador, per tant les variables *indexKick, indexSnare, indexHh*... són els identificadors del so que hi haurà per defecte, variables globals declarades al principi del codi, com són 5 sons diferents carregats per cada tipus d'instrument els índex per defecte comencen per 0 i després múltiples de 5. L'usuari podrà canviar-los més tard mitjançant un selector.

En aquest moment hi ha 8 possibles instruments que poden sonar alhora en l'aplicació: un bombo, una caixa, un hi-hat, un open hi-hat, un clap (picament de mans), altres percussions, un stab (cop d'acord en mode *staccato*) i una vocal.

La crida a *toDestination()* es fa per connectar la sortida al node de destinació del context actual, que en aquest cas seran els mateixos altaveus de l'ordinador o la sortida que tinguem connectada com per exemple uns auriculars [10].

Cal indicar també quin és el BPM (Beats Per Minute), en altres paraules, el tempo de la seqüència. Aquesta part es configura amb una instrucció senzilla que ofereix la llibreria de Tone.js:

```
Tone.Transport.bpm.value = 110;
```

Fem el tempo sigui configurable afegint una entrada per via text de l'usuari amb la funció *createInput().* Cada cop que l'usuari afegeix text en aquesta entrada i cliqui el botó *bpmButton*, la següent funció serà cridada i el tempo canviarà:

```
function changeBpm(){
  let bpm = input.value();
  Tone.Transport.bpm.value = bpm;
}
```

Per fer possible el *looping* s'utilitza *Tone.Transport.scheduleRepeat*, mètode que ens serveix per programar esdeveniments musicals al llarg del temps [10]. L'esdeveniment s'executa quan li diem i durant un temps determinat. El mètode rep dos paràmetres, un és el *callback* i l'altre és el temps cada quan volem que es repeteixi l'esdeveniment, amb la notació "16n" volem indicar que l'esdeveniment es repeteix cada 32 temps. Un *callback* és una funció passada com a argument en una altra funció que és invocada després de realitzar certes accions. En aquest cas el *callback* és la funció *onStep()* que és l'encarregada de reproduir els samples només si estan activats en aquell instant.

```
function onStep(time){
  for(let track=0; track<nTracks; track++){
    if(pattern[track][currentStep] === 1){
      kit.player(drumNames[track]).start(time);
    }
  }
  beats++;
  currentStep = (beats) % nSteps;
}
```

La funció *onStep()* és cridada en cada pas. Les variables *beats* i *currentStep* són comptadors de passos. *Beats* compta de 0 fins que es pausa el transport, és a dir, si deixem sonar tota la seqüencia dues vegades ens comptarà fins a 64, *currentStep* representa en quin temps exactament de la seqüència ens trobem de 0 a 32, tenint en compte que va calculant el residu entre *beats* i *nSteps* que té un valor de 32. Aquestes variables ens

serveixen per poder fer un control en el mateix bucle de quins samples estan activats en cada pas, en cas que ho estiguin es crida *kit.player()* passant-li per paràmetre el nom del tipus d'instrument, contingut en l'array *drumNames[]*, obtenint l'objecte *Player* que volem que sigui reproduït després amb el mètode *start()*. És important passar-li la variable *time* al mètode *start()* indicant-li així quan el sample ha de sonar si volem que la seqüencia vagi sonant a un tempo exacte, el *callback* no està perfectament quantificat però cada cop que s'executa anticipa el següent ritme ben quantificat.

Per acabar amb el seqüenciador cal afegir un botó per iniciar i pausar la seqüència. La llibreria p5.dom ens facilita això. El codi consta de dues parts:

```
playButton = createButton('Play');
playButton.position(150, 540);
playButton.mouseClicked(togglePlay);
```

Creem un element de tipus botó i afegim la posició de la pantalla on volem que estigui. La funció *mouseClicked()* és cridada només quan aquest element és clicat, cridant automàticament a *togglePlay()* que és el *callback*.

```
function togglePlay() {
  if(Tone.Transport.state === "started"){
    Tone.Transport.stop();
    playButton.html('Play');
  } else {
    Tone.Transport.start();
    playButton.html('Stop');
  }
}
```

La seqüencia programada anteriorment s'inicia i es pausa amb *Tone.Transport.start()* o *stop()* respectivament, comprovant abans si aquesta està iniciada amb l'atribut *state*. El text del botó canviarà de *play* a *stop* i viceversa per indicar a l'usuari de forma visual que s'ha iniciat la seqüència o s'ha parat amb la funció *html().* Es pot fer el mateix però mitjançant el teclat de l'ordinador, cosa que pot proporcionar més comoditat a l'usuari. Així doncs, es declara un codi similar però dins de la funció *keyPressed()*.

```
keyPressed(){
  if (keyCode === 32) {
    if (Tone.Transport.state === "started") {
      Tone.Transport.stop();
    } else {
      beats = 0;
      currentStep = 0;
      Tone.Transport.start();
    }
  }
}
```

El funcionament ara canvia lleugerament, utilitzem el codi 32 que representa la tecla d'espai de l'ordinador per iniciar o para la seqüencia. En aquest cas fem que cada cop que s'inicïi per via teclat, la seqüencia començarà des de l'inici indicant que *currentStep* i *beats* tenen un valor de 0, creant un comportament més semblant al d'una DAW.

Des de fa uns anys, molts navegadors han afegit una norma que consisteix en no permetre que un àudio sigui reproduït fins que l'usuari tingui una interacció amb una pàgina web. En conseqüència s'ha d'afegir el següent codi per iniciar el context d'àudio abans que ninguna acció sigui realitzada un cop l'usuari ha clicat en la pàgina.

Error reportat a: https://github.com/Tonejs/Tone.js/issues/341

```
document.documentElement.addEventListener(
  "mousedown", function(){
    mouse_IsDown = true;
    if (Tone.context.state !== 'running') {
      Tone.start();
}})
```

Abans d'avançar amb les següents funcionalitats de l'aplicació cal afegir un detall més a la interfície. Els botons dels samplers moderns solen il·luminar-se a mida que el ritme es va reproduint per ajudar a l'usuari a seguir el que està sonant. Per poder imitar aquesta funcionalitat, desenvolupem una funció senzilla que segueixi els passos mitjançant la variable *beats*.

```
function drawHighlight() {
  highlight = (beats-1) % nSteps;
  fill(200, 60);
  noStroke();
  if (highlight !== -1) {
    rect(highlight*cellWidth + origen, origen, cellWidth, cellWidth*8);
  }
}
```

La variable *highlight* indica el pas actual. La funció *fill()* rep dos paràmetres, el primer és el valor en escala de gris i el segon és la opacitat del color [11], així doncs omplim un rectangle d'un color blanc que ocupa una columna i totes les pistes en el pas actual. Amb *noStroke()* deshabilitem la vora del rectangle [11] ja que no ens interessa perquè ja està dibuixada.

3.2.3 Selectors

Abans de continuar amb el desenvolupament de les altres funcionalitats és imprescindible crear els selectors els quals permetran a l'usuari canviar de sample en cada pista. En els buffers carregats anteriorment tenim en total un número de 40 samples, que seran els que l'aplicació presentarà per defecte. A cada pista li pertanyen 5 d'aquests, i ja estan ordenats.

Declarem una classe *Dropdown* amb el següent constructor:

```
class Dropdown {
  constructor(sampleType, x ,y, track, id){
    this.sampleType = sampleType;
    this.x = x;
    this.y = y;
    this.track = track;
    this.sel = createSelect();
    this.sel.ide = id;
    this.sel.changed(() => this.changeSample());
```

```
}
```

Cada pista té el seu menú així doncs, necessitem l'entitat *sampleType* per indicar el tipus d'instrument. X i y representen coordenades on volem col·locar el selector. *Track* representa el número de pista, que és de tipus enter i serà un número de 1 a 8 perquè tenim 8 pistes. *Sel* és l'objecte del menú desplegable en si, que és un element DOM, aquest es crea amb la funció *createSelect()*. *Sel.ide* és l'identificador del menú, que és de tipus *String*. L'última entitat fa que es cridi el mètode *changeSample()* quan el valor de l'element canvia, és important que declarem això últim en el constructor de la classe perquè sinó, el mètode *changeSample()* rep els camps com a no definits. És important també afegir el *callback* amb la sintaxi '*() =>*', sinó ho fem així, quan el menú és creat *sel.changed()* rep un valor indefinit i rebem un error perquè per defecte, aquesta només rebrà un valor quan l'usuari canviï el valor del selector.

```
create() {
  this.sel.position(this.x, this.y);
  this.sel.style("width", "90px");

  for (i = 1; i < 6; i++) {
    this.sel.option(`${this.sampleType}${i}`);
  }
  this.sel.id(this.sel.ide);
}
```

Després de crear l'objecte Dropdown s'ha de cridar el mètode *create()* el qual ens posiciona el menú i crea les opcions que contindrà amb un bucle. El resultat per exemple en la primera pista serà: *kick1, kick2, kick3*... fins a 5. Després s'assigna l'identificador de l'element amb la funció *id().*

```
changeSample() {
    let bufferIndex;
    let index = this.sel.value().match(/(\d+)/);
    else if (this.sampleType === 'kick') {
      bufferIndex = Number((index[0]-1));
      changePlayers(bufferIndex, 1);
    }
    else if (this.sampleType === 'snare') {
      bufferIndex = Number((index[0]-1)+5);
      changePlayers(bufferIndex, 2);
    }
    else if (this.sampleType === 'hh') {
      bufferIndex = Number((index[0]-1)+10);
      changePlayers(bufferIndex, 3);
    }
    else if (this.sampleType === 'ohh') {
      bufferIndex = Number((index[0]-1)+15);
      changePlayers(bufferIndex, 4);
    }
    else if (this.sampleType === 'clap') {
      bufferIndex = Number((index[0]-1)+20);
      changePlayers(bufferIndex, 5);
    }
```

```
    else if (this.sampleType === 'perc') {
      bufferIndex = Number((index[0]-1)+25);
      changePlayers(bufferIndex, 6);
    }
    else if (this.sampleType === 'stab') {
      bufferIndex = Number((index[0]-1)+30);
      changePlayers(bufferIndex, 7);
    }
    else if (this.sampleType === 'vocal') {
      bufferIndex = Number((index[0]-1)+35);
      changePlayers(bufferIndex, 8);
    }
  }
```

Cada cop que el valor canvia en un menú es crida al mètode *changeSample()*, aquest mètode troba l'índex amb una expressió regular. *This.sel.value()* retorna el contingut de l'opció que s'ha triat, per exemple: *"kick2"*, amb *match(/(\d+)/)* indiquem que volem trobar un dígit contingut en un *String*, així doncs s'extreu el número 2 de *"kick2"*. *BufferIndex* és la variable que conté l'identificador del sample que ara volem que soni. Com els samples s'han carregat de manera ordenada al buffer d'àudio, si canviem per exemple, el selector de l'última pista (de tipus *vocal*) a "vocal5" en comptes de "vocal1", l'identificador serà el valor 5 - 1 + 35 = 39, que serà l'últim dels 40 samples carregats al buffer.

És important extreure l'índex amb la funció *Number()*, que força la conversió a un tipus número.

La implementació de la funció *changePlayers()* és senzilla. Amb els dos paràmetres que rep (*bufferIndex* i el número de pista) crea un nou objecte *Players()* però indicant el nou índex del buffer de la pista que l'usuari hagi seleccionat.

```
function changePlayers(ind, track){
  if (track === 1) {
    indexKick = ind;
  }
  else if (track === 2) {
    indexSnare = ind;
  }
  else if (track === 3) {
    indexHh = ind;
  }
  else if (track === 4) {
    indexOhh = ind;
  }
  else if (track === 5) {
    indexClap = ind;
  }
  else if (track === 6) {
    indexPerc = ind;
  }
  else if (track === 7) {
    indexStab = ind;
  }
```

```
  else if (track === 8) {
    indexVocal = ind;
  }

  kit = new Tone.Players(
    {"kick" : buffer[indexKick],
      "snare" : buffer[indexSnare],
      "hh" : buffer[indexHh],
      "ohh" : buffer[indexOhh],
      "clap" : buffer[indexClap],
      "perc" : buffer[indexPerc],
      "stab" : buffer[indexStab],
      "vocal" : buffer[indexVocal]
  }).toDestination();
}
```

Els menús es col·loquen al costat esquerra de cada pista per indicar de manera clara a l'usuari quin és l'instrument que s'està reproduint en cada pista.

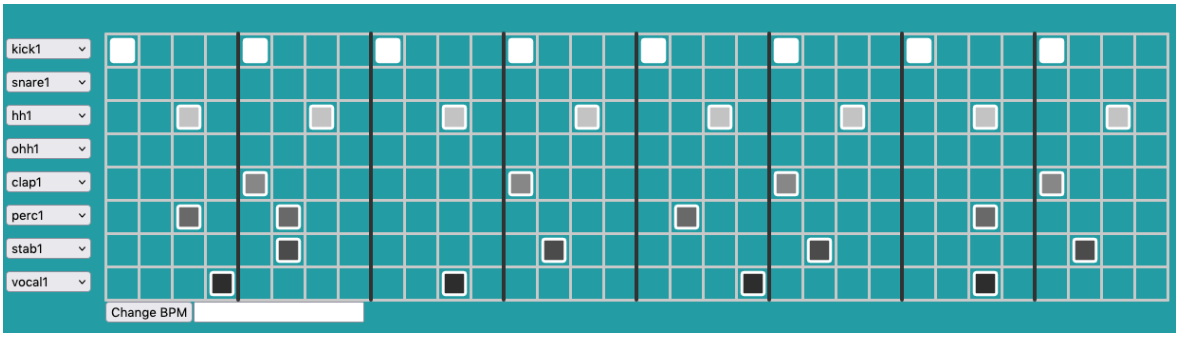

Figura 12 *Interfície del seqüenciador amb els menús desplegables i el botó i entrada per canviar el BPM*

3.2.4 Botons de silenciat

El sampler és un instrument molt utilitzat per fer música en directe, això comporta que de vegades, sigui còmode pels artistes que les utilitzen silenciar una o més pistes, amb la finalitat de fer transicions. Així doncs es desenvolupen botons per silenciar. Declarem la següent classe amb els mètodes essencials per assolir l'objectiu:

```
class MuteButton {

  constructor(x, y, player) {
    this.x = x;
    this.y = y;
    this.player = player;
    this.button = createButton('Mute');
    this.muted = new Switch(false);
    this.button.mousePressed(() => this.mute());
  }
```

```
  create() {
    this.button.position(this.x, this.y);
    this.getState(this.muted);
  }

  mute() {
    if (this.muted.getState() === false) {
      kit.player(this.player).mute = true;
      this.muted.switchState();
      this.button.html('Muted');
    }
    else {
      kit.player(this.player).mute = false;
      this.muted.switchState();
      this.button.html('Mute');
    }
  }

  keepMute() {
    kit.player(this.player).mute = true;
  }

  getState() {
    return this.muted.getState();
  }
}
```

L'entitat *muted* és un objecte de classe *Switch,* que bàsicament té una funció d'interruptor d'encès i apagat. El mètode mute() s'encarrega de silenciar o activar el so de la pista, i és cridat cada cop que el botó és clicat per l'usuari. Els mètodes *keepMute()* i *getState()* ens serveixen per mantenir les pistes silenciades quan canviem un sample en una pista. Com que cada cop que canviem un sample creem un altre objecte *Players()* (a la funció *changePlayers()* mostrada a la pàgina 23), les pistes que hi ha silenciades no mantenen el seu estat. Per solucionar això implementem la funció *unmuteTrackWhenPlayerChanged()*, que serà cridada al final de la funció *changePlayers()*. Aquesta mantindrà l'estat de silenciat de les pistes que ho estan, i activarà el so de la pista a la qual l'usuari li està seleccionant un altre sample amb el menú desplegable corresponent.

```
function unmuteTrackWhenPlayerChanged(track) {
  if (eval(`track${track}MuteButton`).getState() === true) {
    eval(`track${track}MuteButton`).mute();
  }
  for (i = 1; i < 9; i++) {
    if (eval(`track${i}MuteButton`).getState() === true && i != track) {
      eval(`track${i}MuteButton`).keepMute();
    }
  }
}
```

Eval() és una funció que avalua un codi JavaScript representat com una cadena de caràcters. Per tant, executarà com una variable, instrucció o objecte el que li passem com a paràmetre, avaluant així la variable assignada al botó de silenciat.

Per crear els botons s'inicialitza un objecte tipus *MuteButton* indicant la posició del botó i l'instrument (paràmetre *player* del constructor), i després és crida el mètode *create()*.

3.3 Càrrega de fitxers d'àudio per part de l'usuari

Abans de res, cal crear una entrada de fitxers a l'aplicació perquè l'usuari pugui carregar els fitxers d'àudio emmagatzemats en qualsevol directori del seu ordinador. Aquest tipus d'element, d'igual manera que s'ha de fer amb tots els elements DOM, s'ha de declarar a la funció *setup()* ja que ens interessa que només siguin creats una sola vegada.

```
inputSoundFile = createFileInput(handleAudioFile);
inputSoundFile.position(150, 600);
```

Un cop l'usuari hagi carregat un fitxer d'àudio, automàticament la funció *handleAudioFile()* serà cridada com a *callback*.

```
function handleAudioFile(file) {
  if (file.type === 'audio') {
    let blobObj = dataUrlToBlob(file.data, "audio" );
    let fileurl = URL.createObjectURL(blobObj);
    append(buffer, new Tone.Buffer(fileurl));
    uploadedFiles++;
    }
}
```

El primer que fem un cop rebem el fitxer és comprovar que sigui de tipus àudio amb l'atribut *file.type*. Si imprimim l'objecte *file* a la consola del navegador veiem que té la següent forma:

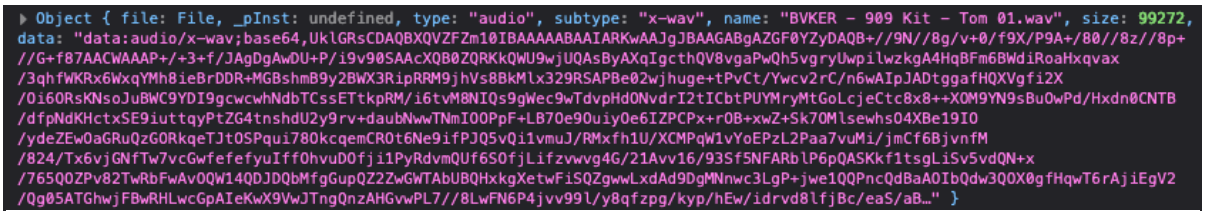

Figura 13 *Representació de l'objecte File en la consola del navegador FireFox*

Es pot observar que té diversos camps que indiquen informació sobre el fitxer, com per exemple el nom, el subtipus, la mida del fitxer en bytes... però el que ens interessa és el camp *data*, que conté les mostres d'àudio codificades en base64. Base64 és un sistema de numeració posicional que fa servir 64 com a base. És la major potència de dos que pot ser representada usant únicament els caràcters imprimibles d'ASCII.

El següent pas és enviar aquest camp a la funció *dataUrlToBlob()* perquè ens la retorni com a un objecte de tipus *Blob*. Un *Blob* és un objecte d'alt nivell que representa un objecte tipus fitxer de dades immutables. Els Blobs representen dades que no es troben necessàriament en un format natiu de JavaScript.

```
function dataUrlToBlob(dataURI, dataTYPE) {
  var binary = atob(dataURI.split(',')[1]), array = [];
  for(var i = 0; i < binary.length; i++) array.push(binary.charCodeAt(i));
  return new Blob([new Uint8Array(array)], {type: dataTYPE});
}
```

La funció *atob()* descodifica tota la cadena a dades binàries. El mètode *charCodeAt()* va retornant nombres enters entre 0 i 65535 que representen la unitat de codi UTF-16, amb els quals formem una array. Aquesta es passa per paràmetre a la funció *Uint8Array()*, la sortida és la representació d'una array d'enters sense signe de 8 bits. Amb aquesta última formem l'objecte Blob indicant el tipus de fitxer que està representant. Si imprimim l'objecte a la consola observem el següent:

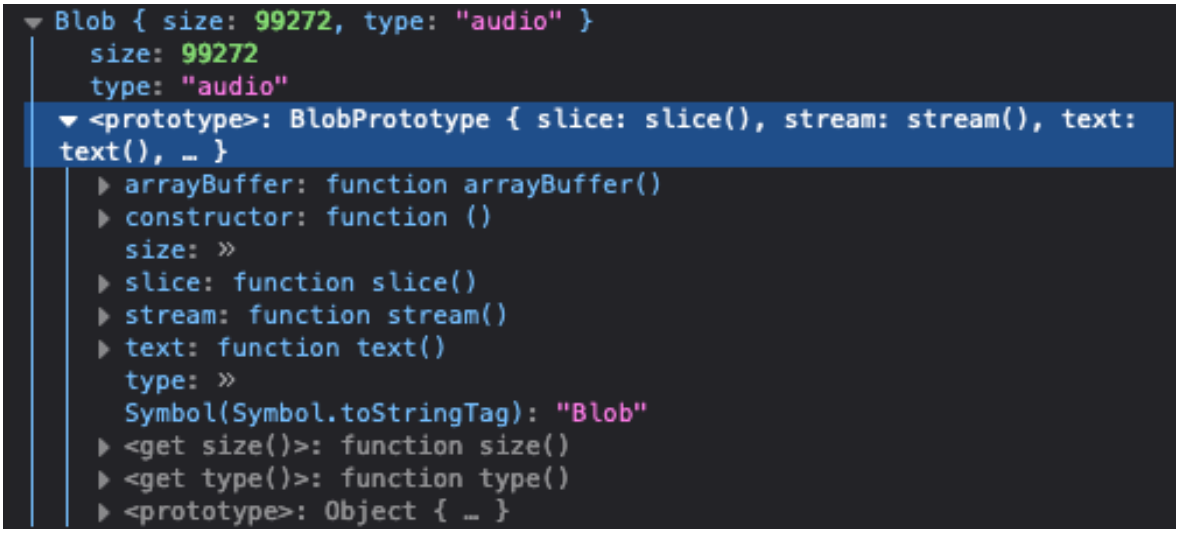

Figura 14 *Representació de l'objecte Blob en la consola del navegador FireFox*

Passem aquest últim objecte com a paràmetre de la funció *createObjectURL()*. Aquesta retorna una URL que representa a l'objecte que li passem per paràmetre, la vida de la URL està lligada al document de la finestra que ha sigut creada, és a dir, quan l'usuari refresca la pàgina, aquesta deixa d'existir [13]. La URL que retorna té la següent forma, tenint en compte que s'està treballant de forma local i amb el port 5502:
blob:http://127.0.0.1:5502/dc44f22c-1d48-45f8-8cc6-07bc68321837

Tal i com s'ha fet anteriorment, el següent pas per fer possible la reproducció del fitxer és afegir aquesta URL al buffer d'àudio amb la funció *append()*:

```
append(buffer, new Tone.Buffer(fileurl));
```

Un cop es té el fitxer d'àudio en el buffer, l'usuari ha de carregar-lo a una de les pistes per fer possible la seva seqüenciació amb els altres instruments. Això es fa mitjançant un altre un altre menú desplegable el qual anomenem *trackSelect*, es col·loca sota la quadrícula per diferenciar-lo dels altres. També afegim un text per indicar-li a l'usuari que el sample s'ha carregat bé en l'aplicació i per pregunta-li a quina pista o pistes vol assignar el fitxer. El menú consta de 8 opcions (track1, track2, track3... track8). Un cop se selecciona una és crida automàticament la funció *addFileNameToATrack()*, encarregada d'afegir l'opció de reproducció en la pista seleccionada per l'usuari.

```
function addFileNameToATrack(value) {
  let index = value.match(/(\d+)/);
```

```
  trackSelected = true;
  trackSelectedName = value;
  if (Number(index[0]) === 1) {
    track1Select.setOption(`usersample${uploadedFiles}`);
  }
  else if (Number(index[0]) === 2) {
    track2Select.setOption(`usersample${uploadedFiles}`);
  }
  else if (Number(index[0]) === 3) {
    track3Select.setOption(`usersample${uploadedFiles}`);
  }
  else if (Number(index[0]) === 4) {
    track4Select.setOption(`usersample${uploadedFiles}`);
  }
  else if (Number(index[0]) === 5) {
    track5Select.setOption(`usersample${uploadedFiles}`);
  }
  else if (Number(index[0]) === 6) {
    track6Select.setOption(`usersample${uploadedFiles}`);
  }
  else if (Number(index[0]) === 7) {
    track7Select.setOption(`usersample${uploadedFiles}`);
  }
  else if (Number(index[0]) === 8) {
    track8Select.setOption(`usersample${uploadedFiles}`);
  }
}
```

La implementació és semblant a la dels menús generats anteriorment, cercant l'índex amb una expressió regular però en aquest cas l'utilitzem per saber quina pista selecciona l'usuari. Com es pot observar, tots els samples carregats per l'usuari tenen un nom tipus "*usersample"* amb un número al costat, que és un comptador dels fitxers que es van carregant a l'aplicació. La variable *trackselected* és un booleà que serveix per activar el text en la funció *draw()* per indicar-li a l'usuari que el seu sample s'ha carregat a la pista on ha indicat.

Per poder afegir l'opció de reproducció d'un sample de l'usuari als menús de cada pista, cal afegir el mètode *setOption()* dins la classe Dropdown, implementada anteriorment:

```
setOption(sampleName) {
  this.sel.option(sampleName);
}
```

No és la única modificació que li fem a la classe Dropdown. Ampliem el mètode *changeSample()* mostrat en la pàgina 25.

```
let isUserSample = this.sel.value().match('usersample');
let associated = false;
if (isUserSample != null) {
  for (i = 0; i < 9 && associated === false; i++) {
     if ((index in Object(userSampleTrack[i])) === true) {
```

```
       //agafem aquell buffer ja associat per reproduir-lo
      bufferIndex = userSampleTrack[i][index];

      //asssociem el buffer associat al sample al track actual per
      userSampleTrack[this.track-1][index] = bufferIndex;

      //parem la cerca
      associated = true;
    }
}

/* Sinó hi ha cap associat significa que haurem de reproduir-ne l'últim del
buffer que es l´últim que hem afegit
 i també associar-lo a la array per facilitar la feina després */
    if (associated === false) {
      bufferIndex = buffer.length-1;
      userSampleTrack[this.track-1][index] = bufferIndex;
    }
      changePlayers(bufferIndex, this.track);
}
```

Per dur a terme aquesta part es declara una matriu tridimensional anomenada *userSampleTrack* amb una estructura que relaciona cada pista, amb cada índex del sample que ha carregat l'usuari i l'índex del buffer on està el fitxer:

[{índex del sample de l'usuari : índex del buffer},{},{},{},{},{},{},{}];

Cada posició representa una pista, i dins de cada pista hi ha un diccionari que relaciona els índex del sample i del buffer. Això ho fem per emmagatzemar els valors i fer una cerca de si el mateix sample s'ha carregat ja en més d'una pista. Sinó ho ha fet, emmagatzemem els valors en la matriu, per tant la variable booleana *associated* seguirà sent falsa perquè no hi ha cap pista associada al sample que ha carregat l'usuari, així doncs, lògicament es reproduirà l'últim sample que hi ha carregat al buffer. Sinó és així, significa que ja aquell sample ja s'ha carregat en una pista, per tant farem servir l'índex del buffer associat anteriorment a aquell sample.

Tot el desenvolupament previ es fa amb l'objectiu de que l'usuari pugui pujar nombrosos fitxers en qualsevol i també a múltiples pistes.

3.4 Gravació per micròfon

Es crea un altre botó amb la instrucció de que cada cop que sigui clicat es cridi a la funció *recordFromMic()*. L'objecte *Tone.UserMedia()* es fa servir per obrir un micròfon extern o una entrada d'àudio. *Tone.Recorder()* és un objecte que fa servir l'API de MediaRecorder per gravar àudio i vídeo provinent tan com del navegador com d'un dispositiu de l'usuari [9]. Els dos objectes en aquest cas també són nodes i els connectem via *connect()*. El micròfon és el node d'entrada i *Recorder* el de sortida.

La implementació es fa de manera que l'usuari no pugui gravar samples més llargs de 2 segons. Això es fa mitjançant *setTimeout()* i *sync*. Tot el que s'executa dins de *setTimeout()* no ho fa fins que han passat 2000 mil·lisegons (2 segons). Una vegada ha passat el temps

establert, el procediment és el mateix que quan l'usuari carrega un fitxer directament: es crea una URL per després ser afegida al buffer, etc.

```
function recordFromMic() {
    if (!initialized) {
    mic = new Tone.UserMedia();
    micRecorder = new Tone.Recorder();
    mic.connect(micRecorder);
    mic.open();
    initialized = true;
  }
  micRecorder.start();

  setTimeout(async () => {
    let data = await micRecorder.stop();
    let blobUrl = URL.createObjectURL(data);
    userPlayer = new Tone.Player(blobUrl, () => {
      userPlayer.start();
      append(buffer, new Tone.Buffer(blobUrl));
      uploadedFiles++;
      fileUploaded = new Switch(true);
      fileUploaded = fileUploaded.getState();
    }).toDestination();
  }, 2000);
};
```

La instrucció *await micRecorder.stop()* va de la mà amb *async*. La paraula clau *async* s'afegeix davant de les funcions per convertir-les en asíncrones i perquè retornin una promesa enlloc d'un valor directament. Aquesta retorna un objecte de tipus *Promise()*, que representa una operació asíncrona. Els mètodes *start()* i *stop()* de *Tone. Recorder* retornen objectes *Promise()*, *stop()* atura la gravació i retorna una promesa amb el contingut que s'ha gravat que serà de tipus ***mime***, que és el format en que l'àudio és codificat.

Una *Promise()* pot estar en un dels següents estats [15]:

· *pending (pendent)*: estat inicial, ni completa ni rebutjada, que és l'estat que retorna el mètode *stop()* inicialment.

· *fullfilled(completa)*: significa que l'operació s'ha completat amb èxit.

· *rejected (rebutjada)*: significa que l'operació ha fallat.

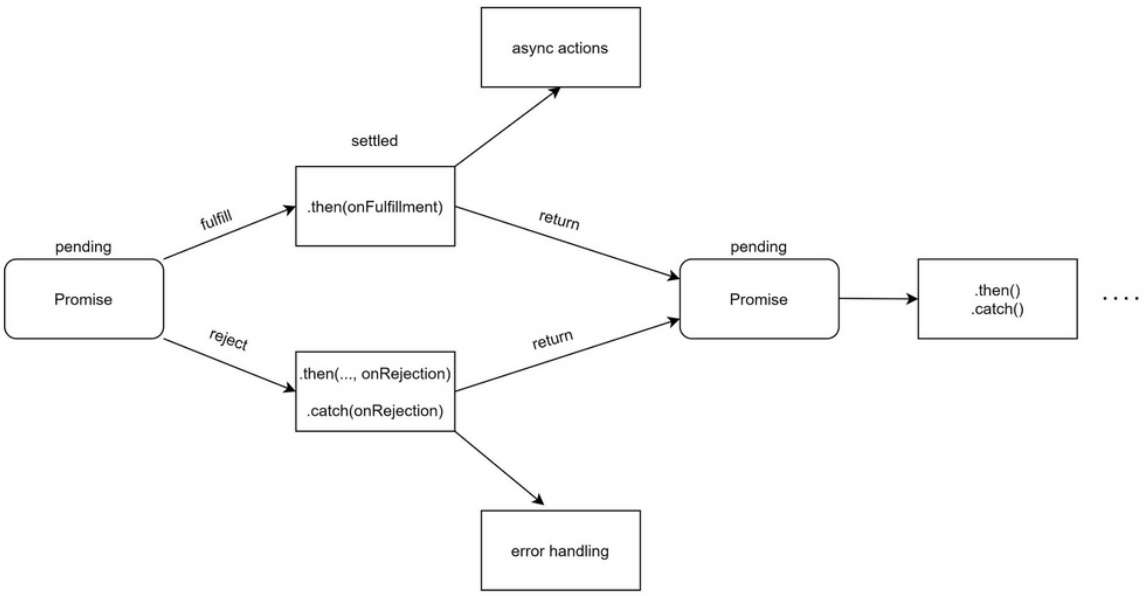

Figura 15 *Diagrama del funcionament de l'objecte Promise() i els seus estats*

3.5 Efectes

L'entorn Tone.js disposa d'una amplia varietat d'efectes, des de filtres, fins a distorsions i reverberacions. L'objectiu es poder aplicar un determinat número d'efectes a cada una de les pistes per separat. L'usuari ha de poder processar cada instrument al seu gust.

En concret, aplicarem els efectes que més s'utilitzen en les samplers en format hardware: delay, reverberació i filtres de freqüència. A part d'això també ens interessa que hi hagi un control de volum a cada pista.

Els efectes van connectats en cadena. L'ordre de la cadena d'efectes és el següent:

1. ***Tone.PitchShift()***: Tots els samplers, tal i com s'ha explicat en l'apartat 2.2.2, disposen d'un control de la nota de cada sample. Així doncs, s'inclou el control de pitch com a primer dels efectes. Cal remarcar que l'aplicació no disposarà d'un anàlisi freqüencial com a les samplers reals, que situen cada sample en una freqüència central per després assignar-les a notes d'un teclat MIDI. La freqüència central en aquest cas serà la del mateix sample que entri, i es farà el canvi de pitch mitjançant semitons amb l'atribut *pitch* que proporciona la pròpia llibreria. Un semitò és el menor dels intervals que es poden produir entre dues notes consecutives. Per exemple, el que hi ha entre un Do i un Do sostingut, és un semitò, i el que hi ha entre un Do i un Re, es un to.
   L'efecte fa un canvi de to gairebé en temps real al senyal que li entrem, accelerant o alentint el el temps de *delay* d'un *DelayNode* mitjançant una ona de dent de serra [10].

2. ***Tone.Volume():*** Es tracta d'un node de volum. El control de volum no és un efecte però és una funció essencial que ha de tenir l'aplicació.

3. ***Tone.Filter():*** El filtre de freqüència és un efecte molt utilitzat en la producció musical. En concret utilitzarem dos tipus de filtre, un passa-alts (HPF) i un altre

passa-baixos (LPF). Aquest permet canviar el factor de roll-off entre -24dB, -48dB i -12dB [10], sent aquest últim el que hi ha per defecte i el que fem servir. El factor de roll-off és el pendent d'una funció de transferència de freqüència.

4. ***Tone.FeedbackDelay():*** Es tracta d'un node tipus *DelayNode*. Funciona retornant el mateix senyal retardat provinent de la sortida [10]. Fem servir dos paràmetres que seran controlables per l'usuari. El *delay time,* que representa la diferència de temps en que es tarden en reproduir les mostres retardades, i el *feedback* que també fa referència al temps, però en aquest cas el temps total fins que deixem de sentir la repetició.

5. ***Tone.Reverb():*** És una simple convolució creada amb soroll decaient [10]. Aquest efecte dona profunditat al so i és generalment molt utilitzat per mesclar cançons. Utilitza *ConvolverNode.*

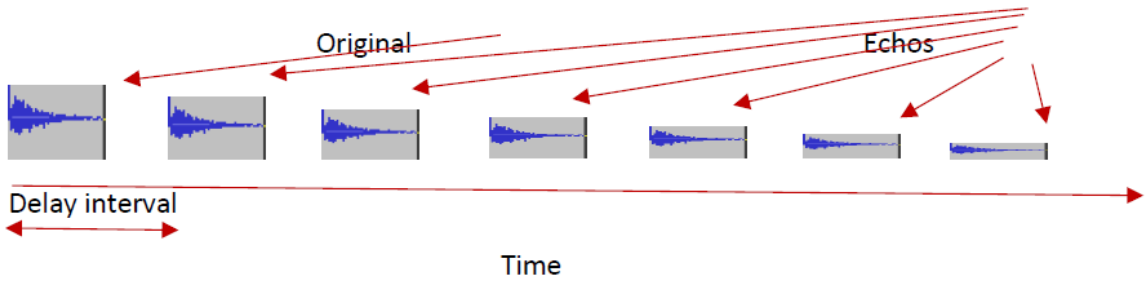

Figura 16 *Esquema explicatiu de l'efecte Delay*

Tractem els efectes de cada pista com a objectes diferent amb la classe *Effects* amb el següent constructor, que és bàsicament la declaració de cada un d'ells:

```
class Effects {
  constructor(player) {
    this.player = player;
    this.delay = new Tone.FeedbackDelay("3n", 0.2);
    this.reverb = new Tone.Reverb(0.9);
    this.vol = new Tone.Volume(0);
    this.hpfilter = new Tone.Filter(20, "highpass");
    this.lpfilter = new Tone.Filter(20000, "lowpass");
    this.pitchShifter = new Tone.PitchShift(0);
  }
```

Tone.Feedbackdelay() rep dos paràmetres, un és el temps de delay (*delay time*) i l'altre el temps de feedback. Amb la notació "3n" indiquem que volem que el sample es repeteixi cada terç de temps. Aquests paràmetres serveixen simplement per inicialitzar el constructor, però realment aquests seran totalment configurables per l'usuari.

Tone.Reverb() rep un paràmetre que és el temps de *decay*, que és la quantitat de temps total en que el so reverberarà. Es decideix que aquest paràmetre si que serà fixe, però l'usuari podrà controlar el *wet*, paràmetre amb el qual manipulem la quantitat de l'efecte que hi ha a la sortida.

Tone.Volume() s'inicialitza a 0, això significa que el volum per defecte serà el mateix que presenta el sample.

El filtre passa-alts que correspon al camp *hpfilter* s'inicialitza indicant el tipus de filtre (*high-pass*) i la freqüència a la qual comença a filtrar, és a dir, la **freqüència de tall**, per defecte la deixem a 20Hz que és la freqüència a partir de la qual l'ésser humà es capaç de sentir. Amb el filtre passa-baixos (*lpfilter*) passa el mateix, indiquem el tipus de filtre, en aquest cas "*low-pass*" i inicialitzem la freqüència de tall a 20.000Hz que és la freqüència màxima a la qual podem sentir.

Un cop tenim el constructor de la classe cal programar els efectes de manera que vagin en cadena, en l'ordre que s'ha indicat anteriorment.

```
kit.player(this.player).chain(this.pitchShifter, this.vol, this.hpfilter,
this.lpfilter, this.delay, this.reverb, Tone.Destination);
```

Si connectem només dos nodes això és fa mitjançant el mètode *connect()*, però com treballem amb múltiples nodes fem servir el mètode *chain()*.

Figura 17 *Esquema de connexió d'efectes amb entrada i sortida amb Tone.js*

3.5.1 Controls

L'usuari ha de poder controlar certs paràmetres dels efectes en temps real, això ho fem a través de la llibreria ***KnobMakerC.js***.

```
MakeKnobC(knobColor, diameter, locx, locy, lowNum, hiNum, defaultNum, numPlaces,
labelText, textColor, textPt)
```

Per inicialitzar l'objecte hem d'indicar-li: el color, el diàmetre, les coordenades on volem que sigui dibuixat, uns valors mínims i màxims, el valor que presenta per defecte, el valor que que es mostra a sota del knob, el text que utilitzem per indicar l'efecte, el color del text, i el tamany del text [16].

Els knobs també els tractarem d'una manera semblant als efectes, és a dir, com a un sol objecte per a cada pista. El constructor de la classe Knobs és el següent:

```
class Knobs {
  constructor(color) {
    this.delayKnob = new MakeKnobC(color, 50, 900, 530, 0, 10, 0, 0,"Delay",
[0,200,200], 12);
    this.delayTimeKnob = new MakeKnobC(color, 50, 1000, 530, 0, 1, 0, 0,"Delay
Time", [0,200,200], 12);
    this.reverbKnob = new MakeKnobC(color, 50, 1100, 530, 0, 10, 0, 0,"Reverb",
[0,200,200], 12);
```

```
    this.volumeKnob = new MakeKnobC(color, 50, 800, 530, -12, 0, 0, 0,"Volume",
[0,200,200], 12);
    this.hpfilterKnob = new MakeKnobC(color, 50, 1200, 530, 20, 10000, 20,
0,"HPF", [0,200,200], 12);
    this.lpfilterKnob = new MakeKnobC(color, 50, 1300, 530, 20000, 20, 20000,
0,"LPF", [0,200,200], 12);
  }
}
```

Creem els knobs tenint en compte els valors que els efectes poden i han de rebre, per exemple, el de volum pot ajustar-se de 0dB a -12dB, sent 0 el valor per defecte. En altres paraules, l'usuari no pot pujar el volum de l'instrument però sí baixar-lo. Un altre exemple és el del filtre passa-alts, amb un valor mínim de 20Hz i un màxim de 10.000Hz, és a dir, es podrà filtrar com a màxim fins a 10.000Hz.

És imprescindible declarar també els següents mètodes, que tenen la funció d'analitzar les accions del ratolí en cada un dels knobs.

```
update() {
  this.volumeKnob.update();
  this.delayKnob.update();
  this.delayTimeKnob.update();
  this.reverbKnob.update();
  this.hpfilterKnob.update();
  this.lpfilterKnob.update();
}

active() {
  this.volumeKnob.active();
  this.delayKnob.active();
  this.delayTimeKnob.active();
  this.reverbKnob.active();
  this.hpfilterKnob.active();
  this.lpfilterKnob.active();
}

inactive() {
  this.volumeKnob.inactive();
  this.delayKnob.inactive();
  this.delayTimeKnob.inactive();
  this.reverbKnob.inactive();
  this.hpfilterKnob.inactive();
  this.lpfilterKnob.inactive();
}
```

La crida al mètode *active()* s'ha de realitzar des de la funció *mousePressed()* per activar el knob. El mètode *inactive()* es crida des de *mouseReleased()*, funció que és cridada després de que l'usuari hagi clicat, per desactivar el knob. El mètode *update()* ha de ser obligatòriament cridat des de *draw()*, ja que és l'encarregat d'actualitzar els valors en funció de com l'usuari va girant el control.

```
knobsTrack1 = new Knobs("#249da3");
```

El comportament és: quan l'usuari clica a sobre, ha de mantenir el ratolí clicat mentre el mou cap a dalt o cap a baix, això fa rodar el knob, el girament es pot observar amb el punt blanc que s'observa en la següent figura:

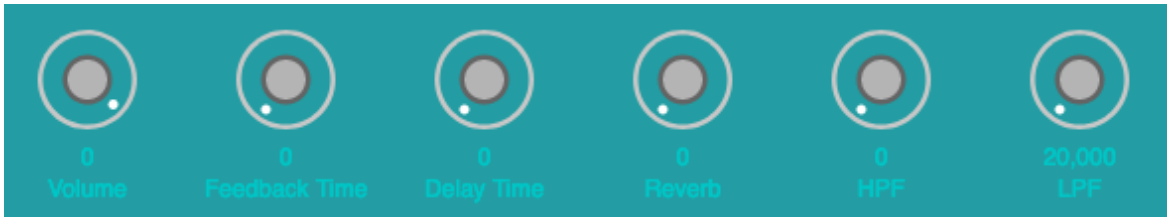

Figura 18 *Interfície dels knobs assignats a la primera pista*

A mida que gira, va retornant diferents valors i passant-los automàticament als efectes que hem creat abans, així doncs, processant el so en temps real.

Abans que això succeeixi és necessari crear una connexió entre aquests i els efectes.

Això ho fem mitjançant primer de tot, un mètode a la classe *Effects* que rebi i assigni els valors que els knobs estan retornant:

```
setValues(value1, value2, value3, value4, value5, value6) {
  this.delay.wet.value = value1*0.1;
  this.delay.delayTime.value = value2;
  this.vol.volume.value = value3;
  this.hpfilter.frequency.value = value4;
  this.lpfilter.frequency.value = value5;
  this.reverb.wet.value= value6*0.1;
}
```

Com el valors poden estar en constant canvi, és imprescindible que cridem aquesta funció des de *draw()*. Alguns els multipliquem per 0.1 perquè els que rebem són valors més fàcils de reconèixer per l'usuari però que no poden assignar-se a un paràmetre perquè necessitem que siguin entre 0 i 1.

A la classe Knobs afegirem el següent mètode que ens permetrà obtenir el valor que està retornant cada knob:

```
getValue(effectKnob) {
  return eval(`this.${effectKnob}.knobValue`);
}
```

Li passarem per paràmetre el nom del knob assignat a l'efecte, declarat en el constructor de la classe Knobs, com un String i ens ho executa com si fos una instrucció més. Per entendre millor aquest últim pas observem el següent codi situat en la funció *draw()*:

```
effectsTrack1.setValues(knobsTrack1.getValue("delayKnob"),
knobsTrack1.getValue("delayTimeKnob"), knobsTrack1.getValue("volumeKnob"),
knobsTrack1.getValue("hpfilterKnob"), knobsTrack1.getValue("lpfilterKnob"),
knobsTrack1.getValue("reverbKnob"));
```

Aquesta última instrucció representa la interfície de connexió que hi ha entre els knobs i els efectes de la primera pista. Cal repetir-la per cada pista.

Com es pot observar, no hem afegit un knob per controlar el pitch, això es perquè això es fa des d'una altra funcionalitat, que es veu en el pròxim apartat.

Per una qüestió d'espai a la pantalla, no ens podem permetre mostrar tots els knobs de totes les pistes alhora, per tant se cerca una alternativa. Primer declarem la classe *KnobsDrawing*.

```
class KnobsDrawing {

  setKnobsDrawing(track) {
    this.knobs = `knobsTrack${track}.update();`;
  }

  getKnobsDrawing() {
    return eval(this.knobs);
  }
}
```

Aquesta conté dos mètodes: *setKnobsDrawing()* assigna al camp *knobs* la funció update() d'una pista que li passem per paràmetre, que a part d'indicar-nos el valor, també és la que dibuixa els knobs; *getKnobsDrawing()* retorna la mateixa funció, per tant, quan sigui cridada ens els dibuixarà.

El pròxim pas és crear uns botons al costat de cada pista que continguin el text "FX" que és l'abreviació de la paraula *effects* en anglès. L'objectiu és que cada cop que l'usuari cliqui un d'aquests botons, se li dibuixin els knobs associats a la pista.

```
knobsDrawing = new KnobsDrawing();
track1FxButton = createButton('FX').mousePressed(()  =>
knobsDrawing.setKnobsDrawing(1));
track1FxButton.position(1390, 205);
```

Cada cop que el botó es clica, assignem els knobs que volem que es dibuixin passant-li en aquest cas un 1 al mètode *setKnobsDrawing(1)* assenyalant que són els de la primera pista. Això es fa a cada pista.

Per últim afegim el següent codi a *draw()* que ens va retornant de forma constant *update()* dels knobs associats a cada pista que selecciona l'usuari.

```
knobsDrawing.getKnobsDrawing();
```

3.6 Control de pitch i interacció amb el teclat

L'objectiu en aquesta funcionalitat és que l'usuari pugui controlar el pitch de cada un dels samples de les pistes, i deixar gravades melodies en la seqüència. També ho és que pugui activar passos per via teclat.

Per poder assolir aquest objectiu és imprescindible treballar amb els codis associats a cada tecla de l'ordinador, els podem veure en la següent taula:

Tecla	Codi
Espai	32
y	89
u	85
i	73
o	79
p	80
`	192
+	171
t	84
r	82
e	69
w	87
q	81
Tab	9
a	65
s	83
d	68
f	70
g	71
h	72
j	74
k	75

Taula 1 *Codis associats a les tecles de l'ordinador. [17]*

Dins la funció *keyPressed()* cal desenvolupar un codi que realitzi determinades accions quan cliquem una d'aquestes tecles.

```
if (keyCode === 65) {
  setSampleAtCurrentStep(1);
}
if (keyCode === 83) {
  setSampleAtCurrentStep(2);
}
if (keyCode === 68) {
  setSampleAtCurrentStep(3);
}
if (keyCode === 70) {
  setSampleAtCurrentStep(4);
}
if (keyCode === 71) {
  setSampleAtCurrentStep(5);
```

```
}
if (keyCode === 72) {
  setSampleAtCurrentStep(6);
}
if (keyCode === 74) {
  setSampleAtCurrentStep(7);
}
if (keyCode === 75) {
  setSampleAtCurrentStep(8);
}
```

Quan cliquem per exemple la tecla 'a', la funció *setSampleAtCurrentStep()* és cridada.

```
function setSampleAtCurrentStep(track) {
  if (!toneStarted()) {
    beats = 0;
    currentStep = 0;
    Tone.Transport.start();
  }
    pattern[track-1][currentStep] = 1;
}
```

La implementació és senzilla. Cada tecla està assignada a una pista. Cada cop que es clica una tecla, primer de tot la seqüència s'inicia, si és que no ho està. Després s'activa aquell pas en aquell instant de temps, permetent a l'usuari gravar ritmes utilitzant el teclat com si fossin els botons d'una caixa de ritmes en temps real, en comptes d'haver-los de pre-programar amb el ratolí. La gravació de la seqüència és polifònica, per això s'ha implementat amb *if()*, és a dir, no cal gravar cada pista per separat.

Un cop aquesta part funciona, procedim a desenvolupar la part de gravació de melodies. El primer es crear nous botons al costat de cada pista amb el text "PITCH". Un cop un d'aquests siguin clicats, la pista associada serà assignada a noves tecles i aquestes assignades a valors de pitch diferents del mateix sample, podent així controlar les notes del mateix instrument.

```
//PITCH CONTROL
if (keyCode === 9) {
  setPitch(-6);
}
if (keyCode === 81) {
  setPitch(-5);
}
if (keyCode === 87) {
  setPitch(-4);
}
if (keyCode === 69) {
  setPitch(-3);
}
if (keyCode === 82) {
  setPitch(-2);
}
if (keyCode === 84) {
  setPitch(-1);
}
if (keyCode === 89) {
```

```
  setPitch(0);
}
if (keyCode === 85) {
  setPitch(1);
}
if (keyCode === 73) {
  setPitch(2);
}
if (keyCode === 79) {
  setPitch(3);
}
if (keyCode === 80) {
  setPitch(4);
}
if (keyCode === 192) {
  setPitch(5);
}
if (keyCode === 171) {
  setPitch(6);
}
```

Prendrem la freqüència central, que es el sample original, com a la tecla 'y' amb el codi associat 89. Les tecles del costat dret, representen el mateix sample més agut, i les del costat esquerra el mateix sample més greu. A cada tecla que ens movem a la dreta o a la esquerra tenim el sample amb un semitò més o menys, intentant imitar l'estructura d'un teclat MIDI o un piano.

```
function setPitch(pitchValue) {
  eval(`effectsTrack${pitchTrack}`).changePitch(pitchValue);
  setSampleAtCurrentStep(pitchTrack);
  for(i = currentStep ; i < pitchValues[pitchTrack].length; i++) {
    pitchValues[pitchTrack-1][i] = pitchValue;
  }
}
```

Els valors associats a cada pas de cada pista s'emmagatzemen en la matriu *pitchValues()*. Una matriu inicialitzada amb zeros inicialment i amb les mateixes dimensions que la matriu *pattern* però que pot tenir valors entre -6 i 6. Ara no només podem gravar els passos, sinó també un pitch associat a cada pas. El valor de pitch associat que s'enregistra en aquell pas, s'associa també a la resta dels passos (d'esquerra a dreta) en la pista seleccionada.

Si volem reproduir la seqüència amb els valors de pitch associats cal modificar la funció *onStep().*

```
function onStep(time){
  for(let track=0; track<nTracks; track++){
    if(pattern[track][currentStep] === 1){
        eval(`effectsTrack${pitchTrack}`).changePitch(pitchValues[track]
[currentStep]);
        kit.player(drumNames[track]).start(time);
    }
  }
```

```
  beats++;
  currentStep = (beats) % nSteps;
}
```

També cal afegir un mètode a la classe *Effects* que ens canviï el pitch de cada sample:

```
changePitch(semitone) {
  this.pitchShifter.pitch = semitone;
}
```

La implementació en aquest cas és també senzilla, només cal assignar-li un valor de semitò al atribut *pitch* del mateix efecte. Observem que si li passem el número 0, sonarà el sample original sense cap modificació.

3.7 Gravació i descàrrega de seqüències

Per desenvolupar aquesta part s'ha de crear un nou botó amb el mateix procediment amb el que creem tots els botons. Aquest conté el text "Record Sequence" i quan és clicat es crida automàticament a la funció *recordSequence().* Abans de començar la gravació és primordial afegir el node ***Recorder*** al final de la cadena. Això ens interessa perquè l'aplicació ha de permetre també enregistrar el processament de les pistes en temps real. Ho fem mitjançant la funció *makechain()*, que crea les cadenes per totes les pistes i el mètode *chain()* [10] de la classe *Effects*.

```
function makeChain(record) {
  effectsTrack1.chain(record);
  effectsTrack2.chain(record);
  effectsTrack3.chain(record);
  effectsTrack4.chain(record);
  effectsTrack5.chain(record);
  effectsTrack6.chain(record);
  effectsTrack7.chain(record);
  effectsTrack8.chain(record);
}
```

Aquest mètode és cridat també després de crear tots els efectes un cop s'inicia l'aplicació, i també quan canviem un sample amb el menú desplegable, però en aquest cas passant com a paràmetre un valor ***null*** per no afegir el node *Recorder* a la cadena.

```
chain(record) {
  if (record === null) {
    kit.player(this.player).chain(this.pitchShifter, this.vol, this.hpfilter,
this.lpfilter, this.delay, this.reverb, Tone.Destination);
  }

  else if (record != null){
    kit.player(this.player).chain(this.pitchShifter, this.vol, this.hpfilter,
this.lpfilter,  this.delay, this.reverb, Tone.Destination, recorder);
  }
}
```

No obstant, sí que ens interessa afegir aquest node quan volem gravar les seqüències, per això passem per paràmetre a la funció *makeChain()* un valor diferent de *null* que pot ser per exemple un 1. Com podem observar en la funció, el funcionament de l'enregistrament és igual que el de la funció *recordFromMic()* mostrada a la pàgina 34, però en aquest cas limitem la gravació a 30 segons i en comptes de fer servir la funció *connect()* per connectar el micròfon d'entrada amb la gravadora, encadenem directament les fonts d'àudio i els efectes amb la gravadora. Cada cop que es clica aquest botó s'inicia la seqüència des del principi assignant a les variables *currentStep* i *beats* un valor de 0. Un cop passen els 30 segons, es crea una URL que representa el fitxer d'àudio i es força la descàrrega amb l'element *anchor* i l'atribut *download* i assignant la URL generada a l'atribut *href*.

Finalment es decideix que el format de la descàrrega és *webm*, un format multimèdia desenvolupat per Google i orientat per utilitzar-se amb HTML5. Els navegadors de forma nativa no codifiquen l'àudio en WAV o MP3. L'API de *MediaRecorder*, que és el que utilitza *Tone.Recorder()*, accepta certs formats com poden ser-ho *ogg* o *webm* amb els còdec *opus*.

```
function recordSequence() {
  recorder = new Tone.Recorder();
  makeChain(1);
  currentStep = 0;
  beats = 0;
  recorder.start();

  setTimeout(async () => {
  const recording = await recorder.stop();
  const url = URL.createObjectURL(recording);
  const anchor = document.createElement("a");
  anchor.download = "userRecording.webm";
  anchor.href = url;
  anchor.click();
}, 30000);

  Tone.Transport.start();
}
```

Un cop desenvolupada la funcionalitat de gravació la cadena té la següent forma:

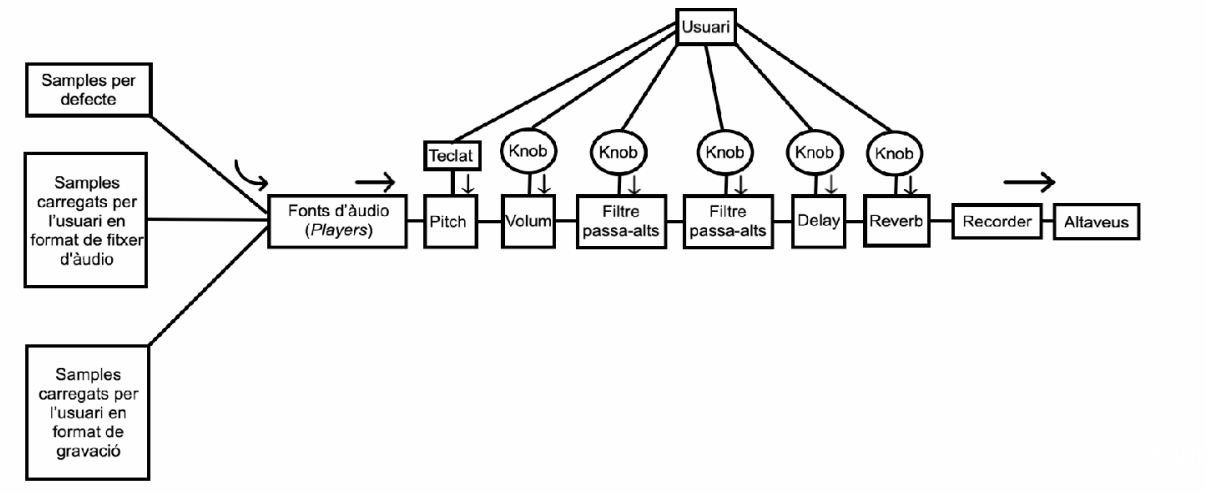

Figura 19 *Diagrama de fluxe del sampler web*

3.8 Guardar i carregar un pattern en format JSON

3.8.1 Guardar

JSON (JavaScript Object Notation) és un format basat en text estàndard per representar dades estructurades amb la sintaxi d'objecte de JavaScript. És típicament utilitzat per transmetre dades en aplicacions web [18]. L'estructura és una cadena amb un format que recorda als objectes literals de JavaScript.

Amb aquesta última funcionalitat volem aconseguir que l'usuari pugui descarregar la matriu *pattern* i els índex dels samples dels menús desplegables, per després permetre que torni a carregar aquest fitxer i reconfigurar de nou l'estructura que tenia prèviament.

El primer pas és crear un botó de descàrrega, amb el mateix comportament que els botons anteriors.

```
downloadJSONButton = createButton('Save Pattern').mousePressed(downloadJSON);
downloadJSONButton.position(900, 600);
```

Quan aquest és clicat, la funció *downloadJSON()* és cridada:

```
function downloadJSON() {
  let JSONPattern = [];
  //per fer el JSON més llegible quan el descarreguem
  for (i = 0; i < pattern.length; i++) {
    JSONPattern[i] = pattern[i].toString().replace(/,/g, "");
  }

  createStringDict({
    pattern : JSONPattern,
    track1SampleIndex : track1Select.getIndexItemSelected(),
    track2SampleIndex : track2Select.getIndexItemSelected(),
    track3SampleIndex : track3Select.getIndexItemSelected(),
    track4SampleIndex : track4Select.getIndexItemSelected(),
    track5SampleIndex : track5Select.getIndexItemSelected(),
    track6SampleIndex : track6Select.getIndexItemSelected(),
    track7SampleIndex : track7Select.getIndexItemSelected(),
    track8SampleIndex : track8Select.getIndexItemSelected()
  }).saveJSON('userpattern');
}
```

En el primer bucle fem una modificació de la matriu *pattern* per fer-la més còmodament llegible, en concret la convertim a String i li eliminem les comes amb la funció *replace()*.

Després obtenim els índex de les opcions seleccionades als menús desplegables de cada pista amb el mètode *getIndexItemSelected()* implementat a la classe *Dropdown*.

```
getIndexItemSelected() {
  return document.getElementById(this.sel.ide).selectedIndex;
}
```

Amb la funció *createStringDict()* creem l'estructura d'un objecte literal de JavasScript [11], associant cada paraula al seu respectiu valor. Quan es clica el botó obtenim un fitxer amb nom "userpattern.json" amb el següent contingut:

```
{
  "pattern": [
    "1000100010001000100011001000100 0",
    "0000000001000000000100000000000 0",
    "0010001000100010001000100010001 0",
    "0000000000000000000000000000000 0",
    "0000100000001000000010000000100 0",
    "0010010000000000010000000010000 0",
    "0000010000000100000001000000010 0",
    "0001000000100000000100000010000 0"
  ],
  "track1SampleIndex": 4,
  "track2SampleIndex": 4,
  "track3SampleIndex": 2,
  "track4SampleIndex": 0,
  "track5SampleIndex": 4,
  "track6SampleIndex": 0,
  "track7SampleIndex": 2,
  "track8SampleIndex": 3
}
```

3.8.2 Carregar

Ja tenim l'estructura del fitxer JSON definida, ara hem de permetre que l'usuari pugui carregar aquest mateix fitxer i que sigui interpretat correctament. Creem una altra entrada de fitxer:

```
inputPatternFile = createFileInput(handlePatternFile);
inputPatternFile.position(900, 680);
```

Quan el fitxer es carrega correctament *handlePatternFile()* és cridada passant-li el fitxer rebut com a paràmetre:

```
function handlePatternFile(file) {
  let base64Str = file.data.split(",")[1];
  let jsonStr = atob(base64Str);
  let JSONObject = JSON.parse(jsonStr);
  let userpattern = JSONObject.pattern;

  loadPattern(userpattern);

  track1Select.selectOption(JSONObject.track1SampleIndex);
  track2Select.selectOption(JSONObject.track2SampleIndex);
  track3Select.selectOption(JSONObject.track3SampleIndex);
  track4Select.selectOption(JSONObject.track4SampleIndex);
  track5Select.selectOption(JSONObject.track5SampleIndex);
  track6Select.selectOption(JSONObject.track6SampleIndex);
  track7Select.selectOption(JSONObject.track7SampleIndex);
  track8Select.selectOption(JSONObject.track8SampleIndex);
}
```

El procediment inicial és similar al de quan carreguem un fitxer d'àudio en l'aplicació. Obtenim el camp data codificat en base64 i el descodifiquem amb la funció *atob().* A la variable *jsonStr* obtenim les dades del fitxer correctament però en un string. Cal passar aquest string com a paràmetre al mètode *parse()* per poder interpretar les dades rebudes com a un objecte de nou i poder tractar amb cada camp per separat.

Posteriorment es crida la funció *loadPattern()* que carrega automàticament el patró en el seqüenciador, i el mètode *selectOption()* de la classe *Dropdown* que selecciona el mateix índex del menú sense necessitat de que l'usuari ho faci manualment:

```
selectOption(index) {
  document.getElementById(this.sel.ide).selectedIndex = String(index);
  this.changeSample(this.sampleType, this.sel.value(), this.track);
}
```

4 Anàlisi de l'aplicació i resultats

4.1.1 Canvis en el desenvolupament

Un cop es finalitza el desenvolupament s'aprecia un problema greu, l'aplicació presenta una alta càrrega de CPU, concretament amb valors al voltant del 120% o més. L'anàlisi es du a terme través de l'aplicació de "Monitor d'activitat" que proporciona el mateix sistema operatiu Mac, la qual mostra el temps i la càrrega de CPU de cada procés que s'està executant en el nostre ordinador. Aquest fet afecta greument al so de l'aplicació, el qual es pausa i distorsiona de manera constant i repetida sense que l'usuari realitzi cap acció. Es realitzen proves en els navegadors: Firefox, Chrome, Safari i Opera. El problema és persistent en cada un d'ells, per tant, es procedeix a investigar què pot estar causant aquest error.

Alguns dels efectes aplicats en cada pista poden estar consumint molts recursos, en concret, el que es basen en *ConvolverNode*. En altres paraules, efectes que utilitzen convolucions per processar el so. Una **convolució** és una operació matemàtica que transforma dues funcions en una tercera que representa la magnitud de superposició de les dues funcions originals [20]. Quan aquests efectes es posen en us, es calculen múltiples FFT per a cada bloc [19]. Un dels efectes basats en *ConvolverNode* és la reverberació.

Cal remarcar que abans d'aplicar l'efecte *Tone.Reverb()* explicat en l'apartat d'efectes de la pàgina 35, s'ha utilitzat un altre efecte similar, *Tone.Freeverb()*. Aquest últim, prioritzant el bon funcionament del programa, és desactivat de cada pista, reduint de gran manera el consum de recursos. Concretament, passem de consumir un 120% de CPU a un 60%. Posteriorment apliquem l'efecte *Tone.Reverb()* que és el que finalment s'aplica. Aquest consumeix aproximadament la meitat que l'altre, augmentant al 90% la càrrega.

No és l'únic canvi que es duu a terme per reduir els costos de processament. S'aplica la funció *frameRate()* que pertany a la llibreria P5.js, aquesta serveix per modificar els fotogrames per segon que representen la freqüència o taxa a la qual s'estan mostrant les imatges. El *frame rate* que proporciona P5.js per defecte és de 60 frames per segon. Tenint en compte que l'objectiu d'aquest projecte no és mostrar una animació molt precisa, reduïm aquest número a només 10 frames per segon, més que suficient per veure com la barra clara que mostra cada pas avança amb el tempo. Després de realitzar aquest últim canvi, podem observar que el consum de CPU de l'aplicació es redueix al 70%, arribant a valors màxims del 80%. Cal destacar que aquest segueix sent un valor alt, però ara el so ja no es veu afectat.

4.1.2 Resultats

El funcionament de l'aplicació és correcte i assoleix els objectius marcats al principi. No només això, sinó que s'han pogut ampliar les funcionalitats per exemple, amb la part de control de pitch, amb les botons de silenciat o amb la descàrrega dels patrons en format JSON.

És cert que un dels objectius principals era que la descàrrega de les gravacions fos en format WAV, no obstant, no s'ha considerat un fet molt rellevant en el treball degut a que l'aplicació no està enfocada a productors musicals professionals que després vulguin adquirir les seqüències gravades per comercialitzar-les, sinó més cap a persones que volen iniciar-se en aquest món i practicar amb una eina intuïtiva.

5 Pressupost

Tenint en compte que s'han utilitzat únicament llibreries *open-source* per a la realització del projecte, i que s'ha utilitzat un ordinador personal, el cost que es té en compte és el de les hores invertides en el projecte. Considerant que aquest projecte l'ha realitzat un becari en pràctiques que cobra aproximadament 9€ l'hora, i que s'han dedicat unes 400 hores per fer el projecte.

Costos	Preu	Quantitat	Cost total
Becari	9€/hora	400 hores	3600€

Taula 2 *Pressupost*

6 Conclusions i treballs futurs

Com ja s'ha comentat en l'apartat de resultats, els objectius s'han assolit amb èxit inclòs podent-ne ampliar les funcionalitats. Qualsevol persona que prova l'aplicació diu que és fluida, ràpida i amb una interfície gràfica còmode i intuïtiva. Permet processar el so en temps real, permetent així a l'usuari conèixer les funcions de cada efecte i jugar amb cada un d'ells analitzant el so resultant, podent ajudar així a qualsevol persona que vulgui iniciar-se en el món de la producció musical.

L'aplicació proporciona uns samples per defecte que estan enfocats especialment als estils House, Techno i derivats. Ara bé, la funcionalitat de carregar fitxers d'àudio, també podent carregar-los mitjançant gravacions des del micròfon, amplia de gran manera les possibilitats en relació a això. Tenint en compte que el tempo es pot canviar fàcilment i que l'usuari pot carregar samples propis de qualsevol estil, podem dir que es poden seqüenciar ritmes de gairebé tots els estils musicals existents.

En opinió personal i com a amant de la música electrònica i de tota la tecnologia que l'envolta, he gaudit i après molt fent aquest projecte, i recomanaria a qualsevol persona també interessada en aquest àmbit, a seguir desenvolupant eines i instruments tant en format software com en format web.

De cara a treballs futurs, és un projecte amb moltes possibilitats d'ampliació, considerant la limitació de no sobrecarregar la CPU:

- Desenvolupar un sintetitzador web capaç de sincronitzar-se i seqüenciar-se en tempo amb el sampler.
- Afegir un control MIDI dels samples i que l'aplicació permeti connectar un teclat extern a part d'utilitzar el teclat de l'ordinador.
- Que l'aplicació permeti gravar seqüències de temps il·limitat i que aquestes siguin adquiribles en diversos formats d'àudio escollits per l'usuari.
- Allotjar l'aplicació en un servidor i modificar el desenvolupament de manera que hi hagi una pàgina de *login* i la possibilitat de registrar-s'hi, permetent també que cada usuari pugui emmagatzemar els seus samples en el seu perfil de l'aplicació.
- Millorar l'aplicació amb l'objectiu de fer-la més eficient, per tal de que consumeixi molts menys recursos.

7 Referències

1. Sampler (musical instrument). A: Viquipèdia [en línia]. Wikimedia Foundation, 2022. [Consulta: 27 desembre 2022]. Disponible a: <https://en.wikipedia.org/wiki/Sampler_(musical_instrument)>
2. Music production history – The 5 most important eras. A: Chris Hörmann [en línia]. 29 gener 2016. [Consulta: 27 de desembre 2022]. Disponible a: <https://chrishoermann.at/en/music-production-history/>
3. Fonógrafo. A: Viquipèdia [en línia]. Wikimedia Foundation, 2022. [Consulta: 27 desembre 2022]. Disponible a: <https://es.wikipedia.org/wiki/Fon%C3%B3grafo>
4. Mellotron. A: Viquipèdia [en línia]. Wikimedia Foundation, 2022. [Consulta: 27 desembre 2022]. Disponible a: <https://es.wikipedia.org/wiki/Mellotron>
5. *Nuevas tendencias en la creación musical propiciadas por las nuevas tecnologías* A: Francisco José Cuadrado Mendez [en línia]. Fundación Telefónica 2018 [Consulta: 28 desembre 2022]. Disponible a <https://telos.fundaciontelefonica.com/archivo/numero106/nuevas-tendencias-en-la-creacion-musical-propiciadas-por-las-nuevas-tecnologias/ >
6. Vienna Symphonic Library. A: Viquipèdia [en línia]. Wikimedia Foundation, 2022. [Consulta: 29 desembre 2022]. Disponible a: <https://en.wikipedia.org/wiki/Vienna_Symphonic_Library>
7. Riffusion. A: Viquipèdia [en línia]. Wikimedia Foundation, 2022. [Consulta: 29 desembre 2022]. Disponible a: <https://en.wikipedia.org/wiki/Riffusion>
8. Control Knob. A: Viquipèdia [en línia]. Wikimedia Foundation, 2022. [Consulta: 30 desembre 2022]. Disponible a: <https://en.wikipedia.org/wiki/Control_knob>
9. Web Audio API. A: Mozilla Foundation [Consulta: 30 desembre 2022]. Disponible a: <https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API>
10. Tone.js Docs. Disponible a <https://tonejs.github.io/docs/14.7.77/index.html>
11. P5.js Reference. Disponible a: <https://p5js.org/es/reference/>
12. Los peligros de la basura electrónica. A: National Geographic España, 2022 [Consulta 27 de desembre]. Disponible a: <https://www.nationalgeographic.com.es/mundo-ng/peligros-basura-electronica_13239>
13. URL.createObjectURL(). A: Mozilla Foundation. Disponible a: <https://developer.mozilla.org/es/docs/Web/API/URL/createObjectURL>
14. Blob(). A: Mozilla Foundation. Disponible a: <https://developer.mozilla.org/es/docs/Web/API/Blob>
15. Promise() A: Mozilla Foundation. Disponible a: <https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Promise>
16. Rotating Knobs with p5 dran buttons (Llibreria KnobMakerC). A: Miles DeCoster, 2020. Disponible a: <https://codeforartists.com/RotatingKnobMaker/>
17. keyCodes JavaScript. Disponible a: https://www.toptal.com/developers/keycode
18. JSON. A: Mozilla Foundation. Disponible a: <https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/JSON>
19. Web Audio API performance and debugging notes. A: Paul Adenot [Consulta: 5 gener 2023]. Disponible a: <https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/JSON>
20. Convolution. A: Viquipèdia. Wikimedia Foundation, 2022 [Consulta 8 gener 2023]. Disponible a: <https://en.wikipedia.org/wiki/Convolution>
21. Sample. A: Viquipèdia, Wikimedia Foundation, 2022 [Consulta 20 desembre 2022] Disponible a: <https://es.wikipedia.org/wiki/Sample>