clive workshop

Claude Heiland-Allen

2019-11-23

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 callbacks.

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

today

- Install Party
  - break
- Motivational Talk
- Time-based Synthesis
  - break
- Feedback Processes
  - break
- Live-coding Performance

Install Party

website

This presentation is online:

https://mathr.co.uk/clive/workshop/2019-11-23.html

https://mathr.co.uk/clive/workshop/2019-11-23.pdf

Access it now so you can copy/paste.

roadmap

- install dependencies
- configure JACK for realtime
- configure PulseAudio on top of JACK, or
- configure JACK on top of PulseAudio
- get `clive` and `exwhyscope` using `git`
- test

Install dependencies

Linux only. Instructions tested on Debian.

```
# apt install \
    sudo git ca-certificates \
    build-essential pkg-config \
    libjack-jackd2-dev qjackctl \
    pulseaudio-module-jack \
    cpufrequtils ecasound \
    xterm htop geany nano \
    python-pygments vorbis-tools \
    libglew-dev libglfw3-dev
```

You may be asked about JACK realtime, accept / say yes.

realtime

Provided by JACK packaging on Debian:

```
$ cat /etc/security/limits.d/audio.conf
@audio   -  rtprio     95
@audio   -  memlock    unlimited
```

To check that you are in the audio group:

```
$ groups
```

Otherwise (replace `myname` with your username):

```
$ sudo adduser myname audio
```

Then log out and log back in.

PulseAudio JACK bridge

Edit PulseAudio configuration:

```
$ sudo nano /etc/pulse/default.pa
```

Append to the bottom of the file:

```
load-module module-jack-source
load-module module-jack-sink
set-default-source jack_in
set-default-sink jack_out
```

PulseAudio vs JACK

PulseAudio is a desktop-oriented audio library. It allows multiple programs to use one audio hardware device.

`pavucontrol` is a graphical user interface for configuring PulseAudio.

JACK is a pro-audio oriented library. It allows audio to be routed between different applications.

`qjackctl` is a graphical user interface for configuring JACK.

PulseAudio on top of JACK

This is the best option for pro-audio hardware interfaces.

`qjackctl` → setup → options → execute script on startup (all on one line)

```
pulseaudio --kill ;
killall -KILL jackdbus ;
killall -KILL jackd
```

`qjackctl` → setup → options → execute script after startup

```
pulseaudio --start
```

JACK D-Bus

Some distributions package PulseAudio and JACK with D-Bus. Disable it:

/etc/pulse/default.pa *contains*

```
.ifexists module-jackdbus-detect.so
.nofail
load-module module-jackdbus-detect channels=2
.fail
.endif
```

You can comment those lines out and that should prevent auto-loading to jackdbus by Pulse Audio.

Credit: https://askubuntu.com/a/1052529

PulseAudio autospawn

Sometimes PulseAudio restarts too quickly and reclaims the soundcard hardware before JACK can start. Disable the autospawn:

> In `/etc/pulse/client.conf`, you can uncomment the line `autospawn=yes` and replace the `yes` with a "`no`".

Credit: https://askubuntu.com/a/10307

jackd

`qjackctl` → setup → realtime → yes

`qjackctl` → setup → interface → your interface

`qjackctl` → setup → sample rate → 48000
(or make a note to change `SR` in the code later)

`qjackctl` → setup → frames / period → 2048

`qjackctl` → setup → periods / buffer → 3

`qjackctl` → start

xruns

XRUNs occur when there is a problem and realtime deadlines are not met.

Symptoms are audible clicks and dropouts.

Diagnose by numbers other than green `0 (0)` in the middle left of the `qjackctl` status panel.

If you get XRUNs with PulseAudio on top of JACK, try JACK on top of PulseAudio in the next slides.

JACK on top of PulseAudio

You only need to do this if you have XRUNs. Find the name of your PulseAudio soundcard sink:

```
pactl list sinks | grep Name
```

If your soundcard is not visible, enable it in `pavucontrol` → configuration. Append to `/etc/pulse/default.pa` (all on one line without \):

```
load-module module-loopback source=jack_in \
  sink=alsa_output.pci-0000_00_1b.0.analog-stereo
```

This is an example, copy/paste yours to `sink=XXXX`

JACK on top of PulseAudio (continued)

`qjackctl` → stop

`qjackctl` → setup → options → disable all scripts

`qjackctl` setup → settings → driver, select `dummy`.

Restart PulseAudio:

```
pulseaudio --kill
pulseaudio --start
```

`qjackctl` → start

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

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
make -C server
make -C client
```

Get exwhyscope

`exwhyscope` is an XY oscilloscope using JACK.

```
mkdir -p ~/code
cd ~/code
git clone \
  https://code.mathr.co.uk/exwhyscope.git
cd exwhyscope
make
./exwhyscope &
```

If the window is too big/small for comfort you can change the two 512 in line 85 of `exwhyscope.c` and recompile.

Test clive

To test:

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

You should hear regular beeping, and see coloured lines in `exwhyscope`.

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

Break time

Take a short break.

Motivational talk

Digital audio

- Real world is continuous.
- Digital world is discrete.
- Sample points equally spaced in time.
- Shannon-Nyquist sampling theorem says when this approximation is ok.

Audio properties

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

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

- Configure Linux for JACK audio
  - hopefully working for everybody by now
- `git` version control system
  - clone, status, commit, branch, tag
- C programming for audio
- Digital audio processing

Example track

https://mathr.co.uk/clive/workshop/2019-11-23/claude.html

time-based synthesis

metronome

Make sure `exwhyscope` is started before clive if it is not already running:

```
cd ~/code/exwhyscope
./exwhyscope &
```

Then launch clive:

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

windows

- `qjackctl` with transport timer and status.
- 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.
- `exwhyscope` oscilloscope for monitoring signals.
- `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. Open a new terminal and run

  `alsamixer`

  Press F6 and use cursor keys to select your device, then reduce the main volume. External cards may have different volume controls. You can use `--card` option if you have no F-keys.
- In `Geany`, set the volume to 10.0 (!) and save.
- In `alsamixer`, increase the main volume as much as is comfortable. ESC exits `alsamixer`.
- 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.

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

`out[2 * n + 0]` is scope X (horizontal)

`out[2 * n + 1]` is scope Y (vertical)

Output signals should be between -1 and 1.

See `qjackctl` → 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); // off-beat
snare = pow(snare, 16);
snare *= noise();
```

Slower beats need a higher power curve to sound similar.

break time

Exit clive: press Ctrl-C in the terminal you launched it from. Keep `exwhyscope` running. Create a git tag to refer to later:

```
git tag -a metronome-jamming \
  -m "workshop 2019-11-23 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/2019-11-23/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.

break time (again)

Make a note of session-branch name, the latest session-2019-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.

Take a break!

live performance

preparation

Plan the progression of the performance.

- beginning
- middle
- end

Aim for 5–10mins.

Avoid huge typing - you can have pre-prepared code to uncomment.

Edits don’t need to be local.

rehearsal

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

Practice intended code edits. Ctrl-C when done. To keep the final state:

```
git status
```

Make a note of branch name, session-2019 etc

```
git checkout workshop-myname
git merge --squash session-branch-name
git commit -m "after rehearsal"
```

play live

Make sure `qjackctl` transport is stopped and rewound.

Make sure `exwhyscope` is running if you want it.

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

Play live for about 5-10mins.

Ctrl-C when done, stop and rewind JACK transport.

render to HTML+Ogg

```
git status
```

Make a note of session branch name with the date. Replace `myname` with your (nick)name in the below:

```
git tag -a myname -m "workshop 2019-11-23"
./extra/session2html.sh myname session-branch
oggenc -b 192 session-branch.wav -o myname.ogg
firefox myname.html
```

If code edits do not appear in the audio player, adjust the time stamp hour (here that would be the 7):

```
./extra/session2html.sh \
  myname session-2019-11-23-171819
```

final presentation

- Collect all the HTML+Ogg recordings via USB
- Play them in random sequence
- (if participants are willing) publish online?
- anonymous feedback questionaire