Fearless low latency audio synthesis

Raph Levien

SF Rust Meetup · 13 Nov 2018

synthesize.rs/nov-2018-talk

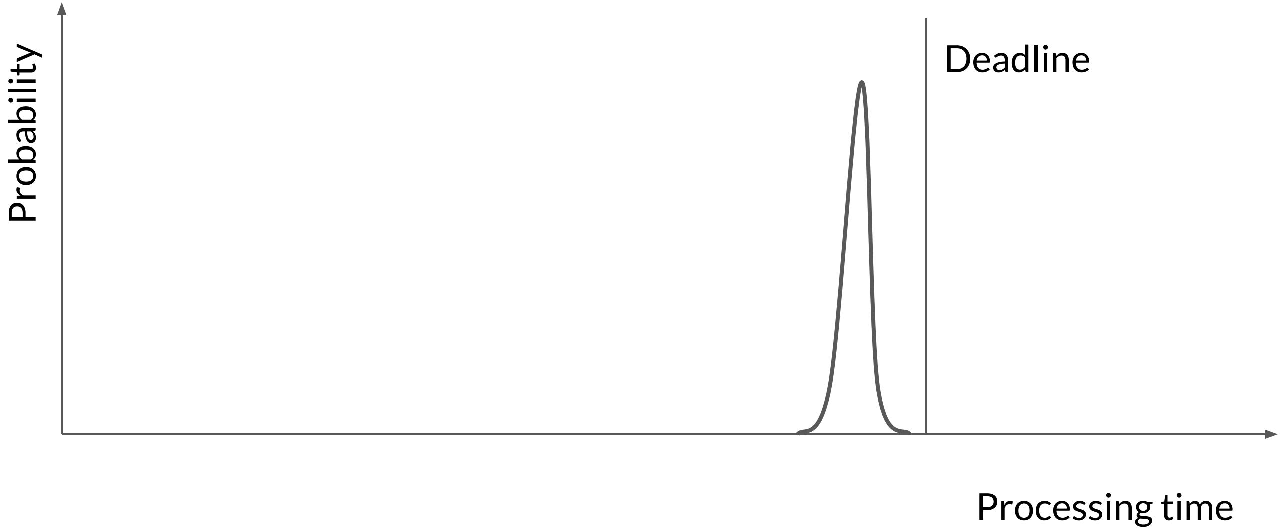
Early ’80s - PC/AT drum machine

- Samples stored in RAM
- 4 voice polyphony
- Audio loop: iterate through 4 voices, fetch sample, add
- Output audio through parallel printer port
  - out 0x378, al
  - 8-bit DAC attached to Centronics port
- Sample rate set by constant time instruction execution
  - No caching, branch prediction, speculation
- Voice allocation done in keyboard interrupt

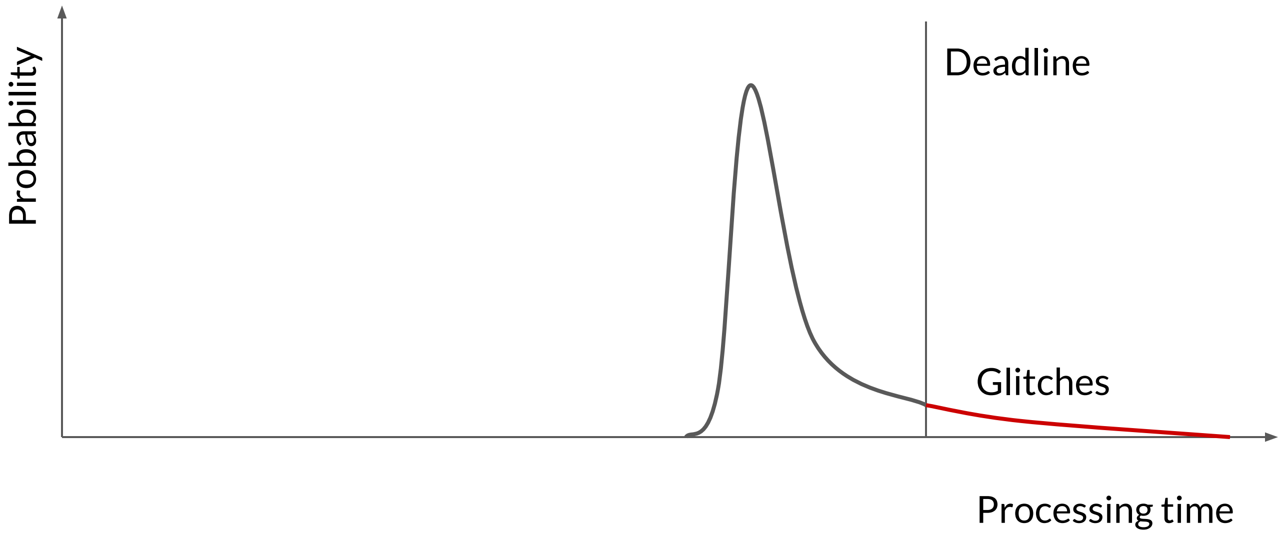
PC/AT drum machine performance analysis

- Throughput
  - ~1MIPS
  - Barely enough to get 4 voices with no processing at 20-30kHz sampling rate
  - Fixed point, but 80287 of the era was ~50 kFLOPS
- Latency
  - ~50μs

PC/AT drum machine performance



Audio performance problems today



Where does the tail come from?

- Low-probability blocking events
    - Filesystem access
    - Blocking on a mutex held by a non-realtime thread (priority inversion)
    - **Allocation**
    - Calls into any code that does this
- A copout: increase deadline

Time waits for nothing

- Make sure OS schedules audio thread as real-time
- Most systems can do this:
  - Windows used to require ASIO drivers, now has WASAPI
  - macOS and iOS have had CoreAudio for years
  - Android had OpenSL ES, now has AAudio
- Problem: make cpal do all processing in real-time thread (cpal#156)
- Enforce rigorous nonblocking in audio processing thread
  - No mutex, no allocations

Further reading: Ross Bencina blog post

Time waits for nothing → wait-free algorithms

- Can’t use Mutex to communicate between processing and engine
- **Must** use wait-free (lock-free) data structures
    - Most popular is a queue
    - Many flavors:
        - Fixed size (ring buffer)
        - Linked list
        - Single or multiple producer
        - Single or multiple consumer
- Wait-free algorithms are tricky

Further reading: Dmitry Vyukov page on queues

MSFA (engine in Dexed)

MSFA: Fortran style

- music-synthesizer-for-android engine
- Accurate emulation of Yamaha DX7
- Useful test case to push Android audio performance (Google IO 2013)
- Engine inside popular Dexed plugin
- Preallocate all synthesis state in fixed size arrays
  - DX7 synthesis state is 100s of bytes
- Ring buffer for MIDI → engine channel
- No return channel

Good performance but limited flexibility

Dynamic behavior

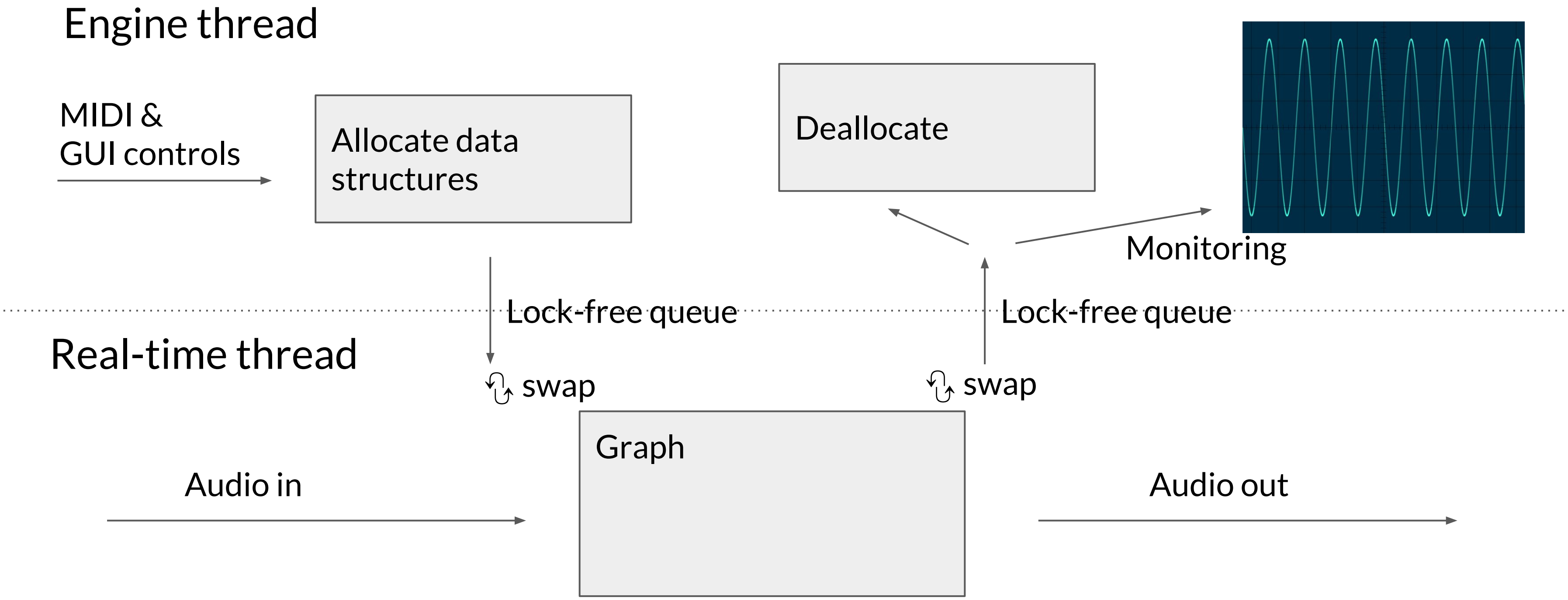
- Instantiate new modules
- Patch into graph
- Change graph wiring
- Load content from files
- Create complex data structures for use in synthesis

What could go wrong?

Undefined behavior in the wild

Video credit: Andreas Belschner

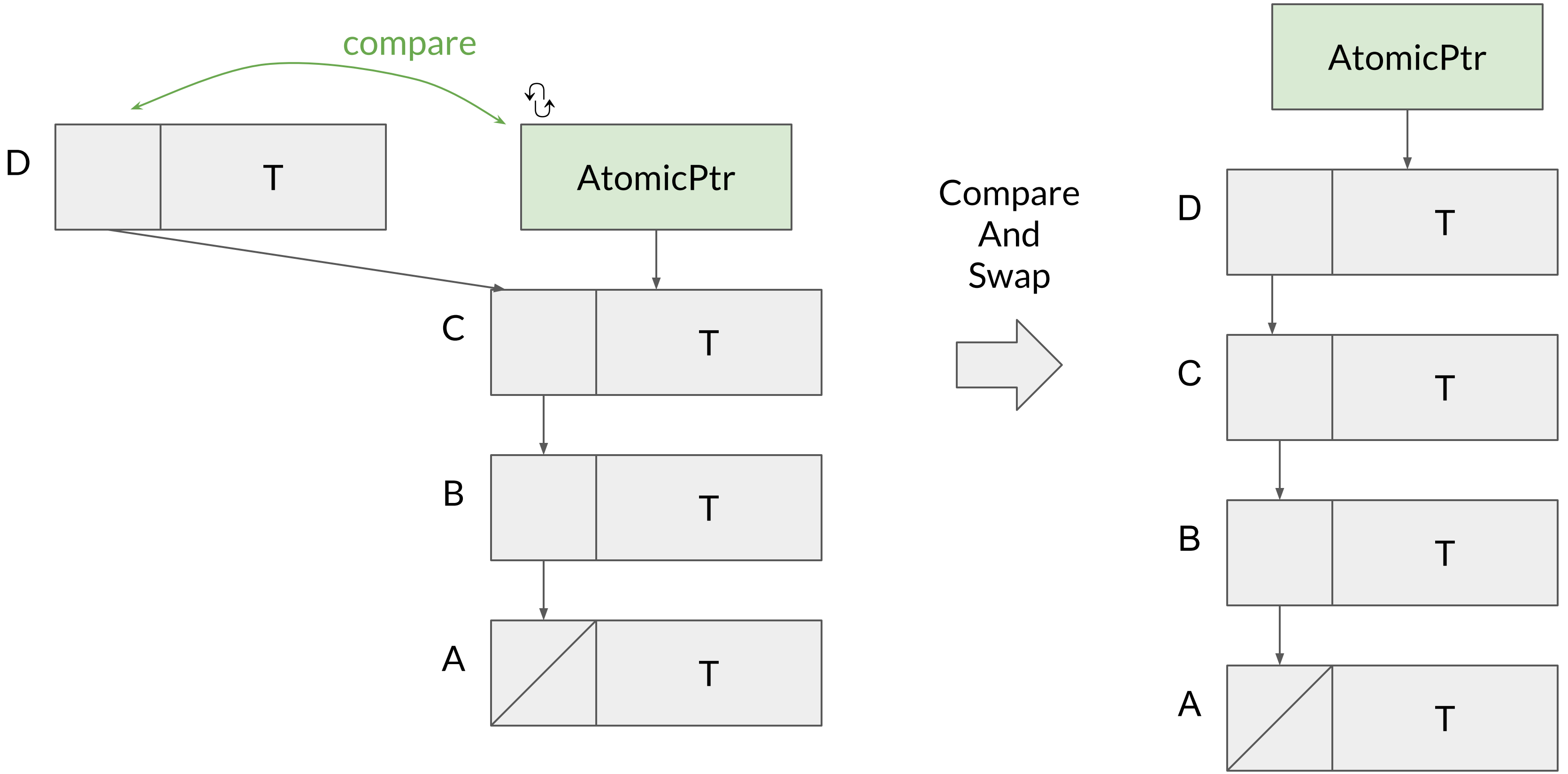
Architecture



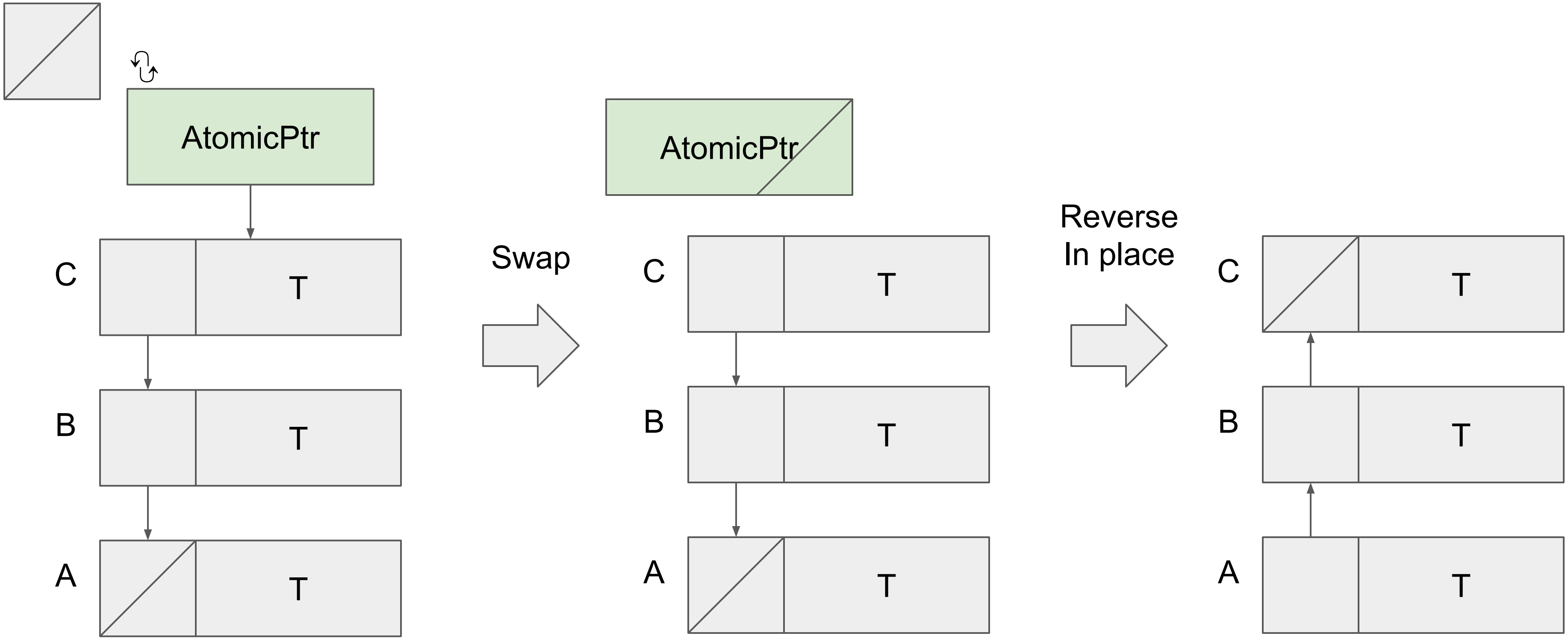
Treiber stack

- One of the simpler lock-free data structures
- Based on linked list
- Unbounded
- Two operations:
  - Push one
  - Take all
- Great for MPSC queues

Treiber stack: push one



Treiber stack: take all



Audio processing trait

```
pub trait Module: ToAny + Send {
    fn process(&mut self, control_in: &[f32], control_out: &mut [f32],
        buf_in: &[&Buffer], buf_out: &mut [Buffer]);

    fn migrate(&mut self, old: &mut Module);

    fn set_param(&mut self, param_ix: usize, val: f32, timestamp: u64);

    ...
}
```

Towards Fearless SIMD

- SIMD is a perfect grain of computation for audio
- Many algorithms get 4x - 8x speedup:
  - Waveform generation (unit generator)
  - FM synthesis
  - FIR & IIR filters
  - Waveshaping
  - Spectrum analysis
- SIMD now in stable Rust! 🎉
- But...

Why is SIMD unsafe?

- Scalar computations are standardized
- SIMD varies widely from chip to chip
  - 128 bits (older x64, ARM)
  - 256 bits (current x64)
  - 512 bits (next-gen x64)
  - Newer masked instructions
- Using an unsupported SIMD instruction is Undefined Behavior
- Binaries need to ship multiple versions & choose at runtime

SIMD with intrinsics

```
unsafe fn quadwave_avx(freq: f32, obuf: &mut [f32]) {
    let mut i = 0;
    let mut phase = _mm256_mul_ps(_mm256_setr_ps(0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0,
7.0),
        _mm256_set1_ps(freq));
    let phaseinc = _mm256_set1_ps(8.0 * freq);
    for i in (0..obuf.len()).step_by(8) {
        let y = _mm256_sub_ps(phase, _mm256_round_ps(phase, 8));
        let y = _mm256_mul_ps(y, _mm256_sub_ps(_mm256_set1_ps(0.5),
            _mm256_abs(y)));
        let y = _mm256_mul_ps(y, _mm256_set1_ps(16.0));
        _mm256_storeu_ps(obuf.as_mut_ptr().add(i), y);
        phase = _mm256_add_ps(phase, phaseinc);
    }
}
```

Same code with fearless_simd

```
struct QuadWaveFn;
impl SimdFnF32 for QuadWaveFn {
    #[inline]
    fn call<S: SimdF32>(&mut self, x: S) -> S {
        let phase = (x - x.round());
        phase * (phase.abs() - 0.5) * -16.0
    }
}

fn gen_quadwave(freq: f32, obuf: &mut [f32]) {
    count(0.0, freq).map(QuadWaveFn).collect(obuf);
}
```

SIMD performance

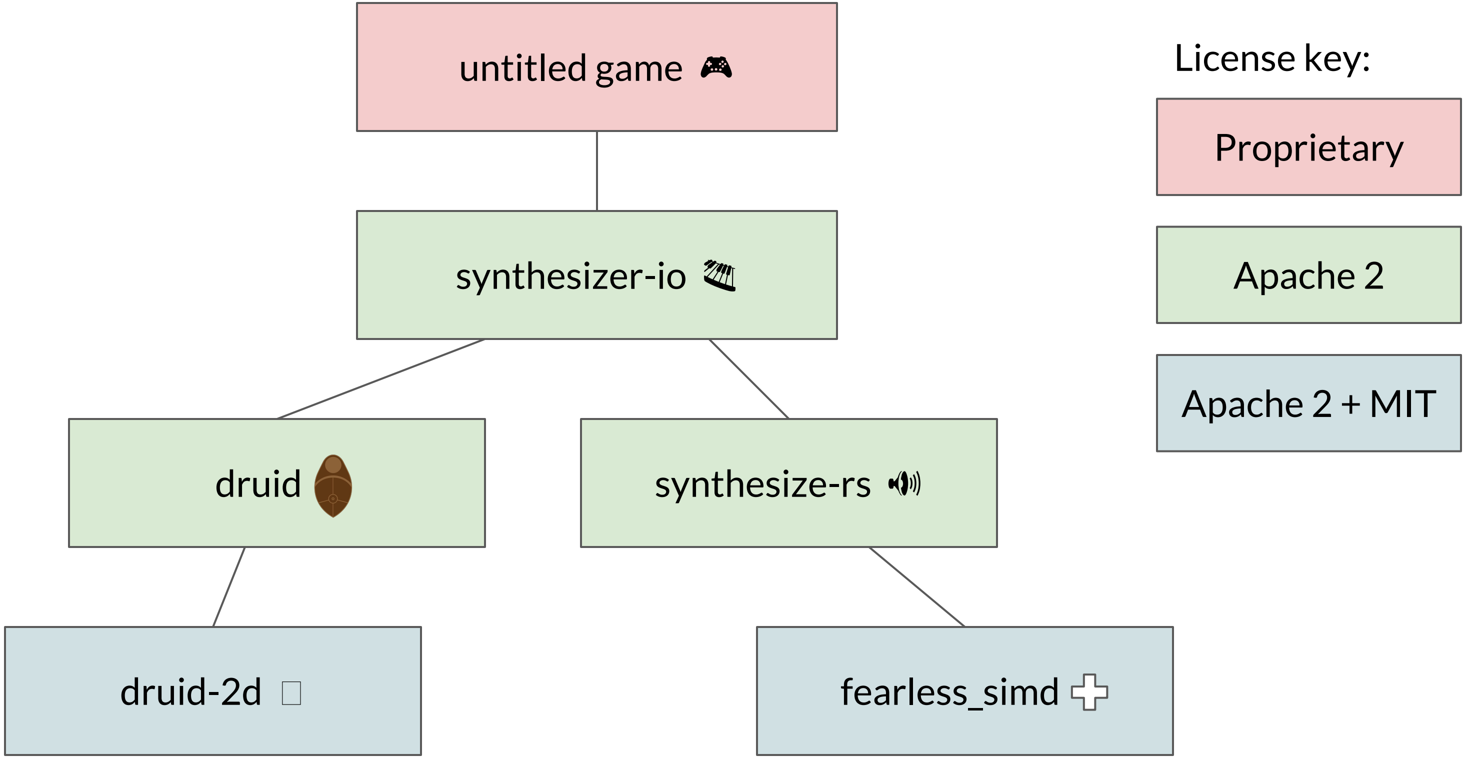
Timings in ns/audio sample (i7 7700HQ @2.8GHz):

	sinewave	tanh
AVX2	0.47	0.45
SSE4.2	0.77	0.78
scalar (std)	7.9	5.8

Thoughts on Rust for creative work

- Creativity works in the presence of constraints
  - And Rust enforces those constraints!
  - C++ lets you shoot yourself in the foot
    - Always the fear of creating crashes or worse - not unfounded
- Rust is about composing pieces
  - Let CS PhD's work out lock-free algorithms, etc.
  - Then compose them in a "fearless" way
  - Focus on the audio algorithm at hand

What I’m building



Follow me on social media

- Code: github.com/raphlinus/synthesizer-io
- Blog: raphlinus.github.io
- Zulip: xi.zulipchat.com
  - #synthesizer stream
- Patreon: raphlinus
- Twitter: @raphlinus