**Software Architecture Description for the Effects System**

**Overview**

This document describes the software architecture of the **Effects System**. The system is designed to apply a series of audio effects to audio frames in a clean, extensible, and decoupled manner.

**Core Design Goals**

* **Polymorphism:** Abstract base class to unify all effect implementations.
* **Pipeline Flexibility:** Allow dynamic chaining and ordering of effects.
* **UI/Logic Separation:** Keep processing logic independent from UI components.
* **Memory Safety:** Use modern C++ smart pointers to manage lifetimes.
* **Extensibility:** Adding new effects should require minimal effort.
* **Metadata Access:** Enable UI to query static metadata about each effect without impacting DSP performance.

**Key Components**

**1. EffectBase (Abstract Interface)**

**Purpose:** Defines the interface that all audio effects must implement.

**Methods:**

* prepare(double sampleRate, int samplesPerBlock): Prepare resources or settings.
* processFrame(float\* data, int numSamples): Apply the effect to an audio frame.
* reset(): Reset internal state.
* getName() const: Return a human-readable name for the effect.
* getMetadata() const: Return a static description object with effect information (for UI use only).

**Key Properties:**

* Pure virtual methods (interface-only, no concrete implementation).
* Separation between real-time processing and UI metadata.

**2. Concrete Effects (e.g., GainEffect)**

**Purpose:** Implements specific effect behavior.

**Example: GainEffect**

* Multiplies incoming audio samples by a gain factor.
* Simple, efficient implementation.

**Responsibilities:**

* Implement EffectBase methods.
* Manage their own internal parameters (e.g., gain value).
* Provide static metadata for UI discovery.

**3. EffectPipeline**

**Purpose:** Manages a collection of effects and processes them in series.

**Responsibilities:**

* addEffect(std::unique\_ptr<EffectBase>): Add a new effect to the chain.
* clear(): Remove all effects.
* prepareAll(double, int): Prepare all effects with the audio system settings.
* processAll(float\*, int): Process audio through all effects in order.
* resetAll(): Reset the state of all effects.

**Internal Structure:**

* std::vector<std::unique\_ptr<EffectBase>> effects;

**4. Effect Metadata System**

**Purpose:**

* Provide rich descriptive information about each effect for use in the UI.
* Facilitate dynamic creation of icons, labels, tooltips, and categorized browsing.

**Structure:**

* EffectMetadata is a lightweight, static struct included with each effect.
* Accessed via getMetadata() method without affecting processing.

**Fields:**

* name: Short name of the effect (e.g., "Gain").
* description: A sentence describing what the effect does.
* category: Broad grouping such as "Utility", "Filter", "Distortion".
* iconName: String referring to a UI resource (icon or sprite).

**Use in UI:**

* Display appropriate icons next to effect names.
* Show help tooltips or detail views based on description.
* Group effects into categories for user browsing.
* Search or filter effects by category or name.

**Performance Considerations:**

* Fully static, read-only access ensures no impact on real-time DSP performance.
* Metadata is only queried when displaying or configuring effects, never during processing.

**5. Effect UI Components (Planned)**

**Purpose:**

* Visualize effects in the system.
* Allow selection, parameter editing, and bypassing.

**Characteristics:**

* Will query EffectBase instances for display names and static metadata.
* Will not directly alter the processing order (delegated to control logic).

**Data Flow Diagram**

1. Audio frame enters the system.
2. The EffectPipeline calls processFrame() on each EffectBase in sequence.
3. Each effect modifies the audio frame.
4. Final audio frame exits the pipeline.

**Design Justification**

* **Open/Closed Principle:** New effects can be added without modifying the core system.
* **Single Responsibility:** Each class has one clear reason to change.
* **Runtime Flexibility:** Effects can be added, removed, or reordered during runtime.
* **Low Coupling:** UI and logic are separated, simplifying testing and maintenance.
* **Real-Time Safe:** Metadata access is strictly separated and never touches DSP-critical code.

**Future Extensions**

* **Effect Bypassing:** Add enable/disable flags for effects.
* **Serialization:** Save and restore pipeline states.
* **Automation:** Allow real-time modulation of effect parameters.
* **Parallel Processing:** Optional, if individual effects are thread-safe.
* **Dynamic UI Construction:** Based on effect metadata.

**Appendix A: Code Examples**

**EffectBase.h**

#pragma once

struct EffectMetadata

{

const char\* name;

const char\* description;

const char\* category;

const char\* iconName;

};

class EffectBase

{

public:

virtual ~EffectBase() = default;

virtual void prepare(double sampleRate, int samplesPerBlock) = 0;

virtual void processFrame(float\* data, int numSamples) = 0;

virtual void reset() = 0;

virtual const char\* getName() const = 0;

virtual const EffectMetadata& getMetadata() const = 0;

};

**GainEffect.h**

#pragma once

#include "EffectBase.h"

class GainEffect : public EffectBase

{

public:

GainEffect(float initialGain = 1.0f);

void prepare(double sampleRate, int samplesPerBlock) override;

void processFrame(float\* data, int numSamples) override;

void reset() override;

const char\* getName() const override;

const EffectMetadata& getMetadata() const override;

void setGain(float newGain);

private:

float gain;

static const EffectMetadata metadata;

};

**GainEffect.cpp**

#include "GainEffect.h"

const EffectMetadata GainEffect::metadata = {

"Gain",

"Adjusts the volume of the incoming signal.",

"Utility",

"gain\_icon"

};

GainEffect::GainEffect(float initialGain)

: gain(initialGain)

{

}

void GainEffect::prepare(double, int)

{

}

void GainEffect::processFrame(float\* data, int numSamples)

{

for (int i = 0; i < numSamples; ++i)

data[i] \*= gain;

}

void GainEffect::reset()

{

}

const char\* GainEffect::getName() const

{

return metadata.name;

}

const EffectMetadata& GainEffect::getMetadata() const

{

return metadata;

}

void GainEffect::setGain(float newGain)

{

gain = newGain;

}

**EffectPipeline.h**

#pragma once

#include <vector>

#include <memory>

#include "EffectBase.h"

class EffectPipeline

{

public:

void addEffect(std::unique\_ptr<EffectBase> effect);

void clear();

void prepareAll(double sampleRate, int samplesPerBlock);

void processAll(float\* data, int numSamples);

void resetAll();

private:

std::vector<std::unique\_ptr<EffectBase>> effects;

};

**EffectPipeline.cpp**

#include "EffectPipeline.h"

void EffectPipeline::addEffect(std::unique\_ptr<EffectBase> effect)

{

effects.push\_back(std::move(effect));

}

void EffectPipeline::clear()

{

effects.clear();

}

void EffectPipeline::prepareAll(double sampleRate, int samplesPerBlock)

{

for (auto& e : effects)

e->prepare(sampleRate, samplesPerBlock);

}

void EffectPipeline::processAll(float\* data, int numSamples)

{

for (auto& e : effects)

e->processFrame(data, numSamples);

}

void EffectPipeline::resetAll()

{

for (auto& e : effects)

e->reset();

}