Network Synthesizer

CS32 Assignment 4

Due Dates:

Assignment Out:	Mar. 12, 1998
Algorithm handin:	Mar. 19, 1998 (11:59pm)
NetSynth Due:	Apr. 2, 1998 (11:59pm)

Introduction

In this assignment, you will NOT have to work with DCOM, swilly-network layer extraordinare, but instead use sockets. Rejoice!

For your fourth and final required assignment in CS32, you will continue to perfect your mastery of Motif/Baum GUI design. You'll also get to work with networking, as previously stated. Perhaps most excitingly, you'll work with audio - yes, you too can produce sounds from the Suns! - and so you'll get to learn a thing or two about how music and sound are produced electronically.

This is a networking assignment. Like Logo, you will do the assignment with a partner. This means one person will write a client, and the other will write a server. The idea is simple: at a basic level, the client takes a special datafile in the .Music format (specified elsewhere in this handout) and sends it over to the server application, which then parses the music file. The user of the client can change the volume of each of the different "channels" of sound as well as change the sounds the synthesizer will use to produce music. When the user of the client is satisfied with the levels and sounds chosen, a button is pressed and the server produces an audio file. The server lets the client know what the file is and the client plays the file. This means that you can consider the client program to be the selector/player, and the server program to the the mixer. As the user of the client, you get to choose how loud you hear the drums or whether you want that saucy melody to be played with a piano or a harmonica. The server then produces audio to match the choices the user made with the client application. We understand that this assignment may seem a bit contrived. But the point is to teach sound, networking, and GUIs, not to create a "useful" application.

Please take the time to read this assignment handout in detail. We're glad to help you with things that don't make sense, but if you don't read it, very little about this project will make sense. For this assignment, you will need headphones to test your code, so bring them with you to the Sunlab.

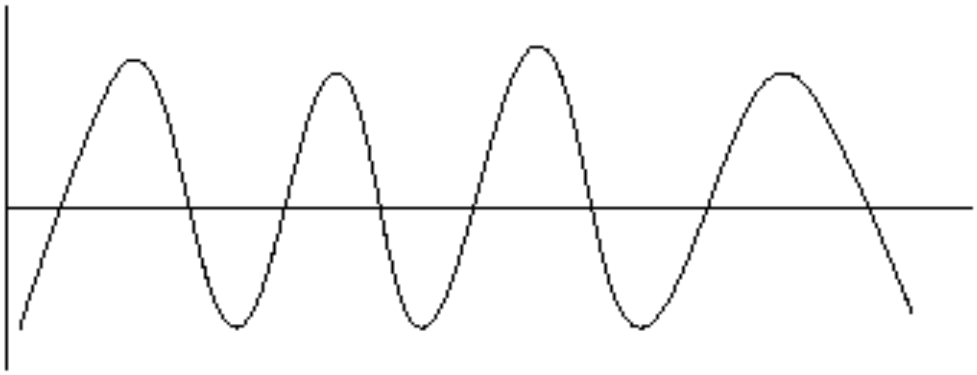
This is a two person assignment. You should try to work with the same person you worked with when you wrote Logo, but this is not required.. If you wrote the Logo Parser, you'll be writing the NetSynth client for this project; if you wrote the Logo GUI, prepare to code the

NetSynth server. The client will involve GUI and network programming, while the server will involve some network programming and audio processing.

All About Audio

You're going to have to write a synthesizer, which means you're going to have to know the basics of how audio is produced in the digital world. In this section, we'll teach you about audio from the ground up. While only the coder of the NetSynth server will have to deal with the information in this section directly, it would be wise if all parties involved read this section. Think of it as an opportunity to learn something interesting and new - with apologies to those who have encountered this information before.

At the basic level, what we percieve as sound is nothing more than compression waves: the air around us is literally becoming a bit more "compressed" and a bit less "compressed" at a particular rate, or frequency. We can describe this action of compression/decompression as a simple wave with respect to time.



There are three basic qualities of a sound wave that humans are able to percieve.

The "frequency" of the wave is the rate at which it repeats itself, and is measured in cycles per second, also known as Hertz. (Hz.) In the real world, the frequency of a wave corresponds to its pitch - that is, what note you hear. For example, if you went to a piano and played the "C" key, the wave produced by the piano would have a different frequency than the wave produced by pressing the "E" key. This would also have a different frequency than the wave produced by pressing the "E" key an octave higher or lower.

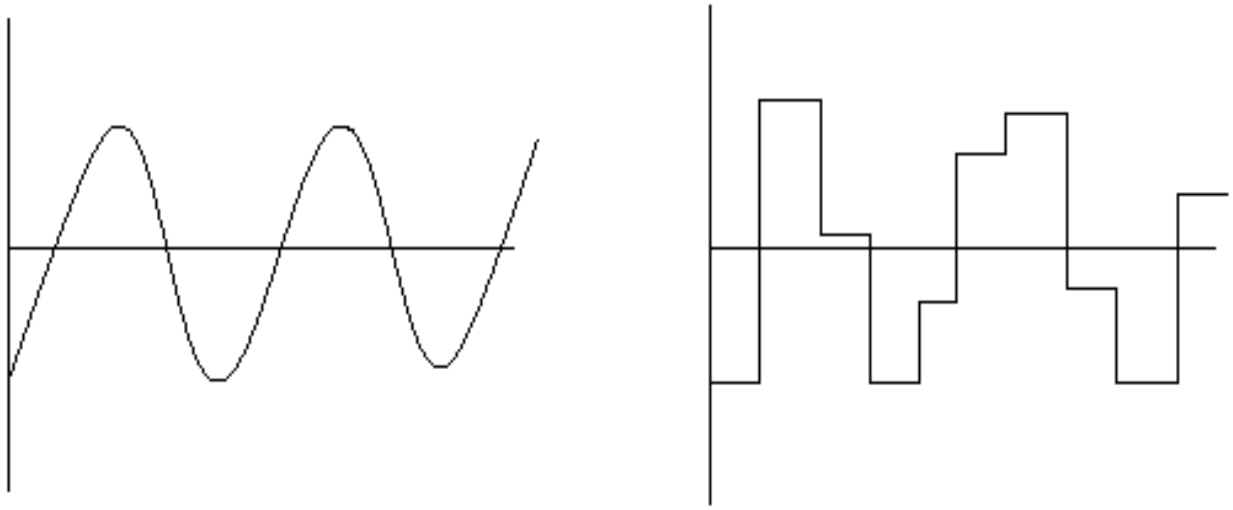
The "amplitude" of the wave is how high or low the signal gets at any one point in time. This directly corresponds to the human perception of loudness.

The "tambre" of the wave is its actual shape with respect to time. For example, in the figure above, we've drawn a sine wave. When you hear a sine wave, it sounds smooth - it isn't grating. However, we could easily force you to listen to a pulse wave (shown below) - which sounds harsh and edgy. Acoustic instruments such as pianos and guitars have extremely complex waveforms which change over time, and thus they have interesting tambres.

It is important to understand that there is a direct relationship between frequency and the pitches of the standard twelve-tone scale, unless you're Dan Gould playing the cello. (The twelve-tone scale is the scale you find when you play a piano. You know - A, A#, B, C, C#, D, D#, E, F, F#, G, and G#.) Each step up in the scale is called a half-step. As it turns out, A4 - that is the note

A in the fourth octave - is exactly 440 Hz. The frequency for the same note an octave higher is exactly double the frequency of the original note. Thus, A5 corresponds to 880 Hz, and A3 corresponds to 220 Hz. It just so happens that you can get the frequency of a note that is one half-step up from any other note by multiplying the original note's frequency by the twelfth root of two. ($2^{1/12}$ = 1.0594630943593) From this information, it is easy to determine the frequency of any note in any octave.

This is all fine and good, but the information above corresponds to (gasp!) the real world - and not the digital world of computers. How do computers digitally represent these analog waveforms? The key technique is that computers "sample" sound. Basically, the computer takes the height of a sound's waveform at any given point in time and stores it as a number. Since we are dealing with digital information, we can only sample the waveform at discrete intervals, which means that if you don't sample a waveform often enough and with enough accuracy, your digital representation of the sound will be very poor. (See the figure below.)



Your CDs are digitally sampled sounds. This means that Audio CDs contain digital samples of actual analog waveforms. When the CD player goes to produce sounds, it uses a special piece of hardware called a digital-to-analog converter to take the samples and produce an actual sound wave. Take EN/163 if you want to build such a device. They're soooooo special.

As it turns out, CDs are sampled at 44.1 kHz. This means that 44,100 times a second, a new sample was taken by whatever digital device was creating the CD. Note that this does *not* affect the frequency of the sound that is being sampled. CDs are sampled at 16-bits per sample. This means that, while sampling, we can only store 2^{16} disctinct values for the amplitude of the wave. (For those who are EE types, the CD player uses capacitors to smooth the output, and other cool stuff.) In addition, CDs store stereo data, which means they have separate waveforms for both the left and right speaker channels stored on them.

Your choice of the sampling rate and sampling bit-depth greatly affect the sound quality of your sample. If you choose a smaller bit depth, you introduce "quantization noise" into your sound. For example, if CDs were recorded with two-bit sound, they wouldn't have a wide dynamic range. (In fact, they'd sound like crap.) Your choice of sampling rate actually affects the sound quality in a more complicated manner. Humans can hear sounds with frequencies between

20 Hz and 22kHz, approximately. There is an important result in signal processing called the Nyquist Limit, which states that if you wish to accurately represent sounds with maximum frequency *f*, you must sample your sound at a frequency of at least 2*f*. If you don't, you will get strange sounding artifacts known as aliases. (This concept shows up everywhere - from neurosciene to computer graphics. Take CS/123 if you want to learn more about it.) This is why CDs are sampled at 44.1kHz.

On the computer, there are many different datafile formats for audio data. Some of the most common are .AIFF (Audio Interchange File Format) and .au (mew-law). On the Sun Workstations, the .au file is most prevelant. You can play a .au file by simply typing "audioplay mySound.au" at the command prompt. AU files contain a header which specifies the format and audio data which may be compressed. Of course, files don't have to be stored with a special scheme - you can actually just store what is known as linear audio data, in which each byte of the file corresponds to a new amplitude. Eight-bit Linear audio data is stored in two's complement. It can be easily converted into the .au format by using the "audioconvert" command, as follows:

```
audioconvert -i format=raw,rate=8k,encoding=linear8,mono
                -f ulaw,rate=8k,mono [linear filename] > [file.au]
```

For this project, your synthesizer will create linear datafiles and then return the name to the client, which will open and play them with the audio class provided.

Now that you know a bit about audio and digital representaitons thereof, let's talk synthesizers. By synthesizers, we mean those big things with piano-like keyboards that produce lots of techno music (when used by people like Dave Peck).

Most synthesizers nowadays are digitally based. This means that, internally, they have tons of small sampled sounds which they use as the basis for their audio production. Synthesizers with good piano sounds generally have good piano samples built into them, and then they use their filtering and effects hardware to embellish the sample. (Actually, good synthesizers do more than just embellish. They truly distort the sound in a multitude of interesting ways. Buy a K2500 to see what I mean. It comes highly recommended to anyone with tons of cash to spare.) But again, I digress. Naturally, the waveform samples inside synthesizers have a particular pitch (note) associated with them. On the K2500, all samples are sampled at A4. This is important to know, because when you want to reproduce the waveform at A5, you have to scale it appropriately. (By scaling, we mean that you have to create a new waveform which has the same basic shape as the original, but which has twice the frequency.)

The Music File Format

For this project, we'll give you some sample music files to use. They have a very basic file format which should make writing parsing code a breeze. This means that if your parsing code takes up more than three or four screens, you're making things hard on yourself.

The music files specify everything required to produce an entire song. They specify what notes to play, when to play them, how loud to play them, and what waveforms to use when playing the notes. Notes get played on "channels," of which there can be no more than eight. We use channels so that we can produce more than one note at the same time. On each channel, at any point in time, only one note from any waveform may play. However, over time, notes from

different waveforms may play on the same channel. Since there are a maximum of eight channels, at any point in time there may be no more than eight notes playing from any selection of waveforms. In addition to a volume for each note, each channel has its own master volume, which is called the mix level for that channel. If a channel's mix level is zero, this means that regardless of how loud the notes on that channel are, the channel produces no overall audio.

In addition to synthesizing the music into a final audio file, it is the job of the server to load and parse the music datafile. To make things easy, you can assume that there are no errors in the datafile. Of course, you will have to handle excess blank space and comments, which in this file start with //. Comments will only appear on lines by themselves.

Let's take a look at the various keywords you will find in a music datafile.

Keyword	Paramaters
WAVE	<name> <waveform's note frequency> <sample file>
NWAVE	<name> <waveform's note frequency> <sample file>
CHANS	<number of channels in file>
BPM	<beats per minute>
VOL	<channel number> <mix level>
N	<channel number> <note> <octave> <amplitude> <duration> <wave name>
S	<channel number> <duration>
LOOP	<number of times> <loop name>
ENDLOOP	<loop name>
ENDALL	
CATCH	<catch up channel> <catch up to channel>

The WAVE and NWAVE keywords specify that the music to be produced will be using a particular waveform. We've sampled about sixty different sounds to use in the final audio production; they're located in `/course/cs032/lib/samples`. (They are all stored as eight-bit linear data, so to listen to them you'll have to use audioconvert.) The waveform <name> parameter is used later on in the music file, and simply serves as a convenient way of representing a particular waveform. The next parameter, <waveform sample frequency>, is a double and represents the note at which we sampled the given waveform. The final parameter, <sample file> specifies the name of the sample waveform data file. The difference between the WAVE and NWAVE is that NWAVEs get played once per note event, and don't loop. WAVEs, on the other hand, loop if necessary. (This is explained in detail below.)

The CHANS keyword specifies the number of audio channels that this particular piece of music will require. Each channel can have its own set of notes played onto it, and each channel has its own master volume. This means that when you're producing audio, you have to keep the audio for each channel separate until you're ready to mix them all together into the final file.

The BPM keyword specifies the beats per minute for the music. This is an integer value which specifies how fast the music should play. It only makes sense when used in conjunction with the <duration> parameter of the N and S keywords, described below.

The VOL keyword sets the mix level for a particular channel. Note that if there are six channels in a file, they are referenced with <channel number> values from zero to five. The <mix level> is an integer value from zero (no volume) to 255 (maximum volume). The VOL command will appear exactly once for any given channel.

The N keyword specifices that a note should be played. The <note> parameter is one or two characters specifying the actual note in the twelve-tone scale. (i.e. A, D#, or B- for B flat. A note that is flat is one which is one half-step down from the note above it. This means that B- is the same as A#.) The <octave> parameter is an integer specifying which octave to play the note at. Remember, A4 corresponds to 440 Hz. The <amplitude> parameter is an integer specifying the loudness of the note, and it ranges from zero to 255. The <wave name> parameter is the name of the waveform to use; this corresponds to the name which was specified with a previous WAVE or NWAVE command.

The S keyword specifies that there should be silence on a particular channel for a particular <duration>.

The <duration> parameter to both the N and S keywords is a double value which, when used in conjunction with the BPM value for the music file, leads to the number of seconds which a note should play for. The actual way to convert a <duration> value into a length in seconds is to use the following simple equation:

lengthInSeconds = ((4 / duration) * 60) / beatsPerMinute;

For those who are familiar with the music parlance, this allows us to easily specify a duration of 4 as a quarter note (i.e. if BPM=60, a duration of 4 corresponds to exactly one second, which is what you'd expect.) It follows that a duration of 2 is a half note, and a duration of 8 is an eighth note. For those not familiar with the music parlance, just use the equation above and everything will turn out spiffy-keen.

When you see a N or S keyword, this is your cue to generate an appropriate waveform of an appropriate length and append the waveform data onto the given channel's audio data. Note that the N and S keywords don't specify *when* a particular note gets played - they just specify *how long* it will be played for. This means that each N and S is assumed to take place directly after the previous one on that channel.

The LOOP keyword assists music programmers in creating portions of music which repeat themselves, such as drum loops or simple melodies. The <number of times> parameter specifies - you guessed it! - how many times a particular loop should be performed. LOOPs can be nested inside of each other, and so the <loop name> parameter is a single string (without spaces!) to specify the name of the LOOP.

The ENDLOOP keyword specificies that a particular loop has ended. The <loop name> will correspond to a previous LOOP command's loop name. Everything between the LOOP and corresponding ENDLOOP should be repeated the appropriate number of times.

The CATCH keyword is used to help out musicians as well. (I didn't want to throw this into the spec, but I really needed it badly. Sorry, all.) It takes two channel numbers as parameters. If the number of bytes of audio currently generated in the first channel is less than the number currently generated in the second channel, you fill the first channel with silence until the two channels are the same length. (Silence corresponds to a linear audio value of zero.) This has the effect of synchronizing the two channels with each other.

Finally, the ENDMUS keyword appears at the end of a sample data file, and this means that you can stop parsing the file. How cool is that?

Client Requirements And Hints

Okay, so now that you know about how sound works, and how we're going to specify music data, let's talk about what you're actually going to do.

The client must perform the following functions:

1. Present a useable GUI interface (Motif / Baum) which allows the user to
    a. Change the mix levels for any channel of audio
    b. Mute and unmute each channel of audio
    c. Play the audio for one particular channel.
    d. Select the waveforms in the music file, listen to them, and change them.
2. Communicate with the server (i.e. use the network class)
    a. Tell it to parse a file
    b. Tell it to mix sounds (either the entire music, or a channel)
    c. Get information about the number of channels, number of waveforms, and their names.
3. Play the audio
    a. Use the Audio class - to be described in Appendix A.
    b. You should be able to coordinate with the server to play a channel, the entire sound, or a sample (which doesn't require server communication).
    c. Display the waveform of the music, and the current position in the waveform (see the demo, or `/usr/openwin/bin/audiotool`, for inspiration)

The client *must not* parse the file. It should leave this task up to the server. The client also should not mix the audio. Again, this task is up to the server. The client *should* play the audio data that the server generated, however.

A suggested interface for the client is found on the last page, but of course, this is your project! Keep in mind that GUI coding lends itself towards huge functions which do everything. Try to keep OO-design in mind, because huge functions are a pain to modify, comprehend, and debug.

Server Requirements And Hints

The server must perform the following functions:

1. Parse the music datafile.
    a. Keep track of waveforms, channels, and channel levels
2. Generate music
    a. The entire music file, or a channel.
3. Communicate with the client

Here are some things to keep in mind while writing the server. There is nothing wrong with maintaining temporary data files - just be sure to clean up after yourself. Of course, you could also

use pipes, or anything else your perky little heart desires. Second, remember that your linear datafiles will have to be in two's compliment - i.e. use a signed int to store a sample value. (Using signed ints also makes it really easy to change the amplitude of your sound - just multiply!) Fourth, keep in mind that your output is at 8kHz. This means that if you want to generate a one-second long tone, you'll have to output exactly 8,000 bytes. The general calculation follows from this.

Scaling waveforms to the correct frequency isn't that hard. First, calculate the number of samples you want to output. Next, allocate enough memory for the new data. Enter into a loop. Grab data from the original sample waveform and put it into your new buffer. You'll want to skip entries in the waveform sample in order to get the correct frequency in your new buffer. For example, if I give you a waveform at 440 Hz and tell you to reproduce it at 880 Hz, you'll want to look at every other entry in the original waveform's wavetable. If you hit the end of the wavetable, loop back to the beginning, unless the waveform was specified NWAVE, in which case just produce silence for the remainder of the time.

Things get a bit trickier if I give you a waveform at 440 Hz and ask you to reproduce it at 520 Hz. While you are *not* required to perform linear interpolation, you have to come up with a clever way of finding the sample nearest the one you want. Think floor, or (even easier!) typecasting from double to int.

This kind of code appears frequently, and is seen in many different realms of science, so it is to your benefit and credit to puzzle through it by yourself. Naturally, TAs will be glad to help out, too.

Turning In The Assignment

We are not going to require a design handin for this assignment. Instead, we are requiring that you turn in an algorithm handin. The handin should describe the complex sections of your code. It should detail how you plan to modify the samples to produce the desired sounds, and then mix the music together. It should also describe how the client will display the waveform, and the current position in it. There should be one handin per group.

When you and your partner are done, handin the assignment with:

cs032_handin audio

There are no design due dates or handins, but if you don't do a design, you may find it difficult to complete this assignment on time. So for your own benefit, sit down with your partner and figure out how it is all going to fit together before you start coding.

Appendix A: Support Code

The Audio class. You can find the class header in `/course/cs032/include/audio.H`

```
typedef class AudioInfo *             Audio;

class AudioInfo {
public:
  AudioInfo();
  virtual ~AudioInfo();

  /* Begin playing sample - you must have set a sample, or this will have
     no effect */
  void play();

  /* Stops playing the sample.   Obviously, this only works if the sample
     is playing */
  void stop();

  /* Pause playback.  This will only work if the audio is playing at
     the time */
  void pause();

  /* Unpauses playback. This only works if the output is currently paused */
  void unPause();

  /* Returns the current position in the sample */
  int getPosition() const;

  /* Sets the start position in the sample */
  void setStartPosition(int start);

  /* Sets the end position in the sample */
  void setStopPosition(int stop);

  /* Sets the current sample.  The audio class will make a copy of the
     buffer passed, and free it when the audio class is deleted. */
  void setSample(const char * sample_data, int sample_length);

};
```

Appendix A: Support Code

The Connect class. You can find the class header in /course/cs032/include/connect.H

```
typedef class ConnectInfo *             Connect;

class ConnectInfo {

private:
   ConnectSocket conn_socket;

public:
   ConnectInfo();
   virtual ~ConnectInfo();

   static XtAppContext setX11Context(XtAppContext ctx = NULL);

   int openServer(const char * file);   // open a server socket
   int openClient(const char * file);   // open a client socket
   int openAccept(Connect);             // open a server, via accept
   void close();                        // close a socket

   void sendMessage(int len,const void * msg);
   void sendMessage(const char * msg)
      { sendMessage(strlen(msg)+1,(void *) msg); }

// These callback methods should be redefined by a subclass.
public:
   virtual void acceptCallback();
   virtual void closeCallback();
   virtual void messageCallback(int len,void * msg);
};
```

The SimpleSocket class. You can find the class header in
`/course/cs032/include/simple_socket.H`

```
typedef class SimpleSocketInfo *        SimpleSocket;

class SimpleSocketInfo {

public:
   typedef int Error;

private:
   int socket_id;

   static Error last_error;

public:
   SimpleSocketInfo();
   virtual ~SimpleSocketInfo();

   bool create();                          // create a socket
   void close();                           // close the socket

   int send(const void * buf,int bln);     // send a message
   int receive(void * buf,int bln);        // receive a message

   bool bind(int port = 0);                // bind socket to local port
   bool getSockName(char * host,int& port);
   bool enableNonblocking(bool);           // enable/disable nonblocking socket
   bool listen(int n = 5);                 // allow socket to listen
   bool accept(SimpleSocket);
   bool connect(char * host,int port);     // client connect to server

   Error getLastError()                          { return last_error; }

public:
   virtual void onAccept();                // accept available on the socket
   virtual void onClose();                 // other end closed the socket
   virtual void onConnect();               // connection ready on this socket
   virtual void onReceive();               // socket has data to read
   virtual void onSend();                  // socket ready to send

public:
   void waitForEvent();

   static XtAppContext useX11(XtAppContext = NULL);
};
```