C++ Real-Time Audio Programming with Bela

Dr Andrew McPherson

Centre for Digital Music
School of Electronic Engineering and Computer Science
Queen Mary University of London

Founder and Director, Bela

C++ Real-Time Audio Programming with Bela

Modular synthesis

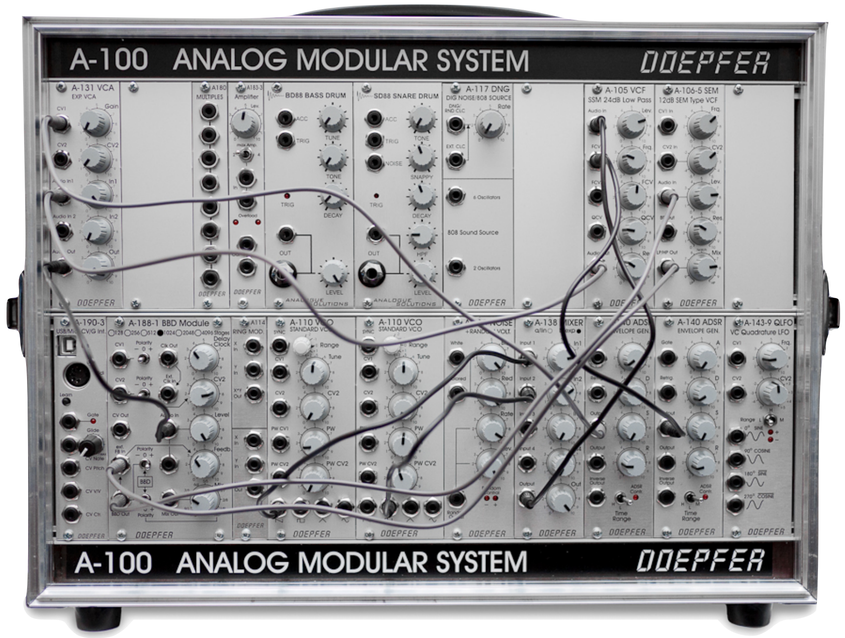
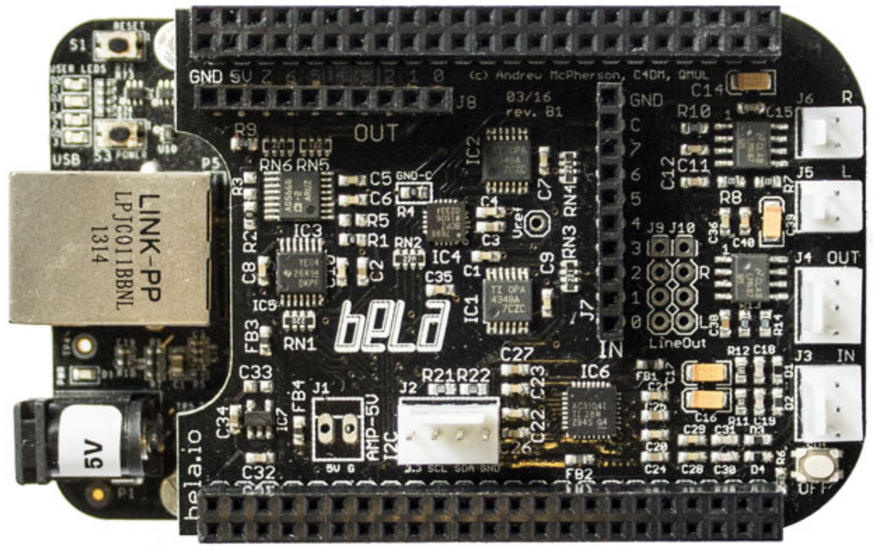


Image credit: Nina Richards, wikimedia (CC-BY 3.0)

C++

Embedded hardware



Course topics

Programming topics

Working in real time
Buffers and arrays
Parameter control
Classes and objects
Analog and digital I/O
Filtering
Circular buffers
Timing in real time
State machines
MIDI
Block-based processing
Threads
Fixed point arithmetic
ARM assembly language

Today

Music/audio topics

Oscillators
Samples
Wavetables
Filters
Control voltages
Gates and triggers
Delays and delay-based effects
Metronomes and clocks
Envelopes
ADSR
MIDI
Additive synthesis
Phase vocoders
Impulse reverb

Lecture 3: Wavetables

What you'll learn today:

Working with buffers (arrays)
Interpolation
The Bela oscilloscope

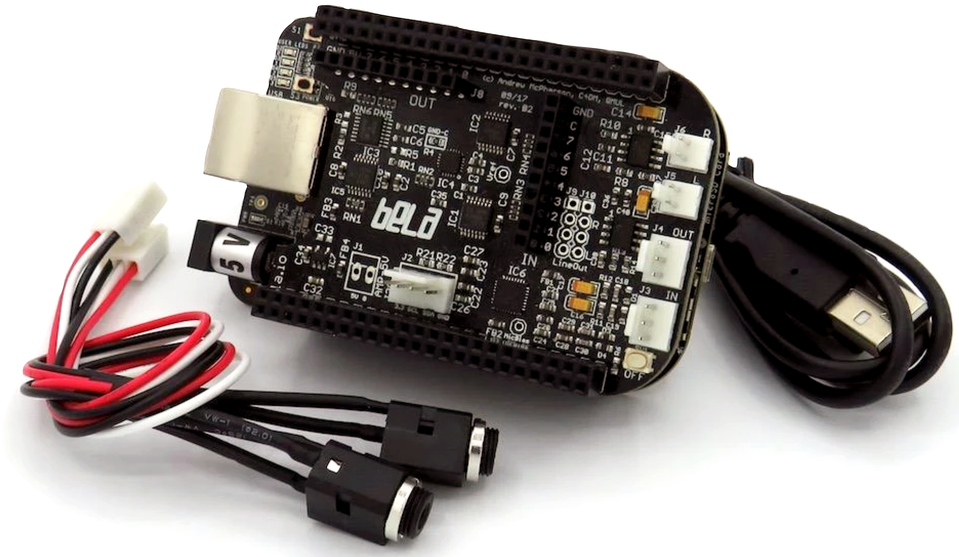
What you'll make today:

Wavetable oscillator

Companion materials:

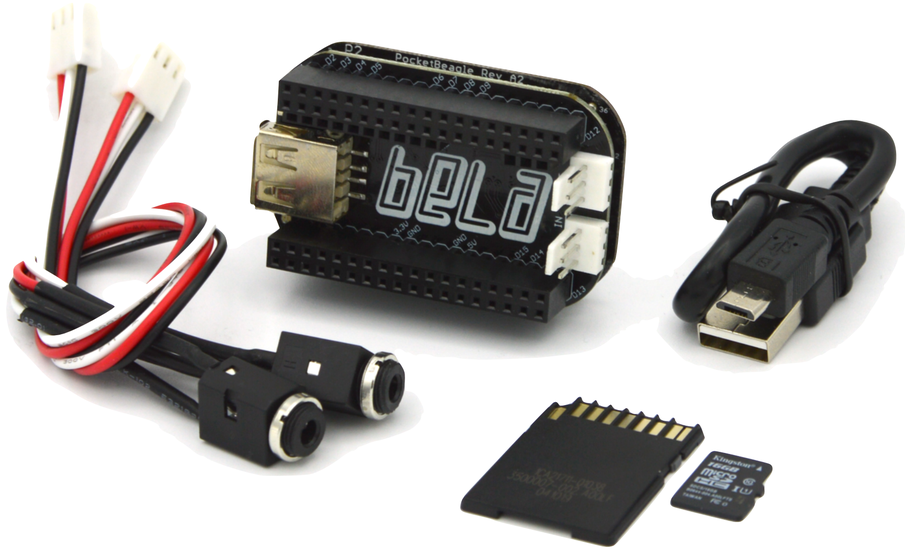
github.com/BelaPlatform/bela-online-course

What you'll need



Bela Starter Kit

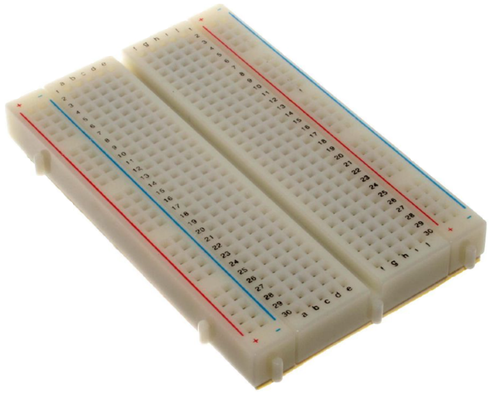
or

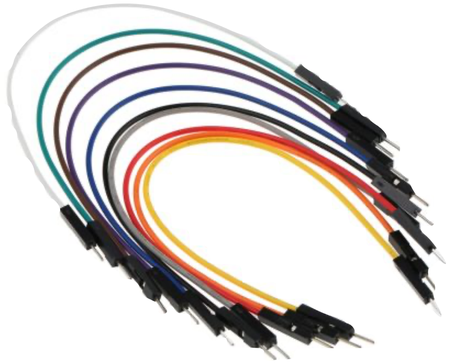


Bela Mini Starter Kit

[shop.bela.io]

Recommended
for some lectures:





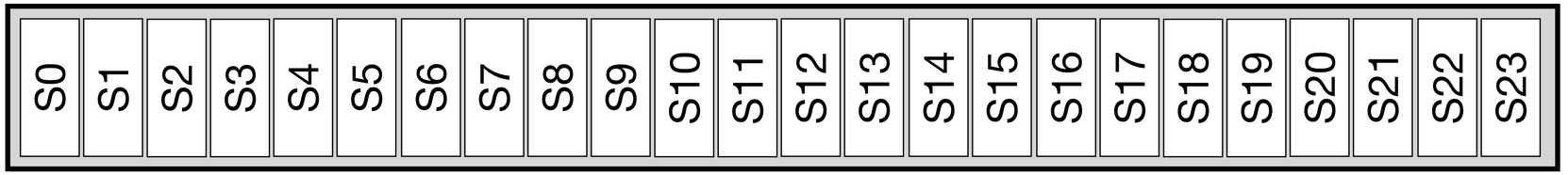
The Bela C / C++ API

- Every Bela program has three main functions:
- `bool setup(BelaContext *context, void *userData)`
  - Runs once at the beginning, before audio starts
  - Data structure `context` holds info on channels, sample rates, block sizes
  - Return `true` if initialisation was successful (`false` stops the program)
- `void render(BelaContext *context, void *userData)`
  - The audio callback function
  - Called automatically by the Bela system for each new block
  - Where most of your code goes: process the samples and return as quickly as possible!
- `void cleanup(BelaContext *context, void *userData)`
  - Runs once at the end, when the program stops
  - Use to release any resources you allocated in `setup()`

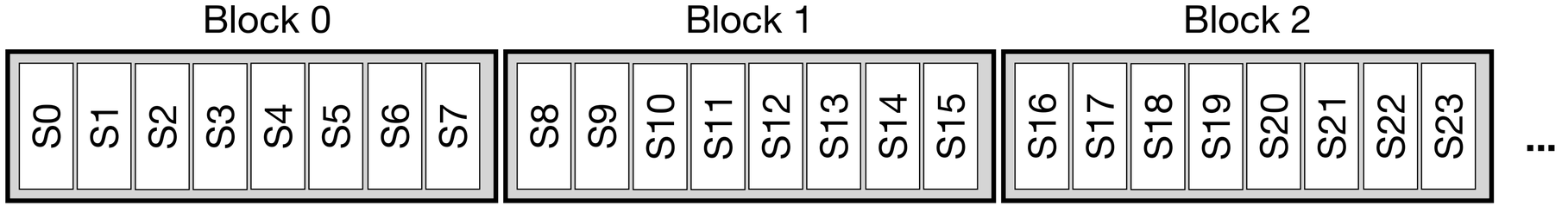
Playing audio samples

- Suppose we want to play a clip of recorded audio
  - For example, something that we load from a WAV file
  - Let's not worry yet about how we load the sound from storage
- The sound, once loaded, will be stored in a buffer (i.e. array)

```
float buffer[24];
```



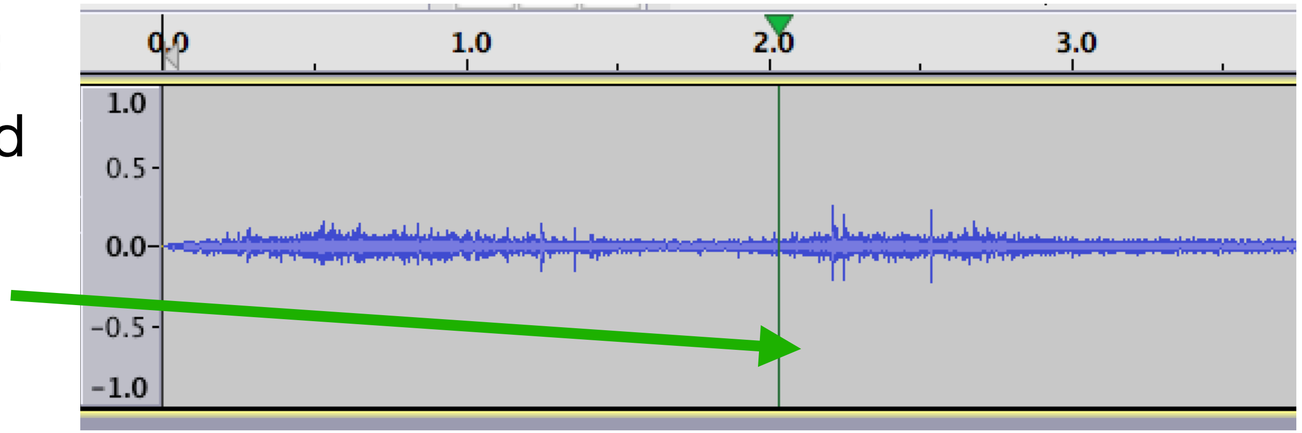
  - The buffer is almost certainly longer than the block size of our real-time system
- We need to copy these samples block-by-block into our audio output



- Besides the buffer itself, what do we need to keep track of to play this sound in real time?

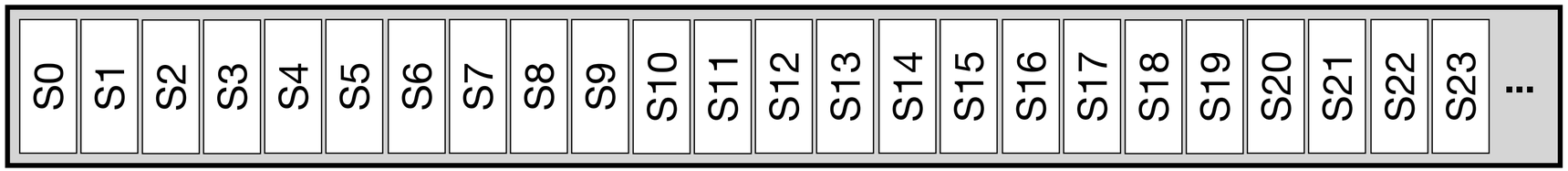
Keeping track of position

- Audio editors show the play head:
  - The location in the buffer being played
  - As the sound plays, the play head moves forward at constant speed



- In DSP programming, the play head is a special case of a read pointer
  - A reference indicating which index in the buffer to play next
  - Just like the phase of an oscillator, we want to remember this from one block to the next
    - So it should be a global variable!

the buffer doesn't change as we play it

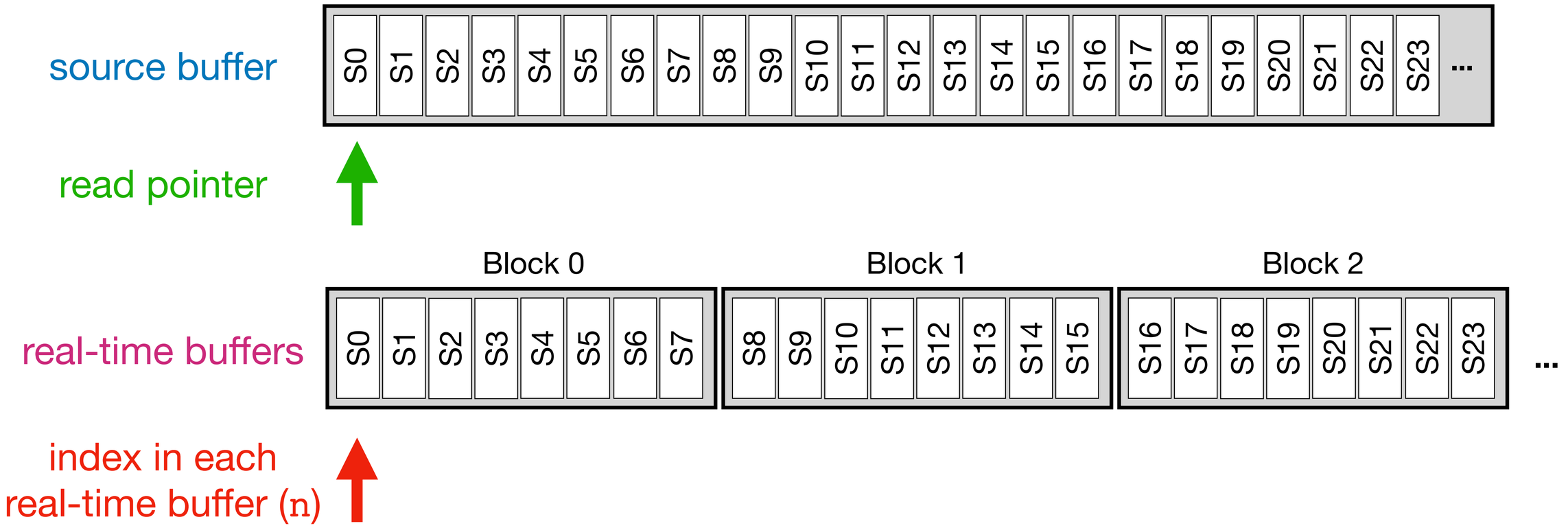


but the read pointer moves

Fun with indexing

- One of the hardest parts of working with buffers can be keeping track of what each index means
- In this case, we've got two different kinds of buffers to think about:
  1. The recorded sound (let's call it the source buffer)
     - Only one buffer whose contents don't change
     - Length: number of samples in the source sound (possibly long)
  2. The buffer for each real-time audio block
     - A new buffer each time `render()` is called, accessed via `audioWrite()`
     - Length: block size of the real-time system (e.g. 16)
- Therefore, we need to keep track of two indexes:
  1. Where are we playing in the source buffer? (read pointer or play head)
  2. Where are we writing in the output buffer? (starts over from 0 each block)

Fun with indexing



`render()` [1st time]
`n` goes from 0 to 7
`gReadPointer` goes from 0 to 7

`render()` [2nd time]
`n` goes from 0 to 7
`gReadPointer` goes from 8 to 15

`render()` [3rd time]
`n` goes from 0 to 7
`gReadPointer` goes from 16 to 23

Playing audio samples

Global variables keep track of source buffer, read pointer

```
std::vector<float> gSampleBuffer;
int gReadPointer = 0;
```

```
// Buffer that holds the sound file
// Position of the last frame we played

void render(BelaContext *context, void *userData)
{
    for(unsigned int n = 0; n < context->audioFrames; n++) {
        // Load a sample from the buffer which was loaded from the file
        float out = gSampleBuffer[gReadPointer];

        // Increment read pointer and reset to 0 when end of file is reached
        gReadPointer++;
        if(gReadPointer >= gSampleBuffer.size())
            gReadPointer = 0;

        for(unsigned int channel = 0; channel < context->audioOutChannels; channel++) {
            // Write the sample to every audio output channel
            audioWrite(context, n, channel, out);
        }
    }
}
```

`gReadPointer` keeps track of where we are in the source buffer

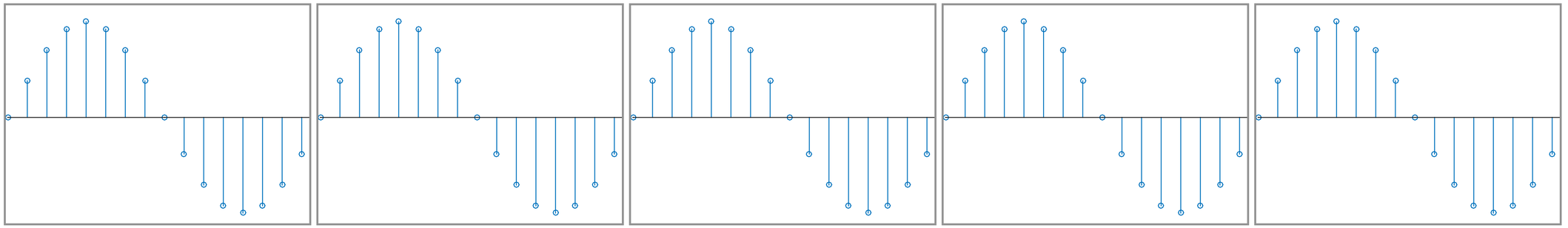
This runs once per sample within the block (default 16 times on Bela)

update `gReadPointer`, whose value persists

`n` is used to specify which frame of the output buffer to write

Uses of repeating buffers

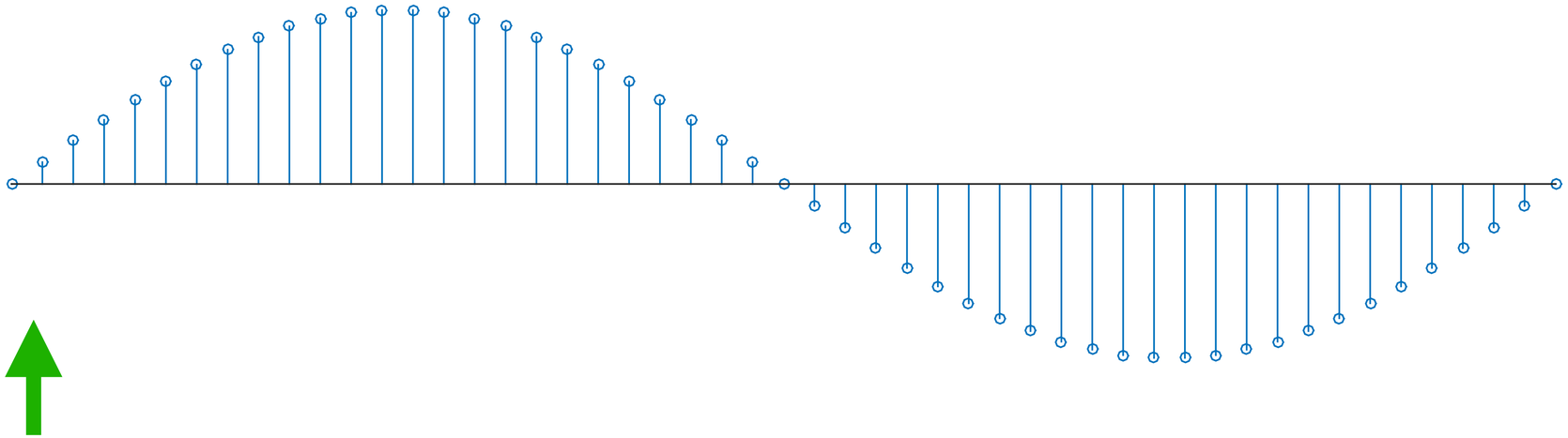
- There's another name for playing a buffer of samples over and over

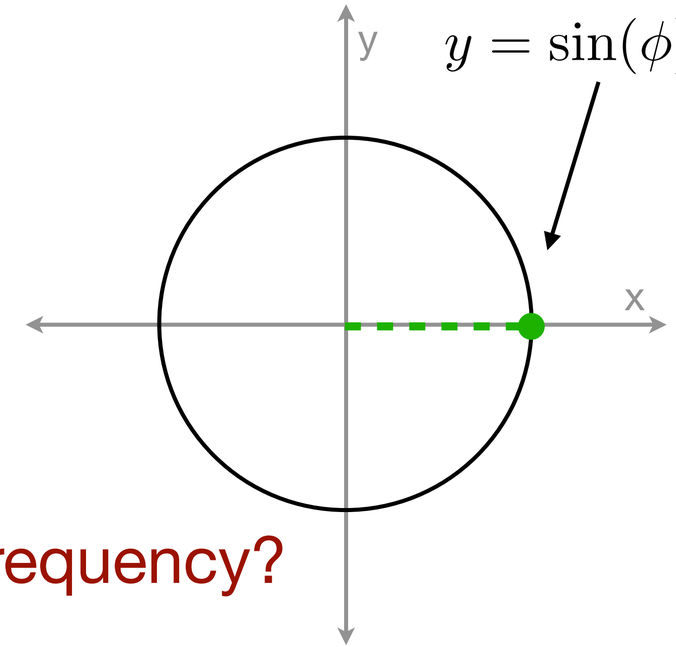


- **Wavetable**
- This is another way of building an oscillator
  - Rather than calculating the output mathematically every sample, play it from a buffer
  - Another way of thinking about it: a look-up table of output values
- Advantages of wavetables over using functions like `sin()`:
  - Less computation time per sample, because we can calculate the table at the beginning
  - More flexibility on the waveform (which could be nearly any shape)

Phase and frequency

- We need to keep track of where we are in the wavetable
  - i.e. keep track of the read pointer in the source buffer
  - We'll need to remember this between calls to `render()`, like before
- The read pointer is analogous to the phase in our previous oscillator:





- If the read pointer corresponds to phase, what determines frequency?
  - Frequency is the derivative (rate of change) of phase
  - Therefore, frequency is given by rate of change of the read pointer

Wavetable task

- In our previous examples, the read pointer always moved at a rate of one sample per sample
  - It advanced one sample in the source buffer for each real-time output sample
- To implement a wavetable oscillator with adjustable frequency, we need to change how much `gReadPointer` increments each sample
  - Don't forget to wrap `gReadPointer` if it falls off the end of the buffer!
  - How do we calculate how far to move based on the frequency?
  - Hint: look at the calculation in the original `sin()` oscillator and think about what should replace 2π
- Task: using the `wavetable` example, implement a wavetable oscillator
  - Adjust the code so the frequency is correct at 220Hz

Wavetable code

Previous approach:

$$x[n] = sin(\phi)$$
$$\phi = \phi + 2\pi f / f_s$$

range [0, 2π)

New approach:

$$x[n] = \text{wavetable}[\phi]$$
$$\phi = \phi + l_{table} f / f_s$$

range [0, l_{table})

```
int gReadPointer = 0;          // Position of the last frame we played

void render(BelaContext *context, void *userData)
{
    for(unsigned int n = 0; n < context->audioFrames; n++) {
        // Load a sample from the buffer
        float out = gAmplitude * gWavetable[gReadPointer];

        // Increment read pointer and wrap around when end of table is reached
        gReadPointer += gWavetableLength * gFrequency / context->audioSampleRate;
        while(gReadPointer >= gWavetableLength)
            gReadPointer -= gWavetableLength;

        for(unsigned int channel = 0; channel < context->audioOutChannels; channel++) {
            // Write the sample to every audio output channel
            audioWrite(context, n, channel, out);
        }
    }
}
```

What is a possible problem here?

Like subtracting 2π in the `sin()` example (don't just reset to 0!)

Minor point: why use `while()` and not `if()`?

Fractional read pointer

- To play the wavetable at a particular frequency, we increment the read pointer by: $\phi = \phi + l_{table} f / f_s$
- The increment might not be an integer
  - For $l_{table} = 512, f = 220\text{Hz}, f_s = 44100\text{Hz}$: increment is 2.55 samples
  - We need a `float` to hold the read pointer
  - If we use an `int`, the increment will be rounded down to 2 samples
    - Notice: `float` to `int` always rounds down (dropping the decimals)
    - What is the frequency in this case?
    - If $\Delta\phi = 2$ is the increment, then $f = \Delta\phi \cdot f_s / l_{table} = 172.2\text{Hz}$
- We can't use a `float` to index an array
  - Doesn't make sense in code: `gSampleBuffer[1.5]` has no meaning
  - Doesn't make sense mathematically: $x[1.5]$ (fractional index in discrete time) is undefined

Fractional read pointer

- What should we do instead?
- We can keep the fraction all the way up to the point we read the array
  - At that point, round down and convert to an `int`
  - Or alternatively, round to the nearest `int`
  - But only do this to access the array; don't change `gReadPointer` itself
- Try this now!
  - Hint: `floorf()` rounds a `float` downward

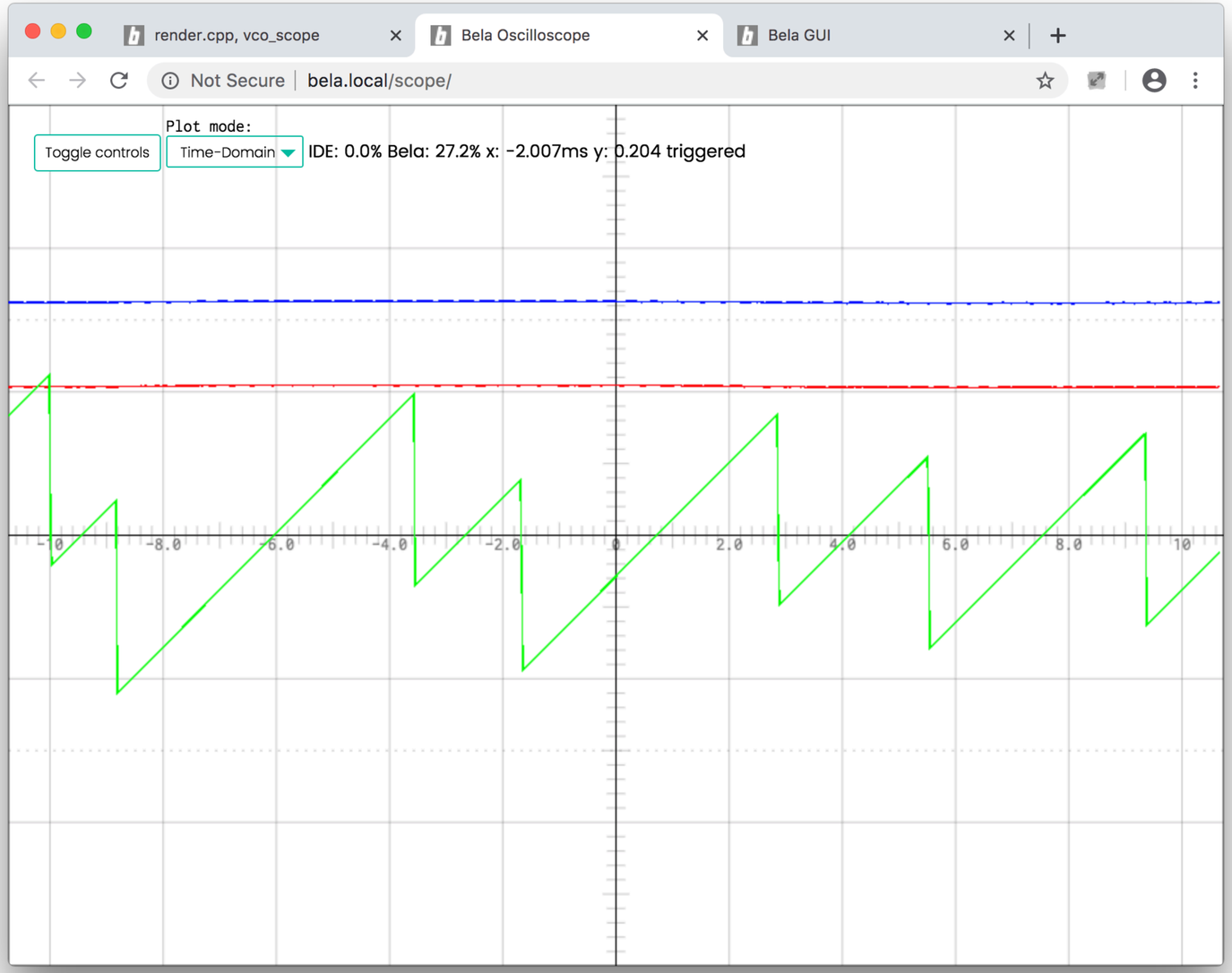
Fractional read pointer

```
float gReadPointer = 0;       // Position of the last frame we played

void render(BelaContext *context, void *userData)
{
    for(unsigned int n = 0; n < context->audioFrames; n++) {
        // Load a sample from the buffer
        int indexBelow = floorf(gReadPointer);
        float out = gAmplitude * gSampleBuffer[indexBelow];

        // Increment read pointer and wrap around when end of table is reached
        gReadPointer += gSampleBufferLength * gFrequency / context->audioSampleRate;
        while(gReadPointer >= gSampleBufferLength)
            gReadPointer -= gSampleBufferLength;

        for(unsigned int channel = 0; channel < context->audioOutChannels; channel++) {
            // Write the sample to every audio output channel
            audioWrite(context, n, channel, out);
        }
    }
}
```

We still need to convert the result to an `int`

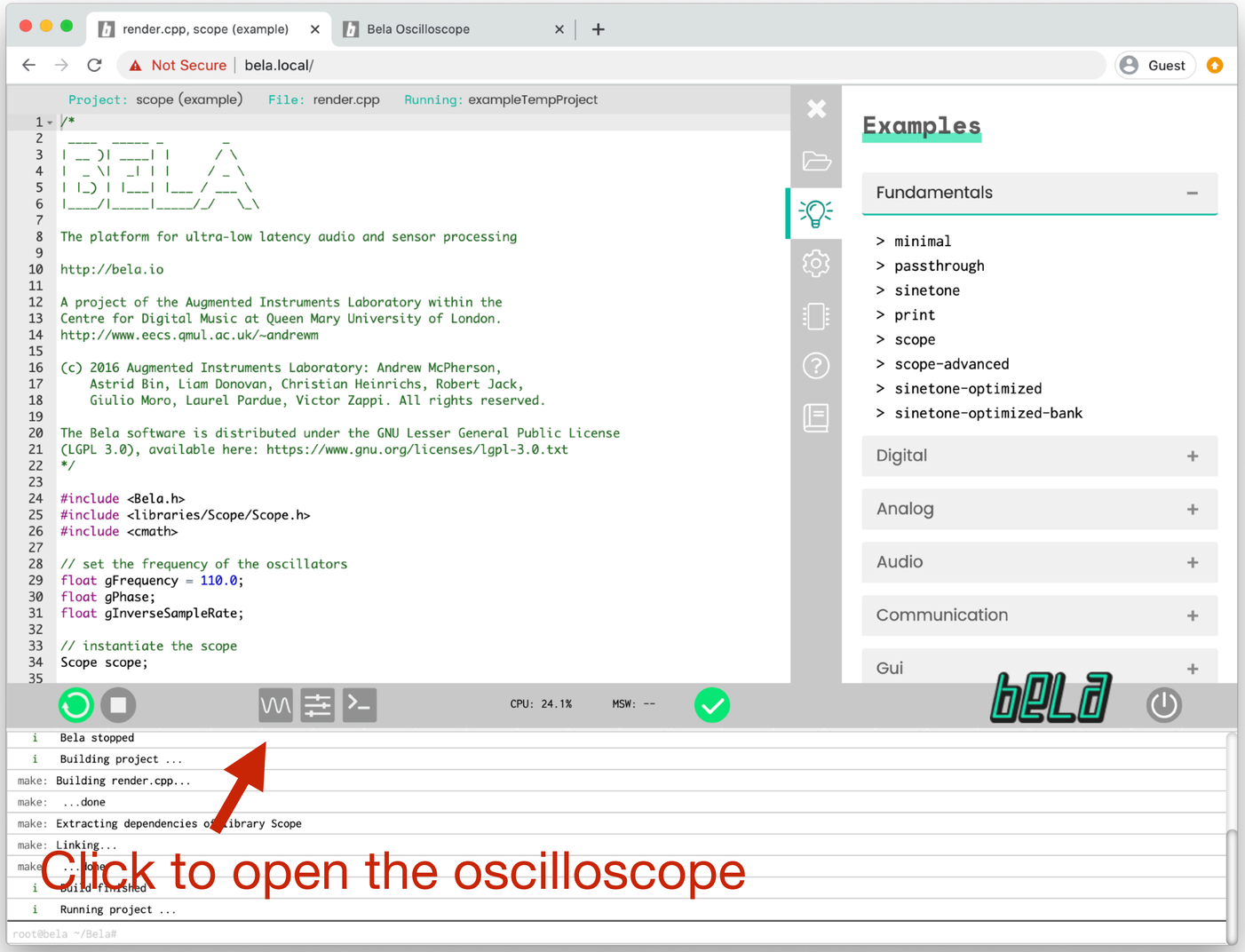
Oscilloscope

- Bela features an oscilloscope that runs in the browser
  - As the name "*oscillo-scope*" implies, a way of viewing oscillations (signals)
- Typical use of oscilloscope is an X-Y plot
  - Time on the X-axis
  - Amplitude on the Y-axis
- The scope can plot any signal made or used by your program
  - Not just I/O signals!



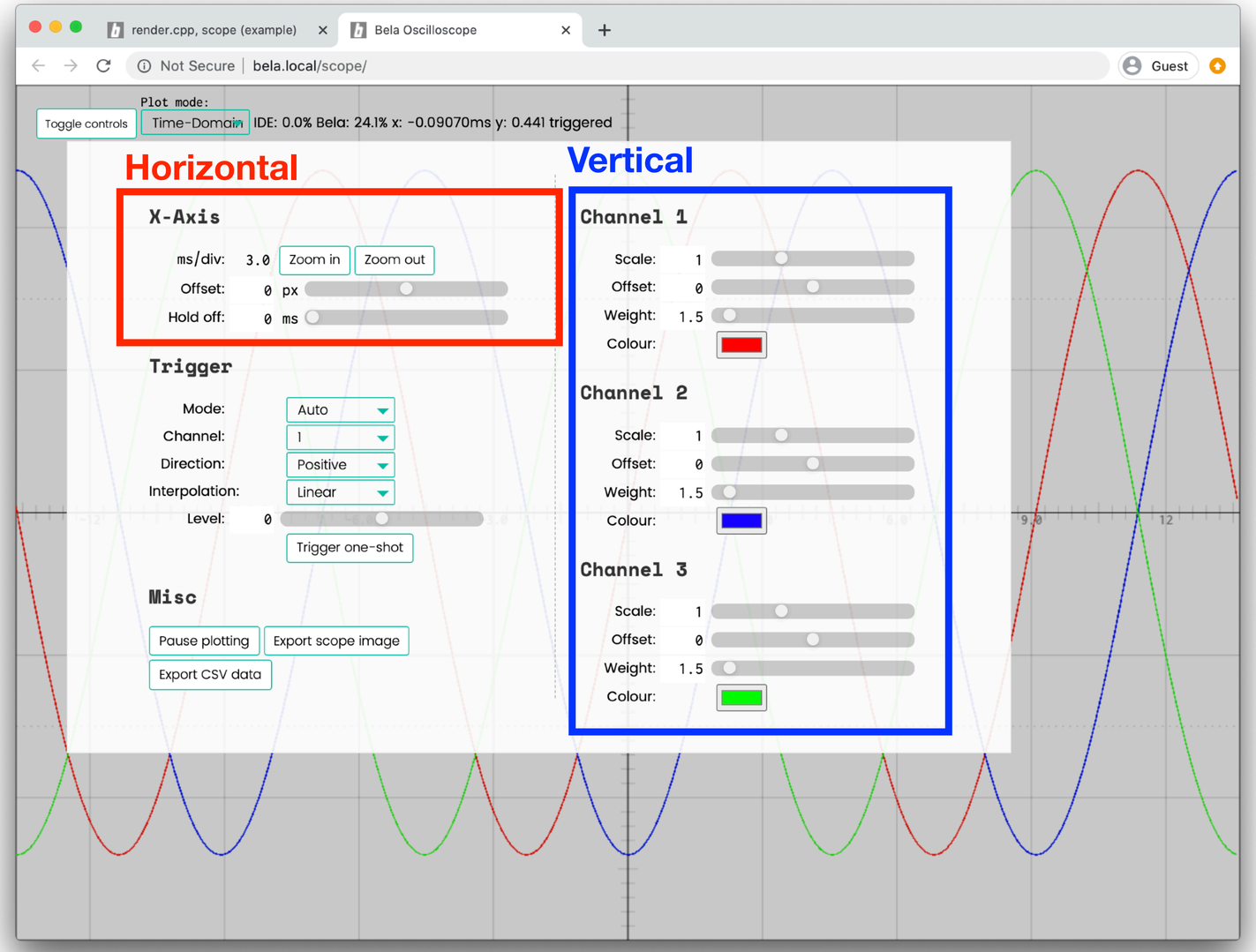
Launching the oscilloscope

- In the Examples tab of the IDE, open the project `Fundamentals/scope`
- Click the oscilloscope button to launch the scope in a new browser tab
  - You'll see the oscillator output in the scope



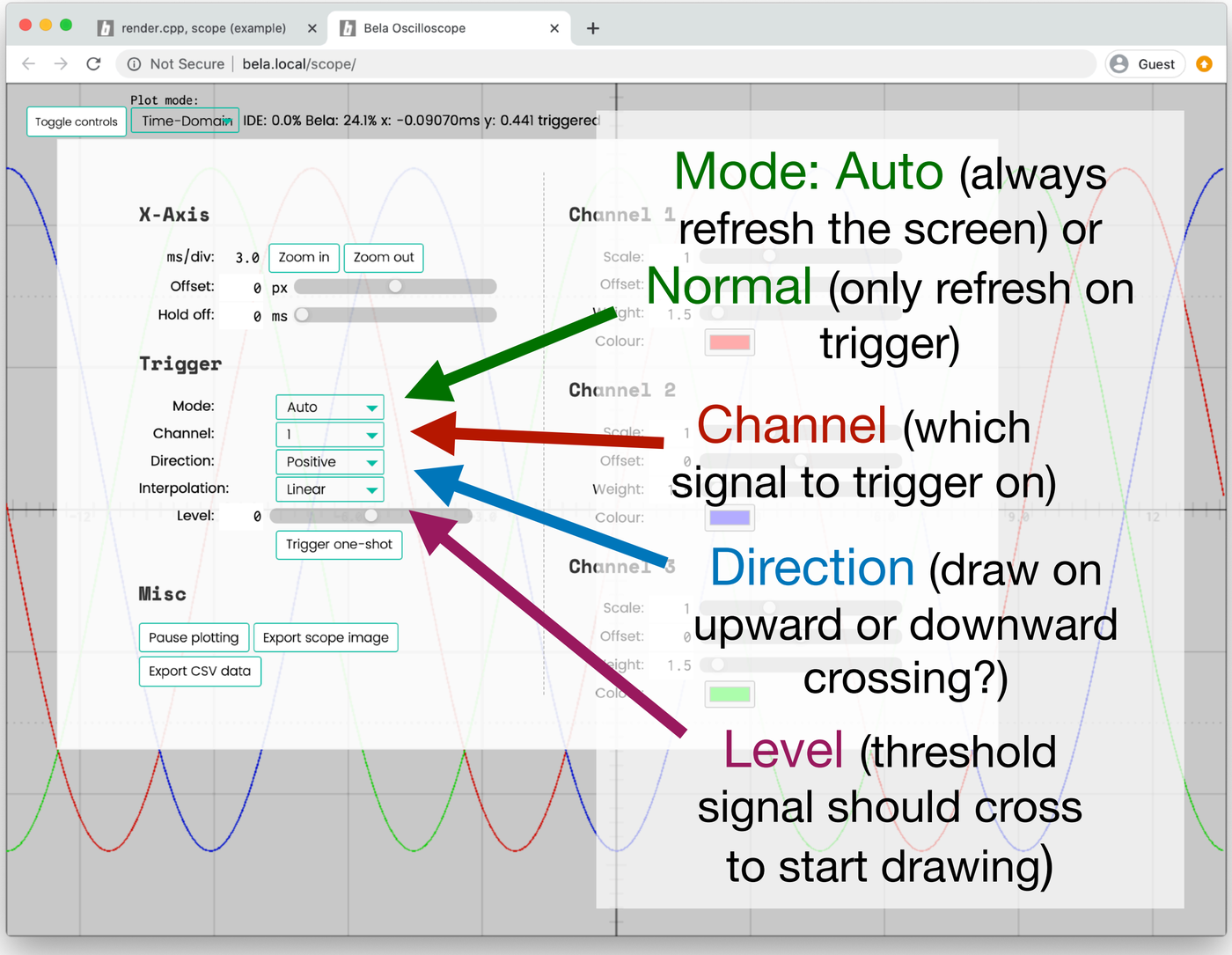
Oscilloscope controls: scaling

- The most important controls on any scope are the scales:
  - Horizontal (time) scale
  - Vertical (signal / voltage) scales
- On Bela, we can adjust:
  - Separate vertical scales for each channel (i.e. each signal)
  - Only one horizontal scale for all channels (why?)
- Remember default ranges:
  - Audio I/O is -1 to 1 full scale
  - Analog I/O is 0 to 1
  - Your own signals might have other ranges!



Oscilloscope controls: trigger

- Oscilloscopes show the signal in brief snapshots
  - One screenful corresponds to a limited window of time
  - To show the signal in real time, the scope regularly refreshes the screen with new time windows
- To keep the picture still, what do we need to do?
  - Start drawing at the same phase in the waveform each time
  - This is what the trigger does: sets where and how the screen starts drawing



Oscilloscope code

- To use the scope, you need to include it in your code and tell it which signals to display
- Steps:
  1. Include `Scope.h`
  2. Make a global variable of type `Scope`
  3. Set the number of channels and the sample rate in `setup()`
  4. Send the signals in `render()`

```
#include <Bela.h>
#include <libraries/Scope/Scope.h>

// Oscilloscope
Scope gScope;

bool setup(BelaContext *context, void *userData)
{
    // Set up the browser-based oscilloscope
    gScope.setup(3, context->audioSampleRate);

    return true;
}

void render(BelaContext *context, void *userData)
{
    for(unsigned int n = 0; n < context->audioFrames; n++) {
        // [calculate signal0, signal1, signal2 here]
        // Log signals to the oscilloscope
        gScope.log(signal0, signal1, signal2);
    }
}
```

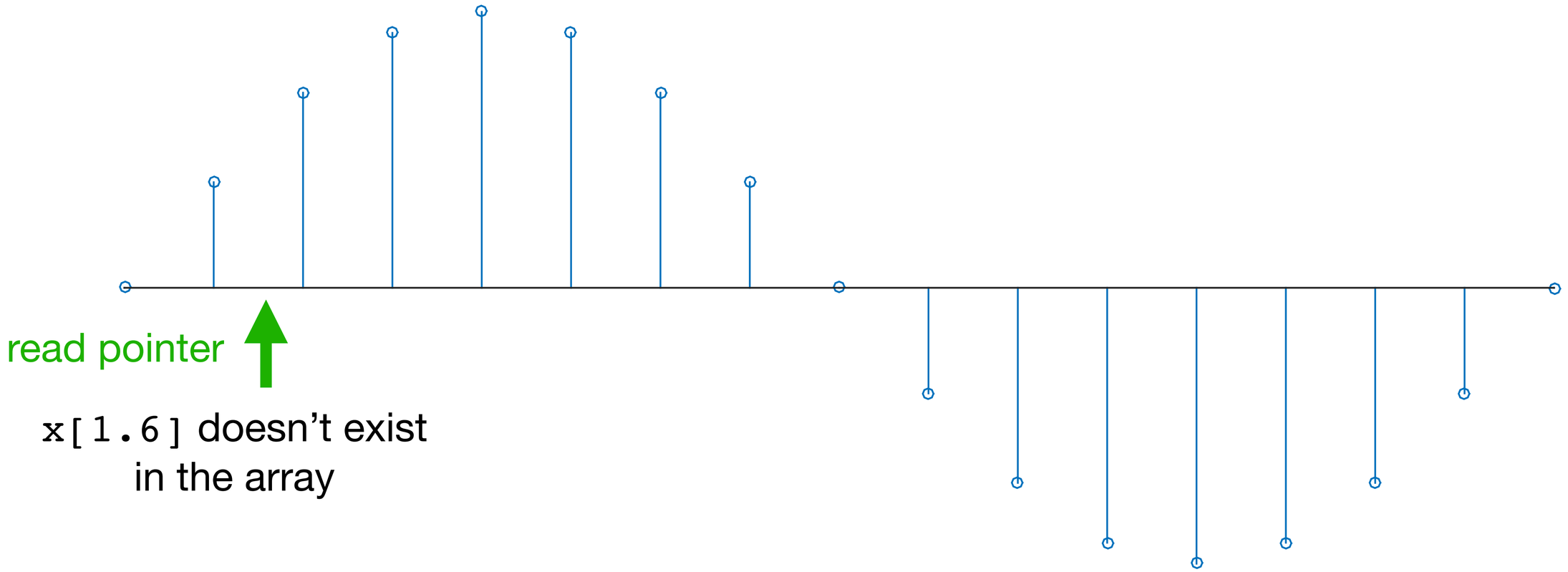
Oscilloscope code

- Use the method `Scope::log()` to send signals to the scope
- Two forms of `log()`:
  - Variable arguments (each channel in its own argument)
  - Or pass an array of `float`
- It's important that you call `log()` at the right rate
  - Call it once per frame at the sample rate you specified at initialisation
  - Need to send the values for all signals at the same time

```
#include <Bela.h>
#include <libraries/Scope/Scope.h>

// Oscilloscope
Scope gScope;

bool setup(BelaContext *context, void *userData)
{
    // Set up the browser-based oscilloscope
    gScope.setup(3, context->audioSampleRate);

    return true;
}

void render(BelaContext *context, void *userData)
{
    for(unsigned int n = 0; n < context->audioFrames; n++) {
        // [calculate signal0, signal1, signal2 here]
        // Log signals to the oscilloscope
        gScope.log(signal0, signal1, signal2);
    }
}
```

A better fractional index?

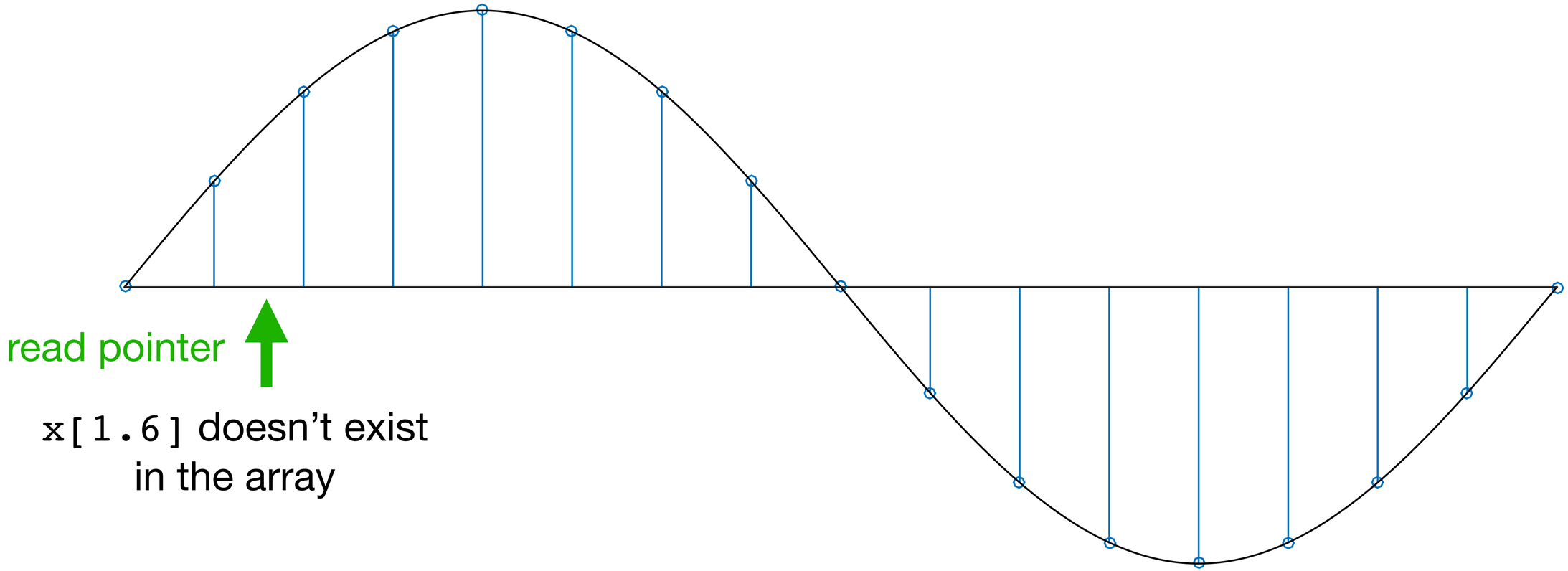
- We want to read fractional array values that, strictly speaking, don't exist...



A better fractional index?

- We want to read fractional array values that, strictly speaking, don't exist...

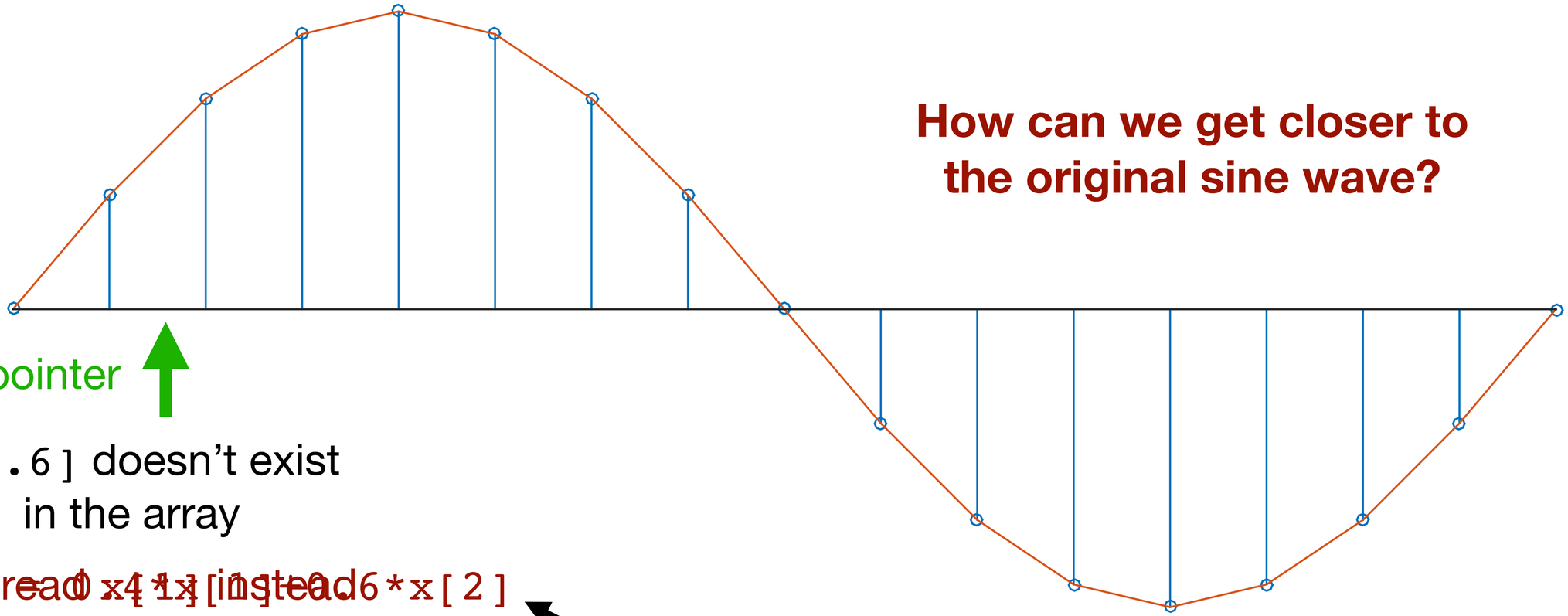
What we wish we could read:



A better fractional index?

- We want to read fractional array values that, strictly speaking, don't exist...

Take a weighted average of the two neighboring samples



How can we get closer to the original sine wave?

read pointer

`x[1.6]` doesn't exist in the array

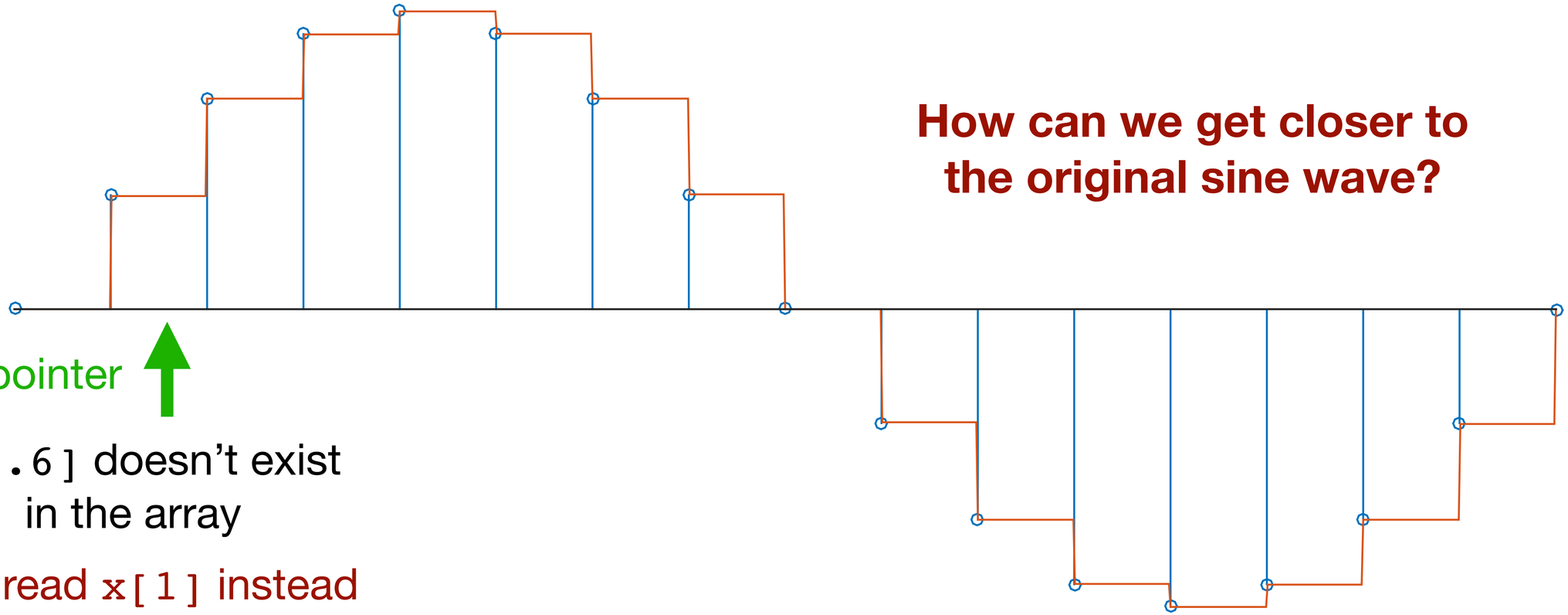
`x[1.6]` read `0.4*x[1]` instead of `0.6*x[2]`

why isn't it the other way around?

A better fractional index?

- We want to read fractional array values that, strictly speaking, don't exist...

What our code is actually reading:



How can we get closer to the original sine wave?

read pointer

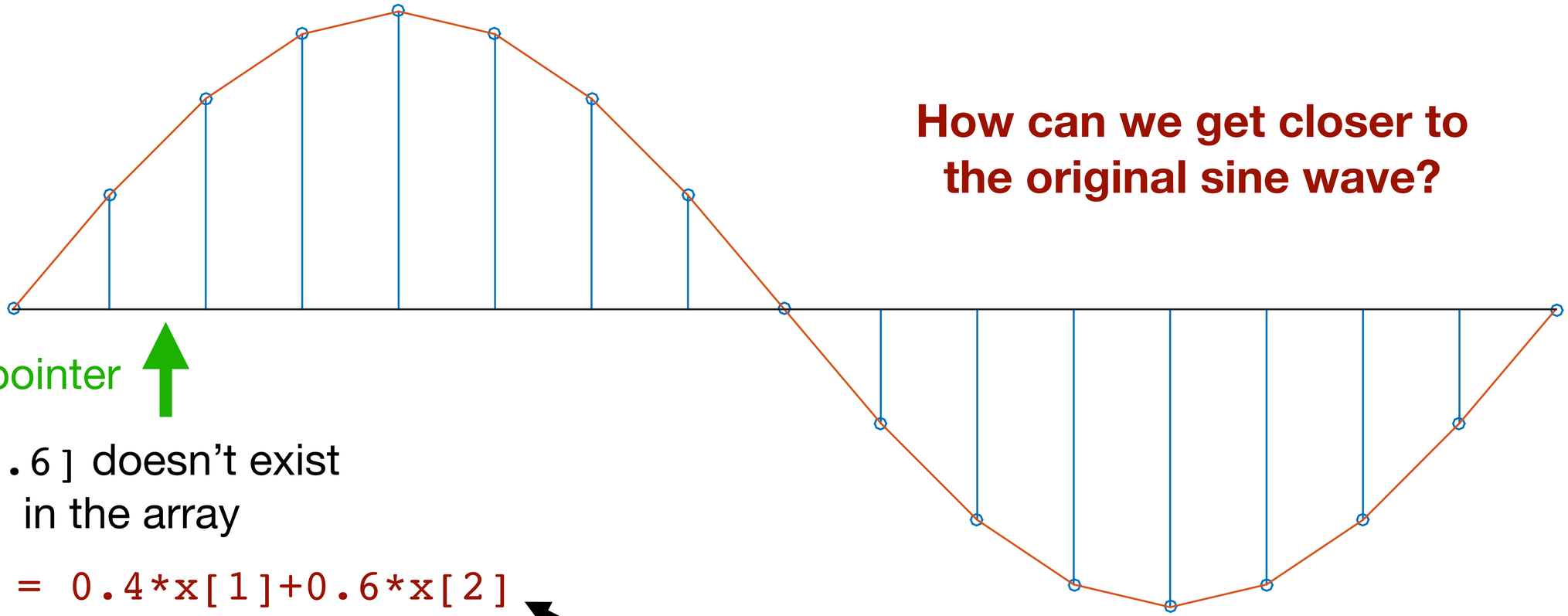
`x[1.6]` doesn't exist in the array

...so read `x[1]` instead

A better fractional index?

- We want to read fractional array values that, strictly speaking, don't exist...

Take a weighted average of the neighbouring samples



How can we get closer to the original sine wave?

read pointer

`x[1.6]` doesn't exist in the array

```
x[1.6] = 0.4*x[1]+0.6*x[2]
```

why isn't it the other way around?

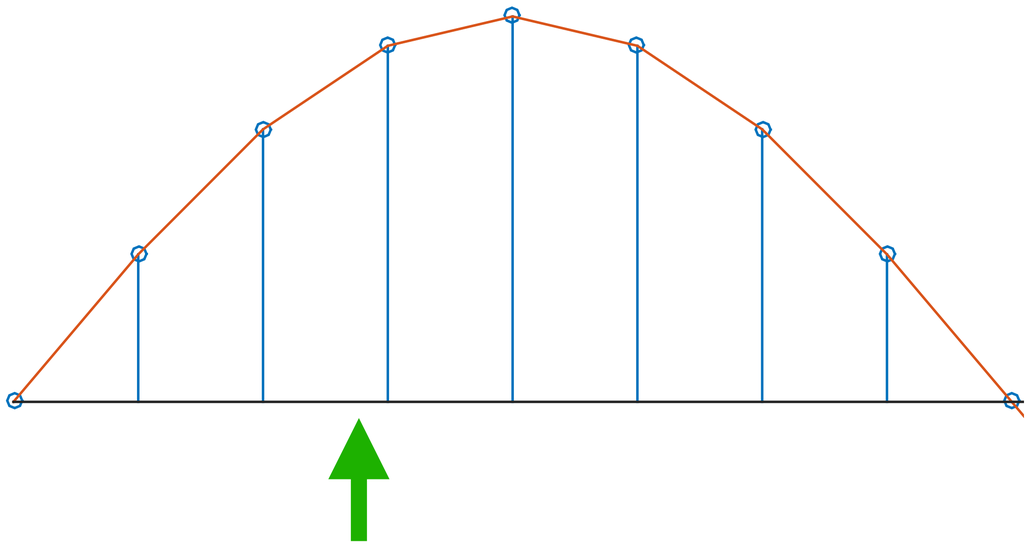
Linear interpolation

- Linear interpolation means taking a weighted average of neighbouring samples
  - The visual metaphor is drawing a straight line between the samples
- It is not a fully accurate reconstruction of an underlying continuous signal
  - Linear interpolation, like nearest-neighbour interpolation, will introduce distortion
  - A perfect mathematical interpolation needs *sinc* functions, which are not causal
  - There are more accurate approximations such as cubic interpolation
  - See Reiss & McPherson, *Audio Effects: Theory Implementation and Application*, p. 26
- To implement linear interpolation, we need to know:
  - The samples on either side of the fractional value
  - The fractional component of the index, to determine the weight of each sample

Linear interpolation

- Suppose *x* is our signal, *n* is the integer part of the index and *f* is the fractional part: 0 ≤ *f* < 1. Then we have:

$$x(n+f) = (1-f)x[n] + fx[n+1]$$



$$x(2.7) = 0.3x[2] + 0.7x[3]$$

- In a wavetable, what happens if *n*+1 is off the edge of the table?
  - e.g. $x(15.1)$ in a 16-sample table
  - Valid indices in the array are 0 to 15
  - Wrap around to the beginning: $x(15.1) = 0.9x[15] + 0.1x[0]$
- Task: implement linear interpolation in the wavetable example

Linear interpolation code

n, the integer below *n+f*

***n*+1**

n*+*f, a fractional value

f, the fractional component

our formula:

$$x(n+f) = (1-f)x[n] + fx[n+1]$$

```
// The pointer will take a fractional index. Look for the sample on
// either side which are indices we can actually read into the buffer.
// If we get to the end of the buffer, wrap around to 0.
int indexBelow = floorf(gReadPointer);
int indexAbove = indexBelow + 1;
if(indexAbove >= gWavetableLength)
  indexAbove = 0;

// For linear interpolation, we need to decide how much to weigh each
// sample. The closer the fractional part of the index is to 0, the
// more weight we give to the "below" sample. The closer the fractional
// part is to 1, the more weight we give to the "above" sample.
float fractionAbove = gReadPointer - indexBelow;
float fractionBelow = 1.0 - fractionAbove;

// Calculate the weighted average of the "below" and "above" samples
float out = gAmplitude * (fractionBelow * gWavetable[indexBelow] +
            fractionAbove * gWavetable[indexAbove]);

// Increment read pointer and reset to 0 when end of table is reached
gReadPointer += gWavetableLength * gFrequency / context->audioSampleRate;
while(gReadPointer >= gWavetableLength)
   gReadPointer -= gWavetableLength;
```

Populating the wavetable

- How can we generate the contents of the wavetable?
  - Load samples from a file
  - Generate mathematically
- Suppose the wavetable has length N
  - Then the range of array indexes goes from 0 to N-1
  - We can populate its contents with a `for()` loop inside of `setup()`:

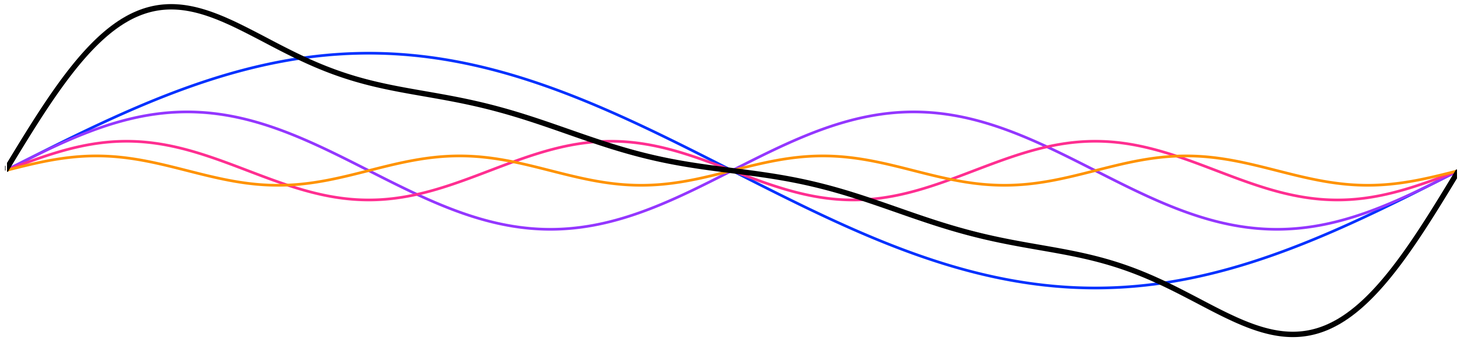
```
for(unsigned int n = 0; n < gWavetableLength; n++) {
    gWavetable[n] =
}
```

- Task: change `wavetable` to implement a sine rather than a triangle wave
  - Remember, the range 0 to N-1 needs to correspond to one complete oscillation
    - i.e. an index of N would correspond to a phase of 2π (`2.0*M_PI`)

More complex waveforms

- One way of building more complex waveforms is to add several harmonics of a fundamental frequency
  - This process is known as additive synthesis, and is a whole topic of its own!
  - In general, you can sum together any phase of integer multiples of the fundamental frequency

$$w(\phi) = 0.5\sin(\phi) + 0.25\sin(2\phi) + 0.125\sin(3\phi) + 0.0625\sin(4\phi)$$



- Task: in the wavetable project, implement the above equation in `setup()`
  - What is the relationship between ϕ and n in the `for()` loop?

Keep in touch!

Social media:

@BelaPlatform

forum.bela.io

blog.bela.io

More resources and contact info at:

learn.bela.io/resources