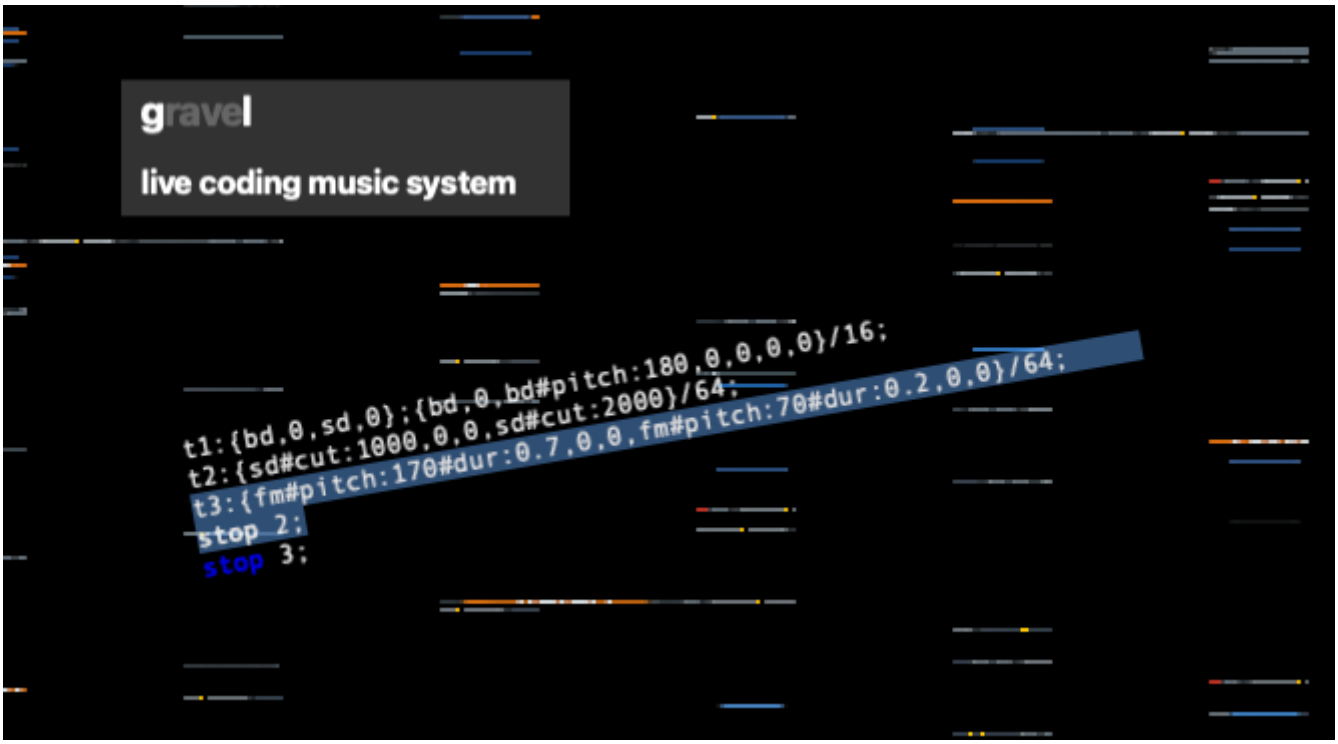
Gravel

manual v0.1 alpha

gravel



- state: alpha version soon available

Gravel was originally started in September 2022 by Tina Mariane Krogh Madsen and Malte Steiner to create a live coding system for their own practises such as the project codepage.

Live coding is the most awkward way to operate a musical sequencer, instead of recording the pressing of buttons and twiddling knobs you write and evaluate code which unfolds to rythmical patterns and bars. So why do you want to do it? What it lacks on physical expressivity, it offers expressivity on code level. With a little code complex polyrythmical sequences can be created which evolves over time. The Github Wiki contains the documentation of Gravel language, but also a PDF will be available.

So basically Gravel is a sequencer with a code editor as GUI. It evaluates the given code and triggers a synthesizer which is implemented in the included Csound. The textfile **csoundInstruments.csd** contains the instrument definitions and is loaded by Gravel on program start. It can be, with some Csound knowledge, customized. It also defines the available parameters which can be automated in the sequencer.

Gravel is written in C++ with the library QT. An alphaversion is soon available for Linx, Mac and Windows. Gravel is in its early stage and not even beta. But it was alreafy successfully used by Malte Steiner in a concert at Piksel 2022 festival in Bergen, NO and will be presented at the Code&Share event in Aarhus 10. December 2022.

What's missing so far and soon will be implemented:

- an easier way to configure the audio device, so far the Csound script itself needs to be adapted
- Gravel now provides several synthesized instruments which can be modulated via several parameters. More instruments are going to be provided inclusive samplers and granular instruments

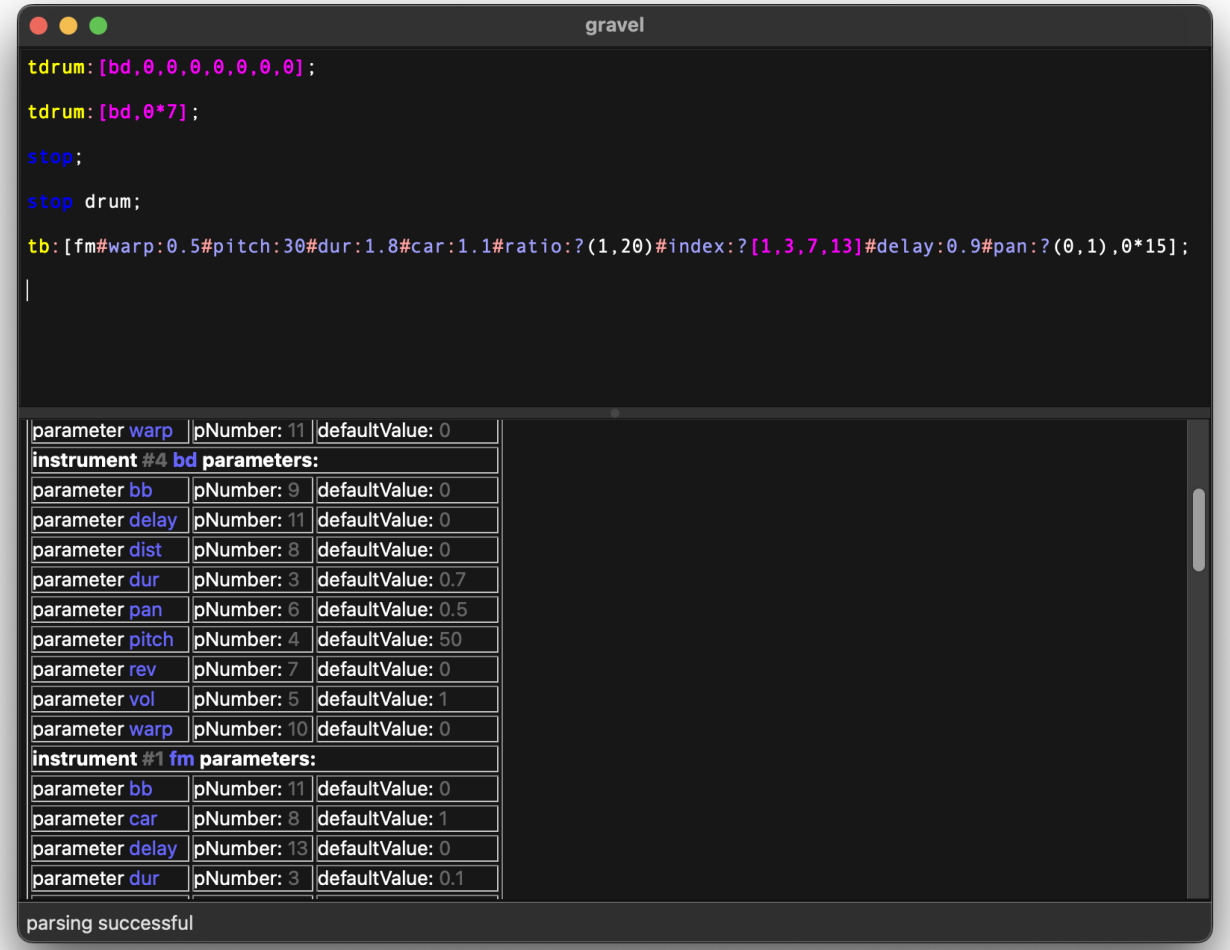
- a way to record audio on command, either the stereo sum or individual busses (synths, drums, fx) as multitrack WAV file
- midi output
- synchronization between other computers and studio equipment
- the domain specific language Gravel is now in the process of being defined, implemented and tested. After Version 1.0 no more breaking changes, only additions

You can support this project by checking out our other music from our label block 4.

Gravel Live Coding system

Gravel is a live coding system consisting on the sequencer and its editor via which the user controls the sequencer via its own domain specific programming language. The sound is generated by the buildin synthesizer which is made with the audio system Csound which loads an instrument definition file at program start. That script defines the different instruments, like bass drum, snare drum etc., which can be triggerd by the sequencer. With some experience in Csound language, the user can edit that file and create custom instruments and effects, and load it at the next program start without the need to recompile the Gravel application.

Additional to the instrument name which is used in Gravel to reference that instruments, various parameters can be also defined to control several aspects like the pitch and duration from the sequencer, which is one of the core features of Gravel. Its easy to vary or randomize these parameters in the sequencer and create so with little code complex, evolving patterns.



The top section with the syntax highlighted code is the main area in which the user types in and evaluates Gravel code. The contents can be stored as a textfile and later opened or merge loaded to existing code via the file function in the menu.

Below is non editable output area which contains useful information like the loaded instrument definition and all the parsed parameters which can be used in the Gravel sequencer. The blue colored names for instruments and parameters are the ones to use as reference in the sequencer.

Getting Gravel to start with the right audio card is a bit complicated because for now the setting can be only done in the csound instrument file **csoundIntruments.csd** The settings can be found on top in the CsOptions section. Gravel put out information about the available audio devices in the non editable section, on top of the table with the instrument and parameter table. For Linux this should simply work if you have JACK running:

```
<CsOptions> -+rtaudio=jack -odac </CsOptions>
```

Alternative is running direct on ALSA, these settings take the first available audiodevice:

```
<CsOptions> -+rtaudio=alsa -odac:hw:0,0 -g -b 512 </CsOptions>
```

For Mac following settings might work, -odac0 points to the first found device in the list. If other are prefered, the 0 has to be replaced with the correct number:

```
<CsOptions> -+rtaudio=auhal -odac0 -g -b 512 </CsOptions>
```

For Windows following might be enough, -odac0 points to the first found device in the list. If other are prefered, the 0 has to be replaced with the correct number

```
<CsOptions> -+rtaudio -odac0 -g -b 512 </CsOptions>
```

Helpful are the Csound Windows documentation

tracks, patterns and events

This sequencer software uses its own domain specific programming language Gravel to create complex sequences, grouped in tracks. Unlike traditional sequencers, tracks in Gravel are not bound to a specific instrument. A track contains one or more patterns which can trigger all available instruments. The purpose of a track is to group these patterns under one name so that they can be stoped together while other tracks continue to play.

Due to the way Csound works, all intruments have unlimited polyphony, they can be triggered and layerd without limits except your CPU. So triggering an instrument in a fast way with the duration set to long time will not only add up the volume but also tax your computer and could lead to drop outs when your CPU can't calculate everything in time.

A simple beat with a bassdrum on first note would be

```
tdrum:[bd,0,0,0,0,0,0,0];
```

Let me explain it from left to right: We create a new track by beginning with the letter **t** and a name which should consist only of letters and numbers, no other signs. **:** signifies that we are done with the name, here drum, and the actual sequence is about to start. The more traditional way are sequences embraced by **[ ]**. A track, like also the other available commands in Gravel, ends with an **;**

Each position fills the pattern and either takes an instrument name, here **bd**, or **0** for a silent pause. In the example it is one event for the instrument **bd** and 7 times pause which results in a persistent beat on the default 140 bpm. After the last event the pattern is repeated so in other words the length of the pattern is defined by the amount of events. The part is evaluated by being in that line with the cursor and press **shift** and **return** or, in case several lines are about to be activated, by selecting the lines completely and then press **shift** and **return**.

That above can be written faster with the multiplier which can be applied to each position:

```
tdrum:[bd,0*7];
```

When a line with a track of the same name is activated with shift + return, it replaces the preceding definition but the change is not imediant, its done after the pattern played its last event or pause.

To stop a specific track you can type the command **stop** followed with the name of the track (exclusive the beginning **t**) and a **;**

```
stop drum;
```

To stop all playing tracks at once use just **stop** and **;**

```
stop;
```

Another way to define patterns with Gravel follow Euclidian ideas, here a bar with a fixed length is filled equally with the provided events. That type is signified with curly brackets **{ }**. So

```
tbdrum:{bd};
```

is basically the same as the example above. The amount of clockbeats the content is destributed can be changed with **/** , default is 16. So this

```
t1:{bd}/32;
```

is slower because the single event is distributed over 32 clockbeats.

```
t1:{bd}/8;
```

is therefore faster Here the trackname is just **1**.

As mentioned above, several patterns can be combined in one track:

```
tCombined:[bd,0*7],{0*3,sd,0,sd};
```

the speed of the sequencer can be changed with the command **bpm**, default are 140bpm. This changes it to 120bpm after being evaluated with **shift** and **return**:

```
bpm 120;
```

parameters and automation

One main advantage of Gravel are the multiple ways to sequence parameters of the instruments. Parameters are defined together with the instruments in the Csound script which is loaded when Gravel starts. The application scans for the definitions and make them available for the sequencer.

In the non editable part of Gravel below the code editor the available instruments and parameters are listed. The names colored in light blue are the ids you can use to refer in your sequence. For instance there is a parameter pan available for all instruments, affecting the stereo panorama position, range is 0.0 (left) - 1.0 (right) with 0.5 being the center and default when no parameter is written in the sequence. Preceding is the parameter sign **#** then parameter name, then a **:** and the value.

So

```
tsynth:[bd#pan:0,0*15];
```

creates a simple bass drum beat on the left speaker.

```
tsynth:[bd#pan:0,0*7,bd#pan:1,0*7];
```

adds another bass drums on the right side, with 7 times pause in between.

Parameters can be stacked, separated with the **#**.

```
tsynth:[bass#pitch:70#pan:0.3,0*15];
```

creates a Moogy synth sound, slighlty off center to the left. When you add more synth trigger events to your pattern with the same parameter settings, you don't have to repeat them, the last used ones can be repeated with the parameter signifier **#** and **$**.

```
tsynth:[bass#pitch:70#pan:0.3,0*3,bass#$,0*7];
```

You can still add new parameters or overwrite one or several of the copied parameters. For example

```
tsynth:[bass#pitch:70#pan:0.3,0*3,bass#$#pan:0.9#rev:0.5,0*7];
```

overwrites the copied panorama parameter and add a reverb parameter for the second synth event.

So far we parameters to a concrete value per step/ event. But there is more possible, i.E. parameter sequences:

`tsynth:[bass#pitch:[180,80,50,50],0*3];` triggers the bass synth a bit faster, but each time take another value for pitch from the list enclosed by the square brackets. The values are taken sequentially, kind of a sequence in the sequence.

Its also possible that the values are picked randomly from the list by preceding the list with an **?** which invokes the randomizer:

```
tsynth:[bass#pitch:?[180,80,50,50],0*3];
```

Another way to invoke the randomizer is to give a range of values. Ranges are encapsulated with **()** brackets and the first and last value are taken for min and max. The randomizer in this case also generates values between:

```
tsynth:[bass#pitch:?(50,100),0*3];
```

this generates a bass sequence with the second event randomly panned in stereo:

```
tsynth:[bass#pitch:?(50,100),0*3,bass#$#pan:?(0,1),0*3];
```