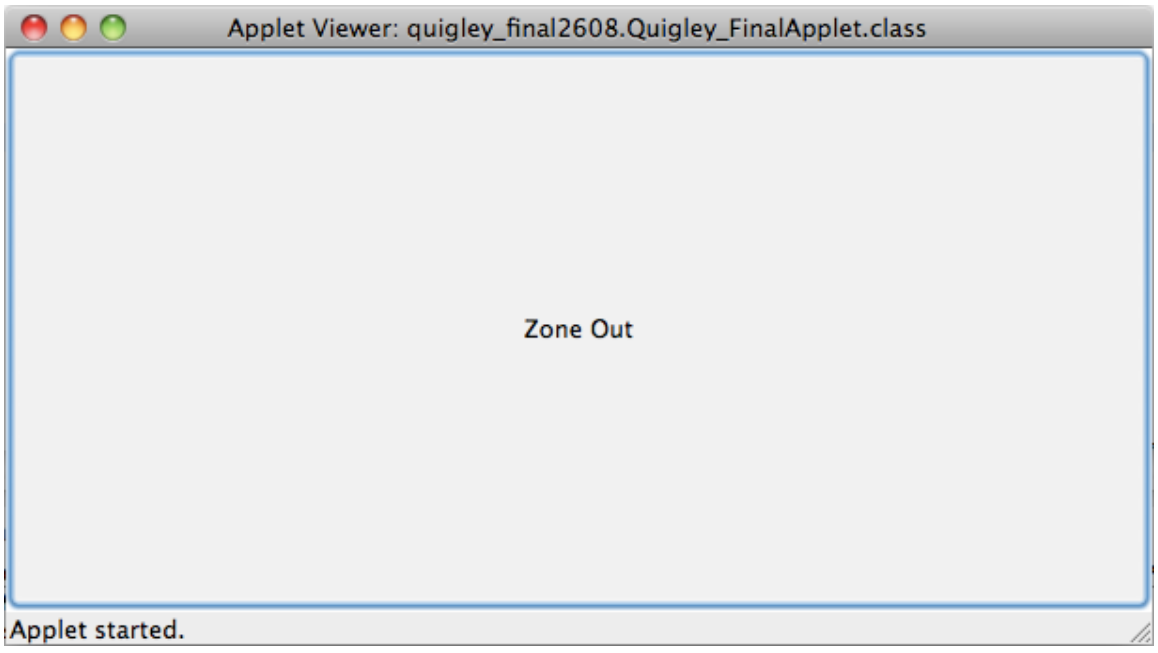
Michael Quigley
Java Music Systems
MPATE-GE 2608
Prof. Didkovsky
Final Project

Zone Out

Zone Out is a java program that creates generative ambient music for three voices, written with JMSL and JSyn.

With all the heavily technical work we do within the Music Technology department at NYU, I found much of what we learned in Java Music Systems to be easily adaptable for more artistic intentions. I chose to create a piece of music that focuses heavily on the sound design capabilities of JMSL and JSyn with the goal of exploring timbre and the self-sufficiency within the piece.

When the piece is deployed as an applet, there is little for the user to do but click a single button, titled "Zone Out". From there the piece begins to produce music on its own with three distinct voices that move subtly across the stereo field. There is no other user interaction but to focus as much or as little on the sound produced as desired. Sit rigidly and do nothing else! Watch tv at the same time! Watch videos of the ocean without any other
audio! Make your kids listen with you!



The piece is first deployed as an applet. The applet presents the user with a single button, titled "Zone Out". The button is made an AcionListener, and once clicked it triggers the actionPerformed function to call the start function and launch three music shapes. The start function then calls a series of functions to build the rest of the program. It calls functions to initialize JMSL, initialize the JSyn Music Device, initialize the JMSL Mixer Container, initialize the JSyn UnitVoiceInstruments and add them to the mixer, build the GUI, and then build the three Music Shapes associated with each JSyn instrument.

In the initJMSL() function, a JMSL.scheduler is created and started, and the JMSL.clock is set to an Advance of 0.1. The initMusicDevices() function then creates a new instance of JSynMusicDevice and opens it. The initMixer() function creates a new JMSLMixerContainer and starts it. Next is the initInstruments() function that creates three separate instruments using custom Circuit programs that implement UnitVoice. The function then sets the name for each voice. The names are never displayed anywhere but during testing I added a control for panning and amplitude of each voice to the applet that displayed the names assigned here. The three instruments are then added to the mixer and instrument one (bass) has its pan set to 0.2 and its amplitude set to 0.5. Instrument two (mid) has its pan set to 0.8 and its amplitude set to 0.4. Instrument three (noise) is left with its pan and amplitude at 0.5.The next step in the process is the building of the three music shapes for each instrument.

The circuit's getOutput() function is set to return the output port of the last Add in the chain (the one that adds the output of the delay with the output of the filter, essentially adding dry and wet signals). The noteOff() function then sets the queueOff for the envelope in order to start the decay portion of the envelope if a note off is called. The noteOn() function takes frequency, amplitude, and timestamp arguments and sets those values of both oscillators, as well as the amplitude of the envelope, accordingly.

The first instrument is a custom Circuit called QuigleyBass.java that implements UnitSource and UnitVoice. QuigleyBass contains a SineOscillator, a SegmentedEnvelope, and a VariableRateMonoReader. It has two UnitInputPorts, one for the amplitude of the oscillator and one for the frequency of the oscillator, as well as a UnitOutputPort for the output. The oscillator (osc) and VariableRateMonoReader (envelopeReader) are initialized and added to the Circuit. The output of the envelopeReader is connected to the amplitude port of the oscillator, and a function is called to build the SegmentedEnvelope. The input ports and output port are then exposed using addPort and are named appropriately for testing with a soundTweaker during development. The range and default values are then setup for the amplitude and frequency ports. The buildSegmentedEnvelope() function then declares and initializes an array of doubles called envData filled with the following data: {6, 1.0, 0.3, 0.7, 0.25, 0.599, 4, 0.0, 0.01, 0.0}. A variable "envelope" is then initialized as a new SegmentedEnvelope with envData as an argument to the constructor and sustain points are set for the beginning and end of the sustain loop. The getOutput() function then is set to return the UnitOutputPort output, which has previously been set to osc.output. The noteOff() function then sets the queueOff for the envelope in order to start the decay portion of the envelope if a note off is called. The noteOn() function takes frequency, amplitude, and timestamp arguments and sets those values of the oscillator, as well as the

amplitude of the envelope, accordingly.

Back in the main applet file, buildBass() is called after all of the initializations in order to build the MusicShape for the bass instrument described above. A MusicShape "bass" is initialized using the DimensionNameSpace of instrument one, which is the QuigleyBass.java Circuit. The repeats for bass are set the 3, the repeatPause set to 5, and the instrument set to ins1 (our JSynUnitVoiceInstrument built from QuigleyBass.java). A custom RepeatPlayable is then added to the MusicShape that allows for the panning of the bass voice to automatically be generated with each repeat. Next there is a for loop that adds 50 elements to the MusicShape with updated values for all 8 dimensions of the instrument. First there is an array of doubles called freqs[] that holds four frequency values, they are {46.25, 58.27, 69.30, 87.31} (F#, A#, C#, E#). A variable of type double is then initialized randomly to one of these frequency values using JMSLRandom.choose(0, freqs.length). This frequency value is then multiplied by 1 or 2, again determined by JMSLRandom.choose and assigned to the oscFreq dimension of the instrument. The duration is determined by JMSLRandom.choose within a range of 3 to 6, the pitch is set to 1 as it is not actually used by the instrument, the amplitude is JMSLRandomly chosen from the amplitude range of the instrument using bass.getLowLimit(2) and bass.getHighLimit(2). In this case "2" is the argument given because amplitude is the third dimension (0, 1, 2…) of the instrument. Hold time is randomly assigned between 5 and 9, and amplitude is randomly assigned between 0.0 and 1.0. All of these dimension variables are then added as an element to the MusicShape. At this point there are 50 unique elements to be played by the bass voice, repeated three times.

The second instrument is a custom Circuit called QuigleyMid.java that implements UnitSource and UnitVoice. In it we declare a SineOscillator, a TriangleOscillator, and FilterBandPass, an InterpolatingDelay, two Adds, a SegmentedEnvelope and a VariableRateMonoReader for the envelope. UnitInputPorts are then created for the amplitude and frequency of the SineOscillator and the FilterBandPass, as well as the Q of the filter and delay time of the InterpolatingDelay. A UnitOutputPort is created for the output of the circuit. Next there is a constructor for the circuit. Inside this constructor we initialize these objects and add them to the circuit. The InterpolatingDelay is allocated a buffer of maximum length 5 seconds (220500 frames). The outputs of the SineOscillator and TriangleOscillator are each connected to an add. That Add's output is then connected to the input of the FilterBandPass. The output of the FilterBandPass is connected to the input of the InterpolatingDelay, as well as the input of the second Add. The output of the InterpolatingDelay is then also connected to the second Add. The output of the VariableRateMonoReader is connected to the amplitude of each oscillator. Next the buildSegmentedEnvelope() function is called to generate the envelope for the circuit. This buildSegmentEnvelope() function is the same as that used by the QuigleyBass circuit, generating long duration swells. The UnitInput and UnitOutput Ports are then exposed to named for the use during the development process. The ranges and default values are then set for each port. The SineOscillator's amplitude port is setup for a range of 0.0 to 0.2 with a default of 0.2. It's frequency port is setup for a range of 20 to 1000 with a default of 440. The FilterBandPass's amplitude port is setup for a range of 0.0 to 0.5 with a default of 0.2. It's frequency port is setup for a range of 20 to 1000 with a default of 440. It's Q factor port is setup for a range of 0.0 to 5 with a

default of 1.0. The InterpolatingDelay's delay time port is setup for a range of 0 to 5 with a default of 2.5. The TriangleOscillator's amplitude and frequency are set to the same values as those of the SineOscillator. This was done so that the circuit would produce single pitches with each MusicShape element but the balance of amplitude between the SineOscillator and TriangleOscillator would allow for varying amount of harmonic frequency content. The circuit's getOutput() function is set to return the output port of the last Add in the chain (the one that adds the output of the delay with the output of the filter, essentially adding dry and wet signals). The noteOff() function then sets the queueOff for the envelope in order to start the decay portion of the envelope if a note off is called. The noteOn() function takes frequency, amplitude, and timestamp arguments and sets those values of both oscillators, as well as the amplitude of the envelope, accordingly.

In the main applet, when the buildMid() function is called it initializes a new MusicShape using the DimensionNameSpace of this QuigleyMid instrument. It sets it's repeats to 4 and sets the repeatPause to 5. It sets the instrument to the QuigleyMid JSynUnitVoiceInstrument created earlier in the circuit. A custom RepeatPlayable is then added to the MusicShape that allows for the panning of the mid voice to automatically be generated with each repeat, the same used for the bass MusicShape. A for loop that adds 50 elements to the MusicShape with updated values for all 10 dimensions of the instrument, similarly to that used by the bass MusicShape. The mid MusicShape is given the same array of frequency values as the bass MusicShape, with an additional index for 103.83Hz (G#). An index is chosen randomly from the array and its value is multiplied by 3 or 4, to bring it above the frequency range of the bass instrument. The rest of the dimensions (duration, pitch, amplitude, hold, oscAmp, oscFreq, filterAmp, filterFreq, filterQ, delayTime) are chosen in a similar fashion to the bass MusicShape, choosing random values within the dimension's range. The range for duration is made slightly shorter than the bass instrument so that mid will have more of a sense of motion than bass.

The second instrument is a custom Circuit called QuigleyNoise.java that implements UnitSource and UnitVoice. In it we declare a SineOscillator, RedNoise, Multiply, Delay, SegmentedEnvelope, and a VariableRateMonoReader. We then declare UnitInputPorts for oscillator amplitude and frequency, noise amplitude and frequency, and a UnitOutputPort for the output. Inside the constructor for the circuit, we initialize the objects declared above and allocate a 2 second (88200 frames) buffer for the delay. The outputs of the oscillator and noise are connected to the inputs of the Multiply. The output of the Multiply is connected to the input of the Delay. The output of the envelope reader is connected to the amplitude of the oscillator. Next the same buildSegmentedEnvelope() function is called to create long duration swells for the circuit's envelope, but with slightly faster attack time. The input and output ports are exposed and named for use during development, and the ranges and default values are set. The SineOscillator's amplitude port is setup for a range of 0.0 to 1.0 with a default of 0.5. Its frequency port is setup for a range of 20 to 1000 with a default of 440. The RedNoise's amplitude port is setup for a range of 0.0 to 1.0 with a default of 0.2. Its frequency port is setup for a range of 20 to 1000 with a default of 880. The circuit's getOutput() function is set to return the output port of the Delay. The noteOff() function then sets the queueOff for the envelope in order to start the decay portion of the envelope

if a note off is called. The noteOn() function takes frequency, amplitude, and timestamp arguments and sets those values of oscillator and noise, as well as the amplitude of the envelope, accordingly.

In the main applet the buildNoise() function initializes the third MusicShape, noise, using the DimensionNameSpace of the QuigleyNoise instrument. The Repeats is set to 3, the repeatPause is set to 1, and its instrument set to the JsynUnitVoiceInstrument built from noise circuit. The custom RepeatPlayable is also added to this circuit. Next a for loop adds 50 distinct elements to the MusicShape. This process is similar to that used for the bass and mid MusicShapes. The noise MusicShape does so with by creating an array whose length is the number of dimensions for the circuit. A nested for loop then cycles through each array index and generates a random value for that index within the the range of the corresponding dimension. Each time through the loop the array is then added to the MusicShape as an element and the process repeats.

The RepeatPlayable that is added to each MusicShape is called Quigley_autoPan, and implements Playable. A JMSLMixerContainer is declared, as well as an variable "index" of type int, and a variable "panChange" of type double. A constructor is made Quigley_autoPan(JMSLMixerContainer mixer, int index) that initializes the mixer and index given as arguments to the mixer and index declared in the class. The play function then calls JMSLRandom.randomize and produces as random value between 0.0 and 0.1 that is either added or subtracted from the current pan value for the MusicShape it was called by, keeping it within a 0.0 to 1.0 range so that the instruments are given a subtle wobble in the stereo field with bass mostly on the left, mid mostly on the right, and noise mostly in the middle.

I was happy with the realization of the piece, I felt I utilized the capabilities of JMSL and JSyn that I learned in an interesting way, changing myself to get a better understanding of much it in the process. I think I was successful in the goal of creating something musical and artistic with a new resource that I had no experience with prior to class. I enjoyed the power available to create very interesting sound design, and the ability to create huge amounts of polyphony essentially on the fly with relative ease, compared to many other computer music environments/tools. I was really happy with my exploration of timbre using this music system and think some more experiments in the future could lead to valid artistic and musical results. I would also really like to experiment more with the scheduling abilities to create some rhythmic oriented works.

Bibliography

Java Music Specification Language
http://www.algomusic.com/jmsl

JSyn
http://www.softsynth.com/jsyn/