Chloé Reddy
MUMT 303, New Media Production 2
Professor Yaolong Ju and Professor Travis West
January 20, 2020

MAX/MSP "ROCKBAND"

Intro

In this paper, I am going to be breaking down the process I went through to create the MAX/MSP "Rock Band" patch. It consists of a midi "band" that act as a backing track and a generated guitar solo, done entirely in MAX using standard MAX objects. I will first introduce my motivation and research done to prepare for the project, as well as explain some struggles I went through during the process and the decisions I made in in order to resolve them.

Motivation

For this project, I knew I wanted to create a program that was both interactive and generative. I came up with the idea to do a computer-generated melody and wanted to find a way I can implement that in a simple and fun program. Thus came the MAX/MSP Rock Band. I wanted to give the user parameters that they could manipulate in order to affect the sound of the guitar and create a backing track played by a midi-band. In the end, I decided to give the user the ability to control the key, tempo, and harmony of the backing track. They would also be able to manipulate the guitar sound and add typical guitar-amp effects to distort or change the sound. My ultimate goal was to create an easy-to-use and entertaining patch with an element of randomness to have a unique melody each play-through.

Random Melody Generation

Before starting this project, I didn't know much about generative music. I knew I was going to have to do some research in order to get a better sense of how I wanted to create this melody. The first technique I researched was using Markov Chain's, which calculates the probability of a certain note (in the case of melody generation) following another note. This is done by having the algorithm analyze many different melodies and finding patterns between pairs of notes then pick a chain of notes based on these patterns. After more research on Markov Chains, it seemed that this method would be extremely difficult to implement in the time given and would require the analyzation of hundreds of guitar solos in order to create something similar, as well as knowledge of natural language processing that I do not have.

Another approach I researched was using Grammars, a technique usually applied to natural languages that could be applied to music as well. It involves formalizing rules, similar to those in grammar that define which notes and rhythms can follow one another. If there are multiple right answers, one is chosen at random. For example, one could implement a rule that an eighth note must be followed by another eighth note or an eighth rest, and the computer would choose one. The system then generates full “sentences” from these rules. Again, in the interest of time, I figured it would be worth my while to implement a technique a little more restrictive than this one.

The last technique I researched was from a paper by Peter Langston, *Six Techniques for Algorithmic Music Composition,* in which he defines six different algorithms for generating rhythms and melodies. I was familiar with his paper and remembered one algorithm he created called Riffology, where a program selectively picks from a hard-coded collection of “riffs” consisting of midi note values.. The problem with this algorithm, in relation to my project, was that it didn’t allow for rhythmic variance and for it to follow a chord progression would be to limit it even more.

I drew my inspiration mostly from Langston’s Riffology to generate the melody, but mostly tried to come up with an algorithm on my own. I wanted to generate the two natural parts of the melody- rhythm and notes separately and combine them at the end to create each note. The first issue I ran into was ensuring that the right notes are being played over the right chords. For the sake of simplicity, I chose to construct the melody out of only chord tones, so as to not have to deal with passing tones and other non-chord tones. I also wanted to include a form of rhythmic variance to add complexity to the melody. For each measure, I would have to keep track of how many notes have already been played as well as their duration, and change chords when a measure was complete. I found the easiest way to achieve this was to create a MAX `coll` object (called `Rhythms` in the patch) that contained 30 hardcoded rhythms, randomly choose a rhythm for each measure and assign each rhythm value a note. Then, the note is sent to the guitar-sound synthesizer and the note is played.

Approach

When creating this patch, I wanted to start with where I was most comfortable and what I found the easiest: the backing track. The backing track (or the "band") is made up of midi drums, a midi bass, and a midi piano. I am not a drummer and have difficulty creating different beats, so I stuck with the most basic rock beat. The drums are made up of a kick, snare, and hi-hat. The `coll` object named `Beat` has four entries, each representing a separate beat. A `1` means that a drum sound plays, while a `0` indicates silence. Then, messages containing the midi message for each drum sound are sent to a `noteout` object.

I implemented the piano and bass in a similar way. First, I came up with four common pop/rock chord progressions- two major and two minor. Then, as a user picks a chord progression, a `switch` object sends one of four different `coll` objects (one for each progression) which contain the midi note values for each note of the chord. The bass plays just the root note, while the piano has a two octave root-position chords made up of six notes. These notes are combined with the program, channel (bass is one, piano is two), velocity, and note duration (calculated from the tempo) in a `midiformat` object and sent to a `midiout` object. Each chord lasts one measure. To get the tempo, the user uses a slider to set their desired bpm, which is then converted into milliseconds. Users can also pick a key, which transposes all of the midi note values by adding or subtracting the difference between C3 and the chosen key in semitones. To play the backing track, there is a `toggle` object that triggers a `metro` that bangs once for each beat. `Sel` objects output the corresponding `coll` entries for each beat or measure.

The next aspect I worked on was synthesizing the guitar sound. I started with the Karplus Strong algorithm that was introduced in class. This algorithm generates a plucked string sound with a filtered delay line and a high order digital filter. To change notes, the delay time is changed. This was done by taking 1/frequency to obtain the period, which is equal to the delay time. When I used the `mtof` object in MAX to convert midi note values in to frequencies, I found that it was always a whole tone off. I added 14 to the midi value to get the correct note and then raise the octave. I also used a `retune~` object to make sure the note was in tune with the "band", since sometimes it was a few cents off.

The algorithm used in class mimicked an acoustic guitar sound, but I wanted the user to be able to manipulate this sound to their liking. So, taking inspiration from guitar amps, I implemented effects that the user could add to the sound, the first being distortion. For this, I used the `degrade` object and changed the bit depth of the sound from 24 to 2 bits. The user can turn on the distortion using a `toggle` object. To add reverb, I used the `yafr` (yet another free reverb) object and used a `slider` to control the reverb time. If a user wants a less intense sound, they could add delay to the guitar using a `slider` to control the delay time of a simple one-tap delay with feedback (similar to the one shown in class). Finally, there are two filters added to boost the bass and treble, using `lowshelf` and `highself` filters, respectively, sent to separate `filtergraph~` objects. I hardcoded the frequencies boosted for each one, so a user can only turn on and turn off each frequency boost using a `toggle` object. These values are then sent to a `biquad~` object. The signal synthesized from the Karplus-Strong algorithm is then sent to one effect and the affected signal is sent to another.

The bulk of the work in the project was done to create the randomly generated guitar solo algorithm. After my research, it became clear that I was going to have to keep the algorithm simple if I wanted it to be functional. I knew I wanted to have an aspect of rhythmic variance but if I let the rhythmic values be completely randomized, as in the computer chooses different note durations, I'd have to keep track of when each measure had enough notes to fill it, and change the notes accordingly. My solution was to hardcode thirty different rhythms representing all the possible combinations of whole, half, quarter, and eighth notes. A `1` represented an eighth note, since it is the smallest unit of rhythm used, and `2`, `4`, and `8` represent quarter, half, and whole notes, respectively. I avoided using rests, otherwise the combinations of rhythms becomes exponentially bigger as more variance is added. My first thought was to add the number representing the rhythm to the `Rhythm coll` object, for example, four quarter notes would be `#, 2 2 2 2;` (with # representing the entry number). But, different rhythms had different number of values, which made it difficult to later separate the list of values into different messages that could be interpreted. The fix was to fill in the space between values with 0's. For example, the four quarter notes now become `#, 2 0 2 0 2 0 2 0;`. Now, each entry has 8 values, since the maximum number of notes a measure would have is eight eighth notes. This made it easier to have a consistent method for later analyzing these rhythms. From there, I used a `random` object to generate a random number from 1 to 30 to select one of the rhythms.

The next step was to generate a note for each rhythm value. As stated before, I wanted to keep the melody simple to ensure that it would not be dissonant or sound muddled. Each note was chosen from the same `coll` objects used for the piano melody transposed up an octave. Using a similar process as the backing track, the `coll` associated with the chosen chord progression is selected using a `switch` object. Then, a random number from 1 to 6 is generated and selects a corresponding note. This note is sent to a `p patcher` called `melodyGenerator`, where the rhythm and the note are combined. For every measure of eight rhythm values, a note is selected for each one, skipping the 0's. Every eight values, a new chord is chosen and new values are assigned. The rhythms are converted from integer values into fractions of a measure, so a quarter note goes from `2` to .25. Individual note and rhythm values are joined and added to a `coll` called `Melody`, where entries are as follows: `#, noteValue rhythmValue.`

A finite number of notes and measures are generated, depending on the tempo, with a maximum duration of the melody as three minutes, inspired by the parameters from prompt one of this project. The bpm is multiplied by four and divided by three to calculate the number of measures necessary. Integer division is used to avoid fractions. After the `melodyGenerator` patch has iterated through this many measures, the generation halts. A `slider` acts as a loading bar to show the progress.

Finally, the melody can be played. The guitar synthesizer receives midi note values from `coll Melody` as well as note values. The `p patcher makeNotes` takes these note values and sends it to the synthesizer and has a `metro` object that bangs after every note is complete, depending on the given note length. To hear the "rock band in full", the user can press a `toggle` object after experimenting with the effects to get the desired sound. They can also reset all the values and generate a new melody from scratch.

Interface

Whenever I approach the task of making an interface, I always aim to create something that is understandable and requires little outside knowledge to use. Minus a basic music background, the information to use the patch should be provided within it and it should be simple to use. Otherwise, it tends to be inaccessible to users. Luckily, MAX has a lot of GUI objects that make this a possibility. I also included a step-by-step "How It Works" section to give the user instructions on which order they should do things.

First, the user should select a tempo, a chord progression, and a key. The tempo is chosen using a `slider` and the chord progression and key are chosen using two `umenus`. Then, there is a `toggle` to click to generate the melody. I included the `coll Melody` object in the interface incase the user was curious and wanted to see what the melody looks like. This is not the best software design practice since now the user has access to the text file and can edit it, but MAX does not have a great system of information hiding in patches anyways, so I decided to leave it and if the user wants to observe it, they can. While the melody is being loaded and the loading icon advances (no more than 30 seconds), the user can take this time to experiment with the guitar effects. Using a `kslider`, they can play different notes and try to create their ideal guitar sound. The distortion and bass and treble boosts are activated using `toggles`. The gain, reverb, and delay can all be changed using `dials`, similar to how they are implemented on a physical guitar amp. Once the melody has been generated, they can start the melody and hear it play. There is also a reset `button` that will reset all the parameters and they can start over.

Limitations/Possible Expansions

The majority of the limitations that exist in this program were a product of time. That being said, there are many possible expansions that I can see the project having in the future. For example, it would be very easy to add more effects to the guitar sound in the forms of more filters, chorus, or a more complex distortion. More freedom can be added by allowing the user to control the frequency range of the filters, something that is not common on a simple guitar amp. A more difficult change to make would be to manipulate the implementation of the Karplus-Strong in order to synthesize a more authentic guitar sound.

Most of my limitations lie in the melody generation. Throughout the process, I had to decide if I wanted to focus more on rhythmic variance or melodic variance. I leaned towards more rhythmic variance, since I knew that I could make a consonant melody with chord tones. But, it would be interesting to add more rules to the algorithm in order to have a more complex melody. For example, the algorithm could be limited in the amount of semitones in can move upwards and downwards. Passing tones can be added on upbeats to help transition between them. And extensions, such as sevenths and ninths can be considered. I also did not take into account the playability of the melody. If a guitar player were to attempt this melody, it could prove to be quite awkward and difficult due to the random nature of it. Taking into account the note distribution on a guitar and building rules around them would lead to a more realistic melody.

The piece generated by the algorithm has little to no structure. When a user toggles the melody, it plays for up to three minutes, and then it ends. The backing track plays the same chords the entire time. It would be possible to create a sort of verse-chorus structure with an intro and outro to mimic a pop/rock song. To expand this even further, a user could pick four different chords for each section. When more choice is given to the user, it then requires an algorithm with stronger rules in order to generate a pleasant-sounding melody. For example, if a user picks four random chords that generally aren't played in that order, for example I ii vii° vii, then it becomes much more difficult to come up with rules that account for these cases. The programmer then has to decide if they want to create rules for chord progressions as a whole or just rules to lead from one chord to the next. The more complexity added in one section (rhythm versus melody), the more work that has to be done in the other in order to ensure the melody is still coherent.

Conclusion

Despite the limitations that exist in the program, it creates a coherent and consonant melody and allows the user to have some control over parameters to affect the outcome, which achieved my original goals. Throughout this project, I also learned about different techniques for melodic generation and explored those options in MAX. The melody I came up with was a bit more trivial than I originally intended, so if I were to continue to pursue this project, using the ideas mentioned above, there is the possibility of creating an algorithm that is even more powerful and realistic.

Bibliography

"How Generative Music Works." *How Generative Music Works*, teropa.info/loop/#/title.

Jaffe, David A., and Julius O. Smith. "Extension of the Karplus-Strong Algorithm." *Computer Music Journal*, vol. 7, no. 2, 1983, pp. 56–68.

Langston, Peter. *Six Techniques for Algorithmic Music Composition*. International Computer Music Conference, 1989.

Powell, Victor, and Lewis Lehe. "Markov Chains." *Setosa*, setosa.io/ev/markov-chains/