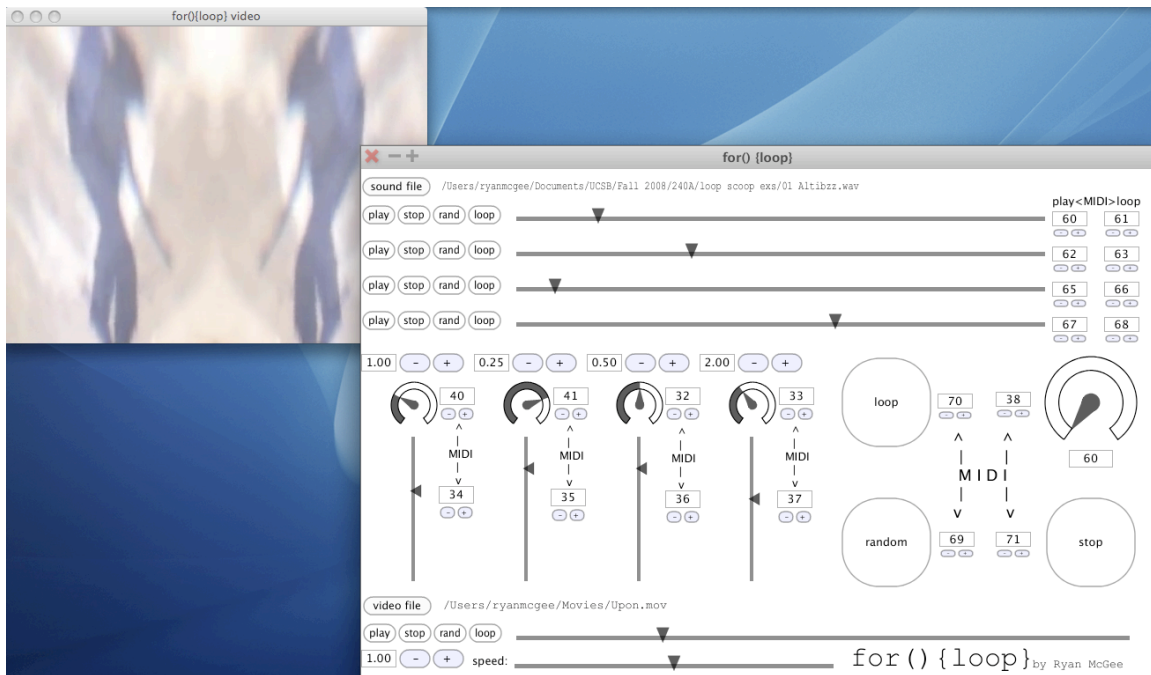


for () { loop }



Ryan McGee

MAT 201B

Winter 2009

ABSTRACT

The purpose of creating For Loop was to explore C/C++ libraries for sound, video, MIDI, and a GUI. For Loop uses JUCE, SDL, and OpenCV. The end product is a tool for creating interesting textures of audio loops accompanied by a video track. Full MIDI control allows For Loop to be used as a performance tool as well.

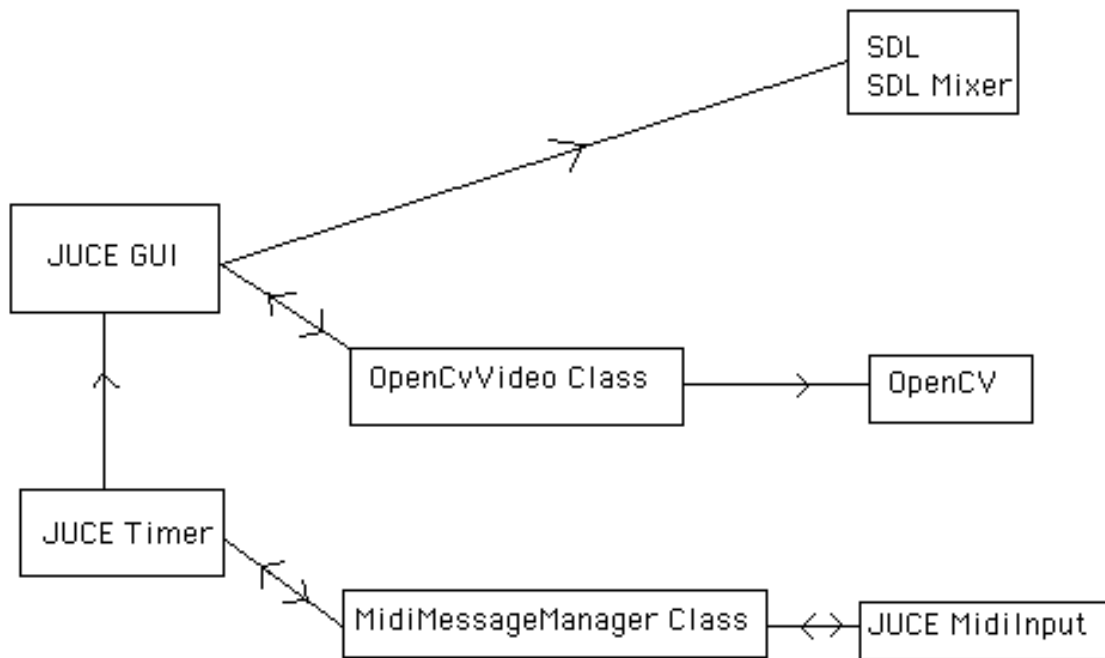
INTRODUCTION

For Loop was begun as a project for MAT 240A. By the completion of that course, For Loop was simply four track audio looping tool featuring a random button to generate random textures of loops. The goal for 201B was to learn and implement APIs for video and MIDI data. In addition to learning new API's and how to handle video and MIDI data, this project was my first experience in authoring my own C++ class.

Initially I had planned to use the ffmpeg library to import video files and convert them to a surface for display using SDL. However, after several frustrating hours of attempting to use ffmpeg, I chose a different route. Though not quite as robust, OpenCV turned out to be a good alternative for reading and displaying video. OpenCV offers support for many video formats including .AVI, .MOV. and . WMV. OpenCV also includes several GUI functions for creating windows to display video frames.

MIDI control has been added thanks to the many MIDI related classes in JUCE. While RT MIDI and PortMIDI were explored, I finally decided to use JUCE since it was used for the GUI as well. SDL and SDL_Mixer libraries remained the top choice for handling audio. SDL works well with .WAV and .AIFF files and has convenient built in functions for looping sounds.

PROGRAM FLOW



My JUCE user interface component interacts directly with SDL and SDL_Mixer to deal with audio. My custom OpenCvVideo class provides a pathway to OpenCV's video functions. The JUCE timer triggers button presses in the GUI by receiving MIDI messages from JUCE's MidiInput class through the MidiMessageManger class.

CODE DESCRIPTION

Firstly, the JUCE GUI component (rmmComponent) is initialized. In the constructor of rmmComponent, SDL and SDL Mixer are initialized with the common parameters sample rate = 44100Hz and number of channels = 2. Initial values for knobs, sliders, and MIDI data are also set during the initialization of the GUI component. Once initialized, the GUI button, sliders, knobs, and value boxes control all events in the program. Additionally, a JUCE Timer checks for MIDI data that can be used to trigger events in the program.

Clicking the “sound file” and “video file” buttons activate JUCE file browser windows and the user selects the audio and video files they wish to manipulate. When the sound file is chosen, the program loads it into four SDL Mix_Chunk structures containing the audio files sound data and length among other information. Since the lengths of each of the four copies of the sound will be altered, an additional reference Mix_Chunk is created to house the length and data of the entire sound file. The max range of the position sliders are then set to match the number of samples in the audio file. After the sound file is loaded, the master JUCE Timer is started. The length and starting play position for each of the four copies of the sound file can then be altered using the length and position GUI controls respectively.

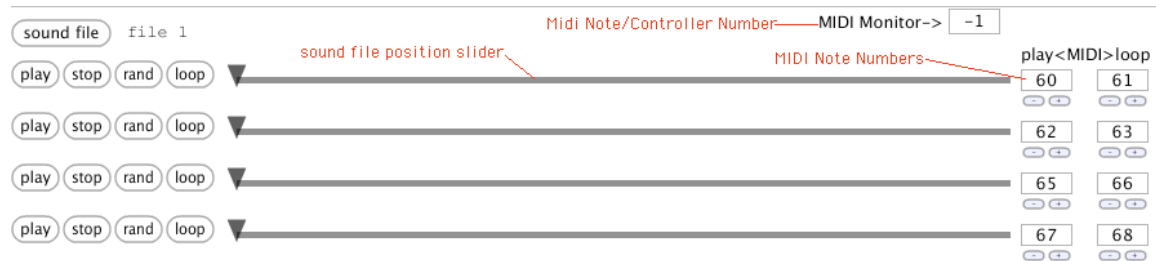
The GUI also provides the user with play, stop, rand, and loop buttons to control the playback of each sound with SDL_Mixer functions. Panning knobs and volume sliders control the mix of the sounds. Detailed explanation of each control along with descriptions for the large “loop”, “random”, and “stop” buttons can be found in the Control Descriptions section of this document.

Rather than calling library functions directly from the GUI, the video component to For Loop used a custom C++ class that interact with OpenCV. Upon selecting a video with the “video file” button, the videoInit() function of my OpenCvVideo class is called. This function creates an OpenCV capture from the chosen video file and computes the video’s frames per second (fps) and total length in frames (totalFrames). Next, videoInit() stores each frame of the capture into an OpenCV IPLImage structure. This process can be somewhat time consuming for longer videos, so an OpenCV window appears to show the status of the load.

Once loaded, the video is ready for display and looping using the videoDisp() function. This function first checks to see if the loop flag is true. If the loop flag is true, then OpenCV playback enters a while loop that plays the video repeatedly until the loop flag is set to false. Playback in the videoDisp() function requires knowing the video frame to start on as well as the video frame to end the loop. The frame to start playback is determined by the getStartPos() function that simply reads the value of the video position slider. The getLoopEndFrame() function returns the appropriate frame to end the sequence by adding the loop length in frames to the

starting position. The loop length in frames (vidLengthFrames) is computed using data from the loop length controller, the video's frames per second, and the videos playback speed set by the video speed slider. Once the start and end position are known, the videoDisp() function simply cycles through the array of stored capture frames at the appropriate indices using a for loop. The cvWaitKey() function is used to determine the amount of delay between for loop iterations, and thus determines the playback speed. The video speed factor (vidSpeedFact) is used along with the video's fps to determine how long OpenCV should wait before displaying the next image in the video sequence. The video speed factor translates values -4 to +4 from the video speed slider into the appropriate factor to divide the wait time in milliseconds. This factor is also used to multiply the required length of frames needed to play back the loop at a particular speed with the same duration in loop time.

CONTROL DESCRIPTIONS



sound file: opens browser window to select sound file. Uses JUCE's FileChooser class

play: plays the sound loop once. Calls SDL's Mix_playChannel(channel#, sound, # of loops = 0)

stop: stops playback on selected channel using Mix_haltChannel(channel#)

rand: uses rand()%(total length in samples of the input file) to generate a random start position, then updates the position slider value

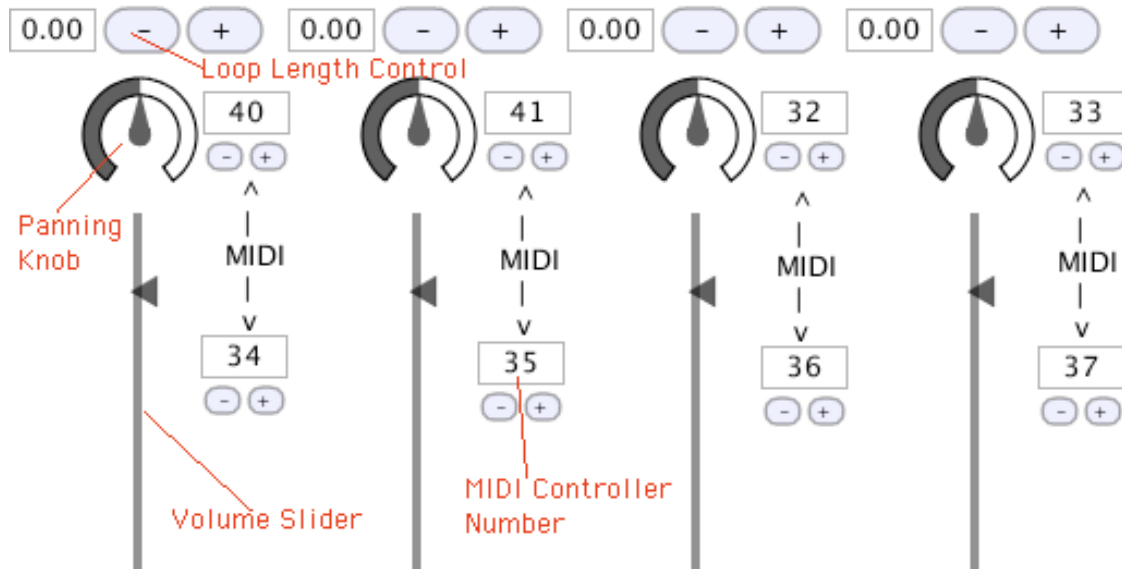
loop: plays the sound loop until stop is clicked or loop is clicked again. Calls SDL's Mix_playChannel(channel#, sound, # of loops = -1 (indefinite))

sound file position slider: sets the start position for the sound loop. Modifies the pointer address in the sounds buffer (address of loop buffer = address of total sound buffer + position slider value)

MIDI Note Numbers: the user can input the desired midi note by which to trigger single or looped playback of the sounds. JUCE's timer callback function checks the value of incoming midi data from the MidiMessageManager class against the values

in these boxes. If there is a match, then the appropriate button is triggered using the button->setTrigger() function in JUCE.

MIDI Monitor: This box monitors incoming MIDI signals and displays the note or controller number of the most recently played key or moved controller respectively. The user can use data from this box to map keys and controllers to the desired functions in For Loop.

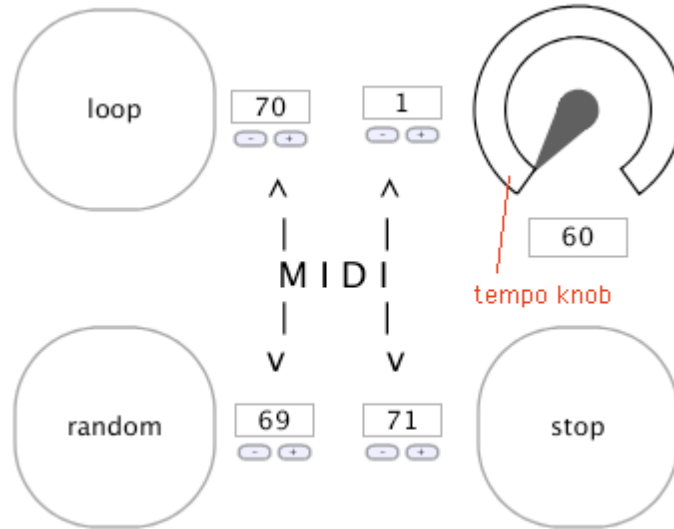


Panning Knob: changes the panning of each mixer channel. Uses `Mix_SetPanning(channel#, left value, right value)`.

Volume Slider: changes the volume of each channel using `Mix_Volume(channel#, value)`

Loop Length Control: sets the length of each loop according to the tempo value (see tempo knob). Values are 0.25 through 4 in 0.25 increments to correspond to quarter notes through 4 whole notes.

MIDI Controller Number: the user can input the desired midi controller by to use in controlling each channels volume and panning. JUCE's timer callback function checks the value of incoming midi controller number from the `MidiMessageManager` class against the values in these boxes. If there is a match, then panning or volume sliders value is updated to match the MIDI controller value.

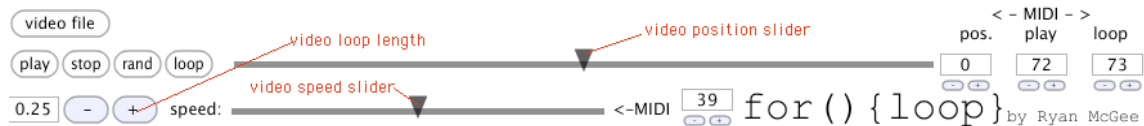


large loop button: starts all channels along with the video looping in synchronization.

large random button: computes random numbers for all parameters of the program using the rand() function. Sounds will be given a random start position, loop length, panning, and volume. The video will be given a random start position, loop length, and speed.

Large stop button: stops playback of all audio and video, Mix_haltChannel(channel# = -1 for all channels) and sets the video loop flag to 0 (false) using the setVidLoop(loop = 0) function.

Tempo knob: sets the tempo for playback in BPM



video file, play, stop, rand, loop: same playback functionality as sound buttons

video loop length: same operation and values as for sound

video position slider: sets the start point for the video loop

video speed slider: sets the playback speed of the video. Values are from -4 (1/4 speed) to 4 (4X speed). This slider determines the video speed factor that is used in the OpenCVVideo class' videoDisp() and setVidLength() functions. It is worth noting that while the playback speed can be changed, the video loop length remains constant to stay in synch with the sound loops.

video MIDI: Just like with the sound loops, the user can control playback of the video with MIDI keys and controllers. In addition to triggering playback and looping with key presses, the user can also map a MIDI controller to control video start position and playback speed.

CONCLUSION

Overall, the project was a success. For Loop is able to read, playback, loop, and mix audio files as well as read, playback, loop, and slow down/speed up video files. The User is able to set lengths accordingly to synchronize audio and video playback and the random button is capable of producing “generative” music/multimedia works of sorts. MIDI control works well and I enjoy the simplicity of the user interface.

However, synchronization between audio and video is far from perfect. Sounds are looped according to their sample length while videos are looped according to their number of frames. Since frames of video are sampled much less than samples of sound (30fps vs 44100 samples/sec), the loops tend to fall out of sync fairly often, though often close enough to be acceptable. Synchronization could be fixed by implementing a global timer along with delay or waiting methods. The WaitableEvent class in JUCE was explored as an option, but with no success by due date of this project.