clive workshop

Claude Heiland-Allen

2022-01-15

clive workshop

Introduction

Claude

My name is Claude Heiland-Allen. I'm from London, UK. I'm interested in art, computers, free software, maths, science.

- https://mathr.co.uk
- https://mathr.co.uk/blog
- mailto:claude@mathr.co.uk

clive

I perform live music by coding in the C programming language, to manipulate audio processing algorithms while they are running.

- https://mathr.co.uk/clive
- https://mathr.co.uk/blog/livecode.html
- https://code.mathr.co.uk/clive-workshop

today

- Install Party
- Digital Audio
- Time-based Synthesis
- Feedback Processes

Install Party

website

This presentation is online:

https://mathr.co.uk/clive/workshop/2022-01-15.html

https://mathr.co.uk/clive/workshop/2022-01-15.pdf

Access it now so you can copy/paste.

roadmap

- install dependencies
- get `clive` using `git`
- test

Linux Sound

"The nice thing about standards is there are so many to choose from"

- OSS - Open Sound System
- ALSA - Advanced Linux Sound Architecture
- JACK Audio Connection Kit ("pro" audio)
- PulseAudio ("consumer" audio)
- PipeWire (the future, unifies all of the above)

clive-server Audio

clive-server was originally JACK-only.

But now it can work with other APIs too, via SDL2.

Configuring Linux Sound

Hopefully you have sound working already.

Goal 0 of this workshop:

- don't break your working configuration, even if it is not "perfect".

Command Line Terminal

Most of the installation and configuration will require using the terminal shell to enter text commands.

Important commands include:

- `ls` – lists the files in the current working directory
- `mkdir ~/code` – makes a directory called code in your home folder
- `cd ~/code` – changes the current working directory

Install Dependencies

For Debian (and other apt-based distributions like Ubuntu):

```
sudo apt install \
    git ca-certificates \
    build-essential pkg-config \
    htop xterm geany nano \
    libsdl2-dev libjack-jackd2-dev
```

Other distributions should have similar packages.

git

`git` is a distributed version control system for managing evolving changes to text documents, primarily computer source code.

We will be working offline, after the initial clone operation.

`git` needs to know who you are when commiting your code changes:

```
git config --global user.name "My Name"
git config --global user.email "my@email"
```

If you publish your repository, this will be made public.

xrandr (optional)

clive window sizes are optimized for 1920x1080 screen.
Check the identifiers of your displays:

```
xrandr
```

My laptop has only 1366x768 display. I fake it:

```
xrandr \
  --output LVDS-0 \
  --scale-from 1920x1080 \
  --same-as HDMI-0
```

This is temporary (until logout).

Get clive

`clive` consists of a server, client, launch scripts, and user code, all in one repository:

```
mkdir -p ~/code
cd ~/code
git clone \
  https://code.mathr.co.uk/clive-workshop.git
cd clive-workshop
```

Build clive-server

clive-server is the part that runs the code and makes the sound.

If you know you are using JACK:

```
make -C server API=jack
```

otherwise you are probably using PulseAudio:

```
make -C server API=sdl2
```

Build clive-client

clive-client is the part that watches for code changes and runs the compiler.

The compiler converts source code text that humans can understands into machine code that the CPU can understand.

```
make -C client
```

Test clive

To test:

```
cd ~/code/clive-workshop
git checkout metronome
./launch/local-native-sse.sh
```

You should hear regular beeping.

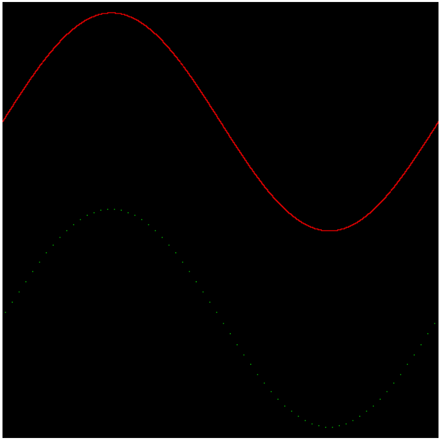
To exit, hit Ctrl-C in the terminal you started the launch script in.

Break time

Take a short break.

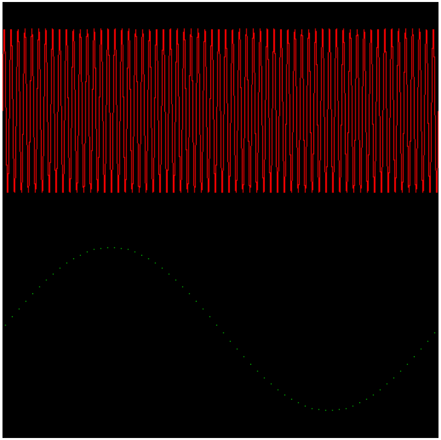
Digital Audio

Digital audio



- Real world is continuous.
- Digital world is discrete.
- Sample points equally spaced in time.

Aliasing

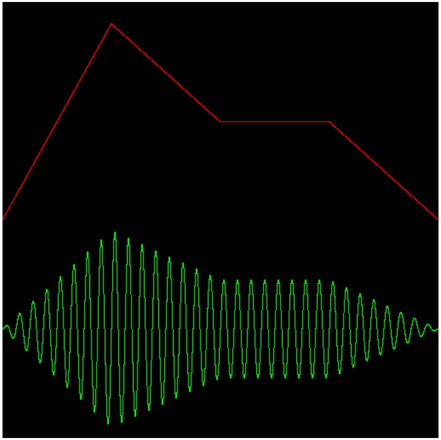


- Shannon-Nyquist sampling theorem says when this approximation is ok.
- Too-high frequencies fold over back into lower frequencies.

Audio properties

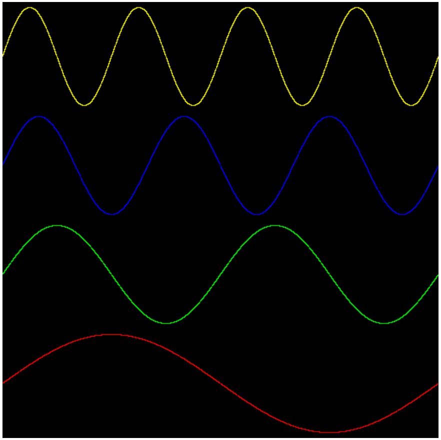
- Volume
- Frequency
  - Pitch
  - Rhythm
- Timbre
  - Tones
  - Noise

Volume



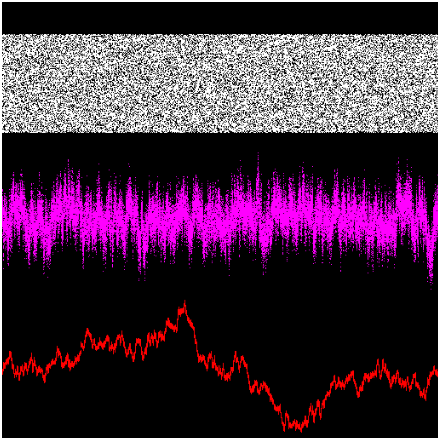
- Larger values are louder
- Logarithmic perception
- Adding 10 decibels increases level by 10 times

Pitch



- Higher frequencies are higher pitch
- Logarithmic perception
- Adding 1 octave increases frequency by 2 times

Noise



- White noise: equal energy per frequency
- Pink noise: equal energy per octave (3 dB/octave falloff)
- Brown noise: low-pass filtered white noise (6 dB/octave falloff)

Audio processing

- Stateful transformation over time.
- Volume control: no history

```
out = V * in
```

- Phasor: internal state

```
phase = wrap(phase + in * factor)
out = phase
```

- Recursive filters: internal state and feedback

```
state, out = function(state, in, previous_out)
```

Architecture

- user code
  - defines audio processing algorithms
- client
  - watches source code directory
  - recompiles to shared library
- server generates audio
  - watches build output directory
  - loads shared library
- launch scripts

Implementation details

- Stack memory is temporary (in, out)
- Heap memory is preserved (state, `S *s`)
- Function is called every sample (`go()`)
- Saving the file triggers recompilation
- Successful compilation triggers code hotswap

Comparison to other software

- Pure-data
  - variable block size down to 1 sample, plus `[fexpr~]`
  - deterministic
  - xruns when recompiling modified DSP graph
- SuperCollider3
  - fixed block size
  - realtime safe
  - sometimes unpredictable latency
- Transfer of knowledge: maths / dataflow / ugens

Workshop goals

0. don't break your working system
1. `git` version control system basics
   - clone, status, commit, branch, tag
2. C programming for audio
3. digital audio processing

Example track

https://mathr.co.uk/clive/workshop/2022-01-15/claude.html

time-based synthesis

metronome

Launch clive:

```
cd ~/code/clive-workshop
git checkout metronome
./launch/local-native-sse.sh
```

If using JACK or PipeWire: useful to have qjackctl or other monitor open.

windows

- The terminal where you started clive from.
- `server` top left, status display in case of crashes.
- `client` middle left, displays compilation messages. Pay attention to this window in case of mistakes.
- `htop` bottom left, system monitor, useful to have.
- `Geany` text editor where the main action happens.

edit

- In `Geany`, change the BPM and hit Ctrl-S to save. You should hear the tempo change (after a short delay; learn the latency of your machine).

  You should see messages in the `client` window, hopefully no errors.

- Change the Hz frequency and hit Ctrl-S to save. You should hear the pitch change.

safety

- Adjust the volume control of your soundcard to very low.
- In `Geany`, set the volume to 10.0 (!) and save.
- Increase the main volume as much as is comfortable.
- In `Geany`, adjust the volume less than 1.0 again.

code structure

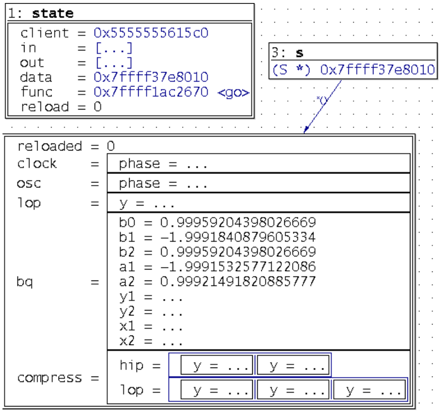
At the top of `go.c` are some boilerplate definitions.

Then the memory layout, `typedef struct { ... } S`. The first member of the `struct` must be an `int`. It gets set to 1 when the code is reloaded.

Finally the callback function `go()`. It gets passed the memory as an `S *` (pointer to `S`), as well as the audio buffers and their sizes.

The other files loaded into `Geany` are to make code completion in the editor work better.

memory layout



```
1: state
client = 0x5555555615c0
in     = [...]
out    = [...]
data   = 0x7ffff37e8010
func   = 0x7ffff1ac2670 <go>
reload = 0

3: s
(S *) 0x7ffff37e8010

*()

reloaded = 0
clock    =  phase = ...
osc      =  phase = ...
lop      =  y = ...
bq       =  b0 = 0.99959204398026669
            b1 = -1.9991840879605334
            b2 = 0.99959204398026669
            a1 = -1.9991532577122086
            a2 = 0.99921491820885777
            y1 = ...
            y2 = ...
            x1 = ...
            x2 = ...
compress =  hip = | y = ... | y = ... |
            lop = | y = ... | y = ... | y = ... |
```

- memory is just a list of numbers
- `struct` defines the memory layout
  - names
  - meaning
- structs can be nested (e.g. COMPRESS has arrays of HIP, LOP).

Example struct

The image in the previous slide corresponds to:

```
typedef struct {
  int reloaded;
  PHASOR clock;
  PHASOR osc;
  LOP lop;
  BIQUAD bq;
  COMPRESS compress;
} S;
```

changing struct

It's only safe to add things at the bottom.

Deleting, inserting or reordering items can cause problems.

This is because the underlying memory is not updated to match the new description.

C language

Most spaces are insignificant.

Statements end in `;`

Blocks are wrapped in `{ }`

Declarations statements have a type (usually `sample`) and a name optionally followed `=` by an initializer (then `;`).

Operators include: `= == < > + - * /`, full list at: https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B#Operator_precedence

C comments

Comments are useful for disabling code without deleting it.

One-line comments start with `//`.

Block comments are wrapped in `/* */`, they don't nest.

Larger blocks can be disabled with `#if 0`:

```
#if 0
...
#endif
```

Re-enable with `#if 1`.

errors

It is easy to make a mistake.

Watch the `client` window. The first error is the most important, you may need to scroll up with Shift-PgUp. Line numbers tell you where the error is.

Most mistakes are typos. Look at the relevant line carefully.

If your edits are not changing the sound at all, you might have an error.

common mistakes

- `error: expected ';' before ...`
  - add a ;
- `error: expected ... before ')'`
  - delete a) or add a (
- `error: expected ')' before ...`
  - add a) or delete a (
  - typo similar to `a 8 b` (should be `a * b`)

spelling mistakes

- `error: stray ‘\###’ in program`

  `error: invalid suffix ...`
  - names are ASCII `A-Za-z_0-9`, no first number
- `warning: unused variable ‘...’`
  - you probably used something else instead
- `error: ‘S’ has no member named ...`
  - you forgot to add the item to the `struct`
- `error: duplicate member ‘...’`
  - you added an item to a `struct` more than once

declaration mistakes

- `error: '...' undeclared`
  - you forgot to declare the variable with `sample`
- `error: redefinition of '...'`
  - each name should only be declared once
- `error: invalid initializer`
  - initializing an array needs `{ ... }`

scope mistakes

- `error: expected identifier or '(' before`
  - usually missing `{`
- `error: expected declaration or statement`
  - usually missing `}`
- `warning: this 'if' clause does not guard`
  - usually missing `{ ... }` around `if` body
- `warning: suggest parentheses around assignment used as truth value`
  - you used = instead of == inside `if()`

structure mistakes

- `error: ‘...’ is a pointer`
  - replace `.` with `->`
- `error: invalid type argument of ‘->’`
  - replace `->` with `.`
- `error: incompatible type for argument ...`
  - often, replace `s->item` with `&s->item`
- `error: label ‘s’ used but not defined`
  - replace `&&s` with `&s`

function mistakes

- `error: too few arguments to function '...'`

  `error: too many arguments to function '...'`

  - check the function prototype in Geany by typing a (after the function name

- `warning: passing argument ... incompatible pointer`

  - you probably meant a different `struct` member

crash errors

- `Illegal instruction ./clive-server`

  `Floating point exception ./clive-server`
  - integer division by 0
- `Segmentation fault ./clive-server`
  - accessing memory out of range
  - array index too big or negative
  - often caused by inserting in middle of `struct`
- `warning: array subscript ... above bounds`
  - your `[number]` is too large – index starts at 0

out

The bottom of `go.c` fills in the output arrays:

`out[0]` is left audio

`out[1]` is right audio

Output signals should be between -1 and 1.

See `qjackctl` $\rightarrow$ connections window.

bpm

A `sample` is a number supporting fractional values:

```
sample bpm; // in struct
```

Choose the tempo in beats per minute:

```
s->bpm = 125; // in go()
```

To convert BPM into (16 beats) per second:

```
sample hz_for_4_bars = s->bpm / 60 / 16;
```

Remember points if a value isn't already a `sample`:

```
125   / 60   == 2
125.0 / 60.0 == 2.0833333333333335
```

phasor

In `struct` section, the `PHASOR` preserves the phase between function calls:

```
PHASOR clock;
```

In `go` section, this advances the phase one sample and returns it into `t`:

```
sample t = phasor(&s->clock, s->bpm / 60 / 16);
```

Audio oscillators have phase too:

```
PHASOR osc;  // in struct
sample p = phasor(&s->osc, 440); // in go()
```

wrap

The clock phase ramps up from 0 to 1 every 4 bars:

```
sample t = phasor(&s->clock, s->bpm / 60 / 16);
```

To make a ramp for 1 bar:

```
sample bar = wrap(4 * t);
```

To make a ramp for 1 beat:

```
sample beat = wrap(4 * bar);
```

Fractional multipliers are also possible:

```
sample beat = wrap(8.0 / 3.0 * bar);
```

envelope

A volume envelope is the variation of volume over time. The beat phase can be used, it fades in:

```
sample env = beat;
```

Another variation is either on or off, a rectangle pulse:

```
sample duty = 0.25;
sample env = beat < duty;
```

A more percussive envelope:

```
sample env = 1 - beat;
```

`beat` ramps from 0 to 1, so `env` ramps from 1 to 0.

pow

Percussive volume envelopes start loud and end quiet:

```
sample env = 1 - beat;
```

Envelopes sound better with a bit of a curve:

```
env = pow(env, 4); // try one at a time
env = pow(env, 8);
env = pow(env, 16);
```

sin

Fourier analysis tells us tones can be formed out of sine waves.

The `sin()` function has a repetition period of `twopi`:

```
sample p = phasor(&s->osc, 440);
sample tone = sin(twopi * p);
```

Multiplying by an integer gives harmonics:

```
sample tone = sin(1 * twopi * p); // fundamental
sample tone = sin(2 * twopi * p); // octave
sample tone = sin(3 * twopi * p); // octave+fifth
sample tone = sin(4 * twopi * p); // two octaves
```

mix

Tones can be mixed, even varying over time:

```
sample complex_tone = mix
  ( sin(1 * twopi * p)
  , sin(2 * twopi * p)
  , 0.25 // mostly the first tone
  );
sample time_varying_tone = mix
  ( sin(2 * twopi * p)
  , sin(3 * twopi * p)
  , wrap(2 * bar)
  );
```

Keyword: additive synthesis

tanh

The `tanh()` function is a soft clipper: the output is always between -1 and 1, but louder input signals get more distorted.

```
sample distorted = tanh(4 * time_varying_tone);
```

When there is more than one sine wave component, more complicated sounds result.

The `sin()` function can also be used for distortion:

```
sample distorted = sin(4 * time_varying_tone);
```

Keyword: wave-shaping

kick

Envelopes can be applied to pitch as well as volume.

A simple kick drum is a decaying downwards sine wave sweep:

```
sample kick = 1 - wrap(beat);
kick = pow(kick, 8);
kick *= sin(12 * twopi * kick);
```

Try changing the curve power and frequency multipler.

snare

A simple snare is enveloped noise:

```
sample snare = 1 - wrap(2 * bar - 0.5);
snare = pow(snare, 16);
snare *= noise();
```

Subtracting 0.5 from the phase makes it on the off-beat.

Slower beats need a higher power curve to sound similar.

break time

Exit clive: press Ctrl-C in the terminal you launched it from.

Create a git tag to refer to later:

```
git tag -a metronome-jamming \
  -m "workshop 2022-01-15 time-based synthesis"
```

Take a break!

feedback processes

git log

The history of your edits is saved in your git repository:

```
git status
git log --oneline
```

The messages are automatic, so not informative. To see what changed, use `git show` with the hash of the commit, for example:

```
git show e02762d
```

git checkout

You can access branches and tags by name:

```
git checkout metronome-jamming
```

You can list branches and tags:

```
git branch -a
git tag -ln
```

workshop

Now we will continue from a different branch:

```
cd ~/code/clive-workshop
git checkout workshop
```

This is the basis of a short live performance. Here's the one I prepared earlier again, this time you should understand more of the code edits:

https://mathr.co.uk/clive/workshop/2022-01-15/claude.html

Then

```
./launch/local-native-sse.sh
```

biquad highpass

The kick is a bit feeble. Add more bass with a resonant filter. Use a high-pass filter to preserve the attack.

```
BIQUAD kickbq; // in struct


sample kkQ = flatq; // resonance
sample kkHz = 60; // pitch
kick = biquad
  ( highpass(&s->kickbq, kkHz, kkQ) // filter
  , kick // input
  );
```

Try changing the Q in 10–100, and the Hz in 20–200.

vcf

Make the snare more interesting: boost mids.

```
VCF snarevcf[2]; // in struct


sample snQ = flatq; // resonance
sample snHz = 600; // pitch
sn[0] = vcf(&s->snarevcf[0],
  snQ * sn[0], snHz, snQ); // filter
```

Try changing the Q in 5–50, and the Hz in 200–2000. Remember to increase the volume if you don't hear any changes:

```
sample snare_gain = 1;
```

samphold

Give the snare a lo-fi feel: quantize in time.

```
PHASOR crush; // in struct
SAMPHOLD snaresh[2];

sample crush = phasor(&s->crush, 4000);
sn[0] = samphold(&s->snaresh[0], sn[0], crush);
sn[1] = samphold(&s->snaresh[1], sn[1], crush);
```

Try changing the phasor frequency in 1000–10000. In `4000 * pow(1, cos(twopi * t))` try changing the base from 1 to 2.

if

Make the second snare in each bar repeat conditionally:

```
sample snare = 1 - wrap(2 * bar + 0.5);
if (bar > 0.75) {
  snare = wrap(8 * snare);
}
```

An `else` branch is optional, `{ ... }` are only needed for more than one statement:

```
if (bar > 0.75) snare = wrap(8 * snare);
else snare = 1 - snare;
```

?:

The ternary operator can be used in expressions:

```
snare = wrap((t < 0.75 ? 8 : 6) * snare);
kick = wrap((bar < 0.75 ? 1 : 2) * kick);
```

Conditionals give 0 and 1, so this could be written:

```
snare = wrap((8 - 2 * (t < 0.75)) * snare);
kick = wrap((1 + (bar < 0.75)) * kick);
```

delay feedback

Filters operate on short delays (a few samples). Longer delays can be used for echoes.

Scroll to around line 130 and change `feedback` to 1.

Note: usually feedback should be between -1 and 1, but this example has a dynamic range compressor to prevent explosions.

Change `feedin` to 0 to hear the delay recirculating without any input.

explosions

If feedback is too large, delay line can explode. In such an emergency, clear the whole memory buffer:

```
if (s->reloaded) {
  memset(s, 0, sizeof(*s));
}
```

Once you have found and fixed the problem, comment out to avoid reset again on next save:

```
if (s->reloaded) {
  // memset(s, 0, sizeof(*s));
}
```

delay time

Scroll to around line 100 to see:

```
sample ms[2] =
  { (1000 * 60 / s->bpm) *
      (wrap(1 * t) < 1 ? 4. / 4 : 1. / 64)
  , (1000 * 60 / s->bpm) *
      (wrap(2 * t) < 1 ? 4. / 4 : 1. / 48)
  };
```

Change the 4./4 to 3./4 or 2./4.

Change the 3 in the `lop()` below to 0.1 and make more changes to the delay factors. Listen carefully.

Change the 0.1 back to 3 and the delay factors to 2./4.

delay time sequencing

Change `wrap(1 * t) < 1` to `wrap(1 * t) < 0.75`.

Change `wrap(2 * t) < 1` to `wrap(2 * t) < 0.50`.

Go back to line 130 or so and change `feedin` to 0.1 to restore input to the delay line.

delay line filters

The delay line has a band-pass filter to cut out very low frequencies and very high frequencies.

Scroll to line 140 or so, and change the `hip()` 10 to 100 to cut more bass, and change the `lop()` 10000 to 1000 to cut more treble.

delay line stereo

This example has two mono delay lines to make a stereo delay. The delay lines are mixed together with a rotation matrix, around line 120.

Try changing the angle from `twopi / 24` to `twopi * t` to make it vary over time.

Try adding a continually changing delay time too, just before `sample del[2]` on line 110 or so insert:

```
ms[0] += 10 * cos(twopi * bar);
ms[1] += 10 * sin(twopi * bar);
```

preparing for rehearsal

Make edits until it sounds like your ideal start of performance.

Exit clive with Ctrl-C in the terminal you started it from.

neatening up

Make a note of session-branch name, the latest session-2022-etc.

```
git status
```

Create a new branch name `workshop-myname`.

```
git checkout workshop
git checkout -b workshop-myname
git merge --squash session-branch-name
git commit -m "getting ready for rehearsal"
```

Now the history is short and sweet.

practice performance

```
git checkout workshop-myname
./launch/local-native-sse.sh
```

Ctrl-C in the terminal to finish.