Audio Graph

***Abstract*—Many tools for computer sound production either require prior knowledge on the subject or don't allow insight into the details of the implementations. This results in a high entry barrier into computer sound production. In this report, we introduce a new application for sound synthesis, which allows for real-time manipulation and viewing of the data while it's being processed. It's a modern and intuitive GUI that follows the visual programming idea and allows users to start using the tool immediately without prior knowledge of programming and computer sound production. In the report, we first do some research on the current relevant tools on the market and present their strengths and weaknesses. Then, the details of the implementation are explained and basic usage of the tool showcased. At the end, the results are showcased and the strengths and weaknesses of this application explained.**

I. Introduction

Computer sound generation became the norm in the current era of entertainment industries. Games, movies, music, and other industries have all adopted computer sound generation, and practically all of the products these industries produce use some form of computer sound production. *Audio Graph*'s main goal is to help users learn the basics of computer sound generation without the need for programming using the concept of visual programming with nodes. The implementation of sound synthesis with visual nodes can help the user visualize the process with which the sound is produced from a signal generator, the part where that signal is modified and transformed to the output at the end.

There are a lot of tools available for programmatic sound synthesis, but either require knowledge of programming (i.e., *JSyn* [1]) or are offered as a complete tool that doesn't allow access to the details of the implementation of each node or part of the sound generation graph (i.e., *AudioNodes* [2]).

Audio Graph offers both of these advantages in one solution, where the user doesn't require any knowledge of computer sound production or programming and can figure it out easily by playing with the nodes themselves while allowing the more advanced user to dive deep into the details of the implementation and even implement their own nodes and add them to the collection.

The next, second chapter goes into the strengths and weaknesses of current tools on the market. Then, the third chapter present the implementation and code, used to implement the application, It also presents three example graphs, assembled with the tool. The fourth chapter talks about what the project resulted in and at the end, the conclusion overviews on the work done and the results achieved.

II. Related work

In the first chapter, Introduction I, we mention *JSyn* [1] and *AudioNodes* [2].

JSyn is an audio synthesis API for Java. It allows development of interactive computer music programs used for sound effects, audio environments, or music. They use unit generators, which can be connected together to produce sound. The API allows implementation of new nodes with the existing API. [1] It doesn't, however, allow users to see the units visually and requires prior knowledge of programming in Java to use. Even with the knowledge of programming in Java, the user will most likely require learning about the tool to use it.

AudioNodes [2] is a tool available for use in the browser or by downloading the application. The application uses visual programming for creating the audio graph. It is similar to *JSyn* in that regard. It doesn't, however, allow programmatic implementation of new nodes, and the user cannot see the programmed implementations of nodes. It also contains a free and a paid tier.

III. Implementation

The implementation of *Audio Graph* is divided into two parts. This allows logical separation between two individual systems. One is `AudioEngine`, which is a synthesis engine, similar to *JSyn*, and the other, `AudioEditor`, which is responsible for editing the audio graph. Another advantage of this method is that the `AudioEngine` part can be swapped for another engine, like *JSyn*.

A. Audio Engine

`AudioEngine` is its own project and has no dependency on the `AudioEditor` project. It's dependent on third-party libraries, *RtAudio* [10] and *AudioFile* [11]. *RtAudio* [10] is used to output the audio data to the default system device. *RtAudio* was a good choice because it is lightweight and only used for input and output of audio and because it is cross-platform. *AudioFile* [11] is used for writing and reading *.wav* files.

The API consists of three main classes:

- `AudioNode`
- `AudioPin`
- `AudioLink`

`AudioNode` is the base class for all nodes. It contains virtual methods, which are then used in the implementation of each node class. `AudioPin` is a generic class of type T and has two implementations:

- `InputAudioPin`
- `OutputAudioPin`

The generic class defines what type is passed through the `AudioPin`. We define `AudioNodes`'s inputs and outputs by defining them as fields in the class implementation. In

the AudioNode constructor we initialize the AudioPins. AudioLink is also a generic class of type T and is used to connect an OutputPin to an InputPin. T defines what data is traveling through the AudioLink. There is one main class, AudioEngine, which is responsible for updating, managing, creating, removing, connecting, and disconnecting nodes. We also define a helper class AudioData which represents audio being sent through a AudioLink. It contains a CircularBuffer<T> and other helping fields. Lastly, the AudioNode implementations are in AudioEngine.

B. Audio Editor

AudioEditor is another project with dependencies to all the GUI libraries. We add dependencies to:

- *glad* for binding to *OpenGL* [3]
- *GLFW* [5] for creating windows and receiving input and events
- *ImGui* [6] and extensions *ImGui Node Editor* [7], *ImPlot* [8] and *ImGuiFileDialog* [9] for creating the application GUI
- AudioEngine

All these dependencies are then used to create an application GUI to interact with the AudioEngine *audio graph*.

C. Sine Oscillator Node

An example implementation of a AudioNode is the SineOscillatorNode. It has two float inputs for frequency and amplitude and one AudioData* output. We define them with fields frequencyIn, amplitudeIn and audioOut. We also have to implement the destructor and abstract method getPins for the AudioEditor to work. This class implements the PlayableAudioNode, which extends the AudioNode and adds two virtual methods, play and stop, which are used to start and stop the streaming of data.

```
class SineOscillatorNode : public PlayableAudioNode
    {
public:
    InputPin<float>* frequencyIn = nullptr;
    InputPin<float>* amplitudeIn = nullptr;
    OutputPin<AudioData*>* audioOut = nullptr;
private:
    RtAudio* audio = nullptr;
    AudioData* data = nullptr;
    float time = 0.0f;
    bool isStreaming = false;
public:
    SineOscillatorNode();
    ~SineOscillatorNode() override;
    int getPins(AudioPinGeneric**& pins) const
        override;
    void play() override;
    void stop() override;
private:
    AudioData* getOutputData();
};
```

We use the constructor to initialize the AudioPins. The output pin audioOut is linked to the function getOutputData, which fills the buffer in AudioData and returns it.

```
SineOscillatorNode::SineOscillatorNode() {
    frequencyIn = new InputPin<float>();
    frequencyIn->name = "Frequency";
    frequencyIn->defaultValue = 440;
    amplitudeIn = new InputPin<float>();
    amplitudeIn->name = "Amplitude";
    amplitudeIn->defaultValue = 0.5;
    audioOut = new OutputPin<AudioData*>();
    audioOut->name = "Audio";
    audioOut->function = [this] { return
        getOutputData(); };
    data = new AudioData(STANDARD_BUFFER_SIZE);
}
```

The function getOutputData takes the inputs for frequency and amplitude and fills the buffer until full with:

$$amplitude * sin(2 * \pi * frequency * phase)$$

```
AudioData* SineOscillatorNode::getOutputData() {
    float frequency = frequencyIn->getValue();
    float amplitude = amplitudeIn->getValue();
    if (isStreaming) {
        while (!data->buffer->full()) {
            float sample = amplitude * sin(2 * M_PI *
                frequency * time);
            data->buffer->push(sample);
            time += 1.0f / static_cast<float>(data->
                sampleRate);
        }
    }
    else {
        data->buffer->clear();
        time = 0.0f;
    }
    return data;
}
```

We need to register this class to a AudioNodeFactory, and the node will automatically be added to the *Audio Editor* shown in figure 1.

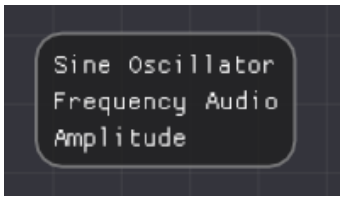

Fig. 1. Sine Oscillator Node in *Audio Editor* with two inputs frequency and amplitude and one output audio.

The same method is used to implement other AudioNodes and enable the user to create their own nodes to manipulate audio.

Current collection of audio nodes categorized:

- Generators
  - Sine Oscillator
  - Triangle Oscillator
  - Sawtooth Oscillator
  - Square Oscillator
- Modifiers
  - Add Signal
  - Sub Signal
  - Float Multiply
  - Simple Click Filter

  - General Click Filter
  - Real Time General Click Filter
- Input
  - Read From WAV
- Output
  - Write To WAV
  - Line Out
- Utilities
  - Float Range
  - Value Float
- Editor
  - Signal Graph
  - File

D. Signal Graph Node

Signal Graph Node is a node for visualizing audio data being sent through links in real-time. It uses *ImPlot* [8] to plot the graph of samples. Example is on figure 2.

E. Click filter

We tried implementing two sorts of *click filters* in `AudioEngine`. A click is an artifact in sound, which we generally want to avoid. Click filters in general try to detect clicks by analyzing samples and then correct or minimize those clicks. There are two methods, which we'll call *simple* and *general*.

The *simple* method compares two neighboring amplitudes and checks for the difference between those values. If the difference is higher than the constant threshold, those amplitudes are set to 0. This approach works fine but has its flaws, and the clicks can still be noticed, as shown in figure 3.

The *general* method, however, propagates two windows through all the samples, one large window and one small window. Then it checks for the difference between the average amplitude of the samples in the small and the large window. This requires processing of data before playing it, which is not suitable for real-time audio processing, done in *Audio Graph*. Because of this, we implement two different nodes:

- General Click Filter
- Real Time General Click Filter

The *General Click Filter* uses the method mentioned before, but that doesn't work in real-time, and when the user plays the audio, the program will process the data and then, after the process, output it. The *Real Time General Click Filter*, however, does the data processing on chunks retrieved from the *Read From WAV* node. This allows us to do real-time processing and playing of the data but produces different results. This node is shown on figure 4.

IV. Results

The project resulted in a tool for curious people learning about computer sound synthesis. *Audio Graph* allows users to create graphs that synthesize data without requiring prior knowledge on the subject and programming skills. The users can start using *Audio Graph* out of the box and try to create a desired sound. It allows for real-time modifications to the synthesis graph while the audio is playing, resulting in better visualization of the process of synthesis and improved tunability. This overall significantly lowers the barrier to entry into computer sound production and can help understand basic concepts. The *Signal Graph Node* allows for on-the-fly viewing of data while it's being processed. For example, figure 3 showcases the same signal while it's being processed before and after entering the *Simple Click Filter*. *Audio Graph* can also be useful to more advanced users, as its API enables programmers and sound engineers to implement and add custom nodes to the *Audio Editor* and see how they manipulate the data.

V. Conclusion

The project addresses the high barrier to entry of computer sound production. A lot of tools on the market currently either require prior knowledge on the subject and programming skills or are offered as a complete package without much customizability and insight into the implementation. We implemented an application that allows users to create new synthesis graphs from scratch just by "playing with" the nodes provided and instantly hear the result of that graph. It allows for on-the-fly viewing of the data while it's being processed. On the other hand, the tool allows for the creation of custom nodes and insight into the implementation of said nodes. The paper showcases and explains a few examples of graphs and nodes assembled with *Audio Graph* and presents the data while it's being modified by nodes. Another aspect of the tool is that it enables simple implementation of new nodes for audio manipulation and can grow and update with time.

References

[1] JSyn: API for developing interactive sound applications in Java https://www.softsynth.com/jsyn/
[2] Audio Nodes https://www.audionodes.com/
[3] OpenGL https://www.opengl.org/
[4] Glad https://github.com/Dav1dde/glad
[5] GLFW https://www.glfw.org/
[6] ImGui: API for GUI in C++ https://github.com/ocornut/imgui
[7] Imgui Node Editor: Implementation of a node editor using ImGui https://github.com/thedmd/imgui-node-editor
[8] ImPlot: Implementation for creating interactive plots https://github.com/epezent/implot
[9] ImGuiFileDialog: Implementation of a file selection dialog using ImGui https://github.com/aiekick/ImGuiFileDialog
[10] RtAudio: API for realtime audio input/output in C++ https://github.com/thestk/rtaudio
[11] AudioFile: A simple header-only C++ library for reading and writing audio files https://github.com/adamstark/AudioFile

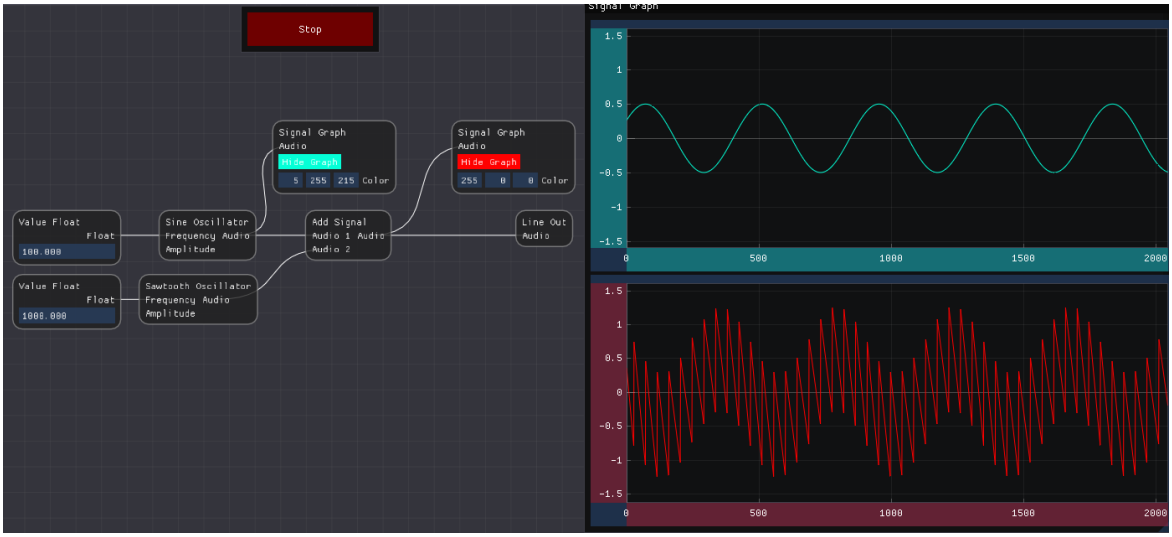

Fig. 2. A sine oscillator signal and a sawtooth oscillator signal added together with intermediate signal graphs. The blue graph represents the raw sine wave from the Sine Oscillator while the red showcases the two signals summed together. The click position is marked with red.

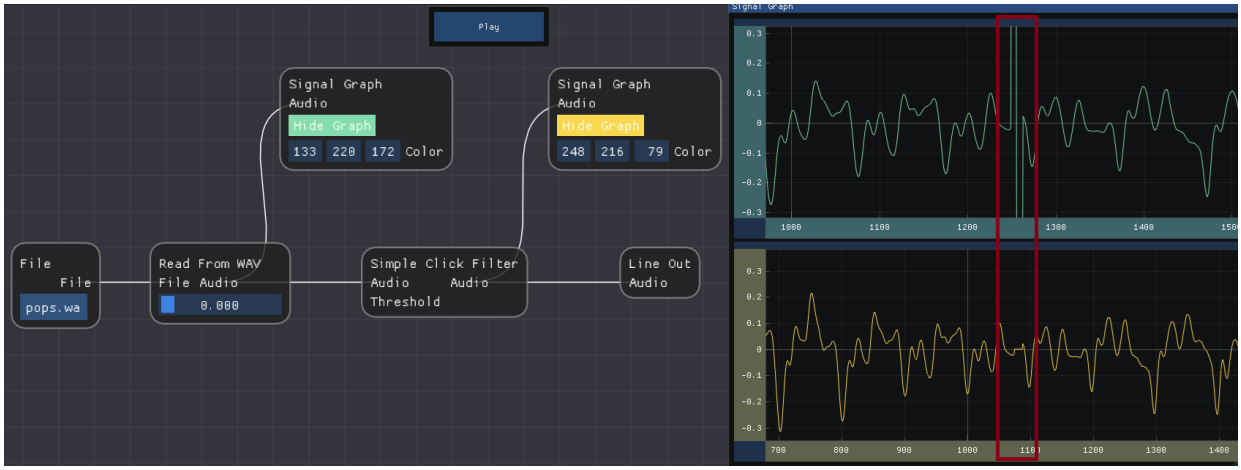

Fig. 3. Graph of a simple click filter on a wav file. Top, blue graph, shows the signal before the click is removed while the second, yellow graph, shows the click being flattened to 0.

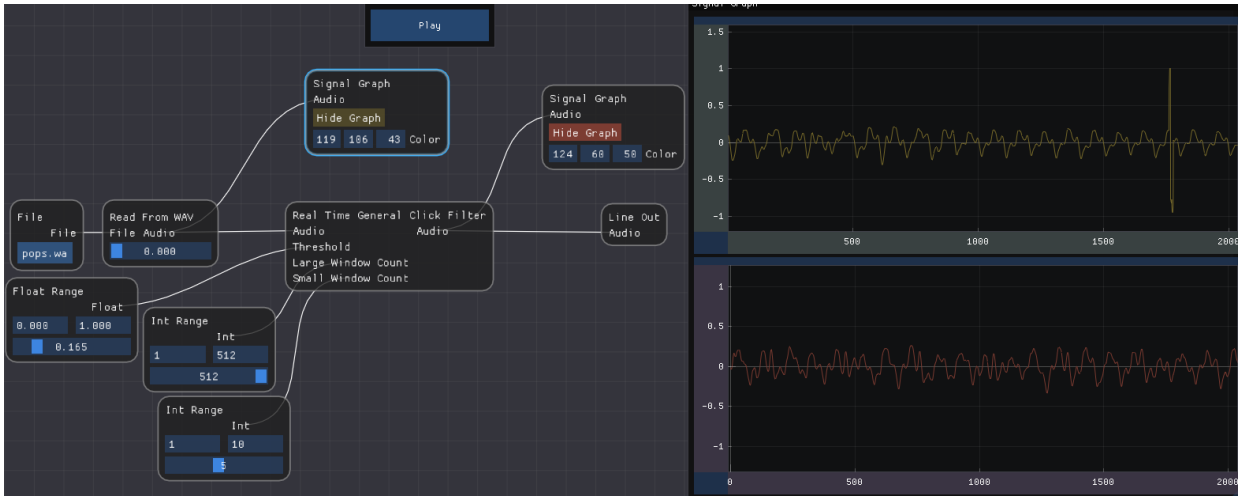

Fig. 4. Graph of a general click filter on a wav file. Top, yellow graph, shows the original signal before the click is removed while the second, red graph, shows the signal being modified with interpolation to remove the click.