Sonifying Processing:
The Beads Tutorial

Evan X. Merz

Sonifying Processing: The Beads Tutorial

To download the source code for the examples in this book, visit
http://www.computermusicblog.com/SonifyingProcessing/

The Beads Library as well as the associated documentation can be found at
http://www.beadsproject.net/

Contents

1. Introduction

Recently, a professor in the music department at UC Santa Cruz said in a casual conversation, “Processing is a great language for the arts, but it’s much better with visuals than with sound.” Although he is one of the most well-informed professors of my acquaintance, I had to interrupt him. I told him that “that’s not true any more. Now, we have Beads!”

This tutorial is an introduction to the Beads library for creating music in Processing. Beads is a fantastic tool for sound art creation, bringing together the best qualities of Java with the ease of programming seen in the patcher languages Max and Pure Data. Although the Minim library serves a similar purpose, and has been around for much longer than Beads, for me it has never been a compelling tool for music creation. Beads is fundamentally a music and sound art library at its core. If you are familiar with other computer music programming paradigms, such as Max, PD, SuperCollider or Nyquist, then you will be immediately comfortable using the objects in Beads.

Who is this tutorial for?

This tutorial is aimed at programmers who are already familiar with Processing, and want to start adding sound into their sketches. This tutorial will also be useful to sound artists who want a music programming paradigm that is more like Java or C than Max or PD (although it still draws on ideas from the latter environments).

How to Use this Tutorial

If you’re familiar with Processing, and you’re familiar with other music programming environments, then just pick an example program and jump right in.

If you’ve never worked with a music programming environment before (Max, PD, SuperCollider, Kyma, Reaktor, Tassman, etc) then it’s probably best to start from the beginning and work your way through.

1.1. Processing

It is assumed that you already have a passing familiarity with the Processing programming language. If you haven’t yet written a few basic programs in Processing, then it’s best to first visit http://processing.org/ and work through the tutorials and examples on that site. Additionally, there are three short introductory examples in the code included with this book.

After you work through some or all of that material, meet me back here!

1.2. Installing Beads

First, go the Beads website (http://www.beadsproject.net/) and click the link that says “Beads Library for Processing.” Unzip the file, then copy the “beads” folder into the “libraries” subfolder within your sketchbook. To find out where your sketch book is located, click “File” then “Preferences” within the Processing window. Then there is an option for “Sketchbook Location.” If the “libraries” folder doesn’t exist, then create it.

1.3. The Beads Community

Remember, if you have any problems along the way, you can post your question to the Beads community, The Beads mailing list is probably the best place to look for help (http://groups.google.com/group/beadsproject). That list is monitored by myself, beads creator Ollie Bown, and many other Beads contributors.

2. Beads Basics / Unit Generators

This chapter introduces the basic concepts that underlie the Beads library. Then we look at three of the most commonly used Beads objects: Gain, WavePlayer and Glide.

2.1. Unit Generators

The concept of the *Unit Generator* is a vital concept to all computer music paradigms of which the author is aware. Whether you're working in Max, Pure Data, Super Collider, Nyquist, Reaktor, Tassman or even on a standalone synthesizer, the concept of unit generators is of paramount importance. Unit generators are building blocks of an audio signal chain. A single unit generator encompasses a single function in a synthesis or analysis program.

Unit generators were pioneered by Max Mathews in the late 1950s with the Music-N computer music languages. At the time, Mathews was working at Bell Labs. He was digitizing sound in order to study call clarity in the Bell telephone network. He created software for analyzing phone calls, and as a side project, he wrote the Music-N languages for creating computer music.

Unit generators were not an entirely new concept for Mathews. Even Ada Lovelace recognized the importance of modularity in computation when she envisioned subroutines in the 19th century. Unit generators can be thought of as subroutines that take input, do something to it, then send the results out of their outputs. The input data can be audio data or control data. Audio data is music or sound. Control data is numerical information.

Guitarists are very familiar with the concept of unit generators, but they just refer to them as *guitar pedals*. For instance, on a delay pedal, a guitarist plugs his guitar into the input, then plugs the delay output into his amplifier. After he turns the knob to the delay time desired, he can play his guitar through the delay pedal (unit generator).

Beads works as a series of unit generators (guitar pedals). By plugging one functional unit into the next, we can create complex sound processing routines without having to understand exactly how the underlying processes work.

Everything in the Beads library is encapsulated as a unit generator, which are referred to as *beads* in the Beads library.

2.2. Beads Architecture

The core of the Beads library is the AudioContext object. The AudioContext is the mother bead (unit generator). It is the bead in which all of the other beads exist. Generally, for a Beads program to function, you must create an AudioContext at some point.

An AudioContext can be created with a call to the AudioContext constructor. The constructor creates a connection to the computer's audio hardware, allocates memory for audio processing, and manages the audio processing threads.

```
ac = new AudioContext();
```

Audio processing can be started by calling the start routine:

```
ac.start();
```

But the AudioContext on its own doesn't create any sound. The AudioContext connects to your computer's hardware, and sends audio data to the speakers, but before it can do that, your software must send some audio into the AudioContext.

2.3. Gain / WavePlayer / Glide

In this section, I'm going to introduce some basic objects that will be used in most Beads programs. The Gain object controls the volume of an audio signal. The WavePlayer object is used to generate simple periodic audio signals such as a sine wave. The Glide object is used to send numerical data to other beads.

2.3.1. Generating a Sine Wave (Hello_Sine)

The first real Beads example generates a simple sine wave oscillating at 440Hz. There are a number of important lines to notice in this example. This line gives the rest of the program access to the Beads library. It's necessary to include this line in any beads program.

```
import beads.*;
```

The next four lines form the entirety of our sound generation code. First the AudioContext is initialized. This connects to your system's audio hardware.

```
ac = new AudioContext();
```

Then the program instantiates a WavePlayer object. WavePlayers are used to generate periodic waveforms such as a sine wave. The WavePlayer constructor takes three parameters. The first parameter is the parent AudioContext object. The next parameter is the frequency, or the unit generator that will control the frequency, as we will see shortly. The third parameter is the type of periodic signal that should be generated. In this case we're generating a sine, but other options are Buffer.SQUARE, Buffer.SAW and Buffer.NOISE (more at http://www.beadsproject.net/doc/net/beadsproject/beads/data/Buffer.html).

```
WavePlayer wp = new WavePlayer(ac, 440, Buffer.SINE);
```

Then the WavePlayer is connected to the AudioContext. The addInput routine will be called any time that we connect two unit generators. In this case, we call the addInput function to connect wp, the WavePlayer object, with ac.out, the main output.

```
ac.out.addInput(wp);
```

Finally, we begin processing audio, which will continue until the program is terminated by clicking the close button, or the stop button within the Processing window.

```
ac.start();
```

Code Listing 2.3.1. Hello_Sine.pde

```
// Hello_Sine.pde

// import the beads library
import beads.*;

// create our AudioContext
AudioContext ac;

void setup()
{
  size(400, 300);

  // initialize our AudioContext
  ac = new AudioContext();
```

```
  // create a WavePlayer
  // WavePlayer objects generate a waveform
  WavePlayer wp = new WavePlayer(ac, 440, Buffer.SINE);

  // connect the WavePlayer to the AudioContext
  ac.out.addInput(wp);

  // start audio processing
  ac.start();
}
```

2.3.2. Using Gain (Hello_Gain)

In this section we're going to use the Gain object for the first time. The Gain object can be inserted between a sound generating object and the AudioContext in order to control the volume of that object. Gain objects are instantiated with a constructor that has three parameters. The first parameter is the master AudioContext for the program. The second parameter is the number of inputs and outputs for the gain object. This will usually be 1. The third parameter is the starting value of the Gain object. In this case the starting value is 0.2 or 20%.

```
Gain g = new Gain(ac, 1, 0.2);
```

Then we connect the WavePlayer to the Gain, and the Gain to the AudioContext.

```
g.addInput(wp);
ac.out.addInput(g);
```

Code Listing 2.3.2. Hello_Gain.pde

```
// Hello_Gain.pde

// import the beads library
import beads.*;

// create our AudioContext
AudioContext ac;

void setup()
{
  size(400, 300);

  // initialize our AudioContext
```

```
  ac = new AudioContext();

  // create a WavePlayer
  // WavePlayer objects generate a waveform
  WavePlayer wp = new WavePlayer(ac, 440, Buffer.SINE);

  // create a Gain object
  // Gain objects set the volume
  // Here, we create a Gain with 1 input and output
  // with a fixed volume of 0.2 (50%)
  Gain g = new Gain(ac, 1, 0.2);


  // connect the WavePlayer output to the Gain input
  g.addInput(wp);

  // connect the Gain output to the AudioContext
  ac.out.addInput(g);

  // start audio processing
  ac.start();
}
```

2.3.3. Controlling Gain Using Glide (Hello_Glide_01)

The Gain object is useful to control the volume of an audio signal, but it's more useful when it can be controlled dynamically. To control an object parameter on the fly, we use the Glide object. The Glide object can be thought of as a knob on a guitar pedal or effect box or synthesizer. You can "turn" the knob by giving it a value on the fly.

The Glide constructor has three parameters. The first parameter is the AudioContext. The second parameter is the starting value of held by the Glide. The final value is the *glide time* in milliseconds. The glide time is how long it takes for the Glide to go from one value to another. If the Glide is like a knob, then the glide time how long it takes to turn the knob.

```
gainGlide = new Glide(ac, 0.0, 50);
```

To connect the Glide to the Gain object we don't use the addInput method. Rather, we want to use the Glide to control the Gain value, the loudness. In this case, we can insert the Glide object into the Gain constructor to tell the Gain that it should use the Glide to get its value.

You can see in this Gain constructor that in place of a default value, we have inserted the Glide object. This connects the Gain value to the value of the Glide.

```
g = new Gain(ac, 1, gainGlide);
```

Then, in order to demonstrate the Glide (knob) being used, we want to change the Glide value within the rest of the program. In this example, we use the mouse position to control the Glide value, which in turn controls the Gain value.

```
gainGlide.setValue(mouseX / (float)width);
```

The mouseX, the position of the mouse along the x-axis, is turned into a percentage by dividing by the width of the window, then this is passed to the Glide object.

Code Listing 2.3.3. Hello_Glide_01.pde

```
// Hello_Glide_01.pde

// import the beads library
import beads.*;

// create our AudioContext
AudioContext ac;

// declare our unit generators (Beads) since we will need to
// access them throughout the program
WavePlayer wp;
Gain g;
Glide gainGlide;

void setup()
{
  size(400, 300);

  // Initialize our AudioContext.
  ac = new AudioContext();

  // Create a WavePlayer.
  wp = new WavePlayer(ac, 440, Buffer.SINE);

  // Create the Glide object.
  // Glide objects move smoothly from one value to another.
  // 0.0 is the initial value contained by the Glide.
  // It will take 50ms for it to transition to a new value.
  gainGlide = new Glide(ac, 0.0, 50);

  // Create a Gain object.
  // This time, we will attach the gain amount to the glide
  // object created above.
  g = new Gain(ac, 1, gainGlide);
```

```
  // connect the WavePlayer output to the Gain input
  g.addInput(wp);

  // connect the Gain output to the AudioContext
  ac.out.addInput(g);

  // start audio processing
  ac.start();

  background(0); // set the background to black
  text("Move the mouse to control the sine gain.", 100, 100);
}


void draw()
{
  // in the draw routine, we will update our gain
  // since this routine is called repeatedly, this will
  // continuously change the volume of the sine wave
  // based on the position of the mouse cursor within the
  // Processing window
  gainGlide.setValue(mouseX / (float)width);
}
```

2.3.4. Controlling Frequency Using Glide (Hello_Glide_02)

In the final example in this section, we're going to demonstrate the Glide object for a second time. The Glide object can be used to control virtually any parameter of another bead. In this example, we attach it to the frequency of a WavePlayer by inserting it into the WavePlayer constructor where a starting frequency would normally be indicated.

```
wp = new WavePlayer(ac, frequencyGlide, Buffer.SINE);
```

Code Listing 2.3.4. Hello_Glide_02.pde

```
// Hello_Glide_02.pde


// import the beads library
import beads.*;


// create our AudioContext, which will oversee audio
// input/output
AudioContext ac;


// declare our unit generators (Beads) since we will need to
// access them throughout the program
```

```
WavePlayer wp;
Gain g;
Glide gainGlide;
Glide frequencyGlide;


void setup()
{
  size(400, 300);


  // initialize our AudioContext
  ac = new AudioContext();

  // create the gain Glide object
  // 0.0 is the initial value contained by the Glide
  // it will take 50ms for it to transition to a new value
  gainGlide = new Glide(ac, 0.0, 50);

  // create frequency glide object
  // give it a starting value of 20 (Hz)
  // and a transition time of 50ms
  frequencyGlide = new Glide(ac, 20, 50);

  // create a WavePlayer
  // attach the frequency to frequencyGlide
  wp = new WavePlayer(ac, frequencyGlide, Buffer.SINE);


  // create a Gain object
  // this time, we will attach the gain amount to the glide
  // object created above
  g = new Gain(ac, 1, gainGlide);


  // connect the WavePlayer output to the Gain input
  g.addInput(wp);

  // connect the Gain output to the AudioContext
  ac.out.addInput(g);

  // start audio processing
  ac.start();

  background(0); // set the background to black
  text("The mouse X-Position controls volume.", 100, 100);
  text("The mouse Y-Position controls frequency.", 100, 120);
}


void draw()
{
  // update the gain based on the position of the mouse
  // cursor within the Processing window
  gainGlide.setValue(mouseX / (float)width);


  // update the frequency based on the position of the mouse
  // cursor within the Processing window
```

```
  frequencyGlide.setValue(mouseY);
}
```

3. Synthesis

This chapter demonstrates basic synthesis techniques, as implemented using Processing and Beads. Each section briefly reviews the fundamental concepts associated with a particular synthesis technique, then demonstrates those concepts in Processing. While reading this chapter, try to think of each synthesis technique as a tool in an arsenal of techniques. Remember, there is no right way to synthesize sound, so these techniques can be recombined in a myriad of interesting ways to generate a limitless variety of timbres.

3.1. Using the Mouse

In this book, the examples will often use the mouse as input. If you're an experienced Processing programmer, then you know that this involves two variables. The variables *mouseX* and *mouseY* are continuously updated by Processing with the position of the mouse pointer, as long as the cursor stays within the Processing window. MouseX contains the position of the mouse along the x-axis. MouseY contains the position of the mouse along the y-axis. Often, we turn the mouse position into a percent, in order to use a number between 0.0 and 1.0 as a parameter in a bead. This calculation is relatively straightforward.

```
float xPercent = mouseX / (float)width;
float yPercent = mouseY / (float)height;
```

For more information on mouse input see http://processing.org/learning/basics/

3.2. Additive Synthesis

Additive synthesis is any type of sound-generating algorithm that combines sounds to build more complex sounds. Some form of additive synthesis can be found in virtually any synthesizer. Further, additive synthesis isn't a new concept, or one that is only at work in the domain of electrical sound synthesis.

In fact, the earliest additive instruments are the pipe organs invented in the middle ages. When air is routed through a pipe organ, each pipe in the organ generates a different set of frequencies. By pulling various register stops, the organist can create additive timbres that add the sounds of multiple pipes. This concept was electrified in Thaddeus Cahill's Telharmonium in the early 20th Century, then refined by Hammond in their incredibly successful organs. Today, additive synthesis is everywhere, and although additive techniques have taken a backseat to modulation synthesis techniques, they're still an important part of a synthesist's aresenal.

In this section, we're going to build a number of increasingly complex additive synthesizers.

3.2.1. Two Sine Waves Controlled by the Mouse (Additive_01)

In the first example, we build on the patch seen in chapter 2 called Hello_Glide_02. In that example, a Glide object controls the frequency of a WavePlayer object. In this example, two different glide objects are used to control the frequencies of two sine waves. In these two lines, a Glide object is initialized, then mapped to the frequency of a WavePlayer.

```
frequencyGlide1 = new Glide(ac, 20, 50);
wp1 = new WavePlayer(ac, frequencyGlide1, Buffer.SINE);
```

Those two lines are repeated for a second Glide object and a second WavePlayer. Then in the draw routine, the frequency of the Glide object is controlled by the mouse. One WavePlayer is controlled by the x-position of the mouse, while the other is controlled by the y-position.

```
frequencyGlide1.setValue(mouseY);
```

If you peruse the entire example, you might notice that there is no "Additive" object. The sine waves are actually summed by the Gain object. By simply routing both WavePlayer unit generators into the same Gain, we can combine the signals and create an additive synthesizer.

```
g.addInput(wp1);
g.addInput(wp2);
```

Code Listing 3.2.1. Additive_01.pde

```
// Additive_01.pde
```

```
// import the beads library
import beads.*;


// create our AudioContext
AudioContext ac;


// declare our unit generators (Beads) since we will need to
// access them throughout the program
WavePlayer wp1;
Glide frequencyGlide1;
WavePlayer wp2;
Glide frequencyGlide2;


Gain g;


void setup()
{
  size(400, 300);


  // initialize our AudioContext
  ac = new AudioContext();

  // create frequency glide object
  // give it a starting value of 20 (Hz)
  // and a transition time of 50ms
  frequencyGlide1 = new Glide(ac, 20, 50);

  // create a WavePlayer, attach the frequency to
  // frequencyGlide
  wp1 = new WavePlayer(ac, frequencyGlide1, Buffer.SINE);


  // create the second frequency glide and attach it to the
  // frequency of a second sine generator
  frequencyGlide2 = new Glide(ac, 20, 50);
  wp2 = new WavePlayer(ac, frequencyGlide2, Buffer.SINE);


  // create a Gain object to make sure we don't peak
  g = new Gain(ac, 1, 0.5);


  // connect both WavePlayers to the Gain input
  g.addInput(wp1);
  g.addInput(wp2);

  // connect the Gain output to the AudioContext
  ac.out.addInput(g);

  // start audio processing
  ac.start();
}
```

```
void draw()
{
  // update the frequency based on the position of the mouse
  // cursor within the Processing window
  frequencyGlide1.setValue(mouseY);
  frequencyGlide2.setValue(mouseX);
}
```

3.2.2. Many Sine Waves with Fundamental Controlled by the Mouse (Additive_02)

This example is a more typical additive synthesis patch. Rather than combining a number of sine waves at unrelated frequencies, we combine sine waves that are multiples of the lowest frequency. The lowest frequency in an additive tone is called the *fundamental frequency*. If a tone is an integer multiple of the fundamental frequency then it is called a *harmonic* or *harmonic partial*. If a sine is not an integer multiple of the fundamental, then it is called a *partial* or *inharmonic partial*. In this example, we sum a sine wave with it's first 9 harmonics. We update the frequency of each sine wave as the mouse moves around the program window.

This example is our first use of arrays of unit generators. In this case there is an array of Glide objects, an array of WavePlayer objects, and an array of Gain objects. Each sine wave in the additive tone requires its own set of unit generators. This presents the reader with the biggest problem with additive synthesis. Additive synthesis is computationally complex. A synthesis program like this one must have a separate set of unit generators for each component of the output spectrum. As we will see in the next section, we can use modulation synthesis to create complex timbres while consuming fewer computer resources.

Code Listing 3.2.2. Additive_02.pde

```
// Additive_02.pde


// this is a more serious additive synthesizer
// understanding this code requires a basic understanding of
// arrays, as they are used in Processing


// import the beads library
import beads.*;


// create our AudioContext
AudioContext ac;

// the frequency of the fundamental (the lowest sine wave in
// the additive tone)
```

```
float baseFrequency = 200.0f;
// how many sine waves will be present in our additive tone?
int sineCount = 10;


// declare our unit generators
// notice that with the brackets []
// we are creating arrays of beads
WavePlayer sineTone[];
Glide sineFrequency[];
Gain sineGain[];


// our master gain object (all sine waves will eventually be
// routed here)
Gain masterGain;


void setup()
{
  size(400, 300);


  // initialize our AudioContext
  ac = new AudioContext();

  // set up our master gain object
  masterGain = new Gain(ac, 1, 0.5);
  ac.out.addInput(masterGain);


  // initialize our arrays of objects
  sineFrequency = new Glide[sineCount];
  sineTone = new WavePlayer[sineCount];
  sineGain = new Gain[sineCount];


  float currentGain = 1.0f;
  for( int i = 0; i < sineCount; i++)
  {
      // create the glide that will control this WavePlayer's
      // frequency
      sineFrequency[i] = new Glide(ac,
                                   baseFrequency * (i + 1),
                                   30);
      // create the WavePlayer
      sineTone[i] = new WavePlayer(ac,
                                   sineFrequency[i],
                                   Buffer.SINE);

      // create the gain object
      sineGain[i] = new Gain(ac, 1, currentGain);
      // then connect the waveplayer to the gain
      sineGain[i].addInput(sineTone[i]);

      // finally, connect the gain to the master gain
      masterGain.addInput(sineGain[i]);

      // lower the gain for the next sine in the tone
```

```
      currentGain -= (1.0 / (float)sineCount);
  }


  // start audio processing
  ac.start();
}


void draw()
{
  // update the fundamental frequency based on mouse position
  // add 20 to the frequency because below 20Hz is inaudible
  // to humans
  baseFrequency = 20.0f + mouseX;

  // update the frequency of each sine tone
  for( int i = 0; i < sineCount; i++)
  {
      sineFrequency[i].setValue(baseFrequency * (i + 1));
  }
}
```

3.2.3. Additive Synthesis Controlled by a Processing Sketch (Additive_03)

The final additive synthesis example is similar to the previous example, except we map the fundamental frequency to the location of an on-screen object. If you are adding sound to a Processing sketch, then you will want to map your on-screen objects to sound parameters in some way. In this sketch, we simply control frequency based on object location, but a mapping need not be so direct or so obvious.

Code Listing 3.2.3. Additive_03.pde

```
// Additive_03.pde


// this is a more serious additive synthesizer
// understanding this code requires a basic understanding of
// arrays, as they are used in Processing


import beads.*; // import the beads library
AudioContext ac; // declare our AudioContext


float baseFrequency = 200.0f; // fundamental frequency
int sineCount = 10; // how many sine waves will be present


// declare our unit generators
WavePlayer sineTone[];
Glide sineFrequency[];
```

```
Gain sineGain[];


// our master gain object
Gain masterGain;


// this is a ball that will bounce around the screen
bouncer b;


void setup()
{
  size(400, 300);

  // initialize our bouncy bal
  b = new bouncer();


  // initialize our AudioContext
  ac = new AudioContext();

  // set up our master gain object
  masterGain = new Gain(ac, 1, 0.5);
  ac.out.addInput(masterGain);


  // initialize our arrays of objects
  sineFrequency = new Glide[sineCount];
  sineTone = new WavePlayer[sineCount];
  sineGain = new Gain[sineCount];


  float currentGain = 1.0f;
  for( int i = 0; i < sineCount; i++)
  {
      // create the glide that will control this WavePlayer's
      // frequency
      sineFrequency[i] = new Glide(ac,
                                   baseFrequency * i,
                                   30);
      // create the WavePlayer
      sineTone[i] = new WavePlayer(ac,
                                   sineFrequency[i],
                                   Buffer.SINE);

      // create the gain object
      sineGain[i] = new Gain(ac, 1, currentGain);
      // then connect the waveplayer to the gain
      sineGain[i].addInput(sineTone[i]);

      // finally, connect the gain to the master gain
      masterGain.addInput(sineGain[i]);

      // lower the gain for the next tone in the additive
      // complex
      currentGain -= (1.0 / (float)sineCount);
  }
```

```
  // start audio processing
  ac.start();
}


void draw()
{
  background(0); // fill the background with black

  b.move(); // move the bouncer
  b.draw(); // draw the bouncer

  // update the fundamental frequency based on mouse position
  baseFrequency = 20.0f + b.x;

  // update the frequency of each sine tone
  for( int i = 0; i < sineCount; i++)
  {
      sineFrequency[i].setValue(baseFrequency *
                          ((float)(i+1) * (b.y/height)));
  }
}


// this class encapsulates a simple circle that will bounce
// around the Processing window
class bouncer
{
  public float x = 10.0;
  public float y = 10.0;
  float xSpeed = 1.0;
  float ySpeed = 1.0;

  void bouncer() { }

  void move()
  {
    x += xSpeed;
    if( x <= 0 ) xSpeed = 1.0;
    else if( x >= width - 10 ) xSpeed = -1.0;

    y += ySpeed;
    if( y <= 0 ) ySpeed = 1.0;
    else if( y >= height - 10 ) ySpeed = -1.0;
  }

  void draw()
  {
    noStroke();
    fill(255);
    ellipse(x, y, 10, 10);
  }
}
```

3.3. Modulation Synthesis

In modulation synthesis, a signal called the modulator is used to effect another signal, called the carrier. In this situation, the modulator controls some parameter of the carrier signal. The modulator might control the amplitude, frequency or filter frequency of a signal.

For example, if the modulator is used to control the amplitude of the carrier signal, and the modulator is a sine wave oscillating at 2 Hertz (2 oscillations per second), then the amplitude of the carrier signal will rise and fall twice in a second. This is a very familiar effect, and at the subaudible range (under 20Hz), we call this effect *tremolo*. If the frequency is varied at a subaudible frequency, then we call the effect *vibrato*.

When the frequency of the modulator rises into the audible range, above 20Hz, then new frequencies are added to the carrier signal. These frequencies are called sidebands, and they have different characteristics based on the type of modulation synthesis. These sidebands are what make modulation synthesis so powerful. With a modulation synthesizer, we can create interesting broad spectrum sounds with a small number of source signals.

In this chapter, we are going to demonstrate how to construct modulation synthesis modules in Beads. We will construct both frequency modulation and amplitude modulation synthesizers, and in the process introduce the concept of custom functions in Beads.

3.3.1. Frequency Modulation (Frequency_Modulation_01)

In the first modulation synthesis example, a simple frequency modulation synthesizer is constructed. It is not interactive, and only generates the sound of a frequency modulation tone with a carrier sine wave oscillating at 200Hz and a modulator sine wave oscillating at 40Hz.

This example is the first situation where we need to extend beyond the standard unit generators provided by the Beads library*. In this case we're going to use a custom function, one of the most valuable and versatile tools provided by the Beads library. By using a custom function, we can build a simple unit generator on the fly, and then use it as we would any other unit generator.

To build a custom function, we simply need to declare it, then override the calculate routine. The calculate function calculates the output of the new unit generator, using whatever other unit generators are provided in the function declaration. In this case, we pass in the modulator unit generator, a WavePlayer that generates a sine wave at 40Hz. To use the value of the modulator unit generator in a calculation, we simply reference x[0]. If we passed multiple unit generators into a custom function, then they would be accessed via x[1], x[2], x[3] and so on. Here is the code for our frequency modulation custom function.

```
  Function frequencyModulation = new Function(modulator)
  {
    public float calculate() {
      // return x[0], which is the original value of the
      // modulator signal (a sine wave)
      // multiplied by 50 to make the sine
      // vary between -50 and 50
      // then add 200, so that it varies from 150 to 250
      return (x[0] * 50.0) + 200.0;
    }
  };
```

After building our custom unit generator, we can use it in our program. In this program, we want to use it to control the frequency of a WavePlayer object. This is accomplished by using it in the WavePlayer constructor in the place where we might normally indicate a frequency.

```
carrier = new WavePlayer(ac,
                         frequencyModulation,
                         Buffer.SINE);
```

* In truth, this could be accomplished using pre-packaged unit generators.

Code Listing 3.3.1. Frequency_Modulation_01.pde

```
// Frequency_Modulation_01.pde


import beads.*; // import the beads library
AudioContext ac; // create our AudioContext


// declare our unit generators
WavePlayer modulator;
WavePlayer carrier;


Gain g;


void setup()
{
  size(400, 300);


  // initialize our AudioContext
  ac = new AudioContext();

  // create the modulator, this WavePlayer will control
  // the frequency of the carrier
  modulator = new WavePlayer(ac, 40, Buffer.SINE);
```

```
  // This is a custom function
  // Custom functions are simple custom Unit Generators.
  // Generally, they only override the calculate function.
  Function frequencyModulation = new Function(modulator)
  {
    public float calculate() {
      // return x[0], which is the original value of
      // the modulator signal (a sine wave)
      // multiplied by 50 to make the sine vary
      // between -50 and 50
      // then add 200, so that it varies from 150 to 250
      return (x[0] * 50.0) + 200.0;
    }
  };


  // create a second WavePlayer, but this time,
  // control the frequency with the function created above
  carrier = new WavePlayer(ac,
                           frequencyModulation,
                           Buffer.SINE);


  // create a Gain object to make sure we don't peak
  g = new Gain(ac, 1, 0.5);


  // connect the carrier to the Gain input
  g.addInput(carrier);

  // connect the Gain output to the AudioContext
  ac.out.addInput(g);

  // start audio processing
  ac.start();
}
```

3.3.2. Frequency Modulation Controlled by the Mouse (Frequency_Modulation_02)

The second frequency modulation example is similar to the first, except we control the frequencies of the carrier and the modulator using the position of the mouse cursor. The frequency of the carrier is controlled within the frequency modulation function by the position along the y-axis.

```
return (x[0] * 200.0) + mouseY;
```

The frequency of the modulator is controlled by a Glide and updated continuously in the draw routine. The modulator frequency is mapped to the mouse position along the x-axis.

```
modulatorFrequency.setValue(mouseX);
```

Code Listing 3.3.2. Frequency_Modulation_02.pde

```
// Frequency_Modulation_02.pde


import beads.*; // import the beads library
AudioContext ac; // create our AudioContext


// declare our unit generators
WavePlayer modulator;
Glide modulatorFrequency;
WavePlayer carrier;


Gain g;


void setup()
{
  size(400, 300);


  // initialize our AudioContext
  ac = new AudioContext();

  // create the modulator, this WavePlayer
  // will control the frequency of the carrier
  modulatorFrequency = new Glide(ac, 20, 30);
  modulator = new WavePlayer(ac,
                             modulatorFrequency,
                             Buffer.SINE);


  // this is a custom function
  // custom functions are a bit like custom Unit Generators
  // but they only override the calculate function
  Function frequencyModulation = new Function(modulator)
  {
    public float calculate() {
      // return x[0], which is the original value of the
      // modulator signal (a sine wave)
      // multiplied by 200.0 to make the sine
      // vary between -200 and 200
      // the number 200 here is called the "Modulation Index"
      // the higher the Modulation Index,
      // the louder the sidebands
      // then add mouseY, so that it varies
      // from mouseY - 200 to mouseY + 200
      return (x[0] * 200.0) + mouseY;
    }
  };
```

```
  // create a second WavePlayer, control the frequency
  // with the function created above
  carrier = new WavePlayer(ac,
                           frequencyModulation,
                           Buffer.SINE);


  // create a Gain object to make sure we don't peak
  g = new Gain(ac, 1, 0.5);


  // connect the carrier to the Gain input
  g.addInput(carrier);

  // connect the Gain output to the AudioContext
  ac.out.addInput(g);

  ac.start(); // start audio processing
}


void draw()
{
  modulatorFrequency.setValue(mouseX);
}
```

3.3.3. Ring Modulation (Ring_Modulation_01)

The final modulation synthesis example demonstrates how to build a ring modulation synthesizer using Beads. As previously mentioned, in modulation synthesis one unit generator is controlling another unit generator. So calling this technique ring modulation doesn't make much sense. In fact, the name is derived from the shape of the circuit that is used when this synthesis is implemented using electronic components. On digital systems, such as our computers, the meaning of the name is lost. Implementing ring modulation synthesis is as easy as multiplying two sine waves.

As in the previous example, the mouse position controls the frequencies of the carrier and the modulator. Also, a custom function is used to run the ring modulation equation. In this example, however, the custom function isn't used to drive another unit generator. Rather, it is used as a standalone unit generator that takes two input unit generators and multiplies their values.

```
  // a custom function for Ring Modulation
  // Remember, Ring Modulation = Modulator[t] * Carrier[t]
  Function ringModulation = new Function(carrier, modulator)
  {
    public float calculate() {
      // multiply the value of modulator by
```

```
      // the value of the carrier
      return x[0] * x[1];
    }
  };
```

Then we connect the ringModulation unit generator to a gain, and connect that gain to the main output.

```
g.addInput(ringModulation);
ac.out.addInput(g);
```

The result of ring modulation synthesis is a signal with two frequency components. The original frequencies of the carrier and modulator are eliminated by the multiplication. In place of them are two sidebands that occur at the sum and difference of the frequencies of the carrier and the modulator.

One popular modification of ring modulation synthesis is called *Amplitude Modulation*. Amplitude modulation is implemented the same as ring modulation, except one of the input signals is kept unipolar, either entirely positive or entirely negative. This can be implemented in Beads by simply modifying the custom function to call the absolute value function on one of the values in the multiplication.

```
return x[0] * abs(x[1]);
```

Code Listing 3.3.3. Ring_Modulation_01.pde

```
// Ring_Modulation_01.pde

import beads.*; // import the beads library
AudioContext ac; // declare our AudioContext

// declare our unit generators
WavePlayer modulator;
Glide modulatorFrequency;
WavePlayer carrier;
Glide carrierFrequency;

Gain g; // our master gain

void setup()
{
  size(400, 300);

  // initialize our AudioContext
```

```
  ac = new AudioContext();

  // create the modulator
  modulatorFrequency = new Glide(ac, 20, 30);
  modulator = new WavePlayer(ac,
                             modulatorFrequency,
                             Buffer.SINE);

  // create the carrier
  carrierFrequency = new Glide(ac, 20, 30);
  carrier = new WavePlayer(ac,
                           carrierFrequency,
                           Buffer.SINE);


  // a custom function for Ring Modulation
  // Remember, Ring Modulation = Modulator[t] * Carrier[t]
  Function ringModulation = new Function(carrier, modulator)
  {
    public float calculate() {
      // multiply the value of modulator by
      // the value of the carrier
      return x[0] * x[1];
    }
  };


  // create a Gain object to make sure we don't peak
  g = new Gain(ac, 1, 0.5);


  // connect the ring modulation to the Gain input
  // IMPORTANT: Notice that a custom function
  // can be used just like a UGen! This is very powerful!
  g.addInput(ringModulation);

  // connect the Gain output to the AudioContext
  ac.out.addInput(g);

  ac.start(); // start audio processing
}


void draw()
{
  // set the freqiencies of the carrier and the
  // modulator based on the mouse position
  carrierFrequency.setValue(mouseY);
  modulatorFrequency.setValue(mouseX);
}
```

3.4. Envelopes

Modulation synthesis is very useful when we want to change a sound very rapidly, but sometimes we want synthesis parameters to change more slowly. When this is the case, we us a time-varying signal called an envelope. An envelope is a signal that rises and falls over a period of time, usually staying within the range 0.0 to 1.0. Most commonly, envelopes are used to control amplitude (gain), but they can be used to control any aspect of a synthesizer or a sampler.

The two most common types of envelopes are Attack-Decay (AD) and Attack-Decay-Sustain-Release (ADSR). AD envelopes rise from 0.0 to 1.0 over a length of time called the *Attack*. Then they fall back to 0.0 over a length of time called the *Decay*. ADSR envelopes rise to 1 during the attack, then fall to a *sustain* value, where they stay until the event ends, and the value falls to 0 over a time called the *Release*.

3.4.1. Frequency Modulation with an Amplitude Envelope (Frequency_Modulation_03)

In the final synthesis example, we're going to attach an envelope to the frequency modulation synthesizer that we created in Frequency_Modulation_02. The envelope will control a Gain object that sets the volume of the synthesized tone. We will implement a simple Attack-Decay envelope that will allow us to create distinct sound events, rather than just one long continuous tone.

The Envelope constructor takes two parameters. As usual, the first parameter is the master AudioContext object. The second parameter is the starting value. Already, you can see that Envelopes are very similar to Glide objects.

```
gainEnvelope = new Envelope(ac, 0.0);
```

Then we connect the Envelope to the Gain object by inserting it into the Gain constructor where we would normally indicate a starting value.

```
synthGain = new Gain(ac, 1, gainEnvelope);
```

Envelope objects can be thought of like automatic Glide objects that can take a series of commands. With a Glide object we can tell it to take a certain value over a certain length of time, but with an Envelope object, we can give it a number of such commands which it will execute one after the other. In this example, we tell the Envelope to rise to 0.8 over 50 milliseconds, then fall back to 0.0 over 300ms.

```
gainEnvelope.addSegment(0.8, 50); // over 50 ms rise to 0.8
```

```
gainEnvelope.addSegment(0.0, 300); // over 300ms fall to 0.0
```

For more on envelopes, see http://www.beadsproject.net/doc/net/beadsproject/beads/ugens/Envelope.html

Code Listing 3.4.1. Frequency_Modulation_03.pde

```
// Frequency_Modulation_03.pde


import beads.*; // import the beads library
AudioContext ac; // create our AudioContext


// declare our unit generators
WavePlayer modulator;
Glide modulatorFrequency;
WavePlayer carrier;


// our envelope and gain objects
Envelope gainEnvelope;
Gain synthGain;


void setup()
{
  size(400, 300);


  // initialize our AudioContext
  ac = new AudioContext();

  // create the modulator, this WavePlayer will
  // control the frequency of the carrier
  modulatorFrequency = new Glide(ac, 20, 30);
  modulator = new WavePlayer(ac,
                             modulatorFrequency,
                             Buffer.SINE);


  // create a custom frequency modulation function
  Function frequencyModulation = new Function(modulator)
  {
    public float calculate() {
      // return x[0], scaled into an appropriate
      // frequency range
      return (x[0] * 100.0) + mouseY;
    }
  };


  // create a second WavePlayer, control the frequency
  // with the function created above
  carrier = new WavePlayer(ac,
```

```
                              frequencyModulation,
                              Buffer.SINE);


  // create the envelope object that will control the gain
  gainEnvelope = new Envelope(ac, 0.0);


  // create a Gain object, connect it to the gain envelope
  synthGain = new Gain(ac, 1, gainEnvelope);


  // connect the carrier to the Gain input
  synthGain.addInput(carrier);

  // connect the Gain output to the AudioContext
  ac.out.addInput(synthGain);

  ac.start(); // start audio processing

  background(0); // set the background to black
  text("Click to trigger the gain envelope.", 100, 120);
}


void draw()
{
  // set the modulator frequency based on mouse position
  modulatorFrequency.setValue(mouseX);
}


// this routine is triggered when a mouse button is pressed
void mousePressed()
{
  // when the mouse button is pressed,
  // add a 50ms attack segment to the envelope
  // and a 300 ms decay segment to the envelope
  gainEnvelope.addSegment(0.8, 50); // over 50ms rise to 0.8
  gainEnvelope.addSegment(0.0, 300); // in 300ms fall to 0.0
}
```

4. Sampling

Sampling, as we use the term here, is any use of pre-recorded audio in music production. It doesn't necessarily mean that you're using someone else's work, as it implied in the early days of sampler technology. Rather, it just means that you are using an audio sample, a bit of recorded audio. This terminology is wonderful for a tutorial on Beads because the Beads library employs a Sample class to encapsulate audio data.

4.1. Playing Recorded Audio

In this section, we're going to look at the SamplePlayer object. The SamplePlayer object is the default Beads object for playing back audio, and working with it in a number of ways.

4.1.1. Loading and Playing a Sound Using SamplePlayer (Sampling_01)

IMPORTANT: For all of the sampling examples, you will need to have audio files where the program is looking for them. If you're copy/pasting this code from the text, then you will need to set up new files in place of the audio files that are provided when the code is downloaded online. To load an arbitrary audio file at run time, you can call the selectInput() function wherever you would normally indicate a file name string.

In the first example, we are going to demonstrate how to set up and use the SamplePlayer object. The first important step in this process is telling the program where the audio file is located. In this example, I stored the audio file in a directory that is in the same directory as the processing sketch. To indicate the directory where the processing sketch is located we use the `sketchPath("")` routine. To indicate the subdirectory and file name, we add `"DrumMachine/Snaredrum 1.wav"` to the result.

```
sourceFile = sketchPath("") + "DrumMachine/Snaredrum 1.wav";
```

Then we need to initialize the SamplePlayer. To do so, we call its constructor with two parameters. The first parameter is the master AudioContext. The second parameter is the Sample that we want to load into the SamplePlayer. In this case, we're constructing a new Sample on the fly based on the filename created earlier. Notice that this code is encapsulated within a try/catch block. Any time you access the file system, where there is the possibility that a file might not be found, you must encapsulate the code in a try/catch block so that the program can handle errors.

```
  try {
    // initialize our SamplePlayer, loading the file
    // indicated by the sourceFile string
    sp = new SamplePlayer(ac, new Sample(sourceFile));
   }
  catch(Exception e)
  {
    …
  }
```

After the SamplePlayer is created, we set the KillOnEnd parameter, then connect it to a Gain object which is in turn connected to the AudioContext. To actually trigger playback of the sample, we respond to mouse clicks in the mousePressed routine.

```
// this routine is called whenever a mouse button is pressed
// on the Processing sketch
void mousePressed()
{
  // set the gain based on mouse position
  gainValue.setValue((float)mouseX/(float)width);
  // move the playback pointer to the first loop point (0.0)
  sp.setToLoopStart();
  sp.start(); // play the audio file
}
```

In this block of code, we set the gain based on the mouse cursor position. Then we tell the SamplePlayer to move to the start of the file, using the setToLoopStart routine. Finally, we call the start routine to actually trigger playback.

Code Listing 4.1.1. Sampling_01.pde

```
// Sampling_01.pde

import beads.*;
```

```
AudioContext ac;


// this will hold the path to our audio file
String sourceFile;
// the SamplePlayer class will play the audio file
SamplePlayer sp;
Gain g;
Glide gainValue;


void setup()
{
  size(800, 600);

  ac = new AudioContext(); // create our AudioContext


  // What file will we load into our SamplePlayer?
  // Notice the use of the sketchPath function.
  // This is a very useful function for loading external
  // files in Processing.
  sourceFile = sketchPath("") +
               "DrumMachine/Snaredrum 1.wav";


  // Whenever we load a file, we need to enclose
  // the code in a Try/Catch block.
  // Try/Catch blocks will inform us if the file
  // can't be found
  try {
    // initialize our SamplePlayer, loading the file
    // indicated by the sourceFile string
    sp = new SamplePlayer(ac, new Sample(sourceFile));
   }
  catch(Exception e)
  {
    // If there is an error, show an error message
    // at the bottom of the processing window.
    println("Exception while attempting to load sample!");
    e.printStackTrace(); // print description of the error
    exit(); // and exit the program
  }

  // SamplePlayer can be set to be destroyed when
  // it is done playing
  // this is useful when you want to load a number of
  // different samples, but only play each one once
  // in this case, we would like to play the sample multiple
  // times, so we set KillOnEnd to false
  sp.setKillOnEnd(false);


  // as usual, we create a gain that will control the volume
  // of our sample player
  gainValue = new Glide(ac, 0.0, 20);
  g = new Gain(ac, 1, gainValue);
```

```
  g.addInput(sp); // connect the SamplePlayer to the Gain

  ac.out.addInput(g); // connect the Gain to the AudioContext

  ac.start(); // begin audio processing

  background(0); // set the background to black
  text("Click to demonstrate the SamplePlayer object.",
         100, 100); // tell the user what to do!
}


// Although we're not drawing to the screen, we need to
// have a draw function in order to wait for
// mousePressed events.
void draw(){}


// this routine is called whenever a mouse button is
// pressed on the Processing sketch
void mousePressed()
{
  // set the gain based on mouse position
  gainValue.setValue((float)mouseX/(float)width);
  // move the playback pointer to the first loop point (0.0)
  sp.setToLoopStart();
  sp.start(); // play the audio file
}
```

4.1.2. Playing Multiple Sounds and Playing in Reverse (Sampling_02)

In the second sampling example, we are going to setup two SamplePlayer objects. One will play a sample forward, as in the previous example. The other will play a sample in reverse. The first will respond to the left mouse button, while the latter will respond to the right mouse button.

This example is very similar to the previous example. We initialize the SamplePlayer objects in the same way, making sure to enclose them in a try/catch block. Then we set their parameters and connect them to the output in the same way as before. The second SamplePlayer is set up slightly differently here. We set up a Glide object to control the playback rate for the second SamplePlayer.

```
rateValue = new Glide(ac, 1, 20);
sp2.setRate(rateValue)
```

In the mousePressed routine, we check for which button was pressed, then trigger the appropriate SamplePlayer. We trigger the first SamplePlayer in the same way as before.

```
// move the playback pointer to the beginning of the sample
sp1.setToLoopStart();
sp1.start(); // play the audio file
```

We trigger the second SamplePlayer so that it will play its file in reverse. This is done by calling the `setToEnd()` routine, setting the playback rate to -1, then calling the start routine.

```
// set the playback pointer to the end of the sample
sp2.setToEnd();
// set the rate to -1 to play backwards
rateValue.setValue(-1.0);
sp2.start(); // play the audio file
```

Code Listing 4.1.2. Sampling_02.pde

```
// Sampling_02.pde
// in this example, we load and play two samples
// one forward, and one in reverse


import beads.*;


AudioContext ac;
SamplePlayer sp1;


// declare our second SamplePlayer, and the Glide that
// will be used to control the rate
SamplePlayer sp2;
Glide rateValue;


// we can run both SamplePlayers through the same Gain
Gain g;
Glide gainValue;


void setup()
{
  size(800, 600);

  ac = new AudioContext(); // create our AudioContext


  // whenever we load a file, we need to enclose the code in
  // a Try/Catch block
  // Try/Catch blocks will inform us if the file can't be
```

```
  // found
  try {
    // initialize the first SamplePlayer
    sp1 = new SamplePlayer(ac,
                           new Sample(sketchPath("") +
                           "DrumMachine/Snaredrum 1.wav"));
    sp2 = new SamplePlayer(ac,
                           new Sample(sketchPath("") +
                           "DrumMachine/Soft bassdrum.wav"));
   }
  catch(Exception e)
  {
    // if there is an error, show an error message
    // at the bottom of the processing window
    println("Exception while attempting to load sample!");
    e.printStackTrace();
    exit();
  }

  // for both SamplePlayers, note that we want to
  // play the sample multiple times
  sp1.setKillOnEnd(false);
  sp2.setKillOnEnd(false);

  // initialize our rateValue Glide object
  // a rate of -1 indicates that this sample will be
  // played in reverse
  rateValue = new Glide(ac, 1, -1);
  sp2.setRate(rateValue);


  // as usual, we create a gain that will control the
  // volume of our sample player
  gainValue = new Glide(ac, 0.0, 20);
  g = new Gain(ac, 1, gainValue);
  g.addInput(sp1);
  g.addInput(sp2);


  ac.out.addInput(g); // connect the Gain to the AudioContext
  ac.start(); // begin audio processing

  background(0); // set the background to black
  text("Left click to play a snare sound.", 100, 100);
  text("Right click to play a reversed kick drum sound.",
       100, 120);
}


// although we're not drawing to the screen, we need to
// have a draw function in order to wait
// for mousePressed events
void draw(){}


// this routine is called whenever a mouse button is
// pressed on the Processing sketch
void mousePressed()
```

```
{
  // set the gain based on mouse position
  gainValue.setValue((float)mouseX/(float)width);

  // if the left mouse button is clicked, then play the
  // snare drum sample
  if( mouseButton == LEFT )
  {
    // move the playback pointer to the beginning
    // of the sample
    sp1.setToLoopStart();
    sp1.start(); // play the audio file
  }
  // if the right mouse button is clicked, then play
  // the bass drum sample backwards
  else
  {
    // set the playback pointer to the end of the sample
    sp2.setToEnd();
    // set the rate to -1 to play backwards
    rateValue.setValue(-1.0);
    sp2.start(); // play the audio file
  }
}
```

4.1.3. Controlling Playback Rate Using Mouse & Glide (Sampling_03)

In this example, we build a SamplePlayer where the playback rate can be controlled by the mouse. The mouse position along the x-axis determines the playback rate. If the cursor is in the left half of the window, then the playback rate will be negative; the file will play in reverse. If the cursor is in the right half of the window then the file will play forward. The closer the cursor is to the center of the screen, the slower playback will be in either direction.

To accomplish this, we set up a single SamplePlayer as before. Again, we attach a Glide object to the playback rate by calling `sp1.setRate(rateValue)`. Then we just need to set the rate value based on cursor position. We do this in the draw routine, which is called over and over again in Processing programs.

```
rateValue.setValue(((float)mouseX - halfWidth)/halfWidth);
```

We also need to make sure that the playback pointer is in the right place. If the playback pointer is at the beginning of the file, but the rate is set to play in reverse, then there would be no audio to play. So we set the playback position based on the cursor location when the user clicks.

```
  // if the left mouse button is clicked, then
```

```
  // play the sound
  if( mouseX > width / 2.0 )
  {
    // set the start position to the beginning
    sp1.setPosition(000);
    sp1.start(); // play the audio file
  }
  // if the right mouse button is clicked,
  // then play the bass drum sample backwards
  else
  {
    // set the start position to the end of the file
    sp1.setToEnd();
    sp1.start(); // play the file in reverse
  }
```

Code Listing 4.1.3. Sampling_03.pde

```
// Sampling_03.pde


// this is a more complex sampler
// clicking somewhere on the window initiates sample playback
// moving the mouse controls the playback rate


import beads.*;


AudioContext ac;


SamplePlayer sp1;


// we can run both SamplePlayers through the same Gain
Gain sampleGain;
Glide gainValue;


Glide rateValue;


void setup()
{
  size(800, 600);

  ac = new AudioContext(); // create our AudioContext


  // whenever we load a file, we need to enclose
  // the code in a Try/Catch block
  // Try/Catch blocks will inform us if the file
  // can't be found
  try {
    // initialize the SamplePlayer
    sp1 = new SamplePlayer(ac,
```

```
                        new Sample(sketchPath("") +
                        "Drum_Loop_01.wav"));
    }
   catch(Exception e)
   {
     // if there is an error, show an error message
     println("Exception while attempting to load sample!");
     e.printStackTrace();
     exit();
   }

   // note that we want to play the sample multiple times
   sp1.setKillOnEnd(false);

   // initialize our rateValue Glide object
   rateValue = new Glide(ac, 1, 30);
   sp1.setRate(rateValue); // connect it to the SamplePlayer

   // as usual, we create a gain that will control the
   // volume of our sample player
   gainValue = new Glide(ac, 0.0, 30);
   sampleGain = new Gain(ac, 1, gainValue);
   sampleGain.addInput(sp1);


   // connect the Gain to the AudioContext
   ac.out.addInput(sampleGain);


   ac.start(); // begin audio processing

   background(0); // set the background to black
   stroke(255);
   // draw a line in the middle
   line(width/2, 0, width/2, height);
   text("Click to begin playback.", 100, 100);
   text("Move the mouse to control playback speed.",
         100, 120);
}


// although we're not drawing to the screen, we need
// to have a draw function in order to wait for
// mousePressed events
void draw()
{
   float halfWidth = width / 2.0;


   // set the gain based on mouse position along the Y-axis
   gainValue.setValue((float)mouseY / (float)height);
   // set the rate based on mouse position along the X-axis
   rateValue.setValue(((float)mouseX - halfWidth)/halfWidth);
}


// this routine is called whenever a mouse button is
// pressed on the Processing sketch
```

```
void mousePressed()
{
  // if the left mouse button is clicked, then play
  // the sound
  if( mouseX > width / 2.0 )
  {
    // set the start position to the beginning
    sp1.setPosition(000);
    sp1.start(); // play the audio file
  }
  // if the right mouse button is clicked, then play the
  // bass drum sample backwards
  else
  {
    // set the start position to the end of the file
    sp1.setToEnd();
    // play in reverse (rate set in the draw routine)
    sp1.start();
  }
}
```

4.2.More Mouse Input

In this section we're going to explore some of the additional data that can be extracted from mouse movement. Not only can we extract position data from the mouse cursor, but we can also get speed and average speed. This allows us to map more musical parameters to the mouse than we could if we used only the cursor position.

This section is included in this book as a demonstration of how an artist might map their Processing sketch to musical parameters. All of this information might be similarly extracted from objects within a sketch, or other parameters might be extracted and mapped from some other source of information.

4.2.1. Getting Mouse Speed and Mean Mouse Speed (Mouse_01)

This program demonstrates simple mouse data extraction without reference to sound or Beads. The draw routine simply maintains a set of mouse-related variables then prints their values on screen.

Code Listing 4.2.1. Mouse_01.pde

```
// Mouse_01.pde


// this short script just shows how we can extract
// information from a series of coordinates, in this case,
// the location of the mouse
```

```
// variables that will hold various mouse parameters
float xChange = 0;
float yChange = 0;
float lastMouseX = 0;
float lastMouseY = 0;
float meanXChange = 0.0;
float meanYChange = 0.0;


void setup()
{
  // create a decent-sized window, so that the mouse has
  // room to move
  size(800, 600);
}


void draw()
{
  background(0); // fill the background with black


  // calculate how much the mouse has moved
  xChange = lastMouseX - mouseX;
  yChange = lastMouseY - mouseY;

  // calculate the average speed of the mouse
  meanXChange = floor((0.5 * meanXChange) + (0.5 * xChange));
  meanYChange = floor((0.5 * meanYChange) + (0.5 * yChange));

  // store the current mouse coordinates for use in the
  // next round
  lastMouseX = mouseX;
  lastMouseY = mouseY;

  // show the mouse parameters on screen
  text("MouseX: " + mouseX, 100, 100);
  text("Change in X: " + xChange, 100, 120);
  text("Avg. Change in X: " + meanXChange, 100, 140);
  text("MouseY: " + mouseY, 100, 160);
  text("Change in Y: " + yChange, 100, 180);
  text("Avg. Change in Y: " + meanYChange, 100, 200);
}
```

4.2.2. Complex Mouse-Driven Multi-Sampler (Sampler_Interface_01)

WARNING: This is a very complex example. It builds on all the previous sampler examples and the all the previous mouse examples. This example also uses arrays of Beads. If you struggle to understand what is in this example, then see the previous examples and make sure you understand arrays as they are implemented in Processing. For more information on arrays, see http://processing.org/reference/Array.html.

In this example, we build an expressive multi-sampler. This multi-sampler loads all the samples in the samples subfolder of the sketch directory, then it plays them back, mapping playback parameters to mouse data. Each sample is loaded into a slice of area along the x-axis. Movement within that range has a possibility to trigger the related sound, as well as randomly triggering other sounds. If the mouse is moving downward, then the sample is played forward. If the mouse is moving up, then the sample is played backward. Further, the sample is pitch-shifted based on the speed of cursor movement.

There's a lot going on in this program, so it's important that you read through it line-by-line and try to understand what each line is accomplishing. We will break it down only briefly here.

The first major task undertaken by the setup routine is to discover the files in the samples folder, then load them into the sourceFile array. After the filenames are stored, we initialize our arrays of Beads, then initialize our objects that create a delay effect (these unit generators will be covered in the next chapter). Then the program loops through the array of filenames and attempts to load each sample. If the samples are loaded properly, then the SamplePlayer is created and the other unit generators associated with the SamplePlayer are initialized. Finally, the new unit generators are connected to the AudioContext.

```
// enclose the file-loading in a try-catch block
try {
  // run through each file
  for( count = 0; count < numSamples; count++ )
  {
    // create the SamplePlayer that will run this
    // particular file
    sp[count] = new SamplePlayer(ac,
                    new Sample(sketchPath("") +
                    "samples/" +
                    sourceFile[count]));
    sp[count].setKillOnEnd(false);


    // these unit generators will control aspects of the
    // sample player
    gainValue[count] = new Glide(ac, 0.0);
    gainValue[count].setGlideTime(20);
    g[count] = new Gain(ac, 1, gainValue[count]);
    rateValue[count] = new Glide(ac, 1);
    rateValue[count].setGlideTime(20);
    pitchValue[count] = new Glide(ac, 1);
    pitchValue[count].setGlideTime(20);


    sp[count].setRate(rateValue[count]);
    sp[count].setPitch(pitchValue[count]);
    g[count].addInput(sp[count]);
```

```
      // finally, connect this chain to the delay and to
      // the main out
      delayIn.addInput(g[count]);
      ac.out.addInput(g[count]);
    }
  }
  // if there is an error while loading the samples
  catch(Exception e)
  {
    // show that error in the space underneath the
    // processing code
    println("Exception while attempting to load sample!");
    e.printStackTrace();
    exit();
  }
```

After that, the program is started. As the draw routine is called repeatedly, mouse parameters are stored and used to trigger sample playback. Sample playback is handled by the triggerSample subroutine. This function triggers a sample either forward or backward, within the given pitch range, using the specified gain.

```
// trigger a sample
void triggerSample(int index,
                   boolean reverse,
                   float newGain,
                   float pitchRange)
{
  if( index >= 0 && index < numSamples )
  {
    gainValue[index].setValue(newGain); // set the gain value
    pitchValue[index].setValue(random(1.0-pitchRange,
                               1.0+pitchRange));

    // if we should play the sample in reverse
    if( reverse )
    {
      if( !sp[index].inLoop() )
      {
        rateValue[index].setValue(-1.0);
        sp[index].setToEnd();
      }
    }
    else // if we should play the sample forwards
    {
      if( !sp[index].inLoop() )
      {
        rateValue[index].setValue(1.0);
        sp[index].setToLoopStart();
      }
    }
```

```
    sp[index].start();
  }
}
```

Code Listing 4.2.2. Sampler_Interface_01.pde

```
// Sampler_Interface_01.pde
// This is a complex, mouse-driven sampler
// make sure that you understand the examples in Sampling_01,
// Sampling_02 and Sampling_03 before trying to tackle this

// import the java File library
// this will be used to locate the audio files that will be
// loaded into our sampler
import java.io.File;

import beads.*; // import the beads library

AudioContext ac; // declare our parent AudioContext as usual

// these variables store mouse position and change in mouse
// position along each axis
int xChange = 0;
int yChange = 0;
int lastMouseX = 0;
int lastMouseY = 0;

int numSamples = 0; // how many samples are being loaded?
// how much space will a sample take on screen? how wide will
// be the invisible trigger area?
int sampleWidth = 0;
// an array that will contain our sample filenames
String sourceFile[];
Gain g[]; // an array of Gains
Glide gainValue[];
Glide rateValue[];
Glide pitchValue[];
SamplePlayer sp[]; // an array of SamplePlayer

// these objects allow us to add a delay effect
TapIn delayIn;
TapOut delayOut;
Gain delayGain;

void setup()
{
  // create a reasonably-sized playing field for our sampler
  size(800, 600);

  ac = new AudioContext(); // initialize our AudioContext

  // this block of code counts the number of samples in
  // the /samples subfolder
  File folder = new File(sketchPath("") + "samples/");
  File[] listOfFiles = folder.listFiles();
  for (int i = 0; i < listOfFiles.length; i++)
```

```
{
    if (listOfFiles[i].isFile())
    {
      if( listOfFiles[i].getName().endsWith(".wav") )
      {
        numSamples++;
      }
    }
}

// if no samples are found, then end
if( numSamples <= 0 )
{
  println("no samples found in " + sketchPath("") +
                                                "samples/");
  println("exiting...");
  exit();
}
// how many pixels along the x-axis will each
// sample occupy?
sampleWidth = (int)(this.getWidth() / (float)numSamples);

// this block of code reads and stores the filename for
// each sample
sourceFile = new String[numSamples];
int count = 0;
for (int i = 0; i < listOfFiles.length; i++)
{
    if (listOfFiles[i].isFile())
    {
      if( listOfFiles[i].getName().endsWith(".wav") )
      {
        sourceFile[count] = listOfFiles[i].getName();
        count++;
      }
    }
}

// set the size of our arrays of unit generators in order
// to accomodate the number of samples that will be loaded
g = new Gain[numSamples];
gainValue = new Glide[numSamples];
rateValue = new Glide[numSamples];
pitchValue = new Glide[numSamples];
sp = new SamplePlayer[numSamples];

// set up our delay - this is just for taste, to fill out
// the texture
delayIn = new TapIn(ac, 2000);
delayOut = new TapOut(ac, delayIn, 200.0);
delayGain = new Gain(ac, 1, 0.15);
delayGain.addInput(delayOut);

// connect the delay to the master output
ac.out.addInput(delayGain);

// enclose the file-loading in a try-catch block
```

```
  try {
    // run through each file
    for( count = 0; count < numSamples; count++ )
    {
      // print a message to show which file we are loading
      println("loading " + sketchPath("") +
                  "samples/" + sourceFile[count]);

      // create the SamplePlayer that will run this
      // particular file
      sp[count] = new SamplePlayer(ac,
                              new Sample(sketchPath("") +
                              "samples/" +
                              sourceFile[count]));
      sp[count].setKillOnEnd(false);

      // these unit generators will control aspects of the
      // sample player
      gainValue[count] = new Glide(ac, 0.0);
      gainValue[count].setGlideTime(20);
      g[count] = new Gain(ac, 1, gainValue[count]);
      rateValue[count] = new Glide(ac, 1);
      rateValue[count].setGlideTime(20);
      pitchValue[count] = new Glide(ac, 1);
      pitchValue[count].setGlideTime(20);

      sp[count].setRate(rateValue[count]);
      sp[count].setPitch(pitchValue[count]);
      g[count].addInput(sp[count]);

      // finally, connect this chain to the delay and to the
      // main out
      delayIn.addInput(g[count]);
      ac.out.addInput(g[count]);
    }
  }
  // if there is an error while loading the samples
  catch(Exception e)
  {
    // show that error in the space underneath the
    // processing code
    println("Exception while attempting to load sample!");
    e.printStackTrace();
    exit();
  }

  ac.start(); // begin audio processing

  background(0); // set the background to black
  text("Move the mouse quickly to trigger playback.",
          100, 100);
  text("Faster movement triggers more and louder sounds.",
          100, 120);
}

// the main draw function
void draw()
```

```
{
  background(0);

  // calculate the mouse speed and location
  xChange = abs(lastMouseX - mouseX);
  yChange = lastMouseY - mouseY;
  lastMouseX = mouseX;
  lastMouseY = mouseY;

  // calculate the gain of newly triggered samples
  float newGain = (abs(yChange) + xChange) / 2.0;
  newGain /= this.getWidth();
  if( newGain > 1.0 ) newGain = 1.0;

  // calculate the pitch range
  float pitchRange = yChange / 200.0;

  // should we trigger the sample that the mouse is over?
  if( newGain > 0.09 )
  {
    // get the index of the sample that is coordinated with
    // the mouse location
    int currentSampleIndex = (int)(mouseX / sampleWidth);
    if( currentSampleIndex < 0 ) currentSampleIndex = 0;
    else if( currentSampleIndex >= numSamples )
                              currentSampleIndex = numSamples;

    // trigger that sample
    // if the mouse is moving upwards, then play it in
    // reverse
    triggerSample(currentSampleIndex,
                  (boolean)(yChange < 0),
                  newGain,
                  pitchRange);
  }

  // randomly trigger other samples, based loosely on the
  // mouse speed
  // loop through each sample
  for( int currentSample = 0;
            currentSample < numSamples;
            currentSample++ )
  {
    // if a random number is less than the current gain
    if( random(1.0) < (newGain / 2.0) )
    {
      // trigger that sample
      triggerSample(currentSample,
                (boolean)(yChange < 0 && random(1.0) < 0.33),
                newGain,
                pitchRange);
    }
  }
}

// trigger a sample
void triggerSample(int index,
```

```
                    boolean reverse,
                    float newGain,
                    float pitchRange)
{
  if( index >= 0 && index < numSamples )
  {
    // show a message that indicates which sample we are
    // triggering
    println("triggering sample " + index);

    gainValue[index].setValue(newGain); // set the gain value
    // and set the pitch value (which is really just another
    // rate controller)
    pitchValue[index].setValue(random(1.0-pitchRange,
                                1.0+pitchRange));

    // if we should play the sample in reverse
    if( reverse )
    {
      if( !sp[index].inLoop() )
      {
        rateValue[index].setValue(-1.0);
        sp[index].setToEnd();
      }
    }
    else // if we should play the sample forwards
    {
      if( !sp[index].inLoop() )
      {
        rateValue[index].setValue(1.0);
        sp[index].setToLoopStart();
      }
    }

    sp[index].start();
  }
}
```

4.3. Granular Synthesis

Granular synthesis is a technique whereby sounds are created from many small *grains* of sound. A good analogy is a dump truck unloading a load of gravel. As each individual pebble hits the ground it makes a sound, but we hear the gravel as one large sound complex. Since the early 1980s, sound artists have been exploring this popular sound-generating paradigm, and Beads provides a wonderful unit generator for easily creating granular synthesis instruments.

The GranularSamplePlayer object allows a programmer to load an audio file, then play it back with automated granulation based on a number of parameters. The parameters on the GranularSamplePlayer object pertain to how the grains are selected and pushed out. These parameters include playback rate, pitch, grain size, grain interval, grain randomness and position. Playback rate, pitch and grain size are pretty straightforward. Grain interval interval sets the time between grains. Grain randomness sets the jitter that will apply to each parameter; it helps the granulation sound less mechanical. Position sets the point in the file from which grains should be drawn, and it assumes that playback rate is zero.

4.3.1. Using GranularSamplePlayer (Granular_01)

In this example we're going to set up a very basic granular synthesizer that is controlled by the mouse cursor. The position of the mouse along the x-axis is mapped to the position in the source audio file from which grains will be extracted. The position of the mouse along the y-axis is mapped to grain duration.

The first thing to notice about this program is that we declare a lot of Glide objects. There are a lot of parameters to the granulation process, and we want to make sure that we have a hook into all of them.

Next, notice that setting up the GranularSamplePlayer is very similar to how we set up the SamplePlayer object. We enclose the file operations within a try/catch, then we call the constructor with two parameters, the AudioContext and the sample that we will granulate.

```
try {
  // load the audio file which will be used in granulation
  sourceSample = new Sample(sketchPath("") + sourceFile);
}
// catch any errors that occur in file loading
catch(Exception e) {
  println("Exception while attempting to load sample!");
  e.printStackTrace();
  exit();
}
// store the sample length
sampleLength = sourceSample.getLength();


// initialize our GranularSamplePlayer
gsp = new GranularSamplePlayer(ac, sourceSample);
```

Next we initialize all of our glide objects, and connect them to the proper parameters.

```
// connect all of our Glide objects to the previously created
GranularSamplePlayer
```

```
gsp.setRandomness(randomnessValue);
gsp.setGrainInterval(intervalValue);
gsp.setGrainSize(grainSizeValue);
gsp.setPosition(positionValue);
gsp.setPitch(pitchValue);
```

We start the granulation by calling the start function. The granulation parameters are controlled within the draw routine. In this case, we perform some math on the granulation parameters. The position value calculation makes sure to output a index number within the audio file. The grain size calculation is simply for personal taste; try playing with it and listening to how the resulting granulation changes!

```
// grain size can be set by moving the mouse along the Y-axis
grainSizeValue.setValue((float)mouseY / 5);


// and the X-axis is used to control the position in the wave
// file that is being granulated
positionValue.setValue((float)(mouseX /
             (float)this.getWidth()) * (sampleLength - 400));
```

Code Listing 4.3.1. Granular_01.pde

```
// Granular_01.pde


// In this granular synthesis demonstration, the mouse
// controls the position and grain size parameters. The
// position within the source audio file is controlled by
// the X-axis. The grain size is controlled by the Y-axis.


import beads.*; // import the beads library


AudioContext ac; // declare our parent AudioContext


// what file will be granulated?
String sourceFile = "OrchTuning01.wav";


Gain masterGain; // our usual master gain


GranularSamplePlayer gsp; // our GranularSamplePlayer object


// these unit generators will be connected to various
granulation parameters
Glide gainValue;
Glide randomnessValue;
Glide grainSizeValue;
```

```
Glide positionValue;
Glide intervalValue;
Glide pitchValue;


// this object will hold the audio data that will be
// granulated
Sample sourceSample = null;

// this float will hold the length of the audio data, so that
// we don't go out of bounds when setting the granulation
// position
float sampleLength = 0;


void setup()
{
  size(800, 600); // set a reasonable window size

  ac = new AudioContext(); // initialize our AudioContext


  // again, we encapsulate the file-loading in a try-catch
  // block, just in case there is an error with file access
  try {
    // load the audio file which will be used in granulation
    sourceSample = new Sample(sketchPath("") + sourceFile);
  }
  // catch any errors that occur in file loading
  catch(Exception e) {
    println("Exception while attempting to load sample!");
    e.printStackTrace();
    exit();
  }
  // store the sample length - this will be used when
  // determining where in the file we want to position our
  // granulation pointer
  sampleLength = sourceSample.getLength();


  // set up our master gain
  gainValue = new Glide(ac, 0.5, 100);
  masterGain = new Gain(ac, 1, gainValue);


  // initialize our GranularSamplePlayer
  gsp = new GranularSamplePlayer(ac, sourceSample);


  // these ugens will control aspects of the granular sample
  // player
  // remember the arguments on the Glide constructor
  // (AudioContext, Initial Value, Glide Time)
  randomnessValue = new Glide(ac, 80, 10);
  intervalValue = new Glide(ac, 100, 100);
  grainSizeValue = new Glide(ac, 100, 50);
  positionValue = new Glide(ac, 50000, 30);
  pitchValue = new Glide(ac, 1, 20);
```

```
  // connect all of our Glide objects to the previously
  // created GranularSamplePlayer
  gsp.setRandomness(randomnessValue);
  gsp.setGrainInterval(intervalValue);
  gsp.setGrainSize(grainSizeValue);
  gsp.setPosition(positionValue);
  gsp.setPitch(pitchValue);


  // connect our GranularSamplePlayer to the master gain
  masterGain.addInput(gsp);
  gsp.start(); // start the granular sample player


  // connect the master gain to the AudioContext's master
  // output
  ac.out.addInput(masterGain);
  ac.start(); // begin audio processing

  background(0); // set the background to black
  text("Move the mouse to control granular synthesis.",
                      100, 120); // tell the user what to do!
}


// the main draw function
void draw()
{
  background(0, 0, 0);


  // grain size can be set by moving the mouse along the Y-
  // axis
  grainSizeValue.setValue((float)mouseY / 5);


  // The X-axis is used to control the position in the
  // wave file that is being granulated.
  // The equation used here looks complex, but it really
  // isn't.
  // All we're doing is translating on-screen position into
  // position in the audio file.
  // this: (float)mouseX / (float)this.getWidth() calculates
  // a 0.0 to 1.0 value for position along the x-axis
  // then we multiply it by sampleLength (minus a little
  // buffer for safety) to get the position in the audio file
  positionValue.setValue((float)((float)mouseX /
             (float)this.getWidth()) * (sampleLength - 400));
}
```

4.3.2. Using GranularSamplePlayer Parameters (Granular_02)

The second granular synthesis example is very similar to the first, with just a few alterations. First, we set up another GranularSamplePlayer to make a thicker, more densely grainy texture. Then we map cursor speed to overall gain. So mouse movement essentially triggers granulation. Finally, we control each granulator slightly differently. Grain size is still mapped to the y-axis and source file position is still mapped to the x-axis, but each granulator uses slightly different numbers than the other. Again, this allows us to create a more complex texture. Finally, we supply random numbers to some other granulation parameters, in order to make the sound a little bit more unpredictable.

Code Listing 4.3.2. Granular_02.pde

```
// Granular_02.pde

// This granular synthesis program is similar to Granular_01,
// except we add a second granulator and a delay to fill out
// the texture. This program is very complex, but it can be
// used to create  an assortment of interesting sounds and
// textures.

import beads.*; // import the beads library

AudioContext ac; // declare our parent AudioContext

// Change this line to try the granulator on another sound
// file. This granulator can be used to create a wide variety
// of sounds, depending on the input file.
String sourceFile = "OrchTuning01.wav";

Gain masterGain; // our usual master gain
Glide masterGainValue;

// these unit generators will be connected to various
granulation parameters

// our first GranularSamplePlayer object
GranularSamplePlayer gsp1;
Gain g1;
Glide gainValue1;
Glide randomnessValue1;
Glide grainSizeValue1;
Glide positionValue1;
Glide intervalValue1;
Glide pitchValue1;

// repeat the same unit generators for our second granulator
GranularSamplePlayer gsp2;
Gain g2;
Glide gainValue2;
Glide randomnessValue2;
```

```
Glide grainSizeValue2;
Glide positionValue2;
Glide intervalValue2;
Glide pitchValue2;

// we add a delay unit just to give the program a fuller
sound
TapIn delayIn;
TapOut delayOut;
Gain delayGain;

// This object will hold the audio data that will be
// granulated.
Sample sourceSample = null;

// This float will hold the length of the audio data, so that
// we don't go out of bounds when setting the granulation
// position.
float sampleLength = 0;

// these variables will be used to store properties of the
// mouse cursor
int xChange = 0;
int yChange = 0;
int lastMouseX = 0;
int lastMouseY = 0;

void setup()
{
  // set a reasonable window size
  size(800, 600);

  // initialize our AudioContext
  ac = new AudioContext();

  // again, we encapsulate the file-loading in a try-catch
  // block, just in case there is an error with file access
  try {
    // load the audio file which will be used in granulation
    sourceSample = new Sample(sketchPath("") + sourceFile);
  }
  // catch any errors that occur in file loading
  catch(Exception e) {
    println("Exception while attempting to load sample!");
    e.printStackTrace();
    exit();
  }
  // store the sample length - this will be used when
  // determining where in the file we want to position our
  // granulation pointer
  sampleLength = sourceSample.getLength();

  // set up our master gain
  masterGainValue = new Glide(ac, 0.9, 100);
  masterGain = new Gain(ac, 1, masterGainValue);
```

```
// these ugens will control aspects of the granular sample
// player
gsp1 = new GranularSamplePlayer(ac, sourceSample);
randomnessValue1 = new Glide(ac, 80, 10);
intervalValue1 = new Glide(ac, 100, 100);
grainSizeValue1 = new Glide(ac, 100, 50);
positionValue1 = new Glide(ac, 50000, 30);
pitchValue1 = new Glide(ac, 1, 20);
gainValue1 = new Glide(ac, 0.0, 30);
g1 = new Gain(ac, 1, gainValue1);
g1.addInput(gsp1);

// connect all of our Glide objects to the previously
// created GranularSamplePlayer
gsp1.setRandomness(randomnessValue1);
gsp1.setGrainInterval(intervalValue1);
gsp1.setGrainSize(grainSizeValue1);
gsp1.setPosition(positionValue1);
gsp1.setPitch(pitchValue1);

// we will repeat the same Unit Generators for the second
// GranularSamplePlayer
gsp2 = new GranularSamplePlayer(ac, sourceSample);
randomnessValue2 = new Glide(ac, 140, 10);
intervalValue2 = new Glide(ac, 60, 50);
grainSizeValue2 = new Glide(ac, 100, 20);
positionValue2 = new Glide(ac, 50000, 30);
pitchValue2 = new Glide(ac, 1, 50);
gainValue2 = new Glide(ac, 0.0, 30);
g2 = new Gain(ac, 1, gainValue2);
g2.addInput(gsp2);

// connect all of our Glide objects to the previously
// created GranularSamplePlayer
gsp2.setRandomness(randomnessValue2);
gsp2.setGrainInterval(intervalValue2);
gsp2.setGrainSize(grainSizeValue2);
gsp2.setPosition(positionValue2);
gsp2.setPitch(pitchValue2);

// Set up our delay unit (this will be covered more
// thoroughly in a later example).
// The TapIn object is the start of the delay
delayIn = new TapIn(ac, 2000);
// the TapOut object is the delay output object
delayOut = new TapOut(ac, delayIn, 200.0);
// connect the first GranularSamplePlayer to the delay
delayIn.addInput(g1);
// connect the second GranularSamplePlayer to the delay
delayIn.addInput(g2);
// set the volume of the delay effect
delayGain = new Gain(ac, 1, 0.15);
// connect the delay output to the gain input
delayGain.addInput(delayOut);

// connect the first GranularSamplePlayer to the master
// gain
```

```
  masterGain.addInput(g1);
  // connect the second GranularSamplePlayer to the master
  // gain
  masterGain.addInput(g2);
  // connect our delay effect to the master gain
  masterGain.addInput(delayGain);
  gsp1.start(); // start the first granular sample player
  gsp2.start(); // start the second granular sample player

  // connect the master gain to the AudioContext's master
  // output
  ac.out.addInput(masterGain);
  ac.start(); // begin audio processing

  background(0); // set the background to black
  text("Move the mouse to control granular synthesis.",
                    100, 120); // tell the user what to do!
}

// the main draw function
void draw()
{

  // get the location and speed of the mouse cursor
  xChange = abs(lastMouseX - mouseX);
  yChange = abs(lastMouseY - mouseY);
  lastMouseX = mouseX;
  lastMouseY = mouseY;

  float newGain = (xChange + yChange) / 3.0;
  if( newGain > 1.0 ) newGain = 1.0;

  gainValue1.setValue(newGain);
  float pitchRange = yChange / 200.0;
  pitchValue1.setValue(random(1.0-pitchRange,
                      1.0+pitchRange));
  // set randomness to a nice random level
  randomnessValue1.setValue(random(100) + 1.0);
  // set the time interval value according to how much the
  // mouse is moving horizontally
  intervalValue1.setValue(random(random(200, 1000) /
                                          (xChange+1)) );
  // grain size can be set by moving the mouse along the Y-
  // axis
  grainSizeValue1.setValue((float)mouseY / 10);
  // and the X-axis is used to control the position in the
  // wave file that is being granulated
  positionValue1.setValue((float)((float)mouseX /
             (float)this.getWidth()) * (sampleLength - 400));

  // set up the same relationships as with the first
  // GranularSamplePlayer, but use slightly different numbers
  gainValue1.setValue(newGain * 0.8);
  pitchRange *= 3.0; // use a slightly larger pitch range
  pitchValue2.setValue(random(1.0-pitchRange,
                                          1.0+pitchRange));
  randomnessValue2.setValue(random(150) + 1.0);
```

```
  // use a slightly longer interval value
  intervalValue2.setValue(random(random(500, 1000) /
     (xChange+1)) );
  grainSizeValue2.setValue((float)mouseY / 5);
  positionValue2.setValue((float)((float)mouseX /
             (float)this.getWidth()) * (sampleLength - 400));

}
```

5. Effects

In this chapter, we're going to look at some of the audio effects that are already implemented in Beads. At the time of this writing, the effects aren't that deep, but I expect that to change soon. In this chapter we will look at delay objects, filters, reverb and a couple others.

5.1. Delay

Delay is a digital echo. An audio signal is stored in memory for a brief duration, then sent to the output. In Beads, delay is implemented using two unit generators: TapIn and TapOut.

5.1.1. Basic Delay (Delay_01)

In this example, we're going to expand on the Frequency_Modulation_03 example by adding a short delay. Delays are very easy to setup. First, we declare the TapIn object and give it a maximum delay time in milliseconds. This is the duration in milliseconds of the audio buffer within the TapIn object. Then we connect our synthesizer output to the TapIn input.

```
delayIn = new TapIn(ac, 2000);
delayIn.addInput(synthGain);
```

Next we set up the TapOut. The TapOut object simply waits for delayed audio from the TapIn. The TapOut constructor has three parameters. First we supply the parent AudioContext, then we supply the name of the TapIn that this TapOut is connected to, then we give the actual delay duration in milliseconds. The delay can be any duration less than the maximum duration supplied to the TapIn object. After that, simply connect the TapOut output to your output objects, and the delay is ready to go.

```
delayOut = new TapOut(ac, delayIn, 500.0);
delayGain = new Gain(ac, 1, 0.50);
delayGain.addInput(delayOut);
```

Code Listing 5.1.1. Delay_01.pde

```
// Delay_01.pde
// This is an extension of Frequency_Modulation_03.pde
```

```
// this example creates a simple FM synthesizer and adds a
// 400ms delay


import beads.*; // import the beads library
AudioContext ac; // create our AudioContext


// declare our unit generators
WavePlayer modulator;
Glide modulatorFrequency;
WavePlayer carrier;


// our envelope and gain objects
Envelope gainEnvelope;
Gain synthGain;


// our delay objects
TapIn delayIn;
TapOut delayOut;
Gain delayGain;


void setup()
{
  size(400, 300);


  // initialize our AudioContext
  ac = new AudioContext();

  // create the modulator, this WavePlayer will control the
  // frequency of the carrier
  modulatorFrequency = new Glide(ac, 20, 30);
  modulator = new WavePlayer(ac,
                             modulatorFrequency,
                             Buffer.SINE);


  // create a custom frequency modulation function
  Function frequencyModulation = new Function(modulator)
  {
    public float calculate() {
      // return x[0], scaled into an appropriate frequency
      // range
      return (x[0] * 100.0) + mouseY;
    }
  };


  // create a second WavePlayer, control the frequency with
  // the function created above
  carrier = new WavePlayer(ac,
                           frequencyModulation,
                           Buffer.SINE);
```

```
  // create the envelope object that will control the gain
  gainEnvelope = new Envelope(ac, 0.0);
  // create a Gain object, connect it to the gain envelope
  synthGain = new Gain(ac, 1, gainEnvelope);
  // connect the carrier to the Gain input
  synthGain.addInput(carrier);


  // set up our delay
  // create the delay input - the second parameter sets the
  // maximum delay time in milliseconds
  delayIn = new TapIn(ac, 2000);
  // connect the synthesizer to the delay
  delayIn.addInput(synthGain);
  // create the delay output - the final parameter is the
  // length of the initial delay time in milliseconds
  delayOut = new TapOut(ac, delayIn, 500.0);
  // the gain for our delay
  delayGain = new Gain(ac, 1, 0.50);
  // connect the delay output to the gain
  delayGain.addInput(delayOut);

  // To feed the delay back into itself, simply uncomment
  // this line.
  //delayIn.addInput(delayGain);

  // connect the Gain output to the AudioContext
  ac.out.addInput(synthGain);
  // connect the delay output to the AudioContext
  ac.out.addInput(delayGain);
  // start audio processing
  ac.start();

  // print the directions on the screen
  background(0);
  text("Click me to demonstrate delay!", 100, 100);
}


void draw()
{
  // set the modulator frequency
  modulatorFrequency.setValue(mouseX);
}


// this routine is triggered whenever a mouse button is
// pressed
void mousePressed()
{
  // when the mouse button is pressed, at a 50ms attack
  // segment to the envelope
  // and a 300 ms decay segment to the envelope
  gainEnvelope.addSegment(0.7, 50);
  gainEnvelope.addSegment(0.0, 300);
}
```

5.1.2. Controlling Delay

At the time of this writing, variable delay seems to be bugged. This section will be updated as soon as there is a fix for variable-length delay.

If it were working, the UGen that controls the variable delay would be inserted into the TapOut constructor. This might be an LFO, or an envelope, or a custom function.

```
delayOut = new TapOut(ac, delayIn, delayEnvelope);
```

5.2. Filters

Filter effects are another class of audio effects that is both familiar and well-implemented in Beads. Technically, filtering is boosting or attenuating frequencies in a sound. Commonly, sound artists talk about *low pass filters* and *high pass filters*. Low pass filters allow the low frequencies to pass while reducing or eliminating some of the higher frequencies. High pass filters do the opposite. You can easily simulate the sound of a low pass filter by cupping your hands over your mouth as you speak. Other filter types include band pass, notch, comb and allpass. Each filter has a different quality, and there are many books that cover just the topic of filters. For now, it's enough to know that filters modify the frequency spectrum of a sound.

5.2.1. Low Pass Filter (Filter_01)

IMPORTANT: This program will only work if the sketch directory contains an audio file called Drum_Loop_01.wav.

In the first filter example, we're going to apply a low pass filter to a drum loop using the OnePoleFilter object. You can left-click on the window to hear the filtered drums, and right-click to hear the original loop.

Setting up the low pass filter is literally three lines of code. Call the OnePoleFilter constructor, passing it the AudioContext and the cutoff frequency. Then connect the input and the output and you're ready to rock.

```
// our new filter with a cutoff frequency of 200Hz
filter1 = new OnePoleFilter(ac, 200.0);
// connect the SamplePlayer to the filter
filter1.addInput(sp);
…
// connect the filter to the gain
g.addInput(filter1);
```

Code Listing 5.2.1. Filter_01.pde

```
// Filter_01.pde

import beads.*;

AudioContext ac;

// this will hold the path to our audio file
String sourceFile;
// the SamplePlayer class will be used to play the audio file
SamplePlayer sp;

// standard gain objects
Gain g;
Glide gainValue;

OnePoleFilter filter1; // this is our filter unit generator

void setup()
{
  size(800, 600);

  ac = new AudioContext(); // create our AudioContext

  sourceFile = sketchPath("") + "Drum_Loop_01.wav";

  try {
    // initialize our SamplePlayer,
    sp = new SamplePlayer(ac, new Sample(sourceFile));
   }
  catch(Exception e)
  {
    // if there is an error, show an error message
    println("Exception while attempting to load sample!");
    e.printStackTrace();
    exit();
  }

  // we would like to play the sample multiple times, so we
  // set KillOnEnd to false
  sp.setKillOnEnd(false);

  // set up our new filter with a cutoff frequency of 200Hz
  filter1 = new OnePoleFilter(ac, 200.0);
  // connect the SamplePlayer to the filter
  filter1.addInput(sp);
```

```
  // as usual, we create a gain that will control the volume
  // of our sample player
  gainValue = new Glide(ac, 0.0, 20);
  g = new Gain(ac, 1, gainValue);
  // connect the filter to the gain
  g.addInput(filter1);


  // connect the Gain to the AudioContext
  ac.out.addInput(g);
  // begin audio processing
  ac.start();

  background(0);
  text("Left click to hear a low pass filter in effect.",
                                                    50, 50);
  text("Right click to hear the original loop.", 50, 70);
}


// Although we're not drawing to the screen, we need to have
// a draw function in order to wait for mousePressed events
void draw(){}


// this routine is called whenever a mouse button is pressed
// on the Processing sketch
void mousePressed()
{
  if( mouseButton == LEFT )
  {
    // set the filter frequency to cutoff at 200Hz
    filter1.setFrequency(200.0);
    // set the gain based on mouse position
    gainValue.setValue((float)mouseX/(float)width);
    // move the playback pointer to the first loop point
    sp.setToLoopStart();
    // play the audio file
    sp.start();
  }
  // if the user right-clicks, then play the sound without a
  // filter
  else
  {
    // set the filter frequency to cutoff at 20kHz -> the top
    // of human hearing
    filter1.setFrequency(20000.0);
    // set the gain based on mouse position
    gainValue.setValue((float)mouseX/(float)width);
     // move the playback pointer to the first loop point
    sp.setToLoopStart();
    // play the audio file
    sp.start();
  }
}
```

5.2.2. Low-Pass Resonant Filter with Envelope (Filter_02)

In this example we're going to attach an envelope to the filter cutoff frequency. This example is going to use a resonant low pass filter implemented in the LPRezFilter object, but the envelope code could easily be applied to any of the Beads filter objects. This example builds on the earlier synthesis examples.

Setting up the filter and envelope is similar to how we have set up filters and envelopes in previous examples. The LPRezFilter constructor takes three arguments, the AudioContext, the cutoff frequency and the resonance (0.0 - 1.0). Since we want to control the cutoff frequency with an envelope, we insert the name of the envelope object where we would normally indicate a cutoff value.

```
filterCutoffEnvelope = new Envelope(ac, 00.0);
lowPassFilter = new LPRezFilter(ac,
                                filterCutoffEnvelope,
                                0.97);
lowPassFilter.addInput(carrier);
```

Then we connect the gain objects as usual, and the only other new addition to this example is the use of a frequency envelope in the mousePressed function. In this example we sweep the cutoff frequency from 0.0 to 800.0 over 1000 milliseconds. Then it returns to 0.0 over 1000 milliseconds.

```
filterCutoffEnvelope.addSegment(800.0, 1000);
filterCutoffEnvelope.addSegment(00.0, 1000);
```

Code Listing 5.2.2. Filter_02.pde

```
// Filter_02.pde


// This is an extension of Frequency_Modulation_03.pde
// this adds a low pass filter controlled by an envelope


import beads.*; // import the beads library
AudioContext ac; // create our AudioContext


// declare our FM Synthesis unit generators
WavePlayer modulator;
Glide modulatorFrequency;
WavePlayer carrier;
```

```
// our gain and gain envelope
Envelope gainEnvelope;
Gain synthGain;


// our filter and filter envelope
LPRezFilter lowPassFilter;
Envelope filterCutoffEnvelope;


void setup()
{
  size(400, 300);


  // initialize our AudioContext
  ac = new AudioContext();

  // create the modulator, this WavePlayer will control the
  // frequency of the carrier
  modulatorFrequency = new Glide(ac, 20, 30);
  modulator = new WavePlayer(ac, modulatorFrequency,
Buffer.SINE);


  // create a custom frequency modulation function
  Function frequencyModulation = new Function(modulator)
  {
    public float calculate() {
      // return x[0], scaled into an appropriate frequency
      // range
      return (x[0] * 500.0) + mouseY;
    }
  };


  // create a second WavePlayer, control the frequency with
  // the function created above
  carrier = new WavePlayer(ac,
                           frequencyModulation,
                           Buffer.SINE);


  // set up our low pass filter
  // create the envelope that will control the cutoff
  // frequency
  filterCutoffEnvelope = new Envelope(ac, 00.0);
  // create the LP Rez filter
  lowPassFilter = new LPRezFilter(ac,
                                  filterCutoffEnvelope,
                                  0.97);
  // connect the synthesizer to the filter
  lowPassFilter.addInput(carrier);

  // set up our gain envelope objects
  gainEnvelope = new Envelope(ac, 0.0);
  // create a Gain object, connect it to the gain envelope
  synthGain = new Gain(ac, 1, gainEnvelope);
  // connect the carrier to the Gain input
```

```
  synthGain.addInput(lowPassFilter);

  // connect the filter output to the AudioContext
  ac.out.addInput(synthGain);
  // start audio processing
  ac.start();

  background(0);
  text("Click me to demonstrate a filter sweep!", 100, 100);
}


void draw()
{
  // set the modulator frequency
  modulatorFrequency.setValue(mouseX);
}


void mousePressed()
{
  // add some points to the gain envelope
  gainEnvelope.addSegment(0.7, 500);
  gainEnvelope.addSegment(0.7, 1000);
  gainEnvelope.addSegment(0.0, 500);

  // add points to the filter envelope sweep the frequency up
  // to 500Hz, then back down to 0
  filterCutoffEnvelope.addSegment(800.0, 1000);
  filterCutoffEnvelope.addSegment(00.0, 1000);
}
```

5.2.3. Band-Pass Filter (Filter_03)

A band pass filter allows a range of frequencies to pass through, while attenuating all other frequencies. That range is referred to as the *band*, hence the name *band pass filter*. This example returns to playing the drum loop that we have used in previous examples. This time we run it through a band pass filter with the cutoff set to 5000Hz and the Q set to 0.5. Q is an interesting property of nearly all filters. Basically, Q specifies the bandwidth of the area effected by the filter. In the case of a band pass filter, a small Q value indicates that a lot of frequencies around the cutoff will be allowed to pass through. A higher Q would indicate that fewer frequencies should pass.

In this example, we use a versatile filter unit generator called BiquadFilter. This unit generator can implement many different filter equations, so we have to specify which type of filter will be applied by the object. We do that in the BiquadFilter constructor, as the parameter after the AudioContext.

```
filter1 = new BiquadFilter(ac,
```

```
                                BiquadFilter.BP_SKIRT,
                                5000.0f,
                                0.5f);
```

Other filter types can be found in the Beads documentation at http://www.beadsproject.net/doc/net/beadsproject/beads/ugens/BiquadFilter.html#setType(int)

Code Listing 5.2.3. Filter_03.pde

```
// Filter_03.pde
// In this example, we apply a band-pass filter to a drum
// loop.


import beads.*;


AudioContext ac;
// this will hold the path to our audio file
String sourceFile;
// the SamplePlayer class will be used to play the audio file
SamplePlayer sp;


// standard gain objects
Gain g;
Glide gainValue;


// this is our filter unit generator
BiquadFilter filter1;


void setup()
{
  size(800, 600);

  ac = new AudioContext(); // create our AudioContext


  sourceFile = sketchPath("") + "Drum_Loop_01.wav";


  try {
    // Initialize our SamplePlayer
    sp = new SamplePlayer(ac, new Sample(sourceFile));
   }
  catch(Exception e)
  {
    println("Exception while attempting to load sample!");
    e.printStackTrace();
    exit();
  }

  sp.setKillOnEnd(false);
```

```
  filter1 = new BiquadFilter(ac,
                             BiquadFilter.BP_SKIRT, 5000.0f,
                             0.5f);
  // connect the SamplePlayer to the filter
  filter1.addInput(sp);


  // as usual, we create a gain that will control the volume
  // of our sample player
  gainValue = new Glide(ac, 0.0, 20);
  g = new Gain(ac, 1, gainValue);
  g.addInput(filter1);


  // connect the Gain to the AudioContext
  ac.out.addInput(g);


  ac.start();

  // print the instructions on screen
  background(0);
  text("Click to hear a band pass filter with cutoff set to
                                              5000Hz.", 50, 50);
}


void draw(){}


void mousePressed()
{
    gainValue.setValue(0.9);
    sp.setToLoopStart();
    sp.start();
}
```

5.3. Other Effects

This section will look at some of the other audio effects objects provided by Beads.

5.3.1. Panner (Panner_01)

In this example we create a Panner object, then control it using a *Low Frequency Oscillator*. Panner objects allow a Beads user to place mono sounds within the stereo field. Low Frequency Oscillators, or LFOs, are just like the other oscillators we've created, but they oscillate at frequencies below 20Hz.

The Panner object takes a value between -1.0 and 1.0, then pans the incoming audio appropriately. A value of -1.0 indicates full left. A value of 1.0 indicates full right. Since sine waves oscillate between -1.0 and 1.0, we can easily use the WavePlayer object to control this parameter.

In this block of code, we create the LFO that will control the Panner, then we instantiate the Panner object, inserting the LFO where we might normally indicate a fixed pan value. Finally, we connect the drum loop, through the Gain object g, to the Panner object.

```
// initialize the LFO at a frequency of 0.33Hz
panLFO = new WavePlayer(ac, 0.33, Buffer.SINE);
p = new Panner(ac, panLFO);
p.addInput(g);
```

Code Listing 5.3.1. Panner_01.pde

```
// Panner_01.pde
// this example demonstrates how to use the Panner object
// this example extends Filter_01.pde


import beads.*;


AudioContext ac;


String sourceFile;
SamplePlayer sp;


// standard gain objects
Gain g;
Glide gainValue;


// our Panner will control the stereo placement of the sound
Panner p;
// a Low-Frequency-Oscillator for the panner
WavePlayer panLFO;


void setup()
{
  size(800, 600);

  ac = new AudioContext();


  sourceFile = sketchPath("") + "Drum_Loop_01.wav";


  try {
    sp = new SamplePlayer(ac, new Sample(sourceFile));
```

```
   }
  catch(Exception e)
  {
    println("Exception while attempting to load sample!");
    e.printStackTrace();
    exit();
  }
  sp.setKillOnEnd(false);


  // as usual, we create a gain that will control the volume
  // of our sample player
  gainValue = new Glide(ac, 0.0, 20);
  g = new Gain(ac, 1, gainValue);
  g.addInput(sp); // connect the filter to the gain


  // In this block of code, we create an LFO - a Low
  // Frequency Oscillator - and connect it to our panner.
  // A low frequency oscillator is just like any other
  // oscillator EXCEPT the frequency is subaudible, under
  // 20Hz.
  // In this case, the LFO controls pan position.
  // Initialize the LFO at a frequency of 0.33Hz.
  panLFO = new WavePlayer(ac, 0.33, Buffer.SINE);
  // initialize the panner. to set a constant pan position,
  // merely replace "panLFO" with a number between -1.0
  // (LEFT) and 1.0 (RIGHT)
  p = new Panner(ac, panLFO);
  p.addInput(g);


  // connect the Panner to the AudioContext
  ac.out.addInput(p);
  // begin audio processing
  ac.start();

  background(0); // draw a black background
  text("Click to hear a Panner object connected to an LFO.",
50, 50); // tell the user what to do
}


void draw(){}


void mousePressed()
{
    // set the gain based on mouse position and play the file
    gainValue.setValue((float)mouseX/(float)width);
    sp.setToLoopStart();
    sp.start();
}
```

5.3.2. Reverb (Reverb_01)

This example builds on the Filter_01 example by adding a thick reverb to the drum loop. Reverb is easy to hear in your own home, just go into your shower and clap. As the clap reverberates around your shower, it makes many quiet echoes that give an impression of the size and shape of the room. The same effect can be heard in a concert hall, or any space with large flat walls.

Fittingly, Beads provides the Reverb object to allow programmers to create reverb effects. The reverb object is easy enough to set up. We only need to set a few parameters then connect the input and output the way we do with nearly every object; however, the reverb object does not have a parameter for reverb mix (the mix between wet and dry signal). So, when setting up a reverb, it's important to route both the reverberated and the dry signal through to the AudioContext.

The Reverb object has four parameters: Damping, Early Reflections Level, Late Reflections Level and Room Size. Damping tells the reverb how to filter out high frequencies. A heavily damped room is like a room that absorbs sound with lots of plush furniture and thick carpets. The early reflections level sets how much the sound seems to bounce back, while the late reflections level controls how the reverb tails off. The room size essentially sets the duration of the reverb, but it's important to play around with all of these settings to really get a feel for them.

In this block of code, we initialize the reverb with a single output chaannel, set the size and damping, then connect an input.

```
r = new Reverb(ac, 1);
r.setSize(0.7);
r.setDamping(0.5);
r.addInput(g);
```

Code Listing 5.3.2. Reverb_01.pde

```
// Reverb_01.pde
// this example demonstrates how to use the Reverb object


import beads.*;


AudioContext ac;


String sourceFile;
SamplePlayer sp;


// standard gain objects
Gain g;
```

```
Glide gainValue;


// our Reverberation unit generator
Reverb r;


void setup()
{
  size(800, 600);

  ac = new AudioContext();


  sourceFile = sketchPath("") + "Drum_Loop_01.wav";


  try {
    sp = new SamplePlayer(ac, new Sample(sourceFile));
   }
  catch(Exception e)
  {
    println("Exception while attempting to load sample!");
    e.printStackTrace();
    exit();
  }
  sp.setKillOnEnd(false);


  // as usual, we create a gain that will control the volume
  // of our sample player
  gainValue = new Glide(ac, 0.0, 20);
  g = new Gain(ac, 1, gainValue);
  g.addInput(sp); // connect the filter to the gain


  // Create a new reverb with a single output channel
  r = new Reverb(ac, 1);
  // Set the room size (between 0 and 1) 0.5 is the default
  r.setSize(0.7);
  // Set the damping (between 0 and 1) - the higher the
  // dampening, the fewer resonant high frequencies
  r.setDamping(0.5);
  // You can also control a Reverb's early reflections and
  // late reverb.
  // To do so, use r.setEarlyReflectionsLevel(0-1); or
  // r.setLateReverbLevel(0-1);
  r.addInput(g); // connect the gain to the reverb


  // connect the Reverb to the AudioContext
  ac.out.addInput(r);

  // Remember, the reverb unit only outputs a reverberated
  // signal. So if we want to hear the dry drums as well,
  // then we will also need to connect the SamplePlayer to
  // the output.
  ac.out.addInput(g);
```

```
  ac.start();

  background(0);
  text("Click to hear a Reverb object in action.", 50, 50);
}


void draw(){}


void mousePressed()
{
    // set the gain based on mouse position
    gainValue.setValue((float)mouseX/(float)width);
    // move the playback pointer to the first loop point
    sp.setToLoopStart();
    sp.start();
}
```

5.3.3. Compressor (Compressor_01)

Compression is one of the most complex and misunderstood audio processing tools. Neophytes tend to think of compression as a black box that can fix all sorts of problems with audio. At its core, however, compression is simply a tool for evening out the loudness of recorded audio. It makes the dynamic range of recorded audio narrower, which allows us to turn up the gain.

Compression has a four basic parameters: threshold, ratio, attack and decay. Threshold is the loudness level at which compression begins to occur. This is usually indicated in decibels, but in Beads this is indicated by a number between 0.0 and 1.0. Ratio is the amount of compression. A 2:1 ratio indicates that for every 2 input decibels above the threshold, the compressor will output 1 decibel above the threshold. So, after the sound crosses the threshold, a 2:1 ratio will apply a gain of 50%. Attack and decay are durations that indicate how long it takes for compression to ramp up once the sound crosses the threshold, and how long it takes for compression to stop, after the sound has dipped below the threshold.

There are many other parameters that might occur on a compressor, and even the Beads Compressor object includes a few other parameters; however, for most jobs, these four parameters are sufficient.

In this example, we set up a compressor, attaching inputs and outputs as usual, then we set the attack, decay, ratio and threshold. In this block of code, we set the attack to 30ms, the decay to 200ms, the ratio to 4:1, and the threshold to 0.6.

```
c = new Compressor(ac, 1);
c.setAttack(30);
c.setDecay(200);
c.setRatio(4.0);
c.setThreshold(0.6);
```

Code Listing 5.3.3. Compressor_01.pde

```
// Compressor_01.pde
// this example demonstrates how to use the Compressor object


import beads.*;


AudioContext ac;


String sourceFile;
SamplePlayer sp;


// standard gain objects
Gain g;
Glide gainValue;


Compressor c; // our Compressor unit generator


void setup()
{
  size(800, 600);

  ac = new AudioContext();


  sourceFile = sketchPath("") + "Drum_Loop_01.wav";


  try {
    sp = new SamplePlayer(ac, new Sample(sourceFile));
   }
  catch(Exception e)
  {
    println("Exception while attempting to load sample!");
    e.printStackTrace();
    exit();
  }


  sp.setKillOnEnd(false);


  // Create a new compressor with a single output channel.
  c = new Compressor(ac, 1);

  // The attack is how long it takes for compression to ramp
  // up, once the threshold is crossed.
  c.setAttack(30);


  // The decay is how long it takes for compression to trail
```

```
  // off, once the threshold is crossed in the opposite
  // direction.
  c.setDecay(200);

  // The ratio and the threshold work together to determine
  // how much a signal is squashed. The ratio is the
  // NUMERATOR of the compression amount 2.0 = 2:1 = for
  // every two decibels above the threshold, a single decibel
  // will be output. The threshold is the loudness at which
  // compression can begin
  c.setRatio(4.0);
  c.setThreshold(0.6);

  // The knee is an advanced setting that you should leave
  // alone unless you know what you are doing.
  //c.setKnee(0.5);

  // connect the SamplePlayer to the compressor
  c.addInput(sp);


  gainValue = new Glide(ac, 0.0, 20);
  g = new Gain(ac, 1, gainValue);
  // connect the Compressor to the gain
  g.addInput(c);


  // connect the Compressor to the AudioContext
  ac.out.addInput(c);
  ac.start();

  background(0);
  text("Click to hear the Compressor object in action.",
                                                    50, 50);
}


void draw(){}


void mousePressed()
{
    // set the gain based on mouse position
    gainValue.setValue((float)mouseX/(float)width);
    // move the playback pointer to the first loop point
    sp.setToLoopStart();
    sp.start();
}
```

5.3.4. WaveShaper (WaveShaper_01)

Wave shaping takes an incoming waveform and maps it to values from a stored waveform. The process is just a table lookup of the value in the input to the value in the waveshape. Wave shaping can be used to apply a lot of different effects, and to expand the harmonic spectrum of a sound. We can load a wave shape from an array, or from a Buffer. In this example, we load a wave shape from an array, then use that wave shape to apply a strange distortion effect to our drum loop.

To instantiate the WaveShaper object, simply provide the AudioContext object and the wave shape. Then set up the input and output as usual, using the addInput method.

```
float[] WaveShape1 = {0.0, 0.9, 0.1, 0.9, -0.9, 0.0, -0.9,
                      0.9, -0.3, -0.9, -0.5};
ws = new WaveShaper(ac, WaveShape1);
```

In order to use a Buffer as a waveshape, you merely need to insert a Buffer object into the WaveShaper constructor. You can experiment with this by inserting any of the pre-defined buffers.

```
ws = new WaveShaper(ac, Buffer.SAW);
```

Code Listing 5.3.4. WaveShaper_01.pde

```
// WaveShaper_01.pde
// This example demonstrates a WaveShaper, which maps an
// incoming signal onto a specified wave shape. You can
// specify wave shape using an array of floats, or by using a
// short audio file. In this example we use an array of
// floats.


import beads.*;


AudioContext ac;


String sourceFile;
SamplePlayer sp;


// standard gain objects
Gain g;
Glide gainValue;


WaveShaper ws; // our WaveShaper unit generator
```

```
void setup()
{
  size(800, 600);

  ac = new AudioContext();


  sourceFile = sketchPath("") + "Drum_Loop_01.wav";


  // Try/Catch blocks will inform us if the file can't
  // be found
  try {
    // initialize our SamplePlayer, loading the file
    // indicated by the sourceFile string
    sp = new SamplePlayer(ac, new Sample(sourceFile));
   }
  catch(Exception e)
  {
    // if there is an error, show an error message (at the
    // bottom of the processing window)
    println("Exception while attempting to load sample!");
    e.printStackTrace();
    exit();
  }

  // we would like to play the sample multiple times, so we
  // set KillOnEnd to false
  sp.setKillOnEnd(false);


  // as usual, we create a gain that will control the volume
  // of our sample player
  gainValue = new Glide(ac, 0.0, 20);
  g = new Gain(ac, 1, gainValue);
  g.addInput(sp); // connect the filter to the gain


  // This wave shape applies a strange-sounding distortion.
  float[] WaveShape1 = {0.0, 0.9, 0.1, 0.9, -0.9, 0.0, -0.9,
                                        0.9, -0.3, -0.9, -0.5};

  // uncomment these lines to set the gain on the WaveShaper
  //ws.setPreGain(4.0);
  //ws.setPostGain(4.0);

  // instantiate the WaveShaper with the wave shape
  ws = new WaveShaper(ac, WaveShape1);
  // connect the gain to the WaveShaper
  ws.addInput(g);


  // connect the WaveShaper to the AudioContext
  ac.out.addInput(ws);
  // begin audio processing
  ac.start();
```

```
  background(0);
  text("Click to hear a WaveShaper in action.", 50, 50);
}


void draw(){}


void mousePressed()
{
    gainValue.setValue((float)mouseX/(float)width);
    sp.setToLoopStart();
    sp.start();
}
```

6. Saving Your Sounds

If you're using a Mac or Linux machine, then the natural choice for saving your work is the RecordToFile object. Unfortunately, this object relies on the org.tritonus library, which is not supported in Windows. Java itself doesn't support streaming audio directly to the disk. Fortunately, we can use the RecordToSample object to perform the task of saving audio to disk. The RecordToSample object is implemented using pure java sound, so it is compatible on any platform.

6.1 Using RecordToSample

The RecordToSample object allows a programmer to store the incoming audio data in a buffer known as a Sample. The Sample class is very useful because, after we have captured some audio data, we can use the Sample class to write an audio file with just a few lines of code. The RecordToSample object works just like most objects, with one exception. Since this object has no outputs, we need to tell it when to update by adding it as a dependent of the master AudioContext. This allows it to update whenever the AudioContext is updated.

Note the import statements at the top of this example. These are necessary in order to specify audio formats for recording and saving.

```
import javax.sound.sampled.AudioFormat;
import javax.sound.sampled.AudioFileFormat.Type;
```

The RecordToSample instantiation process is slightly more intricate than it is for most objects. In this case, we instantiate an AudioFormat that indicates the format for the recorded data. Then we instantiate a Sample object to hold the audio data. Finally, we create a RecordToSample object using the previous two objects as parameters.

```
AudioFormat af = new AudioFormat(44100.0f,
                                 16,
                                 1,
                                 true,
                                 true);
outputSample = new Sample(af, 44100);
rts = new RecordToSample(ac, outputSample,
RecordToSample.Mode.INFINITE);
```

Then we tell the AudioContext that the new object is a dependent.

```
ac.out.addDependent(rts);
```

We can save the recorded data by calling the write function that is provided by the Sample class. All we have to provide is a filename and an audio format.

```
outputSample.write(sketchPath("") + "out.wav",
javax.sound.sampled.AudioFileFormat.Type.WAVE);
```

Code Listing 6.1.1. RecordToSample_01.pde

```
// RecordToSample_01.pde
// This is an extension of Delay_01.pde

// this is necessary so that we can use the File class
import java.io.*;

// these imports allow us to specify audio file formats
import javax.sound.sampled.AudioFormat;
import javax.sound.sampled.AudioFileFormat.Type;

import beads.*; // import the beads library
AudioContext ac; // create our AudioContext

// declare our unit generators
WavePlayer modulator;
Glide modulatorFrequency;
WavePlayer carrier;

// our envelope and gain objects
Envelope gainEnvelope;
Gain synthGain;

// our delay objects
TapIn delayIn;
TapOut delayOut;
Gain delayGain;

// our recording objects
RecordToSample rts;
Sample outputSample;

void setup()
{
  size(600, 300);

  // initialize our AudioContext
  ac = new AudioContext();

  // create the modulator, this WavePlayer will control
  // the frequency of the carrier
  modulatorFrequency = new Glide(ac, 20, 30);
  modulator = new WavePlayer(ac,
                             modulatorFrequency,
```

```
                                Buffer.SINE);

  // create a custom frequency modulation function
  Function frequencyModulation = new Function(modulator)
  {
    public float calculate() {
      // return x[0], scaled into an appropriate
      // frequency range
      return (x[0] * 100.0) + mouseY;
    }
  };

  // create a second WavePlayer, control the frequency
  // with the function created above
  carrier = new WavePlayer(ac,
                           frequencyModulation,
                           Buffer.SINE);

  // create the envelope object that will control the gain
  gainEnvelope = new Envelope(ac, 0.0);
  // create a Gain object, connect it to the gain envelope
  synthGain = new Gain(ac, 1, gainEnvelope);
  // connect the carrier to the Gain input
  synthGain.addInput(carrier);

  // set up our delay
  // create the delay input - the second parameter sets the
  // maximum delay time in milliseconds
  delayIn = new TapIn(ac, 2000);
  // connect the synthesizer to the delay
  delayIn.addInput(synthGain);

  // create the delay output - the final parameter is the
  // length of the initial delay time in milliseconds
  delayOut = new TapOut(ac, delayIn, 500.0);
  delayGain = new Gain(ac, 1, 0.50);// the gain for our delay
  // connect the delay output to the gain
  delayGain.addInput(delayOut);

  // to feed the delay back into itself, simply
  // uncomment this line
  delayIn.addInput(delayGain);

  // setup the recording unit generator
  try{
    // specify the recording format
    AudioFormat af = new AudioFormat(44100.0f,
                                     16,
                                     1,
                                     true,
                                     true);

    // create a buffer for the recording
    outputSample = new Sample(af, 44100);

    // initialize the RecordToSample object
    rts = new RecordToSample(ac,
```

```
                                outputSample,
                                RecordToSample.Mode.INFINITE);
  }
  catch(Exception e){
    e.printStackTrace();
    exit();
  }
  rts.addInput(synthGain);
  rts.addInput(delayGain);
  ac.out.addDependent(rts);

  // connect the Gain output to the AudioContext
  ac.out.addInput(synthGain);
  // connect the delay output to the AudioContext
  ac.out.addInput(delayGain);
  ac.start(); // start audio processing

  background(0);
  text("Click me to demonstrate delay!", 100, 100);
  text("Press s to save the performance and exit", 100, 120);
}

void draw()
{
  // set the modulator frequency
  modulatorFrequency.setValue(mouseX);
}

void mousePressed()
{
  // when the mouse button is pressed, at a 50ms attack
  // segment to the envelope
  // and a 300 ms decay segment to the envelope
  gainEnvelope.addSegment(0.7, 50);
  gainEnvelope.addSegment(0.0, 300);
}

    // event handler for mouse clicks
    void keyPressed()
    {
      if( key == 's' || key == 'S' )
      {
        rts.pause(true);

        try{
          outputSample.write(sketchPath("") +
              "outputSample.wav",
              javax.sound.sampled.AudioFileFormat.Type.WAVE);
        }
        catch(Exception e){
          e.printStackTrace();
          exit();
        }
        rts.kill();
        exit();
      }
    }
```

7. Using Audio Input

There are a million reasons why you might want to use audio input. Perhaps your Processing sketch is interactive. Perhaps you want to use externally-generated audio to shape your Processing visuals. Or maybe you want to use Beads as an effects processor in a sound installation. In this chapter we're going to see how to incorporate audio input into our Processing sketches.

BUG NOTE: At the time of this writing, selecting an audio input other than your default input appears to be buggy. For the time being, if you want to take input from something other than your default, it is probably best just to temporarily change your default audio input. If you're working on a Mac, you might also look into the Jack library (http://jackosx.com/). Unfortunately, the windows port of Jack is quite buggy, in my experience.

7.1. Getting an Audio Input Unit Generator (Audio_Input_01)

The audio input unit generator is easy to use in Beads, and it can be used just like any other unit generator. It only differs from other unit generators in how it is instantiated. One difference is that the audio input unit generator is of type UGen, rather than a class that derives from UGen. So when you set up an audio input unit generator, declare it as a UGen. Also, when you create an audio input, you don't call a constructor, you simply ask the AudioContext to give you access to an audio input by calling the getAudioInput() function. The resulting unit generator can then be connected in the usual fashion using the addInput routine.

```
UGen microphoneIn = ac.getAudioInput();
```

For more on audio input in Beads, see the AudioContext javadoc at www.beadsproject.net/doc/net/beadsproject/beads/core/AudioContext.html

Code Listing 7.1. Audio_Input_01.pde

```
// Audio_Input_01.pde

import beads.*;

AudioContext ac;

void setup() {
  size(800,800);
```

```
  ac = new AudioContext();

  // get an AudioInput UGen from the AudioContext
  // this will setup an input from whatever input is your
  // default audio input (usually the microphone in)
  // changing audio inputs in beads is a little bit janky (as
  // of this writing)
  // so it's best to change your default input temporarily,
  // if you want to use a different input
  UGen microphoneIn = ac.getAudioInput();

  // set up our usual master gain object
  Gain g = new Gain(ac, 1, 0.5);
  g.addInput(microphoneIn);
  ac.out.addInput(g);

  ac.start();
}

// draw the input waveform on screen
// this code is based on code from the Beads tutorials
// written by Ollie Brown
void draw()
{
  loadPixels();

  //set the background
  Arrays.fill(pixels, color(0));

  //scan across the pixels
  for(int i = 0; i < width; i++)
  {
    // for each pixel, work out where in the current audio
    // buffer we are
    int buffIndex = i * ac.getBufferSize() / width;
    // then work out the pixel height of the audio data at
    // that point
    int vOffset = (int)((1 + ac.out.getValue(0, buffIndex)) *
                  height / 2);
    //draw into Processing's convenient 1-D array of pixels
    pixels[vOffset * height + i] = color(255);
  }
  // paint the new pixel array to the screen
  updatePixels();
}
```

7.2. Recording and Playing a Sample (Record_Sample_01)

In this example, we're going to see how we can use Beads to record audio then use that recording later in the program. This program simply records a sample and plays it back, however, since the playback is executed via the SamplePlayer object, we can easily extend that audio chain to manipulate the recorded buffer in innumerable different ways.

There are two basic parts to this program: the recording and the playback. Audio is recorded via the RecordToSample object, which is set up in this block of code.

```
// setup a recording format
AudioFormat af = new AudioFormat(44100.0f,
                                 16,
                                 1,
                                 true,
                                 true);
// create a holder for audio data
targetSample = new Sample(af, 44100);
rts = new RecordToSample(ac, targetSample,
RecordToSample.Mode.INFINITE);
```

The recording is controlled using the left mouse button, and the start and pause routines in the RecordToSample object.

```
// when the user left-clicks
if( mouseButton == LEFT )
{
  // if the RecordToSample object is currently paused
  if( rts.isPaused() )
  {
    // clear the target sample
    targetSample.clear();
    // start recording
    rts.start();
  }
  else
  {
    // stop recording
    rts.pause(true);
  }
}
```

The playback is initiated using the right mouse button. When the user right-clicks, we set up a new SamplePlayer based on the recorded sample, then connect it to our master gain, tell it to destroy itself when it finishes, and call the start function.

```
SamplePlayer sp = new SamplePlayer(ac, targetSample);
g.addInput(sp);
sp.setKillOnEnd(true);
sp.start();
```

Code Listing 7.2. Record_Sample_01.pde

```
// Record_Sample_01.pde

import beads.*;
```

```
// we need to import the java sound audio format definitions
import javax.sound.sampled.AudioFormat;

AudioContext ac;

Gain g; // our master gain

// this object is used to start and stop recording
RecordToSample rts;
// this object will hold our audio data
Sample targetSample;

boolean recording = false;

void setup() {
  size(800,800);

  // initialize the AudioContext
  ac = new AudioContext();

  // get an AudioInput UGen from the AudioContext
  // this will setup an input from whatever input is your
  // default audio input (usually the microphone in)
  UGen microphoneIn = ac.getAudioInput();

  // setup the recording unit generator
  try{
    // setup a recording format
    AudioFormat af = new AudioFormat(44100.0f,
                                     16,
                                     1,
                                     true,
                                     true);

    // create a holder for audio data
    targetSample = new Sample(af, 44100);

    // initialize the RecordToSample object
    rts = new RecordToSample(ac,
                             targetSample,
                             RecordToSample.Mode.INFINITE);
  }
  catch(Exception e){
    e.printStackTrace();
    exit();
  }
  // connect the microphone input to the RecordToSample
  rts.addInput(microphoneIn);
  // tell the AudioContext to work with the RecordToSample
  ac.out.addDependent(rts);
  // pause the RecordToSample object
  rts.pause(true);

  // set up our usual master gain object
  g = new Gain(ac, 1, 0.5);
  g.addInput(microphoneIn);
  ac.out.addInput(g);
```

```
  ac.start();
}

void draw()
{
  background(0);
  text("Left click to start/stop recording. Right click to
                                                  play.", 100, 100);
  if( recording ) text("Recording...", 100, 120);
}

void mousePressed()
{
  // when the user left-clicks
  if( mouseButton == LEFT )
  {
    // if the RecordToSample object is currently paused
    if( rts.isPaused() )
    {
      // note that we are now recording
      recording = true;
      // clear the target sample
      targetSample.clear();
      // and start recording
      rts.start();
    }
    // if the RecordToSample is recording
    else
    {
      // note that we are no longer recording
      recording = false;
      // and stop recording
      rts.pause(true);
    }
  }
  // if the user right-clicks
  else
  {
    // Instantiate a new SamplePlayer with the recorded
    // sample.
    SamplePlayer sp = new SamplePlayer(ac, targetSample);
    g.addInput(sp);
    // tell the SamplePlayer to destroy itself when it's done
    sp.setKillOnEnd(true);
    sp.start();
  }
}
```

7.3. Granulating from Audio Input (Granulating_Input_01)

This example extends the previous example by showing how we can manipulate the audio when we initiate playback. In this case, we simply replace the SamplePlayer object with a GranularSamplePlayer object.

Code Listing 7.3. Granulating_Input_01.pde

```
//Granulating_Input_01.pde
// This example is just like the previous example, except
// when we initiate playback, we use a GranularSamplePlayer
// if you want to automate the recording and granulation
// process, then you could use a clock object

import beads.*;

// we need to import the java sound audio format definitions
import javax.sound.sampled.AudioFormat;

// declare the parent AudioContext
AudioContext ac;
// our master gain
Gain g;
// this object is used to start and stop recording
RecordToSample rts;
// this object will hold our audio data
Sample targetSample;

// are we currently recording a sample?
boolean recording = false;

void setup() {
  size(800,800);

  ac = new AudioContext();

  // get an AudioInput UGen from the AudioContext
  // this will setup an input from whatever input is your
  // default audio input (usually the microphone in)
  UGen microphoneIn = ac.getAudioInput();

  // setup the recording unit generator
  try{
    AudioFormat af = new AudioFormat(44100.0f,
                                     16,
                                     1,
                                     true,
                                     true);

    // create a holder for audio data
    targetSample = new Sample(af, 44100);

    // initialize the RecordToSample object
    rts = new RecordToSample(ac,
                             targetSample,
                             RecordToSample.Mode.INFINITE);
  }
  catch(Exception e){
    e.printStackTrace();
    exit();
  }
  // connect the microphone input to the RecordToSample
```

```
  // object
  rts.addInput(microphoneIn);
  // tell the AudioContext to work with the RecordToSample
  // object
  ac.out.addDependent(rts);
  // pause the RecordToSample object
  rts.pause(true);

  // set up our usual master gain object
  g = new Gain(ac, 1, 0.5);
  g.addInput(microphoneIn);
  ac.out.addInput(g);

  ac.start();
}

void draw()
{
  background(0);
  text("Left click to start/stop recording. Right click to
                                        granulate.", 100, 100);
  if( recording ) text("Recording...", 100, 120);
}

void mousePressed()
{
  // when the user left-clicks
  if( mouseButton == LEFT )
  {
    // if the RecordToSample object is currently paused
    if( rts.isPaused() )
    {
      // note that we are now recording
      recording = true;
      // clear the target sample
      targetSample.clear();
      // and start recording
      rts.start();
    }
    // if the RecordToSample is recording
    else
    {
      // note that we are no longer recording
      recording = false;
      // and stop recording
      rts.pause(true);
    }
  }
  // if the user right-clicks
  else
  {
    // Instantiate a new SamplePlayer with the recorded
    // sample.
    GranularSamplePlayer gsp = new GranularSamplePlayer(ac,
                                                  targetSample);
    // set the grain interval to about 20ms between grains
    gsp.setGrainInterval(new Static(ac, 20f));
```

```
    // set the grain size to about 50ms (smaller is sometimes
    // a bit too grainy for my taste).
    gsp.setGrainSize(new Static(ac, 50f));
    // set the randomness, which will add variety to all the
    // parameters
    gsp.setRandomness(new Static(ac, 50f));
    // connect the GranularSamplePlayer to the Gain
    g.addInput(gsp);
    // tell the GranularSamplePlayer to destroy itself when
    // it finishes
    gsp.setKillOnEnd(true);
    gsp.start();
  }
}
```

8. Using MIDI with Beads

Currently, Beads doesn't provide unit generators for working with the MIDI protocol. This isn't an enormous drawback though; there are good reasons to keep the audio separate from MIDI. Also, Processing already has a great library for simple MIDI input and output: The MIDI Bus. In this chapter, we're going to look at a few ways of integrating MIDI into your Beads projects using The MIDI Bus. It's easier than you might think!

IMPORTANT: Some of the examples in this chapter require you to have a MIDI device connected to your system in order to function properly.

8.1. Installing The MIDI Bus

To install The MIDI Bus, download the latest release from The MIDI Bus website (http://smallbutdigital.com/themidibus.php). Unzip the contents, then copy the "themidibus" directory into the "libraries" folder within your sketchbook. To find out where your sketch book is located, click "File" then "Preferences" within the Processing window. Then there is an option for "Sketchbook Location."

8.2. Basic MIDI Input

In the first two MIDI examples, we're going to see how to use Beads to respond to MIDI events. MIDI events usually come from a MIDI keyboard, but of course, you can route MIDI events from any MIDI enabled device or software.

8.2.1. MIDI-Controlled Sine Wave Synthesizer (MIDI_SYNTH_01)

The first MIDI Bus example plays a sine wave in response to MIDI input. To see this example in action, you will need to have a MIDI device connected to your computer.

Setting up a MidiBus object with the default parameters is a snap. The MidiBus constructor takes four parameters. The first parameter is the calling program. The second parameter is the index of the MIDI input device. The third parameter is the index of the MIDI output device. The final parameter is the name of the new bus. In this example, we will use the default input and output devices, indicated by zeroes.

```
busA = new MidiBus(this, 0, 0, "busA");
```

After the bus is setup, all we have to do is wait for MIDI events, and respond to them as they come in. Using the MIDI Bus library, this can be done by implementing the noteOn function.

Each Beads WavePlayer object can only output a single wave at a time. Hence, if we want to create a polyphonic synthesizer, we need to create a new WavePlayer object each time a note is pressed (and destroy them when a note is released). In this example, we use an ArrayList to store our Beads,* and we use a subclass called SimpleSynth to group the Beads for a single pitch together. The SimpleSynth class allows us to repeatedly set up the unit generators we need to respond to MIDI messages.

The SimpleSynth constructor sets up a WavePlayer, an Envelope and a Gain, then connects all three to the master gain object we created when the program started. Finally, it adds a segment to the envelope, to tell the Gain to rise to 0.5 over 300ms.

```
pitch = midiPitch;
// set up the new WavePlayer,
// convert the MidiPitch to a frequency
wp = new WavePlayer(ac, 440.0 * pow(2, ((float)midiPitch -
                                        59.0)/12.0), Buffer.SINE);
e = new Envelope(ac, 0.0);
g = new Gain(ac, 1, e);
g.addInput(wp);
MasterGain.addInput(g);
e.addSegment(0.5, 300);
```

Notes are then killed within the noteOff function that is called by the MidiBus when a note is released.

* You can also use the BeadArray class provided by the Beads library

Code Listing 8.2.1. MIDI_SYNTH_01.pde

```
// MIDI_SYNTH_01.pde
// this example builds a simple midi synthesizer
// for each incoming midi note, we create a new set of beads
// (encapsulated by a class)
// these beads are stored in a vector
// and destroyed when we get a corresponding note-off message

// Import the MidiBus library
import themidibus.*;

// import the beads library
import beads.*;

// our parent MidiBus object
MidiBus busA;
```

```
AudioContext ac;
Gain MasterGain;

ArrayList synthNotes = null;

void setup()
{
  size(600, 400);
  background(0);

  // the MidiBus constructor takes four arguments
  // 1 - the calling program (this)
  // 2 - the input device
  // 3 - the output device
  // 4 - the bus name
  // in this case, we just use the defaults
  busA = new MidiBus(this, 0, 0, "busA");

  synthNotes = new ArrayList();

  ac = new AudioContext();
  MasterGain = new Gain(ac, 1, 0.5);
  ac.out.addInput(MasterGain);
  ac.start();


  background(0);
  text("This program will not do anything if you do not have
                                    a MIDI device", 100, 100);
  text("connected to your computer.", 100, 112);
  text("This program plays sine waves in response to Note-On
                                     messages.", 100, 124);
}

void draw()
{
  for( int i = 0; i < synthNotes.size(); i++ )
  {
    SimpleSynth s = (SimpleSynth)synthNotes.get(i);
    // if this bead has been killed
    if( s.g.isDeleted() )
    {
      // destroy the synth (set things to null so that memory
      // cleanup can occur)
      s.destroy();
      // then remove the parent synth
      synthNotes.remove(s);
    }
  }
}

// respond to MIDI note-on messages
void noteOn(int channel,
            int pitch,
            int velocity,
            String bus_name)
```

```
{
  background(50);
  stroke(255); fill(255);
  text("Note On:", 100, 100);
  text("Channel:" + channel, 100, 120);
  text("Pitch:" + pitch, 100, 140);
  text("Velocity:" + velocity, 100, 160);
  text("Recieved on Bus:" + bus_name, 100, 180);

  synthNotes.add(new SimpleSynth(pitch));
}

// respond to MIDI note-off messages
void noteOff(int channel,
             int pitch,
             int velocity,
             String bus_name)
{
  background(0);
  stroke(255); fill(255);
  text("Note Off:", 100, 100);
  text("Channel:" + channel, 100, 120);
  text("Pitch:" + pitch, 100, 140);
  text("Velocity:" + velocity, 100, 160);
  text("Recieved on Bus:" + bus_name, 100, 180);

  for( int i = 0; i < synthNotes.size(); i++ )
  {
    SimpleSynth s = (SimpleSynth)synthNotes.get(i);
    if( s.pitch == pitch )
    {
      s.kill();
      synthNotes.remove(s);
      break;
    }
  }
}

// this is our simple synthesizer object
class SimpleSynth
{
  public WavePlayer wp = null;
  public Envelope e = null;
  public Gain g = null;
  public int pitch = -1;

  // the constructor for our sine wave synthesizer
  SimpleSynth(int midiPitch)
  {
    pitch = midiPitch;
    // set up the new WavePlayer, convert the MidiPitch to a
    // frequency
    wp = new WavePlayer(ac,
                        440.0 * pow(2, ((float)midiPitch -
                          59.0)/12.0),
                        Buffer.SINE);
    e = new Envelope(ac, 0.0);
```

```
    g = new Gain(ac, 1, e);
    g.addInput(wp);
    MasterGain.addInput(g);
    e.addSegment(0.5, 300);
  }
  // when this note is killed, ramp the amplitude down to 0
  // over 300ms
  public void kill()
  {
    e.addSegment(0.0, 300, new KillTrigger(g));
  }
  // destroy the component beads so that they can be cleaned
  // up by the java virtual machine
  public void destroy()
  {
    wp.kill();
    e.kill();
    g.kill();
    wp = null;
    e = null;
    g = null;
  }
}
```

8.2.2. MIDI-Controlled FM Synthesizer (MIDI_SYNTH_02)

The second MIDI input example is very similar to the first. This example expands on the first example by using The MIDI Bus to enumerate the available MIDI devices and show which ones are actually connected. The array of available MIDI devices is created by calling the MidiBus.availableInputs function. The list of devices that are being watched is retrieved by calling the attachedInputs function which is exposed by our MidiBus object.

```
String[] available_inputs = MidiBus.availableInputs();
…
println(busA.attachedInputs());
```

The other way that this example differs from the first MIDI input example is that we create a more complex synthesizer. This time, the Synth subclass creates a frequency modulation patch for each incoming note, then attaches a filter with an LFO on the cutoff.

Code Listing 8.2.2. MIDI_SYNTH_02.pde

```
// MIDI_SYNTH_02.pde
// this example builds a simple midi synthesizer
// for each incoming midi note, we create a new set of beads
// (encapsulated by a class)
// these beads are stored in a vector
// and destroyed when we get a corresponding note-off message
```

```
import themidibus.*;
import beads.*;

// The MidiBus object that will handle midi input
MidiBus busA;

AudioContext ac;
Gain MasterGain;

// this ArrayList will hold synthesizer objects
// we will instantiate a new synthesizer object for each note
ArrayList synthNotes = null;

void setup()
{
  size(600,400);
  background(0);

  // List all available input devices
  println();
  println("Available MIDI Devices:");
  //Returns an array of available input devices
  String[] available_inputs = MidiBus.availableInputs();
  for(int i = 0;i < available_inputs.length;i++)
   System.out.println("["+i+"] \""+available_inputs[i]+"\"");

  // Create a first new MidiBus attached to the IncommingA
  // Midi input device and the OutgoingA Midi output device.
  busA = new MidiBus(this, 0, 2, "busA");

  println();
  println("Inputs on busA");
  //Print the devices attached as inputs to busA
  println(busA.attachedInputs());

  synthNotes = new ArrayList();

  ac = new AudioContext();
  MasterGain = new Gain(ac, 1, 0.3);
  ac.out.addInput(MasterGain);
  ac.start();

  background(0);
  text("This program will not do anything if you do not have
                                    a MIDI device", 100, 100);
  text("connected to your computer.", 100, 112);
  text("This program is a synthesizer that responds to Note-
                                     On Messages.", 100, 124);
}

void draw()
{
  for( int i = 0; i < synthNotes.size(); i++ )
  {
    Synth s = (Synth)synthNotes.get(i);
    // if this bead has been killed
    if( s.g.isDeleted() )
```

```
    {
      // destroy the synth (set things to null so that memory
      // cleanup can occur)
      s.destroy();
      // then remove the parent synth
      synthNotes.remove(s);
    }
  }
}

// respond to MIDI note-on messages
void noteOn(int channel,
            int pitch,
            int velocity,
            String bus_name)
{
  background(50);
  stroke(255); fill(255);
  text("Note On:", 100, 100);
  text("Channel:" + channel, 100, 120);
  text("Pitch:" + pitch, 100, 140);
  text("Velocity:" + velocity, 100, 160);
  text("Recieved on Bus:" + bus_name, 100, 180);

  synthNotes.add(new Synth(pitch));
}

// respond to MIDI note-off messages
void noteOff(int channel,
             int pitch,
             int velocity,
             String bus_name)
{
  background(0);
  stroke(255); fill(255);
  text("Note Off:", 100, 100);
  text("Channel:" + channel, 100, 120);
  text("Pitch:" + pitch, 100, 140);
  text("Velocity:" + velocity, 100, 160);
  text("Recieved on Bus:" + bus_name, 100, 180);

  for( int i = 0; i < synthNotes.size(); i++ )
  {
    Synth s = (Synth)synthNotes.get(i);
    if( s.pitch == pitch )
    {
      s.kill();
      synthNotes.remove(s);
      break;
    }
  }
}

// this is our synthesizer object
class Synth
{
  public WavePlayer carrier = null;
```

```
public WavePlayer modulator = null;
public Envelope e = null;
public Gain g = null;

public int pitch = -1;

// our filter and filter envelope
LPRezFilter lowPassFilter;
WavePlayer filterLFO;

Synth(int midiPitch)
{
  // get the midi pitch and create a couple holders for the
  // midi pitch
  pitch = midiPitch;
  float fundamentalFrequency =
            440.0 * pow(2, ((float)midiPitch - 59.0)/12.0);
  Static ff = new Static(ac, fundamentalFrequency);

  // instantiate the modulator WavePlayer
  modulator = new WavePlayer(ac,
                             0.5 * fundamentalFrequency,
                             Buffer.SINE);

  // create our frequency modulation function
  Function frequencyModulation =
           new Function(modulator, ff) {
    public float calculate() {
      // the x[1] here is the value of a sine wave
      // oscillating at the fundamental frequency
      return (x[0] * 1000.0) + x[1];
    }
  };
  // instantiate the carrier WavePlayer
  // set up the carrier to be controlled by the frequency
  // of the modulator
  carrier = new WavePlayer(ac,
                           frequencyModulation,
                           Buffer.SINE);

  // set up the filter and LFO
  filterLFO = new WavePlayer(ac, 8.0, Buffer.SINE);
  Function filterCutoff = new Function(filterLFO)
  {
    public float calculate() {
      // set the filter cutoff to oscillate between 1500Hz
      // and 2500Hz
      return ((x[0] * 500.0) + 2000.0);
    }
  };
  lowPassFilter = new LPRezFilter(ac, filterCutoff, 0.96);
  lowPassFilter.addInput(carrier);

  // set up and connect the gains
  e = new Envelope(ac, 0.0);
  g = new Gain(ac, 1, e);
  g.addInput(lowPassFilter);
```

```
    MasterGain.addInput(g);

    // add a segment to our gain envelope
    e.addSegment(0.5, 300);
  }
  public void kill()
  {
    e.addSegment(0.0, 300, new KillTrigger(g));
  }
  public void destroy()
  {
    carrier.kill();
    modulator.kill();
    lowPassFilter.kill();
    filterLFO.kill();
    e.kill(); g.kill();
    carrier = null;
    modulator = null;
    lowPassFilter = null;
    filterLFO = null;
    e = null; g = null;
  }
}
```

8.3. Basic MIDI Output

In this brief section we see how to use The MIDI Bus to send MIDI events to a MIDI synthesizer residing on your computer. This topic is only briefly touched on because it doesn't involve the Beads library at all.

8.3.1. Sending MIDI to the Default device (MIDI_Output_01)

When we want to send MIDI messages to a MIDI synthesizer, we simply instantiate the MidiBus and tell it which synthesizer to use. Notice that the constructor used here differs from the one used in the previous example. This time we indicate the MIDI output device by name, in this case, "Java Sound Synthesizer."

```
myBus = new MidiBus(this, -1, "Java Sound Synthesizer");
```

These three lines send a note-on message, wait for 100ms, then send a note-off message.

```
// start a midi pitch
myBus.sendNoteOn(channel, pitch, velocity);
// wait for 100ms
delay(100);
// then stop the note we just started
myBus.sendNoteOff(channel, pitch, velocity);
```

And this block of code switches to a random MIDI instrument, as specified by the General MIDI standard.

```
// This is the status byte for a program change
int status_byte = 0xC0;
// random voice
int byte1 = (int)random(128);
// This is not used for program change so ignore it and set
// it to 0
int byte2 = 0;
//Send the custom message
myBus.sendMessage(status_byte, channel, byte1, byte2);
```

Code Listing 8.3.1. MIDI_Output_01.pde

```
// MIDI_Output_01.pde
// As of this writing, Beads doesn't include functions for
// MIDI output. Hence, this example doesn't relate to Beads,
// it's simply a demonstration of how to use The MIDI Bus to
// send MIDI messages. It's based on the Basic.pde example
// from The MIDI Bus.


// Import the midibus library
import themidibus.*;
// declare The MidiBus
MidiBus myBus;


void setup()
{
  size(600, 400);
  background(0);


  // Create a new MidiBus with no input device and the
  // default Java Sound Synthesizer as the output device.
  myBus = new MidiBus(this, -1, "Java Sound Synthesizer");

  background(0); // set the background to black
  text("This program plays random MIDI notes using the Java
                                Sound Synthesizer.", 100, 100);
}


void draw()
{
  int channel = 0;
  int pitch = 48 + (int)random(48);
  int velocity = 64 + (int)random(64);

  // THIS BIT OF CODE PLAYS A MIDI NOTE
  // start a midi pitch
  myBus.sendNoteOn(channel, pitch, velocity);
  // wait for 100ms
  delay(100);
  // then stop the note we just started
```

```
  myBus.sendNoteOff(channel, pitch, velocity);

  // THIS BIT OF CODE CHANGES THE MIDI INSTRUMENT
  // This is the status byte for a program change
  int status_byte = 0xC0;
  // This will be the preset you are sending with your
  // program  change.
  int byte1 = (int)random(128);
  // This is not used for program change so ignore it and set
  // it to 0
  int byte2 = 0;
  //Send the custom message
  myBus.sendMessage(status_byte, channel, byte1, byte2);


  // we could control pitch bend and other parameters using
  // this call
  //int number = 0;
  //int value = 90;
  // Send a controllerChange
  //myBus.sendControllerChange(channel, number, value);

  // wait for a random amount of time less than 400ms
  delay((int)random(400));
}
```

9. Analysis

The primary focus of this tutorial is using Beads to generate sound with the intent of adding sound into a pre-existing Processing sketch; however, there are times when you may want to do the reverse. Rather than generating sound based on visuals, you may want to generate visuals based on sound. Analysis is the act of extracting meaningful information, and this chapter will demonstrate how we can use Beads to extract meaningful information from audio data.

9.1. Audio Analysis with Beads

Analysis programs are executed in three basic steps. First, the incoming audio is segmented into short chunks. This is necessary because many analysis algorithms are impractical or impossible to execute on large or continuous sections of audio. After it is segmented, the audio is passed through a series of analysis unit generators which modify the data and change the data into a usable format. Finally, the salient information, or features, are extracted from the results.

The first step is executed by a unit generator called the ShortFrameSegmenter. The ShortFrameSegmenter is the start of the analysis chain in all of the analysis examples in this chapter. The ShortFrameSegmenter is instantiated and connected like many of the unit generators used for sound generation. The constructor takes the AudioContext as a parameter, then we simply connect an input to send audio to it.

```
ShortFrameSegmenter sfs = new ShortFrameSegmenter(ac);
sfs.addInput(ac.out);
```

To connect other unit generators to the ShortFrameSegmenter, we can call the addListener function. This is slightly different from using the addInput routine that is used in most of the rest of the Beads Library.

```
// connect the FFT object to the ShortFrameSegmenter
sfs.addListener(fft);
```

Finally, the AudioContext must be told when to update the ShortFrameSegmenter. This is accomplished by adding the ShortFrameSegmenter as a dependent.

```
ac.out.addDependent(sfs);
```

9.2. Fast Fourier Transform (FFT_01)

When we think of a sound wave, we usually envision something like a sine wave. A smoothly undulating curve drawn on the face of an oscilloscope or a computer screen. These sort of graphs are called waveforms. Waveforms are a representation of sound pressure variation over time, or voltage variation over time. Waveforms allow us to easily see the amplitude of a sound, but it's more difficult to deduce frequency information from anything anything other than the most simple of waveforms.

The Fourier Transform allows us to transform a time-domain waveform into a frequency-domain spectrum. In other words, it shows us what frequencies are present in a sound, thereby revealing a more clear picture of a sound's timbre.

In the early 1800s, Jean Baptiste Joseph Fourier was studying the flow of heat through metals. Specifically, he was tasked with finding a solution to Napoleon's overheating cannons, but his theoretical work has impacted virtually every branch of mathematics, science and engineering. Fourier showed that a complex periodic waveform, such as those describing the flow of energy through a body, can be broken down into a sum of many sine waves. He showed that through a complex calculation, we can calculate how much each sine wave contributes to the final waveform.

This has obvious applications in electronic music. Every sound is made up of many frequencies, and the Fourier Transform allows us to see those frequencies.

Today, we calculate the Fourier Transform using a class of algorithms known as Fast Fourier Transforms (FFT). In Beads, we can use the FFT object to get the spectrum of a sound. This example, which is based on code from the Beads website, uses the FFT to paint the spectrum of a sound on screen.

We start by setting up a ShortFrameSegmenter object, which breaks the incoming audio into discrete chunks.

```
ShortFrameSegmenter sfs = new ShortFrameSegmenter(ac);
sfs.addInput(ac.out);
```

Then we initialize our FFT. This is where we see a subtle difference in the way that the Beads analysis objects are used. Rather than connecting objects using the addInput function, we use the addListener function to tell one analysis object to take data from another.

```
FFT fft = new FFT();
sfs.addListener(fft);
```

Then we connect a PowerSpectrum object. This simply forwards the part of the FFT that we are interested in.

```
ps = new PowerSpectrum();
fft.addListener(ps);
```

Finally, we tell the AudioContext that it has to monitor and update the ShortFrameSegmenter.

```
ac.out.addDependent(sfs);
```

In the draw function, we interpret the results of the signal chain and paint them on screen. First we get the results, the features, from the PowerSpectrum object.

```
float[] features = ps.getFeatures();
```

Then we loop through each x-coordinate in our window, and paint a vertical bar representing the frequency between 20Hz and 20kHz that corresponds to this location.

```
for(int x = 0; x < width; x++)
{
  int featureIndex = (x * features.length) / width;
  int barHeight = Math.min((int)(features[featureIndex] *
                                          height), height - 1);
  line(x, height, x, height - barHeight);
}
```

Code Listing 9.2. FFT_01.pde

```
// FFT_01.pde
// This example is based in part on an example included with
// the Beads download originally written by Beads creator
// Ollie Bown. It draws the frequency information for a
// sound on screen.

import beads.*;

AudioContext ac;
PowerSpectrum ps;

color fore = color(255, 255, 255);
color back = color(0,0,0);

void setup()
{
```

```
  size(600,600);

  ac = new AudioContext();

  // set up a master gain object
  Gain g = new Gain(ac, 2, 0.3);
  ac.out.addInput(g);

  // load up a sample included in code download
  SamplePlayer player = null;
  try
  {
    // Load up a new SamplePlayer using an included audio
    // file.
    player = new SamplePlayer(ac, new Sample(sketchPath("") +
                                        "Drum_Loop_01.wav"));
    // connect the SamplePlayer to the master Gain
    g.addInput(player);
  }
  catch(Exception e)
  {
    // If there is an error, print the steps that got us to
    // that error.
    e.printStackTrace();
  }

  // In this block of code, we build an analysis chain
  // the ShortFrameSegmenter breaks the audio into short,
  // discrete chunks.
  ShortFrameSegmenter sfs = new ShortFrameSegmenter(ac);
  sfs.addInput(ac.out);

  // FFT stands for Fast Fourier Transform
  // all you really need to know about the FFT is that it
  // lets you see what frequencies are present in a sound
  // the waveform we usually look at when we see a sound
  // displayed graphically is time domain sound data
  // the FFT transforms that into frequency domain data
  FFT fft = new FFT();
  // connect the FFT object to the ShortFrameSegmenter
  sfs.addListener(fft);

  // the PowerSpectrum pulls the Amplitude information from
  // the FFT calculation (essentially)
  ps = new PowerSpectrum();
  // connect the PowerSpectrum to the FFT
  fft.addListener(ps);

  // list the frame segmenter as a dependent, so that the
  // AudioContext knows when to update it.
  ac.out.addDependent(sfs);
  // start processing audio
  ac.start();
}

// In the draw routine, we will interpret the FFT results and
// draw them on screen.
```

```
void draw()
{
  background(back);
  stroke(fore);

  // The getFeatures() function is a key part of the Beads
  // analysis library. It returns an array of floats
  // how this array of floats is defined (1 dimension, 2
  // dimensions ... etc) is based on the calling unit
  // generator. In this case, the PowerSpectrum returns an
  // array with the power of 256 spectral bands.
  float[] features = ps.getFeatures();

  // if any features are returned
  if(features != null)
  {
    // for each x coordinate in the Processing window
    for(int x = 0; x < width; x++)
    {
      // figure out which featureIndex corresponds to this x-
      // position
      int featureIndex = (x * features.length) / width;

      // calculate the bar height for this feature
      int barHeight = Math.min((int)(features[featureIndex] *
                                        height), height - 1);

      // draw a vertical line corresponding to the frequency
      // represented by this x-position
      line(x, height, x, height - barHeight);
    }
  }
}
```

9.3. Frequency Analysis (Frequency_01 and Resynthesis_01)

In these two examples, we see how to calculate and respond to specific frequencies in incoming audio. The first example tries to guess the strongest frequency that is present in a sound, then set a sine wave to play that frequency. The second example does the same thing on a larger scale: it calculates the 32 strongest frequencies in a sound and tries to output sine waves at those frequencies.

If you just open up these samples and run them, using your microphone as input, you might notice that they don't work particularly well. It's important to remember how many variables effect this data. The sound data that is captured by your computer is effected by your microphone, where the microphone is located in relation to the sound source, the room that the sound occurs in, the other electrical signals in that room and the analog to digital converter in the computer. All of these factors can potentially introduce unwanted noise into a signal, not to mention extraneous background noise from your computer fan or other sources. So measuring incoming frequencies is a very tricky thing. I find that the first example works best if you whistle near the microphone. The second example works well on the sound of a speaking voice.

In the first example, we use the Frequency unit generator to try to pick the strongest frequency in an incoming signal. This object is instantiated with a single parameter, the sample rate of the audio. Then it is connected to the PowerSpectrum object. The Frequency object only returns one feature when the getFeatures function is called. This feature is its best guess at the strongest frequency in a signal.

```
float inputFrequency = f.getFeatures();
```

The second example switches out the Frequency object in favor of the SpectralPeaks object. The SpectralPeaks object locates a set of the strongest frequencies in a spectrum. In this case, we tell it to look for the 32 strongest peaks, then we try to follow those frequencies with sine waves. The end effect is a sort of low-fidelity vocoder.

The major difference between the Frequency object and the SpectralPeaks object is that the SpectralPeaks object returns a 2-dimensional array of values. The first dimension in the array specifies the peak number. The second dimension specifies frequency and amplitude. In this chunk of code, we get the 2-dimensional array and loop through it, setting frequencies and amplitudes based on the data.

```
    float[][] features = sp.getFeatures();
    for( int i = 0; i < numPeaks; i++ )
    {
      if(features[i][0] < 10000.0)
                  frequencyGlide[i].setValue(features[i][0]);

      if(features[i][1] > 0.01)
                     gainGlide[i].setValue(features[i][1]);
      else gainGlide[i].setValue(0.0);
    }
```

Code Listing 9.3. Frequency_01.pde and Resynthesis_01.pde

```
// Frequency_01.pde
```

```
// This example attempts to guess the strongest frequency in
// the signal that comes in via the microphone. Then it plays
// a sine wave at that frequency. Unfortunately, this doesn't
// work very well for singing, but it works quite well for
// whistling (in my testing).

import beads.*;

AudioContext ac;
PowerSpectrum ps;
Frequency f;

Glide frequencyGlide;
WavePlayer wp;
float meanFrequency = 400.0;

color fore = color(255, 255, 255);
color back = color(0,0,0);

void setup()
{
  size(600,600);

  // set up the parent AudioContext object
  ac = new AudioContext();

  // set up a master gain object
  Gain g = new Gain(ac, 2, 0.5);
  ac.out.addInput(g);

  // get a microphone input unit generator
  UGen microphoneIn = ac.getAudioInput();

  // set up the WavePlayer and the Glide that will control
  // its frequency
  frequencyGlide = new Glide(ac, 50, 10);
  wp = new WavePlayer(ac, frequencyGlide, Buffer.SINE);
  // connect the WavePlayer to the master gain
  g.addInput(wp);

  // In this block of code, we build an analysis chain
  // the ShortFrameSegmenter breaks the audio into short,
  // discrete chunks.
  ShortFrameSegmenter sfs = new ShortFrameSegmenter(ac);
  // connect the microphone input to the ShortFrameSegmenter
  sfs.addInput(microphoneIn);

  // the FFT transforms that into frequency domain data
  FFT fft = new FFT();
  // connect the ShortFramSegmenter object to the FFT
  sfs.addListener(fft);

  // The PowerSpectrum turns the raw FFT output into proper
  // audio data.
  ps = new PowerSpectrum();
  // connect the FFT to the PowerSpectrum
  fft.addListener(ps);
```

```
  // The Frequency object tries to guess the strongest
  // frequency for the incoming data. This is a tricky
  // calculation, as there are many frequencies in any real
  // world sound. Further, the incoming frequencies are
  // effected by the microphone being used, and the cables
  // and electronics that the signal flows through.
  f = new Frequency(44100.0f);
  // connect the PowerSpectrum to the Frequency object
  ps.addListener(f);

  // list the frame segmenter as a dependent, so that the
  // AudioContext knows when to update it
  ac.out.addDependent(sfs);
  ac.start(); // start processing audio
}

// In the draw routine, we will write the current frequency
// on the screen and set the frequency of our sine wave.
void draw()
{
  background(back);
  stroke(fore);
  // draw the average frequency on screen
  text(" Input Frequency: " + meanFrequency, 100, 100);

  // Get the data from the Frequency object. Only run this
  // 1/4  frames so that we don't overload the Glide object
  // with frequency changes.
  if( f.getFeatures() != null && random(1.0) > 0.75)
  {
    // get the data from the Frequency object
    float inputFrequency = f.getFeatures();

    // Only use frequency data that is under 3000Hz - this
    // will include all the fundamentals of most instruments
    // in other words, data over 3000Hz will usually be
    // erroneous (if we are using microphone input and
    // instrumental/vocal sounds)
    if( inputFrequency < 3000)
    {
      // store a running average
      meanFrequency = (0.4 * inputFrequency) +
                                      (0.6 * meanFrequency);
      // set the frequency stored in the Glide object
      frequencyGlide.setValue(meanFrequency);
    }
  }
}
```

```
// Resynthesis_01.pde
// This example resynthesizes a tone using additive synthesis
// and the SpectralPeaks object. The result should be a very
// simple, low-fidelity vocoder.

import beads.*;

// how many peaks to track and resynth
int numPeaks = 32;

AudioContext ac;
Gain masterGain;
PowerSpectrum ps;
SpectralPeaks sp;


Gain[] g;
Glide[] gainGlide;
Glide[] frequencyGlide;
WavePlayer[] wp;
float meanFrequency = 400.0;

color fore = color(255, 255, 255);
color back = color(0,0,0);

void setup()
{
  size(600,600);

  // set up the parent AudioContext object
  ac = new AudioContext();

  // set up a master gain object
  masterGain = new Gain(ac, 2, 0.5);
  ac.out.addInput(masterGain);

  // get a microphone input unit generator
  UGen microphoneIn = ac.getAudioInput();

  frequencyGlide = new Glide[numPeaks];
  wp = new WavePlayer[numPeaks];
  g = new Gain[numPeaks];
  gainGlide = new Glide[numPeaks];
  for( int i = 0; i < numPeaks; i++ )
  {
    // set up the WavePlayer and the Glides that will control
    // its frequency and gain
    frequencyGlide[i] = new Glide(ac, 440, 1);
    wp[i] = new WavePlayer(ac,
                           frequencyGlide[i],
                           Buffer.SINE);
    gainGlide[i] = new Glide(ac, 0.0, 1);
    g[i] = new Gain(ac, 1, gainGlide[i]);
    // connect the WavePlayer to the master gain
    g[i].addInput(wp[i]);
    masterGain.addInput(g[i]);
  }
```

```
  // in this block of code, we build an analysis chain
  // the ShortFrameSegmenter breaks the audio into short,
  // discrete chunks
  ShortFrameSegmenter sfs = new ShortFrameSegmenter(ac);
  // connect the microphone input to the ShortFrameSegmenter
  sfs.addInput(microphoneIn);

  // the FFT transforms that into frequency domain data
  FFT fft = new FFT();
  // connect the ShortFramSegmenter object to the FFT
  sfs.addListener(fft);

  // the PowerSpectrum turns the raw FFT output into proper
  // audio data
  ps = new PowerSpectrum();
  // connect the FFT to the PowerSpectrum
  fft.addListener(ps);

  // the SpectralPeaks object stores the N highest Peaks
  sp = new SpectralPeaks(ac, numPeaks);
  // connect the PowerSpectrum to the Frequency object
  ps.addListener(sp);

  // list the frame segmenter as a dependent, so that the
  // AudioContext knows when to update it
  ac.out.addDependent(sfs);
  // start processing audio
  ac.start();
}

// in the draw routine, we will write the current frequency
// on the screen and set the frequency of our sine wave
void draw()
{
  background(back);
  stroke(fore);
  text("Use the microphone to trigger resynthesis",
                                                  100, 100);

  // get the data from the SpectralPeaks object
  // only run this 1/4 frames so that we don't overload the
  // Glide object with frequency changes
  if( sp.getFeatures() != null && random(1.0) > 0.5)
  {
    // get the data from the SpectralPeaks object
    float[][] features = sp.getFeatures();
    for( int i = 0; i < numPeaks; i++ )
    {
      if(features[i][0] < 10000.0)
                frequencyGlide[i].setValue(features[i][0]);

      if(features[i][1] > 0.01)
                    gainGlide[i].setValue(features[i][1]);
      else gainGlide[i].setValue(0.0);
    }
  }
```

```
}
```

9.4. Beat Detection (Beat_Detection_01)

In the final analysis example, we look at how we can detect beats in an audio stream. We use an analysis chain to the one used in the previous examples; however, we end with SpectralDifference and PeakDetector unit generators.

Code Listing 9.4. Beat_Detection_01.pde

```
// Beat_Detection_01.pde
// This example is based in part on an example included with
// the Beads download originally written by Beads creator
// Ollie Bown. In this example we draw a shape on screen
// whenever a beat is detected.

import beads.*;

AudioContext ac;
PeakDetector beatDetector;

// In this example we detect onsets in the audio signal
// and pulse the screen when they occur. The brightness is
// controlled by  the following global variable. The draw()
// routine decreases it over time.
float brightness;

// tracks the time
int time;

void setup()
{
  size(300,300);
  time = millis();

  // set up the AudioContext and the master Gain object
  ac = new AudioContext();
  Gain g = new Gain(ac, 2, 0.2);
  ac.out.addInput(g);

  // load up a sample included in code download
  SamplePlayer player = null;
  try
  {
    // load up a new SamplePlayer using an included audio
    // file
    player = new SamplePlayer(ac, new Sample(sketchPath("") +
                                        "Drum_Loop_01.wav"));
    // connect the SamplePlayer to the master Gain
    g.addInput(player);
  }
  catch(Exception e)
  {
    // if there is an error, print the steps that got us to
```

```
    // that error
    e.printStackTrace();
  }

  // Set up the ShortFrameSegmenter. This class allows us to
  // break an audio stream into discrete chunks.
  ShortFrameSegmenter sfs = new ShortFrameSegmenter(ac);
  // how large is each chunk?
  sfs.setChunkSize(2048);
  sfs.setHopSize(441);
  // connect the sfs to the AudioContext
  sfs.addInput(ac.out);

  FFT fft = new FFT();
  PowerSpectrum ps = new PowerSpectrum();

  sfs.addListener(fft);
  fft.addListener(ps);

  // The SpectralDifference unit generator does exactly what
  // it sounds like. It calculates the difference between two
  // consecutive spectrums returned by an FFT (through a
  // PowerSpectrum object).
  SpectralDifference sd = new
                          SpectralDifference(ac.getSampleRate());
  ps.addListener(sd);

  // we will use the PeakDetector object to actually find our
  // beats
  beatDetector = new PeakDetector();
  sd.addListener(beatDetector);

  // the threshold is the gain level that will trigger the
  // beat detector - this will vary on each recording
  beatDetector.setThreshold(0.2f);
  beatDetector.setAlpha(.9f);

  // whenever our beat detector finds a beat, set a global
  // variable
  beatDetector.addMessageListener
  (
    new Bead()
    {
      protected void messageReceived(Bead b)
      {
        brightness = 1.0;
      }
    }
  );

  // tell the AudioContext that it needs to update the
  // ShortFrameSegmenter
  ac.out.addDependent(sfs);
  // start working with audio data
  ac.start();
}
```

```
// the draw method draws a shape on screen whwenever a beat
// is detected
void draw()
{
  background(0);
  fill(brightness*255);
  ellipse(width/2,height/2,width/2,height/2);

  // decrease brightness over time
  int dt = millis() - time;
  brightness -= (dt * 0.01);
  if (brightness < 0) brightness = 0;
  time += dt;

  // set threshold and alpha to the mouse position
  beatDetector.setThreshold((float)mouseX/width);
  beatDetector.setAlpha((float)mouseY/height);
}
```

10. Miscellaneous

This chapter reviews some important unit generators that didn't fit into the earlier sections.

10.1. Clock (Clock_01)

The Clock unit generator is used to generate evenly spaced events, such as beats in a piece of music. It is useful any time you want to synchronize audio to a regular tick. The constructor takes two arguments: the AudioContext and the duration between beats in milliseconds.

```
beatClock = new Clock(ac, 1000);
```

Then we set the number of ticks that occur in each beat, and we remind the AudioContext to update the clock by adding it as a dependent.

```
beatClock.setTicksPerBeat(4);
ac.out.addDependent(beatClock);
```

Finally, we handle the tick events. In this case, we create a new Bead that acts as a tick event handler. The event handler creates a new synthesized tone each time it receives a tick.

```
Bead noteGenerator = new Bead () {
  public void messageReceived(Bead message)
  {
      synthNotes.add(new Synth(20 + (int)random(88)));
  }
};
beatClock.addMessageListener(noteGenerator);
```

Code Listing 10.1. Clock_01.pde

```
// Clock_01.pde
// This example builds a simple midi synthesizer
// then we trigger random notes using the Clock class.

import beads.*;

// the Beads AudioContext that will oversee audio production
// and output
AudioContext ac;
// our master gain object
Gain MasterGain;

// this ArrayList will hold synthesizer objects
// we will instantiate a new synthesizer object for each note
ArrayList synthNotes = null;
```

```
// our clock object will control timing
Clock beatClock = null;

void setup()
{
  size(600,400);
  background(0);

  synthNotes = new ArrayList();

  ac = new AudioContext();

  beatClock = new Clock(ac, 1000);
  beatClock.setTicksPerBeat(4);
  // this tell the AudioContext when to update the Clock
  ac.out.addDependent(beatClock);

  Bead noteGenerator = new Bead () {
    public void messageReceived(Bead message)
    {
        synthNotes.add(new Synth(20 + (int)random(88)));
    }
  };
  beatClock.addMessageListener(noteGenerator);

  MasterGain = new Gain(ac, 1, 0.3);
  ac.out.addInput(MasterGain);
  ac.start();

  // set the background to black
  background(0);
  // tell the user what to do!
  text("This program generates randomized synth notes.",
                                                    100, 100);
}

void draw()
{
  for( int i = 0; i < synthNotes.size(); i++ )
  {
    Synth s = (Synth)synthNotes.get(i);
    if( s.g.isDeleted() )
    {
      // then remove the parent synth
      synthNotes.remove(i);
      s = null;
    }
  }
}

// this is our synthesizer object
class Synth
{
  public WavePlayer carrier = null;
  public WavePlayer modulator = null;
  public Envelope e = null;
  public Gain g = null;
```

```
  public int pitch = -1;

  public boolean alive = true;

  // our filter and filter envelope
  LPRezFilter lowPassFilter;
  WavePlayer filterLFO;

  Synth(int midiPitch)
  {
    // get the midi pitch and create a couple holders for the
    // midi pitch
    pitch = midiPitch;
    float fundamentalFrequency = 440.0 * pow(2,
                              ((float)midiPitch - 59.0)/12.0);
    Static ff = new Static(ac, fundamentalFrequency);

    // instantiate the modulator WavePlayer
    modulator = new WavePlayer(ac,
                               0.5 * fundamentalFrequency,
                               Buffer.SINE);
   // create our frequency modulation function
   Function frequencyModulation = new Function(modulator, ff)
   {
      public float calculate() {
        // the x[1] here is the value of a sine wave
        // oscillating at the fundamental frequency
        return (x[0] * 1000.0) + x[1];
      }
    };
    // instantiate the carrier WavePlayer
    carrier = new WavePlayer(ac,
                             frequencyModulation,
                             Buffer.SINE);

    // set up the filter and LFO (randomized LFO frequency)
    filterLFO = new WavePlayer(ac,
                             1.0 + random(100), Buffer.SINE);
    Function filterCutoff = new Function(filterLFO)
    {
      public float calculate() {
        // set the filter cutoff to oscillate between 1500Hz
        // and 2500Hz
        return ((x[0] * 500.0) + 2000.0);
      }
    };
    lowPassFilter = new LPRezFilter(ac, filterCutoff, 0.96);
    lowPassFilter.addInput(carrier);

    // set up and connect the gains
    e = new Envelope(ac, 0.0);
    g = new Gain(ac, 1, e);
    g.addInput(lowPassFilter);
    MasterGain.addInput(g);

    // create a randomized Gain envelope for this note
```

```
    e.addSegment(0.5, 10 + (int)random(500));
    e.addSegment(0.4, 10 + (int)random(500));
    e.addSegment(0.0, 10 + (int)random(500),
                                          new KillTrigger(g));
  }
  public void destroyMe()
  {
    carrier.kill();
    modulator.kill();
    lowPassFilter.kill();
    filterLFO.kill();
    e.kill();
    g.kill();
    carrier = null;
    modulator = null;
    lowPassFilter = null;
    filterLFO = null;
    e = null; g = null;
    alive = false;
  }
}
```

Appendix A. Custom Beads

In this section, we take a look at how to extend the functionality of the Beads library in Processing. It should be noted, however, that Beads already contains most of the functionality that most users are going to need. This section is aimed at advanced programmers who are comfortable with java, and with the basic concepts of digital signal processing.

A.1. Custom Functions

Most of the time when you want to do something that isn't encapsulated by a pre-made unit generator, you can create that functionality using a custom function. Custom functions are handy little classes that can be embedded right into a Processing/Beads program. We have already encountered custom functions a number of times in this tutorial.

Frequency Modulation

The first custom function we encountered was the frequency modulation function that we created in the example Frequency_Modulation_01. In FM synthesis, one sine wave is used to control the frequency of another sine wave. The sine function, however, oscillates around 0, outputting values between -1.0 and 1.0. To produce sidebands, we need the modulator to oscillate around a value in the audible range. In this first custom function, we take a WavePlayer as input, then we do the math to output values within the range of audible frequencies.

```
Function frequencyModulation = new Function(modulator)
{
  public float calculate() {
    return (x[0] * 50.0) + 200.0;
  }
};
```

Notice the parameter in the constructor. This gives the custom function access to the value of the modulator. Whatever unit generators are passed into the constructor can be accessed by the array x.

```
new Function(modulator)
```

Then notice that the only the calculate function is actually implemented. We get the value from the modulator by accessing x[0], then we multiply it by 50. At this point, the output will oscillate between -50.0 and 50.0. To bring that higher into the audible range, we add 200. So the output oscillates between 150.0 and 250.0 at the speed specified by the frequency of the modulator WavePlayer.

```
return (x[0] * 50.0) + 200.0;
```

Ring Modulation

In the same chapter, we encountered a slightly more complex use of the Function object. This time, the Function takes two unit generators as parameters, and simply outputs the product of their values.

```
Function ringModulation = new Function(carrier, modulator)
{
  public float calculate() {
    return x[0] * x[1];
  }
};
```

Notice how the values of the parameters are accessed within the calculate routine. Since the carrier UGen was passed in first, its value is accessed via x[0]. Since the modulator UGen was passed in second, its value is accessed via x[1]. As you can see in this example as well as the previous example, custom functions are useful any time you want to encapsulate a bit of math.

A.1.1. Custom Mean Filter (Custom_Function_01)

In this example, we show how you can easily create a new signal processing unit generator using the Function object. In this case, we create a simple mean filter unit generator.

A mean filter is one of the simplest filters. It simply averages the last few frames of data and outputs the result. This has an effect similar to a low-pass filter, with more low frequencies filtered out when more frames are averaged. In this example, we average the last four frames.

```
Function meanFilter = new Function(sp)
{
  float[] previousValues = new float[3];
  public float calculate() {
    float mean = 0.25 * (previousValues[0] +
                         previousValues[1] +
                         previousValues[2] +
                         x[0]);
    previousValues[2] = previousValues[1];
    previousValues[1] = previousValues[0];
```

```
      previousValues[0] = x[0];
      return mean;
    }
  };
```

Notice how we connect another unit generator to the custom function. Rather than using the addInput method, we pass unit generators in via the Function constructor. Later, when we want to connect the custom function to another unit generator, we can return to the addInput method.

```
g.addInput(meanFilter);
```

Code Listing A.1.1. Custom_Function_01.pde

```
// Custom_Function_01.pde
// in this example, we create a custom function that
// calculates a mean filter


import beads.*;


AudioContext ac;
// this will hold the path to our audio file
String sourceFile;
// the SamplePlayer class will be used to play the audio file
SamplePlayer sp;


// standard gain objects
Gain g;
Glide gainValue;


void setup()
{
  size(800, 600);

  // create our AudioContext
  ac = new AudioContext();


  sourceFile = sketchPath("") + "Drum_Loop_01.wav";


  // Try/Catch blocks will inform us of errors
  try {
    sp = new SamplePlayer(ac, new Sample(sourceFile));
   }
  catch(Exception e)
  {
    println("Exception while attempting to load sample!");
    e.printStackTrace();
    exit();
  }
```

```
  // we would like to play the sample multiple times, so we
  // set KillOnEnd to false
  sp.setKillOnEnd(false);


  // this custom function calculates a mean filter using the
  // previous 3 values
  Function meanFilter = new Function(sp)
  {
    float[] previousValues = new float[3];
    public float calculate() {
      float mean = 0.25 * (previousValues[0] +
                           previousValues[1] +
                           previousValues[2] +
                           x[0]);
      previousValues[2] = previousValues[1];
      previousValues[1] = previousValues[0];
      previousValues[0] = x[0];
      return mean;
    }
  };


  // as usual, we create a gain that will control the volume
  // of our sample player
  gainValue = new Glide(ac, 0.0, 20);
  g = new Gain(ac, 1, gainValue);
  // connect the filter to the gain
  g.addInput(meanFilter);


  // connect the Gain to the AudioContext
  ac.out.addInput(g);


  // begin audio processing
  ac.start();

  background(0);
  text("Click to hear a mean filter applied to a drum loop.",
                                                    50, 50);
}


void draw(){}


// this routine is called whenever a mouse button is pressed
// on the Processing sketch
void mousePressed()
{
  gainValue.setValue(0.9);
  // move the playback pointer to the first loop point (0.0)
  sp.setToLoopStart();
  sp.start();
}
```

A.2. Custom Beads

In most situations, a custom function can be used for extending the functionality of Beads. In fact, it's difficult for me to think of a situation where creating an entirely new unit generator (Bead) is necessary. But it is possible, and in fact it's not even very difficult (if you are already comfortable in java). In this section, we're going to create some new Bead objects for use withing a Processing sketch.

A.2.1. Custom Buffer (Custom_Beads_01)

In the 1970s, James Moorer described a way to simplify additive synthesis on a digital system. Using Moorer's equation, we can create an additive tone by using a single oscillator and a special buffer, as opposed to traditional methods, which employ many oscillators and simple sine waves. Moorer's discrete summation equation shows us how to create a complex buffer that is the sum of a series of sine waves. In this example, we create a new buffer type that generates discrete summation buffers based on a number of parameters.

When you want to create a new bead, you probably shouldn't start from scratch. It's best to find the piece of the source code that most closely matches the functionality you are trying to create, then build off of that. The source code can be found at http://www.beadsproject.net/svn/beads/Trunk/. If you want to check out an entire copy of the source, you can use the program SVN (http://subversion.tigris.org/), but if you just want to create a handful of new unit generators for use in Processing, then that is overkill. In this case, we want to build a new type of Buffer that can be used by a WavePlayer, so we will start with the Buffer.SINE class as our base. It can be found at http://www.beadsproject.net/svn/beads/Trunk/Beads/src/beads_main/net/beadsproject/beads/data/buffers/SineBuffer.java

After downloading the file, put it in the same directory as the Processing program in which you want to use the new class. Then rename the file, the class and any constructors. Rename them to something that is verbose and captures the purpose of your unit generator. Then make sure you change the name anywhere it is used within the code, as in the constructors.

When you want to create a new unit generator for use in Processing, the import statements should look like the ones you use in processing. In this case, we need to import the Buffer and the BufferFactory class. We could do that by importing beads.*, or by just importing the classes we need. Change the import statements to point to your beads installation.

```
//import net.beadsproject.beads.data.Buffer;
//import net.beadsproject.beads.data.BufferFactory;
```

```
import beads.Buffer;
import beads.BufferFactory;
```

The biggest task in writing a new Bead is in overriding the functions that actually do the work. In many cases, you will want to have a DSP equation in mind before you start working, although your unit generator could simply be a convenience UGen that encompasses a common part of your Beads programs. Usually, the meaty code will be contained within the calculateBuffer function, but in the case of a buffer, it is in the generateBuffer routine.

In this case, we implement Moorer's discrete summation equation, which is beyond the scope of this tutorial.

```
public Buffer generateBuffer(int bufferSize, int
numberOfHarmonics, float amplitude)
{
  Buffer b = new Buffer(bufferSize);

  double amplitudeCoefficient = amplitude / (2.0 *
                                  (double)numberOfHarmonics);
  double theta = 0.0;
  double delta = (double)(2.0 * Math.PI) /
                                  (double)b.buf.length;

  …etc (SEE CODE LISTING)

  return b;
}
```

In Custom_Beads_01.pde, we put the new Buffer into use by instantiating it in the WavePlayer constructor.

```
wavetableSynthesizer = new WavePlayer(ac,
        frequencyGlide,
        new DiscreteSummationBuffer().generateBuffer(44100));
```

Code Listing A.2.1. Custom_Beads_01.pde and DiscreteSummationBuffer.pde

```
// Custom_Beads_01.pde
// this program demonstrates how to create and use custom
// Beads objects in Processing

import beads.*; // import the beads library
AudioContext ac; // create our AudioContext

// declare our unit generators
WavePlayer wavetableSynthesizer;
Glide frequencyGlide;

// our envelope and gain objects
Envelope gainEnvelope;
```

```
Gain synthGain;

void setup()
{
  size(800, 600);

  // initialize our AudioContext
  ac = new AudioContext();

  // create our Buffer
  Buffer wavetable = new
     DiscreteSummationBuffer().generateBuffer(4096, 15, 0.8);

  frequencyGlide = new Glide(ac, 200, 10);
  wavetableSynthesizer = new WavePlayer(ac,
        frequencyGlide,
        new DiscreteSummationBuffer().generateBuffer(44100));

  // create the envelope object that will control the gain
  gainEnvelope = new Envelope(ac, 0.0);
  // create a Gain object, connect it to the gain envelope
  synthGain = new Gain(ac, 1, gainEnvelope);
  // connect the synthesizer to the gain
  synthGain.addInput(wavetableSynthesizer);
  // connect the Gain output to the AudioContext
  ac.out.addInput(synthGain);

  // start audio processing
  ac.start();

  background(0);
  drawBuffer(wavetable.buf);
  text("Click to trigger the wavetable synthesizer.",
                                                    100, 120);
}

void draw()
{
  // set the fundamental frequency
  frequencyGlide.setValue(mouseX);
}

// this routine is triggered whenever a mouse button
// is pressed
void mousePressed()
{
  // when the mouse button is pressed, at a 50ms attack
  // segment to the envelope
  // and a 300 ms decay segment to the envelope
  gainEnvelope.addSegment(0.8, 50); // over 50 ms rise to 0.8
  gainEnvelope.addSegment(0.0, 200); // in 300ms go to 0.0
}

// draw a buffer on screen
void drawBuffer(float[] buffer)
{
  float currentIndex = 0.0;
```

```
  float stepSize = buffer.length / (float)width;

  color c = color(255, 0, 0);
  stroke(c);

  for( int i = 0; i < width; i++ )
  {
    set(i, (int)(buffer[(int)currentIndex] *
                  (height / 2.0)) + (int)(height / 2.0), c);
    currentIndex += stepSize;
  }
}




// DiscreteSummationBuffer.pde
// This is a custom Buffer class that implements the Discrete
// Summation equations as outlines by Moorer, with the Dodge
// & Jerse modification.

// if you want this code to compile within the Beads source,
// then you would use these includes
//import net.beadsproject.beads.data.Buffer;
//import net.beadsproject.beads.data.BufferFactory;

import beads.Buffer;
import beads.BufferFactory;

public class DiscreteSummationBuffer extends BufferFactory
{
  // this is the generic form of generateBuffer that is
  // required by Beads
  public Buffer generateBuffer(int bufferSize)
  {
    // are these good default values?
    return generateBuffer(bufferSize, 10, 0.9f);
  }
  // this is a fun version of generateBuffer that will allow
  // us to really employ the Discrete Summation equation
  public Buffer generateBuffer(int bufferSize,
                               int numberOfHarmonics,
                               float amplitude)
  {
    Buffer b = new Buffer(bufferSize);

    double amplitudeCoefficient = amplitude /
                          (2.0 * (double)numberOfHarmonics);
    double theta = 0.0;
    double delta = (double)(2.0 * Math.PI) /
                                        (double)b.buf.length;
    for( int j = 0; j < b.buf.length; j++ )
    {
      // increment theta
      // we do this first, because the discrete summation
      // equation runs from 1 to n, not from 0 to n-1 ...this
```

```
      // is important
      theta += delta;

      // do the math with double precision (64-bit) then cast
      // to a float (32-bit) ... this is probably unnecessary
      // if we want to worry about memory (nom nom nom)
      double numerator = (double)Math.sin( (double)(theta /
          2.0) * ((2.0 * (double)numberOfHarmonics) + 1.0) );
      double denominator = (double)Math.sin( theta / 2.0);
      float newValue = (float)(amplitudeCoefficient *
                          ((numerator / denominator) - 1.0));

      // set the value for the new buffer
      b.buf[j] = newValue;
    }
    return b;
  }

  // we must implement this method when we inherit from
  // BufferFactory
  public String getName() {
    return "DiscreteSummation";
  }
};
```

A.2.2. Custom WavePlayer (Custom_Beads_01)

In this example, we modify the WavePlayer object so that It can morph between two buffers. Any WavePlayer-like unit generator should start with the WavePlayer source code, which can be found at http://www.beadsproject.net/svn/beads/Trunk/Beads/src/beads_main/net/beadsproject/beads/ugens/WavePlayer.java.

Creating the MorphingWavePlayer class is slightly more difficult than creating a new type of buffer. In this case, we want to add new member variables that will control the mixing between the two buffers, and we want to give other programs access to the new variables. The three new variables we created are the mixEnvelope UGen, the mix float, and the boolean isMixStatic. The UGen can be used to control the mix dynamically. The float is used to pin the mix to a static value. IsMixStatic indicates whether or not the mix is a static value.

```
private UGen mixEnvelope;
private float mix;
private boolean isMixStatic = true;
```

Then we need to create accessor functions to get and set the new variables. I used other get and set routines as the basis for these functions. In this example we also wrote new constructors to set the mix variables when the object is created, but this is purely for convenience.

```
  public UGen getMixUGen() {
```

```
    if (isMixStatic) {
      return null;
    } else {
      return mixEnvelope;
    }
  }
  public float getMix() {
    return mix;
  }

  // these two routines give access to the mix parameter via
  // float or UGen
  public MorphingWavePlayer setMix(UGen mixUGen) {
    if (mixUGen == null) {
      setMix(mix);
    } else {
      this.mixEnvelope = mixUGen;
      isMixStatic = false;
    }
    return this;
  }
  public MorphingWavePlayer setMix(float newMix) {
    if (isMixStatic) {
      ((beads.Static) mixEnvelope).setValue(newMix);
    } else {
      mixEnvelope = new beads.Static(context, newMix);
      isMixStatic = true;
    }
    this.mix = newMix;
    return this;
  }
```

Finally, we want to override the calculateBuffer function with our timbre morphing code. Morphing between buffers is relatively easy; we just multiply buffer1 by the mix level, then multiply buffer2 by 1.0 minus the mix level. Then we sum those values to produce the output.

In Custom_Beads_02.pde you can see that the new MorphingWavePlayer is constructed just like the WavePlayer object, except we give it two buffers, rather than just one.

```
mwp = new MorphingWavePlayer(ac,
                             frequencyGlide,
                             Buffer.SINE,
                             Buffer.TRIANGLE);
```

Code Listing A.2.2. Custom_Beads_01.pde and MorphingWavePlayer.pde

```
// Custom_Beads_02.pde

import beads.*;
AudioContext ac;

// declare our unit generators
```

```
MorphingWavePlayer mwp;
Glide frequencyGlide;
Glide mixGlide;

Gain masterGain;

void setup()
{
  size(800, 600);

  // initialize our AudioContext
  ac = new AudioContext();

  frequencyGlide = new Glide(ac, 200, 10);
  mixGlide = new Glide(ac, 0.5, 10);
  mwp = new MorphingWavePlayer(ac,
                               frequencyGlide,
                               Buffer.SINE,
                               Buffer.TRIANGLE);
  mwp.setMix(mixGlide);

  masterGain = new Gain(ac, 1, 0.8);
  masterGain.addInput(mwp);
  ac.out.addInput(masterGain);

  // start audio processing
  ac.start();

  background(0);
  text("Move the mouse to set the frequency and the mix of
                        the MorphingWavePlayer.", 100, 120);
}

void draw()
{
  frequencyGlide.setValue(mouseX);
  mixGlide.setValue(mouseY/(float)height);
}

// MorphingWavePlayer.pde
// this file demonstrates the creation of custom beads for
// use in Processing
// it expands upon the standard WavePlayer by allowing the
// programmer to mix between two buffers

import beads.AudioContext;
import beads.UGen;
import beads.Buffer;
import beads.SawBuffer;
import beads.SineBuffer;
import beads.SquareBuffer;

public class MorphingWavePlayer extends UGen
{
  private double phase;
  private UGen frequencyEnvelope;
  private UGen phaseEnvelope;
```

```
// the buffers that will be mixed
private Buffer buffer1;
private Buffer buffer2;

// the unit generators that will control the mixing
private UGen mixEnvelope;
private float mix;
private boolean isMixStatic = true;

// The oscillation frequency.
private float frequency;
private boolean isFreqStatic;

// To store the inverse of the sampling frequency.
private float one_over_sr;

// constructors
private MorphingWavePlayer(AudioContext context,
                           Buffer newBuffer1,
                           Buffer newBuffer2)
{
  super(context, 1);
  this.buffer1 = newBuffer1;
  this.buffer2 = newBuffer2;
  mix = 0.5f;
  phase = 0;
  one_over_sr = 1f / context.getSampleRate();
}
public MorphingWavePlayer(AudioContext context,
                          UGen frequencyController,
                          Buffer newBuffer1,
                          Buffer newBuffer2) {
  this(context, newBuffer1, newBuffer2);
  setFrequency(frequencyController);
}
public MorphingWavePlayer(AudioContext context,
                          float frequency,
                          Buffer newBuffer1,
                          Buffer newBuffer2) {
  this(context, newBuffer1, newBuffer2);
  setFrequency(frequency);
}

public void start() {
  super.start();
  phase = 0;
}

// this is the key to this object
// overriding the calculateBuffer routine allows us to
// calculate new audio data and pass it back to the calling
// program
@Override public void calculateBuffer()
{
  frequencyEnvelope.update();
```

```
    if( mixEnvelope != null ) mixEnvelope.update();
    float inverseMix = 1.0f - mix;

    float[] bo = bufOut[0];
    if (phaseEnvelope == null)
    {
      for (int i = 0; i < bufferSize; i++)
      {
        frequency = frequencyEnvelope.getValue(0, i);
        if( mixEnvelope != null )
        {
          mix = mixEnvelope.getValue(0, i);
          inverseMix = 1.0f - mix;
        }
        phase = (((phase + frequency * one_over_sr) % 1.0f) +
                                                    1.0f) % 1.0f;
       bo[i] =
       (mix * buffer1.getValueFraction((float) phase)) +
       (inverseMix * buffer2.getValueFraction((float)phase));
      }
    }
    else
    {
      phaseEnvelope.update();

      for (int i = 0; i < bufferSize; i++)
      {
        if( mixEnvelope != null )
        {
          mix = mixEnvelope.getValue(0, i);
          inverseMix = 1.0f - mix;
        }
        bo[i] = (mix *
        buffer1.getValueFraction(phaseEnvelope.getValue(0,
        i))) + (inverseMix *
        buffer2.getValueFraction(phaseEnvelope.getValue(0,
        i)));
      }
    }
  }

  @Deprecated
  public UGen getFrequencyEnvelope() {
    return frequencyEnvelope;
  }
  public UGen getFrequencyUGen() {
    if (isFreqStatic) {
      return null;
    } else {
      return frequencyEnvelope;
    }
  }
  public float getFrequency() {
    return frequency;
  }

  @Deprecated
```

```
public void setFrequencyEnvelope(UGen frequencyEnvelope) {
  setFrequency(frequencyEnvelope);
}
public MorphingWavePlayer setFrequency(UGen frequencyUGen)
{
  if (frequencyUGen == null) {
    setFrequency(frequency);
  } else {
    this.frequencyEnvelope = frequencyUGen;
    isFreqStatic = false;
  }
  return this;
}
public MorphingWavePlayer setFrequency(float frequency) {
  if (isFreqStatic) {
    ((beads.Static) frequencyEnvelope).setValue(frequency);
  } else {
    frequencyEnvelope = new beads.Static(context,
                                         frequency);
    isFreqStatic = true;
  }
  this.frequency = frequency;
  return this;
}

// these two routines control access to the mix parameter
public UGen getMixUGen() {
  if (isMixStatic) {
    return null;
  } else {
    return mixEnvelope;
  }
}
public float getMix() {
  return mix;
}

// these two routines give access to the mix parameter
//via float or UGen
public MorphingWavePlayer setMix(UGen mixUGen) {
  if (mixUGen == null) {
    setMix(mix);
  } else {
    this.mixEnvelope = mixUGen;
    isMixStatic = false;
  }
  return this;
}
public MorphingWavePlayer setMix(float newMix) {
  if (isMixStatic) {
    ((beads.Static) mixEnvelope).setValue(newMix);
  } else {
    mixEnvelope = new beads.Static(context, newMix);
    isMixStatic = true;
  }
  this.mix = newMix;
  return this;
```

```
  }

  @Deprecated
  public UGen getPhaseEnvelope() {
    return phaseEnvelope;
  }
  public UGen getPhaseUGen() {
    return phaseEnvelope;
  }
  public float getPhase() {
    return (float) phase;
  }

  @Deprecated
  public void setPhaseEnvelope(UGen phaseEnvelope) {
    setPhase(phaseEnvelope);
  }
  public MorphingWavePlayer setPhase(UGen phaseController) {
    this.phaseEnvelope = phaseController;
    if (phaseController != null) {
      phase = phaseController.getValue();
    }
    return this;
  }
  public MorphingWavePlayer setPhase(float phase) {
    this.phase = phase;
    this.phaseEnvelope = null;
    return this;
  }

  // GET / SET BUFFER1
  public MorphingWavePlayer setBuffer1(Buffer b) {
    this.buffer1 = b;
    return this;
  }
  public Buffer getBuffer1() {
    return this.buffer1;
  }

  // get / set buffer2
  public MorphingWavePlayer setBuffer2(Buffer b) {
    this.buffer2 = b;
    return this;
  }
  public Buffer getBuffer2() {
    return this.buffer2;
  }
}
```

About the Author

Evan X. Merz is a graduate student in the algorithmic composition program at The University of California at Santa Cruz. He earned a bachelor's degree in computer science from the University of Rochester in 2004, and a master's degree in computer music from Northern Illinois University in 2010. His music has been performed in *Phono Photo No. 6*, *Silence, Beauty and Horror 2009*, *musicBYTES 2009*, *New Music Hartford*, and *IMMArts TechArt 2008*. His primary interest is biological and bottom-up approaches to computer-assisted composition, which he explores in new software written in java and processing. Evan works heavily as a freelance composer, scoring for numerous videogames and television productions. He is also the SEAMUS Webmaster and the blogger at computermusicblog.com.