5-2011

An expressive vocal MIDI controller

Brady Hurlburt
University of Arkansas, Fayetteville

AN EXPRESSIVE VOCAL MIDI CONTROLLER

AN EXPRESSIVE VOCAL MIDI CONTROLLER

A thesis submitted in partial
fulfillment of the requirements for the degree of
Bachelors of Science in Electrical Engineering

By

Brady L. Hurlburt

May 2011
University of Arkansas

ABSTRACT

The choice of controller often characterizes the performance obtained from an electronic instrument. Controllers exist that allow electronic musicians to mimic the expressive qualities of keyboard, wind, string, and percussion instruments, but vocalists have been largely neglected. To fill this need, a realtime software vocal MIDI controller named Grove is developed and demonstrated. Grove's pitch and time analyses are shown to be accurate, and it has mappable control signals that go beyond the capabilities of previous audio-to-MIDI converters to facilitate an expressive performance. Grove's source code is available under the GNU GPL v3 license.

This thesis is approved for recommendation to the Honors College.

Thesis Director:

Jinxian Wu, Ph.D.

THESIS DUPLICATION RELEASE

I hereby authorize the University of Arkansas Libraries to duplicate this thesis when needed for research and/or scholarship.

Agreed Brady L. Hurlburt

Refused _____

ACKNOWLEDGMENTS

I thank my Lord Jesus, who holds everything together. How do you make it all work?

I thank my brother Will Hurlburt for his advice and help.

I thank my parents Andy and Trudy Hurlburt for listening.

I thank Dr. Jingxian Wu for supervising this project.

I thank the WDL user community for providing assistance from all over the world.

I thank CCRMA at Stanford University for publishing things I want to know.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 Problem

The elements of electronic music that are effective are the ones that earn the listener's trust. They do so by mimicking natural, acoustic instruments. Electronic instruments, or synthesizers, have unlimited capabilities and can produce timbres and execute passages that are impossible for any natural instrument. However, if the listener is unable to reference these feats to some naturally occurring performance he understands, he perceives them as confusing rather than fantastic. Purely abstract electronic sounds have little power; the careful bending of musical laws can have a profound effect.

A great performance on an electronic instrument confines itself in most ways within the bounds of what is naturally feasible. By doing this, it earns the listener's belief and trust. With this reference framework established, the performer can then selectively introduce impossible feats and know that the listener truly appreciates them.

The easiest way for an electronic musician to mimic a natural instrument is to control the electronic instrument in the same way he would control a natural one. Musicians have invented an array of ways to play electronic instruments just like they would a natural instrument. These controllers take the form of keyboards, winds, plucked strings, bowed strings, and percussion devices, and they all output control signals that are sent to a synthesizer. The performance of the synthesizer tends to resemble a performance of the instrument that the controller mimics. Each controller brings with it a core set of expressive cababilities and limitations that make up that all-important reference to reality.

When one chooses a controller, he is choosing his limitations. A simple keyboard controller, for example, is unable to command the synthesizer to grow louder on a sustained note. This mimics the piano, whose hammer has no contact with the sound-producing string after the initial stike. A bowed string controller, on the other hand, has control over its sustained notes but imposes a limited polyphony. The character of the synthesizer's performance closely resembles the character of the controller, and the performer should choose a controller that has the needed expressive characteristics.

The performer's technical abilities also affect the choice of controller. The skills needed to play an electric drum pad are nearly identical to those needed to play a real snare drum. Likewise, a person who cannot play the piano will have no more luck attempting to use a keyboard synthesizer controller.

The wide variety of controllers available today serves most musicians well, both by providing the needed expressive structure and by catering to the performer's technical abilities. However, one type of musician is underserved – the vocalist. There is no commonly available means for a singer to create an electronic performance that mimics the unique expressive features of his singing.

1.2 Objective

The objective is to develop a software application which allows a singer to control a synthesizer expressively with his voice. This software, named Grove, will gather pitch, time and loudness information from the vocalist's performance. It will then send that control information to a software synthesizer to be played back with minimal latency.

This loudness control information will be user-mappable to the various parameters of the synthesizer.

The software should also integrate into the typical workflow of an electronic musician and have a straightforward user interface. The program will be free and available to the growing open-source audio community.

1.3 Approach

The design strategy for this software is to use proven open-source blocks and libraries for certain key operations in order to ensure excellent performance. Other operations are concise implementations of tested algorithms. The controlling state machine is novel and unique to this software. The software is written almost entirely in C++, with a few calls to C functions.

1.4 Potential Impact

This program gives vocal musicians a new creative tool and allows them to approach electronic music comfortably. It can be used to create moving performances that could not be created otherwise.

Because it is free to the audio community, this software has the potential to see widespread use. Also, because it is open-source, it has the ability to evolve to fit the needs of the audio community precisely.

1.5 Organization of this Thesis

The remainder of this thesis is organized as follows. *Section 2* compares a true vocal synthesizer controller to other alternatives and introduces some important terminology. *Section 3* describes the basic operation of the software and the platforms upon which it is built. *Section 4* details its C++ implementation. *Section 5* presents the results of testings, and *Section 6* concludes the thesis.

2. BACKGROUND

2.1 Current State of the Art

This section introduces three technologies that may appear to be suitable alternatives to Grove then explains their critical limitations.

2.1.1 Transcription Software

Software that detects pitch and time information in order to transcribe a performance is commonly available to vocalists. While this technology has a similar structure as the one presented here, it has a smaller goal and more limited feature set. Transcription software does not output control signals in a way that the user can easily map them to synthesizer parameters.

2.1.2 Vocoders

Vocoders are also often used musically to reflect characteristics of the voice onto an electronically generated tone. However, this reflection is much more literal and rigid than the one proposed by a vocal synthesizer controller. The purpose of the vocal controller is not to superimpose the voice over a signal, but rather to gather information about the performance for the musician to use as he sees fit.

2.1.3 Weight Sensitive Keys

Some modern keyboard controllers feature keys that are sentive to weight ever after the note’s attack and during its sustain. These allow for continuous control over the

note during a sustained note. This sort of control, however, is not a part of the typical skill set of a skilled keyboardist and is thus a much less natural way to control sustained growth or decay.

2.2 Collaborating Technologies

Grove interacts with several other types of audio technology, which are introduced in this section.

2.2.1 Synthesizers

Synthesizers consist of banks of audio-range oscillators, envelopes, low-frequency oscillators and filters. The base timbre is first created by combining one or more basic waveforms (often sinusoides, sawtooths, or square waves) created by the audio-range oscillators. That signal is passed to the envelope, which controls the attack, decay, sustain and release times of the tone. Synthesizers are often equipped with filters that allow the user to alter the overtone structure of the sound. Finally, low-frequency oscillators are available to vary many of the other parameters over time.

Synthesizers can exist as hardware or software. Because in this case the controller is software, software synthesizers are also used.

2.2.2 MIDI

The most common music control signal protocol, and the one used exclusively here, is the Musical Instrument Digital Interface (MIDI) protocol. At its core are *note-on*

and *note-off* messages, and it also features many dedicated parameter change messages and General Purpose Control Change (CC) messages.

2.2.3 DAW's and Plug-ins

In order to be integrated into a typical musician's workflow, this software runs as a plug-in inside of a digital audio workstation (DAW). A DAW is software that serves as a studio-in-a-box: it receives audio streams from the external interface through the operating system, records those streams to the harddrive, allows the user to arrange the recorded takes on multiple tracks, and plays back the tracks. Each track has a processing rack that enable the user to send the recorded audio tracks through third-party processing or analysis units. Those units are called plug-ins. Grove runs as a plug-in within a DAW, and it sends MIDI messages within the DAW to a software synthesizer that is also running as a plug-in.

3. APPROACH

3.1 High Level Design

Grove is designed to control a synthesizer to mimic a vocalist's performance in real time. It continuously sends *note-on* and *note-off* messages with corresponding pitches and loudness at attack (called velocity). It also sends that same loudness information out as a General Purpose CC MIDI message for the user to map to any synthesizer parameter.

Grove takes in samples at the rate specified by the DAW until it fills a buffer. The root-mean-square loudness of that window is calculated first. The buffer is then passed onto the Fast Fourier Transform (FFT) block, which returns the frequency spectrum magnitude and phase. The phase is discarded, but the magnitude is passed onto the pitch detection algorithm, which returns a MIDI note number. The loudness/note pair is then given to the MIDI output state machne, which decides what control information to send out. The machine then waits for the buffer to refill with new samples before starting the process again.

3.2 VST and Graphics Library

The plug-in is written entirely in C++ and C and is implemented in Steinberg, Inc.'s Virtual Studio Technology (VST) standard [1]. This standard for audio plugins is compatible with many professional DAW's, both those made by Steinberg and those made by other developers. Grove's VST programming interface is handled by an open-source library named IPlug, which is part of Cockos Inc.'s WDL library [2]. The project

compiles as an x86 Windows dynamic-linked library, and this single file contains everything needed to run the plug-in in a DAW.

WDL also assists in the creation of the graphical user interface (GUI) for the plugin. There is no graphical GUI creation tool, but rather it is designed entirely in code; however, WDL provides many useful objects, such as sliding faders and rotating knobs. Each graphic used in the GUI is used by permission.

3.3 Fast Fourier Transform

The FFT block included in WDL is based upon DJBFFT [3]. Its creator D. J. Berstein of the University of Illinois - Chicago contests that his FFT deserves the speed records claimed by the Massachusetts Institue of Technology's Fastest Fourier Transform in the West. DJBFFT is lightweight and is made up of only two files.

3.4 Pitch Detection

Grove's pitch detection uses the harmonic product spectrum (HPS) frequency-domain detection algorithm. Various detection algorithms and their appropriateness for realtime applications are discussed in a publication by the Center for Computer Research in Music and Acoustics at Stanford University [4]. A frequency-domain algorithm was chosen over time-domain method because the spectral information will likely be needed when additional types of harmonic analysis are added. The HPS specifically was chosen for its simplicity: a single vocalist is almost always a monophonic signal source, so the complexity of polyphonic detection is not needed.

HPS relies on the concept that musical tones are usually harmonic, having each overtone an integer multiple of the fundamental frequency. Therefore, if the frequency spectrum is downsampled by a factor of two, the second overtone of the downsampled buffer will align with the fundamental of the original buffer. Likewise, the third overtone of a buffer downsampled by a factor of three will align with fundamental in the original buffer. By multiplying the original buffer by its downsampled variants then adding the results, the fundamental frequency is made to stand out as a lone maximum. This process can be expressed as

$$\mathrm{Y}(\omega) = \prod_{r=1}^{R} Y(\omega r)$$

then

$$\max\big(T(\omega_i)\big)$$ [4].

4. IMPLEMENTATION

This section highlights the C++ implementation of five of Grove's important functions. *Figure 1* shows the Grove's graphical user interface during operation. All main source code is shown in *Appendices A* and *B* and is also available to download from http://code.google.com/p/grovecontroller/source/browse/ .

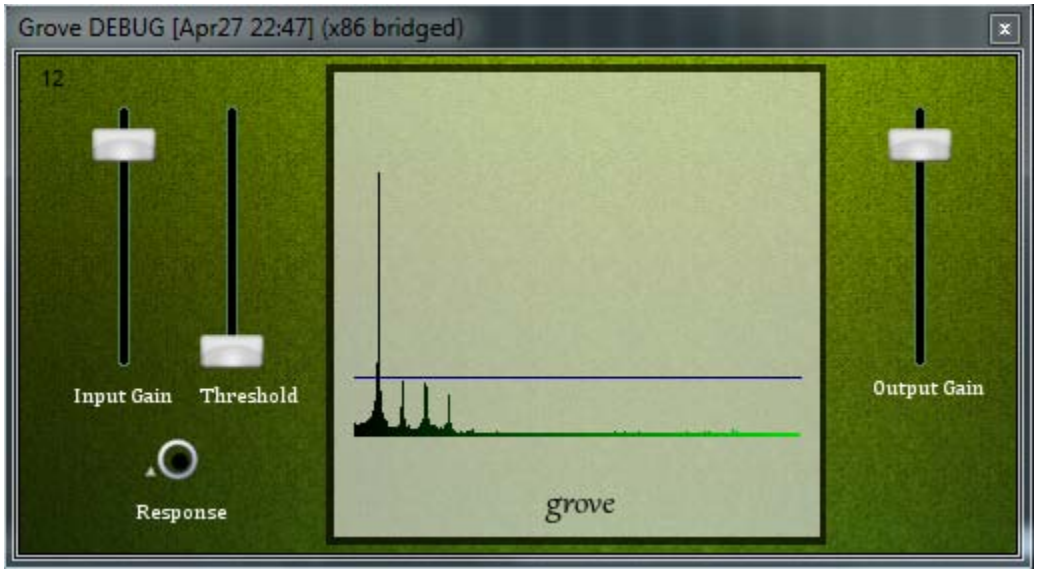

Figure 1: Graphical user interface

4.1 Pitch Detection

The following code implements the harmonic product spectrum pitch detection algorithm. This is demonstrated in [5].

```
int harmonics = 2;
int maxHIndex = FFTLENGTH/harmonics; //to keep in bounds
                //but this also sets an upper limit on detection
int minIndex = 0; //lowest possible place to look for a pitch
int maxLoc = minIndex;
for (int i=0; i<maxHIndex; i++){
    for (int j=1; j<=harmonics; j++){  //the fundamental is doubled
        spectrum[i] *= spectrum[i*j];
```

```
        }
        if (spectrum[i] > spectrum[maxLoc]){
            maxLoc = i;
        }
    }
```

The single line `spectrum[i] *= spectrum[i*j]` completes the resamples and multiplication. Comparing only two harmonics was experimentally determined to be the best choice for this FFT length and sample rate.

Another check adds robustness to the HPS algorithm. The most common error for this method is that it reports one octave higher than the actual pitch. To combat this, the following code checks the octave below the pitch reported by the code above. If the magnitude at that lower pitch is at least two tenths the magnitude of the original prediction, that pitch is mostly likely the actual fundamental.

```
    //double frequency protection
    //first search for max2 in the range below the prev'ly detected one
    int max2 = minIndex;
    int maxSearch = maxLoc * 3/4;
    for (int i=minIndex+1; i<maxSearch; i++){
        if (spectrum[i] > spectrum[max2]){
            max2 = i;
        }
    }
   if (abs(max2 * 2 - maxLoc) < 4) {
   //if it's within 4 of being half of the prev'ly detected
       if(spectrum[maxLoc] != 0){
            if (spectrum[max2]/spectrum[maxLoc] > 0.2) {
   //if it's at least so big compared the prev'ly detected
             maxLoc = max2;
            }
       }
   }
```

Once the maximum FFT bin is determined, that value is used to find the corresponding frequency in Hertz, then the matching MIDI pitch number. The derivation of the latter conversion is shown in [6]. Both conversions are handled by the following

code. The sample rate used here is not hardcoded but rather requested from the host DAW.

```
double freq = maxLoc*sampleRate/FFTLENGTH;
int midiNote = floor(17.3123*log(freq/440)+69 + 0.5);
//calculate MIDI note from fund freq and round.
return midiNote;
```

4.2 MIDI Output

MIDI output occurs once every time the FFT buffer fills. A simplified flowchart for deciding what to output is shown in *Figure 2*. This design checks only if the input window's loudness is above a certain threshold and whether the pitch has changed. If the pitch detection function reports a new pitch, the old note is turned off and a new note is turned on.

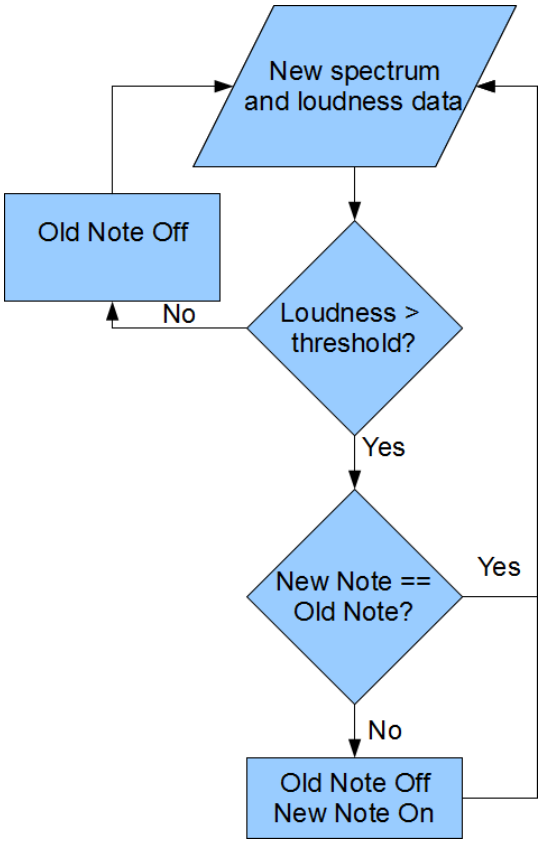

Figure 2: Simplified flowchart

The actual decision making process used in Grove adds three things to the flowchart above. First, it disallows octave jumps. These are a common error in HPS detection and are an uncommon occurance in a vocal line. Next, it requires a certain number of new notes in a row in order to change the pitch. This reduces jitter in the detection. This number can be adjusted by the user as the plug-in is running. Lastly, it sends the loudness data via a General Gurpose CC MIDI command every cycle, regardless of whether the note changed or not. This more practical flowchart is shown in *Figure 3.*

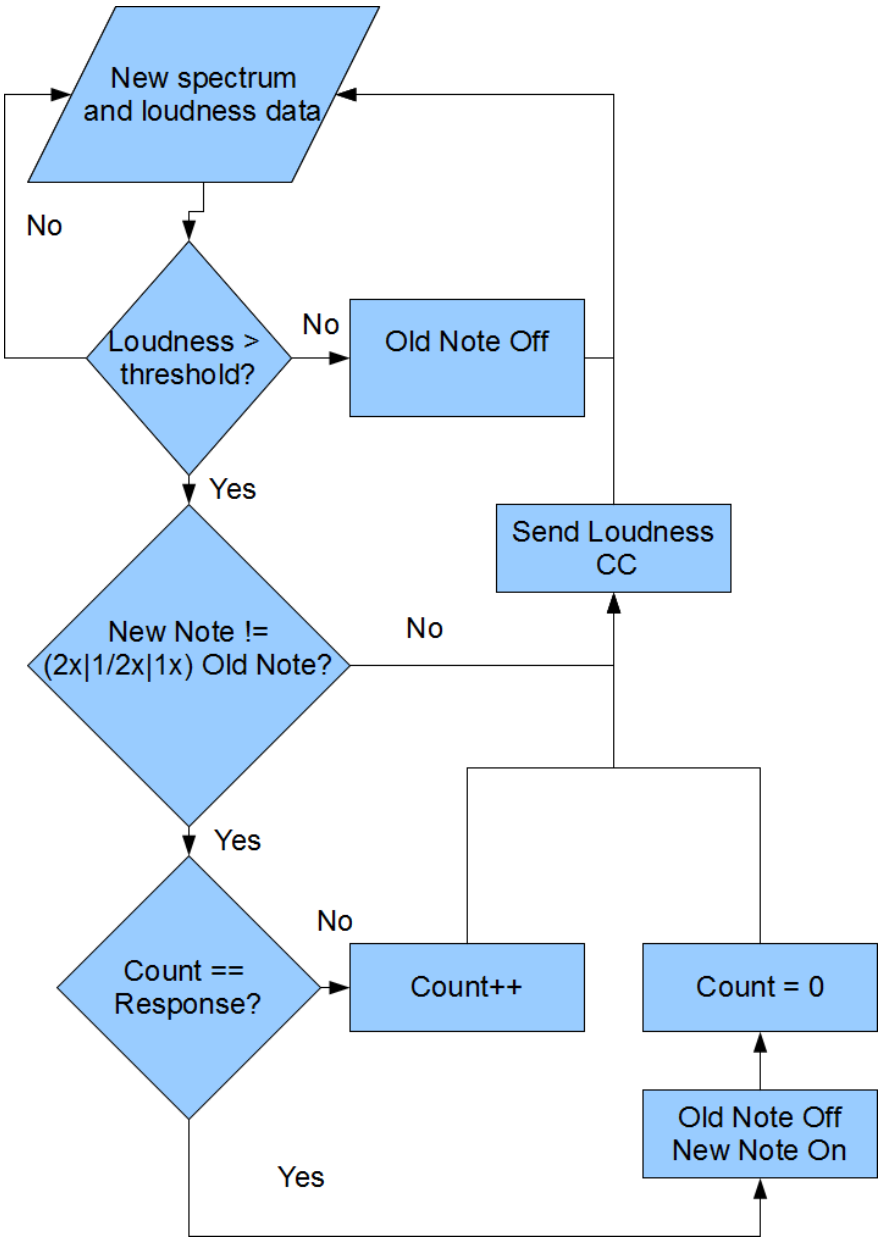

Figure 3: Practical implementation of MIDI output control

4.3 Loudness Signal

The loudness control change command is one of the most important distinguishing features of Grove. It serves as a model for many other possible control signals that could

be generated from other type of analyses. It is send out on MIDI's General Purpose Continuous Control (CC) #1 [7]. Any parameter on the receiveing synthesizer can be set to vary along with (or oppositite of) CC#1. The feature enables many creative configurations.

4.4 User Controls

The GUI gives the user control over four parameters. The user may set the input audio gain, which controls the magnitude of the signal fed into the FFT. He may also control the threshold value as well as the response variable referred to in *Figures 2* and *3*. Finally, he may set the MIDI output gain. These controls are important because the input signal could come in at any strength, and the pitch detection performs best over a narrow range of them.

4.5 Spectrum and Loudness Graphs

The most prominent feature of the user interface is the large graph near the center (refer to *Figure 1*). This graph updates in real time and displays two pieces of information – the frequency spectrum and the loudness of the window. The frequency spectrum begins black in color at the left and becomes green as it moves right into the higher pitches. The loudness is displayed as a horizontal bar that becomes bluer as it rises higher. This was implemented using the following code.

```
pGraphics->DrawLine(new Color(0,0,i,0),kGraph_X+10+i,kGraph_Y+kGraph_H-
50,kGraph_X+10+i,kGraph_Y+kGraph_H-50-mag);

pGraphics->DrawLine(new Color(0,0,0,*loudLine*500),kGraph_X+10
,kGraph_Y+kGraph_H-50-*loudLine*100,kGraph_X+kGraph_W-
10,kGraph_Y+kGraph_H-50-*loudLine*100);
```

5. RESULTS

5.1 Testing Environment

Because of its advanced MIDI routing control and affordable pricing, Reaper by Cockos, Inc. [8] was chosen as the DAW to host Grove and the synthesizer for testing. Triangle II by Cakewalk [9] was chosen as the software synthesizer because its *MIDI Learn* capabilities allow every parameter to be assigned to a MIDI CC. Triangle II is offered as a free download. *Figure 4* shows the complete testing environment.

Grove is the small, predominantly green and grey window in the center left; Triangle II is the larger window to its right. The green bar graph above them is the recored loudness data from the General Purpose CC #1. The green rectancles at the top of the image are recorded pitch and time data.

In this arrangement, although the entire system works in realtime with low latency, both the audio and MIDI stages are recorded and can be played back. Also, because the recorded MIDI data is independent of the synthesizer, the configuration of the synthesizer can be changed to perform the same recorded vocal track in a different way.

Figure 4: Grove and Triangle II running inside of Reaper

5.2 Testing Results

Figure 5 shows the analysis of a vocalist singing an F-Major scale.

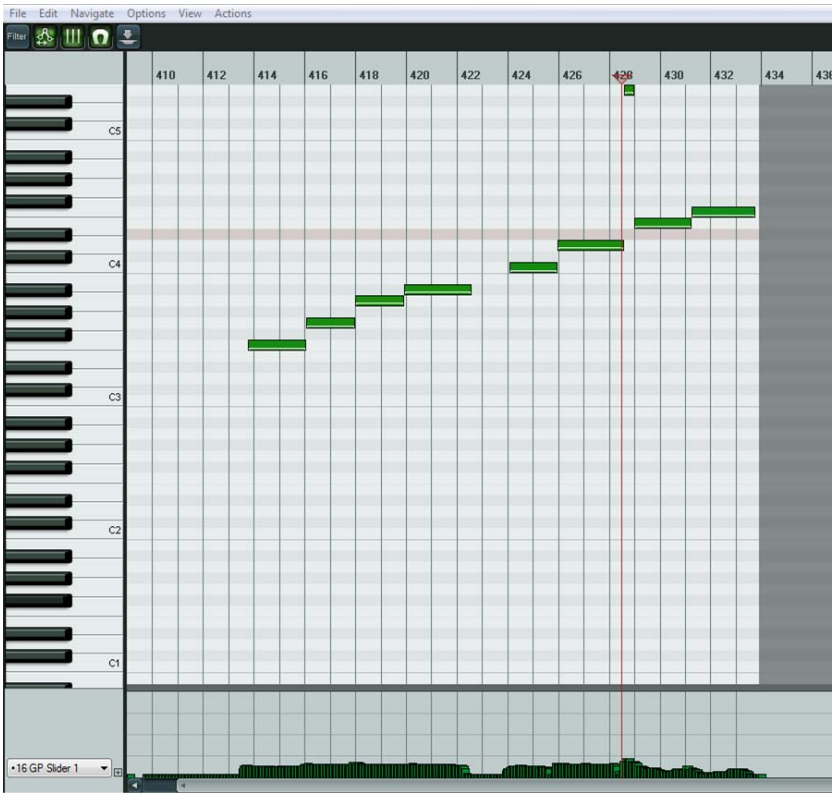

Figure 5: Analysis of a major scale

Here both the terminate-note and change-note branches of the main state machine are seen to be functional. It also shows that the loudness graph continues to change even on a held note, which is characteristic of a singer and an important goal of this project.

5.3 Musical Example

Grove was used in the context of a song to record a short sample that illustrates the importance of each of the project's goals. This audio sample can be downloaded from http://code.google.com/p/grovecontroller/downloads/list . The first audio file plays the

song with the vocal track that was used to control the synthesizer. The second audio file plays the same clip with the vocal track replaced by the synthesizer output.

The musical example features a setup that may be typical for Grove – the loudness signal is tied to several synthesizer parameters. First, it is linked with the master volume, allowing smooth growth and decay over a sustained note. Secondly, it is linked to the cutoff frequency of a low-pass filter. This causes the timbre of the output to become brighter as the note grows louder. Lastly, the loudness signal is tied to the gain of a white noise generator, which causes the synthesizer output to grow harsher as the note grows louder. The result is a long, wind blown tone. This setup demonstrates one of thousands of possible parameter mappings.

5.4 Publishing

The source code and compiled binary are published under the GNU GPL v3 license on the host Google Code at the address http://code.google.com/p/grovecontroller/. This will allow for open dialog about uses of the software as well as quick feedback regarding bugs. Its integrated versioning system allows for continued development in an organized fashion, and the wiki pages can contain detailed instructions for operation. *Figure 6* shows the Grove's Google Code homepage.

Figure 6: Home page for project

6. CONCLUSIONS

6.1 Summary

This thesis outlined the development of a VST MIDI controller designed exclusively for vocalists, who were previously underserved by controller developers. It improves vastly upon narrow-scoped transcription programs and rigid vocoders. Because Grove's source and binary are available as free downloads, its effectiveness and popularity have the potential to grow rapidly as it fills a need in the growing open-source audio software community.

6.2 Future Work

Grove can continue to grow as it adds more types of analysis. Loudness is currently the only parameter being sent out as a mappable control value, but many more are readily obtainable. Brightness, vowel, portamento, and vibrato are examples of continuous and discrete characteristics of the vocalist's input that could be analyzed, quantified and sent out as MIDI control signals. This would further increase the mapping possibilities at the synthesizer.

REFERENCES

[1] Steinberg GmbH. Our technologies. 2011(04/22) Retrieved from http://www.steinberg.net/en/company/technologies.html

[2] Schwartz, J. WDL (whittle): [re]usable C++, modestly. 2011(04/22). Retrieved from http://www.cockos.com/wdl/

[3] Bernstein, D. J. DJBFFT. 2011(4/22). Retrieved from http://cr.yp.to/djbfft.html

[4] Cuadra, P. d. l., Master, A., & Sapp, C. (2001). Efficient pitch detection techniques for interactive music. In Proceedings of the 2001 International Computer Music Conference, La Habana, 403.

[5] Cuadra, P. d. l., & Sapp, C. (2000). Harmonic product spectrum. Retrieved from http://sig.sapp.org/src/sigAudio/Pitch_HPS.cpp

[6] Ordiales, H. (2007). Fundamental (in Hz) to a MIDI note. Audio Research Blog, Retrieved from http://audiores.uint8.com.ar/blog/2007/08/26/fundamental-in-hz-to-a-midi-note/

[7] Cockos Incorporated. (2011). WDL user forum. Retrieved from http://forum.cockos.com/ .

[8] Cockos incorporated. Reaper 2011(04/22). Retrieved from http://www.reaper.fm/

[9] KVR: Audio Plugin News. Triangle II by cakewalk. 2011(4/22). Retrieved from http://www.kvraudio.com/get/203.html

A. SOURCE CODE - MAIN

```
#include "Grove.h"
#include "../IPlug_include_in_plug_src.h"
#include "../IControl.h"
#include "resource.h"
#include <math.h>
#include "fft.h"

enum EParams {
    kIGain,
    kOGain,
    kThresh,
    kResp,
    kGraphDummy,
    kNumParams
};
const int kNumPrograms = 1;

enum ELayout {
    //size of entire GUI
    kW = 500,
    kH = 250,

    //debugging text
    kText_L = 5,
    kText_R = 30,
    kText_T = 5,
    kText_B = 30,

    //length for all faders
    kFader_Len = 140,

    //IGainFader
    kIGain_X = 37,
    kIGain_Y = 19,

    //Threshold Fader
    kThresh_X = 91,
    kThresh_Y = 19,

    //OGain Fader
    kOGain_X = 435,
    kOGain_Y = 19,

    //Response Knob
    kResp_X = 56,
    kResp_Y = 178,

    //Spectrum Graph
    kGraph_X = 158,
    kGraph_Y = 9,
```

```
    kGraph_W = 243,
    kGraph_H = 231
};

IDistributionGraph::IDistributionGraph(IPlugBase *pPlug, int x, int y,
int width, int height, int paramIdx)
: IControl(pPlug, &IRECT(x,y,x+width,y+height ))
{
    OutputDebugString("dist graph ctor");
    m_pPlug=pPlug;
}

bool IDistributionGraph::Draw(IGraphics* pGraphics){
    OutputDebugString("dist graph draw()");
    double mag = 0;
    for (int i=0; i<kGraph_W-20; i++){
        if (FFTmag[i] > kGraph_H-100){
            mag = kGraph_H-100;
        }else{
            mag = FFTmag[i];
        }
        pGraphics->DrawLine(new
IColor(0,0,i,0),kGraph_X+10+i,kGraph_Y+kGraph_H-
50,kGraph_X+10+i,kGraph_Y+kGraph_H-50-mag);
        pGraphics->DrawLine(new
IColor(0,0,0,*loudLine*500),kGraph_X+10,kGraph_Y+kGraph_H-50-
*loudLine*100,kGraph_X+kGraph_W-10,kGraph_Y+kGraph_H-50-*loudLine*100);
    }
    return true;
}

void IDistributionGraph::SetGraphVals(double* mag, double* loud){
    FFTmag = mag;
    loudLine = loud;
    return;
}


int Grove::PitchDetect(){

    //HPS pitch detection
http://sig.sapp.org/src/sigAudio/Pitch_HPS.cpp
    int spectrum[FFTLENGTH] = {0};
    //make local copy of magFFT to mess with
    for (int i=0; i<FFTLENGTH; i++){
        spectrum[i] = magFFT[i];
    }

    int harmonics = 2;
    int maxHIndex = FFTLENGTH/harmonics; //to keep from going out of
bounds
                        //but this also sets an upper limit on detection
    int minIndex = 0; //lowest possible place to look for a pitch
    int maxLoc = minIndex;
```

```
    for (int i=0; i<maxHIndex; i++){
        for (int j=1; j<=harmonics; j++){  //the fundamental is doubled
            spectrum[i] *= spectrum[i*j];
        }
        if (spectrum[i] > spectrum[maxLoc]){
            maxLoc = i;
        }
    }


    //double frequency protection
    //first search for max2 in the range below the prev'ly detected one
    int max2 = minIndex;
    int maxSearch = maxLoc * 3/4;
    for (int i=minIndex+1; i<maxSearch; i++){
        if (spectrum[i] > spectrum[max2]){
            max2 = i;
        }
    }
   if (abs(max2 * 2 - maxLoc) < 4) {
      //if it's within 4 of being half of the prev'ly detected
       if(spectrum[maxLoc] != 0){
            if (spectrum[max2]/spectrum[maxLoc] > 0.2) {
//if it's at least so big compared the prev'ly detected (default 0.2)
             maxLoc = max2;
            }
       }
   }

    //use the following lines to display a value during debugging
    char text[50];
    sprintf(text,"%i",maxLoc);
    debugText->SetTextFromPlug(text);

    double freq = maxLoc*sampleRate/FFTLENGTH;
    int midiNote = floor(17.3123*log(freq/440)+69 + 0.5);
      //calculate MIDI note from fund freq and round.
      //http://audiores.uint8.com.ar/blog/2007/08/26/fundamental-in-
      //hz-to-a-midi-note

    return midiNote;
};

Grove::Grove(IPlugInstanceInfo instanceInfo)
:   IPLUG_CTOR(kNumParams, kNumPrograms, instanceInfo),
fftbuffersize(FFTLENGTH), count(0)
{
    TRACE;

    for (int i=0;i<FFTLENGTH;i++) magFFT[i]=0; //initialize buffer

    //load sample rate from host
    sampleRate = GetSampleRate();
```

```
    //initialize values for MIDI transmission
    oldNote = 0;
    newCount = 0;

    //***SET UP PARAMETERS***
    GetParam(kIGain)->InitDouble("Input Gain", 0.0, -70.0, 12.0, 0.1,
"dB");
    GetParam(kOGain)->InitDouble("Ouput Gain", 0.0, -70.0, 12.0, 0.1,
"dB");
    GetParam(kThresh)->InitDouble("Threshold", 0.01, 0.0, 1.0, 0.01,
"");
    GetParam(kResp)->InitInt("Response",8,0,20,"in a row");

    //***SET UP GUI***
    IGraphics* pGraphics = MakeGraphics(this,kW,kH);
    pGraphics->AttachBackground(BG_ID, BG_FN);
    //IGain fader graphic
    IBitmap bitmap = pGraphics->LoadIBitmap(FADER_ID, FADER_FN);
    pGraphics->AttachControl(new IFaderControl(this, kIGain_X,
kIGain_Y, kFader_Len, kIGain, &bitmap, kVertical));
    //Threshold fader graphic
    pGraphics->AttachControl(new IFaderControl(this, kThresh_X,
kThresh_Y, kFader_Len, kThresh, &bitmap, kVertical));
    //OGain fader graphic
    pGraphics->AttachControl(new IFaderControl(this, kOGain_X,
kOGain_Y, kFader_Len, kOGain, &bitmap, kVertical));
    //Respose knob graphic
    bitmap = pGraphics->LoadIBitmap(KNOB_ID, KNOB_FN);
    pGraphics->AttachControl(new IKnobRotaterControl(this, kResp_X,
kResp_Y, kResp, &bitmap));
    //Spectrum Graph
    graph = new IDistributionGraph(this, kGraph_X, kGraph_Y, kGraph_W,
kGraph_H, kGraphDummy);
    graph->SetGraphVals(magFFT,&loudness); //initialize pointer
    pGraphics->AttachControl(this->graph);
    //Pitch text
    debugText = new ITextControl(this,new
IRECT(kText_L,kText_T,kText_R,kText_B),new IText(&COLOR_BLACK));
    pGraphics->AttachControl(debugText);
    //Enable GUI
    AttachGraphics(pGraphics);

    WDL_fft_init();

    MakeDefaultPreset("-", kNumPrograms);
}


void Grove::ProcessDoubleReplacing(double** inputs, double** outputs,
int nFrames)
{
    double* in1 = inputs[0];
    double* in2 = inputs[1];
```

```
    double* out1 = outputs[0];
    double* out2 = outputs[1];

    //load parameters
    double iGain = GetParam(kIGain)->DBToAmp();
    double oGain = GetParam(kOGain)->DBToAmp();
    double thresh = GetParam(kThresh)->Value();
    int resp = GetParam(kResp)->Value();

    //use the following lines to display a value during debugging
    //char text[50];
    //sprintf(text,"%i",resp);
    //debugText->SetTextFromPlug(text);

    for (int s = 0; s < nFrames; ++s, ++in1, ++in2, ++out1, ++out2)
    {
        if(count == fftbuffersize-1)
        {

      //use FFT buffer to find loudness for simplicity's sake.
      //Must do it here while it's still in the time domain.
            loudness = 0;
            for (int i = 0; i < fftbuffersize; i++){    //RMS
                loudness += pow(fftbuffer[i].re,2);
            }
            loudness = (sqrt(loudness/fftbuffersize)*oGain)/0.4;

            WDL_fft(fftbuffer, fftbuffersize, false);

            for (int i = 0; i < fftbuffersize; i++)
            {
                int j = WDL_fft_permute(fftbuffersize, i);
                sortedbuffer[i].re = fftbuffer[j].re;
                sortedbuffer[i].im = fftbuffer[j].im;
            }

            //do processing on sortedbuffer here
            for (int i = 0; i < fftbuffersize; i++)
            {
                //cartesian to polar
                double mag = sqrt(sortedbuffer[i].re*sortedbuffer[i].re
+ sortedbuffer[i].im*sortedbuffer[i].im);
                double phase = atan2(sortedbuffer[i].im,
sortedbuffer[i].re);
                magFFT[i] = mag;
            }

            int newNote = PitchDetect();

            if (loudness > thresh){ //if loudness above threshold
                    if ((newNote != oldNote) && (newNote != oldNote+12)
&& (newNote != oldNote-12)){ //if new note that's not an octave jump
                            newCount++;
```

```
                            if (newCount == 8){
                                IMidiMsg* pMsg = new IMidiMsg;
                                pMsg->MakeNoteOffMsg(oldNote,0);
                                SendMidiMsg(pMsg);
                           pMsg->MakeNoteOnMsg(newNote,loudness*127,0);
                                SendMidiMsg(pMsg);
                                pMsg->MakeControlChangeMsg(pMsg-
>kGeneralPurposeController1,loudness);
                                SendMidiMsg(pMsg);
                                oldNote = newNote;
                                newCount = 0;
                            }else{
                                IMidiMsg* pMsg = new IMidiMsg;
                                pMsg->MakeControlChangeMsg(pMsg-
>kGeneralPurposeController1,loudness);
                                SendMidiMsg(pMsg);
                            }
                        }else{                    //same note or octave jump
                            IMidiMsg* pMsg = new IMidiMsg;
                            pMsg->MakeControlChangeMsg(pMsg-
>kGeneralPurposeController1,loudness);
                            SendMidiMsg(pMsg);
                        }
                    }else{                    //loudness not above threshold
                        IMidiMsg* pMsg = new IMidiMsg;
                        pMsg->MakeNoteOffMsg(oldNote,0);
                        SendMidiMsg(pMsg);
                    }
                count = 0;
            }else{
                count++;
            }

    fftbuffer[count].re = (WDL_FFT_REAL) *in1*iGain;
    fftbuffer[count].im = 0.;

    *out1 = *in1; //directly from input
    *out2 = *out1; //mono

    }
}
```

B. SOURCE CODE - DECLARATIONS

```
#ifndef __IPlugFFT__
#define __IPlugFFT__

#include "../IPlug_include_in_plug_hdr.h"
#include "fft.h"

#define FFTLENGTH 2048

class IDistributionGraph : public IControl{
public:
    IDistributionGraph(IPlugBase* pPlug, int x, int y, int width, int
height, int paramIdx);
    bool Draw(IGraphics* pGraphics);
    bool IsDirty() {return true;}
    void SetGraphVals(double* mag, double* loud);
private:
    IPlugBase *m_pPlug;
    double * FFTmag;
    double * loudLine;
};

class Grove : public IPlug
{
public:

    Grove(IPlugInstanceInfo instanceInfo);
    ~Grove() {}
    void ProcessDoubleReplacing(double** inputs, double ** outputs, int
nFrames);

private:
    int sampleRate;
    IDistributionGraph * graph;
    ITextControl * debugText;
    int fftbuffersize;
    int count, samps;
    int oldNote;
    int newCount;
    double loudness;
    WDL_FFT_COMPLEX fftbuffer[FFTLENGTH];
    WDL_FFT_COMPLEX sortedbuffer[FFTLENGTH];
    WDL_FFT_COMPLEX unsortedbuffer[FFTLENGTH];
    double magFFT[FFTLENGTH];
    int PitchDetect();
};



#endif
```