FICHE D'EXERCICES – Analyse et programmation JUCE / C++

Objectif général

Étudier le fonctionnement d'un composant audio dans JUCE à travers la génération d'une sinusoïde, tout en consolidant les compétences en C++ moderne (pointeurs, classes, override, RAII, etc.).

EXERCICE 1 — Lecture et compréhension du code

1.1 — Cycle de vie AudioAppComponent

Explique en détail le rôle des méthodes suivantes dans le cycle de vie audio de JUCE :

- `prepareToPlay`
- `getNextAudioBlock`
- `releaseResources`

But : comprendre comment JUCE gère l'audio en temps réel.

1.2 — Analyse du constructeur

Explique ce que fait précisément l'appel :

```
setAudioChannels(0, 2);
```

Pourquoi 0 entrées et 2 sorties ? Que se passe-t-il si on met un autre nombre ?

1.3 — Deux pointeurs de buffers

Dans :

```
auto* left  = bufferToFill.buffer->getWritePointer(0);
```

```
auto* right = bufferToFill.buffer->getWritePointer(1);
```

- À quoi correspondent ces pointeurs ?
- Que se passerait-il si le système audio n'avait qu'un seul canal ?

EXERCICE 2 — Audio et DSP

2.1 — Calcul de la sinusoïde

Explique mathématiquement à quoi correspond :

```
angleDelta = 2π * frequency / sampleRate;
```

Comment cette valeur garantit-elle que l'oscillateur ne se désaccorde pas ?

2.2 — Aliasing

Que se passe-t-il si `frequency` dépasse `sampleRate / 2` ?
Explique le phénomène d'aliasing et comment l'éviter.

2.3 — Expérimentation

Modifie le code pour produire un **signal carré** à la place d'une sinusoïde.
Quelle est la plus grande difficulté de cette approche "naïve" ?

EXERCICE 3 — GUI (Interface graphique)

3.1 — Méthode paint

Explique pourquoi `paint` peut être appelée :

- sans prévenir
- plusieurs fois par seconde
- même si l'audio ne change pas

Comment éviter d’effectuer des opérations lourdes dans cette méthode ?

3.2 — Interaction

Propose une interface simple permettant de modifier la fréquence en temps réel (ex. slider).
Quels problèmes de **thread-safety** faudrait-il gérer ?

EXERCICE 4 — C++ moderne

4.1 — RAII

Pourquoi le destructeur appelle-t-il :

```
shutdownAudio();
```

Que se passerait-il si on l’oubliait ?

4.2 — Pointeurs intelligents

Explique pourquoi `mainWindow` est un :

```
std::unique_ptr<MainWindow>
```

À quel moment ce pointeur est-il automatiquement détruit ?
Pourquoi ne pas utiliser un pointeur brut ?

4.3 — Macro de JUCE

Décris précisément le rôle de :

```
JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR(SineWaveComponent)
```

Comment agirait le programme sans ce macro ?

EXERCICE 5 — Améliorations et refactoring

5.1 — Séparation des responsabilités

Le composant génère l'audio *et* gère l'interface graphique.
Propose une architecture alternative plus modulaire (ex : classe Oscillator).

5.2 — Contrôle du volume

Ajoute un contrôle de volume au programme.
Explique ce qu'il faut modifier dans `getNextAudioBlock`.

5.3 — Suppression des clics audio

Comment éviter les clics lorsque :

- la fréquence change brusquement
- le volume change brusquement

Propose des solutions (ex: ramping).

EXERCICE 6 — Aller plus loin

6.1 — Visualisation audio

Comment dessiner en temps réel la forme d'onde générée dans la fenêtre ?

6.2 — Polyphonie

Comment étendre ce code pour générer plusieurs notes en même temps ?

6.3 — Passage à un vrai synthé

Quelles étapes seraient nécessaires pour transformer ce générateur de sinusoïdes en :

- un synthétiseur piloté par MIDI ?
- un plugin VST/AU ?

ANNEXE

Main.cpp

```
#include <JuceHeader.h>

//==============================================================================
// Composant principal qui génère la sinusoïde
class SineWaveComponent : public juce::AudioAppComponent
{
public:
    SineWaveComponent()
    {
        setAudioChannels(0, 2); // 0 entrée, 2 sorties
        setSize(400, 200);
    }

    ~SineWaveComponent() override
    {
        shutdownAudio();
    }


//==============================================================================

    void prepareToPlay(int /*samplesPerBlockExpected*/, double sampleRate)
override
    {
        currentSampleRate = sampleRate;
        currentAngle = 0.0;
        angleDelta = juce::MathConstants<double>::twoPi * frequency /
currentSampleRate;
    }

    void releaseResources() override {}

    void getNextAudioBlock(const juce::AudioSourceChannelInfo& bufferToFill)
```

```
override
    {
        auto* left  = bufferToFill.buffer->getWritePointer(0,
bufferToFill.startSample);
        auto* right = bufferToFill.buffer->getWritePointer(1,
bufferToFill.startSample);

        for (int sample = 0; sample < bufferToFill.numSamples; ++sample)
        {
            float sampleValue = (float) std::sin(currentAngle);
            currentAngle += angleDelta;

            left[sample]  = sampleValue;
            right[sample] = sampleValue;
        }
    }


//==============================================================================
====

    void paint(juce::Graphics& g) override
    {
        g.fillAll(juce::Colours::black);
        g.setColour(juce::Colours::white);
        g.setFont(24.0f);
        g.drawText("Sinusoïde : " + juce::String(frequency) + " Hz",
                   getLocalBounds(), juce::Justification::centred, true);
    }

    void resized() override {}

private:
    double currentSampleRate = 44100.0;
    double currentAngle = 0.0;
    double angleDelta = 0.0;

    double frequency = 16000.0; // fréquence en Hz (440 = La)

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR(SineWaveComponent)
};

//==============================================================================
====
// Application JUCE (point d'entrée)
```

```
class SineWaveAppApplication : public juce::JUCEApplication
{
public:
    const juce::String getApplicationName() override       { return "Sine
Wave App"; }
    const juce::String getApplicationVersion() override    { return "1.0.0";
}
    bool moreThanOneInstanceAllowed() override             { return true; }

    void initialise(const juce::String&) override
    {
        mainWindow.reset(new MainWindow(getApplicationName()));
    }

    void shutdown() override
    {
        mainWindow = nullptr;
    }

    void systemRequestedQuit() override
    {
        quit();
    }

    void anotherInstanceStarted(const juce::String&) override {}

    //==============================================================

    class MainWindow : public juce::DocumentWindow
    {
    public:
        MainWindow(juce::String name)
            : DocumentWindow(name,

juce::Desktop::getInstance().getDefaultLookAndFeel()

.findColour(ResizableWindow::backgroundColourId),
                                DocumentWindow::allButtons)
        {
            setUsingNativeTitleBar(true);
            setResizable(true, true);
            setContentOwned(new SineWaveComponent(), true);

            centreWithSize(getWidth(), getHeight());
            setVisible(true);
        }
```

```
        void closeButtonPressed() override
        {
            juce::JUCEApplication::getInstance()->systemRequestedQuit();
        }
    };

private:
    std::unique_ptr<MainWindow> mainWindow;
};

// Macro d'entrée de l'application
START_JUCE_APPLICATION(SineWaveAppApplication)
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.15)

#project(JuceSliderDemo VERSION 0.0.1)
project(SineWaveApp VERSION 1.0)

# Modifier avec le bon chemin vers JUCE
add_subdirectory(JUCE)

juce_add_gui_app(SineWaveApp
    PRODUCT_NAME "SineWaveGenerator"
)

juce_generate_juce_header(SineWaveApp)

target_sources(SineWaveApp
    PRIVATE
        Source/Main.cpp
)

target_link_libraries(SineWaveApp
    PRIVATE
        juce::juce_gui_extra
        juce::juce_audio_utils
        juce::juce_gui_basics
        juce::juce_graphics
        juce::juce_core
)
```

```
target_compile_definitions(SineWaveApp PRIVATE
    JUCE_WEB_BROWSER=0
    JUCE_USE_CURL=0
)
```