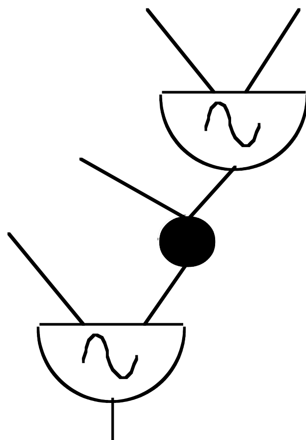
Java Music Systems, Didkovsky
JSyn instrument design

Abstract

This lecture will demonstrate the design of two JSyn instruments: a Chowning FM Pair and Risset's Bell. An FM Pair is an example of "synthesis using distortion techniques" (Dodge/Jerse). Risset's Bell is an example of additive synthesis. You will see, or you may already know, that additive synthesis has the advantage of being able to specify complex time varying spectra directly, at the expense of using lots and lots of oscillators to do so. An FM pair just needs two oscillators and one adder, while Risset's bell uses eleven oscillators, eleven envelopes, and a bus to combine these signals into one. While more efficient, an FM Pair is less intuitively controllable.

Signal Flowcharts

JSyn supports the unit generator paradigm of synthesis design. A common way of representing such an instrument is graphically, as a Signal Flowchart. A sine oscillator is represented as a semicircle with a sine shape drawn into it. An adder is represented by a circle with a "+" sign in it. The Signal Flowchart of a simple Chowning FM Pair is shown in the diagram below.



Simple FM Pair, diagram from R. Rowe's lecture notes.

It is extremely valuable to become familiar with this representation of instruments, as it is truly "cross platform". A JSyn instrument designer can discuss FM pairs with a Csound instrument designer using this representation, for example.

Before we go any further let me reiterate how important it is to have a good computer music reference book on your bookshelves. A book like Charles Dodge / Thomas Jerse's <u>Computer Music, Synthesis Composition, and Performance</u>, for example, has a long life because it is not wedded to a particular software language, choosing instead to talk about the physics and psychoacoustics of sound, implementing synthesis circuits using graphic notation like the diagram above.

Stages of a SynthSound

Recall that we encourage all synth design to be encapsulated in SynthCircuits. The SynthSound superclass of SynthCircuit supports a notion of a “stage” of a sound. Sounds are generally more complex than simply “on” or “off”, there may be a startup stage, and shutdown stage, and numerous stages in between (like shifting gears in a race car). Thus, SynthSound’s **public void setStage(int time, int stage)** method is a good one to keep to override and define your own behavior.

SynthNotes

A SynthNote is simply a subclass of SynthCircuit which has noteOn(), noteOff(), noteOnFor() methods, and two predefined public SynthInputs named **frequency** and **amplitude**. NoteOn()’s parameters are **time** (ie. when to play), **frequency**, and **amplitude**. SynthNote is useful if you want to play circuits that deliver the concept of a singular note-like event. One of the ways to get this sort of singular feel is by putting an envelope over an otherwise sustaining sound, to give it a beginning, a middle, and an end. But we will introduce JSyn's envelopes in the next section.

There is nothing magic about these three methods. They simply call setStage(time, 0) for note on , and setStage(time, 1) for note off, as we see below

```
public void noteOn( int time, double frequency, double amplitude )
        {
                if( this.frequency != null ) this.frequency.set( time, frequency );
                if( this.amplitude != null ) this.amplitude.set( time, amplitude );
                setStage( time, 0 );
        }

public void noteOff( int time )
        {
                setStage( time, 1 );
        }

public void noteOnFor( int time, int duration, double frequency, double amplitude )
        {
                noteOn( time, frequency, amplitude );
                noteOff( time + duration );
        }
```

IMPORTANT: JSyn Envelopes and SynthNotes do not know about each other, care about each other, nor do they particularly need each other. It just makes sense to introduce them together here because our first notion of a SynthNote is typically a sound that goes "boink" when banged, then stops in finite time. This is easily implemented with an Envelope.
IMPORTANT: The Java output of Wire is a SynthNote with these stages defined for you.

Envelopes

An envelope in JSyn is implemented with two objects:
1) SynthEnvelope - contains the time/amplitude data shaping the envelope. A double[] array containing time/value pairs is passed to SynthEnvelope's constructor.
2) EnvelopePlayer - performs a SynthEnvelope's data, connectable to ports

We will now create a class by hand (as opposed to using Wire), called SynthBoinkEnv, which puts an amplitude envelope over a TriangleOscillator.

Note in the source below, a few fine points:

- The amplitude port of the SynthNote is not an alias for the oscillator's amplitude port, rather the EnvelopePlayer's amplitude port.
- EnvelopePlayer is add()'ed to the SynthNote circuit, and its output is wired to the amplitude input of the oscillator.
- The SynthEnvelope is **not** add()'ed to the SynthNote. It is a free-range object that could be used by any EnvelopePlayer that cares to access it. It is not part of the circuit itself. It is used at run time by EnvelopePlayer as a parameter of a call to the method queue().
- EnvelopePlayer, therefore, is not fixed to play a particular envelope shape. It sets its SynthEnvelope at run time with the call to:
  `myEnvelopePlayer.envelopePort.queue(time, `**`mySynthEnvelope`**`);`
- Note that the data[] array is not directly used by EnvelopePlayer, but served to it by SynthEnvelope. SynthEnvelope's constructor is passed the data[] array. Data[] is no longer relevant after this constructor, and can be garbage collected with no effect on the circuit.
- Note that while the data[] array of the envelope rises to max amplitude, at runtime, this entire shape is scaled by the input to the amplitude port of the EnvelopePlayer using the data.
- The envelope data contains a "release jump" which is a short frame at the end that goes to 0
- The input ports use setUp() to inform other classes like SoundTester what the limits and defaults are.
- A main() method is included that tosses up a Frame with a SoundTester in it to test the sound. Make this a common practice!

```
package com.didkovsky.javamusic;
import com.softsynth.jsyn.*;

import java.awt.*;
import java.awt.event.*;

import com.softsynth.jsyn.view11x.SoundTester;


/** Build a simple SynthNote, with an amplitude envelope.

@author Nick Didkovsky 10/3/99

*/

public class SynthBoinkEnv extends SynthNote {

// sound-making units declared here
        private TriangleOscillator osc;
        private SynthEnvelope mySynthEnvelope;
        private EnvelopePlayer myEnvelopePlayer;

// frequency and amplitude input ports are already declared in superclass
// output port already declared in superclass of superclass

// constructor: allocate units, wire units together, define port aliases, set
// signal types for external control
        public SynthBoinkEnv() throws SynthException {
```

```
                // add units to the circuit
                add(osc = new TriangleOscillator());
                add(myEnvelopePlayer = new EnvelopePlayer());

// set up connections - wire the output of the envelope player to
// the amplitude of the oscillator
                myEnvelopePlayer.output.connect(osc.amplitude);

                //      expose the freq and amp of the sineOscillator as aliases
                addPort(frequency = osc.frequency);
                addPort(amplitude = myEnvelopePlayer.amplitude);

                // expose an output port aliased to the oscillator's output
                addPort(output = osc.output);
                buildEnvelope();

                // set the min/max limits and defaults on ports
                 amplitude.setup( 0.0, 0.0, 1.0 );
                 frequency.setup( 0.0, 440, 4186.0 ); // 4186 Hz = C8; highest pitch on
piano
        }


        void buildEnvelope() throws SynthException {
    /* Create an envelope and fill it with recognizable data. */
      double[] data =
             {
                 0.02, 1.0,  /* duration,value pair for frame[0] */
                 0.30, 0.1,  /* duration,value pair for frame[1] */
                 0.50, 0.7,  /* duration,value pair for frame[2] */
                 0.50, 0.9,  /* duration,value pair for frame[3] */
                 0.80, 0.0,   /* duration,value pair for frame[4] */
                 0.01, 0.0   /* duration,value pair for frame[5], the "release jump" */
             };

                mySynthEnvelope = new SynthEnvelope(data);

        }

// Now override setStage to do whatever you want
// stage 0 is typically noteon, stage 1 is note off
public void setStage( int time, int stage ) throws SynthException {
    {
        switch( stage )
        {
        case 0:
                start( time );
                myEnvelopePlayer.envelopePort.clear(time);
        // queue up the envelope, starting at frame 0, running through all frames,
        // and flag it to stop the circuit when completed. Efficient!
                myEnvelopePlayer.envelopePort.queue(time, mySynthEnvelope, 0,
mySynthEnvelope.getNumFrames(), Synth.FLAG_AUTO_STOP);
            break;
        case 1:
                myEnvelopePlayer.envelopePort.clear(time);
        // queue up the release jump of the envelope
                myEnvelopePlayer.envelopePort.queue(time, mySynthEnvelope,
mySynthEnvelope.getNumFrames()-1, 1, Synth.FLAG_AUTO_STOP);
            break;
        default:
            break;
        }
    }
}


public static void main(String args[]) {

        Synth.startEngine(0);
        SynthObject.enableTracking(true);
```

```
        LineOut out = new LineOut();
        SynthBoinkEnv synthNote = new SynthBoinkEnv();

        synthNote.output.connect(0, out.input, 0);
        synthNote.output.connect(0, out.input, 1);

        out.start();

        SoundTester soundTesterPanel = new SoundTester(synthNote);

        Frame f = new Frame("Test sound");
        f.addWindowListener(new WindowAdapter() {
               public void windowClosing(WindowEvent e) {
                      Synth.stopEngine();
                      System.exit(0);
               }
        });
        f.add(soundTesterPanel);
        f.pack();
        f.setVisible(true);
}

}
```

Let's put a Sound Tester for this SynthNote into an applet

```
package com.didkovsky.javamusic;

import com.softsynth.jsyn.*;
import java.awt.*;
import java.awt.event.*;
import com.softsynth.jsyn.view11x.SoundTester;

/**
 * SynthBoinkApplet.java
 *
 * Play a SynthNote in an Applet
 *
 * @author Nick Didkovsky, didkovn@mail.rockefeller.edu
 * (C) 1998 Nick Didkovsky
 *
 * JSyn (C) 1997 Phil Burk, visit www.softsynth.com
 */

public class SynthBoinkApplet extends java.applet.Applet
{
        SynthBoinkEnv mySynthNote;
        LineOut myOut;

/* set up GUI */
public void init() {
}

/*
 * Setup synthesis.
 */
        public void start()
        {
/* Start synthesis engine. */
               Synth.startEngine( 0 );

/* Make waveform unit generators. */
               mySynthNote = new SynthBoinkEnv();
               myOut = new LineOut();

/* Create a SoundTester panel and add to layout */
               SoundTester soundTesterPanel = new SoundTester(mySynthNote);
               add(soundTesterPanel);
```

```
/* Connect oscillator to both channels of stereo player. */
               mySynthNote.output.connect( 0, myOut.input, 0 );
               mySynthNote.output.connect( 0, myOut.input, 1 );

/* Start execution of units. */
               myOut.start();

               getParent().validate();
               getToolkit().sync();
}


public void stop()
        {
/* remove any gui components we may have added */
               removeAll();
/* Turn off tracing. */
               Synth.setTrace( Synth.SILENT );
/* Stop synthesis engine. */
               Synth.stopEngine();
        }

}
```

The HTML to deliver this applet follows

```
<HTML>
<HEAD>
   <TITLE>JSyn Example </TITLE>
</HEAD>
<BODY>

<H2>JSyn SynthBoinkApplet </H2>

<br>
JSyn is a real-time software synthesis API written in Java by Phil Burk.
JSyn outputs CD quality 16 bit stereo sound from synthesis circuits the programmer
designs in Java. JSyn applications can run in a Web Browser with the plugin, like this
one, and as stand-alone applications.<br><br>
<APPLET
    code="com.didkovsky.javamusic.SynthBoinkApplet.class"
    codebase="../classes"
    width=400
    height=300 ></APPLET>


<pre>
* (C) 2001 Nick Didkovsky, <a
href="mailto:didkovn@mail.rockefeller.edu">didkovn@mail.rockefeller.edu</a>
*
* JSyn (C) 1997 Phil Burk, visit <a
href="http://www.softsynth.com/">http://www.softsynth.com</a>
<BR>
</BODY>
</HTML>
```

The JSyn Event Buffer

You can post JSyn events in the future with JSyn's event buffer. All methods in JSyn that set SynthInput values in units have optional time parameters. The time is passed as ticks, which is an integer. The current tick value (int) is returned by Synth.getTickCount(), while the ticks per second (double) is returned by Synth.getTickRate(). For example, you could change the frequency of an oscillator to 440 Hz, exactly ten seconds from now with a statement like:

```
osc.frequency.set(Synth.getTickCount() + (int)(10 * Synth.getTickRate()), 440.0);
```

Notice that you do not need to know what the actual TickCount is or what the actual TickRate is. Your knowing the actual values isn't helpful in the math expression above. It works because you are sure that Synth.getTickCount() means "now", and Synth.getTickRate(), whatever it is, equals the number of ticks in one second.

The Java code in this statement executes almost immediately (a few microseconds), and the frequency will change ten seconds later. In loose terms, the event itself sort of detaches itself from the program flow.

Not surprisingly, you could do some pretty cool things with JSyn's event buffer, like post tons of noteOn()'s in the future with very accurate timing. Here's a simple adaptation of `SynthBoinkApplet`, called `SynthBoinkApplet2`. We add a java.awt.Button which triggers a handleBang() method, which launches ten notes in the future, each 1/3 sec later than the previous.

Notice how little code there is? That's because we inherit from SynthBoinkEnvApplet. The only difference is the button and handleBang(), so this class definition does not need any of the start Synth stuff, GUI layout, etc. Java's pretty great, I must say…

```
package com.didkovsky.javamusic;

import com.softsynth.jsyn.*;
import java.awt.*;
import java.awt.event.*;

/**
 * SynthBoinkAppletEB.java
 *
 * Post a bunch of random SynthNotes in the future by hitting a awt.Button
 * Uses Event Buffer to post notes in future
 *
 * @author Nick Didkovsky, didkovn@mail.rockefeller.edu
 * (C) 2001 Nick Didkovsky
 *
 */

public class SynthBoinkAppletEB extends SynthBoinkApplet implements ActionListener {

Button bangButton;

public void start() {
        super.start();
        add(bangButton = new Button("BANG!"));
        bangButton.addActionListener(this);
        getParent().validate();
        getToolkit().sync();
}

void oneRandomNote(int time) throws SynthException {
        double freq = Math.random() * 500.0 + 200.0;  // frequency between 200 and 700 Hz
        double amp = Math.random(); // some amplitude 0.0 .. 1.0
        mySynthNote.noteOn(time, freq, amp);
}

/* Post ten random notes in the future, 1/3 sec interval */
void handleBang() {
        int when = Synth.getTickCount();
        for (int i=0; i<10; i++) {
                when += (int)(Synth.getTickRate() / 3);  // 3 notes per sec
                oneRandomNote(when);
        }
}
public void actionPerformed(ActionEvent e) {
        Object source = e.getSource();
        if (source == bangButton)  handleBang();
}
}
```

A convenient JSyn scrollbar you should know about.

JSyn has an easy-to-use GUI slider called PortFader which you can wire to an input of your circuits. Then the user can change synthesis values by hand. You've already seen it, as it is used by SoundTester!

A jdk1.0.2 compliant version resides in com.softsynth.jsyn.view102 while a 1.1+ version resides in com.softsynth.jsyn.view11x. Here's an example that adds a PortFader to a layout, which controls the `warpAmplitude` port of a circuit called `warp`.

```
import com.softsynth.jsyn.view11x*;
...
add(new.PortFader(warp.warpAmplitude, "warp amp", 0.0, 0.0, 1.0));
```

FM Pair

Here we review the classic Chowning FM Pair, and implement it in JSyn as a SynthNote. You may refer to the diagram of its signal flowchart at the top of these notes. An FM Pair has one sine oscillator's output modulating the frequency of another (called the carrier). This modulating signal is first added to a carrier frequency. This sum is sent to the frequency input of the carrier.

You see there are four inputs on an FM Pair. We would like to have access to all of them, as they are responsible for pitch, loudness, and timbre. But we also want the simplicity of hitting an FM Pair with a frequency and an amplitude. We will wrap the raw FM pair up as a SynthCircuit, exposing all its ports, then wrap that circuit up in a SynthNote, which exposes only frequency and amplitude. ("*When faced with a choice, do both.*" - Oblique Strategies).

The timbre of an FM Pair is the result of two ratios:
1)Fc:Fm (ie the carrier frequency compared to the modulating frequency)
2)ModAmp:ModFreq (called index of modulation)

Our FM SynthNote shall have these two values specified in the constructor. Then it can be banged with any carrier frequency and amplitude, maintaining these ratios by calculating its own ModFreq and ModAmp. It will calculate its own modulating frequency from ratio 1 above, then calculate its modulating amplitude using ratio 2 above. Note that the calculations have to be done in this order, since the result of 1 (mod freq) is used to calculate 2.

Here is the source for the FM Pair Circuit (thanks to Andrew Gram)

```
package andrewgram;

import com.softsynth.jsyn.*;
import java.awt.*;
import java.awt.event.*;
import com.softsynth.jsyn.view11x.SoundTester;

/**      FM Synthcircuit

         @author Andrew Gram 3/99
         reusable FM pair, mods by ND
*/

public class FMCircuit extends SynthCircuit {

// declare units
         SineOscillator modOsc;
         SineOscillator carOsc;

         AddUnit adder;
         public SynthInput modfrequency;
         public SynthInput modamplitude;
         public SynthInput carfrequency;
```

```
        public SynthInput caramplitude;

        public SynthOutput output;

// constructor
       public FMCircuit() {
// add SynthUnits to circuits
               add(modOsc = new SineOscillator());
               add(carOsc = new SineOscillator());
               add(adder = new AddUnit());

// start connecting
               modOsc.output.connect(adder.inputB);
               adder.output.connect(carOsc.frequency);

// add ports visible to other classes
               addPort(modfrequency = modOsc.frequency, "Mod Frequency");
               addPort(modamplitude = modOsc.amplitude, "Mod Amplitude");
               addPort(carfrequency = adder.inputA, "Carrier Frequency");
               addPort(caramplitude = carOsc.amplitude, "Carrier Amplitude");
               addPort(output = carOsc.output, "Output");

               modfrequency.setup( 0.0, 220, 1000.0 );
               modamplitude.setup( 0.0, 0.0, 1000.0 );
               carfrequency.setup( 0.0, 220, 4186.0 ); // 4186 Hz = C8; highest on piano
               caramplitude.setup( 0.0, 0.5, 1.0 );

       }

       public static void main(String args[]) {

               Synth.startEngine(0);
               SynthObject.enableTracking(true);

               LineOut out = new LineOut();
               FMCircuit sound = new FMCircuit();

               sound.output.connect(0, out.input, 0);
               sound.output.connect(0, out.input, 1);

               out.start();
               sound.start();

               SoundTester soundTesterPanel = new SoundTester(sound);

               Frame f = new Frame("Test sound");
               f.addWindowListener(new WindowAdapter() {
                       public void windowClosing(WindowEvent e) {
                               Synth.stopEngine();
                               System.exit(0);
                       }
               });
               f.add(soundTesterPanel);
               f.pack();
               f.setVisible(true);

       }
}
```

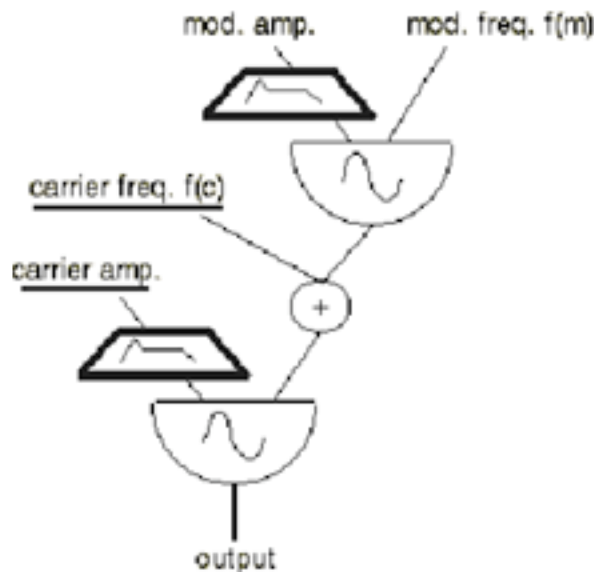
We now wrap this FM circuit in a **SynthNote**. See the circuit flowchart below. We put a couple of envelopes on the carrier amplitude port and the modulator's amplitude port. The setEnvelopes() method receives arrays of doubles which are used to create the envelope data.

A few important points:

NoteOn() is responsible for start()'ing the circuit. This is a safe thing, as subsequent start()'s to an already-started circuit are ignored. Furthermore, we flag the SynthNote's envelope player with Synth.FLAG_AUTO_STOP which means, "if you successfully play through all frames of this envelope, shut down the circuit". Thus, CPU usage is preserved: a SynthNote that is not making noise is not eating CPU cycles.

If we want to avoid a pop at the beginning of a noteOn() when start() is called, we must ensure that the carrier amplitude is zero before start() is called (why?). The constructor does that for the SynthNote's initial state, but what about between notes where the circuit has shut down? To ensure that the amp goes down to zero, if noteOn() finishes playing through its entire envelope, we simply make sure the last frame's value is 0.0. To ensure that the amp goes down to zero if noteOff() is called, we added a "release jump" segment to the end of the envelope. This short segment simply brings the amp down to 0.0 in a tenth of a second. We queue up this portion of the envelope in noteOff() after clear()'ing any pending segments.

Now we are sure start() will never cause a pop. What about a noteOn() that interrupts a sounding circuit? To avoid a pop here, we wrote our envelope data's first frame with a fast duration. The first segment of the envelope shall go to the first desired amplitude very quickly, but not instantly. That avoids a pop if you get a new noteOn while a previous noteOn is still playing. *Remember that envelope players do not start at 0.0!!! When an envelope frame is queued, it specifies the duration that the **current value** shall take to get to the specified value!!!*



FMSynthNote circuit flow chart

Following is the source for our FM SynthNote.

```
package com.didkovsky.javamusic;
import com.softsynth.jsyn.*;
```

```
import com.softsynth.jmsl.util.*;
import andrewgram.*;
import com.softsynth.jsyn.view11x.SoundTester;
import java.awt.*;
import java.awt.event.*;

/** A FM pair SynthNote with envelopes for car amplitude and mod amplitude. <br>
@author Nick Didkovsky, 2/27/2000
*/

public class FMSynthNote extends SynthNote {

        FMCircuit fmCircuit;
        EnvelopePlayer carAmpEnv;
        EnvelopePlayer modAmpEnv;
        SynthEnvelope carAmpEnvData;
        SynthEnvelope modAmpEnvData;

        double modIndex;
        double modfreq;
        double fcfmRatio;

        /** Construct a SynthNote that maintains numer/denom Fc:Fm ratio, with specified
Modulation index */
        public FMSynthNote(double numer, double denom, double index)  {
                fcfmRatio = numer / denom;
                modIndex = index;
                add(fmCircuit = new FMCircuit());
                add(carAmpEnv = new EnvelopePlayer());
                add(modAmpEnv = new EnvelopePlayer());

                carAmpEnv.output.connect( 0, fmCircuit.caramplitude, 0 );
                modAmpEnv.output.connect( 0, fmCircuit.modamplitude, 0 );

/* Make ports on internal units appear as ports on circuit. */
                addPort(amplitude = carAmpEnv.amplitude, "amplitude");
                addPort(frequency = fmCircuit.carfrequency, "frequency");
                addPort(output = fmCircuit.output, "output");

                amplitude.setup( 0.0, 0.0, 1.0 );
                frequency.setup( 0.0, 440, 4186.0 ); // 4186 Hz = C8; highest pitch on
piano

                setDefaultEnvelopes();

}

        public void setEnvelopes(double[] carAmpData, double[] modAmpData)  {
                carAmpEnvData = new SynthEnvelope(carAmpData);
                modAmpEnvData = new SynthEnvelope(modAmpData);
        }

        void setDefaultEnvelopes() {
                double[] carAmpData    = { 0.1, 1.0, 0.9, 0.25, 0.5, 0.0, 0.1, 0.0 };
        // notice last release jump segment
                double[] modAmpData    = { 1.0, 1.0, 0.25, 0.25, 0.25, 0.0, 0.1, 0.0};
                setEnvelopes(carAmpData, modAmpData);
        }


public void setStage( int time, int stage ) throws SynthException {
{
  switch( stage )
    {
        case 0:
                start( time ); // ignored if already started
                double freq = frequency.getCurrent();
                double amp = amplitude.getCurrent();
                System.out.println("fref=" + freq);
                System.out.println("amp=" + amp);
```

```
        // Calculate modulation frequency given the Fc:Fm ratio and the carrier frequency
*/
                modfreq = freq/fcfmRatio;
                fmCircuit.modfrequency.set(time, modfreq);

        // Calculate modulation amplitude
                double modamp = modfreq * modIndex;
                modAmpEnv.amplitude.set(time, modamp);
                carAmpEnv.envelopePort.clear(time);
                carAmpEnv.envelopePort.queue(time, carAmpEnvData, 0,
carAmpEnvData.getNumFrames(), Synth.FLAG_AUTO_STOP);

                modAmpEnv.envelopePort.clear(time);
                modAmpEnv.envelopePort.queue(time, modAmpEnvData, 0,
modAmpEnvData.getNumFrames());
        break;
        case 1:
                carAmpEnv.envelopePort.clear(time);
                // queue the release jump of envelope: a segment that drops to 0.0
quickly.
                carAmpEnv.envelopePort.queue(time, carAmpEnvData,
carAmpEnvData.getNumFrames()-1, 1, Synth.FLAG_AUTO_STOP);
                modAmpEnv.envelopePort.clear(time);
                // don't do the following with auto stop.  why?
                modAmpEnv.envelopePort.queue(time, modAmpEnvData,
modAmpEnvData.getNumFrames()-1, 1);
            break;
        default:
            break;
        }
    }
}


public static void main(String args[]) {
        Synth.startEngine(0);
        SynthObject.enableTracking(true);

        LineOut out = new LineOut();
        FMSynthNote synthNote = new FMSynthNote(1, 1, 5);

        synthNote.output.connect(0, out.input, 0);
        synthNote.output.connect(0, out.input, 1);

        out.start();

        SoundTester soundTesterPanel = new SoundTester(synthNote);

        Frame f = new Frame("Test sound");
        f.addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                        Synth.stopEngine();
                        System.exit(0);
                }
        });
        f.add(soundTesterPanel);
        f.pack();
        f.setVisible(true);

}

}
```

IMPORTANT POINT: note that the Modulating Frequency and Modulating Amplitude are calculated at the Java level. We can get away with this because a single note event assumes these values will stay fixed for the length of the event. If we wanted to have real time control over the index of modulation and the Fc:Fm ratio, we'd have to implement this at the circuit level.

John Chowning's FM Bell

Finally, here is an implementation of FMSynthNote, which sounds bell-like, using John Chowning's recommended Fc:Fm ratio (5:7), Index of Modulation (less than 10), and an exponentially decreasing envelope (you need JMSL for this envelope tool).

```
package com.didkovsky.javamusic;
import com.softsynth.jsyn.*;
import com.softsynth.jmsl.util.ExponentialDecayInterpolator;
import com.softsynth.jmsl.util.EnvelopeDataMaker;
import java.awt.*;
import java.awt.event.*;
import com.softsynth.jsyn.view11x.SoundTester;

/** An implementation of FMSynthNote using Chowning's recommendations for Fc:Fm ratio
(5:7) and modIndex (less than 10)
@author Nick Didkovsky. */
public class ChowningBell extends FMSynthNote {

double steepness;
/** steepness is a value from around 4..32, higher means steeper exponential decay */
public ChowningBell(double envSteepness, double index)  throws SynthException
{
        super(5, 7, index);
        steepness = envSteepness;
        buildEnvelopes();     // max amp, decay dur, # segments
}

void buildEnvelopes() throws SynthException {
        double amp = 1.0;
        double dur = 15;
        int steps = 60;
        ExponentialDecayInterpolator exp = new ExponentialDecayInterpolator(0.0, amp, dur,
0.0);
        exp.setSteepness(steepness);
        EnvelopeDataMaker envDataMaker = new EnvelopeDataMaker(dur, steps, exp); //
segments approximating an exp decay
        setEnvelopes(envDataMaker.getData(), envDataMaker.getData());
}

public static void main(String args[]) {
        Synth.startEngine(0);
        SynthObject.enableTracking(true);

        LineOut out = new LineOut();

        // try different values for steepness and modindex
        ChowningBell synthNote = new ChowningBell(8, 2);

        synthNote.output.connect(0, out.input, 0);
        synthNote.output.connect(0, out.input, 1);
        out.start();
        SoundTester soundTesterPanel = new SoundTester(synthNote);

        Frame f = new Frame("Test sound");
        f.addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                        Synth.stopEngine();
                        System.exit(0);
                }
        });
        f.add(soundTesterPanel);
        f.pack();
        f.setVisible(true);

}
}
```

Risset’s Bell

Risset’s Bell is a patch design which makes a bell-like sound by summing eleven sine oscillators, each of which has its own exponentially decaying amplitude envelope, duration,

frequency and amplitude. The amplitude, frequency, and duration of each is specified in relation to one amp variable, one frequency variable, and one duration variable.

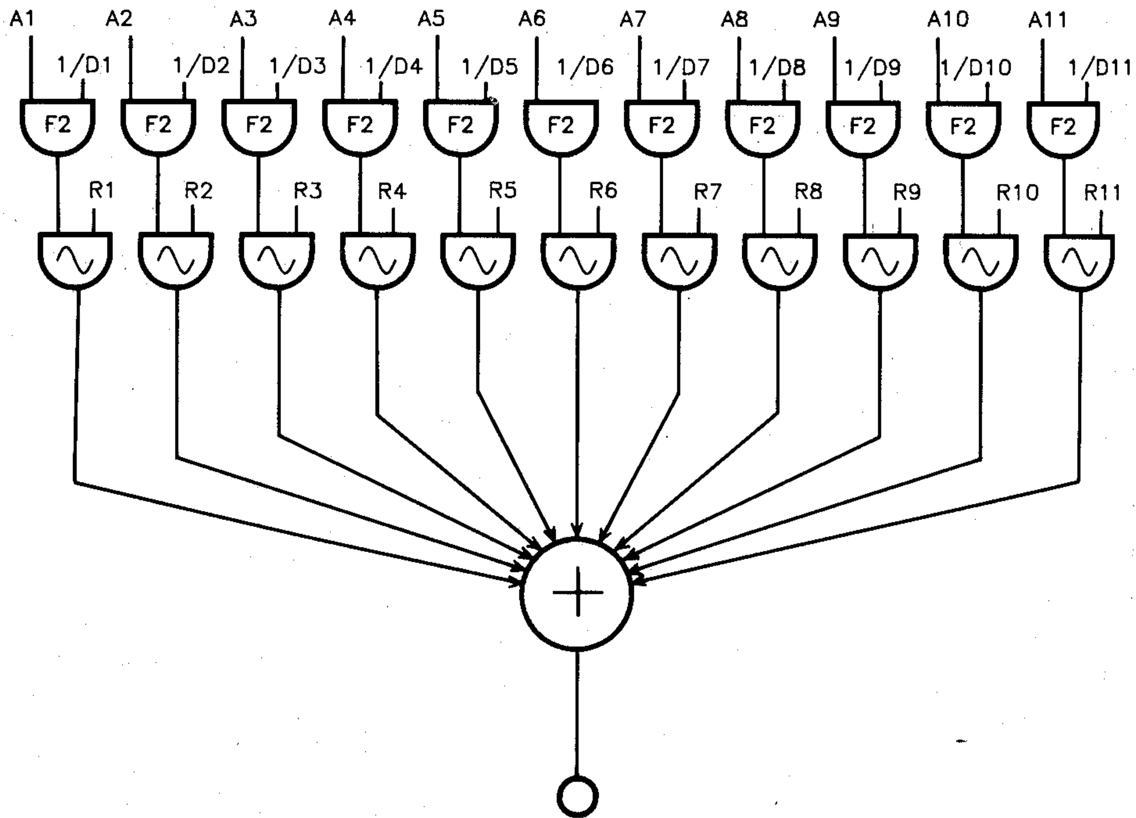


Figure taken from Dodge/Jerse <u>*Computer Music*</u>*, Schirmer Books, ISBN 0-02-873100-X*

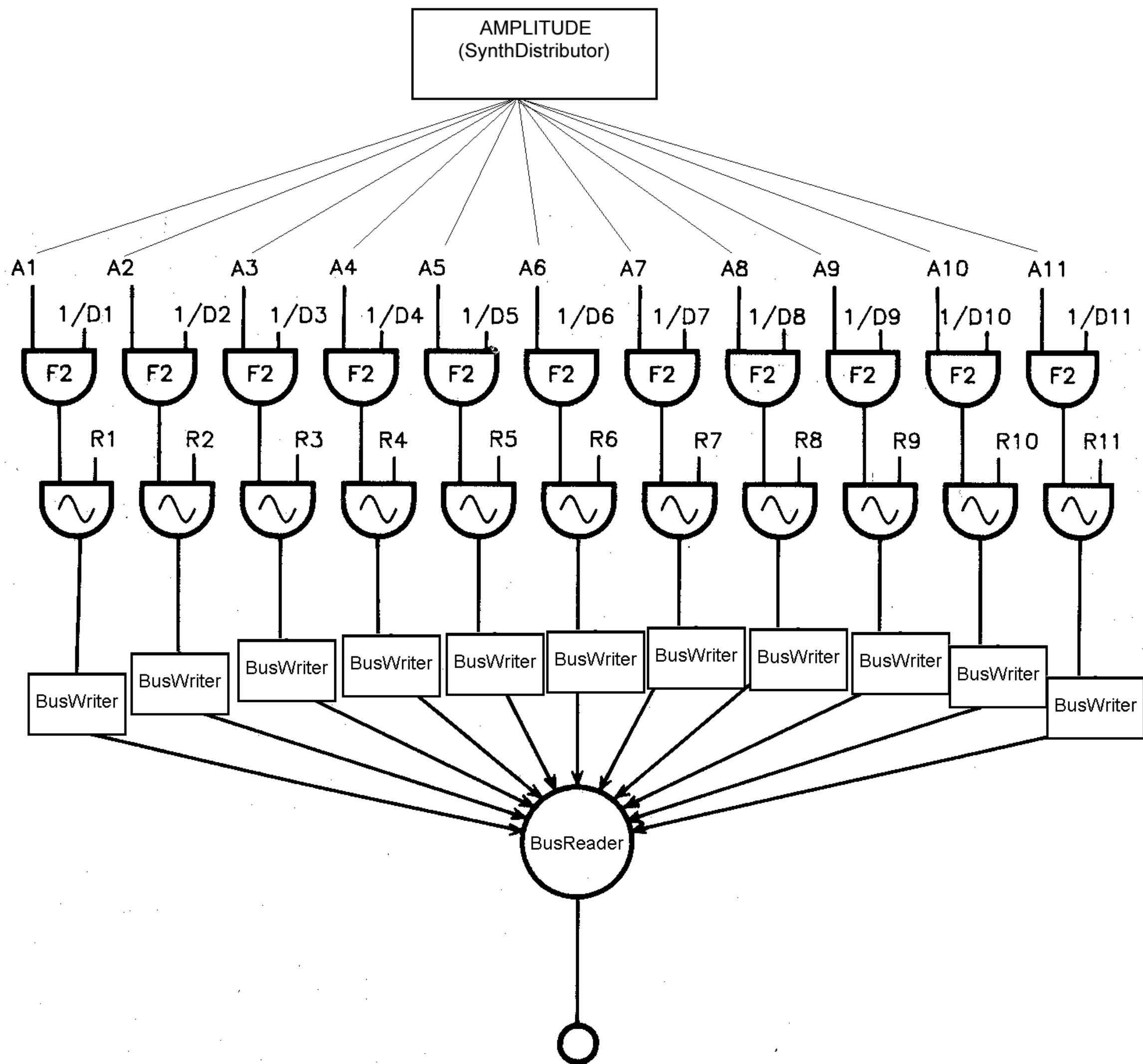
Some frequency specifications follow:
OSC# FREQ
1FREQ*.56
2FREQ*.56+1
3FREQ*.92
4FREQ*.92+1.7

Some amplitude specifications follow:
OSC# AMP
1AMP
2AMP*.67
3AMP
4AMP*1.8

Some duration specifications follow:
ENV# DUR
1DUR
2DUR*.9
3DUR*.65
4DUR*55

JSyn implementation of Risset's Bell

We modify the flowchart a bit to clarify the JSyn implementation of Risset's Bell.



Note that we want ONE amplitude port on the entire circuit. For this, JSyn provides the SynthDistributor class. Also, we want to add the outputs of numerous units together. JSyn's AddUnit only takes two inputs. To bus multiple mono signals together, JSyn provides the BusWriter and BusReader classes. Multiple BusWriters can each receive a signal at their input port, and bus it to a single BusReader. The output of the BusReader becomes the output of the SynthNote.

The JSyn source for Risset's Bell follows. Notice the liberal use of arrays to keep track of multiple units, duplicates who only differ by value. Note the extra credit at the end of these notes for information on how to make this bell better.

```
package com.didkovsky.javamusic;

import com.softsynth.jsyn.*;
import com.softsynth.jmsl.util.ExponentialDecayInterpolator;
import com.softsynth.jmsl.util.EnvelopeDataMaker;

/**
 * Risset's Bell: Add 11 Envelope/oscillator pairs together
 *
 *
 * @author Nick Didkovsky, didkovn@mail.rockefeller.edu
 * (C) 1998 Nick Didkovsky
```

```
 *
 * JSyn (C) 1997 Phil Burk, visit www.softsynth.com
 */

public class RissetBell extends SynthNote {

          public SynthDistributor amplitude;
// one input, fans out to 11 envelopePlayers's amp inputs

          private final int NUM_OSCIL = 11;
          private double ampSum;
          private double[] ampFactors = { 1.0, 0.67, 1.0, 1.8, 2.67, 1.67, 1.46, 1.33, 1.33, 1.0 , 1.33 };
          private double[] durFactors = { 1.0, 0.9, 0.65, 0.55, 0.325, 0.35, 0.25, 0.2, 0.15, 0.1, 0.175 };
          // factors and constants to calculate frequency for each oscillator; Fosc= 0.56*freq + 0.0, for example
          private FreqFunction[] freqFunctions= { new FreqFunction(0.56, 0.0),
                                                  new FreqFunction(0.56, 1.0),
                                                  new FreqFunction(0.92, 0.0),
                                                  new FreqFunction(0.92, 1.7),
                                                  new FreqFunction(1.19, 0.0),
                                                  new FreqFunction(1.7, 0.0),
                                                  new FreqFunction(2.0, 0.0),
                                                  new FreqFunction(2.74, 0.0),
                                                  new FreqFunction(3.0, 0.0),
                                                  new FreqFunction(3.76, 0.0),
                                                  new FreqFunction(4.07, 0.0) };
          private SineOscillator[] oscillators;
          private SynthEnvelope[]  synthEnvelopes;
          private EnvelopePlayer[] envelopePlayers;
          private BusWriter[] busWriters; // 11 writers merge signals into one reader
          private BusReader myBusReader;
          private double duration;      // how long this bell lasts, used to build envelopes

          public RissetBell(double dur) throws SynthException {
                    duration = dur;
                    ampSum = 0.0;       // used for scaling noteOn()'s amp later
                    for (int i=0; i<ampFactors.length; i++) {
                              ampSum += ampFactors[i];
                    }
                    oscillators = new SineOscillator[NUM_OSCIL];
                    synthEnvelopes = new SynthEnvelope[NUM_OSCIL];
                    envelopePlayers = new EnvelopePlayer[NUM_OSCIL];
                    busWriters = new BusWriter[NUM_OSCIL];
                    amplitude = new SynthDistributor( this, "amplitude" );

                    add(myBusReader = new BusReader());

                    // instantiate these units and wire them together while we're at it.
                    for (int i=0; i<NUM_OSCIL; i++) {
                              oscillators[i] = new SineOscillator();
                              busWriters[i] = new BusWriter();
                              envelopePlayers[i] = new EnvelopePlayer();
                              buildEnvelope(ampFactors[i], duration * durFactors[i], i);
// builds double[] and instantiates a SynthEnvelope with it

                              add(oscillators[i]);
                              add(busWriters[i]);
                              add(envelopePlayers[i]);

                              this.amplitude.connect(envelopePlayers[i].amplitude);
                              envelopePlayers[i].output.connect(oscillators[i].amplitude);
                              oscillators[i].output.connect(busWriters[i].input);
                              busWriters[i].busOutput.connect(myBusReader.busInput);

                    }

                    addPort(output = myBusReader.output);
          }

          public void noteOn(int time, double frequency, double amplitude) throws SynthException {
                    start( time );
                    double amp = amplitude/ampSum; // scale down to avoid clipping, ampSum was calc'ed in constructor
                    this.amplitude.set(time, amp);
                    System.out.println("RissetBell.noteOn()");
                    for (int i=0; i<NUM_OSCIL; i++ ) {
                         oscillators[i].frequency.set(time, freqFunctions[i].eval(frequency));
                         /* Calculate unique frequency for each osc */
                    }
                    for (int i=0; i<NUM_OSCIL; i++ ) {
                              envelopePlayers[i].envelopePort.clear(time);
                              envelopePlayers[i].envelopePort.queue(time, synthEnvelopes[i]);
                    }
                    System.out.println("noteOn() done");
          }


          void buildEnvelope(double amp, double dur, int i) {
                    /* Create an envelope and fill it with exponentially decaying data. */
                    ExponentialDecayInterpolator exp = new ExponentialDecayInterpolator(0.0, amp, dur, 0.0);
                    exp.setSteepness(8); // lower number = less steep, longer perceivable decay
                    EnvelopeDataMaker envDataMaker = new EnvelopeDataMaker(dur, 100, exp); // 100 steps
                    synthEnvelopes[i] = new SynthEnvelope(envDataMaker.getData());
          }
}
```

```
/** Calculates the result of a simple function like 2x+10 where 2=factor, 10=constant */
class FreqFunction {

double factor;
double constant;

          FreqFunction(double fact, double k) {
                    factor = fact;
                    constant = k;
          }

          public double eval(double x) {
                    return x * factor + constant;
          }
}
```

Extra Credit Homework Assignment: Polytimbral Polyrhythms [important to read even if you don't intend to do it!!!]

•Design any two SynthNotes you like in Wire. You may improvise their creation by simply dropping units into Wire and interconnecting them, or you may follow a design taken from Dodge/Jerse's book. They must respond to key presses by sounding when a key pressed and be quiet when the key is released (ie they have to behave like "notes")
•Export the SynthNotes as Java source so they can be imported into your homework project
•Design an applet that plays polyrhythms with your two SynthNotes (a polyrhythm for the purposes of this exercise is simply two different subdivisions of the same beat period, played simultaneously) :
•*There should be a textfield for the time (in seconds) to specify beat period. For example, 1.0 means a beat period one second long shall be subdivided.*
•*There should be two additional textfields (one for each SynthNote), where the user can enter the number of subdivisions of a beat period for that SynthNote. For example, if the user types in 4 and 5 respectively, then one SynthNote shall play 4 times a second (1/4 sec between note-ons), and the other SynthNote shall play 5 times a second (1/5 sec between note-ons). Use for loops and the event bufferto post your noteons in the future.*
•*There should be a "GO" button to launch the polyrhythm*
•*You can connect one SynthNote to LineOut's 0th input, and the other SynthNote to LineOut's 1st input.*

Besides delivering your applet, your html page should include links to the Java source and the xml pages saved from Wire

Extra Credit [important to read even if you don't intend to do it!!!]

1) The FM SynthNote defined in this document recalculates its modulating freq and amp every time setStage() is called, at the Java level (ie not at audio rates). We can do this because we want these values to reflect the desired Fc:Fm ratio and index of modulation over the course of the noteOn event. However, if we wanted to expose ports called FcFmRatio and ModIndex, how would we do it? We'd have to calculate these values at audio rates at the circuit level. Design a JSyn circuit that implements this design. Use Wire to make your life easier.

2) Risset's Bell does not have a frequency SynthInput and **it should**. Instead, the frequency is passed in to noteOn(), which calculates the frequency for each oscillator at the Java level. Rewrite RissetBell so it has a real frequency SynthInput, and the math is done with JSyn arithmetic units.