Computer Science 327
Project 1, Part c
C Programming Project
Digital Sound

The Assignment (Part c)

The part continues to extend the capabilities of creating and processing sound. You should also note that I have included a "likely test methodology" in each part to give you an idea on how we will test. Although said in many places. For the sanity of the TAs, please place everything in the top-level directory.

1. In this part we will perform a project cleanup. This often happens with real projects.

   We will do this by creating a new file named "sound.h" that contains the structure definitions for sound. Any new structures definitions or macros pertaining to sound data should go in here. Add this file to the git archive.

   Now modify gensnd.h so that it also includes sound.h at the beginning. Change gensnd.c and gensnd.h so that only the functions that were created in part 2 are present with the following changed names. (i.e., get rid of the 2 where applicable) We will not be using the functions or prototypes from part a.

   ```
   sound * gensine( float hertz, float sample_rate, float duration);
   sound * genDTMF(char key, float sample_rate, float duration);
   sound * genSilence(float sample_rate, float duration);
   ```

   Create and iosnd.h and iosnd.c file that contains the

   ```
   int outputSound( sound *s, FILE *f);
   ```

   prototype and its implementation. Add these files to the git archive.

   At the end of all this you should have three header files and two C implementation files.

2. In this part of the assignment, you will add to your gensnd.c (and gensnd.h) file to include different waveforms to create sound. A good source on the web that will allow you to hear these waveforms and why they sound the way they do is here:

https://www.perfectcircuit.com/signal/difference-between-waveforms.

Modify the gensnd.h file and gensnd.c to add several additional sound generating functions. The prototypes for these functions are:

Generate a *square* wave with the specified frequency, sample_rate, and duration.
`sound * genSquare( float hertz, float sample_rate, float duration);`

Generate a *triangle* wave with with the specified frequency, sample_rate, and duration.
`sound * genTriangle( float hertz, float sample_rate, float duration);`

Generate a *sawtooth* wave with with the specified frequency, sample_rate, and duration.
`sound * genSawtooth( float hertz, float sample_rate, float duration);`

Likely test methodology: Generate a short sample of each of the waveforms and graph the output. Verify visually that the waveform looks correct. Utilize software to determine frequency and duration are correct in the wave data given the parameters.

3. In this part you will be concerned with mixing signals at various sound levels. While usually sound volume is specified in decibel, we will make it simple and just use a simple multiplicative factor. Create files process.c and process.h that will contain the following function and prototype.

   `sound * mix(sound *s[], float w[], int c);`

   This function takes an array of sound points and an array of volume levels represented by floats. The parameter c specifies the number of sounds and volumes present in the arrays. The pointer to a sound structure returned is the sum of all the sounds weighted by the floats in w. Suppose we have two sounds x and y where x contains the samples 0.9, 0.6, 0.3, 0.6; and y contains the samples 0.4, 0.8, 0.8, 0.2, and w is 0.3 and 0.5. The sound created would be the samples

   Sample 0: 0.3*0.9 + 0.5*0.4 = 0.27 + 0.2 = 0.47
   Sample 1: 0.3*0.6 + 0.5*0.8 = 0.18 + 0.4 = 0.58
   Sample 2: 0.3*0.3 + 0.5*0.8 = 0.09 + 0.4 = 0.49
   Sample 3: 0.3*0.6 + 0.5*0.2 = 0.18 + 0.1 = 0.28

   If the samples are not the same length, then sounds that are shorter than the longest sound are assumed to have zero values. The length of the returned sound is the length of the longest sound. The parameter c gives the length of the s and w arrays.

4. In this part you will create an amplitude modulator where the amplitude of one signal controls the amplitude of another signal. While this sounds complicated, it is really quite simple as you just multiply point by point two sound signals together. (Note that this is

how AM radio works but at much higher frequency using electromagnetic waves.) The prototype for this function is:

```
sound * modulate(sound *s1, sound *s2);
```

Note that this function returns NULL indicating an if the two sounds are not of the same length.

5. In this part you will create a digital filter that will be used to create lots of effects. You do not need nor are required to understand the math behind this or why it works. The prototype to the function is

```
sound * filter(sound *s, float fir[], int c);
```

The processing is performed by summing samples by real numbers in an operation called convolution. Let output be the samples returned in the sound structure.

output[i] = the sum of s[i-j]*fir[j] for each j between 0 and c-1.

In the above calculation, if i-j < 0 then fir[i-j] is assumed to be 0. To check if you understand what this is doing, consider the fir array with all the entries of zeros except for location 0 where fir[0] = 1. The output generated by this is exactly the same input sound. Another interesting FIR filter is to set all values of fir to 1/c. This takes the average of the last c sound samples and as the output at time (location) i. If you think about this, it will smooth out quick changing samples. This is a filter that passes only lower frequencies and attenuates higher frequencies of sound.

6. In this part, you will create a digital reverb filter. (Reverb is the sound you hear inside a large room, and you can think of it as a really short echo.) The prototype for the function is

```
sound * reverb(sound *s, float delay, float attenuation);
```

This is computed using the filter function with the fir array set to all zeros except for two locations. The first location in the fir array is zero and its value is set to 1. (Note that this just transfers the original sound sample to the output sound sample.) The second array location in fir that is not zero is determined by how far in the past to add in a past sample. This is given by the delay argument. To determine this location, you must determine how many samples in the past represents the delay. Choose the number of samples that is closest to this number. Thus, the second location in the fir array is this location (since indexes in the array represent past sample times) and has a value of the attenuation parameter. Here is an example:

Suppose you call reverb with reverb(s, 0.05, 0.2) with the sample rate of s of 44100 samples per second. Then the locations to have non zero values in the fir array are 0, and

0.05 seconds * 44100 samples per second = 2205 samples. Thus the fir array must be at least 2205 in size. Hence, the fir array is all zeros, except at location 0 where it has a value of 1, and at location 2205 where it has a value of 0.2.

The reverb function should check for sane values of the of attenuation (less than 1 and greater than 0), delay (greater than 0 and less than 0.1 seconds) and that the pointer to sound is not NULL.

You are required to use the filter function to compute this. While this is not the most efficient way to do this, it does allow for flexibility in the way that reverb is computed. (There are other reverb algorithms that would utilize the entire fir array with non zero entries.)

Jim will be happy to discuss this part of the assignment in class. If there are questions, please ask.

7. In this part you will create an echo function similar to the reverb function, except that the delay parameter is between 0.1 and 1 second.

   `sound * echo(sound *s, float delay, float attenuation);`

   For this function, you do not need to call the filter function and you may directly compute the output from the sound.

8. In this part we will make a song player that reads a file to output a file of samples that if played produces the song. This part is significantly longer than other parts and will be worth more credit. We begin with the song file format which is a text file with the following information in the prescribed format below. The first part of the file contains definitions for sound waves. The first line of the file must begin with the word "SAMPLERATE" with the next line giving the sample rate for sound output. There can be any number of sound waves defined after the SAMPLERATE with each definition beginning with the keyword "WAVE" The next three lines after the WAVE keyword contain three parameters, the name of the wave, the type of the wave, and the parameters for any reverb. All three parameters must be present. The wave name may be any length up to 255 characters and contains only alpha-numeric characters. The wave type may be either "sine", "square" "triangle" or "saw". The parameters for the reverb line contain two float numbers, the delay and attenuation values.

   Example:

   SAMPLERATE
   16000

   WAVE
   mywave1
   saw
   0.02 0.25

WAVE
mywave2
sine
0.01 0.01

You may have any number of WAVE definitions. Note that characters between the end of a wave definition and the next keyword may be ignored (whatever they are).

The next section defines sounds which are combinations of waves mixed together. This is done by the "SOUND" keyword followed a single line that defines the name of a sound. Following the sound name are any number of wave names followed by the level to mix that wave into the final sound. Each wave name and mix parameter is specified one per line with no intervening blank lines. The last wave and mix definition is followed by at least one blank line. Sound names are again only alphanumeric and limited to 255 characters. The following is an example of SOUND section.

SOUND
mysound1
mywave1 0.4
mywave2 0.3
mywave3 0.7

SOUND
mysound2
mywave1 0.9

SOUND
mysound3
mywave2 0.7
mywave1 0.2

The beginning of the last section of the file is labeled with the keyword SONG. In this section lines of text give the time, duration and note that is played in no particular order. Any number of lines may be present with blank lines ignored. Each line must contain the following items.

<soundname> <key number> <time> <duration>

The key number notation is taken from
https://en.wikipedia.org/wiki/Piano_key_frequencies

which also tells you how to convert the key number to an actual frequency. Note that the notes are not required to come in any order, and some notes may overlap other notes in time. A good implementation would sort each of the lines based on the starting time. You may use your own sorting function, or any sorting function you find in the standard C libraries.

For example:

```
mysound1   52  0.0   0.1
mysound1   54  0.2   0.1
mysound1   56  0.3   0.1
mysound2   57  0.4   0.1
mysound2   59  0.5   0.1
mysound3   61  0.6   0.1
mysound3   63  0.7   0.1
mysound1   64  0.8   0.1
```

plays a simple scale in the key of C. Please note that we, (the TAs and myself) and likely other students will post files of recognizable songs for testing.

The input and output file names are given as command line parameters to the program which is named "playsong" For example:

```
playsong  mysong.txt  songsound.txt
```

Please place the main function in playsong.c. If you have helper functions, they may go in this file, or other files named as you wish, especially if it makes sense and thesed helper functions are used in other places.

Hint: Because there are an arbitrary number of waves, sounds, and song notes, you might consider using a linked-list where applicable. You may also consider a resizing array. Be aware of the possibility of memory leaks.

9. Update the Makefile to include a default target of "all" The intent of this target is to build all the files defined in part c, efficiently, i.e, do not compile files unless they need to be.