Write Up
Advanced Computer Music- Task One

For this task I was asked to "Implement a DSP (Digital Signal Processing) routine at the level of individual samples which synthesizes and or/effects sounds."

Since the second world war the idea of performing digital signal processing tasks has been considered, "at the end of the 1940s, Shannon Bode and other researchers at the Bell telephone laboratories discussed the possibility of using digital circuit elements to improve filter functions"[1]. In the 1950s at the Massachusetts Institute of technology the idea of digital signal filtering was shared with graduates, by that point they had developed a good understanding of sampling and it's spectral effects. Despite this earlier research it was not until the mid 1960s that a more evident theory of digital signal processing came forth. In 1957 Max Matthews digitally synthesised sound on a computer and wrote Music 1, the world's first music programming language. In 1961 (using Music IV) he created a digitally synthesised version of a piece by Harry Dacre known as 'A Bicycle Built For Two'. Although when listening to it now the sounds seem quite basic it was a major breakthrough in terms of digital audio synthesis.

The program in which I chose to implement my task one assignment was SuperCollider which uses many of the same principles as MUSIC 1 did such as being made up of a library of functions and often explores the idea of plugging together unit generators. The challenge of this task however was not to use Ugens and to build everything at sample level.

The sample rate I used for my synthesis task was the standard sample rate of 44100. I then made a variable 'freq' containing the value 441 which would be simpler to work with. It was set to 441 because if there was, for example a sine wave which "moved in a phase of π* 0.02, it would be moving at 1/100 cycles per sample"[2]. If it takes 100 samples per cycle the frequency will be 44100/100=441 Hz. This would allow me to measure the change in phase as it is converted to an easy to use frequency in cycles per second.

For my project my original idea was to use wavetables to store my samples which would be made up of different synthesised musical tunes using midi notes. The program would increment through the values in the wavetables, it would reach the end of the table and then loop back to the start again continuously. Changing speed in which the phase increments changes the pitch; fast incrementation resulting in a higher pitch and vice versa. "This in effect, shrinks the size of the wavetable in order to generate different frequencies"[3]. Roads explains that if a wavetable was only scanning through even values it would go twice as fast as the odd values are missed out. In my project the effect of changing frequency is determined by the random midi notes that are generated as reaching them will cause the wavetable to increment at different speeds. Going from a C2 to a D2 would require incrementing to speed up and the opposite would be true for D2 to C2 Wavetables are very efficient and take up minimal space, this this why they were used a lot in early digital synthesis as it meant they could easily be stored on a chip and looped through. Each wave table is set to hold 256 samples:

```
wavetable = Array.fill(numvoices,{ Array.fill(256,{ rrand(-
1.0,1.0)}); });
```

256 is used as each wave table holds a byte (8 bits) of data. This would have also been appropriate in terms of early more widely produced digital synthesis technology as old 8 bit technology would have been used in early common systems around the 1970s as it was the first mass produced and financially viable option. In terms of my project it is fairly cost efficient in regard to CPU. The array is filled with white noise which is uniformly distributed to each sample using a value of -1 to

1.

```
tune = Array.fill(numvoices,{

Array.fill(rrand(10,20),{[rrand(0.0,20.0),rrand(36,84)]});

 });
```

Originally I was going to put a different midi note tune into each wavetable using a dictionary. I found that when programming it, it became inefficient as I was copying and pasting the code and changing the array index each time. In order to make it more efficient I got rid of that and the numvoices variable was added so that the number of wavetables could be specified at the start of the program. The more wavetables there are the more broad the spectrum of sound and more thick sounding in texture. The above line of code shows how the rrands in the array are applied to the numvoices (I.e however many wavetables have been specified). rrand(10,20) randomly changes the number of notes, rrand(0.0,10.0) changes the time in which the notes change and rrand (36,84) randomly selects midi note values between 36 and 84 which are midi numbers for between C2 and C6. I think this worked out better than my original idea as it adds an element of unpredictability and I found it interesting listening to the different outcomes that the code generated.

```
tune = tune.collect{|val|   val.collect{|note|
[(note[0]*sr).asInteger,note[1]] };  };
```

The above line of code collects the value of the notes in the tune array and then multiplies them by the sample rate, this is so that the numbers used are sample numbers as opposed to midi note numbers.

```
file =
SoundFile.new.headerFormat_("WAV").sampleFormat_("int16").numChann
els_(1).sampleRate_(sr);

file.openWrite("/Users/fox/Desktop/writetestg.wav");
```

This part of the code writes a 16 bit mono wav file, I have set the file path to my desktop. This allows the code to run outside of the box. This demonstrates non-real time synthesis and means the localhost server in SuperCollider does not need to be running. Digital sound processing was originally done in non-real time.

```
numvoices.do {|j|

posnow= phase[j]*256;
prevpos= posnow.floor;
nextpos= (prevpos+1)%256;

tune[j].do{|note|  if(note[0]==i,{ phaseincrement[j] =
note[1]/sr; }); };

t= posnow-prevpos; //interpolation position
```

In the above code posnow tells us the current position that we are at in the wave table, prevpos is

the floor of the current position (which means the lowest possible value) and nextpos adds one sample. The modulo operator means that we wrap around to the start of the wavetable once we reach the end. This is an implementation of what Curtis Roads describes as "the most basic oscillator algorithm can be explained as a two step program:
1.Phase_index=mod(previous_phase+increment)
2.Output = amplitude x wavetable[phase_index]"[4]

(mod is denoted)

This makes an assumption that the wavetables are already filled with values. If the frequency and table length are fixed (which they are in my program) then the sound that the oscillator makes depends on the increment.

This formula which I learnt from Curtis Roads Computer Music Tutorial (p93) shows the relationship between a given frequency and increment.

$$\text{Increment} = \frac{L \times \text{Frequency}}{\text{samplingFrequency}}$$

The line of code following tune[j].do makes sure that it iterates through all the values in the array and performs the above formula.

```
output= output +  (((1-t)*wavetable[j][prevpos]) + (t*wavetable[j]
[nextpos]));

phase[j] = phase[j] + phaseincrement[j];

if(phase[j]>=1.0) {

phase[j] = phase[j]%1.0;

};

};
```

The next part of the code (above) performs interpolation. Interpolation is used when we fall between the gaps in the wavetable. It is used to calculates where it thinks the correct point in the wavetable would have been. It "interpolates between the entries in the wavetable to find the one that exactly corresponds to the specified phase index increment[5]". The smoother the interpolation then the smoother the output sound will be.

```
xv[0] = xv[1]; xv[1] = xv[2]; xv[2] = xv[3]; xv[3] = xv[4]; xv[4] = xv[5];
xv[5] = output/8.025238577e+06; //gain
yv[0] = yv[1]; yv[1] = yv[2]; yv[2] = yv[3]; yv[3] = yv[4]; yv[4] = yv[5];
yv[5] =   (xv[0] + xv[5]) + (5 * (xv[1] + xv[4])) + (10 * (xv[2] + xv[3])) +
(0.7582507704 * yv[0]) + (-4.0009954267 * yv[1])+ (8.4507749887 * yv[2]) + (-
8.9314219076 * yv[3]) + (4.7233875877 * yv[4]);
output = yv[5];
```

Every signal is low pass filtered using a Butterworth filter (shown above). I used an online filter design tool which generated the co efficients into C code which I then adapted slightly to work in Super Collider, I have included a link to this website in my references: [6]. A filter takes in the previous outputs and previous inputs into a linear sum with coefficients and multiplies and sums together the values. In digital signal processing any processing can be characterised as a filter if it takes the input and gives an output, it is like a function. Butterworth filters have a flat frequency response. My filter is a low pass filter with corners frequencies set to 400 and 1000 and it has 5 poles. I think it did make quite a lot of difference to the sound produced as it dampened any distortion and now it sounds lower and more resonate but still quite dull and impending. This is because the high end frequencies have been filtered out.

Overall I feel that my program has achieved its objective to Implement a DSP routine at the level of individual samples which synthesizes and or/effects sounds. If I was able to extend and improve it I would like to try and make different types of waves to put in the wavetables rather than just using white noise. I would like to create different filters so I could have achieved my original vision of applying different filters to different wavetables to create a wider variety of sounds. I would have liked to experiment with enveloping in order to shape the dynamics of the sounds created.

The techniques used emulate those that would have originally been used in the 1970s due to the use of 256 sample wavetables. This technique was commonly used in video games in the 1980s as it replaced the chip based techniques that had previously used real time sound synthesis such as in Amiga machines. The use of generating random values to give the musical output demonstrates use of algorithmic composition. “In it's purest form, (computer based) algorithmic music is the output of a standalone program, without user controls, with musical content determined by the seeding of a random music generator”[7]. A famous example of this is the Illiac Suite which was the first piece of music to be generated by a computer. It was created by Lejaren Hiller and Leonard Issacson at the university of Illinois. The computer was programmed to select musical notes randomly subject to general instructions derived from rules of musical composition. The Illiac suite differs quite greatly from my program as the computer that generated the Illiac suite printed out a sequence of letters and numbers that were translated to a musical score. This was played by human musicians in the form of a string quartet as opposed to writing a wav file to the desktop.

1 Stranneby Dag, Walker William (2004) Digital Signal Processing and Applications, (p1), Elsevier, Oxford
2 Collins, Nick (2010) Introduction to Computer Music, (p168), John Wiley and sons, Chichester
3 Roads, Curtis (1996) The Computer Music Tutorial, (p92) MIT Press, Cambridge, MA
4 Roads, Curtis (1996) The Computer Music Tutorial, (p93) MIT Press, Cambridge, MA
5 Roads, Curtis (1996) The Computer Music Tutorial, (p94) MIT Press, Cambridge, MA
6 http://www-users.cs.york.ac.uk/~fisher/mkfilter/
7 Collins, Nick (2010) Introduction to Computer Music, (p300), John Wiley and sons, Chichester