**JAVA AUDIO SYNTHESIS SYSTEM**

PROJECT REPORT

*Submitted by*

AAKANKSHA CHOUDHARY (Roll no. 49)

PRITAM HAZRA (Roll no. 51)

SUNDAL JAIN (Roll no. 55)

*To*

*The Mumbai University*

*in partial fulfilled of the requirements for Semester B.E.(Computer Engineering) course during*

*the academic year 2015-2016*

*Fourth Year in Computer Engineering*

**

DEPERTMENT OF COMPUTER ENGINEERING

VIDYAVARDHINI’S COLLEGE OF ENGINEERING AND TECHNOLOGY

K. T. MARG, VASAI ROAD (W.) DIST-THANE, PIN: 401202

(Affiliated to University of Mumbai)

2015-2016

**Abstract**

* The physical modeling of complex sound generators can only be approached by individually synthesizing and discretizing the objects that contribute to the generation of sounds. This raises the problem of how to correctly implement the interaction between these objects.
* In the past decade, the market has been preparing itself for the maturing of the technology related to the creation of interactive multimedia environments.
* In spite of the abundance of results in the many areas of interest for multimedia applications, little has been done for synergically exploiting those that concern synthetic and natural modeling/rendering of acoustic 3-D environments.
* This environment is based on a foundation structure consisting of a small number of Java interfaces abstract classes, and a potentially unlimited number of unit generators, which are created by extending the abstract classes and implementing a single method.
* Filter-graphs, sometimes called “patches”, are created by linking together unit generators in arbitrary complex graph structures. Patches can be rendered in real-time with special unit generators that communicate with the audio hardware, which we have implemented using the JavaSound API.

**Introduction**

Several software applications for digital audio synthesis are presently available. These applications have varying degrees of user extensibility and customizability. They also differ in price from free to very expensive, and may require specialized hardware or a specific operating system. The target application of these systems varies too, but all systems that we are aware of are primarily focused on the synthesis of music.

All the features we wanted for an audio synthesis environment for these applications could not be found in any single existing environment, and we therefore developed an environment specifically for these kind of sounds, which we have called “JASS” which stands for “Java Audio Synthesis System”.

Here we are using two types of audio data supported by JavaSound API.

1. Sampled Audio Data
2. Musical Instrument Digital Interface

Here we are using Synthetic Sounds using Java. What actually is a Synthetic sound? Synthetic sounds, *(as opposed to sounds that you record via a microphone),* are sounds that you create by executing a mathematical algorithm. Here we are using Sine wave and converting that sine wave into audio.

**Literature Review**

Use of such synthetically generated Sounds:

* Mobile Platforms
* To reach the pitch n frequency that humans need log time to get trained and generate them
* To generate those sounds that are beyond human’s caliber
* To generate stable, repetitive tones continuously
* Assistance in Electronic Music like Trance

**Existing Algorithm**

Java classes preferred here are: AudioFormat, AudioInputStream and SourceDataLine.

Inputs are Frequencies, Sample Rate (The higher the sampling rate, the more samples are required for a fixed amount of time, the more memory is required, and the more computational demands are placed on the computer to be able to handle the audio data in real time) and Number of channels (Mono and Stereo). Java SDK 1.4.1 allows both monaural *(one channel)* and stereo *(two channel)* sound.

Variables used in this project are:

* sampleRate: (Every signal is represented in form of numbers i.e. the amplitude of the signal at different, uniform instances of time known as samples, the rate is known as sampleRate) Allowable 8000,11025,16000,22050,44100
* sampleSizeInBits: (PCM encoding is used wherein the magnitude of the analog signal is sampled at regular to obtain samples)Allowable 8,16
* channels: Java allows 1 or 2
* signed: JAVA allows only positive values of frequencies.

In this project, there are 5 types of sound synthesized by changing the frequencies of sine wave. They are

* Tones
* Stereo Panning
* Stereo Ping Pong
* FM Sweep
* Decay Pulse

**Implementation:**

import javax.swing.\*;

import java.awt.\*;

import java.awt.event.\*;

import javax.sound.sampled.\*;

import java.io.\*;

import java.nio.channels.\*;

import java.nio.\*;

import java.util.\*;

public class AudioSynth01 extends JFrame{

AudioFormat audioFormat;

AudioInputStream audioInputStream;

SourceDataLine sourceDataLine;

float sampleRate = 16000.0F;

int sampleSizeInBits = 16;

int channels = 1;

boolean signed = true;

boolean bigEndian = true;

byte audioData[] = new byte[16000\*4];

final JButton generateBtn = new JButton("Generate");

final JButton playOrFileBtn = new JButton("Play/File");

final JLabel elapsedTimeMeter =new JLabel("0000");

final JRadioButton tones = new JRadioButton("Tones",true);

final JRadioButton stereoPanning = new JRadioButton("Stereo Panning");

final JRadioButton stereoPingpong = new JRadioButton("Stereo Pingpong");

final JRadioButton fmSweep = new JRadioButton("FM Sweep");

final JRadioButton decayPulse =new JRadioButton("Decay Pulse");

final JRadioButton echoPulse = new JRadioButton("Echo Pulse");

final JRadioButton waWaPulse = new JRadioButton("WaWa Pulse");

final JRadioButton listen = new JRadioButton("Listen",true);

final JRadioButton file = new JRadioButton("File");

final JTextField fileName = new JTextField("junk",10);

public static void main(String args[]){

new AudioSynth01();

} //end main

public AudioSynth01{

final JPanel controlButtonPanel =new JPanel();

controlButtonPanel.setBorder(BorderFactory.createEtchedBorder());

final JPanel synButtonPanel = new JPanel();

final ButtonGroup synButtonGroup = new ButtonGroup();

final JPanel centerPanel = new JPanel();

final JPanel outputButtonPanel = new JPanel();

outputButtonPanel.setBorder(BorderFactory.createEtchedBorder());

final ButtonGroup outputButtonGroup = new ButtonGroup();

playOrFileBtn.setEnabled(false);

generateBtn.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent e){

playOrFileBtn.setEnabled(false);

new SynGen().getSyntheticData(audioData);

playOrFileBtn.setEnabled(true);

} //end actionPerformed

} //end ActionListener

); //end addActionListener()

playOrFileBtn.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent e){

playOrFileData();

} //end actionPerformed

} //end ActionListener

); //end addActionListener()

controlButtonPanel.add(generateBtn);

controlButtonPanel.add(playOrFileBtn);

controlButtonPanel.add(elapsedTimeMeter);

synButtonGroup.add(tones);

synButtonGroup.add(stereoPanning);

synButtonGroup.add(stereoPingpong);

synButtonGroup.add(fmSweep);

synButtonGroup.add(decayPulse);

synButtonGroup.add(echoPulse);

synButtonGroup.add(waWaPulse);

synButtonPanel.setLayout(new GridLayout(0,1));

synButtonPanel.add(tones);

synButtonPanel.add(stereoPanning);

synButtonPanel.add(stereoPingpong);

synButtonPanel.add(fmSweep);

synButtonPanel.add(decayPulse);

synButtonPanel.add(echoPulse);

synButtonPanel.add(waWaPulse);

centerPanel.add(synButtonPanel);

outputButtonGroup.add(listen);

outputButtonGroup.add(file);

outputButtonPanel.add(listen);

outputButtonPanel.add(file);

outputButtonPanel.add(fileName);

getContentPane().add(controlButtonPanel,BorderLayout.NORTH);

getContentPane().add(centerPanel, BorderLayout.CENTER);

getContentPane().add(outputButtonPanel, BorderLayout.SOUTH);

setTitle("Synthetic Sound Generator");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

setSize(250,275);

setVisible(true);

} //end constructor

private void playOrFileData() {

try{

InputStream byteArrayInputStream =new ByteArrayInputStream(audioData);

audioFormat = new AudioFormat(sampleRate, sampleSizeInBits, channels, signed, bigEndian);

audioInputStream = new AudioInputStream(byteArrayInputStream, audioFormat, audioData.length/audioFormat. getFrameSize());

DataLine.Info dataLineInfo =new DataLine.Info(SourceDataLine.class, audioFormat);

sourceDataLine = (SourceDataLine) AudioSystem.getLine(dataLineInfo);

if(listen.isSelected()){

new ListenThread().start();

}else{

generateBtn.setEnabled(false);

playOrFileBtn.setEnabled(false);

try{

AudioSystem.write(audioInputStream, AudioFileFormat.Type.AU, new File(fileName.getText() +".au"));

}catch (Exception e) {

e.printStackTrace();

System.exit(0);

} //end catch

generateBtn.setEnabled(true);

playOrFileBtn.setEnabled(true);

} //end else

}catch (Exception e) {

e.printStackTrace();

System.exit(0);

} //end catch

} //end playOrFileData

class ListenThread extends Thread{

byte playBuffer[] = new byte[16384];

public void run(){

try{

generateBtn.setEnabled(false);

playOrFileBtn.setEnabled(false);

sourceDataLine.open(audioFormat);

sourceDataLine.start();

int cnt;

long startTime = new Date().getTime();

while((cnt = audioInputStream.read(playBuffer, 0, playBuffer.length))!= -1){

if(cnt > 0){

sourceDataLine.write(playBuffer, 0, cnt);

} //end if

} //end while

sourceDataLine.drain();

int elapsedTime = (int)(new Date().getTime() - startTime);

elapsedTimeMeter.setText("" + elapsedTime);

sourceDataLine.stop();

sourceDataLine.close();

generateBtn.setEnabled(true);

playOrFileBtn.setEnabled(true);

}catch (Exception e) {

e.printStackTrace();

System.exit(0);

} //end catch

} //end run

} //end inner class ListenThread

class SynGen{

ByteBuffer byteBuffer;

ShortBuffer shortBuffer;

int byteLength;

void getSyntheticData(byte[] synDataBuffer){

byteBuffer = ByteBuffer.wrap(synDataBuffer);

shortBuffer = byteBuffer.asShortBuffer();

byteLength = synDataBuffer.length;

if(tones.isSelected()) tones();

if(stereoPanning.isSelected()) stereoPanning();

if(stereoPingpong.isSelected()) stereoPingpong();

if(fmSweep.isSelected()) fmSweep();

if(decayPulse.isSelected()) decayPulse();

if(echoPulse.isSelected()) echoPulse();

if(waWaPulse.isSelected()) waWaPulse();

} //end getSyntheticData method

void tones(){

channels = 1; //Java allows 1 or 2

int bytesPerSamp = 2;

sampleRate = 16000.0F;

int sampLength = byteLength/bytesPerSamp;

for(int cnt = 0; cnt < sampLength; cnt++){

double time = cnt/sampleRate;

double freq = 950.0;//arbitrary frequency

double sinValue = (Math.sin(2\*Math.PI\*freq\*time) +Math.sin(2\*Math.PI\*(freq/1.8)\*time) +Math.sin(2\*Math.PI\*(freq/1.5)\*time))/3.0;

shortBuffer.put((short)(16000\*sinValue));

} //end for loop

} //end method tones

void stereoPanning(){

channels = 2; //Java allows 1 or 2

int bytesPerSamp = 4; //Based on channels

sampleRate = 16000.0F;

int sampLength = byteLength/bytesPerSamp;

for(int cnt = 0; cnt < sampLength; cnt++){

double rightGain = 16000.0\*cnt/sampLength;

double leftGain = 16000.0 - rightGain;

double time = cnt/sampleRate;

double freq = 600; //An arbitrary frequency

double sinValue =Math.sin(2\*Math.PI\*(freq)\*time);

shortBuffer.put( (short)(leftGain\*sinValue));

sinValue =Math.sin(2\*Math.PI\*(freq\*0.8)\*time);

shortBuffer.put( (short)(rightGain\*sinValue));

} //end for loop

} //end method stereoPanning

void stereoPingpong(){

channels = 2; //Java allows 1 or 2

int bytesPerSamp = 4; //Based on channels

sampleRate = 16000.0F;

int sampLength = byteLength/bytesPerSamp;

double leftGain = 0.0;

double rightGain = 16000.0;

for(int cnt = 0; cnt < sampLength; cnt++){

if(cnt % (sampLength/8) == 0){

double temp = leftGain;

leftGain = rightGain;

rightGain = temp;

} //end if

double time = cnt/sampleRate;

double freq = 600; //An arbitrary frequency

double sinValue = Math.sin(2\*Math.PI\*(freq)\*time);

shortBuffer.put( (short)(leftGain\*sinValue));

sinValue =Math.sin(2\*Math.PI\*(freq\*0.8)\*time);

shortBuffer.put((short)(rightGain\*sinValue));

} //end for loop

} //end stereoPingpong method

void fmSweep(){

channels = 1; //Java allows 1 or 2

int bytesPerSamp = 2; //Based on channels

sampleRate = 16000.0F;

int sampLength = byteLength/bytesPerSamp;

double lowFreq = 100.0;

double highFreq = 1000.0;

for(int cnt = 0; cnt < sampLength; cnt++){

double time = cnt/sampleRate;

double freq = lowFreq +cnt\*(highFreq-lowFreq)/sampLength;

double sinValue =Math.sin(2\*Math.PI\*freq\*time);

shortBuffer.put((short)(16000\*sinValue));

} //end for loop

} //end method fmSweep

void decayPulse(){

channels = 1; //Java allows 1 or 2

int bytesPerSamp = 2; //Based on channels

sampleRate = 16000.0F;

int sampLength = byteLength/bytesPerSamp;

for(int cnt = 0; cnt < sampLength; cnt++){

double scale = 2\*cnt;

if(scale > sampLength) scale = sampLength;

double gain = 16000\*(sampLength-scale)/sampLength;

double time = cnt/sampleRate;

double freq = 499.0;//an arbitrary freq

double sinValue = (Math.sin(2\*Math.PI\*freq\*time) +

Math.sin(2\*Math.PI\*(freq/1.8)\*time) +

Math.sin(2\*Math.PI\*(freq/1.5)\*time))/3.0;

shortBuffer.put((short)(gain\*sinValue));

} //end for loop

} //end method decayPulse

void echoPulse(){

channels = 1; //Java allows 1 or 2

int bytesPerSamp = 2; //Based on channels

sampleRate = 16000.0F;

int sampLength = byteLength/bytesPerSamp;

int cnt2 = -8000;

int cnt3 = -16000;

int cnt4 = -24000;

for(int cnt1 = 0; cnt1 < sampLength; cnt1++,cnt2++,cnt3++,cnt4++){

double val = echoPulseHelper(cnt1,sampLength);

if(cnt2 > 0){

val += 0.7 \* echoPulseHelper(cnt2,sampLength);

}//end if

if(cnt3 > 0){

val += 0.49 \* echoPulseHelper(cnt3,sampLength);

}//end if

if(cnt4 > 0){

val += 0.34 \* echoPulseHelper(cnt4,sampLength);

} //end if

shortBuffer.put((short)val);

} //end for loop

} //end method echoPulse

double echoPulseHelper(int cnt,int sampLength){

double scale = 2\*cnt;

if(scale > sampLength) scale = sampLength;

double gain = 16000\*(sampLength-scale)/sampLength;

double time = cnt/sampleRate;

double freq = 499.0;//an arbitrary freq

double sinValue =Math.sin(2\*Math.PI\*freq\*time) +Math.sin(2\*Math.PI\*(freq/1.8)\*time) +Math.sin(2\*Math.PI\*(freq/1.5)\*time))/3.0;

return(short)(gain\*sinValue);

} //end echoPulseHelper

void waWaPulse(){

channels = 1; //Java allows 1 or 2

int bytesPerSamp = 2; //Based on channels

sampleRate = 16000.0F;

int sampLength = byteLength/bytesPerSamp;

int cnt2 = -8000;

int cnt3 = -16000;

int cnt4 = -24000;

for(int cnt1 = 0; cnt1 < sampLength; cnt1++,cnt2++,cnt3++,cnt4++){

double val = waWaPulseHelper(cnt1,sampLength);

if(cnt2 > 0){

val += -0.7 \* waWaPulseHelper(cnt2,sampLength);

} //end if

if(cnt3 > 0){

val += 0.49 \* waWaPulseHelper(cnt3,sampLength);

} //end if

if(cnt4 > 0){

val += -0.34 \* waWaPulseHelper(cnt4,sampLength);

} //end if

shortBuffer.put((short)val);

} //end for loop

} //end method waWaPulse

double waWaPulseHelper(int cnt,int sampLength){

double scale = 2\*cnt;

if(scale > sampLength) scale = sampLength;

double gain = 16000\*(sampLength-scale)/sampLength;

double time = cnt/sampleRate;

double freq = 499.0;//an arbitrary freq

double sinValue = (Math.sin(2\*Math.PI\*freq\*time) + Math.sin(2\*Math.PI\*(freq/1.8)\*time)

+Math.sin(2\*Math.PI\*(freq/1.5)\*time))/3.0;

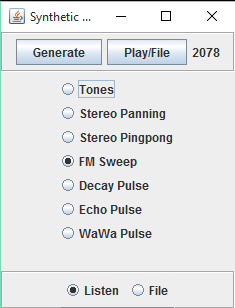
return(short)(gain\*sinValue);

} //end waWaPulseHelper

} //end SynGen class

} //end outer class AudioSynth01.java

**Result**



In this User Interface we have given 7 choices of various sounds. This is generated by changing the frequencies of the sine wave or by merging two or more sine waves.

User selects whichever sound he wants and press Generate button. After the Generate button is pressed that sound file is generated and accordingly user can play that sound file.

**Conclusion**

* The sound synthesized can be used in a variety of fields, would lessen the human efforts in producing a variety of pitches and would be used, widely, in electronic music.
* Thus the synthesized would have a bright scope of progress in the near future.
* The current distribution of the Java Audio system contains a set of unit generators for reading and playing audio files, at varying speeds and volumes, mixers and using various filters.
* For our applications we have found that using a buffer-size of 1 sample results in a slowdown by a factor 4, compared to using a buffer-size of 100. At a sampling rate of 44100Hz, a buffer-size of 100 translates in a latency of about 2 milliseconds, which is excellent for most purposes.
* Unfortunately current JavaSound implementations require large buffers for real-time synthesis on all platforms.
* The JASS system has been used primarily for our research in contact sound generation, but we hope that it will find more widespread usage.

**References**

* IEEE Paper: A JAVA AUDIO SYNTHESIS SYSTEM FOR PROGRAMMERS (JASS)
* [1] T. Kim and M. Lin, “Physically Based Animation and Sound Rendering”, Proceeding of Pacific Graphics, 2004
* K. van den Doel, *Sound Synthesis for Virtual Reality and Computer Games*, Ph.D. thesis, University of British Columbia, 1998.
* http://www.acoustics.salford.ac.uk/acoustics\_info/sound\_synthesis