18-500: ECE Design Experience

Team D8: Stairway to Hamerschlag - A Guitar Pedal Effects Simulator

Joseph Kim, Matthew Kasper, Stephen He

Electrical and Computer Engineering
Carnegie Mellon University

May 9, 2019

Index Terms: Audio Processing, Circuits, Circuit Simulation, Guitar Effects, Guitar Pedals, Transient Analysis.

Abstract

We implement a guitar pedal effects simulator capable of applying both analog and digital effects to live or recorded audio through a user-friendly graphical interface. Ultimately, this has the potential to speed up development time and lower costs for professional guitar pedal developers or hobbyists looking to explore new sounds.

1 Introduction

1.1 Problem Area

Guitar pedals are circuits placed between an electric guitar and an amplifier that applies a transformation to an audio signal, resulting in a sound that can be distorted in interesting ways. It is common for both professional audio engineers and hobbyists looking to explore new effects to manually tinker with circuitry in an effort to produce the highest quality sounds. This can be a painstaking process–making a change requires swapping out real, physical components. This costs both time and money. For example, we purchased a DIY kit for a simple overdrive pedal. Constructing the pedal took $30 and nearly two hours of assembly time. Furthermore, we never had the opportunity to try the pedal out to see if it truly produced the desired effect before investing this sort of time and money. While other tools like LiveSPICE provide similar capabilities, the big drawback is that SPICE is extremely complicated. Our tool should allow users to have a smoother user experience without having to know the ins and outs of a sophisticated circuit simulator like SPICE.

Our project aims to eliminate these problems by delivering a tool to simulate guitar pedal circuits and effects entirely in software. Using our project, users can plug a guitar into the computer, design their circuit on our GUI (or choose from pre-built effects), and simulate what the real pedal would sound like. In addition to allowing users to connect their guitar to our application, we also allow users to run simulations against recorded audio files. We hope that this software will enable users to cut out some of the costs (both in terms of time and money) associated with tinkering on their own guitar pedals.

1.2 Project Goals

Early on, we identified several goals we wanted to achieve in order for our project to be successful. Above all else, we wanted to deliver an excellent all around user-experience. This can be further broken down into three categories:

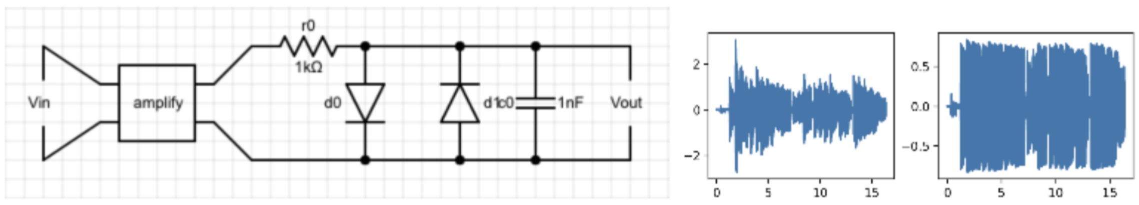


Figure 1: Diode clipping circuit with plots of the input and output waveform. It is clear from the output waveform that the voltage was clipped at ± 0.7V, as expected.

1. We need to deliver a simple, friendly GUI.
2. We need to be able to produce pleasant-sounding and accurate audio effects using our circuit simulator and audio processor's digital effect blocks.
3. Our end-to-end latency for live audio needs to be below 50ms (the threshold at which a delay will start to be perceptible to the human ear).

If the latency is too high, the system will not feel natural for musicians trying to use our project.

2 Design Requirements

2.1 User-friendly Interface

Since our application is primarily built with musicians and guitar pedal designers in mind, a strong user experience is our primary goal. Without an intuitive user interface, the rest of the project could work seamlessly and our target users would still not be able to benefit from our work. For this reason, we have identified the qualitative requirement that our users must find the interface friendly and simple to use.

To evaluate this requirement, we conducted a small user-study of various Carnegie Mellon students. Though we recognize that our fellow students may not be experienced guitar pedal designers, we felt as though this was the most representative feedback we could get with the constraints on our time and resources for this project. During this user-study, we hope to provide users with simple instructions for designing and simulating a basic clipping circuit such as a fuzz pedal using our software. Then, we had our participants rate our application on (1) ease of use, (2) overall satisfaction, and (3) unobtrusiveness of latency. On a scale from 1 to 10, with 10 being the best and 1 being the worst, we hoped to achieve an average score above 7.5 for each of these metrics.

2.2 Accurate Circuit Simulation and Pleasant Audio Effects

Correct circuit simulation is said to be "considered an art... much of it is based on idiosyncrasies of particular simulators or tricks that are based largely on luck." [1]. Currently, the best technique for circuit simulation involves solving a system of equations using Newton's Method and sparse matrix techniques. However, convergence in Newton's Methods is not guaranteed, which may lead to inaccurate simulations in certain cases.

For our application, we wanted to provide an accurate simulation for simple pedal designs and circuits, namely diode clipping circuits, low pass filters, high pass filters, and band pass filters. We constrained our tests to these circuits to avoid the added complexity and divergence issues that more complicated circuits may add to our simulator.

To test our simulation accuracy, we ran sample signals through the test circuit, plot both the input and output signals and their Fourier transformation, and qualitatively judged the output signals for correctness. For example, to test the diode clipping circuit, we will confirm whether or not the output signal looks like the input signal, but clipped at ± 0.7 Volts. A sample circuit and waveform are shown in Figure 1. To test any of the filters, we will confirm whether or not the output signal looks like the input signal with the correct region of frequencies filtered out based on the Fourier transform plot.

In addition to our circuit simulation, our application allows users to use built-in 'digital effect blocks'. In particular, we will be providing the following effect blocks: Fuzz, Distortion, Reverb, Delay, and Amplify. As one of our goals, we want our built-in effect blocks to provide pleasant audio effects.

During our user study, we will ask for participants to rate the pleasantness of our audio effect blocks. On a scale from 1 to 10, with 10 being extremely pleasant and 1 being extremely unpleasant, we are aiming for an average score of 7.5.

2.3 Low Round-Trip Latency Audio Processing

In order to enable effect simulation using live audio signals, we recognize the need to have a "fast" circuit simulator and audio processor, which will enable signals to flow through our system, pass through our digital effect blocks, be transformed by our circuit simulator, and reach the output speaker quickly. We define round-trip latency as the time taken between when a single audio sample hits the audio processor input to when it is available for output to the speaker.

Based on estimates of the limits of the human ear to detect phase differences in audio signals, we decided to aim for an average round-trip latency of no more than 50ms. We plan to test this by instrumenting our audio processor with timers to measure when inputs and outputs are available. This enabled us to measure the round-trip latency of various combinations of effect blocks and simple circuits.

3 Architecture

The user flow is as follows:

1. User opens application, creates or loads a project, and is presented with the visual circuit editor.
2. The user designs and edits the circuit with the graphical interface.
3. To simulate the circuit design on some audio, the user saves their design and runs the simulation.
4. The user chooses between an audio file or live audio as an input. Regardless of choice, the user can listen to the output either from an audio file out or in real time.
5. If unsatisfied, the user can return to the visual circuit editor and make changes to their circuit.

This process can be repeated as many times as needed.

Our system architecture can be broken down into three main components:

- **Frontend** - Manages all interaction with the user, including capturing the pedal design, creating / saving project files, and launching simulations.
- **Circuit Simulator** - Processes all analog effects by simulating user circuit designs against the input signal.
- **Audio Processor** - Acts as the gateway for audio signals to enter and exit our system. Processes all user-specified digital effects.

3.1 Frontend

In our system, the frontend is responsible for managing all interaction with users, including capturing user schematics, creating / saving project files, and launching simulations at the user's request. When a user saves a new pedal design, the frontend generates a netlist file specifying how the components in the circuit connect to one another in a way that can be recognized on the backend. This netlist file will be passed to the backend whenever a user starts a new simulation. The frontend also passes the backend the audio file selected by the user, if applicable.

The frontend is a Javascript application written using the ElectronJS framework, an open-source cross-platform library for developing desktop applications.

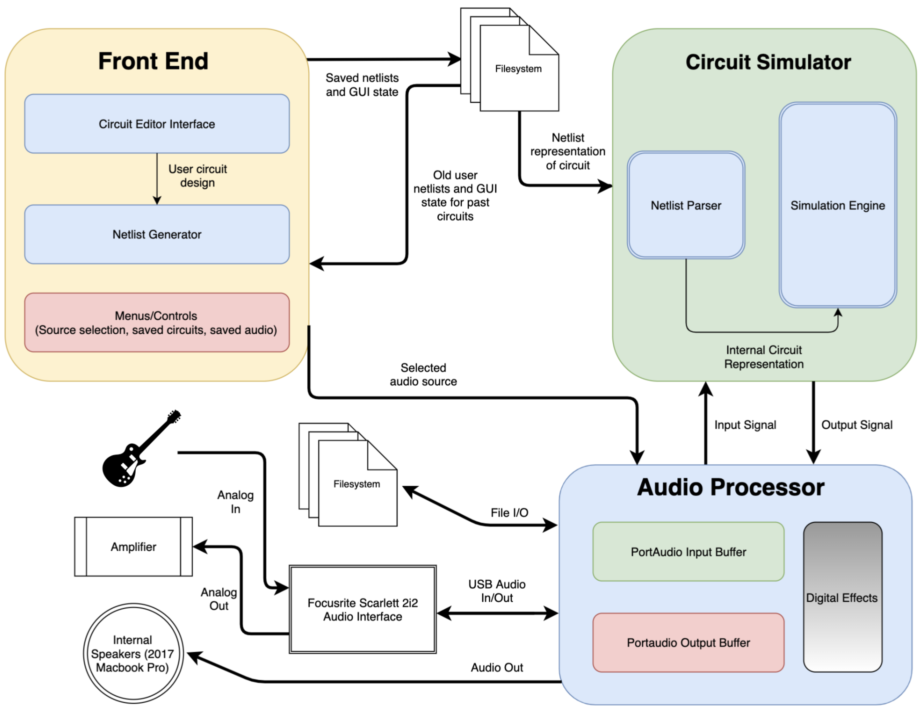


Figure 2: Block diagram showing the three submodules of our system: the frontend (yellow), the circuit simulator (green), and the audio processor (blue). The circuit simulator and audio processor are linked into a single executable, which makes up the backend of our application.

3.2 Circuit Simulator

The circuit simulator has two main responsibilities. First, it is responsible for converting the netlist file for a user-specified design into an internal format that can be used for simulation. During this phase, all digital effects that the user requested are also forwarded to the audio processor. Second, it is responsible for simulating analog effects that a user pedal design applies to an input signal to produce the simulated output. Since this simulation is on the critical path of our application, where latency is tremendously important, the circuit simulator is written in C++. This allows us to have fine-grained control over the performance of our program.

3.3 Audio Processor

The audio processor is responsible for transferring audio input to the circuit simulator and back to different audio destinations. It multiplexes audio inputs and outputs between the filesystem and the live input/output from hardware.

In addition, the audio processor is also responsible for applying any digital effects before samples are sent to the circuit simulator. Five digital effects are implemented in the audio processor: Fuzz, Reverb, Delay, Distortion and Amplification. These digital effects, when combined with a circuit or with one another, allow the user to create a multitude of sounds.

4 Design Trade Studies

4.1 Audio Hardware

Initially, we had planned to use cheap wire adapters to convert the $\frac{1}{4}''$ audio jacks to the 3.5mm jacks supported by our computers. However, it soon became evident that this approach caused a major addition of noise into our signal, which was less than ideal. We tested out whether this was an issue with our software or the hardware by feeding the input through other audio software, such as Quicktime and Audacity. Once we determined that the issue derived from hardware, we researched other methods musicians use to record guitar. We quickly learned that our attempted method is known to have issues with noise, and purchased an audio interface. Luckily, we had not spent too much money on other parts, and it fit easily into our budget.

Once we had the audio interface, we tested our live audio platform. Our setup had the guitar feed through the audio interface, which sent data to the computer via USB. Our system then sent the processed sound to the computer's built-in speaker. However, the sound produced by the speaker was stuttered, which we initially thought to be an artifact of data not being available in time. This was further supported by the fact that this problem was only fixed by artificially setting the output latency of the speaker hardware to a much higher value. This was not a viable option as we wanted to be able to hear the output at the lowest latency possible.

However, after further inspection, we discovered that it may actually have to do with the interaction between the input of the audio interface and the native output of the speaker. Even using the live playback option on Audacity produced the exact same phenomenon. After more experimentation, we found that simply feeding the output back to the audio interface fixed the issue without increasing the latency. In fact, this turned out to be the optimal solution, as it allowed us to connect directly to a guitar amplifier, letting our software truly just be like a guitar pedal.

These fixes produced a final result that was clean and with a low level of noticable latency. In fact, in the user study of our application, we achieved an average score of 9 out of 10 in terms of how good our latency was.

4.2 Smooth User Experience

One of our primary goals was to provide a smooth, simple to user graphical interface for our users. Early on, we produced a minimal interface that was nice for us to use internally, but it was never intended to be a finished product. However, we were not certain what features would be most essential in a finalized version of our GUI. To answer this question, we had a few friends look at our initial prototype. From here, we collected informal feedback about what features should be added. This motivated us to add some new features to our final user interface. First, we made the circuit editor full-screen. Second, we moved much of the functionality that was originally achieved using buttons on our schematic editor into a more organized format in the native MacOS menu bar. Third, we added keyboard shortcuts such as `Cmd+S` to allow users to perform repetitive actions in the fastest way possible. We also added some minor accent features such as a timer and moving indicator in the live-audio menu to show when a simulation is active. We also refactored the way the frontend manipulates project files, so that the user can easily run simulations or save their designs without having to navigate through a sea of popup menus, as was the case in our initial frontend implementation. An overview of the things we changed can be seen in Figure 3. Ultimately, these changes played a part in the final user study scores that we achieved on our project. Five users rated our application as a 7.4/10 on average for overall ease of use and an 8.6/10 for overall satisfaction, which leads us to believe that our efforts spent optimizing the user experience were well worth it. Though we had hoped to score slightly higher in the ease of use category, we think this could be resolved by including better usage instructions. Ultimately, we just didn't have time to incorporate this into our final solution.

5 System Description

5.1 Frontend

The frontend of our system is responsible for the following behaviors:

- Allowing the user to build circuits on a graphical interface
- Managing user works as files within a project folder

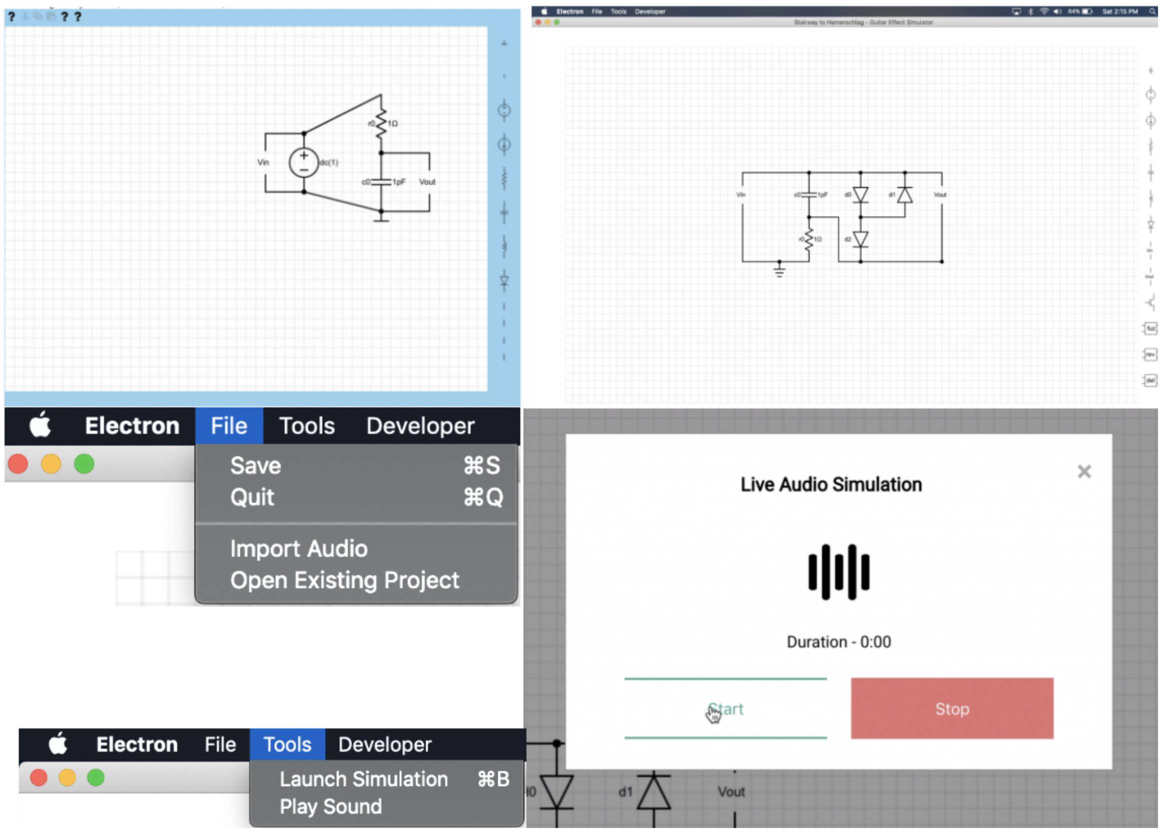


Figure 3: Evolution of our frontend based on feedback. Frontend V0 is pictured on top left. The other images show new features added in our final demo version.

- Generate netlists from the user circuit description
- Selecting audio inputs and outputs
- Forwarding user settings to the backend

The system is built with Electron, a framework for creating native cross platform desktop applications. It allows for development with traditional web technologies like Javascript, HTML and CSS. The fronted allows users to drag and drop circuit components, as well as digital audio effect blocks, onto the canvas and connect them with wires. The user is required to provide a voltage in and voltage out in their circuits, representing the input guitar signal and the modified pedal output respectively. The circuit simulator frontend is written in an object-oriented programming style. Each type of component available has a corresponding constructor function. The frontend system then treats each

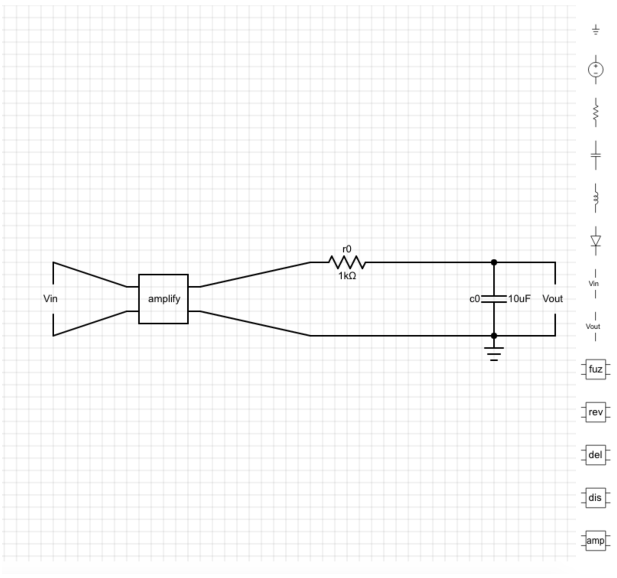


Figure 4: UI with a sample circuit

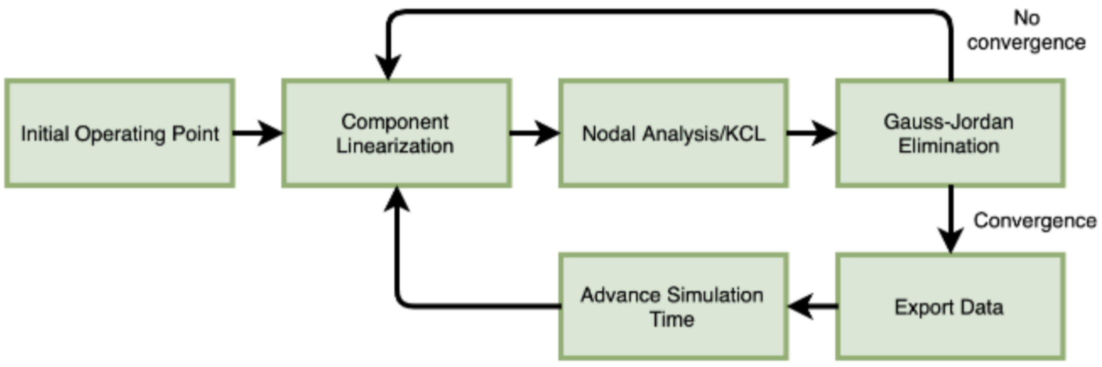


Figure 5: Simulation engine phases.

type of component the same, just with a different draw function and a few different characteristics.

The frontend's netlist generator is responsible for converting the user's arrangement of circuit parts into a netlist to pass on to the circuit simulator. This netlist generation invovles finding all components connected to the same node of the circuit. We use a simple union-find data structure to maintain the components connected at a particular node. Additionally, the order of effect blocks can effect the sound of the output signal, so we use a DFS algorithm to obtain the ordering of effect blocks from the perspective of the input voltage.

5.2 Circuit Simulator

The circuit simulator has two primary responsibilities: parsing netlists containing user circuits into an internal format, and simulating a signal as it propagates through the analog circuitry specified by the user.

5.2.1 Netlist Parser

The netlist parser converts frontend generated netlist files into the circuit representation used by the circuit simulator. The algorithm to do this is simple. For each supported component type, there is a unique identifier that will appear in the netlist. For example, a line in a netlist file that begins with the identifier `RESISTOR` can be used to specify a new resistor in the circuit. All components are also assigned a name in the netlist file, which aids in the development and debugging process by allowing the netlist to be human readable. After the component identifier and component name comes a list of parameters that depend on the component type. These could include the nodes in the circuit that the element spans, an associated component value such as a resistance or capacitance, or other parameters of the device.

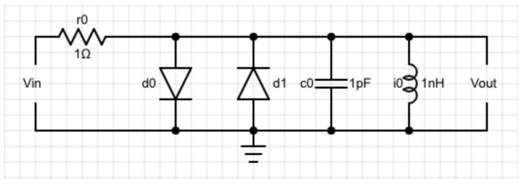


Figure 6: Sample circuit using a variety of component types.

For example, the circuit shown in Figure 6 would have a netlist representation that looks like the following:

```
RESISTOR r0 2 1 1
CAPACITOR c0 2 0 1p
DIODE d0 2 0
DIODE d1 0 2
INDUCTOR i0 2 0 1n
VOLTAGE_IN vin 1 0
VOLTAGE_OUT Vout 2 0
GROUND 0
```

An important feature of this netlist is that every component is labeled according to the nodes it spans. For example, resistor `r0` spans node 2 to node 1 based on this netlist. The circuit simulator contains an abstract class called `Component`, and each line in a netlist gets converted to a component object. For example, there is a `Capacitor` class that derives from `Component`. As the netlist parser iterates over the lines in the netlist file, it builds up a vector of `Component` objects that will later be used in simulation.

5.2.2 Simulation Engine

The simulation engine contains the core logic of the circuit simulator that allows us to do live transient analysis on audio signals. To begin, we choose an initial operating point for all of the unknowns in our system, which may consist of node voltages and currents through certain devices. When the simulation engine is first launched, all unknowns are set to zero.

At each timestep, a KCL is performed at each node in the system to produce a matrix representing the system of equations to solve. This matrix is solved using Gauss-Jordan Elimination, providing an estimate for each unknown. However, some components do not have linear I-V curves, meaning that we cannot obtain an exact solution to this system using standard linear algebra techniques such as Gauss-Jordan Elimination alone. To combat this issue, we use Newton's Method to produce an approximate solution to within a specified error threshold. After running each round of Gauss-Jordan elimination, we check how much our current solution differed from the solution on the previous Newton iteration. Once this delta becomes significantly small, we treat this as the solution for the current time unit and advance to the next sample in the input signal. However, if the error is still large, we perform further iterations of Newton's Method. In order to avoid a scenario where we do not converge on a solution, we put a cap on the maximum number of Newton iterations. Our cap is currently set to 30 iterations. This is helpful, since it gives us more predictable latency. However, it also means that for more complicated circuits, our results may not be as accurate. Figure 5 shows an overview of the simulation engine's core logic.

5.3 Audio Processor

The audio processor manages the inputs and outputs of the system. It abstracts away the input and output sources to the circuit simulator with a single API.

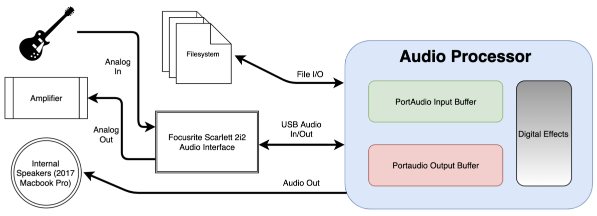


Figure 7: Audio Processor System Overview

The audio processor takes advantage of a couple libraries. To read from and write to the file system in traditional audio file formats, such as `.wav` or `.ogg`, Libsndfile is used to encode the data. The audio processor also supports a custom file format with the `.cso` extension, which stores the samplerate, the length and the samples in an easy to read format. This simple format was useful in generating plots of the input and output.

To read inputs from and write outputs to the hardware in real time, Portaudio was used. Portaudio allows input and output streams to be created, with their own associated callback functions. Since there is inherent, variable latency in processing the samples, the input samples are read and stored into an input queue. When the circuit simulator requests a sample, data is read from this queue. When the circuit simulator reports an output sample, it is stored onto a separate queue, which is flushed into the hardware on the next callback when enough samples have been collected.

If no input samples are available when requested by the circuit simulator, which may happen if the circuit simulator is able to process the samples faster than the hardware retrieves them, the call is blocked, waiting on a condition variable. Once samples are received from the hardware, a signal is sent indicating that the call to retrieve the sample may proceed. This allows the circuit simulator to operate without having to worry about the availability of samples or latencies.

The two libraries were sufficient in retrieving input from the computer's microphone and output them through the computer's speaker. However, our ultimate goal was to be able to play and hear the effects in real time. Thus, we needed a way to direct the audio from an electric guitar into the laptop – using the microphone to get guitar input would cause too much noise.

We settled on using an audio interface, specifically the Focusrite 2i2, which connects to the guitar and sends audio data to the computer via USB. It also allows the output to be routed back to the audio interface, which could then be connected to a guitar amplifier.

5.4 Integration

One of the key design issues we faced was how to integrate all three pieces of our application to provide a smooth user experience. We did this in several ways, depending on the specific case. For example, the frontend allows the users to specify a circuit design and start a new circuit simulation. This involves several steps:

1. Save the user circuit in a netlist format that can be interpreted by the backend.

2. Launch a new backend instance, passing in the netlist file as a command line argument. The backend runs as a separate process from the frontend.

3. If the user elects to stop the simulation, the frontend sends a `SIGUSR1` signal to the backend. The backend has a `SIGUSR1` handler installed that gracefully exits the application upon receiving the signal.

Another complicated integration task was to fit the two modules that make up the backend together. Unlike the frontend and backend, which run as two disjoint processes communicating via signals and the filesystem, the circuit simulator and audio processor are linked together into a single executable. Since both the audio processor and circuit simulator are written in C++, integrating these components boiled down to deciding on the API that each module would expose to the other.

We ultimately settled on an interface that we called the `AudioManager`. This was implemented as a class in our codebase that provides an API function called `AudioManager::getNextVoltage()`, which can be called by the circuit simulator to get the next sample from the input stream. Internally, the `AudioManager` decides whether this sample should come from a file or from an external audio source such as an instrument based on command line flags passed in by the frontend. This interface allows the circuit simulator to be entirely agnostic to the input audio source.

The `AudioManager` also provides another function called `AudioManager::setNextOutput()`, which is called after the circuit simulator finishes propagating an input voltage through the user circuit. The `AudioManager` is completely responsible for doing the proper thing with this output sample depending on the user-specified settings that were passed as command line arguments to the backend. For example, output samples may be logged to a file, or they may be played live through an attached audio device. Once again, this allows the circuit simulator to be agnostic to the settings chosen by the user. Figure 2 details the flow of data through our application.

6 Project Management

6.1 Schedule

Our schedule changed quite a bit over the course of the semester.

In particular, there were many cases where what was supposed to be individual work eneded up being a team effort. In the final stretch of our project, integration and systematic bug fixes took the bulk of our time. In order to progress efficiently, we worked on a lot of problems together. This was beneficial because each of us had a different expertise and understanding of our project.

After the in class demo, we decided to add digital effect blocks into our audio processor. This was not in scope before, so it had to be added into our schedule. Other tasks shifted around or fell out, as adding effect blocks became one of our top priorities late in the semester.

Refer to Appendix B for our updated Gantt chart.

6.2 Division of Labor

The three modules nicely paved way for a three-way split amongst the team members. Stephen He was responsible for the development of the frontend, Matthew Kasper spearheaded the research and implementation of the circuit simulator, and Joseph Kim lead the design choices and creation of the Audio Processor. For all other tasks, work was split evenly.

6.3 Budget

The items we purchased can be found in Appendix A. The main cost we incurred was the audio interface, which ended up being crucial in making live simulation work properly. We also saved a lot of money by using equipment that was available in Professor Sullivan's audio lab, such as the guitar, the amplifier and cables.

We initially bought many cables and adapters, but it became evident that they were not sufficient. We did so in hopes of connecting the guitar with a cable adapter to the computer's audio jack, but it turned out to be fairly noisy. Thus, we had to forego many of the cheap parts and purchased the more expensive audio interface.

6.4 Risk management

One of the major risks involved with this project was the complexity involved in circuit simulation. Thus, we approached this problem carefully and methodically. We limited the scope of our project to include only a few key components. We also decided to focus on implementing more critical circuit elements first. Doing so not only made testing more complicated parts feasible, but also made sure we had a minimum set of working components at the end that could be used to create a meaningful circuit. Thus, we began by implementing resistors, then capacitors, and then diodes, which were imperative in a clipping circuit to make a distorted effect.

Another way we mitigated challenges in implementing models for components was by creating digital effect blocks. Not only was it a useful addition to the user's arsenal of tools, but it also made up for analog components that were out of the scope or not possible to implement this project. For example, op-amps are commonly used to amplify the input signal in many pedals, but were not included in our original plan. Thus, to account for the lack of op-amps in our circuit simulator, we created a digital effect block to amplify signals.

Another form of risk was in the latency we would be able to achieve with our system when using live input and output sources. There were many possible sources of significant latency, including the audio hardware and the circuit simulation. Though this didn't actually turn out to be too big of an issue, we also mitigated this risk by giving users the option to upload audio files, where any latency in our system would not be as detrimental to the user experience.

7 Related Work

There are some other competing software products out there. For example, SPICE is a popular circuit simulator that includes most of the features that our simulator supports. LiveSPICE expands on the groundwork laid by SPICE to get into the live audio signal processing domain. It is capable of doing many of the things our project can do, but there are some features it does not support. In particular, LiveSPICE does not support digital effect blocks, which is a nice feature for a less experienced user. Second, the barrier to entry is much lower for our application. SPICE is an extremely sophisticated tool, but as a result it can be complicated to use. Rolling our own circuit simulation software allowed us to simplify the user experience. It also allowed us to achieve fine grained control over aspects of the circuit simulator that could impact our bottom-line latency.

8 Summary

Overall, we were able to achieve most of our goals. We ended with a fully functional circuit simulator supporting resistors, capacitors, inductors and diodes. We're able to process signals at a fairly low latency of 23.2 ms on certain circuits, and all with a clean user interface. From our user study, we achieved an average score of 8.6 for user satisfaction and an average score of 7.4 for ease of use. These both meet (or come extremely close to) the requirements we laid out in the beginning. During our final demo, a lot of viewers shared same sentiment about our project. We also earned an honorable mention from the Apple team who came to judge the project demos.

References

[1] Achieving Accurate Results with a Circuit Simulator; Kenneth Kundert and Ian Clifford; San Jose, California

[2] Electronic Circuit and System Simulation Methods; Lawrence Pillage.

[3] Spice A Guide to Circuit Simulation and Analysis Using PSpice; Paul W. Tulnenga

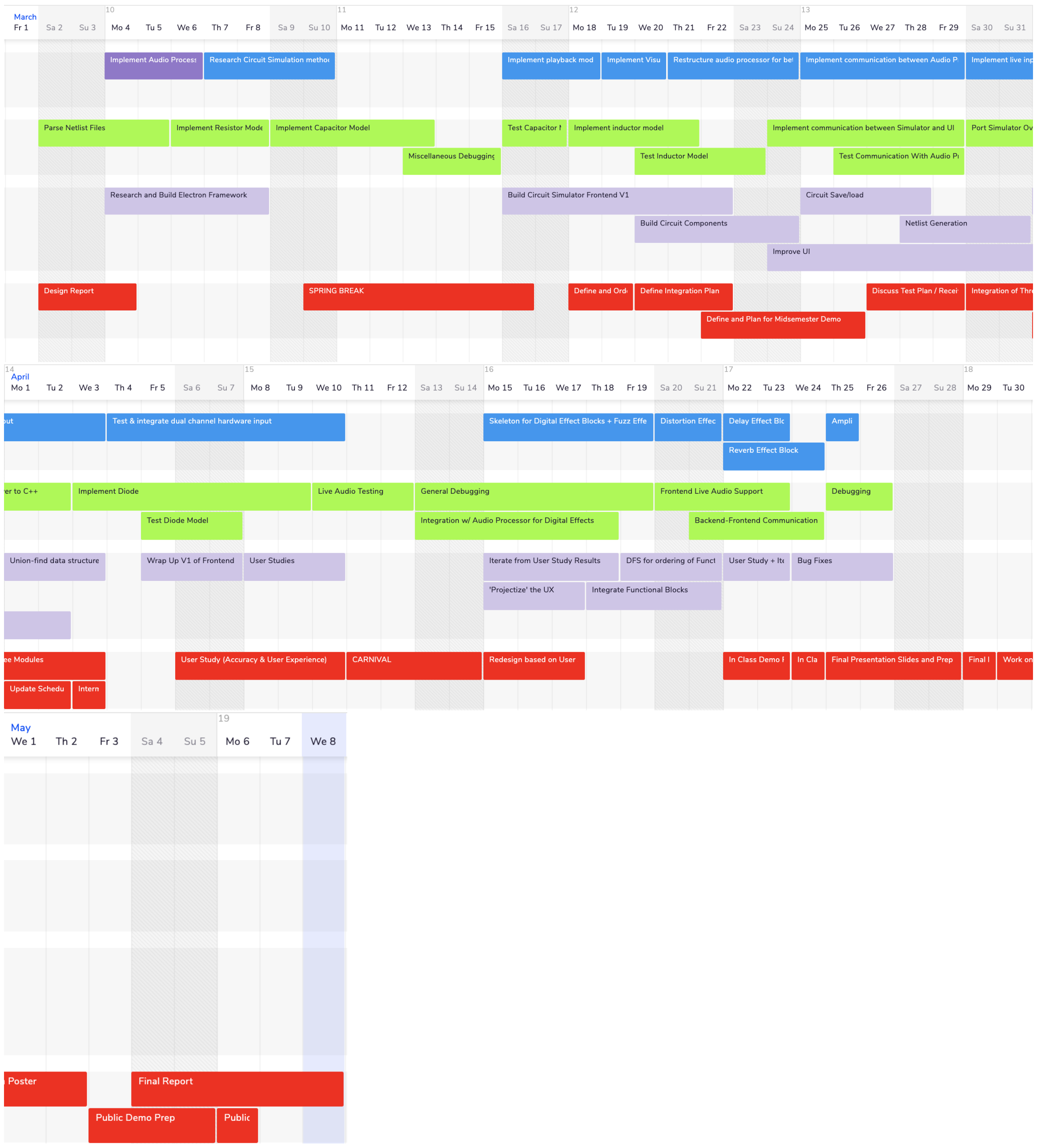
[4] Fundamentals of Computer-Aided Circuit Simulation; William J. McCalla

Appendix A: Budget

Part	Cost	Quantity	Source
Focusrite Scarlett 2i2 (2nd Generation) USB Audio Interface	\$159.99	1	Guitar Center
HOSA CMP-159 Stereo Breakout 3.5mm TRS to Dual 1/4" TS - 10'Cable	\$9.99	2	Sweetwater Sound
HOSA CSS-105 $\frac{1}{4}$" TRS to $\frac{1}{4}$" TRS Balanced Interconnect Cable	\$6.95	2	Amazon
The Thunderdrive overdrive pedal	\$40.69	1	Amplified Parts
AmazonBasics 9 Volt Everyday Alkaline Batteries (8-Pack)	\$9.49	1	Amazon
Hosa YPP-106Y Cable $\frac{1}{4}$" TSF to Dual $\frac{1}{4}$" TS 6 inch splitter	\$5.45	1	Sweetwater Sound

Total: \$232.56

Appendix B: Schedule


March
April
May
Implement Audio Process
Research Circuit Simulation metho
Implement playback mod
Implement Visu
Restructure audio processor for be
Implement communication between Audio P
Implement live inp
Parse Netlist Files
Implement Resistor Mode
Implement Capacitor Model
Miscellaneous Debugging
Test Capacitor
Implement inductor model
Test Inductor Model
Implement communication between Simulator and UI
Test Communication With Audio P
Port Simulator Ov
Research and Build Electron Framework
Build Circuit Simulator Frontend V1
Build Circuit Components
Circuit Save/load
Netlist Generation
Improve UI
Design Report
SPRING BREAK
Define and Ord
Define Integration Plan
Define and Plan for Midsemester Demo
Discuss Test Plan / Recei
Integration of Thr
Test & integrate dual channel hardware input
Skeleton for Digital Effect Blocks + Fuzz Effe
Distortion Effec
Delay Effect Blo
Ampli
Reverb Effect Block
er to C++
Implement Diode
Live Audio Testing
General Debugging
Frontend Live Audio Support
Debugging
Test Diode Model
Integration w/ Audio Processor for Digital Effects
Backend-Frontend Communication
Union-find data structure
Wrap Up V1 of Frontend
User Studies
Iterate from User Study Results
DFS for ordering of Funct
User Study + Ite
Bug Fixes
'Projectize' the UX
Integrate Functional Blocks
e Modules
User Study (Accuracy & User Experience)
CARNIVAL
Redesign based on User
In Class Demo I
In Cla
Final Presentation Slides and Prep
Final I
Work on
Update Schedu
Intern
Poster
Final Report
Public Demo Prep
Public