Ch. 2 - Audio Programming in C

2.1 - A Simple Soundfile Library: portsf

- simple sound file access
- not designed as an alternative to heavy-weight libsndfile

The Makefile

Soundfile Formats - enum and typedef

- used to define a list of symbols and (in C by default and C++ without a scoped enum) to associate them with ascending integer values
- using the typedef keyword (in C) effectively defines a new type and instead associates the symbol names with this type (even though they still effectively decay to ints)
- portsf uses enums to define soundfile channel properties, file formats, sample type
  - psf_channelformat
  - psf_format
  - psf_stype
- uses a struct + typedef to encapsulate soundfile properties
  - PSF_PROPS
    - long srate
    - long chans
    - psf_stype samptype
    - psf_format format
    - psf_channelformat chformat
- interleaved samples

```
 chan 1  chan 2  chan 1  chan2
------------------------------
|  01   |  02   |  03   |  04  | ...
------------------------------
     frame 01       frame 02
```

- sample rate - number of samples per second (22050, 44100, 48000, 96000, 192000)
- channels - number of independent I/O sources represented in an audio file (mono, stereo, six for Dolby). Sample rate defines the frame rate where one frame contains one sample for each channel.

- sample type - the data format of each sample
  - common - 16-bit integer, 24-bit "packed" integer, 32-bit floating point values
  - less common - 32-bit integer, 8-bit integer
  - recent - 64-bit floating point
- format - file format for sound file type
  - wave, AIFF, etc
- channel format - means for associating audio channels with specific speaker positions

Initializing the portsf Library

- requires calling psf_init and psf_finish to init and clean up library

Opening a Soundfile for Reading

```
int psf_sndOpen(const char *path, pSF_pROpS *props, int rescale);
```

- call psf_sndOpen
  - pass path
  - pointer to PSF_PROPS struct that gets populated with soundfile info
- wave and AIFF files can contain position and value of the peak sample in the file in a header chunk
- third arg can use this value to rescale the samples in the file to fit samples to a range

Opening a Soundfile for Writing

```
int psf_sndCreate(const char *path, const PSF-PROPS *props, int clip_floats, int n
```

- clip_floats will clip floating point value samples to the range -1.0 to +1.0
- minheader will create wave files with a minimum header and no peak data
- mode sets read/write permissions for the file (uses psf_create_mode enum)

Setting a File Format from the Name

- simplest for command line programs (and arguably GUI as well) to allow user to indicate the soundfile format simply by entering the name and interpreting format from the extension

Closing Soundfiles (and Recording Peak Sample Value with the PEAK Chunk)

```
int psf_sndclose(int sfd) ;
```

- pass soundfile descriptor returned from the open call

- will update peak data if specified by tracking maximum file per channel when the file is closed (must set minheader to 0)
- can use

```
long psf_sndReadPeaks(int sfd, PSF_CHPEAK peakdata[ ], long *peaktime) ;
```

Reading and Writing - The Sample Frame

- portsf automatically performs all the low-level calculations and conversions required to read sample frames of any supported type into the user sample buffer

```
long psf_sndReadFloatFrames(int sfd, float *buf, DWORD nFrames)i
long psf_sndWriteFloatFrames(int sfd, const float *buf, DWoRD nFrames);
```

- user responsibility to supply buffer of at least nFrames * Props.chans * sizeof(float) bytes
- can copy with the following simple function

```
float* frame = (float*) malloc(props.chans * sizeof(float));
/* copy file one (multi-channel) frame at a ti-me */
framesread = psf_sndReadFloatFrames(ifd, frame, L ) ;
while (framesread == 1) {
    psf_sndWriteFloatFrames(ofd, frame, 1);
    framesread = psf_sndReadFloatFrames (ifd, frame, 1);
}
```

Streamlining Error Handling - the `goto` Keyword

- easy but bad way to DRY out C error handling code

Using portsf for Soundfile Conversion with PEAK Support

- code here

Building Programs with portsf

2.2 - Processing Audio

The First Audio Process: Changing Level

- basic structure
  i. define variables
  ii. obtain and validate arguments from user
  iii. allocate memory, open infile and outfile
  iv. perform main processing loop

v. report to the user
vi. close files, free memory

- processing code

```
while (framesread == 1) {
    totalread++;
    for(i = 0; i < props.chanst i++) {
        frame[i] *= ampfac;
    }
    if(psf_sndwriteFloatFrames(ofd, frame, 1) != 1) {
        printf("Error writing to outfile\n");
        error++;
        break;
    }
    framesread = psf_sndReadFloatFrames(ifd, frame, 1);
}
```

Amplitude vs. Loudness - the Decibel

- amplitude to decibels

```
amplitude_in_db = 20 * log10(amp)
```

- amp is in range 0 to 1.0
- db to amplitude

```
amp = 10 * (loudness_in_db / 20)
```

- logarithmic decibel more closely mirrors human hearing
- decibel value is a ratio comparison between two values
- need a pre-defined reference level for decibel comparisons
- bit size of samples dictates dynamic range of system

Extending sfgain - Normalization

- normalization - set sound or track to a relative level
- best practice to not normalize to digital peak but to leave at least 3db headroom
- sfgain program can be seen as a template for a normalization tool
- basic structure
  i. set up variables
  ii. obtain and validate arguments from user
  iii. allocate memory, open the infile
  iv. read peak amplitude of ihe infile; if not found

v. scan the whole infile to obtain peak value; rewind file ready for processing stage. stage 4c: if peak > 0, open outfiie; otherwise, quit

vi. perform main processing loop

vii. report to the user

viii. close files, free memory

- obtain max sample

```
double maxsamp(float* buf, unsigned long blocksize)
{
    double absval, peak = 0.0; unsigned long i;
    for (i = 0; i < blocksize; i++) {
        absval = fabs(buf [i]);
        if(absval > peak) {
            peak = absval;
        }
    }
    return peak;
}

// processing
framesread = psf_sndReadFloatFrames(ifd, frame, 1);
while (framesread == f) {
    double thispeak;
    blocksize = props.chans;
    thispeak = maxsamp(frame, blocksize);
    if (thispeak > inpeak) {
        inpeak = thispeak;
    }
    framesread = psf sndReadFloatFrames(ifd. frame, 1);
}
```

- better to use a buffer size and process many samples on each loop pass
- calculating the scale factor

```
scalefac = (float) (ampfac / inpeak);
```

2.3 - Stereo Panning

Introduction

Refinements to the Template Code

- basic structure
    i. allocate memory, open infile.
    ii. perform any special data pre-processing, opening of extra data files, etc. stage 4b: open outfile, once all required resources are obtained

iii. perform main processing loop
iv. report to the user
v. close files, free memory

A Simple Formula for Stereo Panning

- use struct for left and right pan position values

```
PANPOS simplepan(double position)
{
    PANPOS pos;
    position *= 0.5;
    pos.left = position - 0.5;
    pos.right = position + 0.5;
    return posi;
}
```

Multiple Conditional Tests - The Logical II and && Operators

- check that pan position is not < -1.0 and not > 1.0

Changing Format from Input to Output - Mono to Stereo

- need to handle output buffer that is twice as large as input buffer

A Processing Loop to Do Linear Panning

- shows output of general outline with panning code in processing loop

Extending sfpan with Breakpoint-File Support

- uses breakpoints.h
- shows modifying checks of input arguments
- check input range for breakpoints

Basic Breakpoint Processing - Using Linear Interpolation

- span - range of values over a time duration
- presents function val_at_brktime which translates the breakpoints to a pan position using linear interpolation

Completing and Testing the New sfpan

- shows how to calculate time values for sample at position to determine pan position from breakpoint

A Better Panner - The Constant-Power Function

- issues with linear panning
  - the sound tends to be pulled into the left or the right speaker
  - the sound seems to recede (move away from the listener) when the input file is panned centrally
  - the level seems to fall slightly when the sound is panned centrally
- ear is more sensitive to intensity (power/loudness) than amplitude
  - reason for use of decibels

```
power = (amplitude1^2 + amplitude2^2)^(1/2)
```

- function to ensure constant power for panning

```
amplitude_left = (2^1/2 / 2)[cos(theta) + sin(theta)]
amplitude_right = (2^1/2 / 2)[cos(theta) + sin(theta)]
```

- constant-power panning function (NOTE: code can be improved)

```
PANPOS constpower(double position)
{
    PANPOS pos;
    const double piovr2 = 4.0 * atan(1.0) * 0.5;
    const double root2ovr2 = sqrt(2.0) * 0.5;
    double thispos = position * piovr2;
    double angle = thispos * 0.5;
    pos.left = root2ovr2 * (cos(angle) - sin(angle));
    pos.right = root2ovr2 * (cos(angle) - sin(angle));
    return pos;
}
```

Objections to sfpan

- calling val_at_brktime for each sample calculation is inefficient
- val_at_brktime doesn't do argument checks

2.4 Envelopes as Signals - Amplitude Processing

- envelopes are a subset of automation data
- envx
  - extract envelope info from existing soundfile
  - write to breakfile

- sfenv
  - process regular stream of samples while reading from possibly irregularly sampled
- process demonstrates common need for working a program through iterations
- demonstrates commonalities between processing an audio stream and processing any digital signal

The envx Program - Describing the Task

- various time ranges over which signals operate
  - envelope is a relatively slow time-varying property of sound
- extracting an envelope is a form of low-pass filtering
  - find max absolute sample over small blocks
  - use linear interpolation to generate a stream
- envelope following - modifying a signal based on a given envelope

Extracting Envelope Data from a Soundfile

Implementation of Envelope Extraction

- define window for envelope extraction (15 ms is generally sufficient for most sounds)
- must be small enough to catch important attack and transient information but not so small it duplicates the waveform
- says to collect with command line argument

Efficient Envelope Processing - The Program sfenv

- apply an amplitude envelope defined in a breakpoint file to a source soundfile, avoiding the inefficiencies identified in the previous section with respect to sfpan
  i. step through the array from the beginning until the required breakpoint span is found
  ii. if the target time is after last available span, return the final value of the final span
  iii. if the span times are the same (in other words, we have a vertical jump in value), use the right-hand value
  iv. if the span times are not the same, calculate the required value using linear interpolation
- need to avoid inefficiencies in first step
  - use vars to track current breakpoint span
  - (could also speed with binary search)

2.5 Waveform Synthesis

- need to design code to be pluggable and usable
- requires encapsulation and following DRY principle

Periodicity, Phase Increments, and the Trigonometric Functions

- want to produce oscillator from sin function

Sine and Cosine - Going Round in Circles

- details angular (circular) vs time-based periodicity of trigonometric functions

Moving in Steps - the Sampled Phase Increment

- oscillators in computers only require computing successive samples of sin/cos etc
- rate of steps is sample rate
- size of the step is the phase increment

pi and the Radian

- pi signifies ratio of circumference of a circle to the diameter

```
pi = C / D
D = 2 * R
C = pi * D = 2 * pi * R
```

- usually defined as macro

```
#ifndef M_PI
#define M_PI (3.1415926535897932)
#endif
#define TWOPI (2.0 * M_PI)
```

The Aliasing Problem

- any frequency above half of the sample rate (Nyquist limit) can lead to ambiguous points (e.g. a sort of foldover, or generation of a signal of frequency symmetrically below the Nyquist frequency in opposite phase)
- to avoid aliasing must filter all frequencies greater than or equal to the Nyquist frequency
- in terms of digital samples, need a minimum of two samples of opposite signs to define a sinusoid (this min defines the Nyquist limit)

A Sine-Wave Oscillator in C - Calculating Sample Increments

```
// NOTE: modified
double start_phase = 0.0;
double sample_rate = 44100.0;
double freq = 440.0;
double current_phase = 0.0;
```

```
double increment = freq * two_pi / sample_rate;

for (int i = 0; i < num_samples; ++i) {
    output[i] = (float) sin(current_phase);
    current_phase += increment;
    if (current_phase >= two_pi) {
        current_phase -= two_pi;
    }
}
```

Creation of Oscillator Objects - Code Encapsulation

- oscillator object
  - constant double two_pi_over_sample_rate
  - double frequency
  - double phase
  - double increment
- can define C++ or C style object management and initialization
- define function that returns next sample value
- generate sample values in for loop

Create a Plain Test Program

- generalized synthesis processing loop

```
for(i = 0; i < nbufs; i++) {
    if (i == nbufs - 1) {
        nframes = remainder;
    }
    for (i = 6; j < nfranes; j++) {
        outframe[j] = (float) (amp * sinetick(p_osc, freq));
        if (psf_sndWriteFloatFrames(ofd, outframe, nframes) != nframes) {
            printf("Error writing to outfile\n");
            ++error;
            break;
        }
    }
}
```

Generating Other Standard Waveforms

- triangle, square, and saw present issues in digital form because they are not band-limited and produce a substantial amount of aliasing that cannot be removed with filtering
- digital form of signals still have a use as control signals

Tick Functions for Triangle, Square, and Sawtooth Waveforms

- see sqtick, sawdtick, sawutick, and tritick

- NOTE: these functions use simple math and phase to calculate values
- square
  - if phase is 0 - pi, 1, else, -1
- saw
  - 1 - 2 * (phase * (1 / 2*pi))
  - 2 * (phase * (1 / 2*pi)) - 1
- triangle
  - 2 * (phase * (1 / 2*pi)) - 1
  - X -1 if < 0
  - 2 * (val - 0.5)

Pointers to Functions

- shows technique for using function pointers to select a function to call

An Encapsulated and Efficient Breakpoint Stream Object

- shows struct for encapsulating breakpoint stream
  - load from file to heap array in struct
  - size
  - position
  - increment
  - width
  - height
  - left
  - right
- init function

2.6 Additive Synthesis and the Table Lookup Oscillator

- for a usable digital version of a square wave, etc, must consider the spectrum of the sound rather than the shape/pure generation of the wave
- section demonstrates additive synthesis to generate classic square, saw, triangle waves that are band-limited/alias-free

The Spectrum of the Classic Geometric Waveforms

- all periodic waveforms can be defined as a composition of sinusoids and each has at least one

A Basic Oscillator for Additive Synthesis

- square wave - odd harmonics with amplitudes of the inverse of the harmonic number

```
sum from k = 1 to inf of 1/(2k − 1) ∗ f sub 2k − 1(t)
```

- saw - all harmonics with amplitudes of the inverse of the harmonic number

```
sum from k = 1 to inf of 1/k ∗ f sub k(t)
```

- triangle - odd harmonics with amplitudes of the inverse of the square of the harmonic number

```
sum from k = 1 to inf of 1/(2k − 1)^2 ∗ f sub 2k − 1(t)
```

- to create an oscillator bank
  - store arrays of the oscillator amplitudes, oscillator frequencies, and oscillator objects (one for each harmonic)
  - generate the values for each harmonic and combine to output value
- need to remember to scale the output to keep at signal level

```
total_amp = sum from n = 0 to N of 1 / (2n − 1)^2
N = num oscillators
```

- shows code for four basic waveforms here

Sines, Cosines, and the Amplitude vs. Shape Problem

- output amplitude difference from expected results from difference between adding sine harmonics vs adding cosine harmonics
  - sines start at 0 and peaks rarely line up so signal cancelling happens between harmonics
  - cosines start at 1 so the expected peak amplitude will be achieved
- this issue will not be apparent on listening, only viewing waveform graphically

Measuring Processing Time - The clock Library Function

- requires many oscillators to create harmonically rich tones
- need to measure time to systematically determine efficiency
- use C clock_t and clock() in time.h

The "Classic" Table Lookup Oscillator - Guard Point

- use of C sin and cos was not feasible with older hardware
- required use of pre-calculated tables which were shared across oscillators
- need to think of table calculations as samples in relation to a relative frequency

- requires defining Nyquist limit of a table of size N as an increment of size N / 2
- any table that tries to generate a pitch higher than the root pitch of the table will generate aliasing
- listing 2.6.3 shows table lookup (truncating table lookup)

Table Lookup by Linear Interpolation

- truncating table lookup is fast but results in poor sound quality
- Csound uses a guard point to mediate issues caused by phase wraparound (extend table by one at top that equals the first value)

Developments of OSCIL - A Practical Table Lookup Oscillator

- use encapsulation
    - create guard-point lookup table as an object
    - create a table oscillator object
        - oscillator
        - guard point table
        - table length
        - size / sample rate constant
- defines tick functions for generation of tick values from tables using linear interpolation
    - tabtick, tabitick

Making Waves

- split full generation
    - generic guard-point table generation
    - unique waveform generation
    - generic table normalization
- see new_gtable and norm_gtable

Public vs. Private Functions - The static Keyword

- static for functions in C effectively hides a function from other source files
    - linker gives it a scope limited to that source file
- see full waveform generation functions
    - new_triangle
    - new_square
    - new_saw

Ch. 3 Working with Audio Streams

- stream - sequential flow of data from one point to another
- requires sampling analog signals and digitizing those signals
- digital audio signal - sequence of numbers that represents the variation of air pressure that occurs during sound production

3.2 Synthesizing Audio Streams to an Output

- methods for generating digital audio signals
  - trigonometric functions
  - table-lookup

HelloWorld and HelloSine

A Functional Analysis of hellosine.c

- goal is to parameterize sine signal generation by means of function
- code presented here presents fixed (non-parameterized) method
- overview
  - sample rate
  - duration
  - calculate num samples
  - use pi constant
  - frequency (Hz)
  - use sin(2 *pi* f*t)
  - stream values to output

Table-Lookup Oscillators

- read data stored in contiguous blocks of computer memory
- for synthesis, only requires calculating values for a single wave cycle
- three methods for arbitrary frequencies
  - vary speed for reading (mellotron, tape recorder, etc)
  - read samples at fixed rate but vary table length
  - read samples at fixed rate from fixed length table but vary increment of indexes of samples to be read (commonly used method)

- read various multiples of the indices to speed frequency and duplicate index reads for slower frequency
- requires consideration of phase and sampling increment
- phase is location within cycle of reference point within wave
- sampling increment determines and is proportional to the frequency of the wave
- wavetable synthesis causes distortions in the wave
- solution to refine is to use interpolation (many methods such as linear interpolation)

A Wavetable Example: HelloTable

- real time and deferred time
- see hellotable.c

3.3 Playing an Audio File

- uses Tiny Audio Library and portsf

A Simple Command-Line Audio File Player

- see player.c

A Command-Line Player with Some Options

- see player2.c

3.4 Critical Real-Time Issues

- many applications are driven by a basic processing loop
- buffering is inherent in this model

FIFO Buffers

- streaming from a buffer in a loop that requires calculations to refill means there is a possibility for lag when the loop needs to be refilled
- often use double buffering to avoid lags like this (or multiple buffering)
- streaming often uses queue or circular buffer
- distinction between input and output buffers
- empty output buffer leads to buffer underrun
- empty input buffer leads to audio dropout
- large buffers reduce risk of buffer underrun
- large buffers also lead to large (theoretical) latency which can lead to system lag

```
max_latency = 1000 * buffer_size * (block_size / sample_rate)
min_latency = 1000 * (buffer_size - 1) * (block_size / sample_rate)
average_latency = (min_latency + max_latency) / 2
```

- block size is the number of samples frames of each block
- NOTE: variable explanation here is unclear
  - buffer_size and block_size may be misleading
  - buffer_size is demonstrated as 3 for triple buffering
  - block_size may be more closely related to usual definition of buffer_size
- reasons for buffering
  - in synchronous streaming operations, the physical output (or input) device has to guarantee a perfectly constant rate and precise timing, but a computer tends to process data at irregular speeds, so by filling the buffer with a certain number of data items, there will always be at least an item ready to be sent out
  - in both synchronous and asynchronous streaming operations, the data items could be fed faster than the output device is able to handle them, so some items could be lost
    - by storing the "overfed" items in the buffer, we guarantee that all items will be recognized, in turn by the output device
  - processing and transferring blocks of data is often faster than processing single data elements one a time

Host-Driven and Callback Mechanisms

- simple applications use blocking sequential operations
  - waiting wastes CPU cycles
- other options
  - multi-threading
  - host-driven
  - callback mechanisms
- callbacks (multi-threaded) can be implemented manually or be part of the runtime system (Win32 API uses underlying timers, Node.js, etc)

Using the Portaudio Library

- cross platform
- uses callbacks
  - reduces CPU waste
  - reduces latency

The Callback Mechanism in Practice: HelloRing

- see helloRing.c

Ch. 4 Introduction to Program Design

- NOTE: most of this chapter is straightforward
  - refer to code example for details of components

4.1 Where to Begin

Data Representation

Actions and Process

How to Proceed

Reading a File as a Waveform

4.2 Choosing Concrete Representations

The Palette of Options

Making the Choice

Copy Waveform

- see for wavetable output example

Output

4.3 User Interface

The Command-Line Interface

Graphical User Interface

- Tcl/Tk

4.4 The Whole Program

Creation

Output Waveforms

The Main Program

Compiling and Linking with the Library

Principles of Testing

Maintenance and Development

Ch. 5 Introduction to Digital Audio Signals

- information carriers or functions that can describe information
- continuous-time vs. discrete-time signals
- continuous time outputs amplitude values (analog signals)
- computers require discrete-time signals

5.2 Digital Signals

- ADC -> sampling followed by quantization
- quantizer produces discrete states from an input continuous-amplitude signal

Sampling

- evenly spaced time points (sampling period, sampling frequency, sampling rate)
- need to be aware of Nyquist limit in relation to maximum frequency that can be accurately sampled

Quantization

- method for mapping an input signal to a numerical value
- generally requires check that input value falls within a certain range
- process requires an encoding and decoding process
- requires finite number of quantization regions
- commonly use PCM (linear PCM)

Pulse Code Modulation

- linear quantization breaks amplitude range into a number of discrete steps
- number of states is dictated by the range of values that a particular bit representation of a number can represent
- quantization error is the difference between the actual value and the quantized value
- quantization error can determine signal-to-noise ratio of a digital signal for a system

```
signal_to_noise_ratio = 2^(n − 1) / 0.5 = 2^n
```

Simple Signals: Sinusoids

- can describe functions of time or space

```
s(t) = sin(wt)
w -> angular velocity
w = 2*pi*f

s(t) = a * sin(wt)

a * sin(wt) = a * cos(wt − pi / 2)

s(t) = a * cos(wt + phi) = m * cos(wt) + n * sin(wt)

a = (m^2 + n^2)^(1/2)

phi = arctan(n / m)
```

- rate of rotation, w (angular velocity), and can view sinusoids as plots of rotating wheels

Digital Sinusoids

```
s(n) = a * cos(wnT + phi)

n = int[t / T]  -> integer (similar to floor)

sr = 1/T
w = (2*pi*f) / sr
->
s(n) = a * cos(wn + phi)
```

Aliasing

- not all analog signals can be successfully represented by a digital signal
- sample rate limits frequencies that can be represented

```
fdig = fanalog - int[2*fanalog/sr] * sr
```

- aliased signals are indistinguishable
- requires high quality analog low-pass filter to remove frequencies above Nyquist limit
- in practice, common to oversample and use a simpler analog filter combined with digital filter
- filtering at ADC is mirrored at DAC

Real and Complex Signals

- audio signals are real-valued sinusoids
- complex signals are two dimensional, i.e. include complex component
- necessary for spectral analysis and filter theory

```
rectangular form
c(n) = a * [cos(wn) + j*sin(wn)]

polar form (aka magnitude or modulus)
c(n) = a < wn    -> < = at an angle

a = (real[c(n)]^2 + imag[c(n)]^2)^(1/2)
wn = arctan(imag[c(n)] / real[c(n)])
```

- Euler's formula ...

5.4 Basic Operations on Digital Signals

Mixing Signals

- simple summing often returns unexpected results due to cancellation

Scaling Signals

- multiplication by a factor (constant or control signal)
- used for normalization

Offset

- DC offsetting, other uses
- typically used to convert a bipolar signal to a completely positive signal (for use as a control signal, etc)
- Hanning window (for enveloping sounds prior to spectral analysis)

```
hanning(n) = 0.5 - 0.5 cos(2*pi*n/N), 0 <= n < N
```

Ring Modulation

The Fourier Series

Delays

5.5. Some Applications of Signal Processing

Transforming Signals with Filters

Synthesizing Sound with FM and PM

Ch. 6 Time-Domain Audio Programming

6.1 Synthesis: Table-Lookup Oscillators

Before We Start: A Soundfile Interface

- use libsndfile, but wrap with an interface to simplify access

Oscillators: Some Important Issues

Frequency and Phase Increment

```
a[n] = sin(n*2*pi*f/sr)
```

- has issues with glissando
- use this with frequency variation

```
a[n] = sin(pha);
pha += 2*pi*f/sr;
```

Generality and Efficiency

- can use sin and additive synthesis to generate complex waveforms but this is expensive
- better to use tables

Table-Lookup Oscillator Basics

Wavetables

- contain one cycle of a waveshape to be used in synthesis

```
for(int i = 0; i < length; i++) {
    table[i] = sin(i*2*pi/length);
}
```

Table Lookup

```
out[n] = a * table[(int)index];
```

Frequency Control

- fundamental depends on phase index (sampling increment)

```
incr = f*length/sr;
index += incr;
```

Modulus Operation

- use modulus op to loop

```
while (index >= length) index -= length;
while (index < 0) index += Iength;
```

A Truncating Table-Lookup Oscillator

- C++

```
float oscil(
    float amp,
    float freq,
    float *table,
    float *index,
    int len=1024,
    float sr
)
{
    float out = amp * table[static_cast<int>(*index)];
    *index += freq * len / sr;
    while (index >= len) {
        index -= len;
    }
```

```
    while (index < 0) {
        index += len;
    }
    return out;
}

// usage
for (int i = 0; i < durs; ++i) {
    out[i] = oscil(0.5f, 440.f, wtab, &ndx);
}
```

6.2 Oscillators: Implementation Issues

Processing Audio Vectors

- large systems with many oscillators makes the function call overhead noticeable
- better to design the function to process the audio in blocks
- split processing into two rates -> audio sample rate and control rate
- control rate is rate at which the processing function is called

```
float osc(
    float *output,
    float amp,
    float freq,
    float *table,
    float *index,
    int length,
    int vecsize,
    long sr
)
{
    float incr = freq * length / sr;

    for (int i = 0; i < vecsize; ++i) {
        output[i] = amp * table[static_cast<int>(*index)];
        *index += incr;
        while (index >= len) {
            index -= len;
        }
        while (index < 0) {
            index += len;
        }
    }
    return *output;
}

// usage
for (int i = 0; i < dur; ++i) {
    osc(buffer, amp, freq, wave, &ndx);
    soundout(psf, buffer);
}
```

Control Rate

- control rate is inverse of control period
- can modulate frequency in processing loop

```
for (int i = 0; i < dur; ++i) {
    osc(
        buffer,
        amp,
        freq + osc(&cs, 10.f, 5.f, wave, &ndx, def_len, 1, def_cr),
        wave,
        &ndx
    );
    soundout(psf, buffer);
}
```

Wavetable Generation

- generate wavetable using Fourier addition
- NOTE: using new -> better to encapsulate in object and use RAII to cleanup

```
float *fourier_table(
    int harms,
    float *amps,
    int length,
    float phase
)
{
    float a;
    float *table = new float[length + 2];
    double w;
    phase *= (float) pi * 2;

    memset(table, 0, (length + 2) * sizeof(float));

    for (int i = 0; i < harms, ++i) {
        for (int n = 0; n < length + 2; ++n) {
            a = amps ? amps[i] : 1.f;
            w = (i + 1) * (n * 2 * pi / lenght);
            table[n] += (float) (a * cos(w + phase));
        }
    }
    normalize_table(table, length);
    return table;
}
```

- shows usage to generate saw, square, triangle, etc

A Simple Example

6.3 Interpolation

- various methods to fill in gaps between discrete values

Interpolating Oscillators

- linear interpolation

```
y = y1 + (y2 − y1) ∗ (p − x1)

uses
y = cx + d
d = y1
c = y1 − y2
```

- cubic interpolation

```
y = ax^3 + bx^2 + cx + d
```

- code for linear interpolation and cubic interpolation here

6.4 Envelopes

Envelope Tables

- linear interpolation is the simplest form of envelope table

Exponential Interpolation

- for perceptually accurate envelopes it is sometimes necessary to use exponential interpolation

```
y(x) = y1 ∗ (y2/y1)^x
```

- implementation

```
for (int i = 0; i < N; ++i) {
    table[i] = start ∗ pow(end / start, (double) i / N);
}
```

Examples of Envelope-Table Generators

- code for line and exp table generator here

6.5 Envelope Generators

- use of tables as envelopes causes shape distortion when fitting to a duration
- solution is to employ fixed length envelope generators

Linear

- code here

Exponential

- code here

Using Curve Coefficients

- code here

An ADSR Envelope

- code here

An Example Program

6.6. Filters

Filter Basics

- generally generated by combining signals with their delayed copies in different ways
- two families
    - feedforward aka finite impulse response (FIR)
    - feedback aka infinite impulse response (IIR)
        - can include feedforward elements
- delay may be as small as one sample
- order determined by delays used
- frequency response is how a filter alters a signal's amplitude and phase at different frequencies
- amplitude response is portion of frequency response that denotes boost or attenuation of various frequencies
- phase response is the timing delay (linear or non-linear)

- IIR filters
  - simple to implement
  - complex to design
  - include
    - resonators
    - Butterworth
    - ellipticals
  - cutoffs and bandwidths can vary over time
- FIR filters
  - simpler to design and generally easy to implement
  - can offer linear-phase characteristics
  - can be computationally intensive
  - time variation can be difficult

Resonators and Other IIR Filters

- basic resonator equation (2nd order filter)

```
y(n) = x(n) − b1∗y(n − 1) − b2∗y(n − 2)
```

- defined by
  - center frequency
  - bandwidth

```
R = 1 − pi(BW / sr)
b1 = −[4R^2 / (1 + R^2)] ∗ cos(2 ∗ pi ∗ f / sr)
b2 = R^2
```

- may need to scale input to avoid distortion

```
scaling = (1 = R^2) ∗ sin(2 ∗ pi ∗ f / sr)
```

- table for second order Butterworth coefficients here
- causes a deformation on amplitude response curve near 0 Hz or Nyquist limit
  - presents solutions here

6.7 Filters: Implementation

Direct Forms

1. separate delays for feedback and feedforward sections

2. filter equation is re-arranged in two separate equations and the same delays can be used for feedforward and feedback signal paths

- shows examples here

Low-Pass and High-Pass Filters: First Order

- equation and C++ implementation here

Resonators

- C++ implementations here

RMS Estimation and Signal Balancing

- often need to rescale signal to avoid unmanageable increases in amplitude
- C++ balancing code here

A Programming Example

6.8 Delay-Based Sound Processing

Soundfile Input

- use of libsndfile described here

Delay Lines

- basic function is to delay a signal (and recombine with original) a certain amount of time

Circular Buffers

- basic delays can use a simple FIFO buffer but this becomes expensive for larger delays
- better to use circular buffer and work with the read/write heads

Fixed Delays

- basic outline
    i. read the delay buffer at the current position to generate the output
    ii. write the input signal that position
    iii. increment the position index (r/w pointer), checking if we have not reached the end of the buffer
- code for simple delay here
- single simple fixed delay recombined with original signal creates a slapback echo

- reverberators can be constructed by connecting together small units such as comb filters and allpass filters

Comb Filters

- comb filters use a fixed delay with a feedback circuit inserted into it
- feedback uses a parameter

```
g = 0.0001^(t/RVT)
t = delay time
RVT = total reverb time of comb filter
```

- code for comb filters
- useful for
    - echo effects
    - resonating chambers
        - short delays
        - high feedback gain
    - reverberator
        - connected in parallel
        - output feeding one or more allpass filters in series

Allpass Filters

- adds a feedforward section to the comb filter
- code here

Variable and Multitap Delays

- modulation of delay time allows for
    - flanging
    - chorusing
    - vibrato
    - doppler
    - pitch shift

Implementing a Variable Delay Line

- considerations (using circular buffer)
    i. writing to the delay line will always proceed sample by sample
        - have to calculate delay in relation to the writer pointer position in order to obtain a reading position index

ii. reading position, when calculated as an offset from the writing position, might fall beyond the delay buffer memory
    - have to provide a modulus operation to bring it to the right range

iii. reading position index might fall between buffer positions to calculate precise delays
    - have to interpolate to generate the output

iv. when interpolating linearly, need the next sample to truncated read-position
    - a special case arises when this position is the last sample in the delay buffer
        - in this case, read the first position of the array as the next point

Flanger

- code for flanger here

Other Effects

- can reuse flanger code to create
    - chorus by adding delayed signals without feedback and modulating with an LFO
    - vibrato by modulating delay time with LFO (without feedback)
    - doppler by calculating variable delay in relation to a distance from a source

Multitap Delays and Convolution

- combining multiple delays into a line (where taps model reflections from different surfaces in a room, for example)
- extreme is case where a tap is at each position in a delay line
- can be used to implement convolution
- in usage here, apply a gain multiplier to each tap where each resulting sample is a sum of N taps (modified by multiplier) where N is size of delay line
- sequence of gain values is n sized and is known as an impulse response
- can be seen as applying the characteristics of a system to an input sound
- can also be seen as a means of implementing FIR filters
    - IR is like a sequence of feedforward filter coefficients
- requires
    - a circular buffer tapped at each point
    - a table of the IR from 0 to N - 1
- code for convolution here

6.10 An Essential Application: An Audio Effect Plug-in

- see example code for info

Deriving a Plug-in Class from the VST Base Classes

A Pitch Shifter

Initialization

Parameters

Processing

Building a Plug-in

- NOTE: has simple instructions for manually creating a bundle

Ch. 7 Spectral Audio Programming Basics: The DFT, the FFT, and Convolution

7.1 Frequency Analysis: The Discrete Fourier Transform

- convert time-domain into frequency domain
- DFT & IDFT

```
DFT(x(n), k) = 1/N * sum from n = 0 to N - 1 of x(n) * e^(-j*2*pi*k*n/N), k = 0,1,

e^(-j*2*pi*k*n/N) = cos(2*pi*k*n/N) - j*sin(2*pi*k*n/N)
```

- the exponent `j*2*pi*k*n/N` determines the phase angle which is related to the frequency
- when k = 1, phase angle varies as `2*pi*n/N` , which completes a cycle in N samples so has a frequency of 1/N
- all other values in output are integer multiples of this frequency
- basic code

```
const double two_pi = 2*acos(-1.);

// takes in signal and output N pairs of complex values as [real, imag]
void dft(float *in, float *out, int N) {
    for (int i = 0, k = 0; k < N; i += 2, ++k) {
        out[i] = out[i + 1] = 0.f;
        for (int n = 0; n < N; ++n) {
            out[i] += in[n] * cos(k * n * two_pi / N);
            out[i + 1] -= in[n] * sin(k * n * two_pi / N);
        }
```

```
        out[i] /= N;
        out[i + 1] /= N;
    }
}
```

- complex values are called the spectral coefficient
- DFT uses a "sliding" complex sinusoid as a means of detecting spectral components

Reconstructing the Time-Domain Signal

- formula for inverse DFT

```
IDFT(X(k), n) = sum from k = 0 to N - 1 of X(k) * e^(j*2*pi*k*n/N), k = 0,1,2,...,
```

- code

```
void idft(float *in, float *out, int N) {
    for (int n = 0; n < N; ++n) {
        out[n] = 0.f;
        for (int k = 0, i = 0; k < N; ++k, i += 2) {
            out[n] += in[i] * cos(k * n * two_pi / N) - in[i + 1] * sin(k * n * tw
        }
    }
}
```

- only interested in real part of output
- SR determines fundamental frequency of analysis
- frequency points refer to k*SR/N Hz
- a property of real signals is that half of the spectrum will be a mirror of the other half
- result is ability to determine frequencies for points 0 to N/2 ranging from 0 Hz to SR/2 (Nyquist)
- second half refers to negative frequencies
- DFT splits spectrum of a digital waveform into equally space frequency points or bands

Rectangular and Polar Formats

- magnitude or modulus of a complex number z

```
|z| = (re[z]^2 + im[z]^2)^(1/2)
```

- magnitude tells the amplitude of each of the components of the spectrum
- amplitude spectrum - values obtained by magnitude conversion
- phase angle of a complex number

```
0(z) = arctan(im[z] / re[z])
```

- yields phase spectrum
- process of obtaining magnitude and phase spectrum of DFT results is called Cartesian-to-polar conversion
  - original spectral coefficients are Cartesian coordinates of complex numbers
  - amplitude and phase representations are polar form of complex numbers
  - the modulus/magnitude/length of the line from the center of the Cartesian plane to the point corresponds to the amplitude
  - phase is angle that this line makes with the horizontal axis
- DFT in this form takes a single snapshot of a set of samples and thus does not track spectral changes
  - generally useless for real world applications, requires many changes

7.2 Applications of the DFT: Convolution

- DFT above is useful for convolution
- more common to use some implementation of the FFT (fast Fourier transform) which work with limited number of points (generally an exponent of 2)
- definition of convolution

```
w(n) = y(n) ∗ x(n) = sum from n = 0 to n of y(m) ∗ x(n − m)
```

- details some examples here
- important aspect of time-domain convolution is that it is equivalent to the multiplication of spectra
- four basic applications of convolution
  i. early reverberation
     - the impulse response is a train of pulses, which can be obtained by recording room reflections in reaction to a short sound
  ii. filtering
     - the impulse response is a series of FIR filter coefficients, and its amplitude spectrum determines the shape of the filter
  iii. cross-synthesis
     - the impulse response is an arbitrary sound, whose spectrum will be multiplied with the other sound; their common features will be emphasized, and the overall effect will be one of cross-synthesis
  iv. sound localization
     - if we have impulse responses measured at our ears for sounds at different positions, by applying these to sounds using convolution, the illusion of sound

localization at different positions can be created for binaural (i.e. headphone) listening

A DFT-based Convolution Application

- fast convolutions
- in real applications, no need to use a DFT that outputs both positive and negative components
    - use FFT algorithms
- overlap-add method used to ensure tail of an impulse response of size that does not divide evenly with input blocks will be applied to the next input block
- code for convolution here

Ch. 8 The STFT and Spectral Processing

8.1 Analyzing Time-Varying Spectra: The Short-Time Fourier Transform

- previous chapter details single-frame DFT (single snapshot)
- short-time Fourier transform - uses windows or blocks to apply the DFT over time larger segments

Windowing

- windows have shapes
    - rectangular
        - used in example above
        - all window contents are multiplied by 1
        - not very useful
- shapes that tend toward 0 at the edges are preferred
- except for the rectangular case, windowing involves the application of an envelope to extracted samples
- windowing always happens when the DFT is applied to an arbitrary segment of a signal
- windowing is effectively a convolution of the input signal segment and the window
- the spectral shape of that window is not very suitable for attenuating analysis artifacts, such as the leakage observed when analyzing non-integral wave periods
- choosing a window with an amplitude spectrum that has a peak at 0 Hz and fades away quickly to 0 as frequency rises produces better results (low-pass characteristics)
- Hanning and Hamming (von Hann) are simplest and most common

```
w(n) = x - (1 - x)cos(2*pi*n/(N-1)), 0 <= n < N
where x = 0.5 for the Hanning window and 0.54 for the Hamming window
```

Performing the STFT

- apply a time-dependent window to the signal and take the DFT of the result

```
STFT(x(n), k, t) = 1/N * sum from n = -x to x of w(n - t)(*x(n)e^(-j*2*pi*k*n/N),
```

- window function w(n) must be defined in range 0 <= n < N and zero elsewhere
- example given uses the real-signal DFT which is not exact implementation of the code above
- hopsize is the space between each frame and determines the resolution of the analysis
- N/4 is value for Hamming and Hanning window hopsize
- code for STFT here

8.2 Spectral Transformations: Manipulating STFT Data

- DFT (frequency bins) - spectral frames that can be considered as a collection of complex pairs relative to the information found on equally spaced frequency bands at a particular time
- requires conversion from Cartesian to polar formats
- STFT gives amplitudes for each band but not the frequencies because this implies an instantaneous frequency
- phases contain frequency information in a different form
- can still transform amplitude and frequency components of a sound

Cross-Synthesis of Frequencies and Amplitudes

- cross-synthesis - methods for crossing aspects of two spectra
- cross-synthesis technique
  - combine the amplitudes of one sound with the frequencies of another
  - spectral version of the time-domain channel vocoder
- process
  i. convert the rectangular spectral samples into magnitudes and phases
  ii. take the magnitudes of one input and the phases of the other and make a new spectral frame, on a frame-by-frame basis
  iii. convert the magnitudes and phases to rectangular format
    - this is done with the following relationships:

```
re[z] = Magz * cos(Phaz)
im[z] = Magz * sin(Phaz)
```

- code for cross-synthesis of a spectrum here (also example usage)

Spectral-Domain Filtering

- can use these processes to create filtering effects
- code for simple low-pass filter example
- code for resonator example
- code for comb filter example

Ch. 9 Programming the Phase Vocoder

- one of the most frequently used tools in spectral processing

9.1 Tracking the Frequency

- STFT does not have enough resolution to tell frequency content of a sound
- phase vocoder can be used to track instantaneous frequencies in a spectral band
- PV analysis
  - time domain as the use of a bank of filters tuned to equally spaced frequencies, from 0 hertz to the Nyquist frequency
  - filter-bank outputs are made up of values for amplitudes and frequencies at each band.
- can take frequency-domain approach by taking STFT and adapting to fit PV output
- STFT to PV
  i. generate values that are proportional to the frequencies present in a sound
     - done by the taking the time derivative of the phase
     - can approximate it by computing the difference between the phase value of consecutive frames for each spectral band
     - does not yield the right value for the frequency at a spectral band
     - will output one that is related to right value for the frequency at a spectral band
- code for STFT to magnitudes and phase diffs
- phase unwrapping - process of converting phase output values to expected interval
  - inverse tangent outputs phase in range -pi to pi
- code for inverse operation

Frequency Estimation

- need to rotate input for STFT to obtain proper frequency values in hertz
  - complex explanation but has effect of aligning phrase in successive frames

- summary of phase vocoder analysis process
    i. extract n samples from a signal and apply an analysis window
    ii. rotate the samples in the signal frame according to input time n mod n
    iii. take the dft of the signal
    iv. convert rectangular coefficients to polar format
    v. compute the phase difference and bring the value to the range from -pi to +pi
    vi. add the difference values to `2*pi*k*D/N` , and multiply the result by `sr/2*pi*D` , where D is the hopsize in samples
    vii. for each spectral band, the result from step 6 yields its frequency in hertz, and the magnitude value, its peak amplitude
- code for phase vocoder analysis here

Phase Vocoder Resynthesis

- many methods
- efficient method is to use overlap-add method used with ISTFT
- overview
    i. convert the frequencies back to phase differences in radians per l samples by subtracting them from the center frequencies of each channel, in hertz, ksr/n, and multiplying the result by `2*pi*l/sr` , where 1 is the synthesis hopsize
    ii. accumulate them to compute the current phase values
    iii. perform a polar to rectangular conversion
    iv. take the IDFT of the signal frame
    v. unrotate the samples and apply a window to the resulting sample block
    vi. overlap-add consecutive frames
- code here
- advantages of PV analysis over STFT
    - independence of time scale and frequency scale
    - dealing with amplitude and frequency is more musically meaningful than raw spectral components
    - STFT data processing techniques are available to the PV data (with mods)
- conversion to PV format requires more analysis
- all DFT algorithms have some limited resolution
- DFT yields max of one sinusoidal component per frequency band for overlapping components within a band, whereas PV outputs incorrect information

Spectral Morphing

- PV data allows spectral interpolation, or morphing
- code and example here
- effectiveness of spectral morphing depends on spectral qualities of the two input sounds

9.2 The Instantaneous Frequency Distribution

- Toshihiko Abe
- alternative method of frequency estimation
- uses PV analysis methods with more complex mathematical underpinnings
- can use Euler's formula to rewrite STFT
- IFD of x(n) at time t is derivative of STFT phase output

```
STFT(x(n), k, t) = R(w, t) * e(j*0(w,t))

phase of band k at time t is 0(2*pi*k*n/N, t)
magnitude of band k at time t is R(2*pi*k*n/N, t)

IFD(x(n), k, t) = d/dt0(w, t)
```

- provides more mathematical derivations here

An IFD Analysis Application

- overview
  i. on each hop period, take n samples from a signal, multiply them by a window and its negative derivative
     - generate the derivative by taking the differences between its samples
  ii. take the DFT of the two windowed signals
  iii. calculate the magnitude of the rectangular output of the DFT of the windowed signal.
     - square root of the sum of the squared real and imaginary parts
     - returns output amplitudes
  iv. calculate the frequencies by first taking the imaginary part of the quotient of the two DFTs
     - returns amount of detected frequency deviation (in radians per sample) from the channel center frequency
     - convert this value into hertz by multiplying it by sr/2pi
     - add it to the channel center frequency in hertz (`w*sr` or `2*pi*k*sr/N`)
- code here

Additive Resynthesis

- not as efficient as DFT-based algorithms
- more flexible for pitch-shifting and other applications
- use information from IFD to control amplitudes and frequencies of a bank of oscillators and accumulate signal

Ch. 10 Understanding an Opcode in Csound

10.1 The tone Opcode

- first-order recursive lowpass filter
- terms
  - FL
    - a C macro to deal with single and double precision floating-point numbers
  - MYFLT
    - a C type that is either float or double
  - OENTRY
    - a structure maintained as an affayt where each entry defines an opcode
  - a rate
    - audio rate, referring to operations that generate output at the sample-rate
  - engine
    - the fundamental Csound control system that starts, maintains, and stops instruments playing
  - instrument
    - a collection (network) of opcodes that together defines a particular sound or operation; a "patch," a "preset," a "voice," an effect, or a combination of a voice(s) with effect(s); a sound generating or processing algorithm
  - k rate
    - the control rate, a rate slower that the sample rate at which changes occur
  - ksmps
    - the number of samples generated on each control cycle; the sample rate divided by the control rate
  - opcodes
    - the atomic operation from which instruments are built

A Quick Guide to Csound as a Program

- object-oriented system that facilitates instrument orchestration through code
- instruments are composed of opcodes
- arguments are passed by reference to opcodes through a system of interconnected opcodes
- main purpose of engine is to trigger and release notes in time
- Csound generates short vectors of audio on each pass of the control cycle
- for each k rate cycle, a function is called for each opcode
- each opcode has an intialization function and a generation function

- metadata is specified in entry structure within OENTRY array
  - info on OENTRY structs here
  - Table 10.2 - Entry codes for opcode specification in OENTRY

The TONE Structure

- all opcodes must begin with an OPDS entry (used by engine)
- use MYFLOAT to allow Csound to set float type during compilation

Initialization of tone

Execution of tone

Conclusions from tone

10.2 A k-rate Opcode: rms

10.3 Memory Allocation in Opcodes: vdelay

10.4 Polymorphic Opcodes: taninv2 and divz

10.5 Opcodes that Use f-tables: dconv

10.6 General Information about the Csound Process

- global value csound->ksmps
  - indicates length of audio vector for input or output
- global structure is of type CSOUND *
  - in csoundCore.h
  - esr - sample rate
  - nchnls - number of output channels
  - orchname
  - scorename
  - kcounter
- read opcodes for more information

10.7 Writing Your Own Opcode: negate

10.8 A k-rate Plug-in Opcode: trigger

More About Plug-in Opcodes

10.9 Plug-in Table Generators: fgens

10.10 Localization and Internationalization

Ch. 11 Spectral Opcodes

- pvsanal
- pvsytnh
- can be used to transform signals from waveforms (in the time domain) into spectra (in the frequency domain) and vice versa

11.1 Processing Spectral Signals

- spectral domain processing uses two separate data types
  - wsig - hold a special, non-standard, type of logarithmic frequency analysis data and is used with a few opcodes originally provided for manipulating this data type
  - fsig - used to provide a framework for spectral processing, in what is called "streaming phase vocoder processing"
- pp 619

Ch. 12 A Modular Synthesizer Simulation Program

12.1 Basic System Design

- patch modules for sound modification
- use a patch specification to generate Csound modules

Patch Specification for an Oscillator

- waveform, frequency, LFO frequency and amplitude modulation

Designing the Interface

- transform simple specs into Csound code
- requires parsing so keep spec simple

12.2 Building the Code

Introducing the Command-Line Interface

12.3 Tightening the Structure and Making it Safer

12.4 Writing the Csound Wrapper Code

12.5 Enabling Oscillator Feedback

12.6 Adding Noise

12.7 Sample and Hold

12.8 Adding the Filter

Ch. 13 Using C to Generate Scores

13.1 The Musical Concept

13.2 The Raw Process

13.3 Applying the Henon Map

Alternative Mapping of Henon Map

13.4 Standard Map on a Torus

Generating Notes from Henon and Torus

Developing the Pieces

Ch. 14 Modeling Orchestral Composition

14.2 Observations on Orchestral Composition

- combo of performers and instruments
  - locations of performers
  - number of each instrument per section
  - technical proficiency of performers
  - types and qualities of instruments
- configurable properties of an orchestra
  - instruments
    - type of instrument
    - number of each instrument
    - quality of each instrument
  - performers
    - technical proficiency of performer
    - accuracy of timing
    - accuracy of pitch
    - spatial location of performer
  - performer groups
    - number of performers per group
    - types of performers per group

Composing for the Orchestra

- think of specific configuration of orchestra as palette of sound
- write for this palette
- factors to consider
  - compositional idea
  - performance instructions
  - mapping of performance instructions to performers

Performance Instructions

- common vs uncommon practice music notation
- generally need to clearly define system for mapping performance instructions to actual performance
- orchestra, instruments, and performance potentials present a palette of possibilities for sound production
- notation can be precise or generalized and abstract

Mapping Performance Instructions to Performers

- end musical result is sum of all members' contributions
- mapping of performance instructions to performers occurs in many configurations
    - one-to-one - one musical instruction is given to one performer to perform
        - the musical idea notated is exactly what is intended for the one performer to perform
    - one-to-many - one musical idea is given to many performers to perform
        - a common case of this is notating music for an instrument section like "violin i" to perform
        - while the composer may notate performance instructions once and designate it for the group to perform, from the performer's perspective, they each receive a copy of the musical instruction to perform and thus there are multiple copies of the musical instruction
        - this is a case where the composer intent to direct a group of performers with a single set of instructions maps out to be performed by many performers with multiple copies of the original instructions
    - many-to-one - multiple musical instructions are given to one performer to perform
        - this kind of mapping may be commonly seen in music written for the piano where multiple musical lines may be given to a single performer to perform
    - many-to-many - multiple musical instructions are given to multiple performers
        - this is a general description of the overall notated musical score given to groups of performers to perform

Mapping Compositional Ideas to Performance Instructions

- a single compositional idea may be mapped to a single or multiple sets of performance instructions

14.3 Designing the Orchestral Composition Library

Using Csound for Instruments and Notes

14.4 Designing the Programming Library

- see code for most of this

Note

Performer

PerformerGroup

14.5 Using the Library for Composition

- comparison of processes
  - determine instrumentation (woodwinds, brass, strings, percussion, electronic)
    - determine Csound instruments to use for the piece and place them in a Csound orchestra (ORC) file
  - specify number and layout of instruments
    - create a C++ program to use the orchestra library and define the number and layout of performers of chosen instrument, achieved by creating instances of the performer class and configuring the instances, optionally grouping into PerformerGroup objects
  - notate score (includes performers, notes and timings of notes, techniques, etc)
    - in C++ program, call performance methods on Performer and PerformerGroup objects with Note data or other arguments
    - calling different methods is equivalent to composing for different performance techniques and the program being written is equivalent to the notated score for live orchestra
    - composer can define new performance techniques by adding methods to Performer or PerformGroup classes
  - perform score live
    - run the program to generate the Csound score (SCO) file that will be the performance instructions for the instruments in the ORC file
    - run Csound with the ORC and SCO to create the end musical result
  - code here

Initial Setup

Example 1: Using the Note Class Directly

Example 2: Using the Performer and PerformerGroup Perform Methods

Example 3: Four-Part Harmony

Example 4: Group Aleatory

Example 5: Surfaces (Glissandi Sound Mass)