A Python-Based Computer Music System

Performer

Introduction

Performer is a Python package designed to be an alternative to some of the features provided in Max/MSP and PureData+Automatonism. The three main goals of performer are efficiency, expandability and generality. The source code for this package is available in the GitHub repository `tareqdandachi/Performer`.

Implementation

Performer was implemented for use in Python. It uses numpy arrays to generate and manage audio streams. Numpy is a well-known package with plenty of documentation and an efficient implementation, making it easy and efficient to develop in for advanced Performer users. Performer relies on low-level C bindings, pre-compilation and smart memory allocation to manage its audio buffers efficiently.

Unlike Max/MSP and PureData, Performer has a centralized queue that manages all time updates. This means that every element is updated on request by a queue manager (housed in the AudioOut routine since it is always going to be the most updated thing). The queue manager knows how to optimally update things and allow for all updates to happen in sync and tries to make updates take a constant amount of time.

Performer was designed to be easily interface-able and extended, making it much easier to generate new Performer modules and use external packages with Performer. Unlike its counterparts, Max/MSP and PureData, Performer doesn't only support a fixed number of data types (signals, messages, etc.). Modules in performer take in any input type (in Max, the only input type to a DAC object is a signal). This means modules can take in mutable or immutable types, time-varying, constant or parameterized types and somehow has to update them without slowing down the queue, but also update them regularly enough.

Performer uses its queue routine to schedule updates in a way that doesn't hinder the AudioOut routine or block processing of different sections.

There are 5 main types of objects:

1. Audio Objects - AudioOutput, AudioInput, etc.

2. Oscillators - two main types, (1) simple oscillators, parametrized by time and variable objects only e.g. LFO, RAND, OSC. And (2) thing that are parametrized by time, variables and previous states: ADSR, reverb, etc.

3. Generators - a way to encode arbitrary functions into efficient multi-dimensional precomputed wavetables

4. Visualization - oscilloscope, spectrogram (need to manually run it and is not implemented into the queue yet)

5. Controllers - Managers that keep track of all variables, propagate changes and interface with modules.

I will skip details on the implementations but will elaborate a little on the features of controllers and queues in the next sections.

Signal Flow - OOP vs. SP

People coming from a Max/MSP program have experience in the signal processing (SP) way of implementing code. Being able to preserve this is important. People coming into digital music from a non-Max background, like me, are more familiar with the OOP structure. These two ways of programming are practically different but computationally equivalent. In this implementation, the code is written in an OOP structure with some functional advantage where appropriate. This gives the code the ability to run faster than a traditional SP program, however, it also adds a way of interpreting code in an SP way. This example showcases two different ways of writing the same piece of code using both an OOP and an SP structure.

Below are three examples showcasing different coding styles Performer was designed to accommodate. You can mix styles and it will be valid, as long as it makes sense to the musician. Regardless of the style, the code is invariant and adds no latency based on the code style chosen. The goal is that hopefully people familiar with the signal flow in Max, adding things onto one line sequentially (top-down processing) would be familiar with the signal pathway way of programming. It would also be easier to move into a GUI implementation similar to Max if I ever decide to move the package in that direction.

```python
### Object-Oriented w/ properties

from performer import *

audio = AudioOut()

out = ADSR( OSC(f=note.C4) + \
            OSC(f=note.A4) )

audio.audio = out
```

```python
### Object-Oriented w/ signals

from performer import *

osc1 = OSC(f=note.C4)
osc2 = OSC(f=note.A4)

sum = SUM(input=osc1, input=osc2)

out = ADSR(input=sum)

AudioOut(out)
```

```python
### Signal Pathway

from performer import *

out = AUDIO.empty_buffer()

OSC(f=note.C4, output=out)
OSC(f=note.A4, output=out)

ADSR(input=out, output=out)

AudioOut(out)
```

Interfacing with random packages

Being built in python, the package can interface with any python package! Using controller objects and Variable objects, you can propagate states instantly and add package support without changing much. Unlike Max for example, if you wanted to add CV support to make a face detection theremin, you would have to wait for someone to port OpenCV into Max. In this package, by changing 5 lines of code from an online example:

```python
while True:
    # Capture frame-by-frame
    ret, frames = video_capture.read()
    gray = cv2.cvtColor(frames, cv2.COLOR_BGR2GRAY)
    faces = faceCascade.detectMultiScale(
        gray,
        scaleFactor=1.1,
        minNeighbors=5,
        minSize=(30, 30),
        flags=cv2.CASCADE_SCALE_IMAGE
    )
    # Draw a rectangle around the faces
    for (x, y, w, h) in faces:
        cv2.rectangle(frames, (x, y), (x+w, y+h), (0, 255, 0), 2)
    # Display the resulting frame
    cv2.imshow('Video', frames)

    if len(faces) > 0:

        (x, y, w, h) = faces[0]

        print(x)
        print(y)

    else:

       print("no face")

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
```

```python
def control(controller):
    # Capture frame-by-frame
    ret, frames = video_capture.read()
    gray = cv2.cvtColor(frames, cv2.COLOR_BGR2GRAY)
    faces = faceCascade.detectMultiScale(
        gray,
        scaleFactor=1.1,
        minNeighbors=5,
        minSize=(30, 30),
        flags=cv2.CASCADE_SCALE_IMAGE
    )
    # Draw a rectangle around the faces
    for (x, y, w, h) in faces:
        cv2.rectangle(frames, (x, y), (x+w, y+h), (0, 255, 0), 2)
    # Display the resulting frame
    cv2.imshow('Video', frames)

    if len(faces) > 0:

        (x, y, w, h) = faces[0]

        controller.set_param('freq', x)
        controller.set_param('amp', y/500)

    else:

        controller.set_param('amp', 0)

    if cv2.waitKey(1) & 0xFF == ord('q'):
        pass
```

The package has a lot more features and design decisions I would love to get into, but to stick to the page limit, I decided to include the things that are easiest to show off non-interactively and without a lot of technical details. I have been adding examples over the past 2 weeks of different features of the package and am slowly moving them from the demo and examples directory to a readable format in the README.md with explanations. There were a lot of design decisions inspired by Max and PureData that I also couldn't get into here, including how to handle Max datatypes and cool MIDI messaging features!! The package supports MIDI inputs and outputs and can connect to any DAW as a MIDI device too!