#pragma once

#include <vector>

#include <iostream>

#include <windows.h>

using namespace std;

struct VoiceParagraph

{

unsigned long begin; //语音段落开始点

unsigned long end; //语音段落结束点

unsigned long voiceLength; //语音段落长度

VoiceParagraph(unsigned long be, unsigned long en, unsigned long vo) { //构造初始化

begin = be;

end = en;

voiceLength = vo;

}

};

class WavFile

{

private:

FILE \*fp; //文件指针

char \*fileName; //文件名或地址

char id\_RIFF[5]; //RIFF块标志

unsigned long fileSize; //文件的总字节数

char id\_WAVE[5]; //WAV标志

char id\_FMT[5]; //格式块标志

unsigned long formatLength; //格式块长度，16为正常，18说明有附加信息

short formatTag; //格式类别，值=1 表示编码方式为PCMu律编码

short channelsNumber; //声道数

unsigned long sampleRate; //每秒的样本数

unsigned long secondBytes; //每秒数据的字节数

short chunkAlign; //采样字节数

short sampleBits; //采样位数

short appendInformation; //附加信息，通过formatLength来判断

char id\_FACT[5]; //附加块标志

unsigned long appendLength; //附加块长度

unsigned long appendNone; //未知

char id\_DATA[5]; //附加块标志

unsigned long dataSize; //数据部分字节数

char \*data; //数据部分

long \*dataTuple; //每个样本的数据

unsigned long dataNumber; //样本的数据个数

long dataMax; //样本数据的最大值

long dataMin; //样本数据的最小值

bool Conversion\_Tuple(void); //将直接读取的数据转换为样本数据

long MakeWord(long NumberA, long NumberB); //合并字节

public:

WavFile(void) {

dataMax = 0;

dataMin = 1000000;

}

WavFile(FILE \*f) {

fp = f;

::WavFile();

}

~WavFile() {}

void Give\_FP(FILE \*f) {

fp = f;

}

bool Read\_File(void); //读取文件

unsigned long Get\_SampleRate(void); //获取采样频率

short Get\_ChunkAlign(void); //获取样本字节数

long Get\_Data(unsigned long Number); //获取某个位置上的数据

unsigned long Get\_dataNumber(void); //获取样本数据个数

void ShowData(void); //输出数据

void SaveNewWav(unsigned long voiceNumber, vector<VoiceParagraph> voiceParagraph); //保存去掉空白处的语音文件，参数1为段落的个数，参数2为各个语音段落的开始点与结束点信息

};

#include "../stdafx.h"

#include "WavFile\_Struct.h"

bool WavFile::Conversion\_Tuple(void)

{

try

{

cout << "TIP : Change to real format data ..." << endl;

dataNumber = dataSize / chunkAlign; //求出数据个数

dataTuple = (long\*)malloc(sizeof(long)\*dataNumber); //为新字节数据空间赋值

long tempNum = 0; //用来保存每个新字节数据

for (unsigned long i = 0; i < dataNumber; ++i) { //总共有dataNumber个数据

int Flag = 0;

for (short j = 0; j < chunkAlign; ++j) { //每个数据有两个字节

tempNum = MakeWord((long)data[i\*chunkAlign + j], tempNum); //每次将tempNum做为高字节数据，新字节为低字节数据制作新数据

}

dataTuple[i] = tempNum; //保存下新字节数据

tempNum = 0; //初始化缓冲区操作

if (dataMax < dataTuple[i]) { //求出最大值

dataMax = dataTuple[i];

}

if (dataMin > dataTuple[i]) { //求出最小值

dataMin = dataTuple[i];

}

}

}

catch (invalid\_argument &e) {

cerr << e.what() << endl;

MessageBoxA(NULL, e.what(), "ERROR", MB\_ICONHAND);

return false;

}

return true;

}

long WavFile::MakeWord(long NumberA, long NumberB) //合并字节，NumberA表示高字节位，NumberB表示低字节位

{

int Flag = 0; //标志，用来表示最后的数的正负，1为负，0为正

if (NumberA >= 0) {

if (NumberB >= 0) {

return NumberA \* 256 + NumberB; //高低字节都是正数，则直接移位合并

}

else {

return NumberA \* 256 + abs(NumberB) + 128; //高字节为正数，低字节为负数

}

}

else {

Flag = 1;

if (NumberB > 0) { //高字节为负数，低字节为正数

return -1 \* (abs(NumberA) \* 256 + NumberB);

}

else { //高低字节都是负数

return -1 \* (abs(NumberA) \* 256 + abs(NumberB) + 128);

}

}

}

bool WavFile::Read\_File(void)

{

try

{

cout << "TIP : Reading file ..." << endl;

fread(id\_RIFF, sizeof(char), 4, fp); //读取'RIFF'

id\_RIFF[4] = 0; //末尾添零

if (strcmp(id\_RIFF, "RIFF")) {

throw invalid\_argument("ERROR : File not RIFF file !");

}

fread(&fileSize, sizeof(unsigned long), 1, fp); //读取文件的大小

fread(id\_WAVE, sizeof(char), 4, fp); //读取'WAVE'

id\_WAVE[4] = 0;

if (strcmp(id\_WAVE, "WAVE")) {

throw invalid\_argument("ERROR : File not WAVE file !");

}

fread(id\_FMT, sizeof(char), 4, fp); //读取'FMT'

id\_FMT[3] = 32;

id\_FMT[4] = 0;

fread(&formatLength, sizeof(unsigned long), 1, fp); //

fread(&formatTag, sizeof(short), 1, fp); //读取文件标签

fread(&channelsNumber, sizeof(short), 1, fp); //读取通道数目

fread(&sampleRate, sizeof(unsigned long), 1, fp); //读取采样频率

fread(&secondBytes, sizeof(unsigned long), 1, fp); //读取每秒数据量

fread(&chunkAlign, sizeof(short), 1, fp); //读取块对其

fread(&sampleBits, sizeof(short), 1, fp); //读取样本大小

if (formatLength > 16) {

fread(&appendInformation, sizeof(short), 1, fp); //读取附加信息

}

fread(id\_DATA, sizeof(char), 4, fp); //读取'DATA'

id\_DATA[4] = 0;

fread(&dataSize, sizeof(unsigned long), 1, fp); //读取数据大小

data = (char\*)malloc(sizeof(char)\*dataSize); //申请数据的存储空间

fread(data, sizeof(char), dataSize, fp); //读取数据

fclose(fp);

}

catch (invalid\_argument &e)

{

MessageBoxA(NULL, e.what(), "ERROR", MB\_ICONHAND);

return false;

}

if (Conversion\_Tuple() == true) { //转换格式成功

MessageBoxA(NULL, "TIP : File read ok !", "TIP", MB\_ICONASTERISK);

}

else {

MessageBoxA(NULL, "ERROR : Data change failure !", "ERROR", 0);

return false;

}

return true;

}

unsigned long WavFile::Get\_SampleRate(void) //获取采样频率

{

return sampleRate;

}

short WavFile::Get\_ChunkAlign(void) //获取样本字节数

{

return chunkAlign;

}

long WavFile::Get\_Data(unsigned long Number) //获取某个位置上的数据

{

if (Number >= dataNumber) { //如果所需要的数超过了数据个数

MessageBoxA(NULL, "ERROR : Over list !", "ERROR", MB\_ICONHAND);

return -1;

}

else {

return dataTuple[Number];

}

}

unsigned long WavFile::Get\_dataNumber(void) //获取样本数据个数

{

return dataNumber;

}

void WavFile::ShowData(void)

{

cout << "TIP : Show data ..." << endl;

cout << id\_RIFF << endl;

cout << fileSize << endl;

cout << id\_WAVE << endl << endl;

cout << id\_FMT << endl;

cout << formatLength << endl;

cout << formatTag << endl;

cout << channelsNumber << endl;

cout << sampleRate << endl;

cout << secondBytes << endl;

cout << chunkAlign << endl;

cout << sampleBits << endl;

cout << appendInformation << endl << endl;

cout << id\_FACT << endl;

cout << appendLength << endl;

cout << appendNone << endl << endl;

cout << id\_DATA << endl;

cout << dataSize << endl;

for (unsigned int i = 0; i < dataNumber; ++i) {

cout << dataTuple[i] << "\t";

}

cout << endl;

cout << "MAX :" << dataMax << endl;

cout << "MIN :" << dataMin << endl;

}

void WavFile::SaveNewWav(unsigned long voiceNumber, vector<VoiceParagraph> voiceParagraph) //保存去掉空白处的语音文件，参数1为段落的个数，参数2为各个语音段落的开始点与结束点信息

{

unsigned long endPointLength = 0;

for (unsigned long i = 0; i < voiceNumber; ++i){

VoiceParagraph temp = voiceParagraph[i];

endPointLength += temp.voiceLength;

}

unsigned long removerLength = dataNumber - endPointLength;

unsigned long removerSize = removerLength\*chunkAlign;

//更改数据操作较多，暂时无法完成

//Wav\_File\* newWav = (Wav\_File\*)malloc(sizeof(Wav\_File));

//newWav = this;

FILE \*fp;

if ((fp = fopen("new file.wav", "wb")) == NULL) {

cout << "ERROR : File open failed !" << endl;

exit(-1);

}

try

{

unsigned long tempLength;

cout << "TIP : Writing file ..." << endl;

fwrite(id\_RIFF, sizeof(char), 4, fp); //写入'RIFF'

tempLength = fileSize - removerSize;

fwrite(&tempLength, sizeof(unsigned long), 1, fp); //写入文件的大小

fwrite(id\_WAVE, sizeof(char), 4, fp); //写入'WAVE'

fwrite(id\_FMT, sizeof(char), 4, fp); //写入'FMT'

fwrite(&formatLength, sizeof(unsigned long), 1, fp); //写入格式块长度

fwrite(&formatTag, sizeof(short), 1, fp); //写入文件标签

fwrite(&channelsNumber, sizeof(short), 1, fp); //写入通道数目

fwrite(&sampleRate, sizeof(unsigned long), 1, fp); //写入采样频率

fwrite(&secondBytes, sizeof(unsigned long), 1, fp); //写入每秒数据量

fwrite(&chunkAlign, sizeof(short), 1, fp); //写入块对其

fwrite(&sampleBits, sizeof(short), 1, fp); //写入样本大小

if (formatLength > 16) {

fwrite(&appendInformation, sizeof(short), 1, fp); //写入附加信息

}

fwrite(id\_DATA, sizeof(char), 4, fp); //写入'DATA'

tempLength = dataSize - removerSize;

fwrite(&tempLength, sizeof(unsigned long), 1, fp); //写入数据大小

unsigned long n = 0;

for (unsigned long i = 0; i < voiceNumber; ++i){

VoiceParagraph tempParagraph = voiceParagraph[i];

for (unsigned long j = tempParagraph.begin; j < tempParagraph.end; ++j, n += 2){

fwrite(&data[j \* 2], sizeof(char), 1, fp);

fwrite(&data[j \* 2 + 1], sizeof(char), 1, fp);

}

}

fclose(fp);

}

catch (invalid\_argument &e)

{

MessageBoxA(NULL, e.what(), "ERROR", MB\_ICONHAND);

}

}

#pragma once

#include "WavFile\_Struct.h"

#include <cmath>

#define MUTEPARAGRAPH 0 //静音段

#define INTERIMPARAGRAPH 1 //过渡段

#define VOICEPARAGRAPH 2 //语音段

#define OVERPARAGRAPH 3 //语音结束

class WavFile\_Initial :protected WavFile

{

private:

double energyHigh; //短时帧能量高门限

double energyLow; //短时帧能量低门限

double zcrHigh; //短时过零率高门限

double zcrLow; //短时过零率低门限

static const unsigned long minSilence; //最短静音长度

static const unsigned long minVoiceLength; //最短语音长度

double \*dataDouble; //新转换的Double型数据，控制范围在[-1,1]

vector<double> dataEnergy; //保存短时帧能量，个数为dataNunber-N

vector<double> dataZCR; //保存短时过零率

double maxEnergy; //用于保存最大的短时帧能量

double minEnergy; //用于保存最小的短时帧能量

double maxZCR; //用于保存最大的短时过零率

double minZCR; //用于保存最小的短时过零率

unsigned long voiceNumber; //语音段落个数

vector<VoiceParagraph> voiceParagraph; //保存这个语音文件中所有要处理的语音段落

bool Conversion\_Double(void); //用来将新字节数据转换为Double数据

double Hamming\_window(double data); //汉明窗函数

short Sign\_Function(double data); //求短时过零率的辅助符号函数

bool Frame\_Energy(void); //用于求短时帧能量

bool Frame\_ZCR(void); //用于求短时过零率

bool Frame\_EnergyZcr(void); //用于同时求取短时帧能量与短时过零率

public:

static const int N; //表示每个窗的帧长

static const int FrameShift; //窗函数的帧移

static const double PI; //数学圆周率PI

static const double preCoefficient; //预加重系数

WavFile\_Initial(void) {}

WavFile\_Initial(FILE \*f) :WavFile(f) {

try

{

if (WavFile::Read\_File() == false) { //调用父函数读取文件

throw;

}

dataDouble = (double\*)malloc(sizeof(double)\*WavFile::Get\_dataNumber()); //为Double型数据申请内存空间

if (dataDouble == NULL) { //分配空间未成功

throw invalid\_argument("ERROR : Memory failure !");

}

if (Conversion\_Double() == false) { //将新字节数据转换为Double数据

throw invalid\_argument("ERROR : Data change failure !");

}

this->Endpoint\_Detection(); //开始端点检测

}

catch (invalid\_argument &e) {

MessageBoxA(NULL, e.what(), "ERROR", MB\_ICONHAND);

exit(-1);

}

}

~WavFile\_Initial() {

free(dataDouble);

}

double\* Get\_WavFileData(void); //获取合成完毕的语音数据

unsigned long Get\_SampleRate(void); //获取采样频率

vector<double> Get\_DataEnergy(void); //获取短时帧能量的数据

vector<double> Get\_DataZCR(void); //获取短时过零率的数据

double Get\_maxEnergy(void); //获取最大短时帧能量

double Get\_minEnergy(void); //获取最小短时帧能量

double Get\_maxZCR(void); //获取最大短时过零率

double Get\_minZCR(void); //获取最小短时过零率

double Get\_dataNumber(void); //获取Double数据的个数

double Get\_dataEZNumber(void); //获取能量过零率的个数

double Get\_DataDouble(unsigned long Number); //获取转换后的Double数据

double Get\_DataEnergy(unsigned long Number); //依据序号找到对应的短时帧能量

double Get\_DataZCR(unsigned long Number); //依据序号找到对应的短时过零率

int Get\_WindowLength(void); //获取帧长（窗的大小）

unsigned long Get\_voiceNumber(void); //获取语音段落个数

unsigned long Get\_frameNumber(void); //获取端点检测后每个段落的帧数和

unsigned long Get\_frameNumber(double dataSize); //计算长度内的帧数

unsigned long Get\_frameNumber(VoiceParagraph voiceParagraph); //获取某个语音段落的帧数

VoiceParagraph Get\_dataVoicePoint(unsigned long Number); //获取某个语音段落

void ShowData(void); //覆盖父类的展示数据函数

void SaveNewWav(void); //保存去掉空白处的语音文件

void Pre\_emphasis(VoiceParagraph voiceParagraph, double \*dataDouble); //对一个段落内的数据进行预加重处理 （注：预加重处理可以安防在分帧前，也可安放在分帧后）

bool Frame\_Data(double \*data, unsigned long index, double\* dataSpace, int dataSpaceSize); //获取端点检测后第index帧的分帧加窗操作

bool Frame\_Data(double \*data, double dataSize, unsigned long index, double\* dataSpace, int dataSpaceSize); //对部分数据进行分帧加窗操作

bool Endpoint\_Detection(void); //端点检测函数

};

#include "../stdafx.h"

#include "WavFile\_Initial.h"

const int WavFile\_Initial::N = 256; //初始化每个窗的窗长

const int WavFile\_Initial::FrameShift = 125; //初始化窗函数的帧移

const double WavFile\_Initial::PI = 3.14159; //初始化圆周率

const double WavFile\_Initial::preCoefficient = -0.98; //预加重系数

const unsigned long WavFile\_Initial::minSilence = 6; //最短静音长度

const unsigned long WavFile\_Initial::minVoiceLength = 15; //最短语音长度

bool WavFile\_Initial::Conversion\_Double(void) //用来将新字节数据转换为Double数据

{

try

{

cout << "TIP : Change to double data ..." << endl;

const double Flag = pow((double)2, WavFile::Get\_ChunkAlign() \* 8); //表示的是原数据最大值的一半

for (unsigned long i = 0; i < WavFile::Get\_dataNumber(); ++i) { //遍历每个数据

dataDouble[i] = (double)WavFile::Get\_Data(i) / Flag; //控制每个数据在[-1,1]之间

}

}

catch (invalid\_argument &e){

cerr << e.what() << endl;

MessageBoxA(NULL, e.what(), "ERROR", MB\_ICONHAND);

return false;

}

return true;

}

double WavFile\_Initial::Hamming\_window(double data) //汉明窗函数

{

if (data >= 0 && data <= (N - 1)) { //0<= n <= N-1的情况

return 0.54 - 0.46\*cos(2 \* PI\*data / (N - 1)); //返回数值

}

else {

return 0;

}

}

short WavFile\_Initial::Sign\_Function(double data) //求短时过零率的辅助符号函数

{

if (data >= 0) {

return 1;

}

else {

return 0;

}

}

bool WavFile\_Initial::Frame\_Energy(void) //用于求短时帧能量

{

cout << "TIP : Calculate Energy ..." << endl;

maxEnergy = 0; //最大短时帧能量置0

minEnergy = 1000000; //最小短时帧能量置1000000

double sum = 0;

for (unsigned long i = 0; i < this->Get\_dataNumber() - N; i += WavFile\_Initial::FrameShift){ //这是所有短时帧能量数据的个数

for (unsigned long j = i; j < i + N; ++j) { //遍历窗中的每一个数据

sum += pow(dataDouble[j] \* Hamming\_window(i + N - 1 - j), 2); //求每一个数据的能量

}

if (sum > maxEnergy) { //求取最大短时帧能量

maxEnergy = sum;

}

if (sum < minEnergy) { //求取最大短时帧能量

minEnergy = sum;

}

dataEnergy.push\_back(sum); //将此帧的短时帧能量保存

sum = 0;

}

return true;

}

bool WavFile\_Initial::Frame\_ZCR(void) //用于求短时过零率

{

cout << "TIP : Calculate ZCR ..." << endl;

maxZCR = 0; //最大短时过零率置0

minZCR = 1000000; //最小短时过零率置1000000

double sum = 0;

for (unsigned long i = 0; i < this->Get\_dataNumber() - N; i += WavFile\_Initial::FrameShift) {//这是所有短时帧过零率数据的个数

for (unsigned long j = i; j < i + N; ++j) { //遍历窗中的每一个数据

sum += abs(Sign\_Function(dataDouble[j]) - Sign\_Function(dataDouble[j - 1]))//过零率中的绝对值部分

\*Hamming\_window(i + N - 1 - j);

}

sum /= 2;

if (sum > maxZCR) { //求取最大短时过零率

maxZCR = sum;

}

if (sum < minZCR) { //求取最大短时过零率

minZCR = sum;

}

dataZCR.push\_back(sum);

sum = 0;

}

return true;

}

bool WavFile\_Initial::Frame\_EnergyZcr(void) //用于同时求取短时帧能量与短时过零率

{

cout << "TIP : Calculate Engry and ZCR ..." << endl;

maxEnergy = 0; //最大短时帧能量置0

minEnergy = 1000000; //最小短时帧能量置1000000

maxZCR = 0; //最大短时过零率置0

minZCR = 1000000; //最小短时过零率置1000000

double sumEnergy = 0;

double sumZcr = 0;

double hanming = 0;

for (unsigned long i = 0; i < this->Get\_dataNumber() - N; i += WavFile\_Initial::FrameShift) {//这是所有短时帧能量数据的个数

for (unsigned long j = i; j < i + N; ++j) { //遍历窗中的每一个数据

hanming = Hamming\_window(i + N - 1 - j);

sumEnergy += pow(dataDouble[j] \* hanming, 2); //求每一个数据的能量

sumZcr += abs(Sign\_Function(dataDouble[j]) - Sign\_Function(dataDouble[j - 1])) //过零率中的绝对值部分

\*hanming;

}

sumZcr /= 2;

if (sumEnergy > maxEnergy) { //求取最大短时帧能量

maxEnergy = sumEnergy;

}

if (sumEnergy < minEnergy) { //求取最大短时帧能量

minEnergy = sumEnergy;

}

if (sumZcr > maxZCR) { //求取最大短时过零率

maxZCR = sumZcr;

}

if (sumZcr < minZCR) { //求取最大短时过零率

minZCR = sumZcr;

}

dataEnergy.push\_back(sumEnergy); //将此帧的短时帧能量保存

dataZCR.push\_back(sumZcr);

sumEnergy = 0;

sumZcr = 0;

}

return true;

}

double\* WavFile\_Initial::Get\_WavFileData(void) //获取合成完毕的语音数据

{

return this->dataDouble;

}

unsigned long WavFile\_Initial::Get\_SampleRate(void) //获取采样频率

{

return WavFile::Get\_SampleRate();

}

vector<double> WavFile\_Initial::Get\_DataEnergy(void) //获取短时帧能量的数据

{

return this->dataEnergy;

}

vector<double> WavFile\_Initial::Get\_DataZCR(void) //获取短时过零率的数据

{

return this->dataZCR;

}

double WavFile\_Initial::Get\_maxEnergy(void) //获取最大短时帧能量

{

return maxEnergy;

}

double WavFile\_Initial::Get\_minEnergy(void) //获取最小短时帧能量

{

return minEnergy;

}

double WavFile\_Initial::Get\_maxZCR(void) //获取最大短时过零率

{

return maxZCR;

}

double WavFile\_Initial::Get\_minZCR(void) //获取最小短时过零率

{

return minZCR;

}

double WavFile\_Initial::Get\_dataNumber(void) //获取Double数据的个数

{

return WavFile::Get\_dataNumber();

}

double WavFile\_Initial::Get\_dataEZNumber(void) //获取能量过零率的个数

{

return min(dataEnergy.size(), dataZCR.size());

}

double WavFile\_Initial::Get\_DataDouble(unsigned long Number) //获取转换后的Double数据

{

if (Number >= (this->Get\_dataNumber()) || Number < 0) { //如果所需要的数超过了数据个数

MessageBoxA(NULL, "ERROR : Over list !", "ERROR", MB\_ICONHAND);

throw invalid\_argument("ERROR : Over list !");

return -1;

}

else {

return dataDouble[Number];

}

}

double WavFile\_Initial::Get\_DataEnergy(unsigned long Number) //依据序号找到对应的短时帧能量

{

if (Number >= dataEnergy.size() || Number < 0) { //如果所需要的数超过了数据个数

MessageBoxA(NULL, "ERROR : Over list !", "ERROR", MB\_ICONHAND);

throw invalid\_argument("ERROR : Over list !");

return -1;

}

else {

return dataEnergy[Number];

}

}

double WavFile\_Initial::Get\_DataZCR(unsigned long Number) //依据序号找到对应的短时过零率

{

if (Number >= dataZCR.size() || Number < 0) { //如果所需要的数超过了数据个数

MessageBoxA(NULL, "ERROR : Over list !", "ERROR", MB\_ICONHAND);

throw invalid\_argument("ERROR : Over list !");

return -1;

}

else {

return dataZCR[Number];

}

}

int WavFile\_Initial::Get\_WindowLength(void) //获取帧长（窗的大小）

{

return N;

}

unsigned long WavFile\_Initial::Get\_voiceNumber(void) //获取语音段落个数

{

return this->voiceNumber;

}

unsigned long WavFile\_Initial::Get\_frameNumber() //获取端点检测后每个段落的帧数和

{

unsigned long sumNumber = 0;

for (unsigned long i = 0; i < this->voiceParagraph.size(); ++i) {

sumNumber += this->Get\_frameNumber(this->voiceParagraph[i]);

}

return sumNumber;

}

unsigned long WavFile\_Initial::Get\_frameNumber(double dataSize) //计算长度内的帧数

{

unsigned long frameNumber = (unsigned long)((dataSize - WavFile\_Initial::N) / WavFile\_Initial::FrameShift); //计算这段数据内有多少帧

unsigned long end = (frameNumber - 1) \* WavFile\_Initial::FrameShift + WavFile\_Initial::N; //求出当前计算帧数中所包含的数据量

if (end < this->Get\_dataNumber()) { //如果没有包含所有的数据，则帧数+1

frameNumber++;

}

return frameNumber;

}

unsigned long WavFile\_Initial::Get\_frameNumber(VoiceParagraph voiceParagraph)//获取某个语音段落的帧数

{

return this->Get\_frameNumber(voiceParagraph.voiceLength);

}

VoiceParagraph WavFile\_Initial::Get\_dataVoicePoint(unsigned long Number) //获取某个语音段落

{

if (Number >= voiceNumber || Number < 0) { //如果所需要的数超过了数据个数

MessageBoxA(NULL, "ERROR : Over list !", "ERROR", MB\_ICONHAND);

throw invalid\_argument("ERROR : Over list !");

}

else {

return voiceParagraph[Number];

}

}

void WavFile\_Initial::ShowData(void) //覆盖父类的展示数据函数

{

int max = 0, min = 0;

cout << "TIP : Double data " << endl;

for (unsigned long i = 0; i < this->Get\_dataNumber(); ++i) {

cout << dataDouble[i] << "\t";

if (dataDouble[i] > dataDouble[max]) {

max = i;

}

if (dataDouble[i] < dataDouble[min]) {

min = i;

}

}

cout << endl;

cout << "Max " << dataDouble[max] << endl;

cout << "Min " << dataDouble[min] << endl;

cout << "TIP : Parameter " << endl;

cout << "Max energy " << Get\_maxEnergy() << endl;

cout << "Min energy " << Get\_minEnergy() << endl;

cout << "Max ZCR " << Get\_maxZCR() << endl;

cout << "Min ZCR " << Get\_minZCR() << endl;

/\*

cout << "TIP : Energy " << endl;

for (auto i : dataEnergy) {

cout << i << "\t";

}

cout << endl;

cout << "TIP : ZCR " << endl;

for (auto i : dataZCR) {

cout << i << "\t";

}

cout << endl;

\*/

for (int i = 0; i < (WavFile\_Initial::Get\_dataNumber() - N); ++i) {

cout << Get\_DataEnergy(i) << endl;

}

for (int i = 0; i < (WavFile\_Initial::Get\_dataNumber() - N); ++i) {

cout << Get\_DataZCR(i) << endl;

}

}

void WavFile\_Initial::SaveNewWav(void) //保存去掉空白处的语音文件

{

WavFile::SaveNewWav(voiceNumber, voiceParagraph); //调用父类的生成函数

}

void WavFile\_Initial::Pre\_emphasis(VoiceParagraph voiceParagraph, double \*dataDouble) //对一个段落内的数据进行预加重处理

{

for (unsigned long i = 0; i < voiceParagraph.voiceLength; ++i) {

unsigned long dataIndex = voiceParagraph.begin + i;

if(dataIndex == 0 || dataIndex == this->Get\_dataNumber()) {

continue;

}

dataDouble[dataIndex] = dataDouble[dataIndex] - WavFile\_Initial::preCoefficient \* dataDouble[dataIndex - 1]; //加一阶数字滤波器

}

}

bool WavFile\_Initial::Frame\_Data(double \*data, unsigned long index, double\* dataSpace, int dataSpaceSize) //获取端点检测后第index帧的分帧加窗操作

{

if (dataSpaceSize < WavFile\_Initial::N) { //预分配的空间不足一帧时

return false;

}

VoiceParagraph voiceParagraph(-1, -1, -1);

for (unsigned long i = 0; i < this->voiceParagraph.size(); ++i) {

if ((int)(index - this->Get\_frameNumber(this->voiceParagraph[i])) <= 0) { //如果减去此段落的帧数，数据小于0，则此帧为当前数据段

voiceParagraph = this->voiceParagraph[i];

break;

}

index = index - this->Get\_frameNumber(this->voiceParagraph[i]);

}

if (voiceParagraph.begin == -1 || voiceParagraph.end == -1 || voiceParagraph.voiceLength == -1) {

return false;

}

unsigned long begin = voiceParagraph.begin + (index - 1) \* WavFile\_Initial::FrameShift;

unsigned long end = begin + WavFile\_Initial::N - 1;

unsigned long voiceLength = WavFile\_Initial::N;

if (end >= voiceParagraph.end) {

end = voiceParagraph.end;

voiceLength = end - begin + 1;

}

voiceParagraph.begin = begin;

voiceParagraph.end = end;

voiceParagraph.voiceLength = voiceLength;

for (unsigned long i = voiceParagraph.begin; i <= voiceParagraph.end; ++i) {

dataSpace[i - voiceParagraph.begin] = data[i] \* this->Hamming\_window(i - voiceParagraph.begin); //加窗功能

}

return true;

}

bool WavFile\_Initial::Frame\_Data(double \*data, double dataSize, unsigned long index, double\* dataSpace, int dataSpaceSize) //对部分数据进行分帧加窗操作

{

if (dataSpaceSize < WavFile\_Initial::N) { //预分配的空间不足一帧时

return false;

}

unsigned long frameNumber = this->Get\_frameNumber(dataSize);

if (index < 1 || index > frameNumber) { //帧位不属于数据段允许范围内

return false;

}

unsigned long begin = (index - 1) \* WavFile\_Initial::FrameShift;

unsigned long end = begin + WavFile\_Initial::N - 1;

unsigned long voiceLength = WavFile\_Initial::N;

if (index == frameNumber && end != dataSize) { //如果长度不为整帧

if (end < dataSize) { //全部数据多余

} else if (end > dataSize) { //全部数据缺少

end = (unsigned long)(dataSize - 1);

voiceLength = (unsigned long)(dataSize - begin);

}

}

VoiceParagraph voiceParagraph(begin, end, voiceLength);

for (unsigned long i = voiceParagraph.begin; i <= voiceParagraph.end; ++i) {

dataSpace[i - voiceParagraph.begin] = data[i] \* this->Hamming\_window(i - voiceParagraph.begin); //加窗功能

}

return true;

}

bool WavFile\_Initial::Endpoint\_Detection(void) //端点检测函数

{

//this->Frame\_Energy(); //计算短时帧能量

//this->Frame\_ZCR(); //计算短时过零率

this->Frame\_EnergyZcr(); //计算短时帧能量与短时过零率

energyHigh = 10; //初始化短时帧能量高门限

energyLow = 2; //初始化短时帧能量低门限

zcrHigh = 10; //初始化短时过零率高门限

zcrLow = 5; //初始化短时过零率低门限

energyHigh = min(energyHigh, Get\_maxEnergy() / 4); //调整短时帧能量高门限

energyLow = min(energyLow, Get\_maxEnergy() / 8); //调整短时帧能量低门限

zcrHigh = min(zcrHigh, Get\_maxZCR() / 4); //调整短时帧能量高门限

zcrLow = min(zcrLow, Get\_maxZCR() / 8); //调整短时帧能量低门限

int statusFlag = 0; //设置语音状态标志

unsigned long begin = 0; //语音段落的起点

unsigned long end = 0; //语音段落的终点

unsigned long voiceLength = 0; //语音段落的长度

unsigned long silence = 0; //静音段落的长度

voiceNumber = 0;

for (unsigned long i = 0, frame = 0; i < this->Get\_dataNumber() - N; ++i) { //遍历每一帧

frame = (i - N) / WavFile\_Initial::FrameShift + 1;

if (i <= 256){

frame = 0;

}

switch (statusFlag)

{

case MUTEPARAGRAPH:

case INTERIMPARAGRAPH:

if (Get\_DataEnergy(frame) > energyHigh) { //帧能量大于能量高门限,进入语音段

begin = (unsigned long) max((int)(i - voiceLength - 1), 0);

statusFlag = VOICEPARAGRAPH;

voiceLength++;

silence = 0;

}

else if (Get\_DataEnergy(frame) > energyLow || Get\_DataZCR(frame) > zcrLow) { //过渡段

statusFlag = INTERIMPARAGRAPH;

voiceLength++;

}

else { //静音段

statusFlag = MUTEPARAGRAPH;

voiceLength = 0;

}

break;

case VOICEPARAGRAPH:

if (Get\_DataEnergy(frame) > Get\_minEnergy() || Get\_DataZCR(frame) > Get\_minZCR()) { //保持在语音段

voiceLength++;

}

else { //语音将结束

silence++;

if (silence < minSilence) { //静音还不够长，尚未结束

voiceLength++;

}

else {

if (voiceLength < minVoiceLength) { //语音段长度太短，认为是噪声

statusFlag = MUTEPARAGRAPH;

silence = 0;

voiceLength = 0;

}

else { //语音结束

statusFlag = OVERPARAGRAPH;

end = max(begin + voiceLength, 0);

}

}

}

break;

case OVERPARAGRAPH:

voiceParagraph.push\_back(VoiceParagraph(begin, end, voiceLength)); //保存语音段落信息

voiceLength = 0;

voiceNumber++; //语音段落+1

statusFlag = MUTEPARAGRAPH;

break;

default:

MessageBoxA(NULL, "ERROR : Status failure !", "ERROR", MB\_ICONHAND);

return false;

break;

}

}

if (statusFlag == VOICEPARAGRAPH) { //说明语音信号还没有结束，以当前记录下的最后一个点为终点保存语音段

end = begin + voiceLength;

voiceParagraph.push\_back(VoiceParagraph(begin, end, voiceLength));

++voiceNumber;

}

if (voiceNumber == 0 && voiceParagraph.size() == 0) { //说明没有检测到语音段落，直接将整段语义合成为一个语音段落

end = 0 + voiceLength;

voiceParagraph.push\_back(VoiceParagraph(0, end, voiceLength));

++voiceNumber;

}

cout << "TIP : Voice number is " << voiceNumber << endl;

return true;

}

#pragma once

#include "WavFile\_Initial.h"

class CharaParameter

{

private:

unsigned long frameNumber; //帧数量

double \*\*frameData; //帧数据

double \*\*frameFFTParameter; //帧数据通过快速傅里叶变换后的数据

double \*\*frameMelParameter; //FFT数据通过Mel频率滤波器组后的数据

//底层内存操作

double \*DistributionSpace(unsigned long col); //分配一维数组空间

double \*\*DistributionSpace(unsigned long row, unsigned long col); //分配二维数组空间

void DestorySpace(double\* space); //销毁一维数组空间

void DestorySpace(double\*\* space, unsigned long row); //销毁二位数组空间

//数据校对操作

void ShowDataValue(bool showOnTerminal = true); //显示求值过程中的数据并保存在文件中

//傅里叶变换操作

unsigned long fftNumber; //傅里叶变换后的数据的个数

double\* FFT(double \*data, unsigned long dataNumber); //快速离散傅立叶变换，无虚部 (严重警告，因为可能出现扩展内存的情况，所以必须返回新地址)

bool IFFT(double \*data, unsigned long dataNumber); //快速离散逆傅立叶变换，无虚部

bool DCT(double \*\*dataEngrgy, double \*\*dataRet, unsigned long row, unsigned long col, int degree); //离散余弦变换

public:

static const int MelCoefficient; //计算mel频率的系数，简称mel系数

static const int MelFilterNumber; //mel频率滤波器组的滤波器个数

static const int MelDegreeNumber; //mel频率的阶数

CharaParameter(unsigned long frameNumber) {

this->frameNumber = frameNumber;

this->frameData = new double\* [this->frameNumber]; //分配指针数组的空间

}

bool Push\_data(unsigned long index, double \*frame); //初始化特征参数类使用，将index帧的数据存放如类内

unsigned long Get\_frameNumber(); //获取帧数量

double\* Get\_frameMelParameter(unsigned long row); //获取Mel特征参数中的第Row行

double Get\_frameMelParameter(unsigned long row, unsigned long col); //获取Mel特征参数中的[row][col]

//特征参数求解列表

double\*\* MFCC\_CharaParameter(unsigned long sampleRate); //求解MFCC特诊参数

};

#include "../stdafx.h"

#include "WavData\_CharaParameter.h"

const int CharaParameter::MelCoefficient = 1125; //计算mel频率的系数，简称mel系数 或是哟个2595

const int CharaParameter::MelFilterNumber = 24; //mel频率滤波器组的滤波器个数

const int CharaParameter::MelDegreeNumber = 13; //mel频率的阶数

double\*\* CharaParameter::MFCC\_CharaParameter(unsigned long sampleRate) //求解MFCC特诊参数

{

cout << "TIP : Calculate MFCC parameter ..." << endl;

//1.拷贝数据，保存原先分帧后的数据

this->frameFFTParameter =

this->DistributionSpace(this->frameNumber, WavFile\_Initial::N); //分配FFT保存的内存

for (unsigned long i = 0; i < this->frameNumber; ++i) {

memcpy(this->frameFFTParameter[i], this->frameData[i],

sizeof(double) \* WavFile\_Initial::N); //拷贝帧数据，预备后期计算FFT

}

//2.帧内求解快速傅里叶变换数据

for (unsigned long i = 0; i < this->frameNumber; ++i) {

double \*space =

this->FFT(this->frameFFTParameter[i], WavFile\_Initial::N); //保存下返回的地址，因为这有可能是新数据的地址

if (space != NULL) {

this->frameFFTParameter[i] = space;

}

}

//3.求频谱的平方，得到谱线的能量

double\*\* spectrumEnergy =

this->DistributionSpace(this->frameNumber, WavFile\_Initial::N); //为能量谱新建地址，但不需保存

for (unsigned long i = 0; i < this->frameNumber; ++i) {

for (int j = 0; j < WavFile\_Initial::N; ++j) {

spectrumEnergy[i][j] = pow(this->frameFFTParameter[i][j], 2); //平方求谱线能量

}

}

//4.将数据通过Mel滤波器组

// 1)求最大/最小mel频率，及滤波器的中心间距

double melFreMax = CharaParameter::MelCoefficient \*

log((long double)1 + (sampleRate / 2) / 700); //求最大mel频率

double melFreMin = CharaParameter::MelCoefficient \*

log((long double)1); //求最小mel频率

double centerSpace = (melFreMax - melFreMin) /

(CharaParameter::MelFilterNumber + 1); //求姐mel滤波器组中每个滤波器的中心距离

// 2)求mel滤波器中每个滤波器的实际频率值

double \* melFrequency =

this->DistributionSpace(CharaParameter::MelFilterNumber + 2); //为mel滤波器组的实际频率创建内存空间,但不需要保存

for (int i = 0; i < CharaParameter::MelFilterNumber + 2; ++i) {

melFrequency[i] =

floor(((WavFile\_Initial::N + 1) \* 700 \*

(exp((melFreMin + i \* centerSpace) / CharaParameter::MelCoefficient) - 1))

/ sampleRate); //求解每个滤波器的实际频率

}

// 3)将线谱能量通过mel滤波器组

double\*\* melEnergy =

this->DistributionSpace(this->frameNumber, CharaParameter::MelFilterNumber); //分配保存Mel特诊参数的内存

double tempCoefficient = 0;

for (unsigned long i = 0; i < this->frameNumber; ++i) {

for (int j = 1; j <= CharaParameter::MelFilterNumber; ++j) {

tempCoefficient = 0;

for (int k = 0; k < WavFile\_Initial::N; ++k) { //将数据频率与实际频率对应起来

if (k >= melFrequency[j - 1] && k <= melFrequency[j]) {

tempCoefficient = (k - melFrequency[j - 1]) \* (melFrequency[j] - melFrequency[j - 1]);

} else if (k >= melFrequency[j] && k <= melFrequency[j + 1]) {

tempCoefficient = (melFrequency[j + 1] - k) \* (melFrequency[j + 1] - melFrequency[j]);

}

melEnergy[i][j - 1] += this->frameFFTParameter[i][k]\* tempCoefficient;

}

}

}

//5.求对数

for (unsigned long i = 0; i < this->frameNumber; ++i) {

for (int j = 0; j < CharaParameter::MelFilterNumber; ++j) {

if (melEnergy[i][j] < 0) { //处理数据为负数的情况，以免出现无效数据

melEnergy[i][j] = log(fabs(melEnergy[i][j])) \* -1;

} else {

melEnergy[i][j] = log(melEnergy[i][j]);

}

}

}

//6.进行离散余弦变换

this->frameMelParameter = this->DistributionSpace(this->frameNumber, CharaParameter::MelDegreeNumber);

this->DCT(melEnergy, this->frameMelParameter, this->frameNumber,

CharaParameter::MelFilterNumber, CharaParameter::MelDegreeNumber);

//销毁内存

this->DestorySpace(melEnergy, this->frameNumber);

this->DestorySpace(melFrequency);

this->DestorySpace(spectrumEnergy, this->frameNumber);

return this->frameMelParameter;

}

#include "../stdafx.h"

#include "WavData\_CharaParameter.h"

double\* CharaParameter::DistributionSpace(unsigned long col) //分配一维数组空间

{

double \*tempspace = new double[col];

if (tempspace == NULL) {

throw invalid\_argument("ERROR : Memory failure !");

return false;

} else {

for (unsigned long i = 0; i < col; ++i) {

tempspace[i] = 0;

}

}

return tempspace;

}

double\*\* CharaParameter::DistributionSpace(unsigned long row, unsigned long col) //分配二维数组空间

{

double \*\*tempspace = new double\* [row];

if (tempspace == NULL) {

throw invalid\_argument("ERROR : Memory failure !");

return false;

}

for (unsigned long i = 0; i < row; ++i) {

tempspace[i] = this->DistributionSpace(col);

if (tempspace[i] == NULL) {

throw invalid\_argument("ERROR : Memory failure !");

return false;

}

}

return tempspace;

}

void CharaParameter::DestorySpace(double \*space) //销毁一维数组空间

{

delete space;

}

void CharaParameter::DestorySpace(double \*\*space, unsigned long row) //销毁二位数组空间

{

for (unsigned long i = 0; i < row; ++i) {

delete space[i];

}

delete space;

}

void CharaParameter::ShowDataValue(bool showOnTerminal) //显示求值过程中的数据并保存在文件中

{

cout << "TIP : This is orgin data :" << endl;

for (unsigned long i = 0; i < this->frameNumber; ++i) {

for (int j = 0; j < WavFile\_Initial::N; ++j) {

cout << this->frameData[i][j] << "\t";

}

cout << endl;

}

cout << endl;

cout << "TIP : This is FFT past data :" << endl;

for (unsigned long i = 0; i < this->frameNumber; ++i) {

for (int j = 0; j < WavFile\_Initial::N; ++j) {

cout << this->frameFFTParameter[i][j] << "\t";

}

cout << endl;

}

cout << endl;

cout << "TIP : This is Mel data :" << endl;

for (unsigned long i = 0; i < this->frameNumber; ++i) {

for (int j = 0; j < CharaParameter::MelDegreeNumber; ++j) {

cout << this->frameMelParameter[i][j] << "\t";

}

cout << endl;

}

}

double\* CharaParameter::FFT(double \*data, unsigned long dataNumber) //快速离散傅立叶变换，无虚部

{

short power = 0;

for (int i = dataNumber - 1; i > 0; ++power, i /= 2);

fftNumber = (unsigned long)pow((float)2, (int)power);

double \*dataFFT = DistributionSpace(fftNumber);

for (unsigned long i = 0; i < dataNumber; i++) {

dataFFT[i] = data[i]; //初始化快速傅立叶变换数据

}

double \*W = DistributionSpace(fftNumber / 2); //计算旋转因子

for (unsigned long i = 0; i < fftNumber / 2; i++) {

W[i] = cos(2 \* i \* WavFile\_Initial::PI / fftNumber);

}

unsigned long ulGroupLength = 1; //段的长度

unsigned long ulHalfLength = 0; //段长度的一半

unsigned long ulGroupCount = 0; //段的数量

double cw, c1, c2;

for (short b = 0; b < power; b++) //计算FFT

{

ulHalfLength = ulGroupLength;

ulGroupLength \*= 2;

for (unsigned long j = 0; j < fftNumber; j += ulGroupLength)

{

for (unsigned long k = 0; k < ulHalfLength; k++)

{

cw = W[k \* fftNumber / ulGroupLength], dataFFT[j + k + ulHalfLength];

c1 = dataFFT[j + k] + cw;

c2 = dataFFT[j + k] - cw;

dataFFT[j + k] = c1;

dataFFT[j + k + ulHalfLength] = c2;

}

}

}

free(W);

if (fftNumber == dataNumber) { //如何FFT运算时个数正好，没有扩展，则需要拷贝数据

for (unsigned long i = 0; i < dataNumber; ++i) {

data[i] = dataFFT[i];

}

free(dataFFT);

return NULL;

}

else { //否则需要返回新的数据地址

free(data);

return dataFFT;

}

}

bool CharaParameter::IFFT(double \*data, unsigned long dataNumber) //快速离散逆傅立叶变换，无虚部

{

short power = 0;

double \*dataIFFT;

for (unsigned long i = dataNumber - 1; i > 0; ++power, i /= 2); //计算次幂

for (unsigned long i = 1, j = dataNumber / 2; i < (unsigned long)WavFile\_Initial::N - 1; ++i) { //雷德算法重排位置

if (i < j) {

double temp = data[i];

data[i] = data[j];

data[j] = temp;

}

unsigned long k = dataNumber / 2;

while (k <= j) {

j = j - k;

k = k / 2;

}

j = j + k;

}

dataIFFT = DistributionSpace(dataNumber);

for (unsigned long i = 0; i < dataNumber; ++i) { //逆操作

dataIFFT[i] = data[i] / dataNumber;

}

for (short p = 0; p < power; ++p) { //IFFT计算

for (unsigned long i = 0; i < pow((long double)2, (long double)p); ++i) {

double cw = i\*pow((long double)2, (long double)power - (p + 1)); //计算旋转因子

for (unsigned long j = i; j < dataNumber - 1; j += (unsigned long)pow((long double)2, (long double)(p + 1))) {

unsigned long index = j + (unsigned long)pow((long double)2, (long double)p);

cw = dataIFFT[index] \* cos(2 \* WavFile\_Initial::PI \* cw / dataNumber);

dataIFFT[index] = dataIFFT[j] - cw;

dataIFFT[j] = dataIFFT[j] + cw;

}

}

}

for (unsigned long i = 0; i < dataNumber; ++i) { //拷贝数据到自身

data[i] = dataIFFT[i];

}

free(dataIFFT);

return true;

}

bool CharaParameter::DCT(double \*\*dataEngrgy, double \*\*dataRet, unsigned long row, unsigned long col, int degree) //离散余弦变换

{

for (unsigned long i = 0; i < row; ++i) {

for (int j = 0; j < degree; ++j) {

for (unsigned long k = 0; k < col; ++k) {

dataRet[i][j] += dataEngrgy[i][k] \* cos(WavFile\_Initial::PI \* j \* (0.5 + k) / (col)); //计算公式

}

}

}

return true;

}

unsigned long CharaParameter::Get\_frameNumber() //获取帧数量

{

return this->frameNumber;

}

double\* CharaParameter::Get\_frameMelParameter(unsigned long row) //获取Mel特征参数中的第Row行

{

if (row < 0 || row >= this->frameNumber) {

return NULL;

}

return this->frameMelParameter[row];

}

double CharaParameter::Get\_frameMelParameter(unsigned long row, unsigned long col) //获取Mel特征参数中的[row][col]

{

if (row < 0 || row >= this->frameNumber ||

col < 0 || col >= (unsigned long) CharaParameter::MelDegreeNumber) {

return NULL;

}

return this->frameMelParameter[row][col];

}

bool CharaParameter::Push\_data(unsigned long index, double \*frame) { //初始化特征参数类使用，将index帧的数据存放如类内

if (index < 1 || index > frameNumber) {

return false;

}

this->frameData[index - 1] = frame;

return true;

}

// Voiceprint RecognitionDlg.cpp : 实现文件

//

#include "stdafx.h"

#include "Voiceprint Recognition.h"

#include "Voiceprint RecognitionDlg.h"

#include "afxdialogex.h"

#ifdef \_DEBUG

#define new DEBUG\_NEW

#endif

// 用于应用程序“关于”菜单项的 CAboutDlg 对话框

class CAboutDlg : public CDialogEx

{

public:

CAboutDlg();

// 对话框数据

enum { IDD = IDD\_ABOUTBOX };

protected:

virtual void DoDataExchange(CDataExchange\* pDX); // DDX/DDV 支持

// 实现

protected:

DECLARE\_MESSAGE\_MAP()

};

CAboutDlg::CAboutDlg() : CDialogEx(CAboutDlg::IDD)

{

}

void CAboutDlg::DoDataExchange(CDataExchange\* pDX)

{

CDialogEx::DoDataExchange(pDX);

}

BEGIN\_MESSAGE\_MAP(CAboutDlg, CDialogEx)

END\_MESSAGE\_MAP()

// CVoiceprintRecognitionDlg 对话框

CVoiceprintRecognitionDlg::CVoiceprintRecognitionDlg(CWnd\* pParent /\*=NULL\*/)

: CDialogEx(CVoiceprintRecognitionDlg::IDD, pParent)

{

m\_hIcon = AfxGetApp()->LoadIcon(IDR\_MAINFRAME);

}

void CVoiceprintRecognitionDlg::DoDataExchange(CDataExchange\* pDX)

{

CDialogEx::DoDataExchange(pDX);

DDX\_Control(pDX, IDC\_SHOCKWAVEFLASH1, flashshow);

DDX\_Control(pDX, IDC\_LIST1, listCtrl\_1);

DDX\_Control(pDX, IDC\_LIST2, listCtrl\_2);

DDX\_Control(pDX, IDC\_BUTTON1, buttonCtrl\_1);

}

BEGIN\_MESSAGE\_MAP(CVoiceprintRecognitionDlg, CDialogEx)

ON\_WM\_SYSCOMMAND()

ON\_WM\_PAINT()

ON\_WM\_QUERYDRAGICON()

ON\_BN\_CLICKED(IDC\_BUTTON1, &CVoiceprintRecognitionDlg::OnBnClickedButton1)

ON\_BN\_CLICKED(IDC\_BUTTON2, &CVoiceprintRecognitionDlg::OnBnClickedButton2)

ON\_BN\_CLICKED(IDC\_BUTTON3, &CVoiceprintRecognitionDlg::OnBnClickedButton3)

ON\_BN\_CLICKED(IDC\_BUTTON4, &CVoiceprintRecognitionDlg::OnBnClickedButton4)

ON\_BN\_CLICKED(IDC\_BUTTON5, &CVoiceprintRecognitionDlg::OnBnClickedButton5)

END\_MESSAGE\_MAP()

// CVoiceprintRecognitionDlg 消息处理程序

BOOL CVoiceprintRecognitionDlg::OnInitDialog()

{

CDialogEx::OnInitDialog();

// 将“关于...”菜单项添加到系统菜单中。

// IDM\_ABOUTBOX 必须在系统命令范围内。

ASSERT((IDM\_ABOUTBOX & 0xFFF0) == IDM\_ABOUTBOX);

ASSERT(IDM\_ABOUTBOX < 0xF000);

CMenu\* pSysMenu = GetSystemMenu(FALSE);

if (pSysMenu != NULL)

{

BOOL bNameValid;

CString strAboutMenu;

bNameValid = strAboutMenu.LoadString(IDS\_ABOUTBOX);

ASSERT(bNameValid);

if (!strAboutMenu.IsEmpty())

{

pSysMenu->AppendMenu(MF\_SEPARATOR);

pSysMenu->AppendMenu(MF\_STRING, IDM\_ABOUTBOX, strAboutMenu);

}

}

// 设置此对话框的图标。当应用程序主窗口不是对话框时，框架将自动

// 执行此操作

SetIcon(m\_hIcon, TRUE); // 设置大图标

SetIcon(m\_hIcon, FALSE); // 设置小图标

// TODO: 在此添加额外的初始化代码

CRect rectCtrl;

GetDlgItem(IDC\_SHOCKWAVEFLASH1)->GetWindowRect(&rectCtrl);

rectCtrl.left = 35;

rectCtrl.right = 217;

rectCtrl.top = 45;

rectCtrl.bottom = 200;

flashshow.MoveWindow(&rectCtrl, true);

TCHAR strCurDrt[500];

int nLen = ::GetCurrentDirectory(500, strCurDrt);

if (strCurDrt[nLen] != '\\') {

strCurDrt[nLen++] = '\\';

strCurDrt[nLen] = '\0';

}

CString strFileName = strCurDrt;

strFileName += "flash.swf";

this->flashshow.LoadMovie(0, strFileName);

this->flashshow.Play();

this->flashshow.Stop();

SetWindowPos(&this->flashshow,0,0,0,0, SWP\_NOMOVE | SWP\_NOSIZE);

CRect rect;

this->listCtrl\_1.GetHeaderCtrl()->EnableWindow(false); //固定标题不被移动

listCtrl\_1.GetClientRect(&rect); //获取编程语言列表视图控件的位置和大小

listCtrl\_1.SetExtendedStyle(listCtrl\_1.GetExtendedStyle()

| LVS\_EX\_FULLROWSELECT | LVS\_EX\_GRIDLINES); //为列表视图控件添加全行选中和栅格风格

listCtrl\_1.InsertColumn(0, \_T("文件名"), LVCFMT\_CENTER, rect.Width() / 2, 0);

listCtrl\_1.InsertColumn(1, \_T("录音人"), LVCFMT\_CENTER, rect.Width() / 2, 1);

this->listCtrl\_2.GetHeaderCtrl()->EnableWindow(false); //固定标题不被移动

listCtrl\_2.GetClientRect(&rect); //获取编程语言列表视图控件的位置和大小

listCtrl\_2.SetExtendedStyle(listCtrl\_2.GetExtendedStyle()

| LVS\_EX\_FULLROWSELECT | LVS\_EX\_GRIDLINES); //为列表视图控件添加全行选中和栅格风格

listCtrl\_2.InsertColumn(0, \_T("文件名"), LVCFMT\_CENTER, rect.Width() / 2, 0);

listCtrl\_2.InsertColumn(1, \_T("所属人"), LVCFMT\_CENTER, rect.Width() / 2, 1);

this->flagRecord = false;

this->OnBnClickedButton4();

this->OnBnClickedButton5();

return TRUE; // 除非将焦点设置到控件，否则返回 TRUE

}

void CVoiceprintRecognitionDlg::OnSysCommand(UINT nID, LPARAM lParam)

{

if ((nID & 0xFFF0) == IDM\_ABOUTBOX)

{

CAboutDlg dlgAbout;

dlgAbout.DoModal();

}

else

{

CDialogEx::OnSysCommand(nID, lParam);

}

}

// 如果向对话框添加最小化按钮，则需要下面的代码

// 来绘制该图标。对于使用文档/视图模型的 MFC 应用程序，

// 这将由框架自动完成。

void CVoiceprintRecognitionDlg::OnPaint()

{

if (IsIconic())

{

CPaintDC dc(this); // 用于绘制的设备上下文

SendMessage(WM\_ICONERASEBKGND, reinterpret\_cast<WPARAM>(dc.GetSafeHdc()), 0);

// 使图标在工作区矩形中居中

int cxIcon = GetSystemMetrics(SM\_CXICON);

int cyIcon = GetSystemMetrics(SM\_CYICON);

CRect rect;

GetClientRect(&rect);

int x = (rect.Width() - cxIcon + 1) / 2;

int y = (rect.Height() - cyIcon + 1) / 2;

// 绘制图标

dc.DrawIcon(x, y, m\_hIcon);

}

else

{

CDialogEx::OnPaint();

}

}

//当用户拖动最小化窗口时系统调用此函数取得光标

//显示。

HCURSOR CVoiceprintRecognitionDlg::OnQueryDragIcon()

{

return static\_cast<HCURSOR>(m\_hIcon);

}

void CVoiceprintRecognitionDlg::OnBnClickedButton1()

{

// TODO: 在此添加控件通知处理程序代码

if (flagRecord) {

this->OnButton1\_cancel();

this->flashshow.Stop();

this->flagRecord = false;

this->OnBnClickedButton4();

this->OnBnClickedButton5();

SetDlgItemText(IDC\_BUTTON1, (CString)"录音");

SetDlgItemText(IDC\_EDIT1, (CString)"");

} else {

bool success = false;

//弹出窗口并显示

CFileDialog opendlg(FALSE, \_T("\*.wav"), \_T("\*.wav"), OFN\_HIDEREADONLY | OFN\_OVERWRITEPROMPT, \_T("所有文件(\*.wav\*;)|\*.wav\*||"), NULL); //打开文件选择框

if (opendlg.DoModal() == IDOK)

{

CString fileName = opendlg.GetPathName(); //获取选择的文件名

int nameLen = WideCharToMultiByte(CP\_ACP, 0, fileName, -1, NULL, 0, NULL, NULL);

char \*fileNameChar = new char[nameLen + 1];

WideCharToMultiByte(CP\_ACP, 0, fileName, -1, fileNameChar, nameLen, NULL, NULL); //将CString转为char\*

char \*fileNameTemp = new char[nameLen + 20];

int index = 0; //用于保存新文件名长度

for (int i = 0; i < nameLen + 1; ++i) { //处理'\'为'\\'，若文件中的路径分隔符为'\'则无法准确定位

fileNameTemp[index++] = fileNameChar[i];

if (fileNameChar[i] == '\\') {

fileNameTemp[index++] = '\\';

}

}

fileNameTemp[index] = 0;

if (fileNameTemp[index - 1] != 'v' || fileNameTemp[index - 2] != 'a' ||

fileNameTemp[index - 3] != 'w' || fileNameTemp[index - 4] != '.' ) {

fileNameTemp[index ] = '.';

fileNameTemp[index + 1] = 'w';

fileNameTemp[index + 2] = 'a';

fileNameTemp[index + 3] = 'v';

fileNameTemp[index + 4] = 0;

}

::fileName\_t = fileNameTemp;

success = this->OnButton1\_record(fileNameTemp); //赋文件名给线程操作函数

delete fileNameChar;

//delete fileNameTemp; //交给初始化完录音后清空

}

if (success) {

SetDlgItemText(IDC\_BUTTON1, (CString)"停止");

this->flashshow.Play();

this->flagRecord = true;

}

}

}

void CVoiceprintRecognitionDlg::OnBnClickedButton2()

{

// TODO: 在此添加控件通知处理程序代码

int selectIndex = this->GetItemSelect(0);

FILESTRUCT selectItem = this->wavLib[selectIndex];

if (strcmp(selectItem.peopleName.data(), "未知") == 0 ||

strcmp(selectItem.peopleName.data(), "unknow") == 0) {

MessageBoxA(NULL, "未知的用户语音无法训练进入语音库", "错误", MB\_ICONHAND);

return ;

}

char szModuleFilePath[MAX\_PATH];

int n = GetModuleFileNameA(0, szModuleFilePath, MAX\_PATH); //获得当前执行文件的路径

szModuleFilePath[strrchr(szModuleFilePath, '\\') - szModuleFilePath + 1] = 0; //将最后一个"\\"后的字符置为0

int index = 0;

char filePath[MAX\_PATH];

for (int i = 0; i < (int) strlen(szModuleFilePath); ++i) { //补全//

filePath[index++] = szModuleFilePath[i];

if (szModuleFilePath[i] == '\\') {

filePath[index++] = '\\';

}

}

filePath[index++] = 0; //末尾归零

char wavfilePath[MAX\_PATH], gmmfilePath[MAX\_PATH];

strcpy\_s(wavfilePath, filePath);

strcpy\_s(gmmfilePath, filePath);

strcat\_s(wavfilePath, "wavLib\\\\");

strcat\_s(gmmfilePath, "voiceLib\\\\");

strcat\_s(wavfilePath, selectItem.fileName.data());

strcat\_s(gmmfilePath, selectItem.peopleName.data());

strcat\_s(gmmfilePath, "-gmm(-).txt");

if (::charaParameter != NULL) {

delete ::charaParameter;

}

trainingWAV(wavfilePath, gmmfilePath);

char infofilePath[MAX\_PATH];

strcpy\_s(infofilePath, filePath);

strcat\_s(infofilePath, "voiceLib\\\\");

strcat\_s(infofilePath, "info.list");

ofstream out(infofilePath, ios::app);

if (out.is\_open()) {

string str\_f, str\_p;

CChineseCode::GB2312ToUTF\_8(str\_f, (char\*) getFileName(gmmfilePath).data(), getFileName(gmmfilePath).length());

CChineseCode::GB2312ToUTF\_8(str\_p, (char\*) selectItem.peopleName.data(), selectItem.peopleName.length());

out << str\_f.data() << "\t" << str\_p.data() << endl;

}

out.close();

this->OnBnClickedButton5();

}

void CVoiceprintRecognitionDlg::OnBnClickedButton3()

{

// TODO: 在此添加控件通知处理程序代码

char szModuleFilePath[MAX\_PATH];

int n = GetModuleFileNameA(0, szModuleFilePath, MAX\_PATH); //获得当前执行文件的路径

szModuleFilePath[strrchr(szModuleFilePath, '\\') - szModuleFilePath + 1] = 0; //将最后一个"\\"后的字符置为0

int index = 0;

char filePath[MAX\_PATH];

for (int i = 0; i < (int) strlen(szModuleFilePath); ++i) { //补全//

filePath[index++] = szModuleFilePath[i];

if (szModuleFilePath[i] == '\\') {

filePath[index++] = '\\';

}

}

filePath[index++] = 0;

char gmmfilePath[MAX\_PATH];

strcpy\_s(gmmfilePath, filePath);

strcat\_s(gmmfilePath, "voiceLib\\\\");

this->OnBnClickedButton2(); //先训练目标数据

int selectIndex = this->GetItemSelect(0);

FILESTRUCT selectItem = this->wavLib[selectIndex];

if (strcmp(selectItem.peopleName.data(), "未知") == 0 ||

strcmp(selectItem.peopleName.data(), "unknow") == 0) {

return ;

}

int countMax = voiceprintRecognition(gmmfilePath, this->voiceLib);

char VPR\_result[256] = "系统识别结果-说话人为：";

strcat\_s(VPR\_result, this->voiceLib[countMax].peopleName.data());

MessageBoxA(NULL, VPR\_result, "信息", MB\_ICONASTERISK);

}

void CVoiceprintRecognitionDlg::OnBnClickedButton4()

{

// TODO: 在此添加控件通知处理程序代码

this->CompoundFile(this->wavLib, 0);

this->OnButton4\_refresh();

}

void CVoiceprintRecognitionDlg::OnBnClickedButton5()

{

// TODO: 在此添加控件通知处理程序代码

this->CompoundFile(this->voiceLib, 1);

this->OnButton5\_refresh();

}

#include "stdafx.h"

#include "VPR/Model\_GMM.h"

#include "VPR/Model\_KMeans.h"

#include "Voiceprint RecognitionDlg.h"

#include "VPR/WavData\_CharaParameter.h"

WaveRecorder waveRecorder; //全局录音对象

char\* fileName; //文件对比之用

string fileName\_t; //文件对比之用

double\* mfccData; //用于保存当前语音训练出的数据

CharaParameter\* charaParameter; //用于保存当前语音训练出的参数

void CVoiceprintRecognitionDlg::CompoundFile(vector<FILESTRUCT>& fileLib, int flag) //用于将txt信息与当前文件夹下内容相结合

{

char szModuleFilePath[MAX\_PATH];

int n = GetModuleFileNameA(0, szModuleFilePath, MAX\_PATH); //获得当前执行文件的路径

szModuleFilePath[strrchr(szModuleFilePath, '\\') - szModuleFilePath + 1] = 0; //将最后一个"\\"后的字符置为0

int index = 0;

char filePath[MAX\_PATH];

for (int i = 0; i < (int) strlen(szModuleFilePath); ++i) { //补全//

filePath[index++] = szModuleFilePath[i];

if (szModuleFilePath[i] == '\\') {

filePath[index++] = '\\';

}

}

filePath[index++] = 0; //末尾归零

char path[MAX\_PATH];

strcpy\_s(path, filePath);

if (flag == 0) { //按照要求连接文件夹

strcat\_s(path, "wavLib");

} else if (flag == 1) {

strcat\_s(path, "voiceLib");

}

vector<string> files;

getFiles(path, files); //获取文件夹内的所有文件名（路径名）

strcat\_s(path, "\\\\info.list"); //指定文件夹的读取文件

ifstream in(path);

fileLib.clear();

readList(in, fileLib); //读取list文件内容

in.close();

vector<FILESTRUCT> newLib;

for (int i = 0, j = 0; i < (int) files.size(); ++i) {

string fileName = files[i];

if (strcmp(getFileName(fileName).data(), "info.list") == 0) { //如果当前文件的文件名为info.list说明其为配置文件

continue;

}

for (j = 0; j < (int) fileLib.size(); ++j) {

if (strcmp(getFileName(fileName).data(), fileLib[j].fileName.data()) == 0) { //如果当前文件名与列表名相匹配，则说明这是一个不变的列表项

newLib.push\_back(fileLib[j]);

break;

}

}

if (j == fileLib.size()) {

FILESTRUCT item;

item.fileName = getFileName(fileName).data();

if (flag == 0) {

if (::fileName != NULL && (strcmp(::fileName\_t.data(), fileName.data()) == 0)) { //如果录音文件路径与当前这个未知的文件路径相同

CString str;

GetDlgItem(IDC\_EDIT1)->GetWindowText(str); //获取所属人名

string tempstr = CStringA(str);

if (tempstr.size() == 0) { //所属人名没写

item.peopleName = "未知";

} else {

item.peopleName = tempstr;

}

} else {

item.peopleName = "未知";

}

} else if (flag == 1) {

item.peopleName = "未知";

}

newLib.push\_back(item);

}

}

ofstream wavOut(path);

writeList(wavOut, newLib); //重新写入数据

wavOut.close();

fileLib.clear();

fileLib = newLib;

}

int CVoiceprintRecognitionDlg::GetItemSelect(int index) //获取某个listControl当前选中项的行号

{

int count = 0;

switch (index)

{

case 0 : count = this->listCtrl\_1.GetItemCount(); break;

case 1 : count = this->listCtrl\_2.GetItemCount(); break;

default : break;

}

for (int i = 0; i < count; ++i) {

switch (index)

{

case 0 :

if (this->listCtrl\_1.GetItemState(i, LVIS\_SELECTED) == LVIS\_SELECTED) {

return i;

}

break;

case 1 :

if (this->listCtrl\_2.GetItemState(i, LVIS\_SELECTED) == LVIS\_SELECTED) {

return i;

}

break;

default : break;

}

}

return -1;

}

bool CVoiceprintRecognitionDlg::OnButton1\_record(char\* fileName) //开启录音线程

{

::fileName = fileName;

pthread\_attr\_t attr; //线程属性结构体，创建线程时加入的参数

pthread\_attr\_init(&attr); //初始化

pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_JOINABLE); //是设置你想要指定线程属性参数

int ret = pthread\_create(&thread\_recordID, &attr, record, (void\*)&fileName);

if(ret != 0) {

MessageBoxA(NULL, "ERROR : Can't create thread !", "ERROR", MB\_ICONHAND);

return false;

}

return true;

}

bool CVoiceprintRecognitionDlg::OnButton1\_cancel() //结束录音

{

::waveRecorder.Stop();

::waveRecorder.Reset();

pthread\_cancel(this->thread\_recordID);

return true;

}

bool CVoiceprintRecognitionDlg::OnButton4\_refresh() //录音文件刷新

{

CString str\_f, str\_p;

listCtrl\_1.DeleteAllItems();

for (int i = 0; i < (int) this->wavLib.size(); ++i) {

FILESTRUCT item = this->wavLib[i];

str\_f = item.fileName.c\_str();

str\_p = item.peopleName.c\_str();

listCtrl\_1.InsertItem(i, str\_f); //设置列表文件名信息

listCtrl\_1.SetItemText(i, 1, str\_p); //设置列表用户信息

}

return true;

}

bool CVoiceprintRecognitionDlg::OnButton5\_refresh() //模型文件刷新

{

CString str\_f, str\_p;

listCtrl\_2.DeleteAllItems();

for (int i = 0; i < (int) this->voiceLib.size(); ++i) {

FILESTRUCT item = this->voiceLib[i];

str\_f = item.fileName.c\_str();

str\_p = item.peopleName.c\_str();

listCtrl\_2.InsertItem(i, str\_f); //设置列表文件名信息

listCtrl\_2.SetItemText(i, 1, str\_p);; //设置列表用户信息

}

return true;

}

void\* record(void\* args) //录音线程

{

::waveRecorder.set\_FileName((char\*)fileName);

::waveRecorder.Start();

delete fileName;

return NULL;

}

string getFileName(string path) //将某个路径转换为某个文件名

{

return path.substr(path.rfind('\\') + 1, path.size() - path.rfind('\\') - 1);

}

void getFiles(string path, vector<string>& files) //获取path文件夹下的所有文件名

{

long hFile = 0;

struct \_finddata\_t fileinfo;

string p;

if((hFile = \_findfirst(p.assign(path).append("\\\*").c\_str(), &fileinfo)) != -1) {

do {

if((fileinfo.attrib & \_A\_SUBDIR)) { //判断是否为文件夹

if(strcmp(fileinfo.name, ".") != 0 && strcmp(fileinfo.name, "..") != 0) {

getFiles(p.assign(path).append("\\").append(fileinfo.name), files);

}

} else {

files.push\_back(p.assign(path).append("\\\\").append(fileinfo.name));

}

} while(\_findnext(hFile, &fileinfo) == 0);

\_findclose(hFile);

}

}

void readList(ifstream& in, vector<FILESTRUCT>& list) //读取文件的内容到list中

{

char buffer[512];

string str\_f, str\_p;

char fileName[256], peopleName[256];

if (in.is\_open()) {

while (!in.eof()) {

in.getline(buffer, 512);

if (strlen(buffer) == 0) { //防止到了最后一行只是一个换行还重复读取

continue;

}

sscanf(buffer, "%s %s", &fileName, &peopleName); //格式化字符串

CChineseCode::UTF\_8ToGB2312(str\_f, fileName, strlen(fileName));

CChineseCode::UTF\_8ToGB2312(str\_p, peopleName, strlen(peopleName));

FILESTRUCT item(str\_f, str\_p);

list.push\_back(item);

memset(buffer, 0, 512);

}

}

}

void writeList(ofstream& out, vector<FILESTRUCT>& list) //将list文件内容写入数据流

{

string str\_f, str\_p;

if (out.is\_open()) {

for (int i = 0; i < (int) list.size(); ++i) {

CChineseCode::GB2312ToUTF\_8(str\_f, (char\*) list[i].fileName.data(), list[i].fileName.length());

CChineseCode::GB2312ToUTF\_8(str\_p, (char\*) list[i].peopleName.data(), list[i].peopleName.length());

out << str\_f.data() << "\t" << str\_p.data() << endl;

}

}

}

bool extractParameter(string wavfilePath) //训练目标路径的语音文件的特征参数

{

FILE \*fp;

if ((fp = fopen(wavfilePath.data(), "rb")) == NULL) { //打开语音文件

cout << "ERROR : File open failed !" << endl;

return false;

}

//Todo 初始化语音文件类 读取语音文件数据

WavFile\_Initial \*wavFile = new WavFile\_Initial(fp); //读取语音文件数据

fclose(fp);

for (unsigned long i = 0; i < wavFile->Get\_voiceNumber(); ++i) {

wavFile->Pre\_emphasis(wavFile->Get\_dataVoicePoint(i), wavFile->Get\_WavFileData()); //对可用范围内的数据进行预加重

}

//Todo 初始化特征参数类 计算语音数据特征参数

double \*dataSpace = NULL;

CharaParameter \*charaParameter = new CharaParameter(wavFile->Get\_frameNumber()); //初始化特诊参数类

for (unsigned long i = 1; i <= wavFile->Get\_frameNumber(); ++i) { //逐帧遍历

dataSpace = new double[WavFile\_Initial::N]; //新建帧数据空间

memset(dataSpace, 0, sizeof(double) \* WavFile\_Initial::N);

wavFile->Frame\_Data(wavFile->Get\_WavFileData(), i, dataSpace, WavFile\_Initial::N); //分帧并加窗

charaParameter->Push\_data(i, dataSpace); //将分帧完成的数据保存进特征参数备用

}

unsigned long sampleRate = wavFile->Get\_SampleRate();

delete wavFile;

//Todo 计算MFCC参数

charaParameter->MFCC\_CharaParameter(sampleRate); //计算MFCC特征参数

//Todo 初始化Kmeans数据

::mfccData = new double[charaParameter->Get\_frameNumber() \* CharaParameter::MelDegreeNumber];

for (unsigned long i = 0; i < charaParameter->Get\_frameNumber(); ++i) {

memcpy(&::mfccData[i \* CharaParameter::MelDegreeNumber],

charaParameter->Get\_frameMelParameter(i), sizeof(double) \* CharaParameter::MelDegreeNumber); //拷贝mfcc数据到一段连续的存储空间中备用

}

::charaParameter = charaParameter;

return true;

}

bool trainingWAV(string wavfilePath, string gmmfilePath) //训练wav文件

{

bool success = extractParameter(wavfilePath);

if (!success) {

return false;

}

//Todo 开始Kmeans聚类操作

KMeans\* kmeans = new KMeans(CharaParameter::MelDegreeNumber, KMeans::ClusterNumber); //使用阶数跟簇数初始化Kmeans类

int\* labels = new int[::charaParameter->Get\_frameNumber()];

kmeans->SetInitMode(KMeans::InitUniform); //设置数据的初始化方法

kmeans->Cluster(::mfccData, ::charaParameter->Get\_frameNumber(), labels);//开始聚类

//Todo 初始化GMM数据

double \*\*test\_data = new double\*[KMeans::ClusterNumber];

for (int i = 0; i < KMeans::ClusterNumber; ++i) {

test\_data[i] = new double[CharaParameter::MelDegreeNumber];

double \*tempSpace = kmeans->GetMean(i);

for (int j = 0; j < CharaParameter::MelDegreeNumber; ++j) {

test\_data[i][j] = tempSpace[j];

}

}

delete[]labels;

delete kmeans;

//Todo GMM训练数据

GMM \*gmm = new GMM(CharaParameter::MelDegreeNumber, GMM::SGMNumber);

gmm->Train(::mfccData, ::charaParameter->Get\_frameNumber()); //GMM训练数据

//Todo save GMM to file

ofstream gmm\_file(gmmfilePath.data());

assert(gmm\_file);

gmm\_file << \*gmm;

gmm\_file.close();

delete gmm;

return true;

}

int voiceprintRecognition(string rootPath, vector<FILESTRUCT> voiceLib) //声纹识别

{

char filePath[MAX\_PATH];

GMM \*\*gmmLib = new GMM\*[voiceLib.size()];

ifstream \*gmm\_file = new ifstream[voiceLib.size()];

for (int i = 0; i < (int) voiceLib.size(); ++i) {

strcpy\_s(filePath, rootPath.data());

strcat\_s(filePath, voiceLib[i].fileName.data());

gmm\_file[i].open(filePath);

assert(gmm\_file[i]);

gmmLib[i] = new GMM(CharaParameter::MelDegreeNumber, GMM::SGMNumber);

gmm\_file[i] >> \*gmmLib[i];

gmm\_file[i].close();

}

//Todo 识别计算

cout << "TIP : Begin reservation ..." << endl;

double \*libProbability = new double[voiceLib.size()];

for (int i = 0; i < (int) voiceLib.size(); ++i) {

libProbability[i] = 0;

for (unsigned long j = 0; j < charaParameter->Get\_frameNumber(); ++j) { //计算当前GMM下，目标特征参数集在GMM模型下的概率密度

double tempData = gmmLib[i]->GetProbability(charaParameter->Get\_frameMelParameter(j)); //获取GMM的数值

if (tempData > 0) { //取对数操作

tempData = log10(tempData);

}

libProbability[i] += tempData;

}

}

cout << "TIP : Probability data is ";

for (int i = 0; i < (int) voiceLib.size(); ++i) {

cout << libProbability[i] << "\t";

}

cout << endl;

int countMax = 0;

for (int i = 1; i < (int) voiceLib.size(); ++i) {

if (libProbability[i] > libProbability[countMax]) {

countMax = i;

}

}

return countMax;

}

#include "stdafx.h"

#include "WaveRecorder.h"

// 静态变量初始化

array <char, CHUNCK\_SIZE> WaveRecorder::ChunkData = {};

vector<array<char, CHUNCK\_SIZE>> WaveRecorder::RawData;

UINT WaveRecorder::ChunksCount = 0;

WAVEHDR WaveRecorder::pwh[BUFFER\_LAYER] = {};

BOOL WaveRecorder::bCallback = false;

CNKDATAUpdateCallback WaveRecorder::callback = NULL;

bool WaveRecorder::stop = false;

bool WaveRecorder::dat\_ignore = false;

void WaveRecorder::set\_Callback(CNKDATAUpdateCallback fn)

{

bCallback = true;

callback = fn;

}

void WaveRecorder::set\_FileName(string Target)

{

// 若启用存档则保存一个wav文件

bSaveFile = true;

dest\_path = Target;

// 尝试打开文件【不存在则创建】

errno\_t err = fopen\_s(&fp, dest\_path.c\_str(), "wb");

if (err > 0) {

#if \_DEBUG

cout << "文件创建失败：" << err << " 检查文件名和占用" << endl;

#endif

bSaveFile = false;

}

}

void WaveRecorder::Start()

{

for (int layer = 0; layer < BUFFER\_LAYER; layer++) {

// 配置缓冲区

pwh[layer].lpData = new char[CHUNCK\_SIZE];

pwh[layer].dwBufferLength = CHUNCK\_SIZE;

pwh[layer].dwBytesRecorded = 0;

pwh[layer].dwUser = layer;

pwh[layer].dwFlags = 0;

pwh[layer].dwLoops = 0;

pwh[layer].lpNext = NULL;

pwh[layer].reserved = 0;

// 排进缓冲区

waveInPrepareHeader(hwi, &pwh[layer], sizeof(WAVEHDR));

waveInAddBuffer(hwi, &pwh[layer], sizeof(WAVEHDR));

}

// 初始化裸数据缓存

RawData.clear();

RawData.reserve(10);

// 发送录音开始消息

waveInStart(hwi);

}

void WaveRecorder::Stop()

{

// 停止标记

stop = true;

// 设备停止

waveInStop(hwi);

waveInReset(hwi);

// 释放缓冲区

for (int layer = 0; layer<BUFFER\_LAYER; layer++) {

waveInUnprepareHeader(hwi, &pwh[layer], sizeof(WAVEHDR));

delete pwh[layer].lpData;

}

// 保存Header+RawData

if(bSaveFile) {

WaveFileWrite();

}

}

void WaveRecorder::Reset()

{

// 静态变量初始化

RawData.clear();

ChunksCount = 0;

bSaveFile = false;

bCallback = false;

callback = NULL;

stop = false;

dat\_ignore = false;

}

void WaveRecorder::WaveInitFormat(LPWAVEFORMATEX WaveFormat, WORD Ch, DWORD SampleRate, WORD BitsPerSample)

{

// 自动配置参数

WaveFormat->wFormatTag = WAVE\_FORMAT\_PCM;

WaveFormat->nChannels = Ch;

WaveFormat->nSamplesPerSec = SampleRate;

WaveFormat->nAvgBytesPerSec = SampleRate \* Ch \* BitsPerSample / 8;

WaveFormat->nBlockAlign = Ch \* BitsPerSample / 8;

WaveFormat->wBitsPerSample = BitsPerSample;

WaveFormat->cbSize = 0;

#if \_DEBUG

cout << " 采样参数：" << endl;

cout << " 声道数" << Ch << endl;

cout << " 每秒采样率" << SampleRate << "Hz" << endl;

cout << " 位深" << BitsPerSample << endl;

#endif

}

void WaveRecorder::WaveFileWrite()

{

// 编辑并写入Wave头信息

WavHeader.data\_size = CHUNCK\_SIZE\*RawData.size();

WavHeader.size\_8 = WavHeader.data\_size + 32;

fwrite(&WavHeader, sizeof(WavHeader), 1, fp);

// 追加RawData

fwrite(RawData.data(), CHUNCK\_SIZE\*RawData.size(), 1, fp);

// 写入结束

fclose(fp);

}

DWORD WaveRecorder::WaveXAPI\_Callback(HWAVEIN hwavein, UINT uMsg, DWORD dwInstance, DWORD dwParam1, DWORD dwParam2)

{

// 消息switch

switch (uMsg)

{

case WIM\_OPEN: // 设备成功已打开

ChunksCount = 0; // 这次"session"的数据块计数

break;

case WIM\_DATA: // 缓冲区数据填充完毕

// 停止后会频繁发出WIM\_DATA,已经将数据转移所以不必理会后继数据【后继数据在这里来看是是重复的】

if (!dat\_ignore) {

// 把缓冲区数据拷贝出来

memcpy(ChunkData.data(), ((LPWAVEHDR)dwParam1)->lpData, CHUNCK\_SIZE);

// 没有录进去的被填充为0xcd,改成0来避免末尾出现爆音【只在结束录音时进行，不影响添加缓存效率】

if (((LPWAVEHDR)dwParam1)->dwBytesRecorded < CHUNCK\_SIZE) {

for (size\_t i = ((LPWAVEHDR)dwParam1)->dwBytesRecorded; i < CHUNCK\_SIZE; i++) {

ChunkData.at(i) = 0;

}

}

// 添加这一帧

RawData.push\_back(ChunkData);

// 如果你设置了回调函数

if (bCallback) {

callback(ChunkData, ChunksCount, stop);

}

}

ChunksCount += 1;

// 如果需要停止录音则不继续添加缓存

if (!stop) {

waveInAddBuffer(hwavein, (LPWAVEHDR)dwParam1, sizeof(WAVEHDR)); //添加到缓冲区

}

else { // 防止重复记录数据

dat\_ignore = true;

}

break;

case WIM\_CLOSE:

break;

default:

break;

}

return 0;

}

WaveRecorder::WaveRecorder()

{

this->fp = NULL;

/\*

this->WavHeader = {

{ 'R', 'I', 'F', 'F' },

0,

{ 'W', 'A', 'V', 'E' },

{ 'f', 'm', 't', ' ' },

sizeof(PCMWAVEFORMAT) ,

WAVE\_FORMAT\_PCM,

1,

SAMPLE\_RATE,

SAMPLE\_RATE\*(SAMPLE\_BITS / 8),

SAMPLE\_BITS / 8,

SAMPLE\_BITS,

{ 'd', 'a', 't', 'a' },

0

};

\*/

this->WavHeader.riff[0] = 'R';

this->WavHeader.riff[1] = 'I';

this->WavHeader.riff[2] = 'F';

this->WavHeader.riff[3] = 'F';

this->WavHeader.size\_8 = 0;

this->WavHeader.wave[0] = 'W';

this->WavHeader.wave[1] = 'A';

this->WavHeader.wave[2] = 'V';

this->WavHeader.wave[3] = 'E';

this->WavHeader.fmt[0] = 'f';

this->WavHeader.fmt[1] = 'm';

this->WavHeader.fmt[2] = 't';

this->WavHeader.fmt[3] = ' ';

this->WavHeader.fmt\_size = sizeof(PCMWAVEFORMAT);

this->WavHeader.format\_tag = WAVE\_FORMAT\_PCM;

this->WavHeader.channels = 1;

this->WavHeader.samples\_per\_sec = SAMPLE\_RATE;

this->WavHeader.avg\_bytes\_per\_sec = SAMPLE\_RATE\*(SAMPLE\_BITS / 8);

this->WavHeader.block\_align = SAMPLE\_BITS / 8;

this->WavHeader.bits\_per\_sample = SAMPLE\_BITS;

this->WavHeader.data[0] = 'd';

this->WavHeader.data[1] = 'a';

this->WavHeader.data[2] = 't';

this->WavHeader.data[3] = 'a';

this->WavHeader.data\_size = 0;

// 如果没有输入设备则析构

if (!waveInGetNumDevs()) {

#if \_DEBUG

cout << "Windows没有找到音频输入设备" << endl;

#endif

}

#if \_DEBUG

WAVEINCAPS WaveInCaps;

MMRESULT mmResult = waveInGetDevCaps(0, &WaveInCaps, sizeof(WAVEINCAPS));

cout << "默认设备描述：(" << WaveInCaps.szPname<<")" << endl;

#endif

WAVEFORMATEX pwfx;

WaveInitFormat(&pwfx, CHANNEL\_MUM, SAMPLE\_RATE, SAMPLE\_BITS);

waveInOpen(&hwi, WAVE\_MAPPER, &pwfx, (DWORD)WaveXAPI\_Callback, NULL, CALLBACK\_FUNCTION);

}

WaveRecorder::~WaveRecorder()

{

// 关闭设备并发出WIM\_CLOSE

waveInClose(hwi);

}