Overview

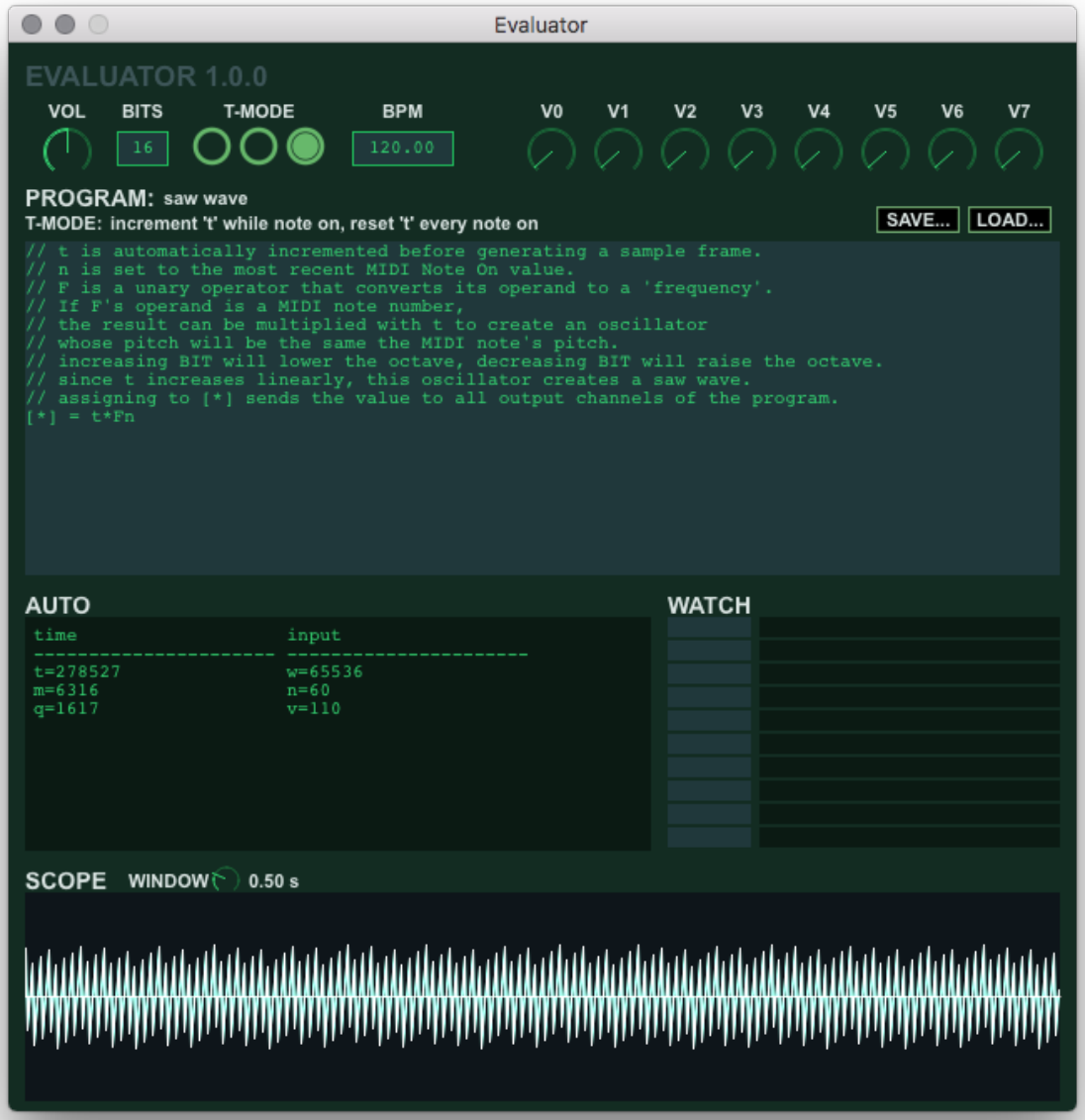
Evaluator allows you to write short C-style programs whose output is used to generate sound.

It is inspired by bytebeat, a "genre" of music discovered by viznut, which he documented in several youtube videos.

Evaluator's approach, however, is not purist, and its language is *not* C. The language contains much of the same syntax, most of the operators, and the same operator precedence, but introduces some additional features that make generating musical sounds a little bit easier and also make real-time MIDI control of the program possible. It also includes several built-in presets that demonstrate all the language features, supports save/load to fxp, and loading a program from a plain text file.

The basic idea is that a program operating on 64-bit unsigned integers is used to generate every audio sample. The program essentially runs in a while loop that automatically increments the variables *t, m,* and *q* before executing the program code each time. These will be described in more detail below.

Interface


Evaluator
EVALUATOR 1.0.0
VOL
BITS
16
T-MODE
BPM
120.00
V0
V1
V2
V3
V4
V5
V6
V7
PROGRAM: saw wave
T-MODE: increment 't' while note on, reset 't' every note on
SAVE...
LOAD...
// t is automatically incremented before generating a sample frame.
// n is set to the most recent MIDI Note On value.
// F is a unary operator that converts its operand to a 'frequency'.
// If F's operand is a MIDI note number,
// the result can be multiplied with t to create an oscillator
// whose pitch will be the same the MIDI note's pitch.
// increasing BIT will lower the octave, decreasing BIT will raise the octave.
// since t increases linearly, this oscillator creates a saw wave.
// assigning to [*] sends the value to all output channels of the program.
[*] = t*Fn
AUTO
time
input
t=278527
w=65536
m=6316
n=60
q=1617
v=110
WATCH
SCOPE
WINDOW
0.50 s

VOL

The VOL knob controls the overall volume of the output. Since the output of programs can often be quite harsh, it is usually nice to keep this around 50%.

BITS

The BITS value describes how the output of the program is interpreted in terms of bit depth. In other words, should the program output be treated like 8-bit audio, 16-bit audio, 2-bit audio, etc. Programs operate on 64-bit unsigned integers to allow for the largest mathematical space possible, but the output values are wrapped to [0, 1<<BITS] and then converted to [-1,1] floating point values. The "wrap" value is automatically set in the *w* variable, which can be seen in AUTO window in the screenshot above, because it is used by some of the language's unary operators, but it is also often useful within a program.

T-MODE

The T-MODE setting controls how the variable *t* will be automatically incremented. In the Standalone version, there are three possibilities:

- increment *t* every audio sample, regardless of the current input to the program
- increment *t* every audio sample only when there is at least one active MIDI Note On
- increment *t* every audio sample only when there is at least one active MIDI Note On, but reset *t* to zero every time a new Note On is received

When running in a DAW, a fourth option is available:

- set *t* to the current project time in samples

This fourth option is particularly useful when a program's output changes a lot over time and you want to be able to jump directly the output that would result at a particular point in time. The first setting will work similarly if you start playback in the DAW from the very beginning of the project, but it won't work if you start playback somewhere in the middle, due to the fact that DAWs typically reset plugins when playback begins, which will cause *t* to start counting up from zero.

This mode also impacts when *m* and *q* are incremented, since these are based on the current value of *t*:

- *m* how many milliseconds the current value of *t* represents
- *q* is how many 128th notes the current value of *t* represents

BPM

The BPM setting is only available in the Standalone version. When running in a DAW the BPM will always match the project tempo. In both cases, BPM controls how *q* is calculated from the current value of *t*.

V0 – V7

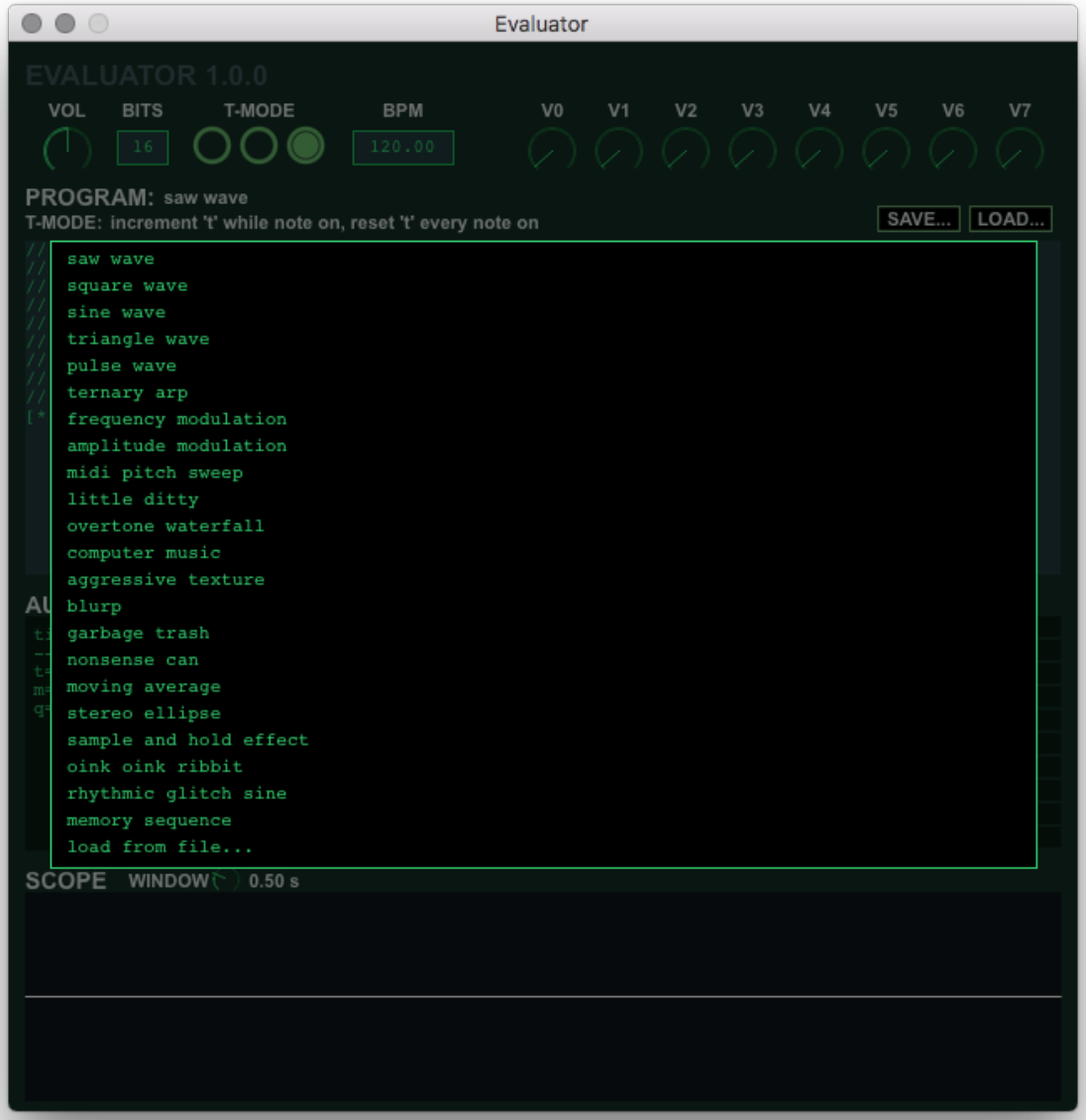
The V knobs are generic controls whose values can be accessed by the program by using the *V* unary operator. The knobs range in value from 0 to 255 and can be automated when running in a DAW.

SAVE

The Save button can be used to save the entire state of the interface to an fxp file.

LOAD

The Load button opens an overaly that allows for recall of several built-in presets, as well as the option to "load from a file...". When loading from a file you can choose either a plain text file, whose contents will replace what is in the Program Window, or from an fxp that was previously created with the Save button.



PROGRAM

The Program Window is where the program itself is edited. Click anywhere in the window and type as you would in a text editor. To compile the program so that it can be used to generate sound, either click outside of the Program Window or press Ctrl+Enter (Cmd+Enter on Mac).

AUTO

The Auto Window is used to display program variables that are automatically updated by the app:

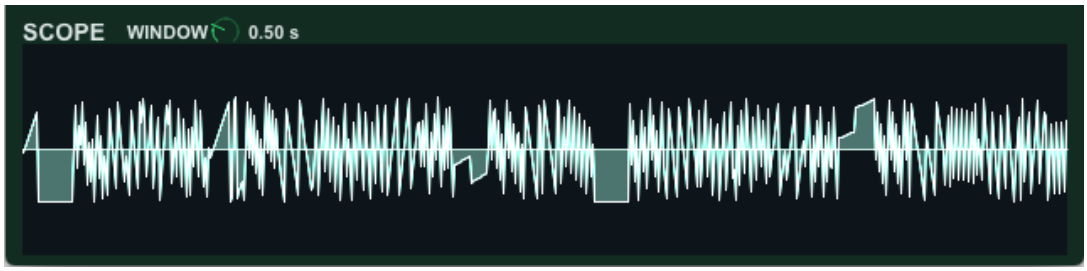
- *t* is incremented every audio sample, based on the current T-MODE
- *m* is how many milliseconds the current value of *t* represents
- *q* is how many 128th notes the current value of *t* represents
- *w* is the value the program output will be wrapped to before being converted to [-1,1] floating point values. It will always be equal to 1<<BITS.
- *n* is set to the most recent MIDI Note On note that has been received. It will be 0 if there aren't any active notes, and will never be greater than 127.
- *v* is set to the velocity of the most recent MIDI Note On and will always be in the range [0,127].

WATCH

The Watch section can be used to monitor values in program memory that are not present in the Auto Window. These include user-defined variables, user memory addresses, V knobs, and MIDI CC values. We'll return to this after discussing all of the language features (ie how to write programs).

SCOPE

The Scope displays the output of the program in the style of an oscilloscope. The Window knob can be used to control length of time in seconds that is being visually represented.



Program Syntax

Basics

Evaluator's language is largely based on C but lacks control structures, functions, types, arrays, and memory management. There are no types because every value in a program is a 64-bit unsigned integer. There is no memory management because all program memory is allocated when it is compiled and accessed through various means. There are no control structures, functions, or arrays because these are hard to implement.

A program must *output* values in order to be a valid program. The simplest program you could write that has non-constant output might look like this:

```
[*] = t;
```

This program assigns the current value of *t* to all outputs. The line is terminated with a semi-colon, as in C-style languages, although the final line of a program doesn't require this. Here's a program that assigns different values to left and right outputs:

```
[0] = t;
[1] = -t;
```

Neither of these programs will produce audible output because they are both essentially generating very low frequency saw waves (recall that program output is wrapped to 1<<BITS).

In order to hear anything interesting, we have to write some kind of arithmetic expression. Here's a surprisingly simple one from viznut's collection that sounds nice with BITS set to 8:

```
[*] = t&t>>8
```

All of the arithmetic and bitwise operators from C are available in Evaluator's language. The main thing worth noting is that bitwise left shift and bitwise right shift will shift by the *right-hand operand modulo 64*, so it is not possible to over-shift the left-hand side.

A program can also use *[0]* and *[1]* to access audio coming *into* the program from the Host. This could be microphone input in the Standalone app or it could be the output of whatever comes before Evaluator in an effects chain in your DAW. This program swaps the left and right channels of the incoming audio:

```
a = [0];
b = [1];
[0] = b;
[1] = a;
```

Finally, two forward slashes can be used to add comments to a program:

```
// this is a comment
[*] = t*128 // also a comment
```

Variables

The small programs above access the variable *t,* which has its value set automatically before every execution of the program, however *any lowercase letter* can be used as a variable in a program. Variables don't need to be declared before being used, their value will be zero until assigned a new value by a program. Values assigned to variables will persist from one execution to the next, with the exception of variables in the Auto Window, whose values are sometimes changed before a program is executed, but there's nothing stopping a program from changing their values during execution. Here's a program that increments a variable every execution:

```
a = a + 1;
[*] = a;
```

The output of this program across 10 executions will be: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Once a program becomes sufficiently complicated, it can be nice to be able to monitor the value of variables that are in use. Any variable name can be entered into one of the Watch rows to do this.

Ternary and Relational Operators

Evaluator's language includes a ternary operator and two relational operators. Using these it is possible to achieve behavior that is similar to branching. Consider the "ternary arp" preset:

```
// q is incremented every 128th note, so we can divide by 32
// to get a number that increments every sixteenth note.
s = q/32;
// if s is odd, we want to increase the note by an octave.
a = n + (s%2 ? 12 : 0);
[*] = t*Fa
```

Since the language does not have a boolean type, the ternary operator will resolve to the second operand if the first operand is non-zero and to the third operand if the first operand is zero. In this example, it suffices to use *s%2* to get a steady alteration, since that expression on its own oscillates between 0 and 1. When dealing with a larger range, we can use < or > to create a 0 or 1 value. Here's an example that turns a saw wave into a pulse wave using this method:

```
a = (t*F60)%w;
[*] = a < 8800 ? w-1 : 0;
```

Of course, it is also possible to use the result of a relational operator arithmetically. The second line of the above program could be more succinctly written:

```
[*] = (a < 8800)*(w-1);
```

An important detail to remember when using the ternary operator is that *it doesn't actually branch*. All three operands of the ternary operator are evaluated before the final value is determined. This means that if part of the ternary operation changes the value of a variable or memory address, it will *always* change that value, even when the result is not used. In this program, *a* is incremented every execution, even though the output of the program is sometimes zero:

```
[*] = a%128<1 ? a=a+1 : 0;
```

Unary Operators

Evaluator's language includes only the +, -, ~, and ! unary operators from C, but adds additonal operators to aid in the creation of short programs that produce musical ouput.

F will convert a MIDI note number to a "frequency" suitable for generating a sound with the same pitch. This program will produce a saw wave with a pitch of C, but the octave will depend on the BITS setting:

```
[*] = t*F60
```

will "square" a value by changing it to 0 if is less than w/2 or 1 if not. This program will produce a square wave with a pitch of C:

```
[*] = #(t*F60)
```

$ will "sine" a value by remapping it to a sine wave with a period of *w*. This program will produce a sine wave with a pitch of C:

```
[*] = $(t*F60)
```

T will "triangle" a value by remapping it to a triangle wave with a period of *w*. This program will produce a triangle wave with a pitch of C:

```
[*] = T(t*F60)
```

R will generate a random value between 0 and its operand. This program will sound like white noise:

```
[*] = Rw
```

V will retrieve the value of one of the corresponding V knobs. So *V4* will be the value of the the V4 knob and *Va* will be the value of the knob corresponding to the current value of *a*. The operand value will be wrapped to the number of knobs, so you'll always get a useful value out of this. Here's a program that uses the value of the V0 knob to control the speed of a "modulator":

```
[*] = t*Fn + $(t*V0)
```

C will retrieve the value of the corresponding MIDI CC value. So *C1* will be the value of the Mod Wheel CC (see: https://www.midi.org/specifications/item/table-3-control-change-messages-data-by...). Similarly, *Ca* will access the MIDI CC value corresponding to the current value of *a*. Like with the V operator, the operand is wrapped to the number of CC messages, which is 128. Here's a program that uses the Mod Wheel CC to sweep an oscillators pitch up:

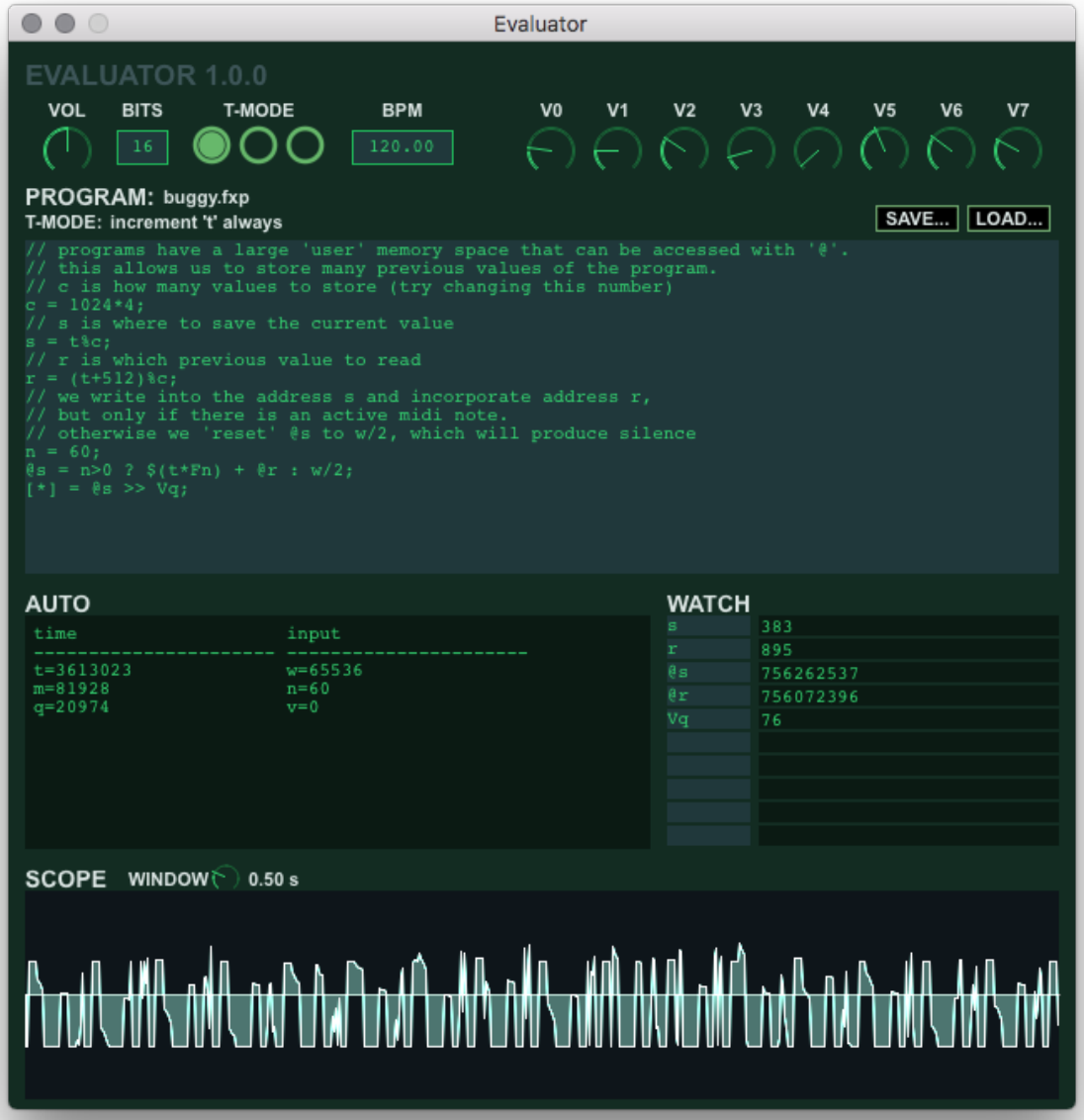
```
a = Fn + (F(n+12) - Fn)*C1/127;
o = n>0 ? o + a : w/2;
[*] = o;
```

@ will use the value of the operand to allow access to "user" memory, which contains 65536 contiguous values that can be read from or written to, just like variables. This is most useful if you want to store "arrays" of values that the program can access by index. The memory for variables immediately follows this "user" memory, so *@65536* is the same as *a*. The operand will be wrapped to the full size of the program's memory space before being used for access to prevent out-of-bounds access that might crash the app. Here's the "memory sequence" preset, which demostrates this operator nicely:

```
@0 = 0; @1 = 4; @2 = 7; @3 = 12;
i = q/32 % 4;
[*] = n>0 ? t*F(n+@i) : w/2;
```

Watches

It can be helpful to monitor variables in a running program to understand what is going on or to debug why something isn't working as expected. The Watch section of the interface provides 10 slots where you can enter variable names and some simple expressions. The unary operators *V, C,* and @ can be used in Watch rows in conjuction with a number or variable name. *V0, Va, C15, Cb, @128,* and *@c* are all legal Watch expressions.



Program Errors

A program can produce compliation errors and runtime errors. Compilation errors occur when there is a mistake in the program text that prevents compiling it into a set of instructions. Runtime errors occur when executing a compiled program fails for some reason (usually due to divide by zero). In both cases, the error will be shown in the Auto Window instead of the usual program state.