# Documentation des apprentissages

# Création d'un plugin Audio avec JUCE

## Semaine 1

### 1) Utilisation global de JUCE

- Installation du framework JUCE
- Utilisation du Projucer
- Création d'un projet dans le Projucer
- Génération des fichiers sources
- Génération du vst3 avec vs2022
- Intégration du vst3 dans Ableton live

### 2) Codification

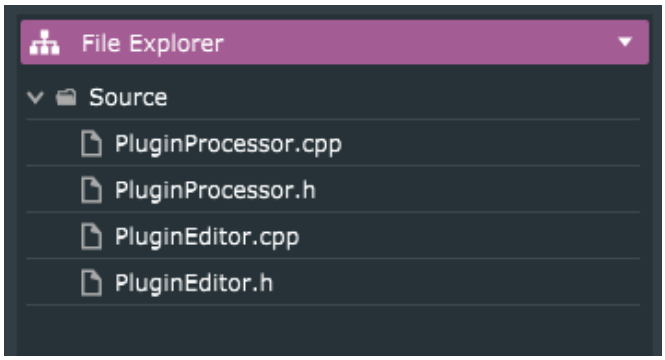
- Utilisation et organisation des fichiers C++ et Header
- Familiarisation avec le langage C++
- Prototypage d'un fader audio
- Théorie et utilisation du DSP (Digital signal processing)

## Semaine 2

### 1) Git/GitHub and JUCE

- Création d'un répertoire git
- Git ignore (Builds & JuceLibraryCode)
- Il faut push uniquement le project.jucer et le dossier Source

## 2) Architecture des Plug-in Audio




Le Plugin Processor :

- Où se trouve le traitement sonore
- On y retrouve les datas et les signaux audios

Les fonctions du processor.cpp:

- processBlock est une des fonctions les plus importantes dans le plugin processor. C'est dans cette fonction que le signal audio entre et sort.

- getStateInformation est une fonction qui sauvegarde les paramètres définis.

- setStateInformation est une fonction qui permet de charger les valeurs que l'on veut afficher à l'ouverture du plugin.

Le Plugin Editor :

- Ce que l'utilisateur voit
- L'interface, les boutons etc…
- Ce qui permet à l'utilisateur de modifier le son

Les fonctions du editor.cpp :

- PluginProcessorEditor (le constructeur) est une fonction où on définit des paramètres et des valeurs pour construire l'interface du plugin. Il faut tout d'abord initialiser la taille du plugin ex: setSize (400,300);
- Le constructeur est une fonction initialisée par JUCE lors de la création d'un projet.

- paint est une fonction qui définit les couleurs et la taille des fonts dans le plugin.

- resized est une fonction ou on définit la position de certains éléments dans le plugin.

## 3) Les premières étapes de création d'un plugin (de distorsion)

- AudioProcessorValueTreeState Class peut se connecter presque automatiquement à un slider ou un bouton pour garder son état UI et son processeur à jour.
-

ref: https://docs.juce.com/master/tutorial_audio_processor_value_tree_state.html

Pour utiliser cette classe il faut avoir des objets visuels par exemple des sliders et des boutons. On a aussi le data de ces objets qui ont une certaine valeur. Il peut aussi avoir une valeur booléenne connecté à ces objets.

Étape à suivre pour l'intégrer :

1- Dans le Plugin Processor.h on crée notre objet ValueTreeState :

juce::AudioProcessorValueTreeState apvts;

```
juce::AudioProcessorValueTreeState apvts; //creation de l objet tree state
```

2- On crée nos paramètre:

```
private:

    juce::AudioProcessorValueTreeState::ParameterLayout createParamaters();
```

3- Dans le Plugin Processor.cpp
On implémente la class pour créer nos paramètres

```
juce::AudioProcessorValueTreeState::ParameterLayout DistoVSTAudioProcessor::createParameters() {

}
```

4- On crée une liste des paramètres

```
juce::AudioProcessorValueTreeState::ParameterLayout DistoVSTAudioProcessor::createParameters() {

    std::vector<std::unique_ptr<juce::RangedAudioParameter>> params; // listes des parameters

}
```

5- On crée les valeurs des paramètres
(**String** parameterID, **String** parameterName, float minValue, float maxValue, float defaultValue)

ex:

```
params.push_back(std::make_unique<juce::AudioParameterFloat>("INPUTDB", "InputDB", 0.0f, 1.0f, 0.5f));
```

6- On retourne la valeur à la fin de la class

```
return { params.begin(), params.end() };
```

7- Dans la première fonction du audioProcessor.cpp il faut référencer la class apvts créée à l'étape 1

- Important la ligne 22 commençant par la virgule

```
11
12    //==============================================================================
13    DistoVSTAudioProcessor::DistoVSTAudioProcessor()
14    #ifndef JucePlugin_PreferredChannelConfigurations
15        : AudioProcessor (BusesProperties()
16                          #if ! JucePlugin_IsMidiEffect
17                           #if ! JucePlugin_IsSynth
18                            .withInput  ("Input",  juce::AudioChannelSet::stereo(), true)
19                           #endif
20                            .withOutput ("Output", juce::AudioChannelSet::stereo(), true)
21                          #endif
22                            ), apvts (*this, nullptr, "Parameters", createParameters())
23    #endif
```

8- Créer un slider
Dans le PluginEditor.h on crée un Slider

```
juce::Slider inputKnob; //input pointer
juce::Slider driveKnob;
juce::Slider mixKnob;
juce::Slider volumeKnob;
```

9- Créer un slider visuel

- Dans le PluginEditor.cpp

```
inputKnob.setSliderStyle(juce::Slider::SliderStyle::RotaryHorizontalVerticalDrag);
inputKnob.setTextBoxStyle(juce::Slider::NoTextBox, false, 100, 100);
addAndMakeVisible(inputKnob);
```

## 9.5 Astuce pour créer des sliderStyle

- Créer un void pour un slider pour éviter de répéter le style du slider
  L'étape 9 peut être placé dans ce void

```
void DistoVSTAudioProcessorEditor::setSliderParams(juce::Slider& slider)   // fonction qui definit un slider
                                                                           // evite de repetter le code pour crer plusieurs slider
{
    slider.setSliderStyle(juce::Slider::SliderStyle::RotaryHorizontalVerticalDrag);
    slider.setTextBoxStyle(juce::Slider::NoTextBox, false, 50, 50);

    addAndMakeVisible(slider);


}
```

- Dans l'editor.h
- Intégrer le void setSliderParams

```
//==============================================================================
/**
*/
class DistoVSTAudioProcessorEditor  : public juce::AudioProcessorEditor
{
public:
    DistoVSTAudioProcessorEditor (DistoVSTAudioProcessor&);
    ~DistoVSTAudioProcessorEditor() override;

    //==============================================================================
    void paint (juce::Graphics&) override;
    void resized() override;

private:

    void setSliderParams(juce::Slider& slider);

    juce::Slider inputKnob; //input pointer
    juce::Slider driveKnob;
    juce::Slider mixKnob;
    juce::Slider volumeKnob;
```

- Inclure ensuite dans le processorEditor
- On applique la classe du slider a tous les sliders qu'on a créé

```
//==============================================================================
DistoVSTAudioProcessorEditor::DistoVSTAudioProcessorEditor (DistoVSTAudioProcessor& p)
    : AudioProcessorEditor (&p), audioProcessor (p)
{


    setSliderParams(inputKnob); // reference a la construction dun slider (void en bas)
    setSliderParams(driveKnob);
    setSliderParams(mixKnob);
    setSliderParams(volumeKnob);
```

## 10- Relier les apvts à l'éditeur

- Créer un slider attachement

```
std::unique_ptr<juce::AudioProcessorValueTreeState::SliderAttachment> inputAttachment;  // input attachement
std::unique_ptr<juce::AudioProcessorValueTreeState::SliderAttachment> driveAttachment;
std::unique_ptr<juce::AudioProcessorValueTreeState::SliderAttachment> mixAttachment;
std::unique_ptr<juce::AudioProcessorValueTreeState::SliderAttachment> volumeAttachment;
```

11- Arrangement de la disposition visuel des faders

- Cette disposition sert à centrer les slider en répartissant les knobs également dans le canva
- Il est préférable d'utiliser des flexbox puisque ça permet d'être plus flexible
- Il est préférable de donner une valeur fixe au width et au height pour ne pas à devoir changer les nombres partout

```
void DistoVSTAudioProcessorEditor::resized()
{
    // This is generally where you'll want to lay out the positions of any
    // subcomponents in your editor..
    // on divise lespace pour que chaque knob soient centre
    inputKnob.setBounds(((getWidth() / 5) * 1) - (100 / 2), (getHeight() / 2) - (100 / 2), 100, 100);
```

12- Dans le PluginEditor.cpp il faut attacher le data du audioProcessor au slider

```
inputAttachment = std::make_unique<juce::AudioProcessorValueTreeState::SliderAttachment>(audioProcessor.apvts, "INPUTDB", inputKnob);
```

13- Justification des textes pour les paramètres

```
//==============================================================================
void DistoVSTAudioProcessorEditor::paint (juce::Graphics& g)
{
    // (Our component is opaque, so we must completely fill the background with a solid colour)
    g.fillAll (getLookAndFeel().findColour (juce::ResizableWindow::backgroundColourId));

    g.setColour (juce::Colours::white);
    g.setFont (15.0f);
   // g.drawFittedText ("Hello World!", getLocalBounds(), juce::Justification::centred, 1);

    g.drawText("InputDB", ((getWidth() / 5) * 1) - (100 / 2), (getHeight() / 2) + 5, 100, 100, juce::Justification::centred, false);
    g.drawText("DriveDB", ((getWidth() / 5) * 2) - (100 / 2), (getHeight() / 2) + 5, 100, 100, juce::Justification::centred, false);
    g.drawText("Mix", ((getWidth() / 5) * 3) - (100 / 2), (getHeight() / 2) + 5, 100, 100, juce::Justification::centred, false);
    g.drawText("VolumeDB", ((getWidth() / 5) * 4) - (100 / 2), (getHeight() / 2) + 5, 100, 100, juce::Justification::centred, false);
}
```

14- updateParams (à suivre)

15- ChainSettings class

- La création d'une class pour nos paramètres permet d'aller chercher la valeur de nos paramètres en les regroupant dans une class spécifique

```
ChainSettings getChainSettings(juce::AudioProcessorValueTreeState& apvts) {}
```

    - À l'intérieur de cette class il faut aller chercher la valeur de nos paramètres.

- Il faut intégrer la class dans le PluginProcessor.h

```
22      ChainSettings getChainSettings(juce::AudioProcessorValueTreeState& apvts);
```

- Il faut déclarer les paramètres et leur valeurs initial

```
#pragma once

#include <JuceHeader.h>

struct ChainSettings
{
    float inputDB{ 0 };
    float driveDB{ 0 };
    float mix{ 0 };
    float volumeDB{ 0 };

};

ChainSettings getChainSettings(juce::AudioProcessorValueTreeState& apvts);

```

17- Get Raw parameter value (important)

Le getRawParameterValue permet de pointer vers la valeur d'un paramètre en temps réel ex: (pendant que le user change les valeurs des knobs).

- Dans le void que nous venons de créer il faut maintenant aller chercher la valeur de nos paramètres.
- return les settings par la suite

```
ChainSettings getChainSettings(juce::AudioProcessorValueTreeState& apvts)
{
    ChainSettings settings;

    settings.inputDB = apvts.getRawParameterValue("INPUTDB")->load();
    settings.driveDB = apvts.getRawParameterValue("DRIVEDB")->load();
    settings.mix = apvts.getRawParameterValue("MIX")->load();
    settings.volumeDB = apvts.getRawParameterValue("VOLUMEDB")->load();

    return settings;
}
```

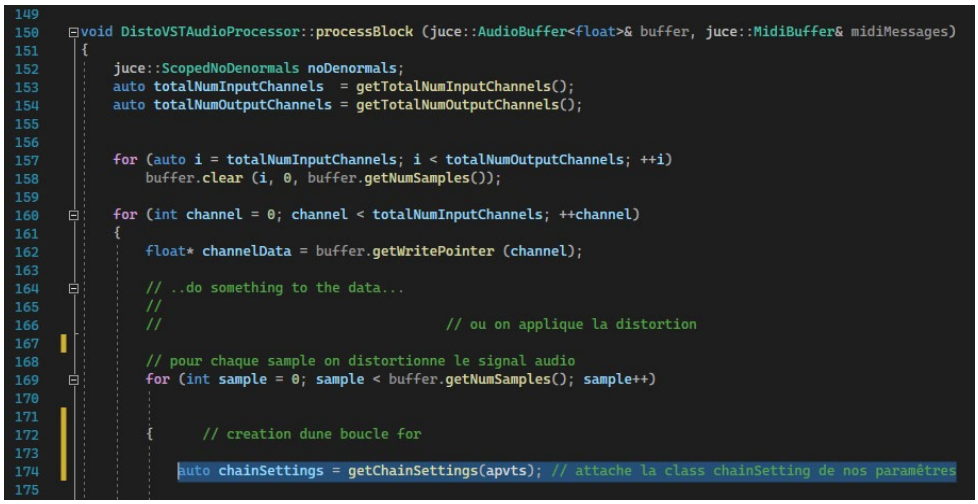
18- Dans le void prepareToPlay

- On déclare les chainSettings

```
//==============================================================================
void DistoVSTAudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)
{

    // Use this method as the place to do any pre-playback
    // initialisation that you need..

    auto chainSettings = getChainSettings(apvts);


}
```

19- Dans le processBlock (pluginProcessor.cpp)

- On déclare les chainSettings



```
149
150  void DistoVSTAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer, juce::MidiBuffer& midiMessages)
151  {
152      juce::ScopedNoDenormals noDenormals;
153      auto totalNumInputChannels  = getTotalNumInputChannels();
154      auto totalNumOutputChannels = getTotalNumOutputChannels();
155
156
157      for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
158          buffer.clear (i, 0, buffer.getNumSamples());
159
160      for (int channel = 0; channel < totalNumInputChannels; ++channel)
161      {
162          float* channelData = buffer.getWritePointer (channel);
163
164          // ..do something to the data...
165          //
166          //                                  // ou on applique la distortion
167
168          // pour chaque sample on distortionne le signal audio
169          for (int sample = 0; sample < buffer.getNumSamples(); sample++)
170
171
172          {       // creation dune boucle for
173
174              auto chainSettings = getChainSettings(apvts); // attache la class chainSetting de nos paramètres
175
```

20- Processing de ces valeurs à suivre après la théorie sur la distorsion (plus bas)

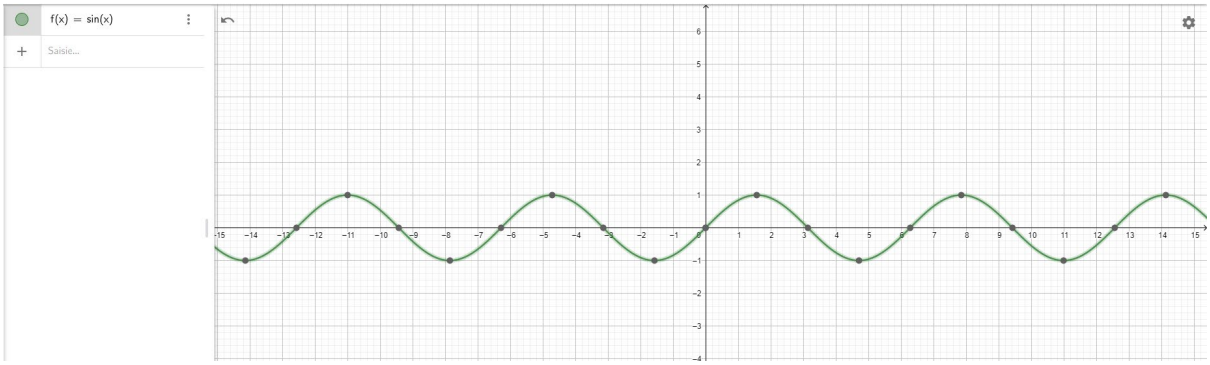
## 4) Les paramètres et caractéristique de la distorsion

Un signal audio (waveform) distortionne lorsque la valeur de son amplitude franchit la limite.

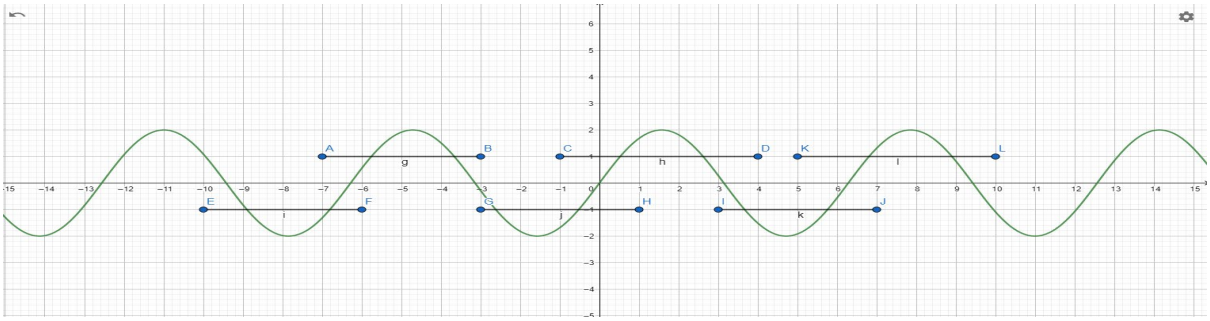
Si on augmente l'amplitude (x2) le signal ne distortionne pas, il sera simplement 2x plus fort.

En assignant une limite par exemple à 1, le signal audio va clipper lorsqu'il touchera le seuil définit, ce qui résulterait en une nouvelle waveform (distorsionné)

1) clean waveform




2) distorted waveform




Pour créer cette première distorsion, il faut définir une valeur max et min…

// si le signal audio est plus grand que 1 alors ça valeur est de 1

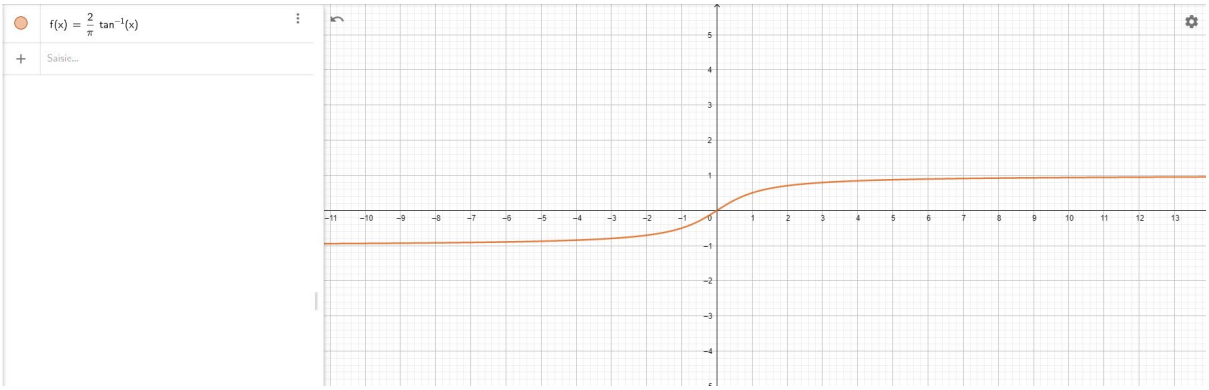
ex: *channelData *= drive;

```
if (*channelData > 1){
    *channelData = 1;
}
```

Par contre cette distorsion est intense et n’est pas plaisante pour nos oreilles.

Il faut créer une distorsion plus plaisante et smooth en créant une limite moins drastique.

f(x)=((2)/(π)) tan^(-1)(x)

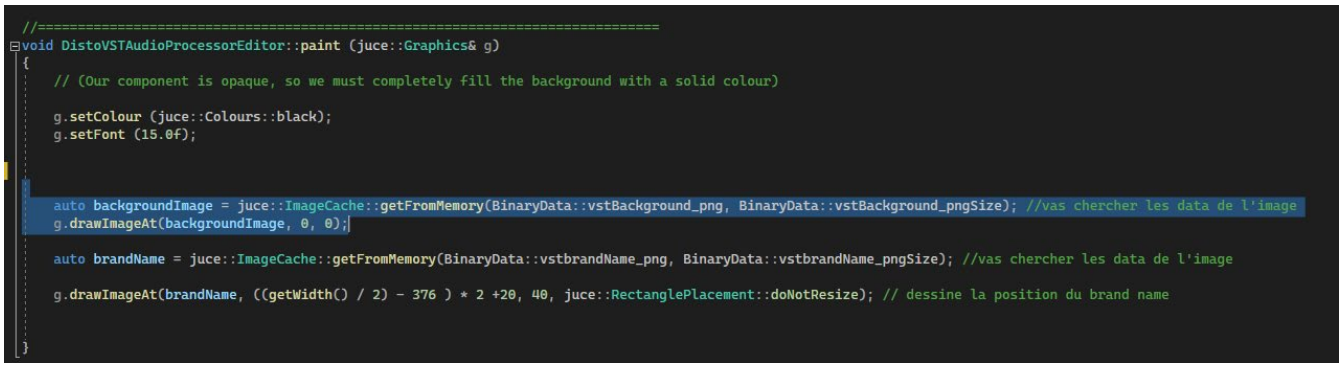



```
// pour chaque sample on distortionne le signal audio
for (int sample = 0; sample < buffer.getNumSamples(); sample++)

{      // creation dune boucle for

    auto chainSettings = getChainSettings(apvts); // attache la class chainSetting de nos paramêtres

     //--------------------------------------------------------------------------nouvelle version disto----------------

        const auto input = *channelData * juce::Decibels::decibelsToGain(chainSettings.inputDB);

        const auto disto = piDivisor * std::atanf(input * juce::Decibels::decibelsToGain(chainSettings.driveDB));

        auto melange = input * (1.0 - chainSettings.mix) + disto * chainSettings.mix;

        melange *= juce::Decibels::decibelsToGain(chainSettings.volumeDB);

        *channelData = melange;

        channelData++; // increment le point pour que ca pointe vers le prochain channel data

}
```

# 5) Gui et interface utilisateur

## 5.1) Utilisation d'image pour l'interface

- Dans le PluginEditor.cpp
- Dans le void paint



```cpp
//==============================================================================
void DistoVSTAudioProcessorEditor::paint (juce::Graphics& g)
{
    // (Our component is opaque, so we must completely fill the background with a solid colour)

    g.setColour (juce::Colours::black);
    g.setFont (15.0f);


    auto backgroundImage = juce::ImageCache::getFromMemory(BinaryData::vstBackground_png, BinaryData::vstBackground_pngSize); //vas chercher les data de l'image
    g.drawImageAt(backgroundImage, 0, 0);

    auto brandName = juce::ImageCache::getFromMemory(BinaryData::vstbrandName_png, BinaryData::vstbrandName_pngSize); //vas chercher les data de l'image

    g.drawImageAt(brandName, ((getWidth() / 2) - 376 ) * 2 +20, 40, juce::RectanglePlacement::doNotResize); // dessine la position du brand name


}
```

## 5.2) Création de Level meters graphique

1- Component/ classes
2- RMS Level et calcul
3- Les graphiques et les animations

À consulter pour le script:
https://github.com/labbeLouis98/VerticalRMS

## 5.3) FlexBox

- Les flexbox permettent d’organiser le layout visuel plus facilement.
- Dans le PluginEditor.cpp

```
// on divise lespace pour que chaque knob soient centre

//flex rules

/*-----------------------------------------------*/
// header

Rectangle<int> bounds = getLocalBounds();

//row header
FlexBox flexboxHead; //item
flexboxHead.flexDirection = FlexBox::Direction::row;
flexboxHead.flexWrap = FlexBox::Wrap::noWrap;
flexboxHead.alignContent = FlexBox::AlignContent::center;
flexboxHead.justifyContent = juce::FlexBox::JustifyContent::spaceBetween;

// ajoute des items dans le tableau du conteneur
Array<FlexItem> itemArrayHead;
itemArrayHead.add(FlexItem(80,80,toggleBypass).withMargin(20));
itemArrayHead.add(FlexItem(80, 80, menu).withMargin(20));

flexboxHead.items = itemArrayHead;
flexboxHead.performLayout(bounds.removeFromTop((bounds.getHeight() / 3) - 90));

/*-----------------------------------------------*/
```

## 6) Utilisation du data (xml) in JUCE

- Sauvegarde des states des paramètres

```
//==============================================================================
void DistoVSTAudioProcessor::getStateInformation (juce::MemoryBlock& destData)
{
    // You should use this method to store your parameters in the memory block.
    // You could do that either as raw data, or use the XML or ValueTree classes
    // as intermediaries to make it easy to save and load complex data.

    juce::MemoryOutputStream mos(destData, true); // save la state du plugin
    apvts.state.writeToStream(mos);

}

void DistoVSTAudioProcessor::setStateInformation (const void* data, int sizeInBytes)
{
    // You should use this method to restore your parameters from this memory block,
    // whose contents will have been created by the getStateInformation() call.


    auto tree = juce::ValueTree::readFromData(data, sizeInBytes);

    if(tree.isValid())

    {
        apvts.replaceState(tree);

    }

}
```

Sauvegardes des presets :

À suivre…

## Important :

L'ensemble des scripts se retrouve dans un repository GitHub :
https://github.com/labbeLouis98/DistoVST
Une grande partie du code est généré par JUCE.
L'entièreté du code n'est pas couverte dans ce document.
Ce document couvre l'essentiel du code pour créer un plugin audio.
Si certaines lignes de code n'est pas commenté, veuillez-vous référer à ce document et à la documentation JUCE : https://docs.juce.com/master/index.html