15-323/15-623 Spring 2018

Project 2. Scheduling, MIDI, Graphics

Due Feb 15

1 Overview

In this project, you will build a real-time program that outputs music using MIDI. The program will implement music generation and image generation algorithms of your design.

The main goal of the project is to create something using real-time scheduling techniques learned in class. A secondary goal is to use wxSerpent to create a graphical user interface (although in this project, the interface requirements are minimal).

2 A Generative Music Algorithm

Write a program that generates music according to some algorithm. The overall program should show some effort and strategies, but need not be a monumental effort. A trivial loop to play an 8-note sequence is too simple, but a program that harmonizes melodies in the style of Bach is probably too complex.

- The program should create multiple "voices," that is, more than one note will play at a time. (DEFINITIONS: A "voice" can refer to on singer, i.e. literally a voice, so a piece for 4 voices means 4 singers, usually singing in harmony. But a choir can sing the same music by having multiple singers singing each of the four "voices." Here, "voice" becomes a little ambiguous, as you could say there are 3 voices on each part, or you could say this is a 4-voice piece for chorus. In any case, the term "voice" also applies to instrumental music: "4 voices" can mean 4 independent parts for different instruments.)
- The program should use a scheduler and precise timing
- The program should have at least a tempo control.
- The program should not schedule more than one measure ahead of real time to permit interactive control (even if the program is not interactive, you should design for that as a future possibility).
- The program should generate images along with music. There should be some correspondence between image elements or their animation and the notes being generated and played.

You are encouraged to design your own algorithms, but here are some examples (and you may implement any of these directly if you wish):

2.1 High and Low, Fast and Slow

Make two "voices" that we'll call *soprano* and *bass*. The soprano plays random pitches in the (high) octave from C4 to C5 while the bass plays random (low) pitches from C2 to C3. The soprano plays (fast) eighth notes (half a beat each) while the bass plays (slow) half notes (2 beats each).

In addition, you can constrain the voices to different scales. Some candidates are:

- Major: 0 2 4 5 7 9 11

- Minor: 0 2 3 5 7 8 10
- Harmonic Minor: 0 2 3 5 7 8 11
- Pentatonic: 0 2 5 7 9

Where the numbers are half steps (midi key numbers) mod 12. I.e. A4 = MIDI key number 69 is in the Major scale because 69 mod 12 = 9, which is one of the numbers in the scale, but A4 is not in the Minor scale. Any of these scales can be *transposed* (i.e. mathematically "rotated") by adding an offset mod 12, e.g. if you take the Major scale and add 3 mod 12, you get 3 5 7 8 10 0 2, which tells you the pitches (mod 12) that are in E-flat Major.

You can also make small intervals more likely than large ones, i.e. pick pitches that are near the previous pitch with a higher probability, especially in the soprano (high) "voice."

Adding some controls to change scales (Major, Minor, etc.) or keys (transposition amounts) will allow you to manually control some changes. You could also pick new scales or transpositions randomly, but do it rarely enough that you get the sense of being in one scale or key before changing to another.

2.2 Chord Arpeggiator

Pick a chord. Candidates are

- Major: 0 4 7
- Major 7th: 0 4 7 11
- Minor: 0 3 7
- Dominant: 0 4 7 10

(Other chords are possible; you are not limited to these.)

As with scales, you can "rotate" or transpose these by adding an offset. You can also extend the chords by repeating them higher or lower by 12 (an octave up or down). E.g. a simple Major triad starting on C4 (MIDI 60) would be 60+0, 60+4, 60+7, or 60, 64, 67. But you can extend this to the next octave by adding 12 to each pitch to get: 60, 64, 67, 72, 76, 79, or 3 octaves (one lower one higher) to get: 48, 52, 55, 60, 64, 67, 72, 76, 79. You can transpose these chords by any number from 0 through 11 to get different chords, e.g. if you add 2 you get notes from the D Major chord: 50, 54, 57, 62, 66, 69, 74, 78, 81.

Now, an arpeggiator just plays the notes in sequence up, down, or up and down, e.g. for just one octave you could play 60, 64, 67, 60, 64, 67, 60, 64, 67, etc. Or you could go up and down: 60, 64, 67, 64, 60, 64, 67, 64, 60, 64, 67, 64, etc. You can also repeat either or both of the top and bottom notes, sometimes to get the pattern to come out to an even multiple of 2 or 4 notes: 60, 64, 67, 67, 64, 60, 60, 64, 67, 67, 64, 60, 60, 64, 67, 67, 64, 60, etc.

Since a basic requirement (above) is "The program should create multiple 'voices,' that is, more than one note will play at a time," you can (perhaps as an option) double octaves ("doubling octaves" means when you play a note at pitch P, you play another note at the same time at pitch P+12 or P-12, and for that matter, P-24 is a nice effect, or try P+19, which is a striking effect). Alternatively, you can have two different arpeggios running at once; a conventional-sounding combination might be a slow arpeggio in a lower octave and a faster one (e.g. 2x speed) in an upper octave (but don't feel limited to making things sound conventional!).

As with "High and Low, Fast and Slow" (above), adding some controls to change scales (Major, Minor, etc.) or keys (transposition amounts), and perhaps other parameters, will make allow you to manually control some changes. You could also pick new scales or transpositions randomly, but do it rarely enough that you get the sense of being in one scale or key before changing to another.

2.3 Riff-o-matic

The basic idea here is to write a short melodic "riff" and repeat it. To write a riff, at minimum you need to create a sequence of pitches. They could be random, perhaps constrained to an octave range (12 MIDI note numbers, e.g. from 60 to 72), could be constrained to a scale (see above), could be weighted toward small intervals, etc. You'll need to pick a duration. "Riff" implies short, so something in the range of 2 to 4 beats is about right.

The rhythm in your riff can be simple, e.g. all eighth notes (0.5 beats), or complex, e.g. pick random but short durations of 0.25, 0.5, 0.75, or 1. However, if you do that, you'll find the riff might sound rhythmically confusing because there are few notes landing on downbeats. An alternative to that is to assemble the riff from some beat-long rhythmic units, which might include: [0], [0, 0.5], [0, 0.25, 0.75], [0, 0.5, 0.75], [0, 0.25, 0.5, 0.75]. Each number in these arrays represents a note to be played at the given offset from the beat (beat times are assumed to be integers 0, 1, 2, … , but you might scale everything to achieve a different tempo rather than have exactly 1 beat per second). Notice that every array includes zero, so this *always* plays a note on the beat. Of course, these can be weighted to be more or less probable, other rhythms can be added, durations can be greater than one beat, etc.

Once you've created a riff, let's assume 4/4 time (4 beats per measure) and play the riff on a measure downbeat (beat *mod* 4 == 0). Then, 1, 2, or 4 measures later, play the riff again. Maybe play the riff as much as 4 to 8 times (experiment with this). Then make another riff and play it for a while. Once you've created the expectation that riffs always repeat, you can break the rule (sometimes). You could also modify or mutate riffs rather than computing each one from scratch.

This all gives you a large space of decisions and control. You can make decisions manually through graphical controls or keyboard input, or you can make decisions with some kind of randomized planning process.

Once again, a basic requirement (above) is "The program should create multiple 'voices,' that is, more than one note will play at a time," and as with arpeggios, you can (perhaps as an option) double octaves (see arpeggios for details), generate and play 2 riffs at once, or satisfy this requirement in some more creative way.

2.4 Other Ideas

You do not have to be an experienced musician to take this class, so we've provided several fairly detailed project directions in the previous subsections. However, we encourage you to develop your own ideas if you want to try something different. Maybe you have some compositional experience and want to implement some intuitions about how to generate your style of music. That's fine. The only pitfall you should avoid is trying to recreate some existing composition: e.g. it's fine if you love Beethoven's "Moonlight Sonata," but don't try to force fit your desire to have a machine perform your favorite music into an assignment where the goal is to generate new music. (Maybe we can find an outlet in another project for you if you desire to work with human-composed music sequences – let the staff know.)

3 Image Generation

The image generation part of this project is also yours to design. Unlike MIDI, where you turn a note on, creating a "sound object" that adds to existing "sound objects," with graphics in wxSerpent, you write code to draw on a *canvas* on *every frame*. You do not create, say, a circle "object" that persists, but instead redraw the circle at every frame. It is up to you whether you create a list of "objects" to draw (e.g. you could have an array of shapes and colors that you update from time to time and draw by iterating through the shapes in the array) *or* just draw procedurally (e.g. you could have a single *paint* method that has code to draw each shape, probably with conditionals to control what or how many to draw, and parameters to control details).

Keep in mind that drawing on a canvas is a somewhat indirect process. See "Interactive Drawing" in `serpent/doc/serpent-by-example.htm` and `serpent/doc/src/drawing.srp` for some sample code that uses a canvas. Also, `serpent/wxslib/wxs_test.srp` has a canvas and shows lots of other wxSerpent capabilities. We will also provide a minimal program to draw on a canvas with this project.

To draw on a canvas, you need to:

- Subclass `Canvas`
- Define (at least) a `paint(x)` method in the subclass to override the inherited `paint` method, which does nothing.
- Call drawing primitives inside `paint(x)`, e.g. `draw_ellipse(x, y, w, h)` will draw an ellipse. See `serpent/doc/wxserpent.htm` for descriptions of all the drawing primitives.
- Make an instance of your `canvas` subclass. Let's call it `p2canvas`.
- When you want to (re)draw the canvas, call `p2canvas.refresh(t)`. This can be called anywhere and anytime in the program. At some *later* point in time (but as soon as the control returns from the Serpent interpreter to wxWidgets), wxWidgets will set up a drawing context for the canvas and invoke the `paint(x)` method, expecting you to call drawing primitives.

One possible direction for image generation is "string art," especially with dynamically changing parameters: c.f. https://courses.ideate.cmu.edu/15-104/f2017/week-4-due-sep-23/ - stringart but please use your imagination and have fun – we are not grading this as fine art.

4 Implementation

It should be obvious in this class that you should use a single scheduler with precise timing.

You should run the scheduler by polling for work roughly every 2 milliseconds. This should be done within the same thread handling the graphical interface. See the example code `p2canvas.srp`.

Your program should be written for `wxserpent64`. You do not need an elaborate interface, but you should have start and stop buttons and a spin control or slider for tempo, plus a canvas for graphical output.

Use `programs/mididevice.srp` to provide menu-based selection of the output device.

Be sure to call `sched_select(rtsched)` in the appropriate places. If you are not processing a scheduled event, for example you might be processing a callback from a button or slider, then the scheduler has no notion of the current time (either virtual or real), so you must call `sched_select(rtsched)` to establish a context in which the scheduler can schedule something – otherwise, timing anomalies might occur.

5 15-623 and Extra Features

15-623 students, in addition to the above requirements, should

- Incorporate their drum machine sequencer from Project 1 into this project (or rewrite it to provide that functionality).
- At least have a mode in which all music generation is automatic. We suggest building some manual controls anyway for testing. When you get something you can control manually, then play with it and get a sense of what control changes, how often, how much, etc. seem to make for interesting music. Then design some random-choice algorithms that will, from time-to-time automatically make a parameter change. One way to do this is to simply manipulate sliders and buttons (that you have created for manual control) via algorithms. If you do this, you'll see what your algorithm is doing because controls will be updated on the screen. See the method (which applies to Slider, Button, and other classes) `set_and_act(x)`, which is probably the best way to do this.

15-323 *and* 15623 students should feel free to add additional features. The graders can award up to 10% extra credit for interesting extra work in the area of music control, scheduling, and algorithmic composition. (Since this is not a graphical user interface class and interfaces can become a large implementation effort, we will not award extra points for elaborate or extra-beautiful interfaces, so keep it simple and functional.)

6 Grading Criteria

1. Correctness: Does the program compile, run, and produce the desired behavior?
    a. Program should generate some coherent music as intended (or incoherent for that matter, but it should correctly implement your intention.)
    b. Program should generate some kind of visual animation that is coordinated with the sounds that it makes.
2. Modularity: Your program should isolate different concerns. In particular, you should clearly separate MIDI I/O, the Graphical Interface, data input (if any), and the implementation of application-specific real-time behavior (sequencer, drumming, or composition).
3. Programming style:
    a. Tabs are forbidden: they are invisible and cause obscure bugs. Fix your editor to insert 4 spaces when you type the tab key. (We highly recommend this for all programming languages, but Serpent and Python especially. See `serpent/extras` for some editor configuration files.)
    b. Code should be clearly written and commented to optimize readability (include high-level design and specifications at the top of the file or in a separate document, give concise comments within the code, avoid verbose

or redundant inline comments.) A 2-page instruction manual in Latex is overkill. A program with 10 lines of comments needs work. Low-level comments on every variable and method is not a substitute for quality documentation.

4. Effort: As stated above, keep it simple, but if your program is not in line with the examples we described and/or if the graphics is merely a lame green oval that moves along the diagonal just like the `p2canvas.srp` example, that would be sad; don't expect full credit for that.

Hand-in Instructions

You should put all your work, Source files and Documentation, into a zip file and submit it through Autolab.