| | |
|---|---|
| **Subject Name:** | **CS3220 Digital Audio and Applications** |
| **Subject Lecturer:** | **Nuno Barreiro** |
| **Date of Submission:** | **12.12.16** |
| **Student Marking ID:** | **8582** |
| **Team:** | **A** |

# CS3220 Assessed Coursework 2 Digital Audio and Applications Project 2 - Individual Report

# Features

This pure data patch is divided into four sections, each displaying a different aspect of the specified deliverables; generate music patterns, be able to adjust the speed at which they play and create smooth transitions between those music patterns in the form of a crossfade.

Some sound files need to be loaded in for the patch to work (please refer to README.txt). The blue section of the patch is used to check if the sounds have been initialised properly.

The orange section is used to display both the transition of sound as well as changing the rates that they play. The Volume v-slider should be adjusted first before testing. Two bangs are used to transition between the sounds, whilst the Speed-of-Transition-from-Piano-to-Guitar v-slider is able to adjust the rate at which this happens (between half a second and two seconds). The Speed-For-Sounds v-slider changes the speed of the currently playing sound. Both v-sliders can be used in tandem with each other; a fast-tempo version of Piano can transition into a version of Guitar that plays at the same BMP.

The green section is used to illustrate a manual crossfade between both wav files.

Finally, the pink section displays transitions between two auto-generated sound patches. The Background-On-Switch and Discovery-On-Switch bangs need to be pressed first to activate the patches. The six bangs above carry out transitions with pre-set change rates between the patches whilst the left v-slider can perform manual transitions. The right v-slider influences the sensitivity of the Background music patch.

# Research

Whilst our main focus of this project was to develop a pd patch that can crossfade between two auto-generated music patches, we first wanted to understand how crossfading and transitioning worked between two different sounds. Therefore, as a proof of concept, our first goal was to demonstrate how to import sound to the pd patch so that we can use it for testing.

Freesound.org is a website/collaborative database for common sounds where users can upload and use recorded sounds without having to worry about licencing. This site provided us with the two found files that we needed in order to prove that our project contains the desired deliverables. Two different sound files of a piano melody[1] and an acoustic guitar rift[2] were selected specifically for their short runtimes of around 5 to 8 seconds (as to not have too big of file size) and the fact that a piano and guitar are two instruments that produce distinctively different sounds. Once downloaded, these wav files were altered in Audacity (a sound editing software) and cropped so that they do not play their silent sections. They were also cropped so that when these file were looped, they played on beat.

Speaking of looping, that was the next major issue we had to overcome. Importing these files was a simple issue to solve by simply loading a bang through an 'open' object, but it would cause it to only play once. After doing some more research, we studied a few papers and online tutorials[5] that taught us about the 'readsf~' object. This object has two outlets; the hot outlet for the sound and the cold outlet that indicate that the sound has finished playing. Using this new knowledge, we can loop the sound back to the open object in order to play the sound again.

Now that we could both import and loop sounds, the next step was to develop a mechanism that was able to perform audible transitions between two sounds, all preferably controllable from one slider. After going through official tutorials on how to implement objects in pd[3], we realised that we could connect both audio files to run through a single slider object and then filtering which sounds were heard via a minus modulator object, so that one file plays when the slider is set to zero and then slowly lowers the amplitude of this sound whilst increasing the other sound until the slider reaches a value of one, where the original sound can no longer be heard as its amplitude is now at zero and the new sound is set at full volume. This proof of concept would eventually form the basis of what would eventually become the VolumeChangerForTwoSounds pd patch.

Another problem we faced was trying to implement a v-slider that would alter a sound's sampling rate thus changing the rate of which the sound plays, causing it to speed up or slow down. It is at this point in our research that we came across the works of Dr Rafael Hermandez[4], who directed and supervised online tutorials for applying specific techniques in Pd.

As mentioned before, our goal was to achieve these techniques and apply them to generative sound as well instead of only reading in sounds from files. To do this, we needed to obtain a generative music patch and adapt our current program to accommodate the change in which music is played. Eventually we found some papers on auto generative music[6][7] and also some obtained pd music patches developed by Martin Brinkmann. Although we did not develop these patches ourselves, we have modified them so that they can send and receive data between different layers of our program more efficiently using a series of bangs and multiplication modulators.

We also investigated sound modelling and morphing between the piano and guitar. Morphing two sounds on their own both sound pleasant yet are distinctively different from each other, yet they still sound pleasant to the human ear when combined.[9]

Finally, we went back through some of the lab worksheets in order to help us as well. For example, one of the deliverables was the ability to adjust the rate of transition between two sounds. This meant we had to make the transition time customisable. To do this, we implemented both packing and unpacking techniques and Attack Sustain Release models[8] learned in Lab 4 where we feed a value given by a slider that determines the duration of the transition into the attack message box, thus affecting the speed of the transition.

## *Bibliography:*

[1] Lemoncreme, Piano Melody, freesound.org, 2013
[2] timcam, acoustic-guitar, freesound.org, 2010
[3] Johannes Kreidler, Programming Electronic Music in Pd, Wolke Publishing House, 2013
[4] Rafael Hermandez, Pure Data, cheetomosqueto.com, 2009
[5] Ob Wex, Tutorials in Pd, examelo.com, 2012
[6] Andy James Farnell, Synthetic Game Audio with Puredata, obiwannabe.co.uk, 2006
[7] Leonard J Paul, Video Game Audio Prototyping with Pure Data, Vancouver Film School 2007
[8] Lab 4, Royal Holloway Moodle
[9] Kelly Fitz, Sound Modelling and Morphing, CERL Sound Group, 2007
[10] http://forum.pdpatchrepo.info

# Code

As mentioned before, the program is split into four different sections. I will do my best to discuss all aspect of the program however I shall mainly focus on the more intricate and fundamental mechanisms in the program and skim over the less relevant ones. The Pd patch is presented as a nicely formatted GUI, where the inner mechanisms of the program are hidden beneath multiple layers of sub patches. These patches are connected via a network of sliders and bangs that can send and receive information between these layers.

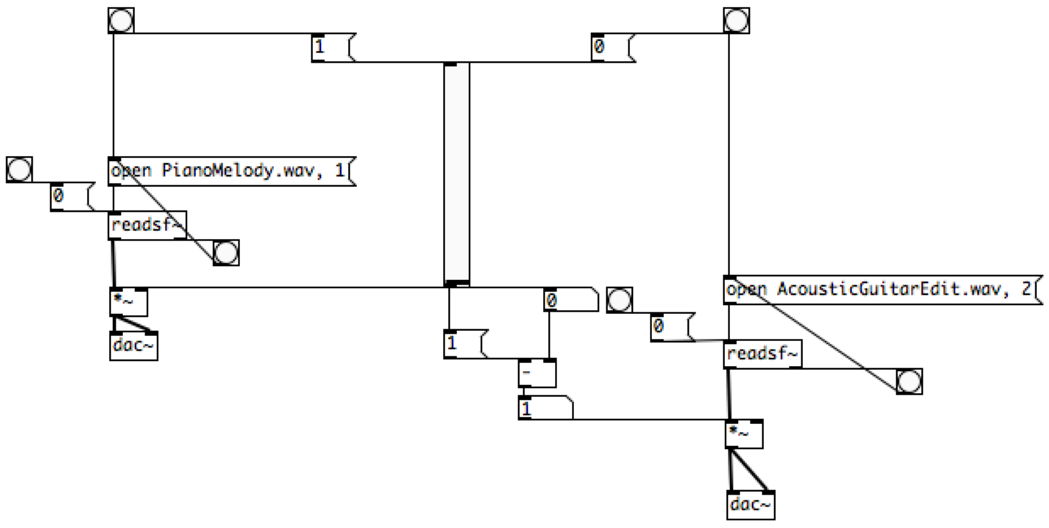

*Figure 1 – VolumeChangerForTwoSounds pd patch*

The VolumeChangerForTwoSound pd patch (blue) is one of the most essential patches in our project. It uses a combination of looping and amplitude modulation techniques in order to produce one of two playable sounds. Using the figure above as reference, the two bang objects in the top corners receive information from Start_Guitar and Start_Piano from the main Pd patch. Open messages are used to obtain the data from the sound files and the readsf objects read them. This gets fed into a dac that plays the sound. Because we want the sound to loop, a bang is connected to the cold out of readsf and is sent to a bang that reopens and rereads the file again once the last iteration of the sound has finished playing. Turning off the sound is done by a bang sending a zero message to the inlet of readsf. Both sounds would now play simultaneously so we need to be able to control which sound we want to hear. This is all carried out by amplitude modulation via a single v-slider that sends opposing data to both sounds by keeping one version of the outlet intact and another version the inverse of the slider's readings.

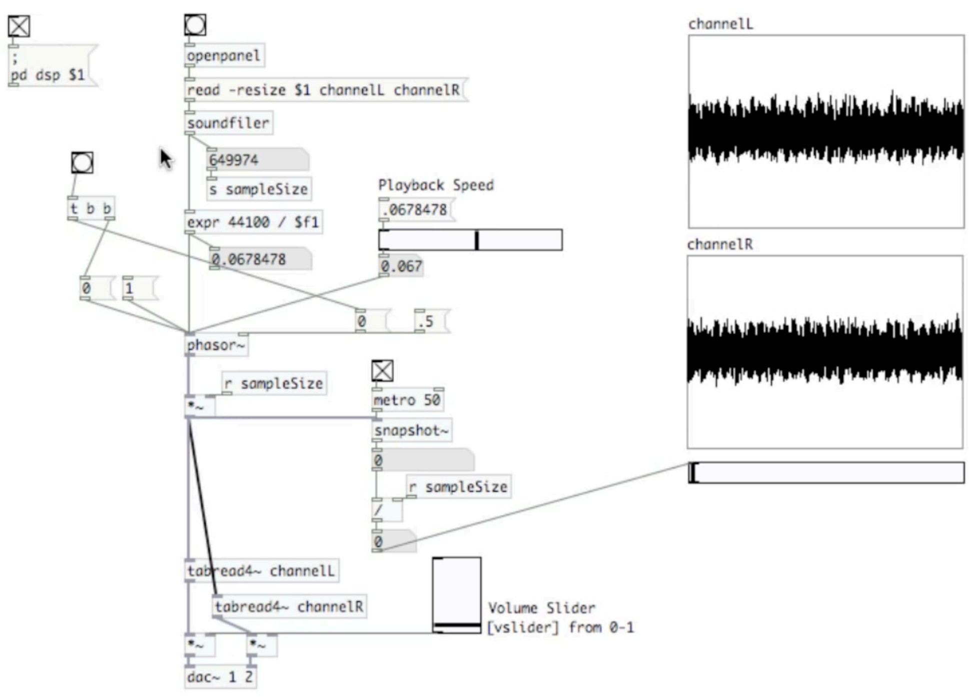

*Figure 2 – Speed Changer pd patch (taken from Dr Hermandez's tutorial on tabread4~)*

The SpeedChanger pd patch (orange) is almost an altered version of the VolumeChanger pd patch, the only difference is the added mechanism to change the speed at which the sounds play at. This is done by implementing tabread4 objects to produce graphs that represent the entire sound waves made by the files. The value taken from a soundfiler object that stores a value representing the sample size of the file whilst an expr object returns the program's reading rate of said file. The sound wave is also divided by the sample size and fed to an h-slider that now represents what point in the file is currently playing. As for changing the sampling rate, we can do this by sending a multiplier to the data being into a phasor object that converts the data from the file into actual sounds. This multiplier can even be of a negative value and as a result play the sound in reverse. Finally, the program implements packing and unpacking of customised values so that we can adjust the rates of transitioning between both sounds.

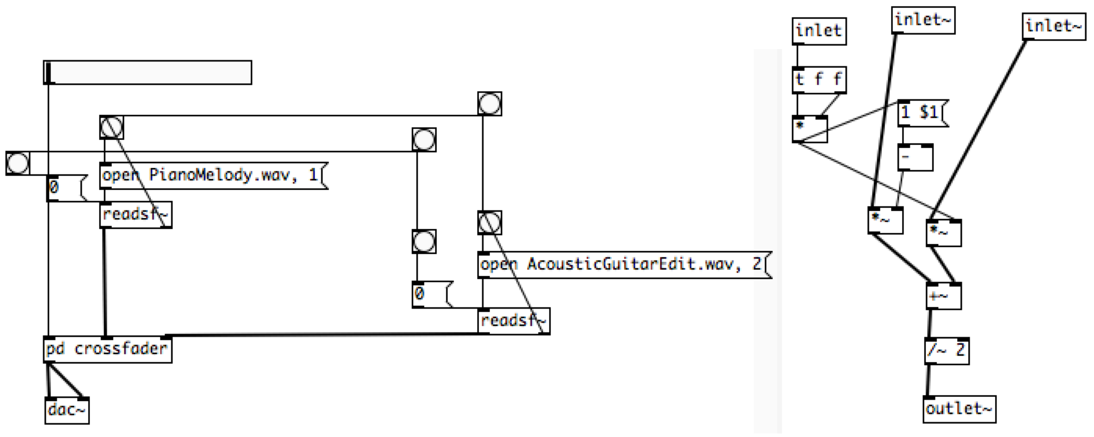

*Figure 3 – CrossFadeForTwoSounds pd patch (left) and crossfader pd sub-patch (right)*

The Crossfader patch (green) is also similar to the VolumeChange patch but instead of carrying out the audio modulation processes in the same patch, a crossfader object is created that controls which sound is playing via a slider. As usual, the sounds can start and stop by applying a bang that sends a 1 or 0 to the sound to louden or quieten it respectively. As for the crossfader object, it takes in three separate inputs; the value from the slider representing how loud the sound plays and the two sound files. One of these sound files will simply be multiplied by the exact value from the slider whilst the other is multiplied by one minus the slider value. This causes both sounds to be of opposite amplitudes; as one sound increases, the other decreases and vice versa. This information then gets sent to the dac where the sounds can be heard. Also, the slider in this patch receives data from the Slider_For_Crossfade slider in the main patch, thus the user can perform manual crossfades between the two sounds.

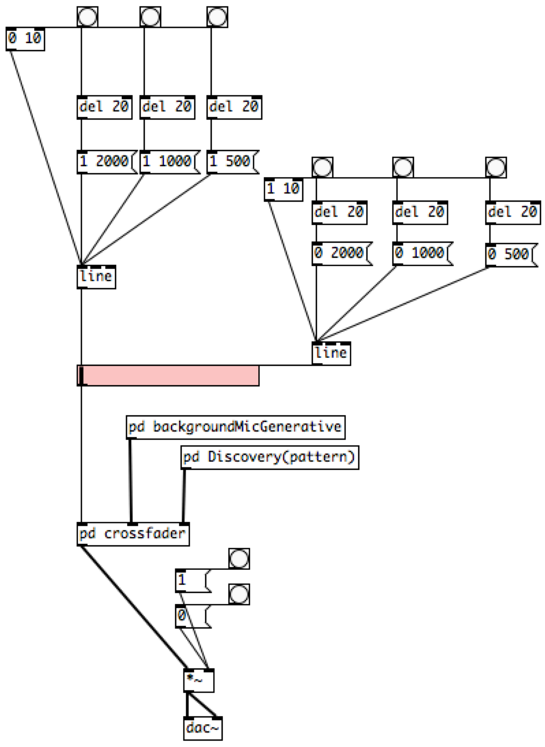

*Figure 4 – CrossFadeBetweenTwoGenerative pd patch*

Lastly, there is the CrossfadeGenerative pd patch (pink). As mentioned before, the two generative music patches were not created by us, however we have adapted them to send sound through our program. The two bangs near the bottom of Figure 4 receive data from the main pd patch and are used for turning both sounds on and off. The crossfader object is also used here since we still wish to perform transitions between two sounds. The BackgroundMicGenerative patch creates music based on sound fed into the computers microphone whilst the Discovery patch generates music using random number generators and other sub patches.

## Conclusion

The program is able to perform both automatic and manual transitions between two different sound files in a smooth manner. It can alter the sampling rates of these files in order to change their play speed, and can even maintain that speed as it carries out the transition. Finally, it can produce generative music in the form of a pd patch and can also perform transitions between two different patches. It is able to do all of these things by using a combination of techniques learned throughout the course (packing and delays), methods that were researched for this project to work (altering speed and looping sound), and connecting everything together using bangs that send and receive data between layers of sub-patches.

As for improvements that we could have applied, one would be to integrate sound morphing techniques and apply them to the transition of our sounds. This would have allowed us to transit not just amplitude but also the frequency between sounds. We also had the idea of importing sounds from midi generation software in order to produce the generative music. This would have allowed us to create our own music, save recordings of them as wav files and read them into the final program. It would have also been nice if we could adjust the sampling rates for the background and discovery music patches as well, rather than just the wav files or to have the “start” and “stop” toggles to be merged into one single toggle.