Chapter 7  PROGRAMMING EXERCISE

Creating a Pitch Glide in C++

A Risset pitch glide is an audio illusion that sounds like a constantly rising pitch. It is the aural equivalent of the visual image of a stripe on a barber pole that seems to be rising constantly. The pitch glide is created from a sum of $n$ audio components each of which goes up in frequency through $n$ octaves, from frequency $f$ through $2^n f$. (It’s possible to use a fraction of an octave as well.) For example, if the starting frequency is 13.75 Hz and there are eight component sounds, each component would sweep from 13.75 to 27.5 Hz in the first octave, from 27.5 to 55 Hz in the second octave, from 55 Hz to 110 Hz in the third octave, from 110 Hz to 220 Hz in the fourth octave, from 220 Hz to 440 Hz in the fifth octave, from 440 to 880 Hz in the sixth octave, from 880 to 1760 Hz in the seventh octave, and from 1760 to 3520 Hz in the eighth octave. However, each component begins its sweep at a different octave. The illusion of a constantly rising pitch that is created when these components are summed results from the following:

- Each component makes the sweep through the rising frequencies at a logarithmic rate over time.
- As each component sweeps through the frequencies, its amplitude rises to a maximum halfway through and then falls to 0 at the highest frequency. The amplitude envelope has a Gaussian shape like the one below. Note that the x-axis is time and the y-axis is amplitude in the figure.

![Amplitude envelope](image)

- Each component begins the sweep at the beginning of a different octave, and each component’s beginning amplitude is relative to its beginning frequency. If there are
eight components, for example, they would be shaped like this, where time is on the horizontal axis and amplitude is on the vertical axis:
Your assignment is to create a pitch glide in a C++ program.
Solution:

```cpp
#include "PCMWave.h"
#include <iostream>
#include <cassert>
#include <cmath>
using namespace std;

//nice, accurate value for pi without typing in lots of decimal places
const double pi = atan(1.0)*4;
//maximum 16-bit sample
#define SAMPLE_MAX 32767

int main() {
    PCMWave outWave; // container to write the output .wav data
    string outfilename; // name of the output file
    double lowFreq; // low frequency of the pitch glide
    double highFreq; // high frequency of the pitch glide
    int divisions; // number of components of the pitch glide (I swear, there
    // was some reason I called this divisions)
    double seconds; // length of the pitch glide, in seconds
    int sampRate; // Sampling rate for the output .wav data
    cout << "Please enter the output file name: ";
    cin >> outfilename;
    cout << "Please enter the low frequency of the pitch glide: ";
    cin >> lowFreq;
    if(lowFreq <= 0) lowFreq = 55; // default
    cout << "Please enter the high frequency of the pitch glide: ";
    cin >> highFreq;
    if(highFreq <= 0) highFreq = 880; // default
    if(highFreq < lowFreq) { // swap them
        double x = highFreq;
        highFreq = lowFreq;
        lowFreq = x;
    }
    cout << "Please enter the number of components to the pitch glide: ";
    cin >> divisions;
    if(divisions <= 0) divisions = 16; // default
    cout << "Please enter the number of seconds for the pitch glide to last: ";
    cin >> seconds;
    if(seconds <= 0) seconds = divisions*2; // default
    cout << "Please enter the sampling rate of the output file: ";
    cin >> sampRate;
    if(sampRate <= 0) sampRate = 44100;

    //The following code ensures that there is an integer number of samples per
division
    //first calculate number of samples, and round down
    int samples = static_cast<int>(seconds*sampRate);
```

//now calculate number of samples per division, and round down
int samplesPerDivision = samples/divisions;

//test in case there's somehow fewer samples than divisions (GIGO)
if(samplesPerDivision <= 0) {
    cerr << "Fewer samples than divisions; bailing" << endl;
    return -1;
}

//Recalculate number of samples based on integer number of samples per division
samples = divisions*samplesPerDivision;

//Recalculate number of seconds based on integer number of samples per division
//not needed because seconds is never used again in this program
//seconds = static_cast<double>(samples)/sampRate;

//this way we have an integer number of samples per division, which makes things much much easier

cout << "Calculating frequencies..." << endl;
//frequencies, exponentially spaced from lowFreq to highFreq, including lowFreq but not highFreq
//powers of highFreq/lowFreq are calculated separately, rather than iteratively, to avoid massive floating-point round-off error in the latter stages of the iteration
vector<double> freqs(samples);
for(int i = 0; i < samples; ++i)
    freqs[i] = lowFreq*pow(highFreq/lowFreq,static_cast<double>(i)/samples);

cout << "Calculating amplitude envelope...

" << endl;
//amplitude envelope, so gliders/components fade in then out
vector<double> amp(samples);
for(int i = 0; i < samples; ++i)
    amp[i] = (-cos(2*pi*i/samples) + 1) / (2*divisions);

cout << "Calculating one component...

" << endl;
//First glider; starts at lowFreq and ends at highFreq
vector<double> slider0(samples,0);
for(int i = 0; i < samples; ++i)
    slider0[i] = amp[i]*sin(2*pi*freqs[i]*i/sampRate);

cout << "Calculating rest of components...

" << endl;
//Output vector
vector<double> results = slider0;
//for each division
for(int i = 1; i < divisions; ++i) {
    //cut the original glider at an appropriate point in the middle, swap the portions, put back together and add to results
    for(int j = 0; j < (samplesPerDivision*i); ++j)
        results[j] += slider0[j+(samplesPerDivision*(divisions-i))];
    for(int j = (samplesPerDivision*i); j < samples; ++j)
        results[j] += slider0[j-(samplesPerDivision*i)];
} //actual output vector (in short rather than double)
vector<short> resultsshort(samples);
for(int i = 0; i < samples; ++i)
    //rescale from -1..1 to -SAMPLE_MAX..SAMPLE_MAX
    resultsshort[i] = static_cast<short>(floor(SAMPLE_MAX*results[i]));

cout << "outputting wave file..." << endl;

//set the data in the output wave container
outWave.setData(resultsshort);
//set the sampling rate; rest of header is set by setData
outWave.setSampRate(sampRate);
//write the wave file
outWave.write(outfilename);

    cout << "Mission complete!" << endl;
    return 0;
}