DESIGNING AND DEVELOPING A VIRTUAL SYNTHESIZER VST PLUG-IN

MUS 495

DR. SAMPLES

ZACH OHARA

11 JUNE 2021

The current state of music production technology and the open source programming community has made it feasible for individual developers and hobbyists to develop useful and marketable projects such as virtual synthesizer apps and plug-ins. However, the cross-disciplinary nature of these projects means that they are not usually covered by collegiate music programs or computer science degrees. Additionally, most of the developers and hobbyists taking on these projects are trained computer scientists and programmers, but lack formal music education. For my senior project, I have developed a virtual synthesizer that exists both as a standalone Windows application and a VST plugin for use with digital audio workstations. This project demonstrates a confluence of two distinct disciplinary areas of pursuit. My goal in this project is to design a synthesizer that conforms to established standards of synthesizer design, and that is simple, straightforward, and easily usable by those with limited knowledge or experience with digital synthesis. In this paper I cover my goals, my design philosophies, the end result that I have created, and some topics of discussion that have resulted from this project.

Background

A digital audio workstation, also called a DAW, is a computer program used for the composition, production, or live performance of digital music. Although there are a huge variety of DAW’s on the market today, each with their own features and intended audiences, one thing they all have in common is the ability to run virtual instruments through a platform called Virtual Studio Technology, or VST. VST instruments, which can be distributed either as plug-ins for DAW’s or as standalone applications, are platform-specific but DAW-independent. For example, this means that a VST plug-in compiled for 64-bit Windows can be used with any DAW, so long as that DAW is also running on 64-bit Windows. The ability of these plug-ins to run in any

DAW has created an environment in which developers of virtual instruments are not pressed to ensure compatibility with any specific program or group of programs.

Additionally, recent developments in computer science, especially the growing popularity of open source software, have expanded the accessibility of computer programming to hobbyists and amateurs. Combined with the widespread use of the VST platform, this means that interested hobbyists as well as professional developers are easily able to create and distribute new virtual instruments, with the most common variety being virtual synthesizers. The flexibility provided by the VST platform means that these synthesizers are able to emulate order analog technology and physical synthesizers, pursue novel or esoteric sounds through creative use of digital technology, or some combination of the two.

The cross-disciplinary nature inherent to the development of virtual instruments means that such projects are typically neglected by academia and limited to enthusiasts. Music education programs cannot teach virtual instrument development because of the technical knowledge and ability that is required to write code for audio processing, and computer science programs can’t teach it because of the musical experience and ability that is required to design a musical instrument. Furthermore, the hobbyists and enthusiasts in the field today are experienced programmers, and some are even accomplished amateur musicians, but most lack the formal music education that is offered by university or conservatory programs. As a result, there is a noticeable difference in quality and usability between virtual synthesizers developed by hobbyists without musical training, and those developed by professional developers and studios, which typically hire musicians to consult in the development process.

I have experimented with developing virtual synthesizers and audio plug-ins before, and this experimentation has led me to an interest in taking on the current project. As I have an ongoing interest in digital synthesis, one of my goals was to use this project as a means to further explore the discipline and learn more about the conventions and best practices of synthesizer development. Two resources that have proven especially valuable in this project are the books *BasicSynth: Creating a Music Synthesizer in Software* by Daniel R. Mitchell and *The Scientist and Engineer's Guide to Digital Signal Processing* by Steven W. Smith. For the design of the user interface and visual appearance of my synthesizer, I have adhered to some of the principles and recommendations of the Material Design specification, which is freely available online. [1]

Methods

I built this project using the C++ programming language, as it is the industry standard for digital audio plug-ins and applications, and used Microsoft's Visual Studio 2019 Community as a development environment. I have also used the iPlug2 library, [2] a C++ framework that facilitates easier development of audio applications, which is an open source project maintained by Oli Larkin and is freely available on GitHub. iPlug enables me to simultaneously develop my project as a standalone Windows application and a VST plug-in. iPlug also provides a system to handle basic functionality used by all audio plug-ins, such as managing parameters and integrating audio processing with the visual interface. I have also used Git and GitHub to assist in the development and organization of my code. [3]

A wide variety of virtual synthesizer plug-ins exist on the market today, with an equally wide variety of design philosophies and musical uses. Some projects focus on using digital technology to create novel sounds and providing maximum flexibility to experienced users.

[1] Material Design: https://material.io/
[2] iPlug2: https://github.com/iPlug2/iPlug2
[3] GitHub repository: https://github.com/ZachOhara/SeniorProjectSynth

Others aim simply to emulate existing technology in a digital medium, especially technology like analog synthesizers or other vintage instruments. My project aims to digitally re-create some of the functionality and sounds commonly found in analog synthesis, but does not attempt to emulate any specific technology or hardware device. Additionally, I will leverage the flexibility of digital interface design to create an end product that is easily navigable, even by users without previous synthesizer experience, as my design will not be subject to the limitations inherent to physical hardware interfaces. Many virtual synthesizers on the market today come with visual interfaces that are unique and highly stylized, and many attempt to re-create the look and feel of physical synthesizer interfaces. While there is certainly an argument to be made for the marketability of a unique and recognizable style, a cluttered or poorly organized visual interface can be a barrier to entry for new or inexperienced users.

A clean, well-organized, and presentable interface is the primary objective for this project. Controls for the various modules of the synthesizer are organized left-to-right and top-to-bottom in the same order that each module affects the generated sound, so that users may develop a conceptual model that aligns with how each module impacts the tone of the instrument. Additionally, the interface is designed so that the controls for more frequently modified parameters, such as the cutoff frequency of the low-pass filter or the master volume of the instrument, are larger and more visually apparent from a quick glance. This design has the added benefit of visually encoding information about how the application is intended to be used, which will further assist users in learning how to use the synthesizer.

It is difficult to imagine a sound or timbre that has no possible musical use; however, it is also inarguable that some timbres have a wider variety of musical application than others, especially when considering the wide range of possible sounds and timbres that may result from sophisticated or esoteric synthesizer modules. In designing my synthesizer, I aimed to make the range of producible sounds as widely usable as possible, even when considering the diverse number of configurations that may result from different parameters. In other words, my goal is to limit as much as possible the ability of users to create “bad” or unpleasant sounds, or sounds which have limited or niche musical use. I have accomplished this primarily by restricting the possible range of input parameters to values which I believe are the most practical, and by designing interface controls that operate musically rather than numerically. This means parameters may be subject to exponential curves so that a constant adjustment will have a constant influence on the perceived timbre, rather than a constant numerical change. For example, the difference in perceived tone when increasing the cutoff frequency of a low-pass filter from 200 to 400 Hz is much more significant than an increase from 16,000 to 16,200 Hz, so the interface control for this parameter is presented in such a way that the difference between 200 and 400 Hz is visually more significant than the difference between 16,000 and 16,200 Hz, even though each pair of values represents a constant adjustment of 200 Hz.

Secondary to the goal of usability, but still an important design consideration, is the musical flexibility of my synthesizer and the variety of timbres it is capable of producing. While limiting the ability of users to create arcane, esoteric, or otherwise less-usable sounds, I have also aimed to design my application with the flexibility to generate a wide variety of tones, so that it may fit as many distinct musical applications as possible. This is accomplished by providing editable parameters for many aspects of the synthesizer that could meaningfully affect sound generation, and ensuring that parameters cannot conflict with one another. For example, there are a handful of parameters that modify either the vibrato or portamento modules of the synthesizer. Even though both modules affect the pitch of generated tones in distinct ways, I have structured both systems so that they do not interfere with the other. The vibrato module will continue to subtly alter the pitch of the note even while the portamento module is gliding across an interval, thus ensuring that all possible configurations of both modules will work as intended in all cases.

Since my synthesizer will be distributed as a VST plug-in in addition to a standalone Windows application, the conventions of VST instruments will have a considerable impact on the underlying technology driving the project. For example, in order to maintain compatibility with a variety of MIDI controllers, my synthesizer must be able to accept basic MIDI inputs such as note attacks and releases, as well as common auxiliary inputs such as a sustain/damper pedal or a pitch modulation wheel. In addition, the application must be designed so that information about parameters and settings can be sent or received via MIDI protocols, so that a VST host or digital audio workstation (DAW) may be able to read and modify the parameter values of the synthesizer.

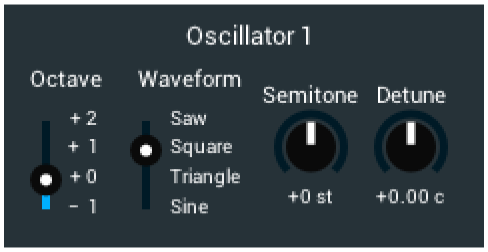
To achieve the visual design goals of this project, I will be following selected elements of the Material Design specification, developed and published by Google for use in mobile and web apps. Because my synthesizer is not a web app, not everything from the specification will apply; however, basic design elements such as the font family and color palette can still be used, as well as principles like the proximity of related controls, or immediate reactions to user inputs. All the text in the interface of my synthesizer is displayed in Roboto, a font family created by Google for use with Material applications, and specifically designed to increase readability of text on digital screens. Additionally, all colors in my interface are taken from the Material Design color palette,[4] and are applied in ways consistent with the philosophy of Material Design. Text is

[4] Material Design color system: https://material.io/design/color/the-color-system.html#color-usage-and-palettes

written in white, the background is a dark gray, a bright saturated blue is used as an accent color, and the black elements of the controls are actually slightly gray in order to better cohere with the background.

Results

The oscillator section of my synthesizer comprises two oscillators, a sub oscillator, and an oscillator mixer. Each oscillator has parameters that control the octave, the waveform, a semitone de-tuner, and a fine-tuning control. The octave parameter can be used to create a pitch in a different octave than the note that was played, and the octave settings of multiple oscillators can be manipulated independently to create more complex sounds. The range of possible values is from -1 to +2, with the default value 0. The waveform parameter is used to select between sine waves, triangle waves, square waves, and sawtooth waves, and just like the octave parameter, can be controlled independently for each oscillator to create a wider variety of tones. The semitone de-tune control is a knob that can be used to raise or lower the pitch of the oscillator by up to 12 semitones (one octave), which can be used by performers to create parallel intervals that follow any melodic line, and the fine-tuning control can be used to change the pitch of the oscillator by up to 100 cents (one semitone). When the tuning controls of multiple oscillators are combined, this can result in sounds that are intentionally dissonant or out of tune.


Oscillator 1

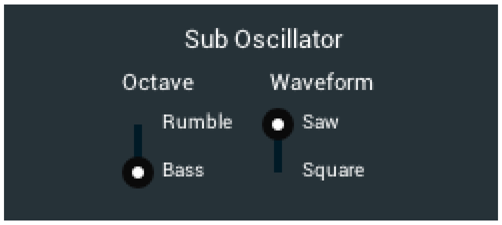
Octave: +0

Waveform: Square

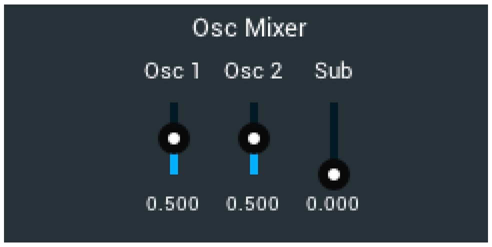
Semitone: +0 st

Detune: +0.00 c

The sub oscillator, also called a bass oscillator, is an oscillator that specializes in producing low frequencies. Like the other oscillators, it also has parameters that control the octave and waveform, but these controls operate somewhat differently. The octave parameter on the sub oscillator does not simply change the sounding octave, but rather controls the blend of two different octaves. The two extremes are labeled bass, which produces a sound one octave below the played note, and rumble, which equally mixes together sounds that are one octave and two octaves below the played note. The waveform parameter of the sub oscillator is limited to only producing square waves and saw waves, because I found in my experimentation that these are the two most useful waveforms to use for low frequencies, since they have the brightest overtones and harmonic content.

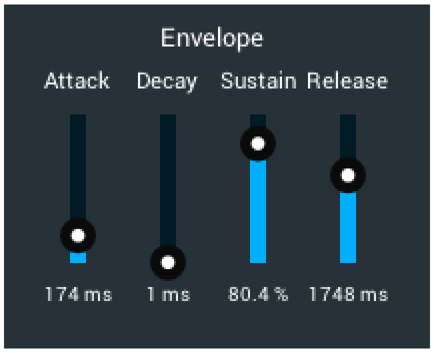


The oscillator mixer features three faders – labeled ‘osc 1,’ ‘osc 2,’ and ‘sub’ – which each correspond to the relative volume produced by the different oscillators. These volumes are represented by numbers between zero and one, with one being the loudest and zero being silent. The default volumes for the two primary oscillators are each 0.5, so they can be adjusted either up or down, while the default volume for the sub oscillator is zero so that it remains silent until explicitly enabled by the performer.



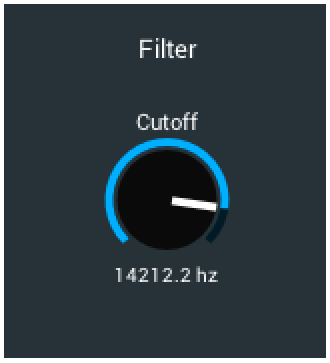
The oscillator section occupies the left half of the synthesizer's interface and the individual modules are organized from top to bottom in order of oscillator one, oscillator two, sub oscillator, and the oscillator mixer. The sub oscillator is placed below the two primary oscillators and occupies less space in both the horizontal and vertical dimensions. This was an intentional decision, as it communicates that the sub oscillator is less important – or rather, more niche – then the two primary oscillators. Additionally, the oscillator mixer is placed at the bottom of the section in accordance with my philosophy that top-to-bottom layout should accurately represent the order of control flow between the modules, and the oscillator mixer only acts upon the signals generated by the three oscillator modules.

The right half of the synthesizer interface begins with the envelope module, which contains parameters for the attack, decay, sustain, and release of every note. Each of these parameters corresponds to a different phase of a note's envelope cycle. The attack phase begins when the note is played by the performer, and a note in the attack period will gradually increase in volume from silence to its peak. This may happen over the course of anywhere between several milliseconds to several seconds. As the note reaches its peak volume, it begins the decay phase, in which it gradually decreases in volume until it reaches its sustain level. Similar to the attack, this may last a few milliseconds to several seconds. Once the sustain volume is reached, the sustain phase lasts until the note is released by the performer. The release phase then gradually decreases the volume of the note until it is silent.



The parameters that control attack, decay, and release all relate to the length of time that each phase should last. These parameters are visually displayed as faders which range from 1 millisecond to 5000 milliseconds (5 seconds). Since one of my design philosophies in this project is to communicate musical meaning in the visual appearance of the synthesizer, these parameters are all displayed on an exponential curve. Since the difference between 1 ms and 100 ms is much more musically significant than the difference between 4900 ms and 5000 ms, the exponential curve enables these parameters to be controlled much more finely at lower values. The sustain parameter does not control a length of time, because the sustain phase will last for as long as the note is held by the performer. Instead, the sustain level is displayed as a percentage that represents a fraction of the peak level. For example, a sustain level of 100% will make the decay phase irrelevant, since the sustain level is the same as the peak level, while a sustain level of 0% will cause a note to become silent at the end of its decay phase. There is no exponential curve applied to the sustain parameter, since all possible values are roughly equal in usability.

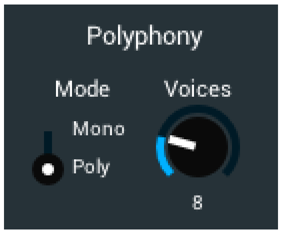
Below the envelope module is the filter module, which contains a low-pass filter, as is ubiquitous in nearly every synthesizer design. The low-pass filter reduces the volume of high frequencies in its input while allowing low frequencies to pass through unchanged. The filter module of my synthesizer contains only one parameter, labeled ‘cutoff,’ which controls the limit of frequencies that are allowed to pass through the filter. This parameter ranges from 15 Hz to 22 KHz, which is only slightly outside the human hearing range of 20 Hz to 20 KHz. This range is intentionally wider than necessary because limitations in the implementation of digital filters mean that frequencies near the cutoff limit may not be properly discriminated. For instance, while a theoretically perfect low-pass filter with a cutoff frequency of 20 Hz will completely block a 23 Hz tone, limited computing power means that any practical filter design would partially allow a 23 Hz tone. Therefore, it makes sense to allow the cutoff frequency to be set as low as 15 Hz so that all audible frequencies may be eliminated if desired. Additionally, the cutoff parameter is also displayed with an exponential curve since fine control of high values is less musically important than fine control of lower values.



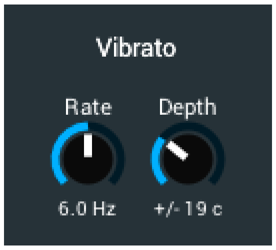
When I began this project, I initially aimed to produce a filter that emulated characteristics of analog low-pass filters, including a resonant frequency peak and separate parameters that could control the amplitude and width of that peak. However, it soon became apparent that the mathematics involved in designing such a filter exceeded the scope of this project and the time in which I had to complete it. Although many other open source projects with implementations of these filters can be found online, I believe that using a pre-existing implementation without fully understanding the mathematics would be against the spirit of my project. While I didn’t have sufficient time to experiment with more advanced filter designs for this synthesizer, this is something I am likely to explore further in the future.

The vertical positioning of the envelope and filter modules, as well as the left-to-right order of the entire interface, are representative of the order in which each module processes the audio signal. In the left half of the interface, the oscillators generate tones which are then combined into one signal by the oscillator mixer. This signal is then processed by the envelope module, and then the filter module removes unwanted high frequencies from the compound result.

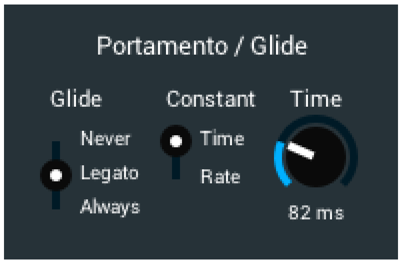
At the bottom of the interface is a group of modules that don’t directly manipulate audio signals, but instead control general settings of the synthesizer. This begins on the left with the polyphony settings. My synthesizer is capable of operating polyphonically with up to 32 notes sounding simultaneously, although this may be voluntarily reduced by a user to any number between 1 and 32. If this is set to 4, for example, then playing a fifth note while already holding a four-note chord will result in one of those notes ending while the fifth begins. The released note will be the first note that was played, as this design always attempts to preserve the most recently played pitches. The number of available voices may be arbitrarily limited based on available processing power – since more simultaneous voices will put a greater strain on a computer’s processor, potentially leading to undesirable artifacts or temporary silences if the processor is unable to keep up – or may be set to achieve a specific musical result. Since most early analog synthesizers were only able to produce one note at a time, modern polyphonic synthesizers typically include an option to switch to ‘monophonic mode,’ which seeks to emulate the early synthesizers by only producing one note at a time. In this setting, holding one note while playing a second will result in the first being released and replaced by the second. In my interface, I decided to include both a knob that controls the number of available voices, and a switch that alternates between monophonic and polyphonic modes; however, this switch is provided only for convenience, as switching to monophonic mode is functionally the same thing as setting the number of available voices to one.



To the right of polyphony settings is the vibrato module, which can be used to add vibrato simultaneously to every sounding voice. There are two parameters in this module: one which controls the vibrato rate, and another which controls the vibrato depth. The rate parameter controls the speed of the vibrato solution and is measured in hertz, ranging from 0 Hz to 12 Hz, while the depth parameter controls the pitch modulation away from the center in both directions, ranging from 0 (no vibrato) to 200 cents (two semitones). For example, if the vibrato module is set to a 2 Hz rate and 100 cent depth, every note will oscillate between one semitone and above and one below, and a full oscillation will occur twice every second.



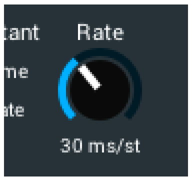
In the bottom right corner of the interface is the portamento module, which can be used to add a glissando or ‘glide’ effect between notes. The parameter that controls the mode of this module, labeled ‘glide,’ has three possible options: never, legato, and always. The ‘never’ option is self-explanatory: notes will not glide in any circumstance. The ‘always’ option is also described simply: whenever a note is played, there will be a brief glissando effect that begins at the frequency of the most recently played note and ends at the frequency of the current note. When the mode setting is set to ‘always,’ the portamento will be applied regardless of how much time has passed in between these notes, and whether or not the last played note is still being held. The ‘legato’ mode option lies between these two extremes: portamento will be applied for notes that are struck while another is still being held, but a note played without any immediate predecessor will not have any portamento. The idea for these varied portamento modes came from the open source VST synthesizer Helm[5], in which the same three functional modes are instead called ‘off,’ ‘auto,’ and ‘on.’ I decided to rename these options to ‘never,’ ‘legato,’ and ‘always’ respectively as these terms provide additional clarity to the user as to how each mode behaves.

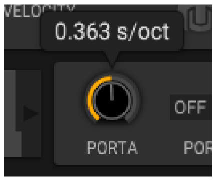


The portamento module also feature settings that control the time or rate of the glissando. A parameter labeled ‘constant’ allows the performer to choose between a glissando effect that takes place over a predetermined amount of time, measured in milliseconds, or an effect that changes pitch with a predetermined rate, measured in milliseconds per semitone. While most

[5] Helm: https://tytel.org/helm/

synthesizers with portamento support allow for a constant time to be set, the aforementioned Helm project features a portamento setting with a constant rate, measured in seconds per octave. I have decided to include both options, as I believe there are musical uses for both types of portamento. I decided to change the rate unit from seconds per octave to milliseconds per semitone, since I believe a rate measured in milliseconds per semitone is easier to understand and more meaningful to the performer. Additionally, most useful portamento rates appear as a fraction of a second when measured in seconds per octave, but can be represented as an integer number of milliseconds per semitone. I believe that an integer number of milliseconds is easier to read and understand then an arbitrary fraction of a second, which further supports the decision to use this unit in my synthesizer.

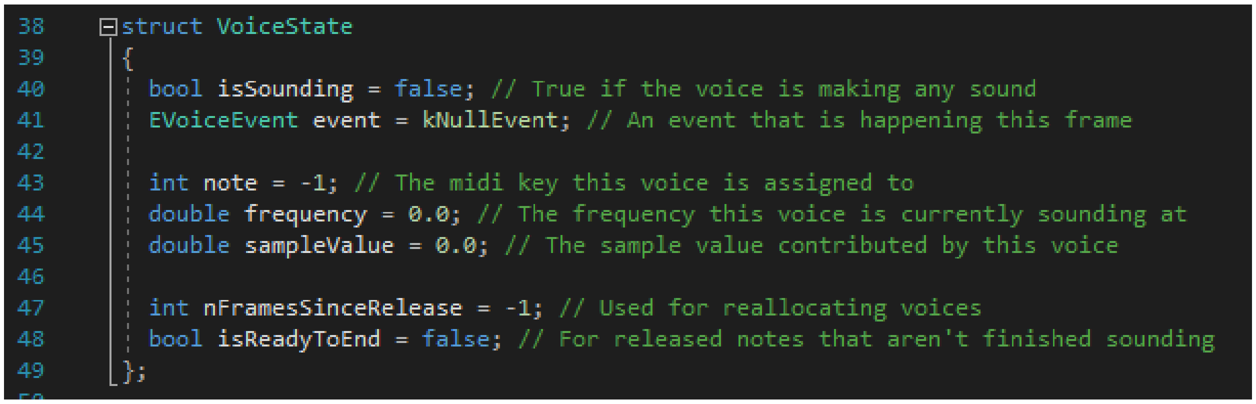




Discussion

Arguably the single most important decision I made in this project is the way I constructed the MIDI input processing system, and how I structured the code that allows the various audio modules to communicate and transmit information within the system. While many hobby projects similar to my synthesizer are limited to only one voice, since a monophonic synthesizer is far easier to build than a polyphonic instrument, I knew early on in the planning phase of this project that I wanted my synthesizer to be capable of polyphony. In a different synthesizer project that I built previously as a way to experiment with the technology, I structured the code so that the state of all 128 possible MIDI pitches are continuously tracked. While this provided the advantage of theoretically allowing for all 128 pitches to sound simultaneously, the inefficiency of my code meant that only around 16 pitches could be sustained before the processor was unable to keep up with its demand, and audio glitches and artifacts came as a result. Additionally, this design was inflexible in that it only allowed for either 1 or 128 voices to be active at once. There was no way to implement an intermediate setting, such as two or four active voices.

In order to combat some of these issues while still supporting polyphony, I constructed a system that keeps track of 32 voices, and each voice may be assigned to any arbitrary MIDI pitch or frequency. There is a dedicated system within my code that is responsible for the management of these voices, and the allocation of MIDI pitches within the set. The screenshot below shows the code that manages information for each voice.



```
struct VoiceState
{
    bool isSounding = false; // True if the voice is making any sound
    EVoiceEvent event = kNullEvent; // An event that is happening this frame

    int note = -1; // The midi key this voice is assigned to
    double frequency = 0.0; // The frequency this voice is currently sounding at
    double sampleValue = 0.0; // The sample value contributed by this voice

    int nFramesSinceRelease = -1; // Used for reallocating voices
    bool isReadyToEnd = false; // For released notes that aren't finished sounding
};
```

The first variable here, called ‘isSounding’, is a Boolean value (either true or false) that marked whether or not the voice is currently producing a sound. This increases the computational efficiency of the entire synthesizer, as each module can check if a voice is sounding before spending any time on processing data for silent voices. The next variable, named ‘event’ contains information on anything that is happened with the voice in the current frame, such as a key press, a key release, or the end of the note’s duration.

The next section of code in this screenshot contains the most important information about the note’s current state. Since each module processes the voices in a predetermined order, these variables are also the primary platform which the modules may use to communicate with each other. The ‘note’ variable is set by the voice management system after a note has been played, and this value represents the MIDI pitch (an integer between 0 and 127) of the note. This data is then read by the tuning system, which calculates the sounding frequency of the given MIDI pitch and store this information in the ‘frequency’ variable. The frequency can then be further modified by the portamento module, and then by the vibrato module.

Once the frequency is determined by all the relevant systems, this information is read by the oscillators and used to calculate the actual value of the current sample for the voice. Each of the three oscillators in the synthesizer calculates their own sample value for each voice depending on the relative settings of each oscillator, and these values are combined by the oscillator mixer to produce the final sample value. After the oscillators have been combined for each voice, the sample value is then further modified by the envelope module and attenuated by the filter module.

The design of this system and the sequential processing of all voices by each module represents the fundamental backbone of the technical construction and design of this project. This process is orchestrated by the voice management system, which is responsible for processing real-time MIDI inputs and allocating played notes to the available voices. The management system tracks which notes are played and in what order, as well as the current setting for the number of active voices in the system. For example, if the current configuration is set to four active voices, and a fifth note is played while four are already sounding, the voice manager will determine which of the four sounding pitches is oldest, and reallocate its voice to be used for the new pitch. Additionally, if the new note is released while the original four are still being held, the voice manager will reallocate the appropriate voice and re-attack the original note. In computer science terminology, the currently sounding voices represent a first-in first-out system, also called a queue, while the reallocation of voices to new pitches represents a first-in last-out system, also called a stack.

Another priority in structuring the technical design for this project is that I wanted the code for each module to be reusable in future projects. As previously mentioned, this project is not my first attempt at designing a virtual synthesizer, and I don’t believe it will be my last. In order for each audio module to be reusable in future projects with limited modification or redesign, the code has to be as agnostic as possible about the surrounding system. This is also supported by the voice management system, since each module sends and receives audio processing information only through the voice system, and not from direct inter-module communication. For one example, each oscillator only receives input about a note’s frequency from the frequency variable in the voice structure. The oscillator has no need to know about or interact with the various modules that affect frequency, such as the tuning system, the portamento module, or the vibrato module. As long as the frequency of the voice is correctly set by the time it is processed by the oscillator for a given frame, the oscillator has all the information needed to produce a tone. The result of this is that the oscillator code can now be reused in a brand-new unrelated project, as long as the voice management system in such a project operates in the same structure. The new synthesizer would not need to support vibrato or portamento for the oscillator to still function correctly. Alternatively, a new project may include additional modules in the chain that affect note frequency, such as additional fine-tuning controls or temperament settings, and this would not adversely affect the oscillator. Since each module in the system is designed to function completely independently and communicate only through the voice system, the same thing could be said about any module in this project.

Conclusion

With the completion of this project, I have designed and built a virtual synthesizer with a variety of basic and relatively commonplace features. Moving forward, although this synthesizer could be used as a functional and flexible musical instrument in its current state, I plan to use this project as a platform to explore more advanced synthesizer features and further my understanding of musical synthesis and digital signal processing. Some specific ideas that have come forward over the course of this project include a low-pass filter design that seeks to emulate characteristics of an analog filter constructed with physical circuit, or a more sophisticated portamento system that can better handle the intricacies of modern polyphonic music and can understand musical constructs such as voice leading or chord voicing. Additionally, I'd like to explore technology that expands the flexibility of tuning beyond the system of 12-tone equal temperament that is commonplace today. Such technology may include support for other tuning systems, such as well-tempered or mean-tone systems, or the ability to recognize and understand harmony in real-time, and adjust the tuning of individual notes to better fit the chord as it is being played.

Additionally, I have publicly released all of the code I wrote for this project on GitHub, a website dedicated to the hosting and sharing of open source software projects. There, I hope it may be used by other hobbyists and programmers to further their own understanding of digital synthesis, and to improve the technology and capabilities of other similar projects. A core philosophy of open source software is that public availability and large-scale collaboration is a significant force in improving the state of the art, and I have provided my code as free and open source in pursuit of this principle. Although this particular project has concluded, I have learned a large number of valuable lessons, and I intend to use this experience to continue developing new virtual instruments and audio plug-ins in the future.

Bibliography

Google Inc. “Material Design.” https://material.io/

Larkin, Oli. iPlug2. https://github.com/iPlug2/iPlug2

Mitchell, Daniel R. “BasicSynth: Creating a Music Synthesizer in Software.” 2009.

Smith, Steven W. "The Scientist and Engineer’s Guide to Digital Signal Processing." 1999.

Tytel, Matthew. Helm. https://tytel.org/helm/