Functional, Declarative Audio Applications

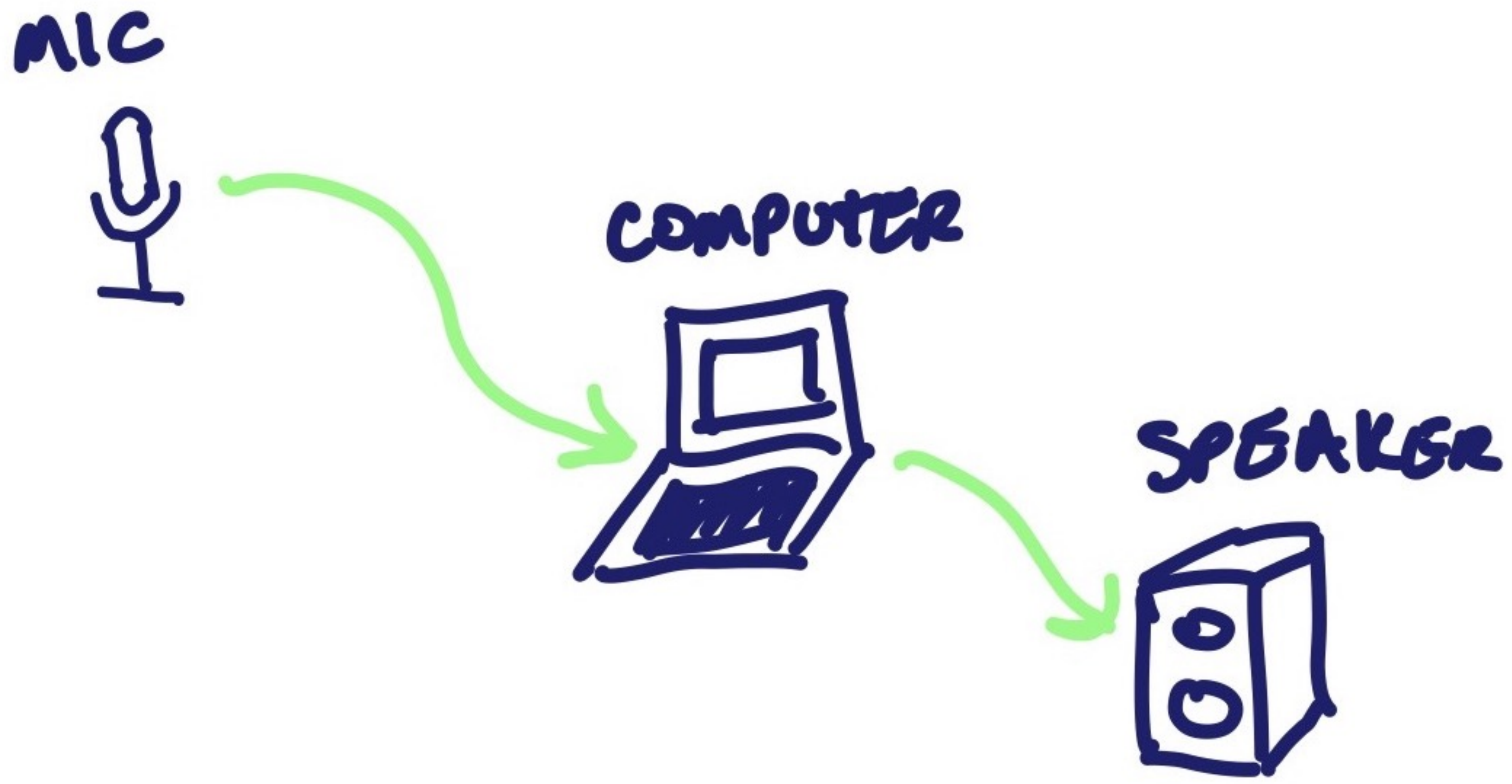
Nick Thompson

- Independent audio software developer, contractor, consultant
- Elementary Audio
- Creative Intent
- React-JUCE

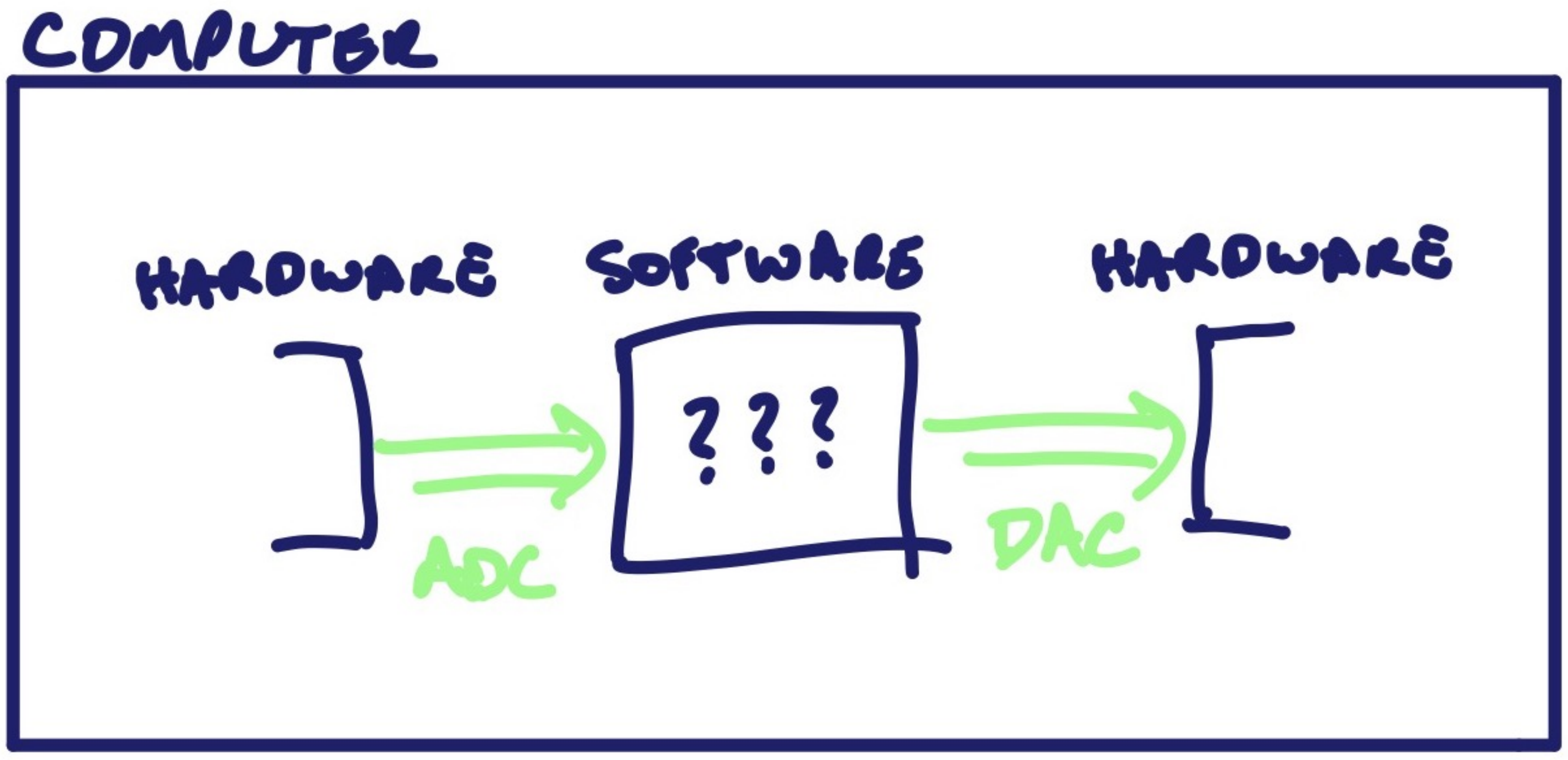
Agenda

- Setting: What is Audio Software?
- The Golden Rules
- Traditional Audio Software Architecture
- A Declarative Approach
- Elementary Audio Drum Synthesis

Audio Software



Audio Software



Golden Rules

Realtime audio applications must deliver a block of audio received from the driver *back* to that driver without any discontinuity in the resulting signal

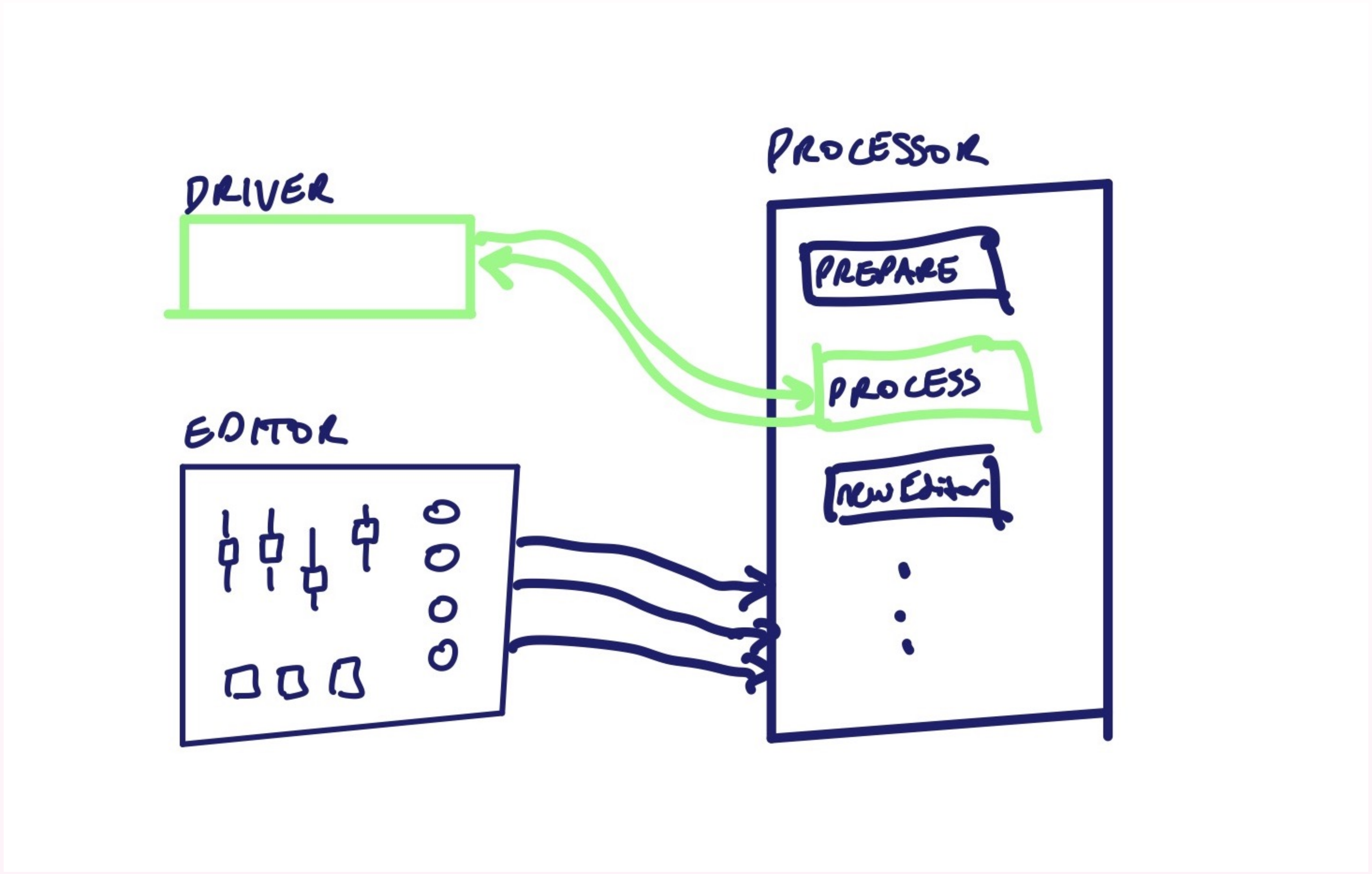
Golden Rules

- Reliability
- Determinism
- Performance
    - Typically, delivering an immense amount of numerical computation with a few milliseconds

Traditional Architecture

- Typically C/C++:
    - Direct access to memory
    - Direct access to underlying platform or hardware
    - Direct access to threading primitives
- Fair enough!

Traditional Architecture



Traditional Filter 101

```
class Processor {
public:
    // Called on the main thread
    void prepare (double sampleRate, int blockSize);

    // Called on the realtime thread
    void processBlock (AudioBlock& block);
};
```

Traditional Filter 101

```
class Processor {
public:
    // Called on the main thread
    void prepare (double sampleRate, int blockSize);

    // Called on the realtime thread
    void processBlock (AudioBlock& block);

private:
    BiquadFilter bq;
};
```

Traditional Filter 101

```
class Processor {
public:
    Processor()
      : bq(1, 0, 0, 0, 0) {}

    // Called on the main thread
    void prepare (double sampleRate, int blockSize);

    // Called on the realtime thread
    void processBlock (AudioBlock& block);

private:
    BiquadFilter bq;
};
```

Traditional Filter 101

```
class Processor {
public:
    Processor()
      : bq(1, 0, 0, 0, 0) {}

    // Called on the main thread
    void prepare (double sampleRate, int blockSize) {
        bq.prepare(sampleRate, blockSize);
    }

    // Called on the realtime thread
    void processBlock (AudioBlock& block);

private:
    BiquadFilter bq;
};
```

Traditional Filter 101

```
class Processor {
public:
    Processor()
      : bq(1, 0, 0, 0, 0) {}

    // Called on the main thread
    void prepare (double sampleRate, int blockSize) {
        bq.prepare(sampleRate, blockSize);
    }

    // Called on the realtime thread
    void processBlock (AudioBlock& block) {
        bq.processBlock(block);
    }

private:
    BiquadFilter bq;
};
```

Traditional Filter 101

What about Composition?

Traditional Filter 101

```
FloatVectorOperations::negate(xfadeGains, xfadeGains, len);
FloatVectorOperations::add(xfadeGains, 1.0f, len);

// Now we can handle the distorted low band
dsp::ProcessContextReplacing<SampleType> lowBandContext (lowBlock);
m_lowBandProcessor.process(lowBandContext);
addWithArrayMultiply(wetBlock, lowBlock, xfadeGains);

// And we can use the same gain array for the raw high band...
addWithArrayMultiply(wetBlock, highBlock, xfadeGains);

// Now we have to flip the xfade gain array for the distorted high band.
FloatVectorOperations::negate(xfadeGains, xfadeGains, len);
FloatVectorOperations::add(xfadeGains, 1.0f, len);

// And handle the high band...
dsp::ProcessContextReplacing<SampleType> highBandContext (highBlock);
m_highBandProcessor.process(highBandContext);
addWithArrayMultiply(wetBlock, highBlock, xfadeGains);
```

Traditional Filter 101

What about state changes?

Traditional Filter 101

```
class Processor {
public:
    ...

    // Called on the main thread
    void onUserInput (double newFrequency) {
        auto const cs = computeCoeffs(newFrequency);
        bq.b0 = cs[0];
        bq.b1 = cs[1];
        bq.b2 = cs[2];
        ...
    }

    ...
};
```

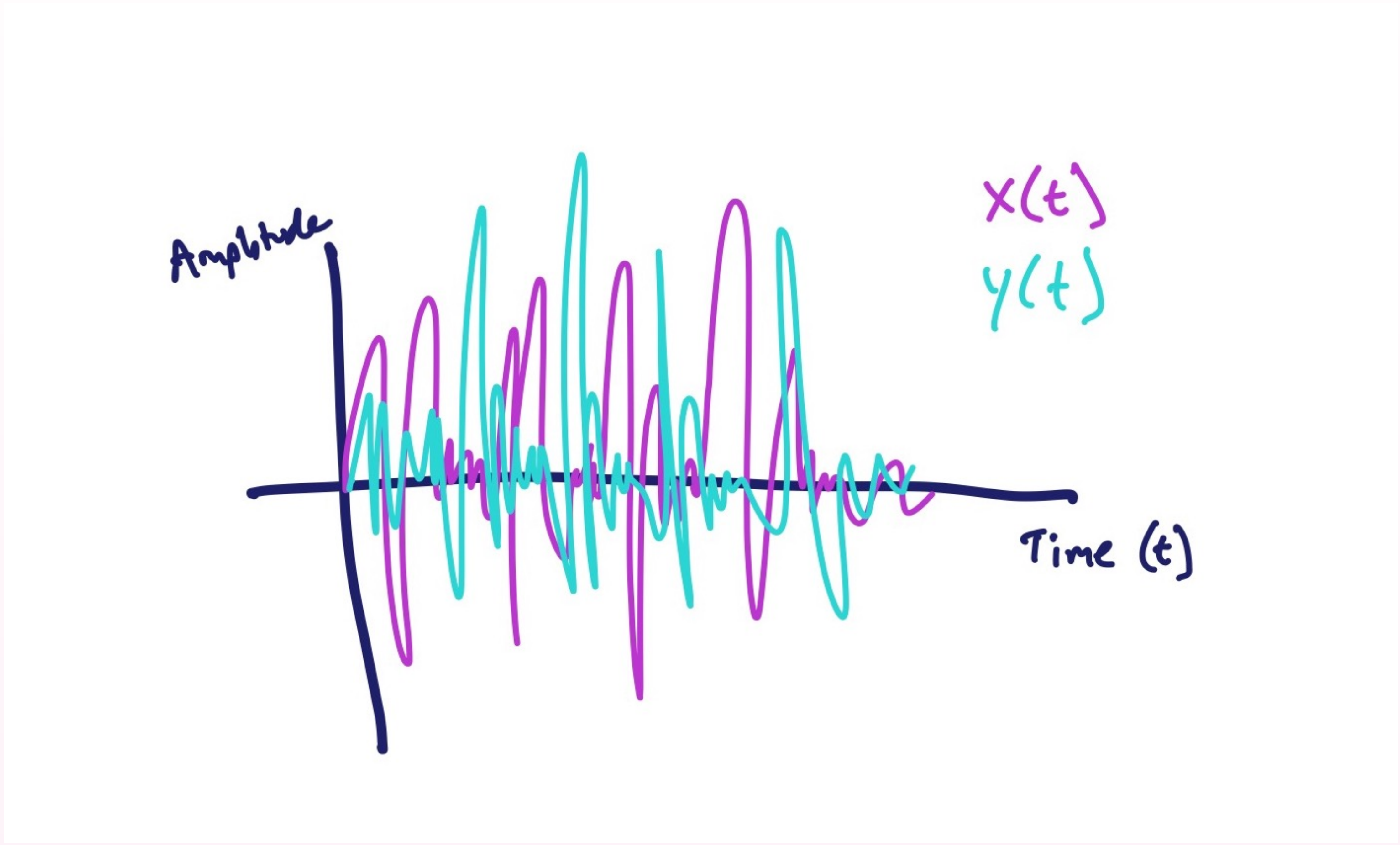
Traditional Filter 101

```
class Processor {
public:
    ...

    // Called on the main thread
    void onUserInput (double newFrequency) {
        auto const cs = computeCoeffs(newFrequency);
        // Assume type std::atomic<Coefficients>
        bq.coeffs.store(BiquadFilter::Coefficients {
            cs[0],
            cs[1],
            cs[2],
            ...
        });
    }

    ...
};
```

Traditional Filter 101

Takeaway: it's hard!

A Declarative Approach



A Declarative Approach

```
y(x, t) = f(x(t))
or maybe
y(x, t) = f(x(t), t)
```

A Declarative Approach

```
y(x, t) = filter(x(t))
  where
    filter(x(t)) = lowpass(800Hz, 1.414, x(t))
    lowpass(hz, q, x(t)) =
      b0, b1, b2, a1, a2 = computeCoeffs(hz, q)
      biquad(b0, b1, b2, a1, a2, x(t))
```

A Declarative Approach

```
y(x, t) = filter(x(t))
  where
    filter(x(t)) = sum(
      lowpass(800Hz, 1.414, x(t)),
      highpass(2000Hz, 1.414, x(t)),
    )
```

Elementary Audio

A JavaScript framework for functional, declarative expression of realtime audio signal processes.

+

A native audio engine to deliver a high performance realization of the given audio process.

Elementary Audio

```
import {
  ElementaryWebAudioRenderer as core,
  el
} from '@nick-thompson/elementary';

core.on('load', function(e) {
  let x_of_t = el.in({channel: 0});
  let y_of_t = el.lowpass(800, 1.414, x_of_t);
  core.render(y_of_t);
});
```

Elementary Audio

```
import {
  ElementaryWebAudioRenderer as core,
  el
} from '@nick-thompson/elementary';

function myRender(cutoff) {
  let x_of_t = el.in({channel: 0});
  let y_of_t = el.lowpass(cutoff, 1.414, x_of_t);
  core.render(y_of_t);
});

core.on('load', (e) => myRender(800));
core.on('change', (newCutoff) => myRender(newCutoff));
```

Drum Synthesis

Demo!

Thank You

- Me
  - https://www.nickwritesablog.com/
  - https://github.com/nick-thompson/
- Elementary Audio
  - https://www.elementary.audio/

Elementary