All together Now

ALL TOGETHER NOW
soloist, audience and electronics

for Michael Baldwin

A listening-movement concerto for any space exhibiting strong resonant frequencies. In preparation, a group of frequencies are selected that, when played from a sine wave oscillator from one loudspeaker in the space, exhibit a strong degree of dynamic instability when moving in any part of the space where the audience and/or soloist will be located. The work lasts any duration, but consideration is given to the number of frequencies selected for a performance, where more frequencies suggests a longer duration and less frequencies a shorter duration.

During the performance, each selected frequency is played by a sine wave oscillator from one loudspeaker in the same position as when selecting frequencies. An attack-sustain-release envelope is applied to each oscillator. All oscillators begin and end together. During the first half of the performance, each oscillator reaches its sustain segment at equally spaced points in time, one after another, from highest to lowest sounding frequency, and during the second half of the performance, each oscillators begins its decay segment at equally spaced points in time, one after another, from highest to lowest sounding frequency—revealing and concealing a gradient of resonant frequencies. Additionally, the peak amplitude for each oscillator is attenuated following an equal-loudness contour, the so-called Fletcher-Munson curves, resulting in relatively equally perceived loudness amongst oscillators sounding at different frequencies.

During the performance, the soloist turns and tilts their head and torso back and forth within yaw, pitch and roll axes of movement, where movements away from the body's central resting position are understood as positive and negative degrees of off-center movement. The changing positive and negative amplitude values of six sine functions, at different harmonic frequencies from a fundamental whose period of oscillation is equal to the duration of the performance, are mapped to one of the six head and torso yaw, pitch and roll off-center movements—eliciting the body's prescribed six axes of movement to oscillate at six different harmonically related frequencies. Any six unique harmonic frequencies are selected, but consideration is given to the duration of the performance, where a longer duration suggests higher harmonics and a shorter duration lower harmonics. Additionally, the changing amplitude of a cosine function, at a frequency whose period of oscillation is equal to the duration of the performance, is mapped to the positive and negative peak amplitude values of each of the six sine functions, where the positive peak amplitude of the cosine function correlates to a zero peak amplitude of the sine functions, thus zero degrees of axes movement, and where the negative peak amplitude of the cosine function correlates to a maximum positive and negative peak amplitude of the sine functions, thus maximum positive and negative degrees of axes movement—eliciting the body's prescribed six axes of movement to expand and contract in range over the entire course of the performance. All sine and cosine functions begin at a phase value of zero at the beginning of the performance and subsequently return to zero at the end of the performance. In order to perform the prescribed movements, a score of some kind will prove necessary. Computer code for generating a video score is provided, but if possible, the movements may be memorized.

During the performance, the soloist and audience may sit, stand and/or walk around in any position within the space. The audience may leave and come back at any point and is encouraged to mirror all or part of the movements of the soloist all or part of the time, once or on multiple occasions, on their own terms and timing. Lastly, a group of unannounced performers may be planted in the audience who mirror all or part of the soloist all or part of the time to help guarantee a certain amount of audience movement—hopefully encouraging everyone to join in on the fun.

Robert Blatt, 2015 (revised and fully notated in 2016)

```
//All Together Now
//Robert Blatt, October 2015, revised May/June 2016

//step 1: boot the SuperCollider server and load the provided SuperCollider SynthDefs
//step 2: make any necessary changes to the initial variables in the Processing code and run the Processing sketch

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

//SuperCollider 3.6.6

SynthDef(\sineEnv, { |outbus = 0, equalLoudnessAmp = 0.0, freq = 2000, phase = 0, attackTime = 0.1, sustainTime = 1, releaseTime = 0.1,
gate = 1|
		var env, sine;
		env = EnvGen.kr(Env([0.0, 1.0, 1.0, 0.0], [attackTime, sustainTime, releaseTime], \sine), gate, doneAction: 2);
		sine = SinOsc.ar(freq, phase, equalLoudnessAmp) * env;
		Out.ar(outbus, sine);
}).store;

SynthDef(\sineEnvTest, { |outbus = 0, equalLoudnessAmp = 0.0, freq = 2000, phase = 0, waitTime = 0, attackTime = 0.1, releaseTime = 0.1,
gate = 1|
		var env, sine;
		env = EnvGen.kr(Env.asr(attackTime, 1, releaseTime, \sine), gate, doneAction: 2);
		sine = SinOsc.ar(freq, phase, equalLoudnessAmp) * env;
		Out.ar(outbus, sine);
}).store;

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

//Processing 1.5.1
//dependencies:
//oscP5 0.9.8 library
//SuperCollider 0.3.1 library
//a bitmap font file with Helvetica font at 128 size (can be created with Processing but must be named "Helvetica-128.vlw" and located
//inside a folder named "data" inside the sketch's folder)


//the values of the following variables are to be modified as necessary:

//set testTone to true to choose frequencies for the performance space
//move cursor along frequency track and left click to save frequencies for playback
//set testChord to true to hear the selected frequencies played back as a chord
//set both testTone and testChord to false to run the piece
boolean testTone = true;
boolean testChord = false;

//duration of the piece in seconds
float durationOfPiece = 30.0 * 60.0;

//maximum angle of rotation for yaw, pitch and roll parameters for head and torso
float headYawRange = 33.75;
float headPitchRange = 33.75;
float headRollRange = 33.75;
float torsoYawRange = 33.75;
float torsoPitchRange = 33.75;
float torsoRollRange = 33.75;

//harmonic assignments for yaw, pitch and roll parameters for head and torso
//enter unique harmonic numbers (positive whole numbers)
//their location in the array determines the harmonic assignment of each parameter
//[Head + Yaw, Head + Pitch, Head + Roll, Torso + Yaw, Torso + Pitch, Torso + Roll]
float[] rotationHarmonics = {
  8, 7, 6, 3, 4, 5
};

//outbus channel for SuperCollider SynthDef calls
int outbus = 0;

//adjust pixel width and height for video
//for full screen mode, select the width and height of your display
//formatted around a 16:9 aspect ration, diverging too far from this ratio will cause text to overlap
```

```
int scoreWidth = 1280;
int scoreHeight = 720;


//all remaining code should not need to be modified

import supercollider.*;
import oscP5.*;
import netP5.*;
float[] ranges = {
  headYawRange, headPitchRange, headRollRange, torsoYawRange, torsoPitchRange, torsoRollRange
};
float[] tones = new float[0];
String[] tonesString = new String[0];
int currentToneIndex = 0;
float[] chord;
float currentTone = 20.0;
Synth testSynth;
Synth[] chordSynths;
int specifiedFrameRate = 24;
float currentFrameCount = 0;
PFont font;
int fontSize = 128;
boolean testAudio = true;
boolean stopPiece = false;
float[] rotationsPrior = {
  -1, -1, -1, -1, -1, -1
};
boolean[] rotationsUp = {
  true, true, true, true, true, true
};


void setup() {
  size(scoreWidth, scoreHeight, P2D);
  smooth();
  frameRate(specifiedFrameRate);
  font = loadFont("Helvetica-128.vlw");
  font = createFont("Helvetica-128", fontSize, true);
  if ((testTone == false) && (testChord == false)) {
    testAudio = false;
  }

  if ((testTone == true) && (testChord == false)) {
    for (int i = 0; i < tonesString.length; i++) {
      tonesString[i] = str(tones[i]);
    }
    testSynth = new Synth("sineEnvTest");
    testSynth.set("equalLoudnessAmp", 0.75);
    testSynth.set("freq", 20);
    testSynth.set("phase", 0);
    testSynth.set("outbus", outbus);
    testSynth.set("attackTime", 0.25);
    testSynth.set("releaseTime", 0.25);
    testSynth.create();
  }

  if (((testChord == true) && (testTone == false)) || (testAudio == false)) {
    String[] chordString = loadStrings("Chord.txt");
    chord = float(chordString);
    chordSynths = new Synth[chord.length];
    float equalLoudnessScaler = 0.0;
    float[] chordEqualLoudnessAmps = new float[chord.length];
    float EqualLoudnessAmpsSum = 0.0;
    float ScaleAmpsTo1;
    //Equal-loudness curve to balance chord
    for (int i = 0; i < chordSynths.length; i++) {
      if (chord[i] < 30) {
        equalLoudnessScaler = map(chord[i], 20, 30, 0.0, -10.0);
      }
      if ((chord[i] >= 30) && (chord[i] < 50)) {
        equalLoudnessScaler = map(chord[i], 30, 50, -10.0, -20.0);
      }
      if ((chord[i] >= 50) && (chord[i] < 100)) {
        equalLoudnessScaler = map(chord[i], 50, 100, -20.0, -30.0);
      }
      if ((chord[i] >= 100) && (chord[i] < 200)) {
        equalLoudnessScaler = map(chord[i], 100, 200, -30.0, -40.0);
      }
      if ((chord[i] >= 200) && (chord[i] < 600)) {
        equalLoudnessScaler = map(chord[i], 200, 600, -40.0, -50.0);
      }
      if ((chord[i] >= 600) && (chord[i] < 1000)) {
        equalLoudnessScaler = -50.0;
      }
      if ((chord[i] >= 1000) && (chord[i] < 1500)) {
        equalLoudnessScaler = map(chord[i], 1000, 1500, -50.0, -48.0);
      }
      if ((chord[i] >= 1500) && (chord[i] < 3500)) {
        equalLoudnessScaler = map(chord[i], 1500, 3500, -48.0, -53.0);
      }
      if ((chord[i] >= 3500) && (chord[i] < 10000)) {
        equalLoudnessScaler = map(chord[i], 3500, 10000, -53.0, -38.0);
      }
      if (chord[i] >= 10000) {
        equalLoudnessScaler = constrain(map(chord[i], 10000, 15000, -38.0, -43.0), -43.0, -38.0);
      }
      //convert dB to amplitude
      chordEqualLoudnessAmps[i] = pow(10, (equalLoudnessScaler/20.0));
    }
    //sum all equal loudness adjustments
    for (int i = 0; i < chordEqualLoudnessAmps.length; i++) {
      EqualLoudnessAmpsSum = chordEqualLoudnessAmps[i] + EqualLoudnessAmpsSum;
    }
    ScaleAmpsTo1 = 1.0/EqualLoudnessAmpsSum;

    if ((testChord == true) && (testTone == false)) {
      for (int i = 0; i < chordSynths.length; i++) {
        chordSynths[i] = new Synth("sineEnvTest");
        chordSynths[i].set("equalLoudnessAmp", chordEqualLoudnessAmps[i] * ScaleAmpsTo1 * 0.75);
        chordSynths[i].set("freq", chord[i]);
        chordSynths[i].set("phase", 0);
        chordSynths[i].set("outbus", outbus);
        chordSynths[i].set("attackTime", 0.125);
        chordSynths[i].set("releaseTime", 0.125);
        chordSynths[i].create();
      }
    }

    if (testAudio == false) {
      //reverse chord frequency and equal loudness scaling value order
      chord = reverse(chord);
      chordEqualLoudnessAmps = reverse(chordEqualLoudnessAmps);
      for (int i = 0; i < chordSynths.length; i++) {
        chordSynths[i] = new Synth("sineEnv");
        chordSynths[i].set("equalLoudnessAmp", chordEqualLoudnessAmps[i] * ScaleAmpsTo1 * 0.75);
        chordSynths[i].set("freq", chord[i]);
        chordSynths[i].set("phase", 0);
        chordSynths[i].set("outbus", outbus);
        chordSynths[i].set("attackTime", (durationOfPiece/2.0) * ((i+1.0)/chordSynths.length));
        chordSynths[i].set("sustainTime", (durationOfPiece/2.0) - ((durationOfPiece/2.0) * (1.0/chordSynths.length)));
        chordSynths[i].set("releaseTime", (durationOfPiece/2.0) * (1.0 - ((i*1.0)/chordSynths.length)));
        chordSynths[i].create();
      }
    }
  }
}


void mouseMoved ()
```

```
{
  if ((testTone == true) && (testChord == false)) {
    float freqX = map(constrain(mouseX, 0, scoreWidth), 0, scoreWidth, 20, 80);
    if ((mouseY >= (1/8.0)*scoreHeight) && (mouseY < (2/8.0)*scoreHeight)) {
      freqX = map(constrain(mouseX, 0, scoreWidth), 0, scoreWidth, 320, 80);
    }
    if ((mouseY >= (2/8.0)*scoreHeight) && (mouseY < (3/8.0)*scoreHeight)) {
      freqX = map(constrain(mouseX, 0, scoreWidth), 0, scoreWidth, 320, 1280);
    }
    if ((mouseY >= (3/8.0)*scoreHeight) && (mouseY < (4/8.0)*scoreHeight)) {
      freqX = map(constrain(mouseX, 0, scoreWidth), 0, scoreWidth, 5120, 1280);
    }
    if (mouseY >= (4/8.0)*scoreHeight) {
      freqX = map(constrain(mouseX, 0, scoreWidth), 0, scoreWidth, 5120, 20480);
    }
    currentTone = freqX;
    float testAmp = 0.0;
    //Equal-loudness curve for test tone
    if (freqX < 30) {
      testAmp = map(freqX, 20, 30, 0.0, -10.0);
    }
    if ((freqX >= 30) && (freqX < 50)) {
      testAmp = map(freqX, 30, 50, -10.0, -20.0);
    }
    if ((freqX >= 50) && (freqX < 100)) {
      testAmp = map(freqX, 50, 100, -20.0, -30.0);
    }
    if ((freqX >= 100) && (freqX < 200)) {
      testAmp = map(freqX, 100, 200, -30.0, -40.0);
    }
    if ((freqX >= 200) && (freqX < 600)) {
      testAmp = map(freqX, 200, 600, -40.0, -50.0);
    }
    if ((freqX >= 600) && (freqX < 1000)) {
      testAmp = -50.0;
    }
    if ((freqX >= 1000) && (freqX < 1500)) {
      testAmp = map(freqX, 1000, 1500, -50.0, -48.0);
    }
    if ((freqX >= 1500) && (freqX < 3500)) {
      testAmp = map(freqX, 1500, 3500, -48.0, -53.0);
    }
    if ((freqX >= 3500) && (freqX < 10000)) {
      testAmp = map(freqX, 3500, 10000, -53.0, -38.0);
    }
    if (freqX >= 10000) {
      testAmp = constrain(map(freqX, 10000, 15000, -38.0, -43.0), -43.0, -38.0);
    }
    //convert dB to amplitude
    testAmp = pow(10, (testAmp/20.0));
    testSynth.set("freq", currentTone);
    testSynth.set("equalLoudnessAmp", testAmp * 0.75);
  }
}


void mousePressed ()
{
  if ((testTone == true) && (testChord == false)) {
    boolean notAlreadySelected = true;
    //has currentTone already been chosen
    for (int i = 0; i < tones.length; i++) {
      if (currentTone == tones[i]) {
        notAlreadySelected = false;
      }
    }
    //save currentTone to arrays
    if (notAlreadySelected == true) {
      tones = append(tones, currentTone);
      tonesString = append(tonesString, "");
      currentToneIndex = currentToneIndex + 1;
      tones = sort(tones, currentToneIndex);
      for (int i = 0; i < tonesString.length; i++) {
        tonesString[i] = str(tones[i]);
      }
      saveStrings("Chord.txt", tonesString);
    }
  }
}


void draw() {
  background(0);

  if ((testTone == true) && (testChord == false)) {
    float freqX = map(constrain(mouseX, 0, scoreWidth), 0, scoreWidth, 20, 80);
    if ((mouseY >= (1/8.0)*scoreHeight) && (mouseY < (2/8.0)*scoreHeight)) {
      freqX = map(constrain(mouseX, 0, scoreWidth), 0, scoreWidth, 320, 80);
    }
    if ((mouseY >= (2/8.0)*scoreHeight) && (mouseY < (3/8.0)*scoreHeight)) {
      freqX = map(constrain(mouseX, 0, scoreWidth), 0, scoreWidth, 320, 1280);
    }
    if ((mouseY >= (3/8.0)*scoreHeight) && (mouseY < (4/8.0)*scoreHeight)) {
      freqX = map(constrain(mouseX, 0, scoreWidth), 0, scoreWidth, 5120, 1280);
    }
    if (mouseY >= (4/8.0)*scoreHeight) {
      freqX = map(constrain(mouseX, 0, scoreWidth), 0, scoreWidth, 5120, 20480);
    }
    line(0, (1/8.0)*scoreHeight, (31/32.0)*scoreWidth, (1/8.0)*scoreHeight);
    line((1/32.0)*scoreWidth, (2/8.0)*scoreHeight, scoreWidth, (2/8.0)*scoreHeight);
    line(0, (3/8.0)*scoreHeight, (31/32.0)*scoreWidth, (3/8.0)*scoreHeight);
    line((1/32.0)*scoreWidth, (4/8.0)*scoreHeight, scoreWidth, (4/8.0)*scoreHeight);
    stroke(255);
    String[] frequency = new String[2];
    frequency[0] = str(freqX);
    frequency[1] = " Hz";
    textFont(font, fontSize);
    textSize(round((1.0/16.0)*scoreWidth));
    textAlign(LEFT, BOTTOM);
    text(join(frequency, ""), (1/32.0)*scoreWidth, (26/32.0)*scoreHeight);
    textSize(round((1.0/112.0)*scoreWidth));
    textAlign(LEFT, TOP);
    int lengthValue;
    for (int i = 0; i < tones.length/20.0; i++) {
      if (((i*20)+20) <= tones.length) {
        lengthValue = 20;
      }
      else {
        lengthValue = 20 - (((i*20)+20) - tones.length);
      }
      text(join(subset(tonesString, i*20, lengthValue), ", "), (1/32.0)*scoreWidth, ((26/32.0)*scoreHeight) + (i*(1/32.0)*scoreHeight));
    }
  }

  if ((testChord == true) && (testTone == false)) {
    textFont(font, fontSize);
    textSize(round((1.0/80.0)*scoreWidth));
    textAlign(LEFT, CENTER);
    text("chord in Hz", (1/64.0)*scoreWidth, (1/4.0)*scoreHeight);
    int lengthValue;
    for (int i = 0; i < chord.length/10.0; i++) {
      if (((i*10)+10) <= chord.length) {
        lengthValue = 10;
      }
      else {
        lengthValue = 10 - (((i*10)+10) - chord.length);
      }
      text(join(str(subset(chord, i*10, lengthValue)), ", "), (1/64.0)*scoreWidth, (1/4.0)*scoreHeight+(i*(1/32.0)*scoreHeight
+(1/32.0)*scoreHeight);
    }
```

```
}

if ((testChord == true) && (testTone == true)) {
  textFont(font, fontSize);
  textSize(round((1.0/80.0)*scoreWidth));
  textAlign(LEFT, CENTER);
  text("error: testChord and testTone cannot both be true", (1/64.0)*scoreWidth, (1/2.0)*scoreHeight);
}

if ((testAudio == false) && (stopPiece == false)) {
  //one cycle of fundamental over total duration of piece
  float frequencyInRadians = map(frameCount, 0, (durationOfPiece * specifiedFrameRate), 0.0, (PI * 2));
  //map expansion and contraction of movement range to cosine of fundamental
  float[] rotationsRangeLow = new float[6];
  float[] rotationsRangeHigh = new float[6];
  for (int i = 0; i < ranges.length; i++) {
    rotationsRangeLow[i] = map(cos(frequencyInRadians), 1.0, -1.0, 0.0, ranges[i] - (ranges[i] * 2));
    rotationsRangeHigh[i] = map(cos(frequencyInRadians), 1.0, -1.0, 0.0, ranges[i]);
  }
  //map each movement parameter to a different harmonic from common fundamental
  float[] rotations = new float[6];
  for (int i = 0; i < ranges.length; i++) {
    rotations[i] = map(sin(frequencyInRadians * rotationHarmonics[i]), -1.0, 1.0, rotationsRangeLow[i], rotationsRangeHigh[i]);
  }
  //prepare degree values
  String[] rotationArraysJoined = new String[6];
  for (int i = 0; i < ranges.length; i++) {
    String[] tempArray = new String[2];
    if ((rotations[i] < 10.0) && (rotations[i] > -10.0)) {
      tempArray[0] = nfp(rotations[i], 1, 2);
    }
    else
    {
      tempArray[0] = nfp(rotations[i], 2, 2);
    }
    tempArray[1] = "°";
    rotationArraysJoined[i] = join(tempArray, "");
    if (rotations[i] > rotationsPrior[i]) {
      rotationsUp[i] = true;
    }
    else
    {
      rotationsUp[i] = false;
    }
    rotationsPrior[i] = rotations[i];
  }
  int rowOne = round((1/3.0)*scoreHeight);
  int rowTwo = round((2/3.0)*scoreHeight);
  int rowThree = round(scoreHeight);
  int columnOne = round(scoreWidth);
  int columnTwo = round((1/6.0)*scoreWidth);
  int columnThree = round((3/6.0)*scoreWidth);
  int columnFour = round((5/6.0)*scoreWidth);
  int[] xPositionsForDegrees = {
    columnTwo, columnThree, columnFour, columnTwo, columnThree, columnFour
  };
  int [] yPositionsForDegrees = {
    rowOne, rowOne, rowOne, rowTwo, rowTwo, rowTwo
  };
  textFont(font, fontSize);
  textSize(round((1.0/80.0)*scoreWidth));
  fill(204);
  textAlign(CENTER, BOTTOM);
  pushMatrix();
  translate(columnOne, rowOne);
  rotate((PI * 1.5));
  translate(-columnOne, -rowOne);
  text("HEAD", columnOne, rowOne);
  popMatrix();
  pushMatrix();
```

```
    translate(columnOne, rowTwo);
    rotate((PI * 1.5));
    translate(-columnOne, -rowTwo);
    text("TORSO", columnOne, rowTwo);
    popMatrix();
    text("YAW", columnTwo, rowThree);
    text("PITCH", columnThree, rowThree);
    text("ROLL", columnFour, rowThree);
    textAlign(CENTER, CENTER);
    textSize(round((1.0/14.0)*scoreWidth));
    for (int i = 0; i < ranges.length; i++) {
      if (rotationsUp[i] == true) {
        fill(255, 255, 0);
      }
      else
      {
        fill(0, 0, 255);
      }
      text(rotationArraysJoined[i], xPositionsForDegrees[i], yPositionsForDegrees[i]);
    }
    if (frequencyInRadians >= (PI * 2)) {
      stopPiece = true;
    }
  }
}


void exit() {
  if ((testTone == true) && (testChord == false)) {
    testSynth.set("gate", 0);
  }
  if ((testChord == true) && (testTone == false)) {
    for (int i = 0; i < chordSynths.length; i++) {
      chordSynths[i].set("gate", 0);
    }
  }
  if (testAudio == false) {
    for (int i = 0; i < chordSynths.length; i++) {
      chordSynths[i].free();
    }
  }
  super.exit();
}
```