Politecnico di Milano
School of Industrial and Information Engineering

Computer Music – Languages and Systems

Homework #2

Magical 8bit Plug

Group composition:

Brusca Alfredo	10936149
Marazzi Alice	10625416
Pomarico Riccardo	10661306

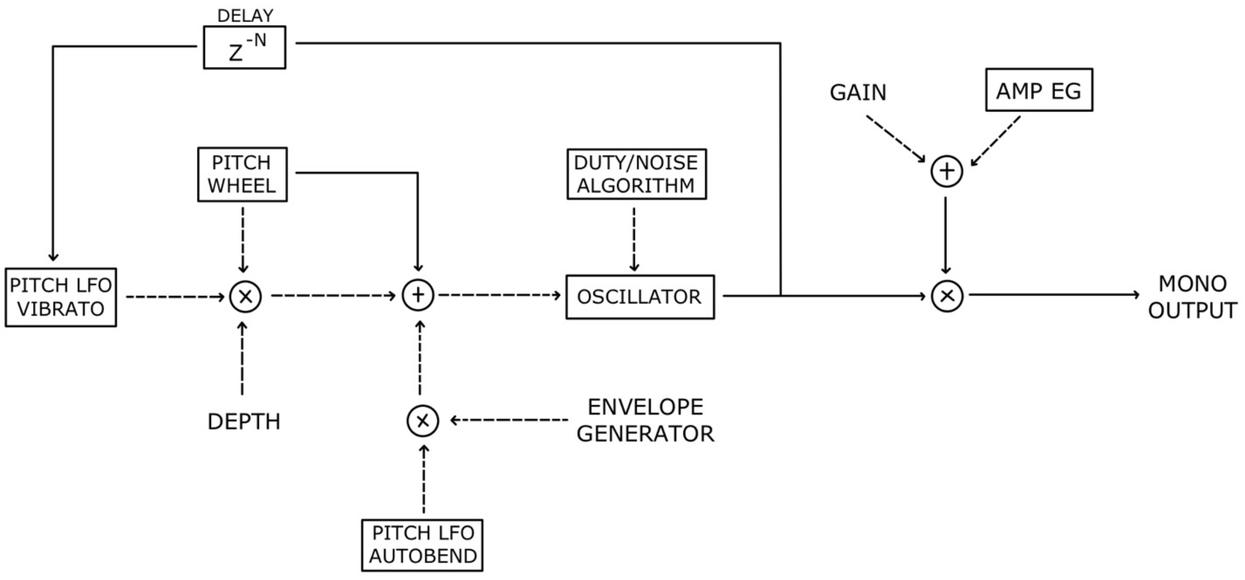
Magical 8bit Plug 2 is a software-synthesizer developed by Yokemura YMCK which generates primitive electronic sounds like the old 8bit game consoles. It can be used as a plug-in for the host applications that support Audio Units or VST.

Magical 8bit Plug 2 can produce the 8bit-specific sounds, namely pseudo triangle and low resolution noise, that are hard to reproduce with ordinary synthesizers. And on top of that it implements precise controls for 8bit-style expressions like pseudo polyphony, duty envelope etc.

Features:

- Basic 8bit-style waveforms
- As the new waveforms, 1bit noise is introduced, which can emulate the sound of original 8bit console more accurately
- Envelope forming with so-called ADSR
- Pitch bend capability with depth control
- Auto Bending which is suitable for making sound effects and drum sound
- Vibrato to enrich your musical expression
- Custom envelopes, which gives you more precise control over volume, pitch and pulse duty values
- Portamento for smooth transition between the notes

Block diagram



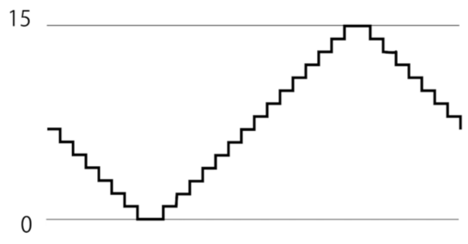
Oscillator

The oscillator block is responsible for the waveshape generation.
The waveshapes are obtained in the time domain by different approaches based on the wave type. The main options are triangular wave, rectangular wave, and noise, generated by *TriangleVoice*, *PulseVoice* and *NoiseVoice* respectively. These all inherit from TonalVoice which inherits from BaseVoice, which lastly inherits from SynthesizerVoice.

The ***triangular wave*** returns the value of the signal in the time domain from an array containing 32 values:

```
int sequence[32] = { 1,  2,  3,  4,  5,  6,  7,  8,
                     8,  7,  6,  5,  4,  3,  2,  1,
                     0, -1, -2, -3, -4, -5, -6, -7,
                    -7, -6, -5, -4, -3, -2, -1, 0
                   };
```

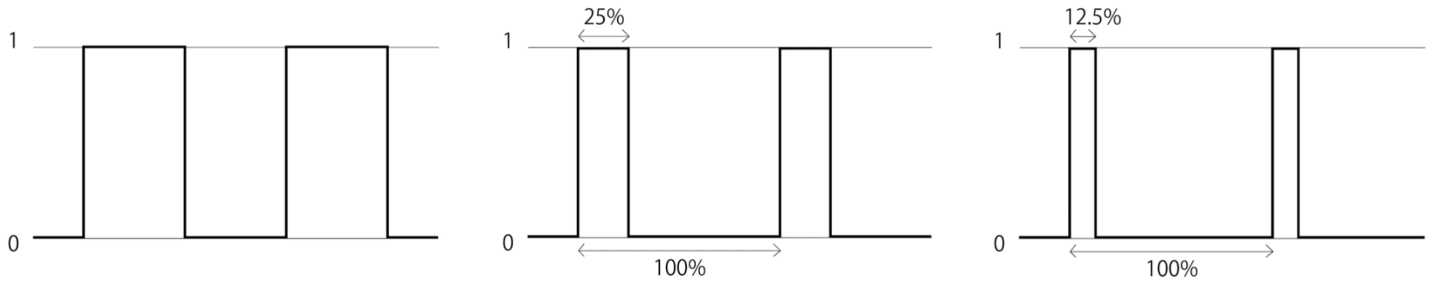
This is not a real triangular wave as it creates a step-wise motion meant to recreate the 8-bit sound of old gaming consoles.



The ***pulse wave*** has three different duty cycle parameters that change the duration of the high level with respect to the low level.

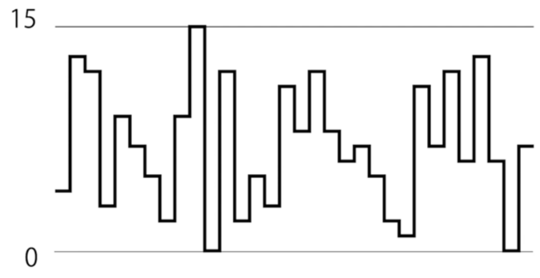
$$\begin{cases} s(i) = -1 \text{ for angle} < \text{pi*rate} \\ s(i) = 1 \text{ for angle} > \text{pi*rate} \end{cases}$$

(where rate is 1 for a 50% duty cycle, 0.5 for 25% and 0.25 for 12.5%)



Lastly the ***noise*** has three possible configurations:
- a random 4-bit noise,
- a pseudo-random 1 bit short noise,
- a pseudo-random 1 bit long noise.

The 4-bit configuration returns a random number between -2 and 2 at each update.



The 1 bit configurations are meant to replicate exactly the pseudo-random nature of the old NES console, and it does so by working with a binary register that is constantly rotated. The difference between the long and short cycle is the length of the register, as the short cycle takes 1 value every 6 of the register. This is done with the following code:

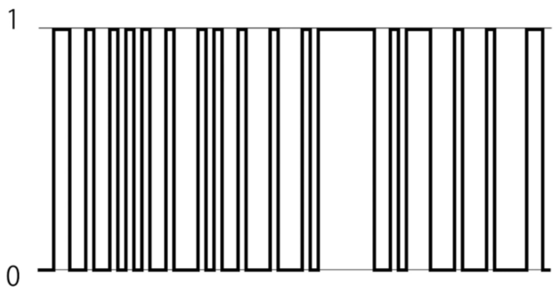
```
int compareBitPos = settingRefs->noiseAlgorithm() == kNoiseLong ? 1 : 6;
unsigned int bit0 = rgstr & 0b0000001;
unsigned int compared = (rgstr & 1 << compareBitPos) >> compareBitPos;
unsigned int feedback = bit0 ^ compared;
rgstr = rgstr >> 1;
unsigned int mask = ~ (1 << 14);
unsigned int writeback = feedback << 14;
rgstr = (rgstr & mask) | writeback;
currentVoltage = (float) bit0 - 0.5;
```

Long cycle

1 0 1 0 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 0 . . . 1 0 1 0 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 0 . . .

Short cycle

1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1

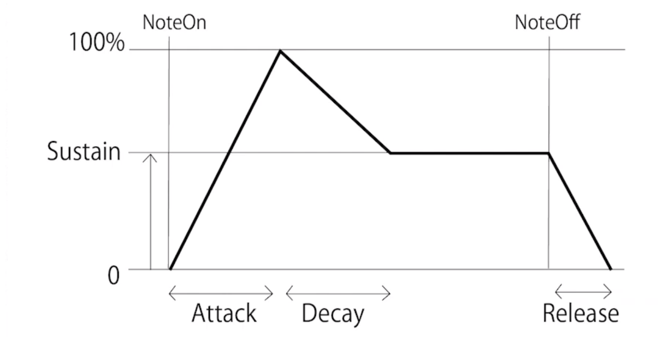


Polyphony

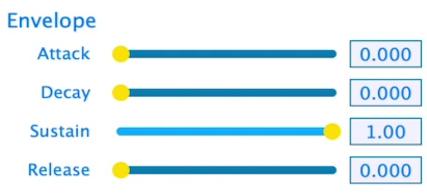
The polyphony is managed by creating a Synth object inside the processor with arbitrary number of voices limited by the maximum value of polyphony. Every voice has the same the parameters. If the mono mode is selected (we have only one voice) some parameters are unlocked, specifically the possibility of playing legato and arpeggio.

Envelope

The envelope is handled by dividing the signal into 4 phases. For attack, decay and release the time is specified while the values are fixed and for the sustain phase the value can be specified.

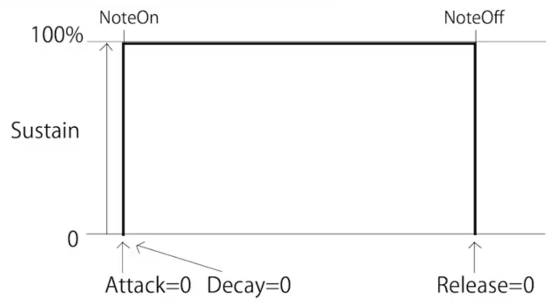


The default setting applied to a pulse or square wave create a sound that can be associated with a typical beep sound, with the Sustain level variable set a 1.00 and the time variables set at 0.000.

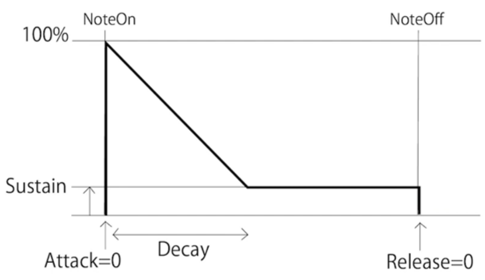


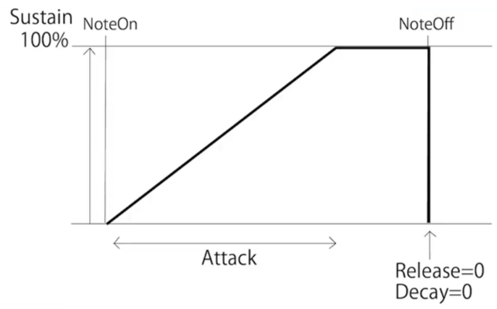
Envelope

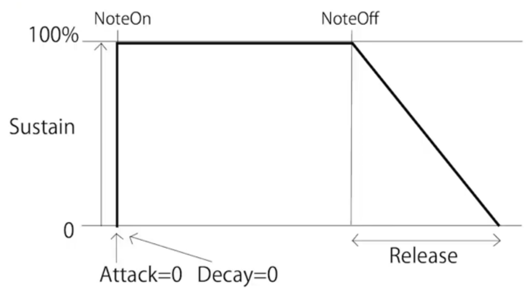
Attack	0.000
Decay	0.000
Sustain	1.00
Release	0.000



To have piano attenuation the sustain level can be lowered and the decay increased. Instead, to have a rising sound similar to an organ the attack can be slowed down by increasing its value. Adding a release value means creating a reverberation once the keyboard is let go.







The envelope applied to a noise wave can create sounds similar to those created by a drum.
If the decay is shortened, a hi-hat sound is created on a high note, meanwhile if it is played on a low note, it sounds like a snare drum.
If the decay is lengthened the sound obtained is a crash cymbal.

Bend Range and Auto Bend

The Bend Range parameter is responsible for the change in pitch of the signal. The bigger the Bend Range is, the greater the change in pitch.
It is expressed in semitone units (i.e., 1 is a semitone, 12 is an octave).
The value that corresponds to no bend is 8192, so the bend parameter is set referring to that default value.

```
currentBendAmount = *(settingRefs-> bendRange) * (
(double)currentPitchBendPosition-8192))/8192.0;
currentPitchSequenceFrame = 0; vibratoCount = 0;
float iniPitch = * (settingRefs->sweepInitialPitch);
float time = * (settingRefs->sweepTime);
currentAutoBendAmount = iniPitch;
autoBendDelta = -1.0 * iniPitch / (time * getSampleRate());
```

Pitch bending can also be automatically applied with the parameter Auto Bend. Its parameters are the *Initial pitch* that represents how far away from the original pitch the pitch starts (-12 means the pitch starts one octave below the original) and is saved in the *currentAutoBendAmount* value, and the *Time* that represents how much time it takes for the pitch to get to its original. This time set the *autoBendDelta* parameter that is responsible for the pitch changes between two sequential frames. The Bend Amount is computed as the difference between the previous note and the midi note, and the autoBendDelta is then computed as the ratio between the Bend Amount and the portamento.

```
if (portamentoTime > 0) {
      currentAutoBendAmount = (double)(previousNoteNo - midiNoteNumber);
      autoBendDelta = -1.0 * currentAutoBendAmount /
      (portamentoTime*getSampleRate());
      }
```

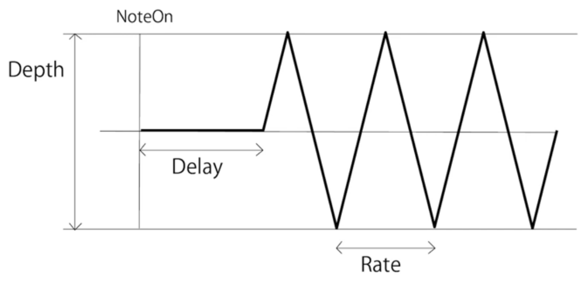
Vibrato

Another important parameter in the plug-in is the Vibrato. Its parameters are the *Rate*, i.e. its speed, the *Depth* in semitone steps and the *Delay*, i.e. the time it takes for the delay to start playing. Applying vibrato means applying a sinusoidal change to the original sound.

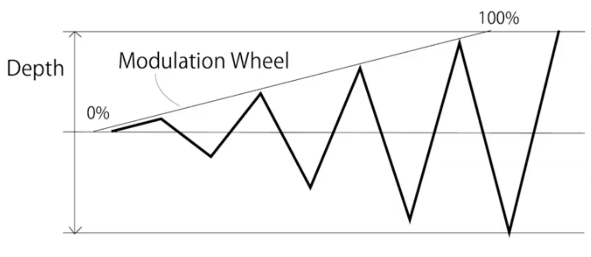
The parameter ***Ignores Wheel*** can be checked. If it is unchecked, the vibrato will respect the value of the modulation wheel (set by the value *bywheel*). On the other side, if it is checked, vibrato will be applied even if the modulation wheel is set at zero (in this case, *bywheel* will be **1.0**).

```
double byWheel = settingRefs->vibratoIgnoresWheel() ? 1.0 :
currentModWheelValue;
double vibratoAmount = * (settingRefs->vibratoDepth) *
sin(getVibratoPhase()) * byWheel;
```

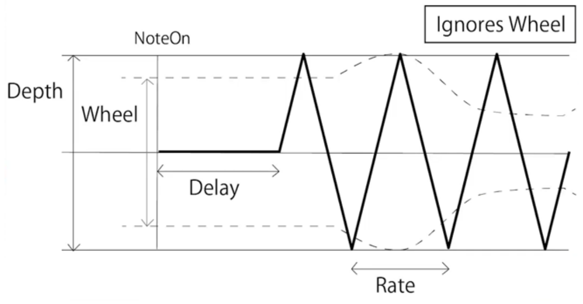
Vibrato description



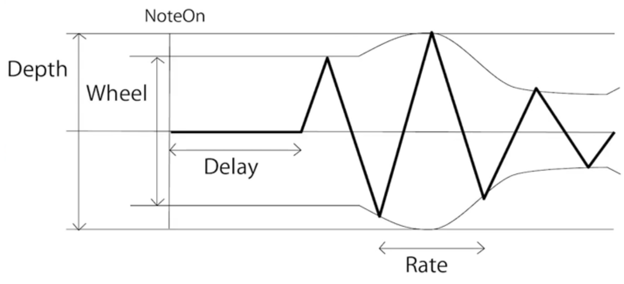
Modulation wheel with no vibrato



Vibrato with ignored wheel checked



Vibrato and wheel



Advanced options

The user has the possibility to specify custom numbers for volume and pitch envelope and duty.
When the Custom Envelope is selected, in *AdvancedParamsComponent.cpp* is created a new *CustomEnvelopeComponent* where it is initialized the volume, the pitch, the duty and the choice component regarding the resolution, by specifying the following parameters: Magical8bitPlug2AudioProcessor& p, String type, String displayName, String flagParameterName.

By checking the different checkboxes, the user will enable custom envelopes, and if it is activated, the normal envelope will be disabled.
The basic format is just placing numbers separating by commas. In the case of ***volume***, zero is silence, so if the last one ends with a non-zero value, the last one is retained while the key is on.
There is a way that allows you to keep the same value for several frame in these advanced options and that is: 15x5f, 0. It keeps the maximum volume for 5 frames and then goes to 0.
You can't do this with a normal envelope, although you can write a tone that goes to zero when you release it, but you can't write a tone that goes to zero on its own after 5 times, unless you use Custom Envelope.
Another useful and interesting notation is the *Repeat* one. If you write [15, 10, 5]the values between parenthesis will be repeated.
Another interesting one is the *Release notation*. You write vertical bars and then it is followed by a change in volume after you release the keys: 15 | 15to0in30.

You can do it in the same way for ***pitch*** as well. For example, if you write decreasing numbers it will sound like the pitch is coming down from the top; whereas if you write it using negative values, it goes up from the lower pitch.
If you put parentheses around an example like this, you can create this kind of jumpy and repetitive tone, which cannot be created with vibrato, which is one of the typical situations that you need Custom Envelope. The interesting thing about this is that it can "coexist" with auto-bend and vibrato values. Custom envelope of volume overrides the normal envelope, but in the case of pitch it can be merged.

There are two types of resolution: ***Coarse*** and ***Fine***.
While *Coarse* is selected, the numbers are in semitone units.
This is implemented by taking the current pitch value.
On the other hand, while *Fine* is selected a value of 8 corresponds to a semitone.
This is implemented by dividing by 8 the current pitch.

```
case kPitchSequenceModeFine:
  finePitchInSeq = (double)settingRefs->pitchSequence.valueAt
(currentPitchSequenceFrame) / 8.0;
               break;
```

```
case kPitchSequenceModeCoarse:
  noteNumberMod = settingRefs->pitchSequence.valueAt
(currentPitchSequenceFrame);
                break;
```

We can use custom envelope for ***duty*** as well. You can write 0, 1 or 2 values here and they correspond to: 0 is 12.5%, 1 is 25% and 2 is 50%.