Desarrollo de un Plugin de ritmos euclidianos con JUCE

Javier Cano Salcedo

Este proyecto nace de mi curiosidad por el desarrollo de extensiones de , con el objetivo final de que se pueda utilizar en una DAW, **Reaper** en mi caso.

Las características del plugin son las siguientes:

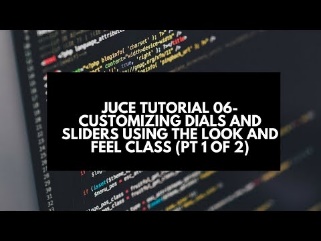
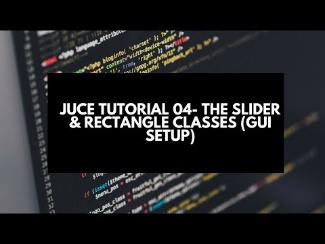
Desarrollo de un plugin VST en JUCE que lance eventos MIDI según patrones rítmicos euclidianos especificados mediante una división pulsos/compás, basado en HY-RPE2. Estos eventos podrán definirse directamente, aleatorizarse, relativizarse a otro canal o modificarse desde MIDI entrante. Al menos se podrán configurar 2 schedulers a la vez, pero me gustaría generalizarlo para n.

# Tecnologías utilizadas

## Juce

Juce es un framework de código abierto para desarrollar aplicaciones de audio para Windows, macOS, Linux, iOS y Android; y plugins VST, VST3, AU, AUv3, AAX y LV2; en C++.

Para aprender a usar Juce, he usado esta [lista de reproducción](https://youtube.com/playlist?list=PLLgJJsrdwhPxa6-02-CeHW8ocwSwl2jnu) de [The Audio Programmer](https://www.youtube.com/@TheAudioProgrammer), además de, evidentemente, la [API oficial de Juce](https://docs.juce.com/master/index.html). En particular, los vídeos que más me han ayudado en la realización de este proyecto han sido:

[](https://www.youtube.com/embed/tgf6J8foCiw?list=PLLgJJsrdwhPxa6-02-CeHW8ocwSwl2jnu)[](https://www.youtube.com/embed/po46y8UKPOY?list=PLLgJJsrdwhPxa6-02-CeHW8ocwSwl2jnu)[](https://www.youtube.com/embed/viR1Z7byTYw?list=PLLgJJsrdwhPxa6-02-CeHW8ocwSwl2jnu)[](https://www.youtube.com/embed/U5eHees1qTE?list=PLLgJJsrdwhPxa6-02-CeHW8ocwSwl2jnu)

## HY-RPE2

HY-RPE2 es un plugin secuenciador MIDI, con 2 modos de secuenciación: cuadriculada y euclidiana. A mí me interesa analizar la secuenciación euclidiana, que se presenta en esta ventana:

Interfaz de usuario gráfica

Descripción generada automáticamente

Lo que más llama la atención es la representación de ritmos euclidianos como una serie de circunferencias divididas concéntricas de colores. Cada circunferencia representa la fila de su color, y enviará un evento MIDI cuando la división que toque en ese momento esté coloreada. Voy a fijarme en los parámetros de cada fila y voy a intentar reproducirla.

# Desarrollo del plugin

Juce ofrece varias plantillas de proyectos. Para mi objetivo he seleccionado la plantilla “Plug-In>Basic”, que crea automáticamente archivos .h y .cpp de PluginProcessor y PluginEditor, que implementan en mi caso las clases JavierCano\_ProyectoFinalAudioProcessor y JavierCano\_ JavierCano\_ProyectoFinalAudioProcessorEditor, respectivamente. Por abreviar, las continuaré llamando PluginProcessor y PluginEditor.

La clase PluginEditor es la que coloca y pinta los elementos gráficos en la ventana. Principalmente quiero que tenga una serie de filas como las de HY-RPE2, cada una representando un ritmo euclidiano. Para ello he creado una clase EuclideanRhythm que hereda de juce::Component.

#define ROWS\_NUMBER 4

EuclideanRhythm euclideanRhythm[ROWS\_NUMBER];

De esta manera, aunque el número de filas no sea una variable, programaré el plugin de manera generalizable para n filas. A continuación, muestro como queda con 4 filas u 8.

Una captura de pantalla de un celular

Descripción generada automáticamente con confianza media

Interfaz de usuario gráfica

Descripción generada automáticamente

En estas capturas podemos ver todos los parámetros que tiene una clase EuclideanRhythm. El funcionamiento de esos parámetros es el esperable, y su implementación es trivial, con la excepción del tipo de MIDI. Esto es el desplegable con 4 opciones: absolute, relative, incoming MIDI y random range.

La clase PluginProcessor es la que se encarga procesar los bloques de audio y MIDI. El plugin de ritmo euclidiano necesita generar eventos MIDI en los pulsos correspondientes. El problema se descompone en 2:

1. Calcular el reparto de pulsos según el algoritmo euclidiano
2. Enviar los eventos MIDI cuando correspondan

Gráfico, Gráfico de líneas

Descripción generada automáticamenteEl problema 1 se resuelve mediante una implementación de un algoritmo que resuelva el problema. Al principio, traté de implementar el [**algoritmo de Toussaint**](https://en.wikipedia.org/wiki/Euclidean_rhythm#Summary_of_algorithm), que consiste en utilizar una matriz para transformar un vector de tamaño t comienza con todos los pulsos al principio y termina con el reparto más uniforme posible, donde t es el numero de tiempos y p es el numero de pulsos marcados en ese tiempo. Pero tras una clase de laboratorio, Jaime me enseñó el [**algoritmo de Bresenham**](https://chromatone.center/theory/rhythm/system/euclidean/), que es mucho más sencillo y directo. Se basa en aplicar el mismo cálculo que para trazar una línea recta mediante píxeles: simplemente aumentar un valor en coma flotante de forma constante y truncar su valor al entero por debajo.

vector<bool> EuclideanRhythm::bresenhamAlgorithm(int steps, int pulses)

{

float slope = (float) pulses / steps;

vector<bool> rhythm;

int previous = -1;

for (int i = 0; i < steps; i++) {

int current = floorf(i \* slope);

rhythm.push\_back(previous != current);

previous = current;

}

return rhythm;

}

El problema 2 se soluciona gracias a uno de los tutoriales que comenté anteriormente, en el que enseñan a crear y modificar mensajes MIDI. Utilizo las variables beat, channel, pitch, velocity para marcar el tiempo y generar la nota correspondiente, que se actualizan con cada tick (60 Hz) y se utilizan en cada procesamiento de bloque (48 kHz).

void EuclideanRhythm::processMIDI(MidiBuffer& midiMessages)

{

if (!enabled || mute)

return;

//…

if (currentBeat != previousBeat) {

previousBeat = currentBeat;

if (getBeat(currentBeat % (int) stepsSlider.getValue())) {

MidiMessage newMessage = MidiMessage::noteOn(channel, pitch, velocity);

processedBuffer.addEvent(newMessage, ++samplePosition);

}

}

midiMessages.swapWith(processedBuffer);

}

Al probar el plugin en Reaper, me encuentro con un problema: el plugin sólo funciona cuando la interfaz existe. Esto se debe a que los ritmos euclidianos los creo en el editor, por lo que debo cambiar la estructura teniendo esto en cuenta. Por suerte, al haberlos creado en una clase separada, basta con cambiar el propietario de los ritmos euclidianos al procesador. De esa manera, continúa sonando todo el rato, y además los valores se mantienen automáticamente.

Me gustaría que el plugin funcionara al compás del DAW, es decir, sólo siempre que se esté reproduciendo y siempre de la misma forma en todas las reproducciones. Ahora mismo, el plugin manda mensajes independientemente del estado del DAW. La solución que se me ocurre es que mi variable beat venga dada por el DAW. Ya no necesito ni saber los BPM.

void JavierCano\_ProyectoFinalAudioProcessor::getCurrentDAWBeat()

{

//Acceder al tiempo del DAW

AudioPlayHead\* playHead = getPlayHead();

if (playHead != nullptr) {

auto playPosition = playHead->getPosition();

if (playPosition != nullopt) {

auto playPositionInQuarterNotes = playPosition->getPpqPosition();

if (playPositionInQuarterNotes != nullopt)

beat = \*playPositionInQuarterNotes;

}

}

}

Lo único que le falta ahora para ser un plugin de verdad, es que el estado del plugin se mantenga entre sesiones. Para hacer esto, es necesario crear parámetros de forma que el DAW pueda guardar su estado y cargarlo más adelante. Estos parámetros se gestionan gracias a la clase AudioProcessorValueTreeState. He tenido que refactorizar mis variables para cambiarlas por parámetros, y vincularlas con su botón o control deslizante correspondiente, según [este ejemplo](https://docs.juce.com/master/tutorial_audio_processor_value_tree_state.html). Automáticamente, esto permite guardar el estado del plugin y cargarlo, hacer Ctrl + Z, e incluso automatizar los valores desde el DAW.

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

# Resultados

## Posibles mejoras