Building a Spatial Audio Plugin

Allen Lee
2020/10/01

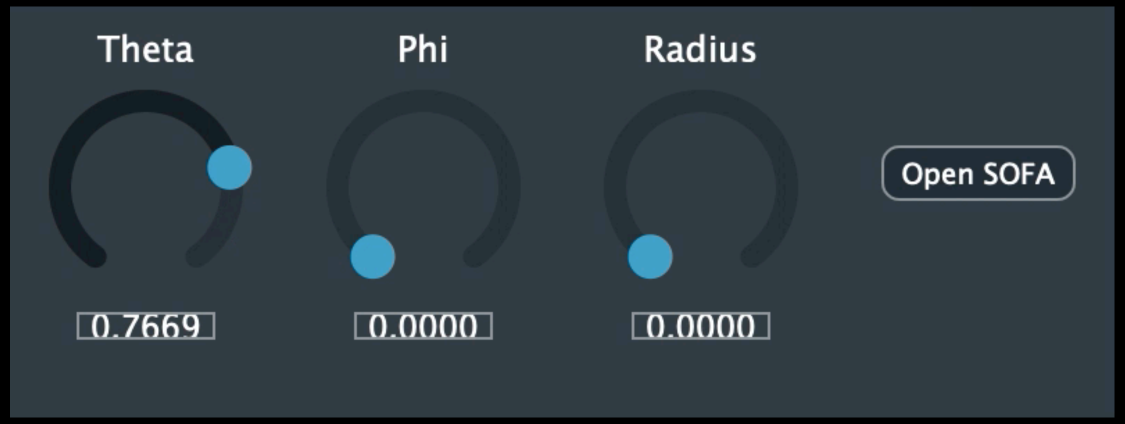
Overview

- Not a spatial audio expert!
- Recently grew interested in learning more about SA
- Created scripts to apply spatial processing offline
- Wanted something more real-time
- Also wanted to try creating a plugin…
- …and get to know JUCE better

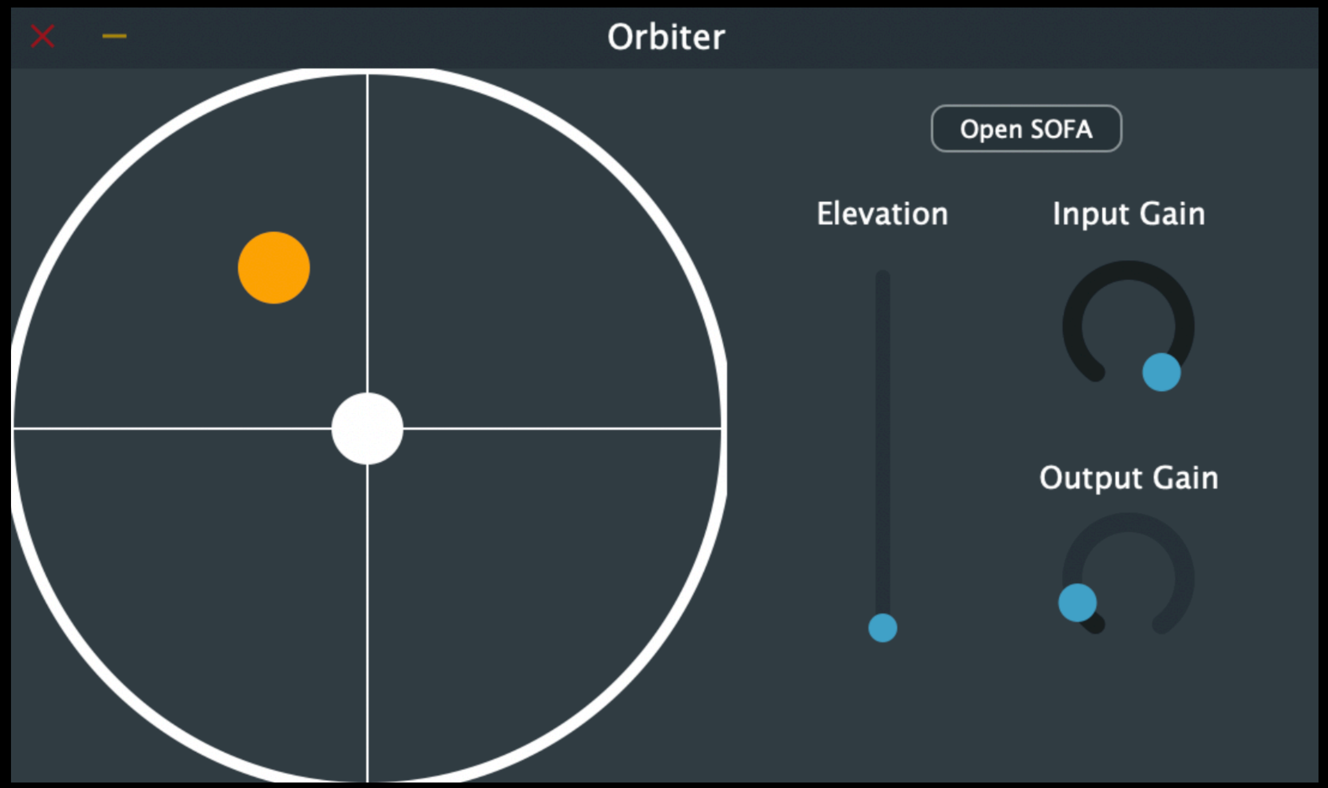
Orbiter

- 3D panner plugin
- User specifies HRTF datasets
- Made with JUCE

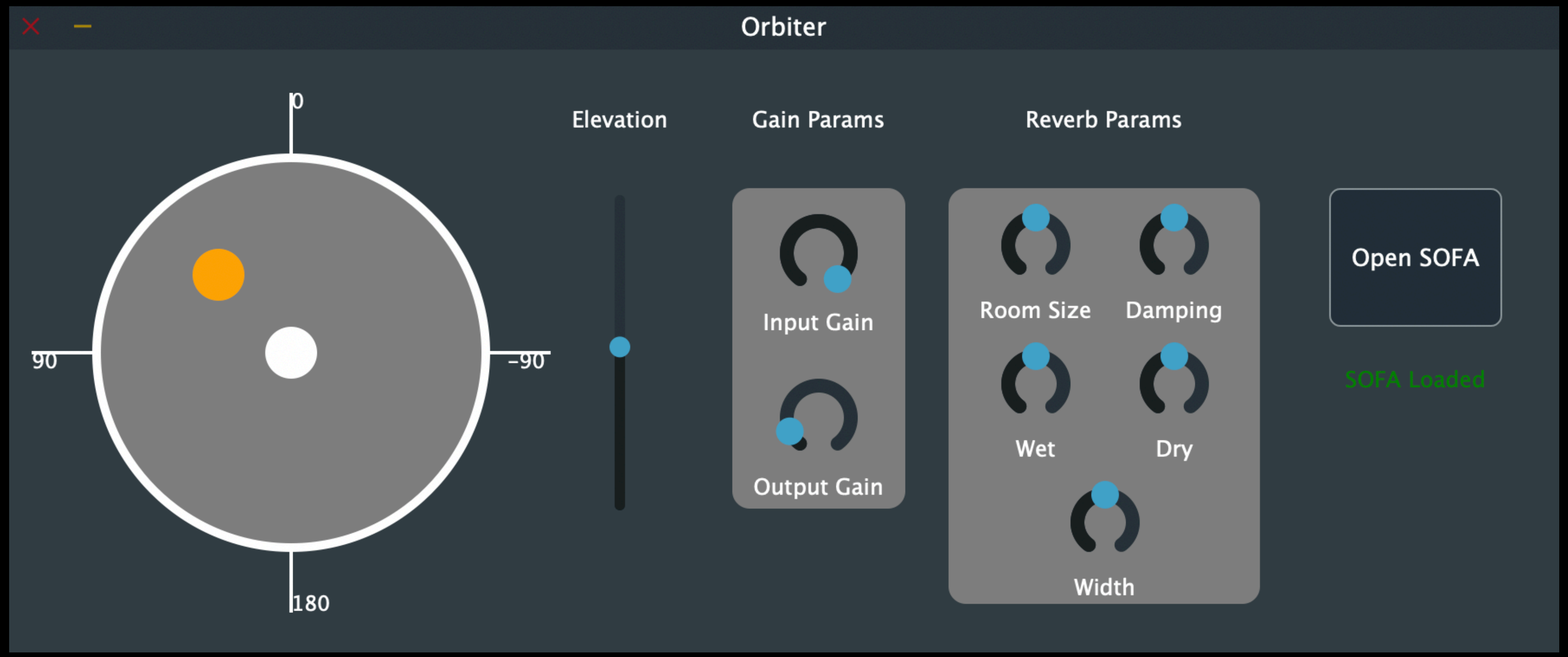
Version 0



Version 0.1



Version 0.2



Overview

- Brief Introduction to Spatial Audio
- Plugin Development

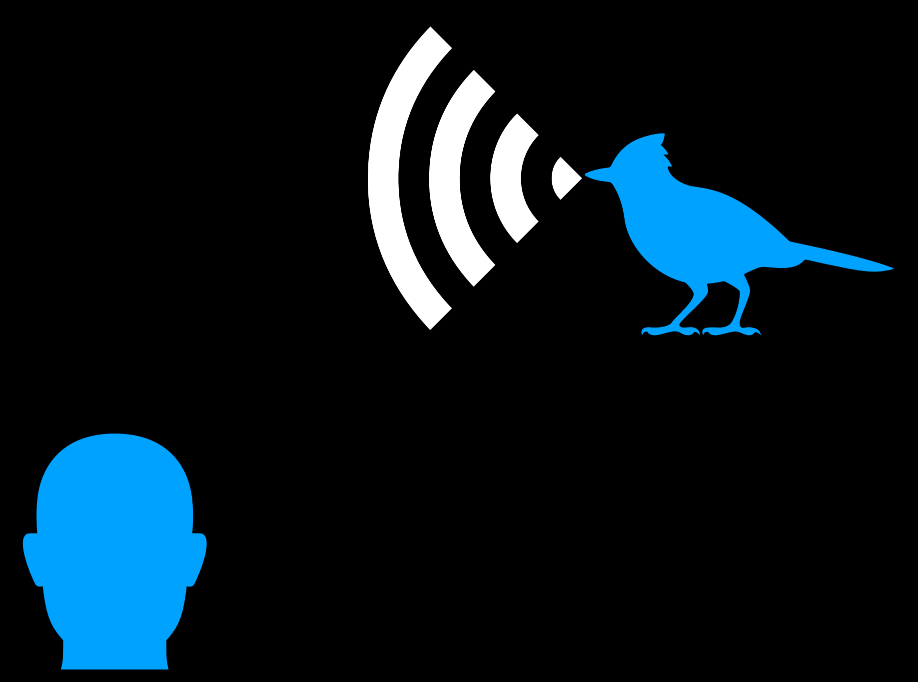
Introduction to Spatial Audio

Spatial Audio

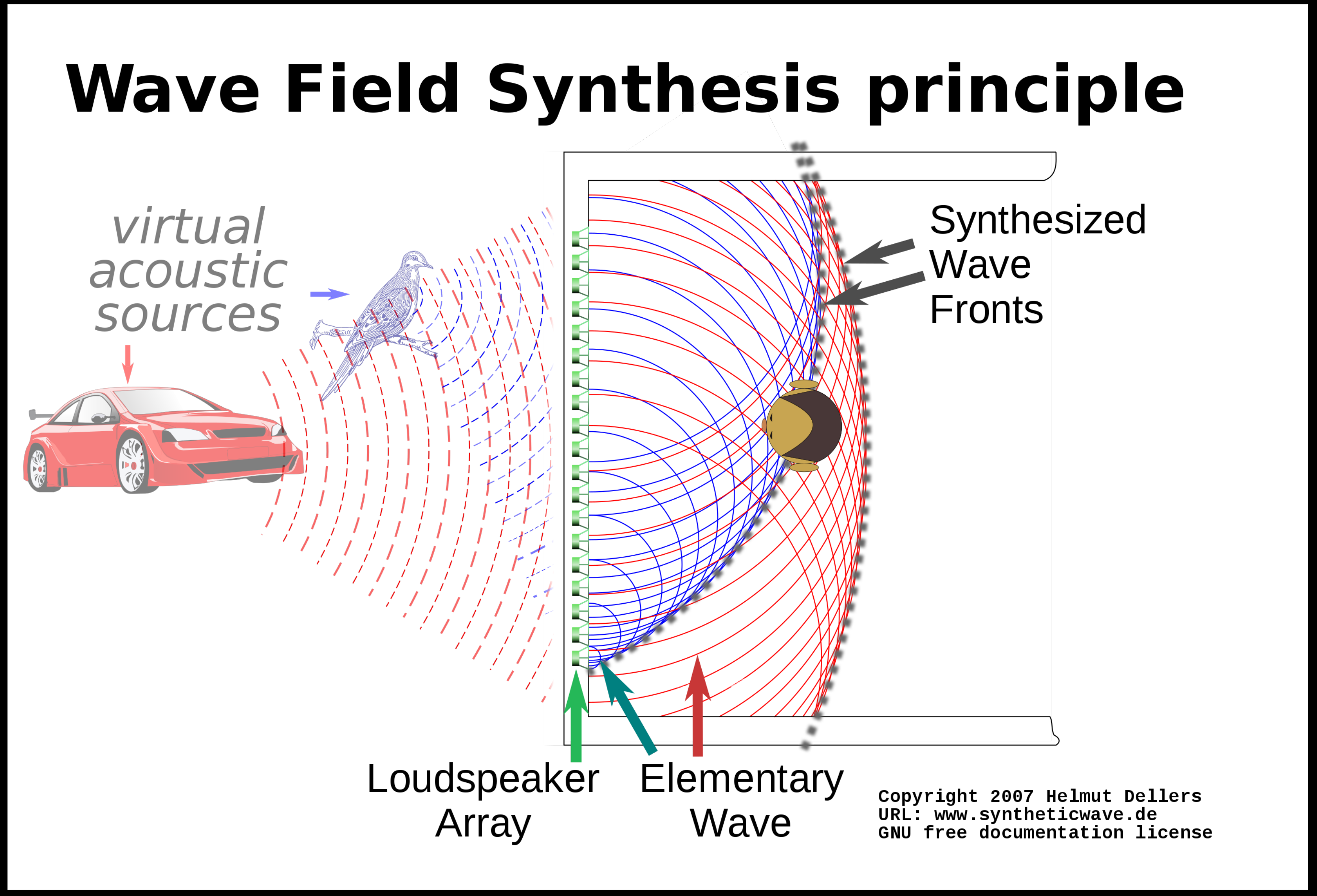
Overview

- Creating the illusion of hearing an audio source from a position in space
- Several ways to achieve this
  - Surround Sound (5.1, 7.1, 22.2 surround…)
  - Wave field synthesis
  - Binaural reproduction

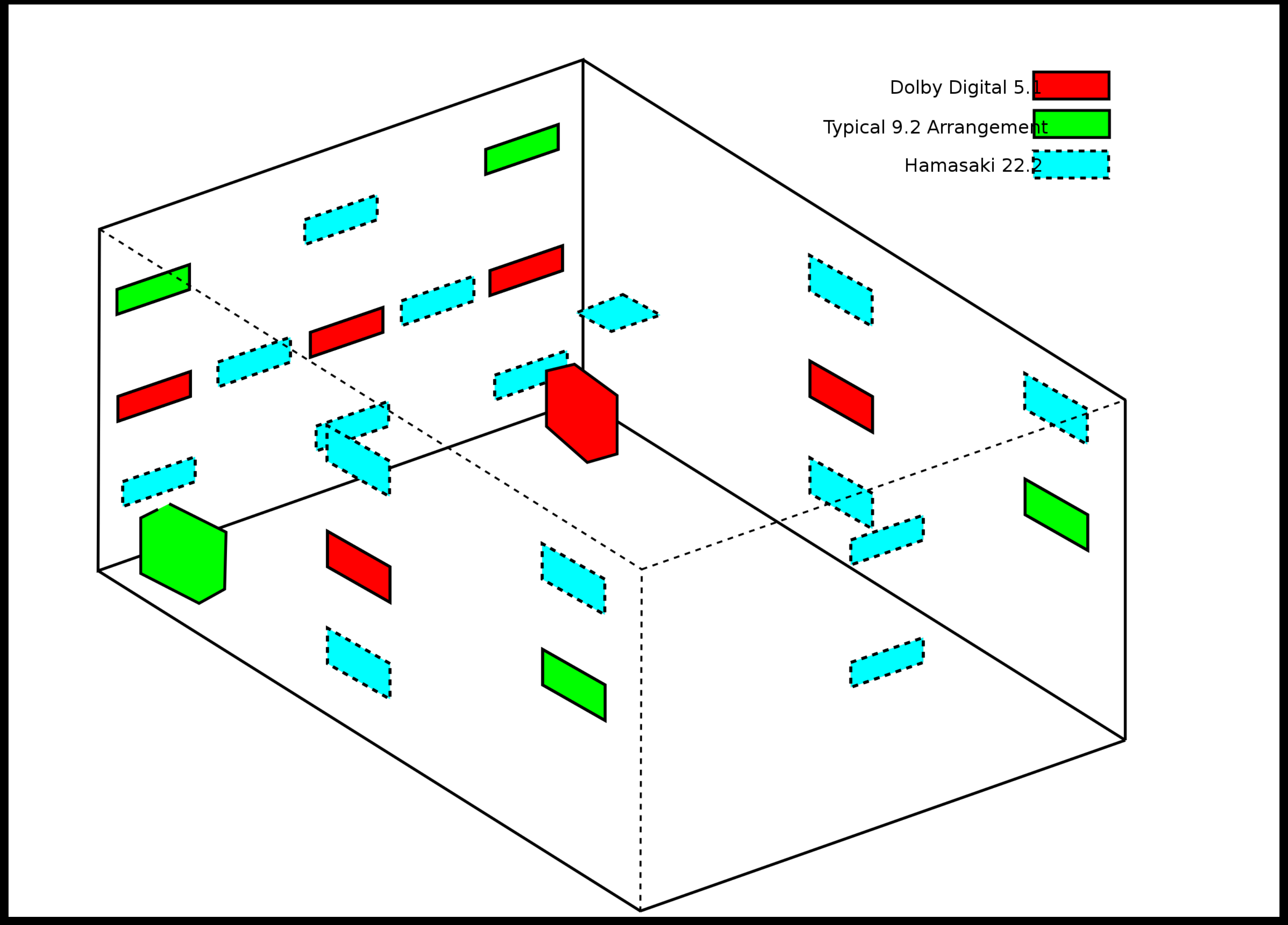
Localizing Direction of Chirp



Wave Field Synthesis



Surround Sound



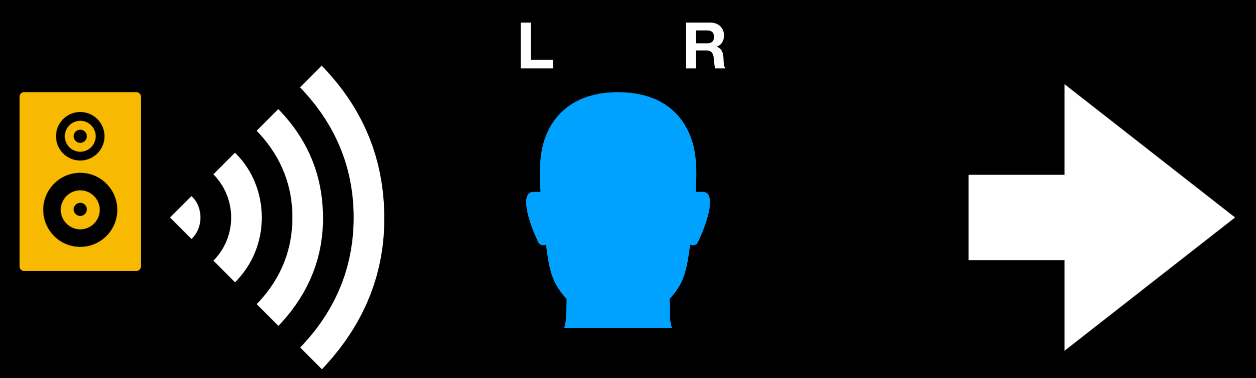
Source: Andrew Butitta
https://commons.wikimedia.org/wiki/File:Multi_Channel_Audio_Diagram.svg

Spatial Audio

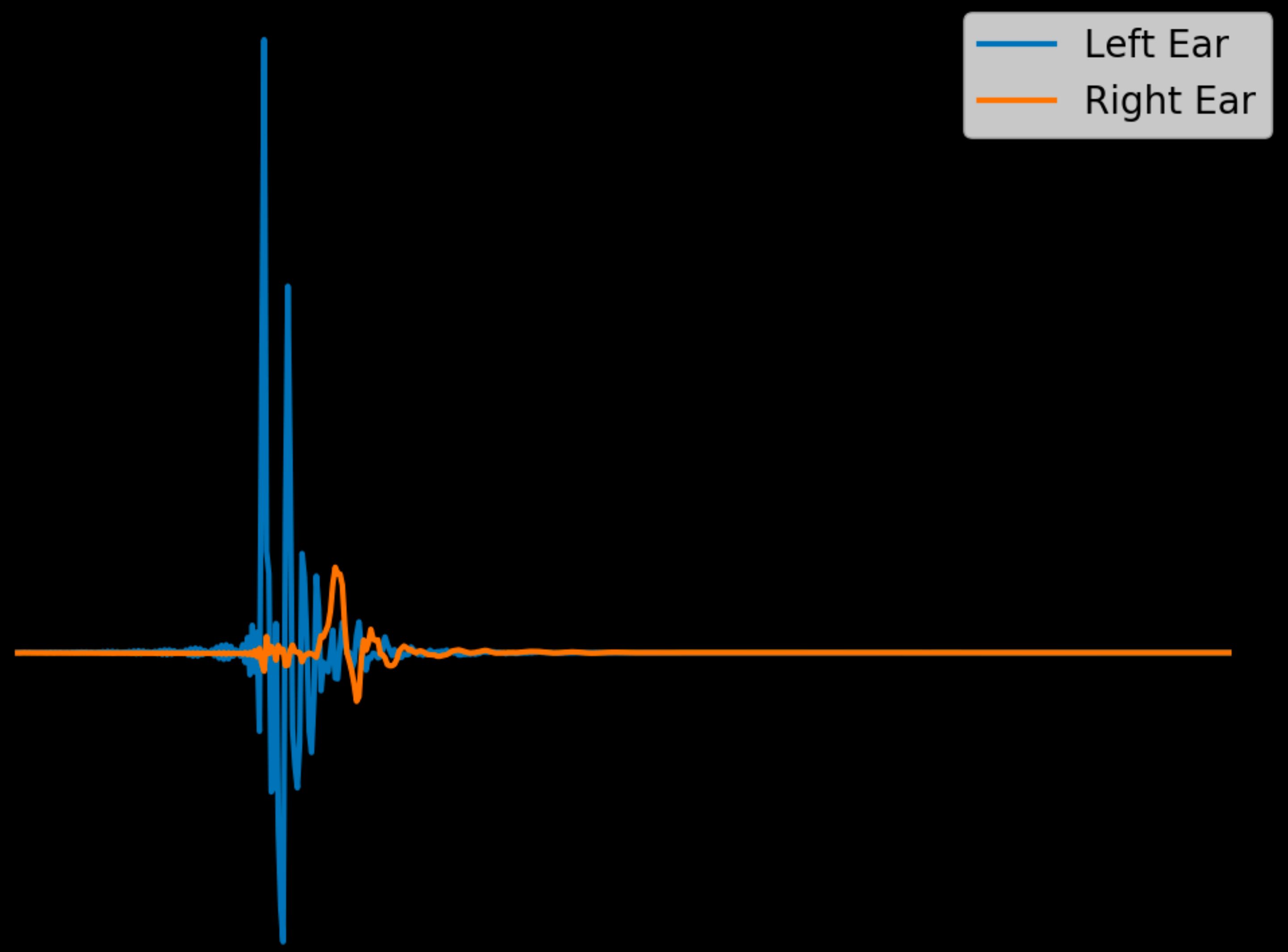
Binaural Reproduction

- Reproducing spatial audio through headphones
- A sound wave arrives at each ear at slightly different times and at different intensities
- Our brain processes these differences to determine where the sound came from
- ILD (Intra-aural Level Difference) and ITD (Intra-aural Time Difference)
- ILDs and ITDs are captured in *Head Related Impulse Responses (HRIR)*

Audio Source positioned at 90°



Recorded Audio



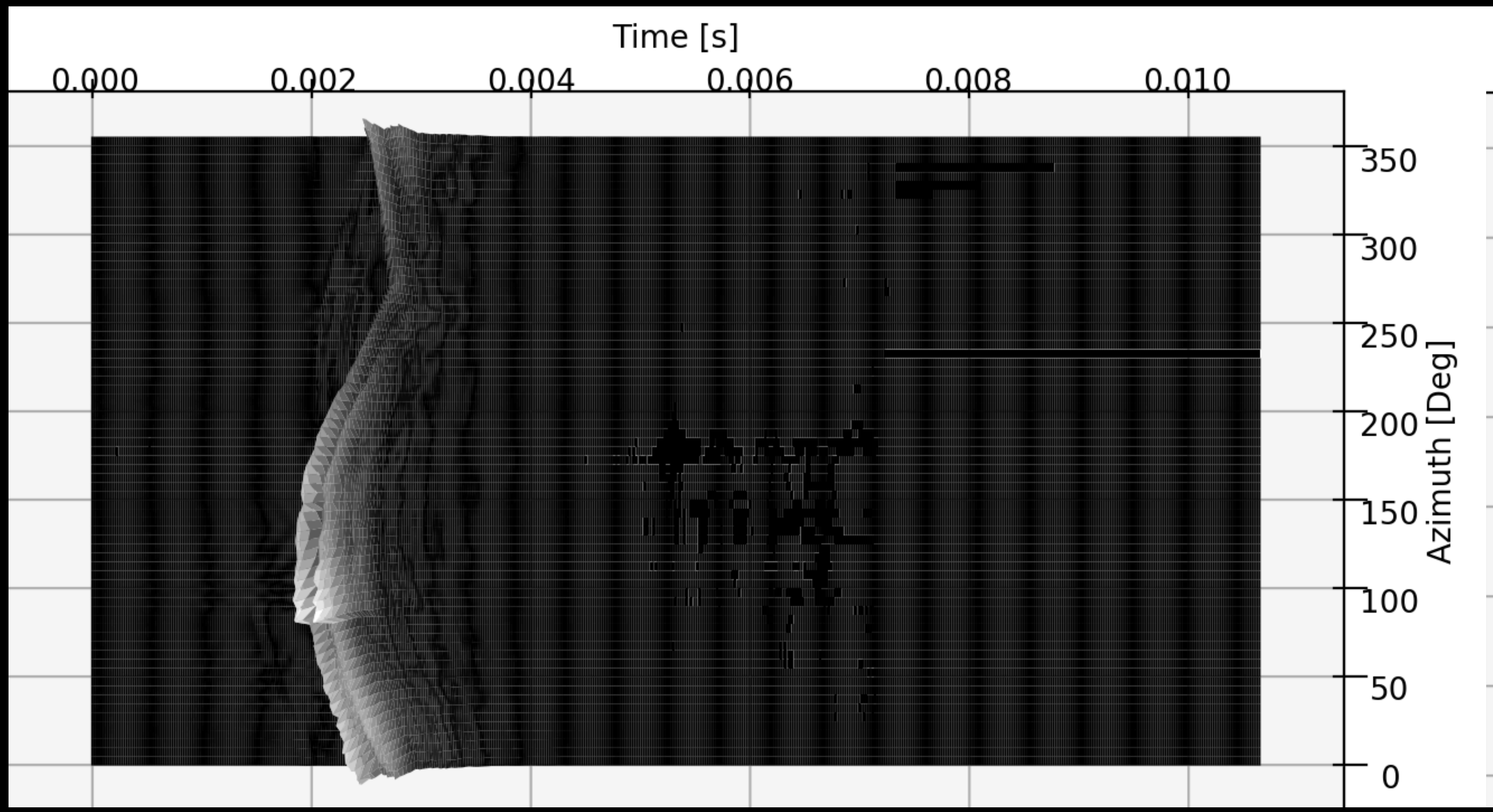
Impulse reaches the left ear first
The head attenuates impulse level which arrives at the right ear later and 'quieter'

Dataset used for charts are from Tohoku University RIEC HRTF Datasets
http://www.riec.tohoku.ac.jp/pub/hrtf/index.html

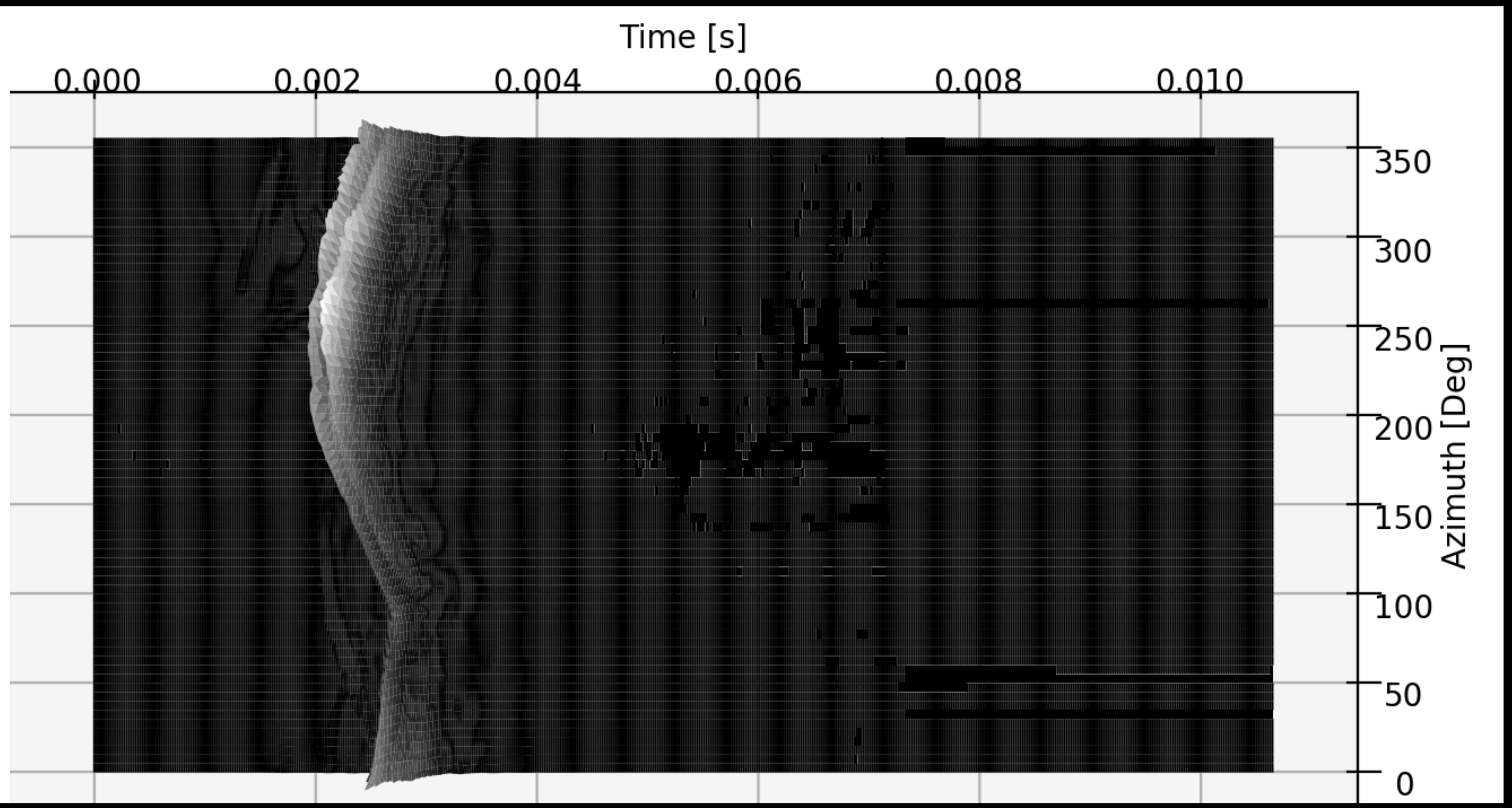
Spatial Audio

Map of HRIRs for Varying Azimuths

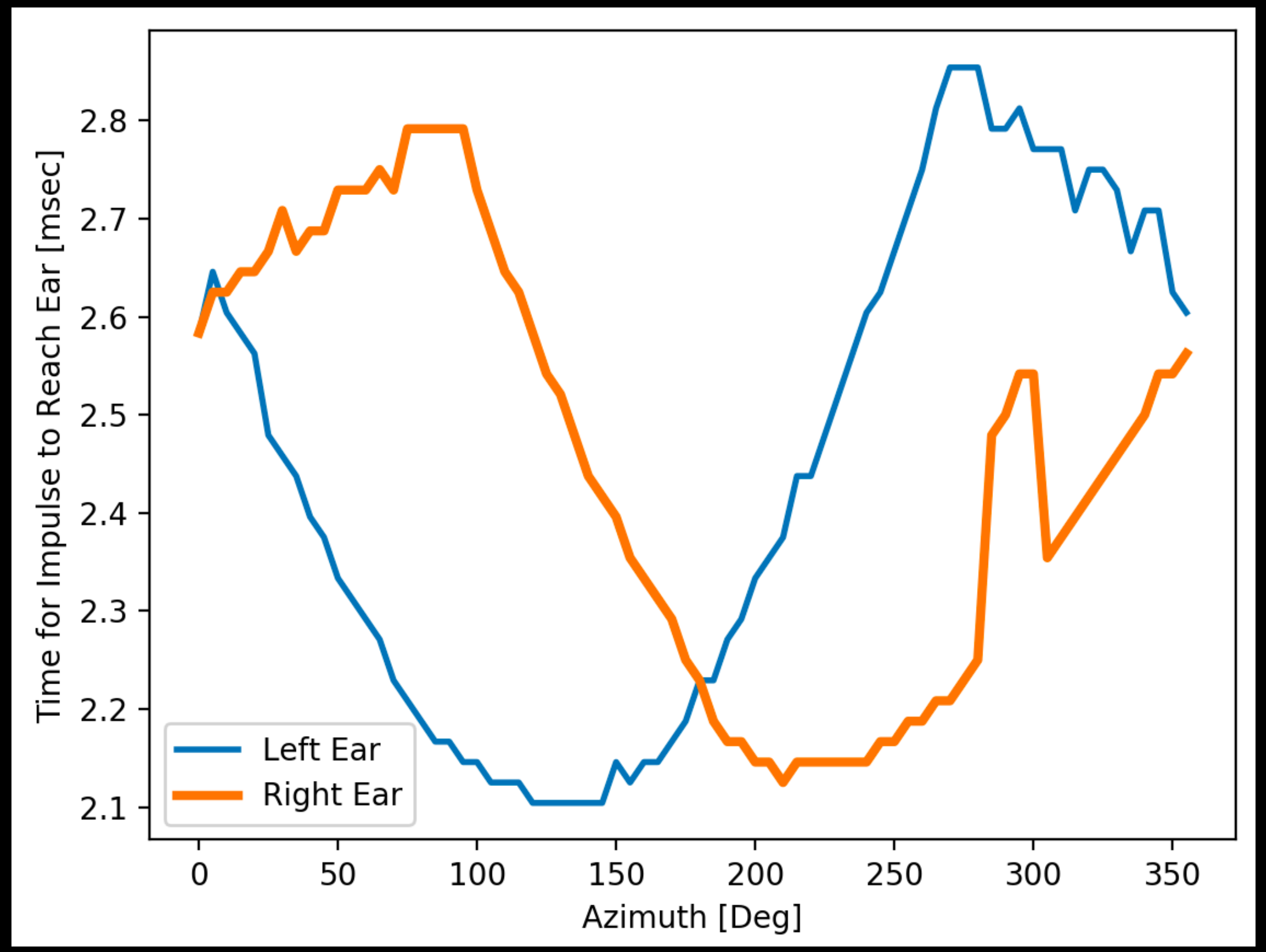
Left Impulse Responses



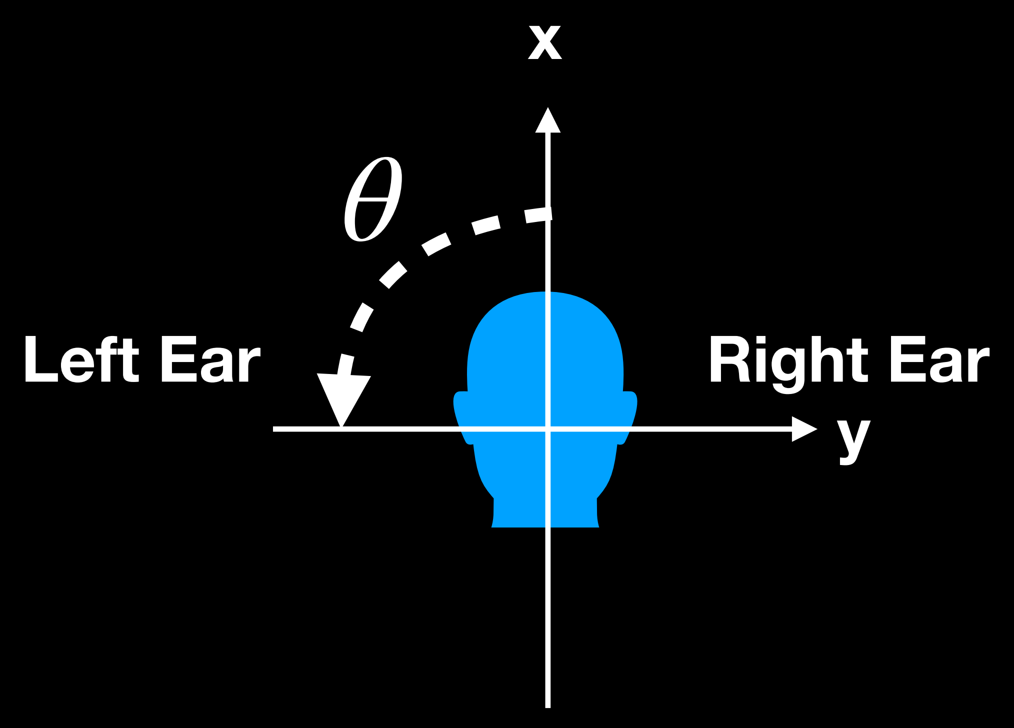
Right Impulse Responses



Impulse Time of Arrival



Dataset used for charts are from Tohoku University RIEC HRTF Datasets
http://www.riec.tohoku.ac.jp/pub/hrtf/index.html



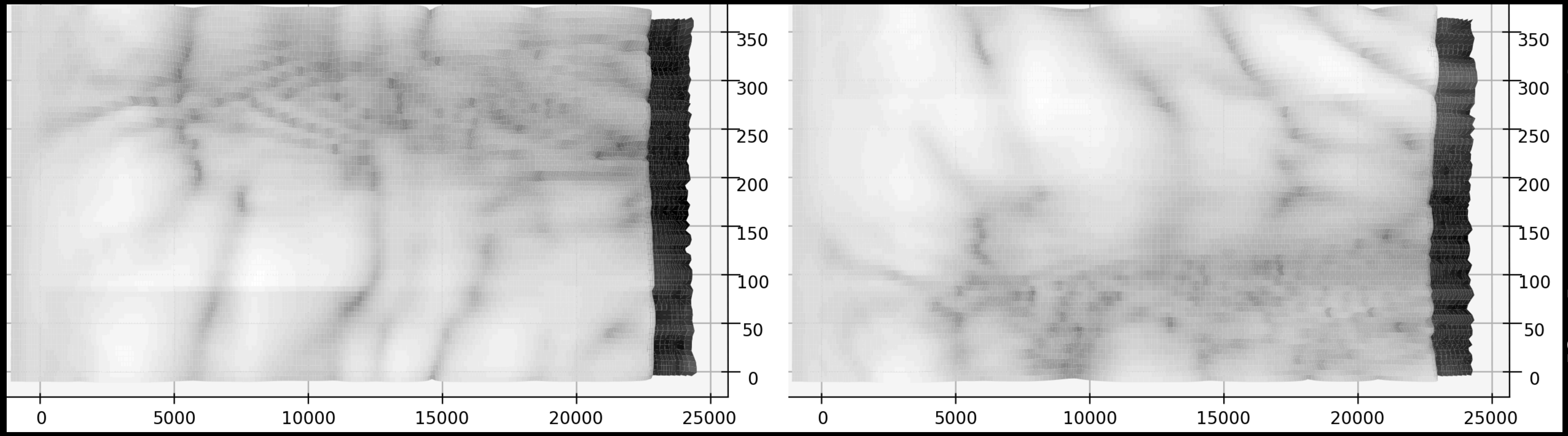
Azimuth Convention

Spatial Audio

Head Related Transfer Function

- Anatomy of the outer ear (pinnae) also plays an effect in localization
- Pinnae filters out different frequencies which changes with direction
- Your brain also performs frequency analysis for localization
- Your ear's filter characters can be seen by taking the FFT of the HRIR
- Often called the *Head Related Transfer Function (HRTF)*

Left HRTF **Right HRTF**



y axis: Azimuth [Deg]

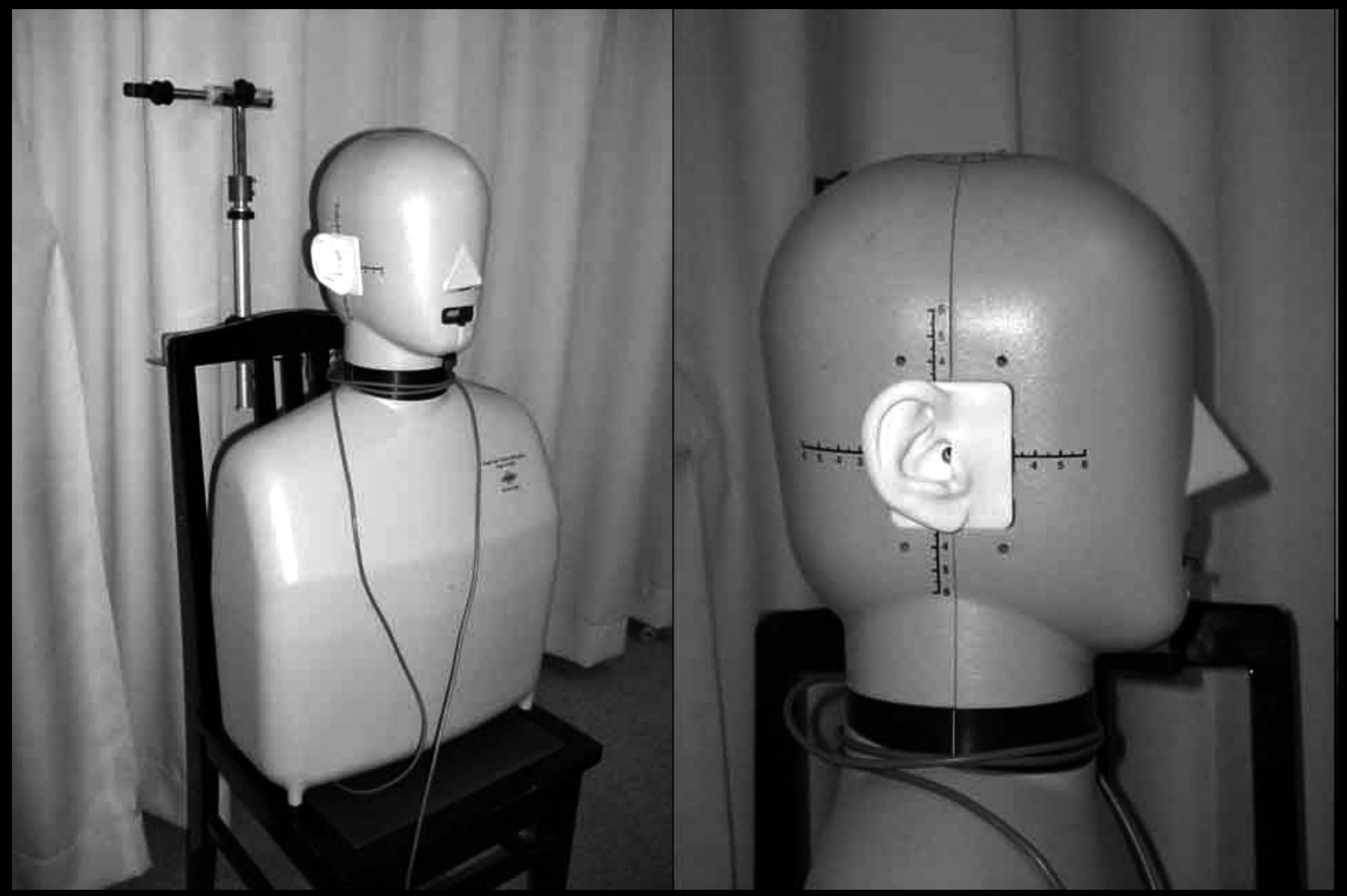
x axis: Frequency [Hz]

Data used for charts are from Tohoku University RIEC HRTF Datasets
http://www.riec.tohoku.ac.jp/pub/hrtf/index.html

Spatial Audio

HRTF Measurement

- Personalized HRTFs can be measured by wearing special microphones in the ears, and recording audio impulses from different angles
- Can also use a special dummy head microphone fitted with anatomically matching ears
- Some research in computing HRTFs from 3D scanned images of the head



Brujel & Kjaer Head and Torso Simulator
https://commons.wikimedia.org/wiki/File:Head_and_torso_simulator.jpg

Spatial Audio

HRTF Measurement

- Users often report that audio with HRTFs applied seem to come from *inside* their head
- Need room reverberation effects to add to the realism
- Can mix reverberated signal with the binaural signal OR put the HRTF measurement setup in a reverberant room
- HRIRs with room characteristics are called *Binaural Room Impulse Responses (BRIR)*

Example BRIR Measurement Setup

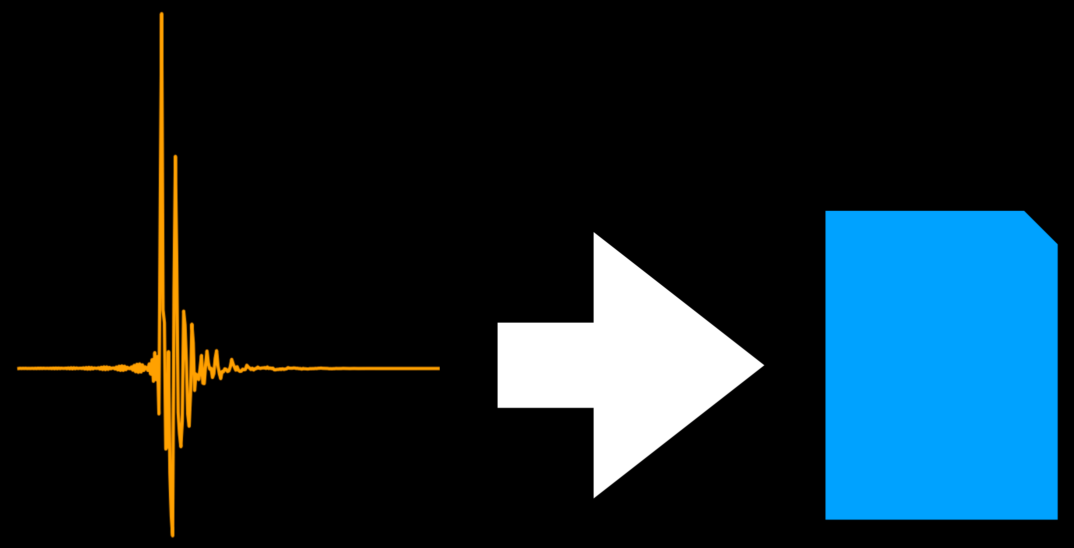
TU Conference Room BRIR Measurement Setup
https://github.com/ShanonPearce/ASH-IR-Dataset/blob/master/Images/Rooms/Conference_Room_TU_Ilmenau.jpg

Spatial Audio

HRTF Data Storage

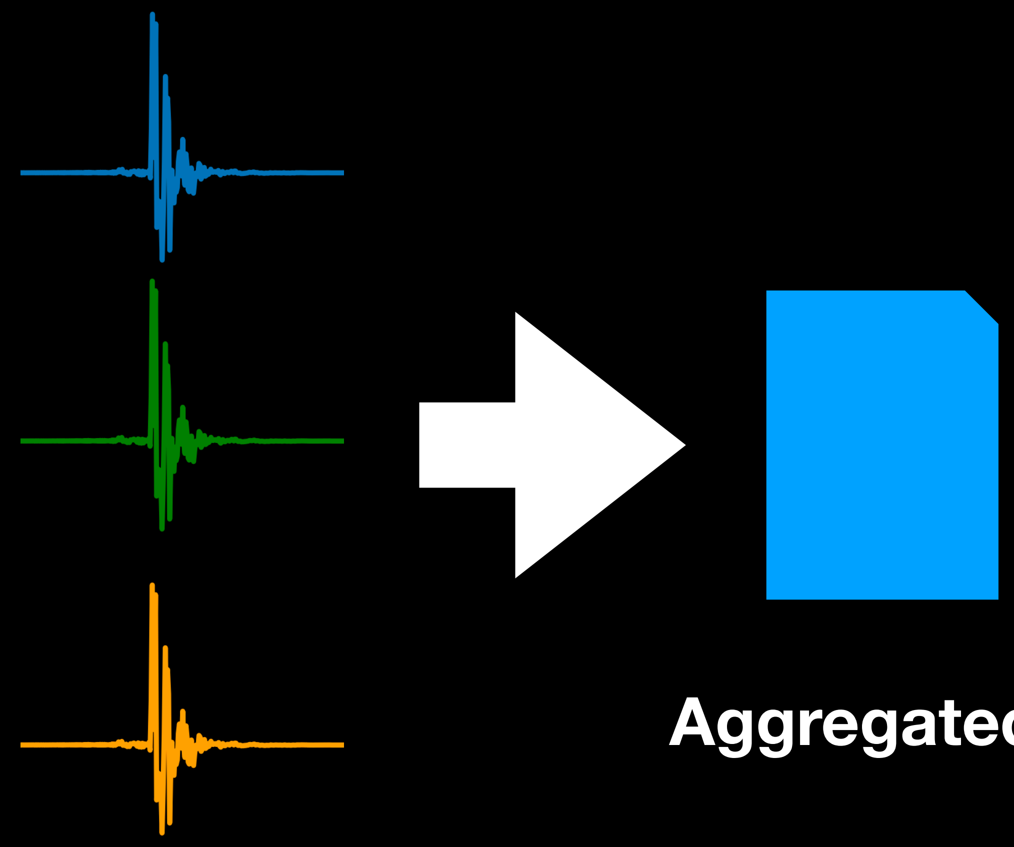
- HRIRs/HRTFs can be stored in a number of ways
- One way is to store the impulse data in an uncompressed audio file
- What if you wanted to store many different HRIRs from a single measurement session?

Storing a Single HRIR



HRIR/BRIR **.wav**

Storing Multiple HRIRs



Aggregated File

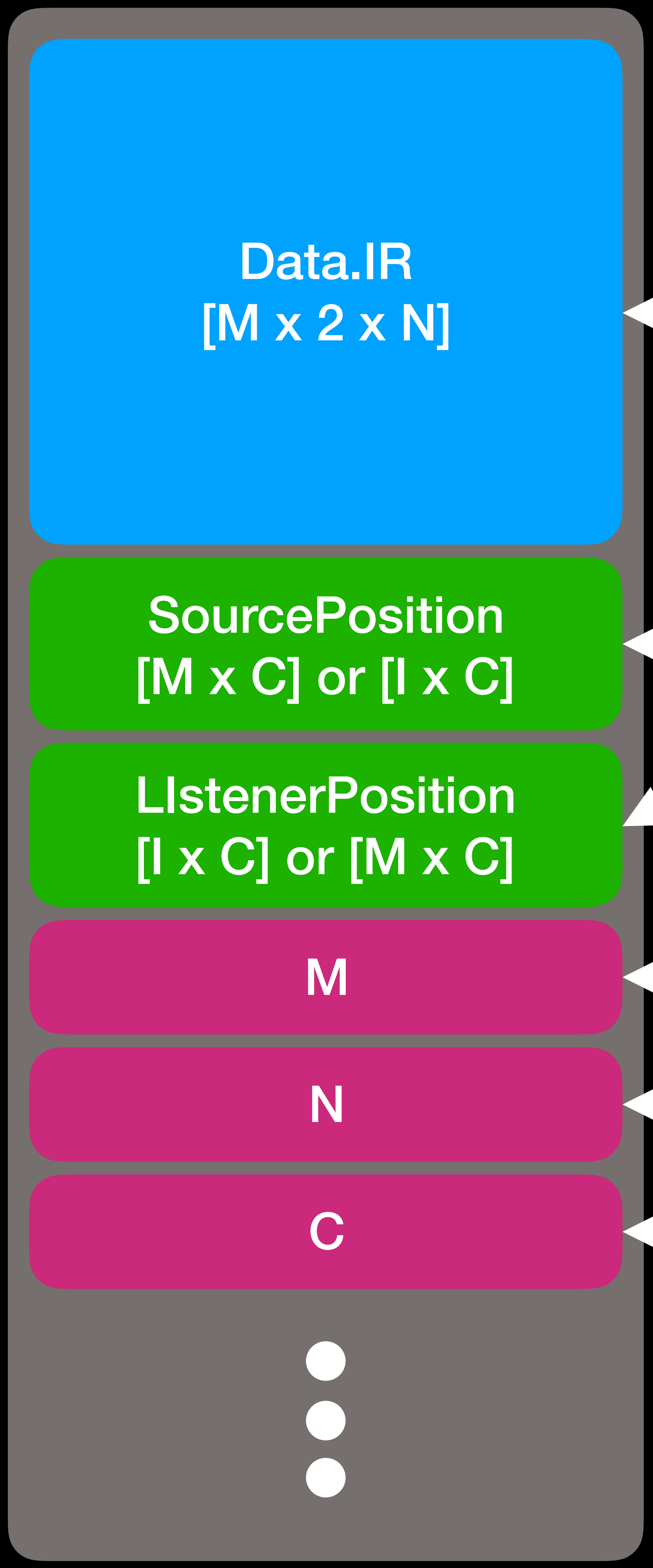
SOFA File Format

Overview

- Spatially Oriented Format for Acoustics
- AES69-2015
- File format to store HRIRs and measurement setup information
- Based on netCDF (which is based on HDF5)

SOFA File Format

SOFA File Contents



HRIRs
Stored as a packed 1D array of size M x 2 x N

Audio Source/Listener Locations
Coordinates of audio sources or listener during the measurement process
Stored as a packed 1D array of size M x C or I x C

Number of Measurements

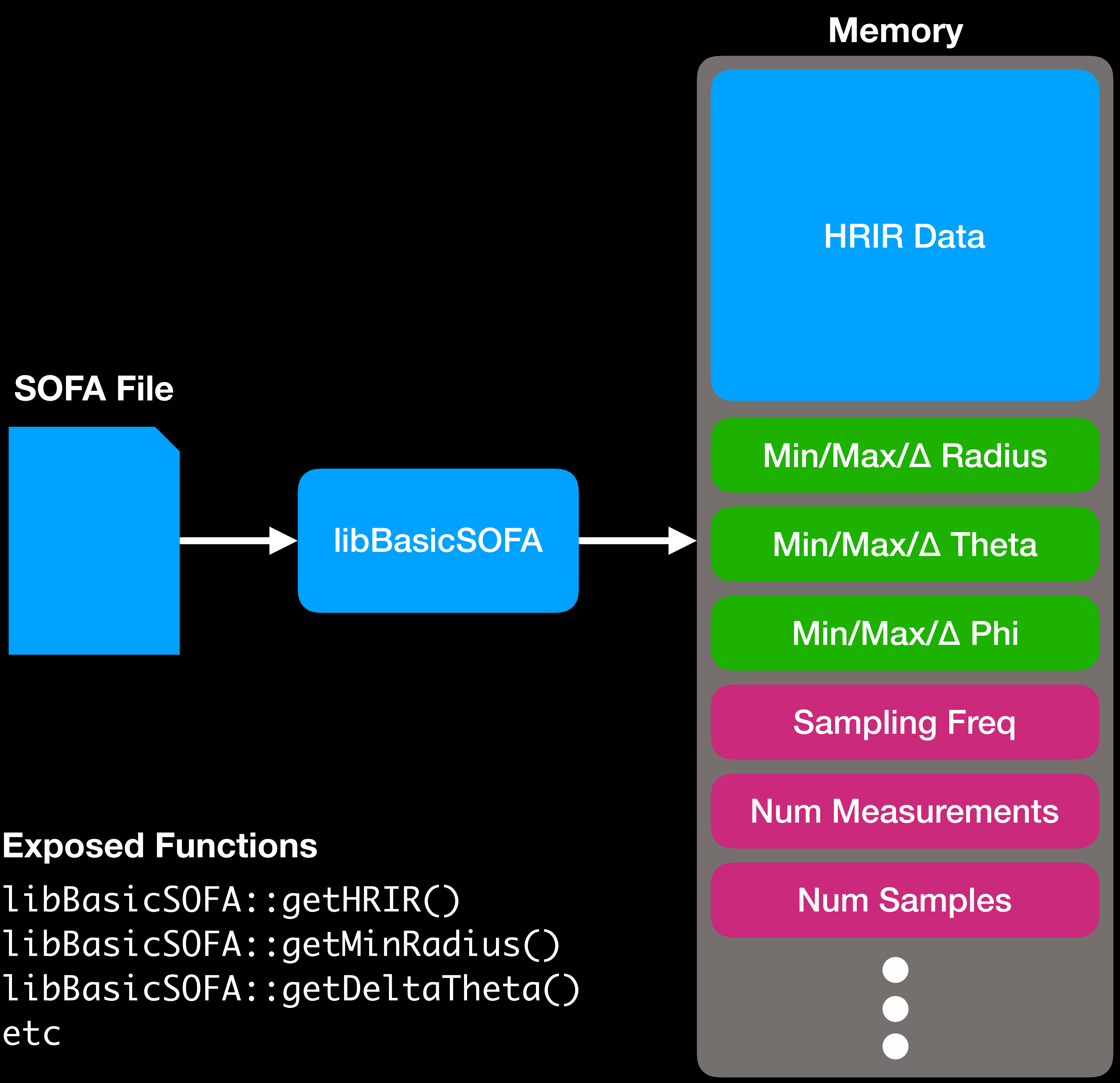
HRIR Length

Coordinate Triplet (*always 3*)

libBasicSOFA

Overview

- A very bare bones library to read SOFA files
- Extract HRIRs from file and place in memory
- Extract measurement setup information

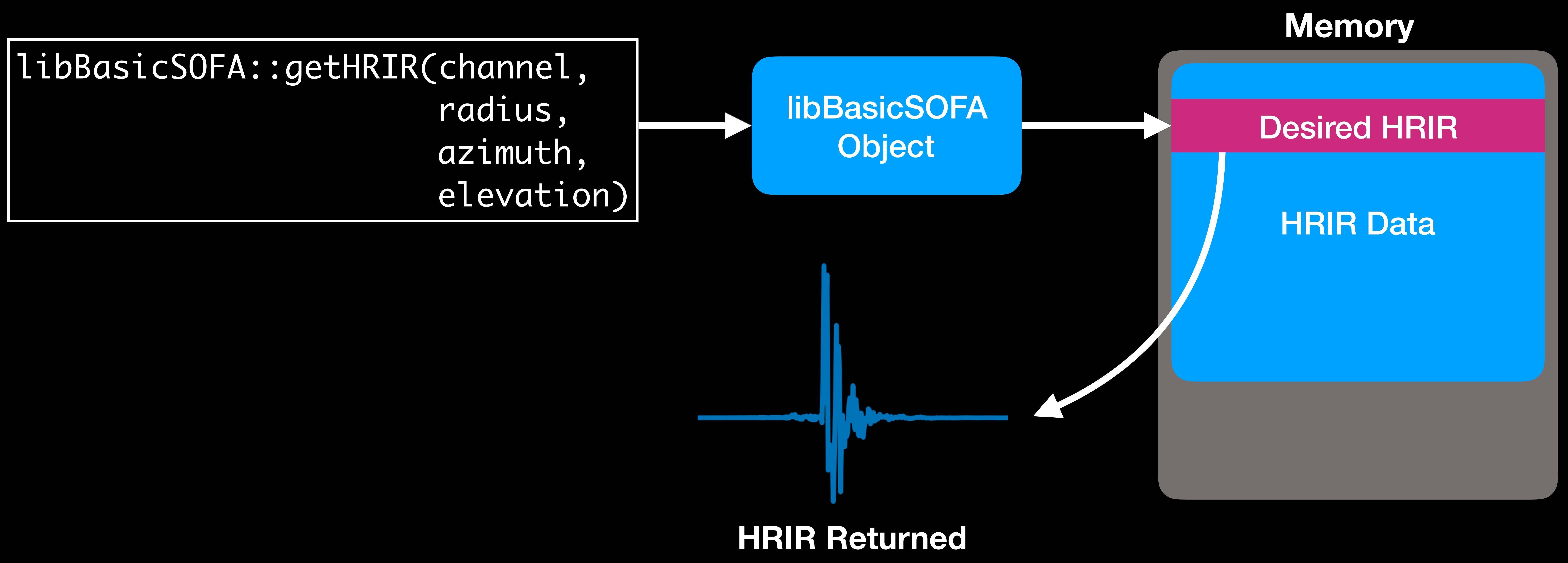


Exposed Functions

```
libBasicSOFA::getHRIR()
libBasicSOFA::getMinRadius()
libBasicSOFA::getDeltaTheta()
etc
```

libBasicSOFA

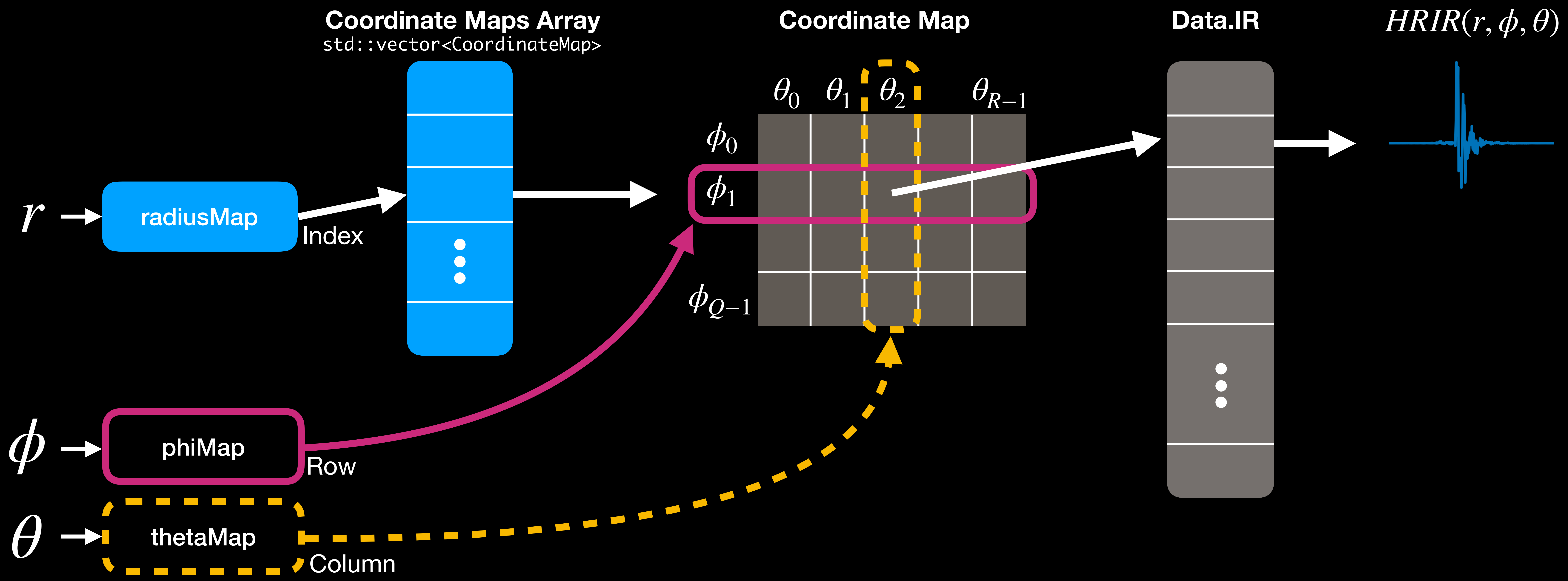
Overview



libBasicSOFA

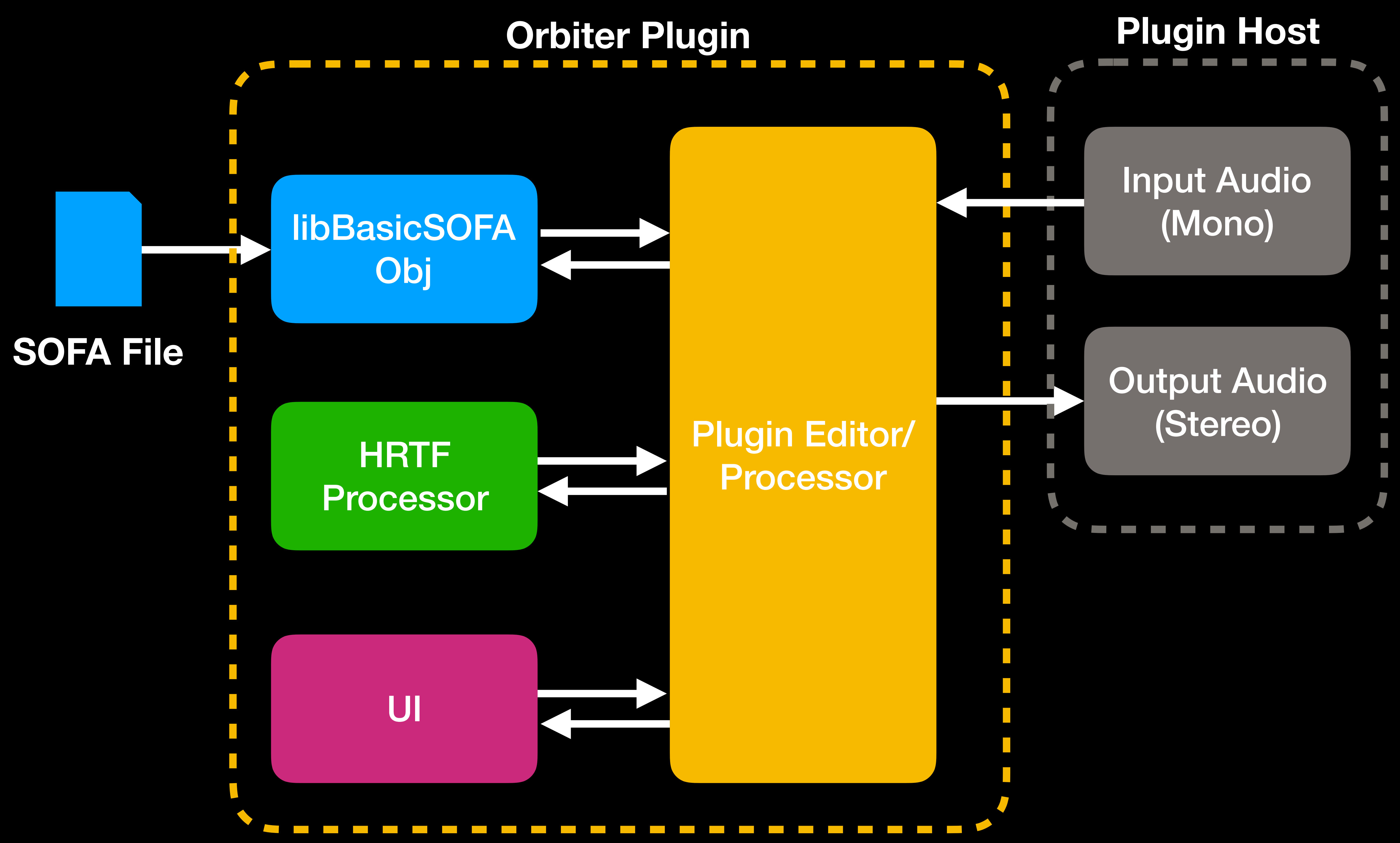
HRIR Location Mapping

- Index of an HRIR for a given elevation and azimuth is stored in a 2D array called the *Coordinate Map*
- Each **radius** has a Coordinate Map associated with it



Orbiter Architecture

Orbiter High Level Architecture

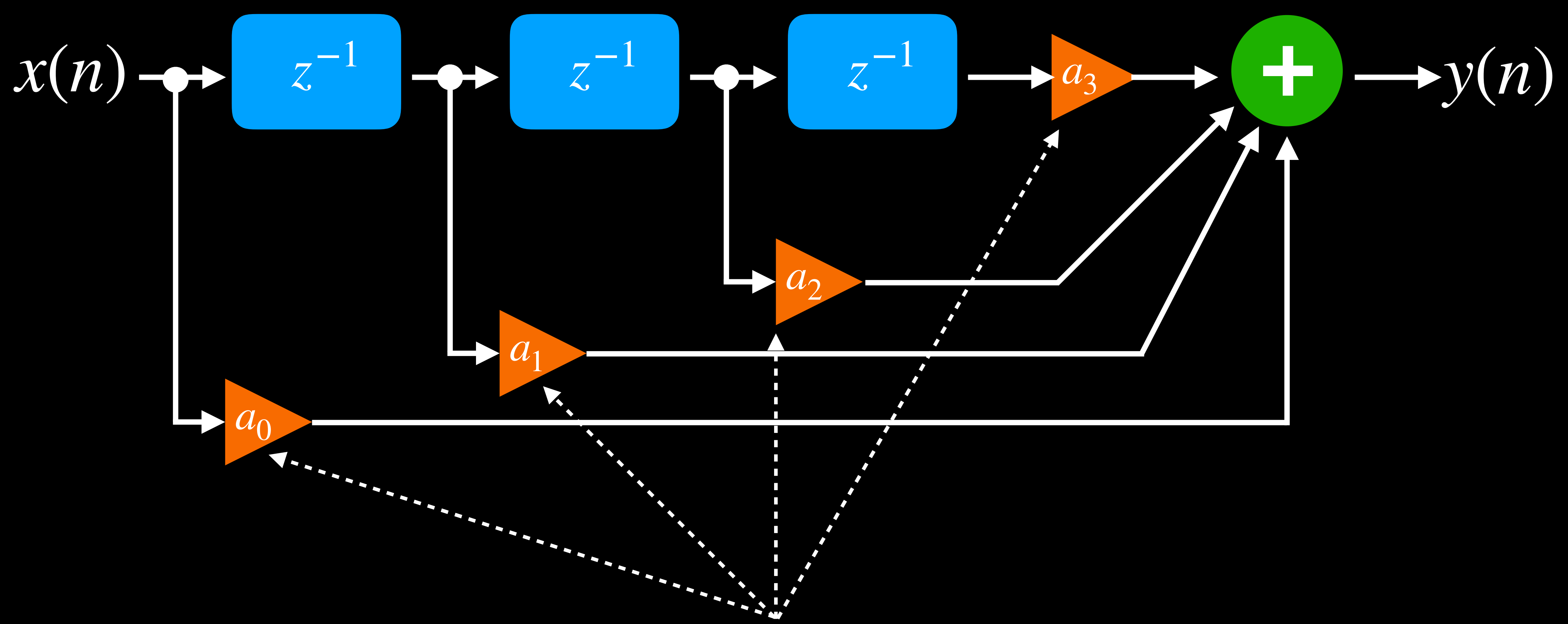


Orbiter

Applying HRTFs

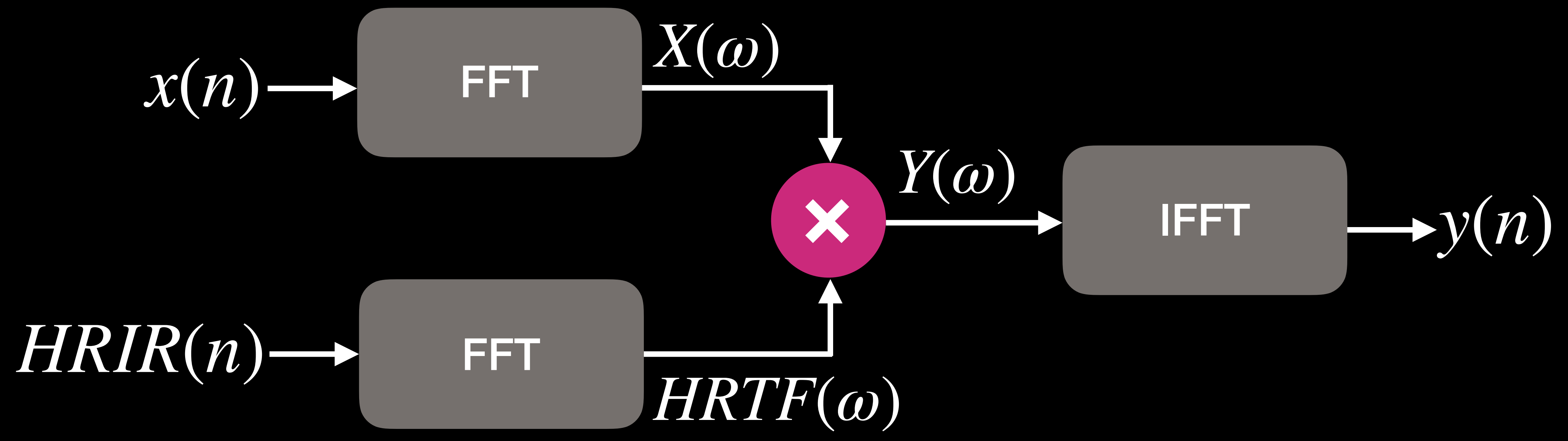
- Applying HRTFs to an audio signal is essentially applying a FIR filter
- Two ways to implement the filter

Time Domain Convolution



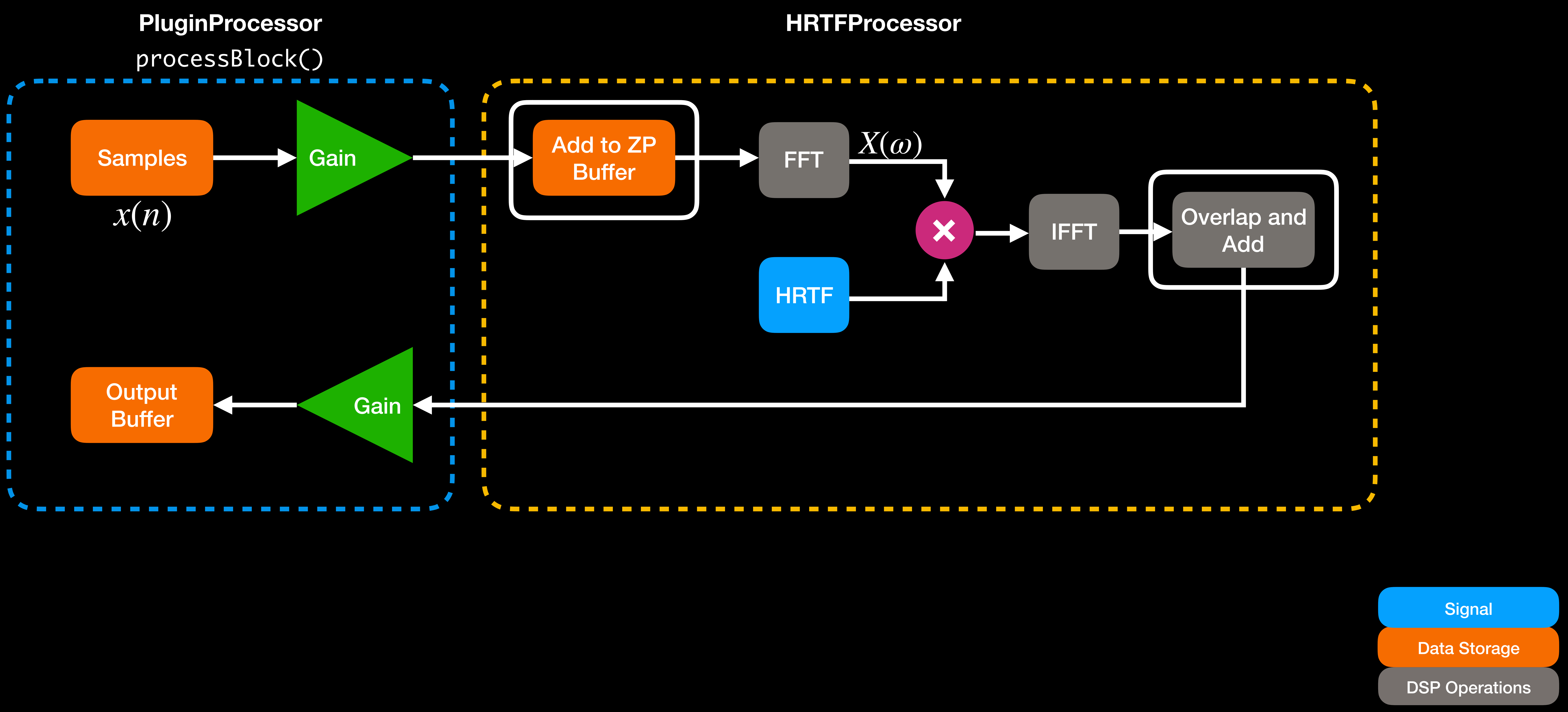
Values of HRIR
For HRIR Length of N, you will need N coefficients

Frequency Domain Convolution



Orbiter

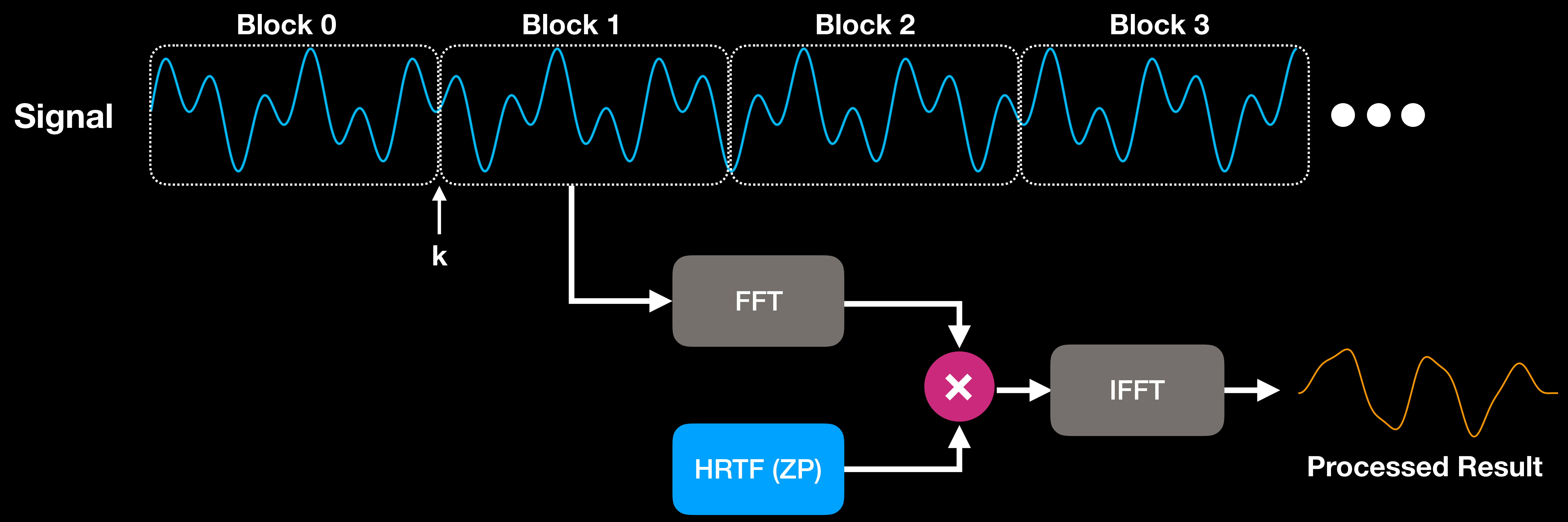
HRTFProcessor Flow



Orbiter

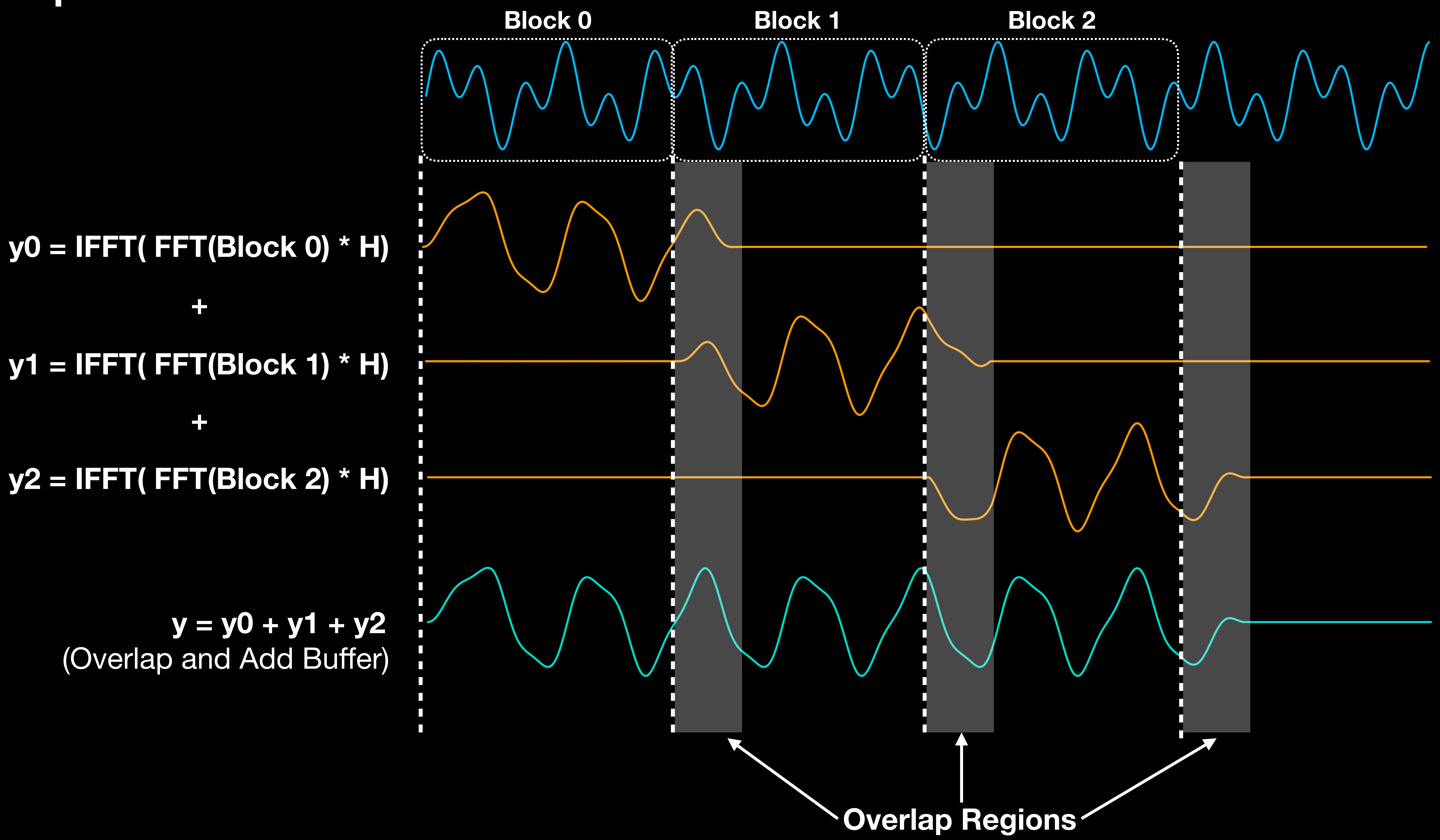
Overlap and Add

- Split a signal into N sections of size M
- Take a signal block starting at sample k and perform FFT
- Perform processing and run inverse FFT to get the time domain result
- Place processed block in an *overlap and add buffer*, shift by k samples and add



Orbiter

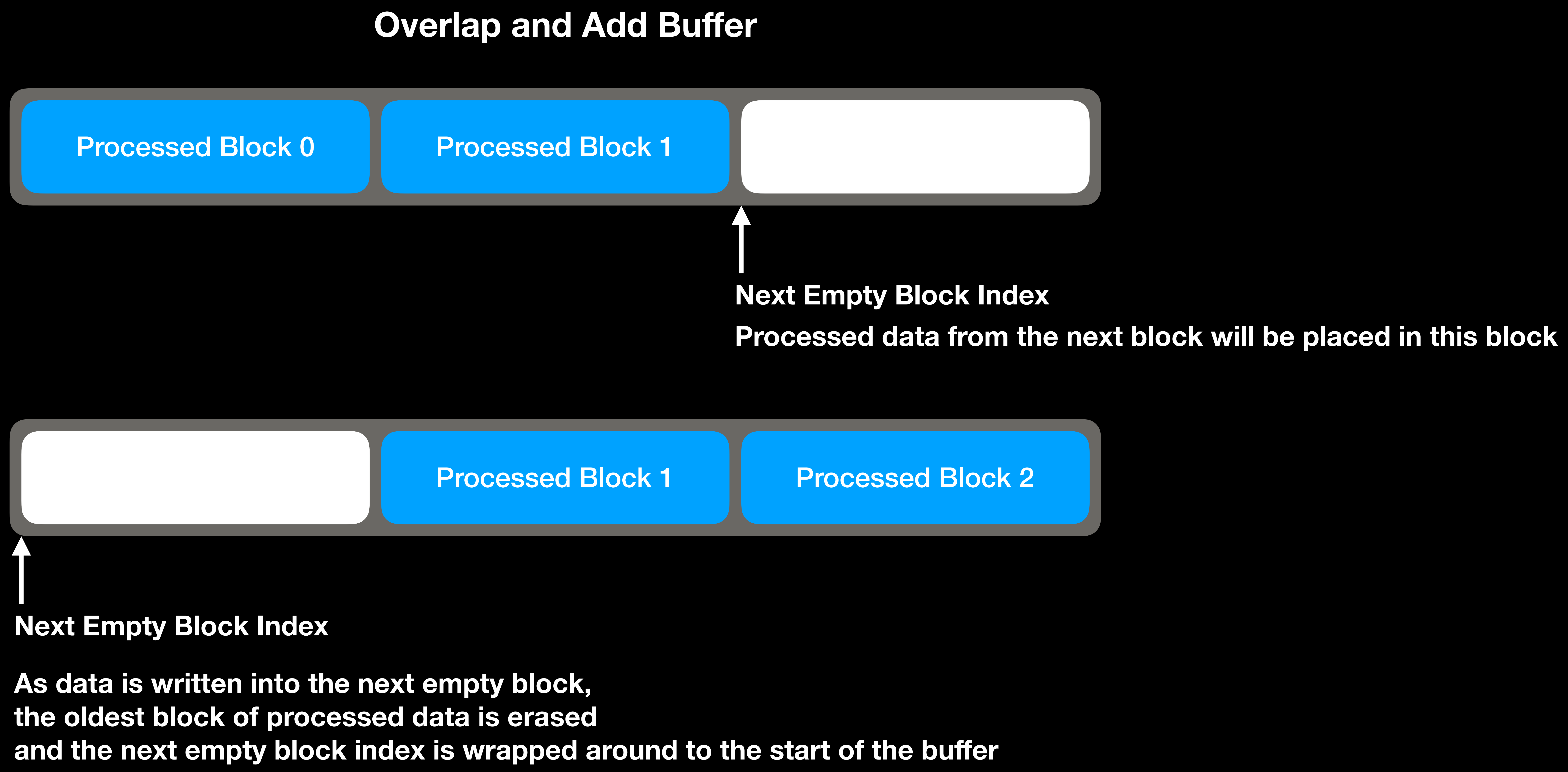
Overlap and Add



Orbiter

Implementing Overlap and Add

- Overlap and Add buffer is implemented as a *circular buffer*

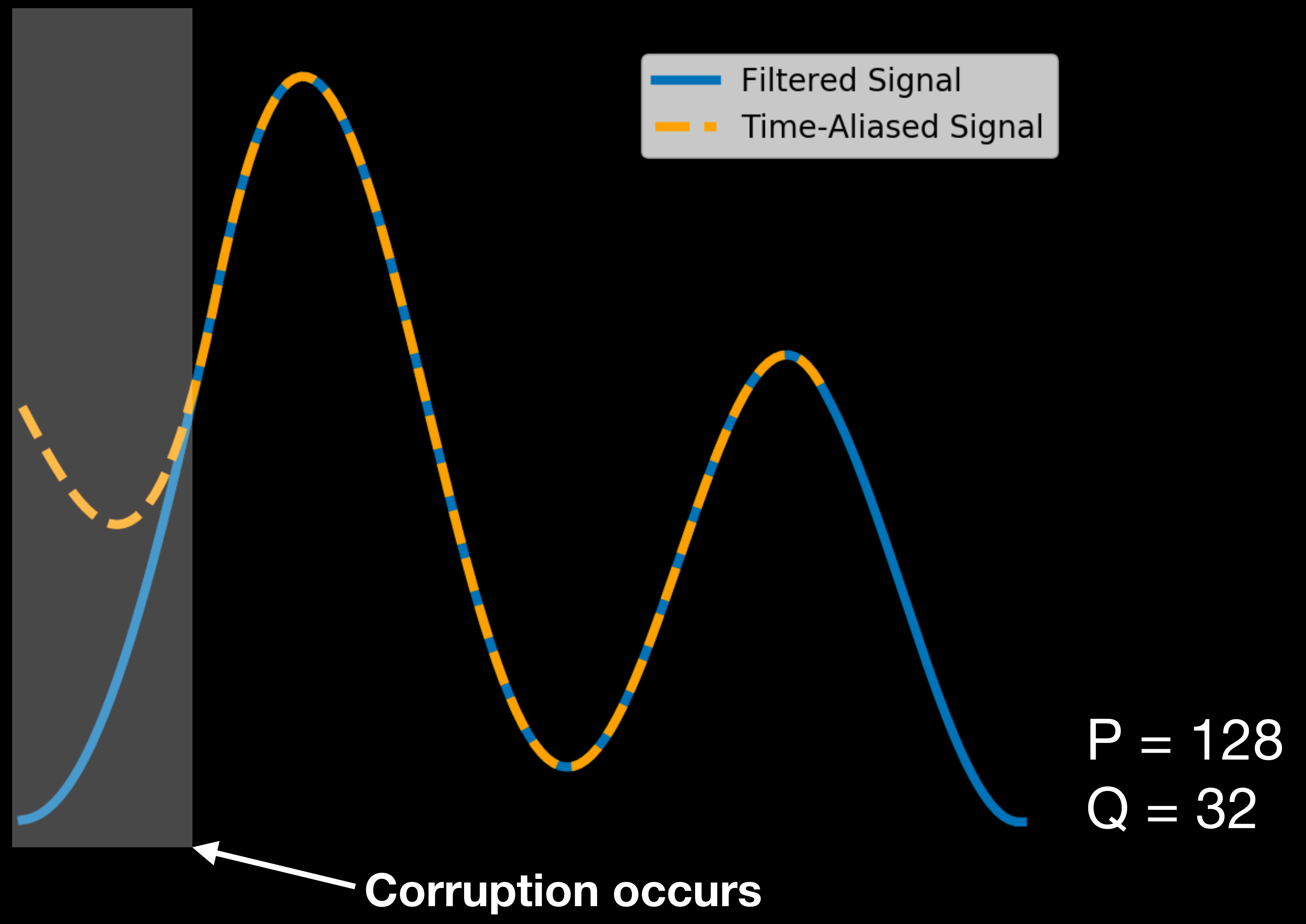


Orbiter

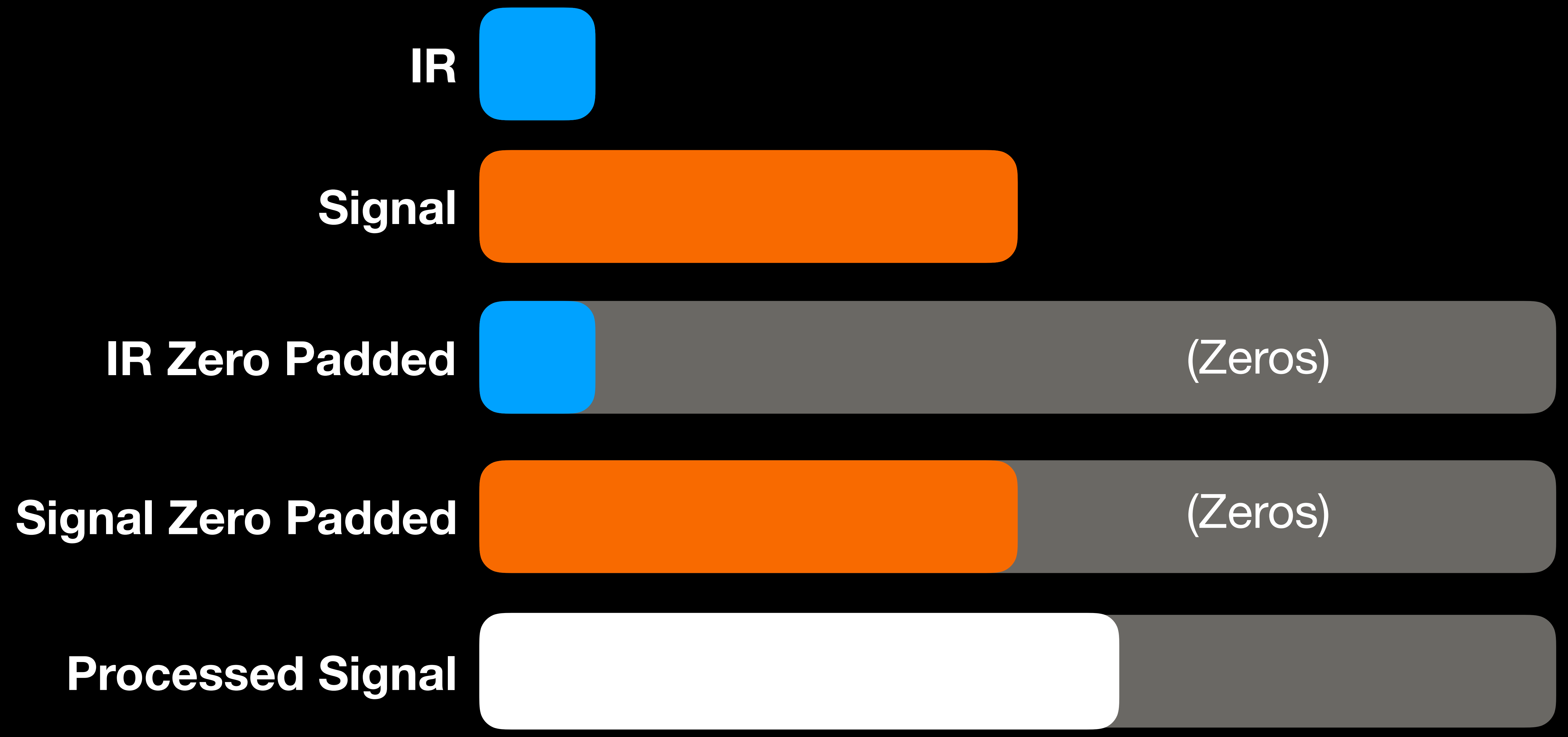
Zero Padding

- For signal length, P and impulse response length, Q
- Processed signal is length P + Q - 1
- Therefore, FFT size should be at least this length!

Applying a Simple LPF



Setting up Signals for Processing

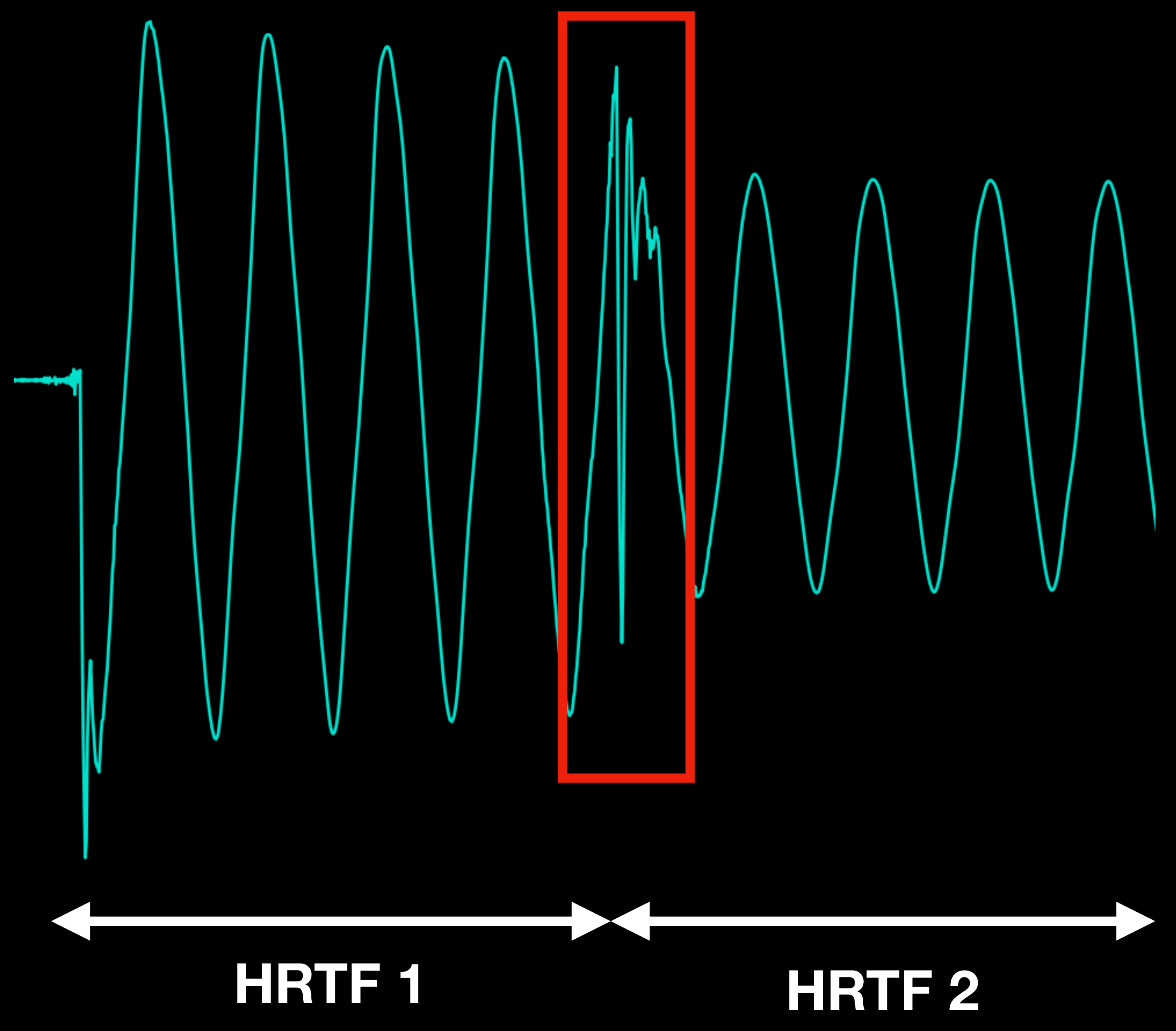


Orbiter

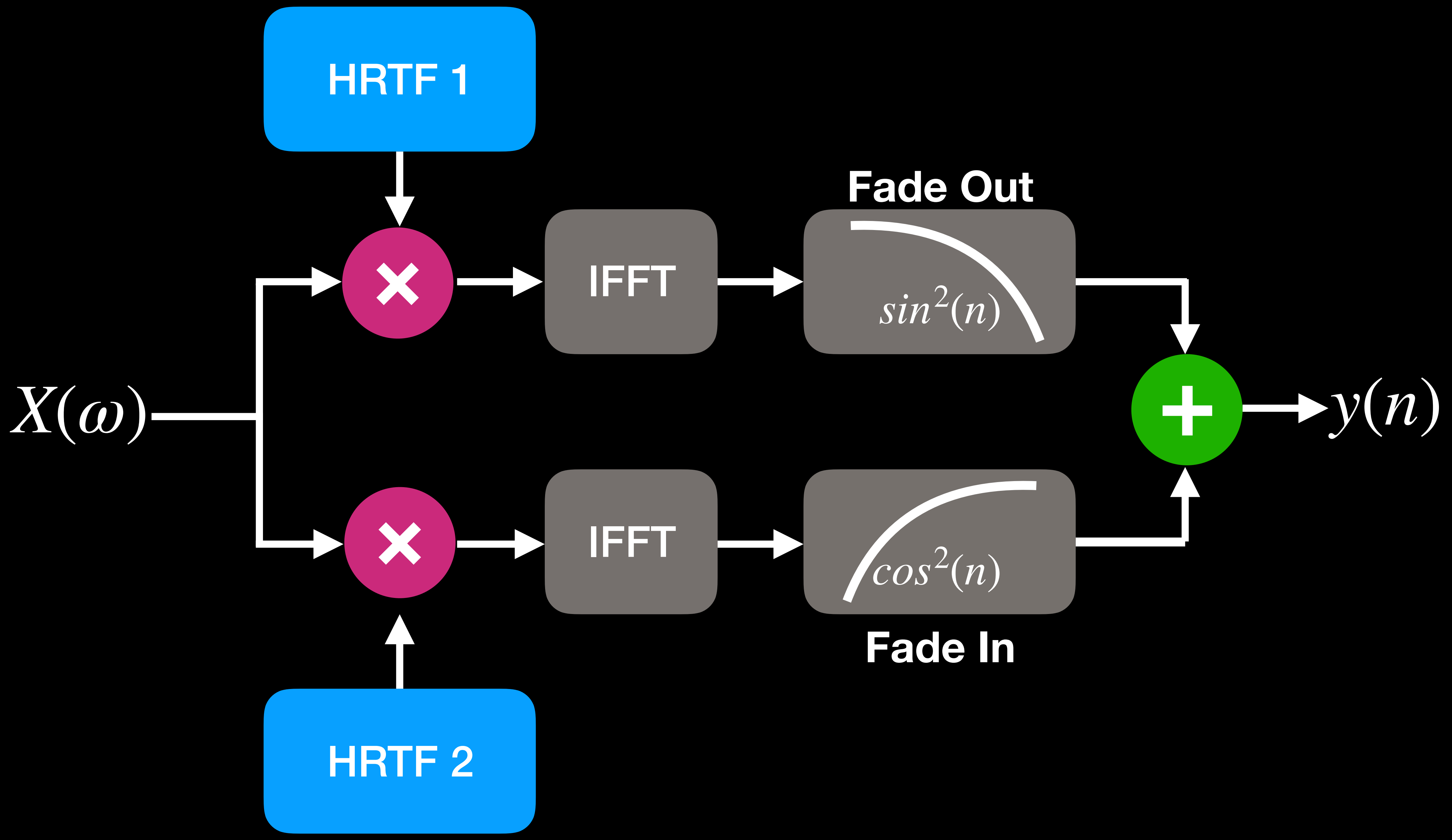
Changing HRTF

- Abruptly changing HRTF between processing blocks will create zipper noise
- Need to crossfade between the HRTF changes

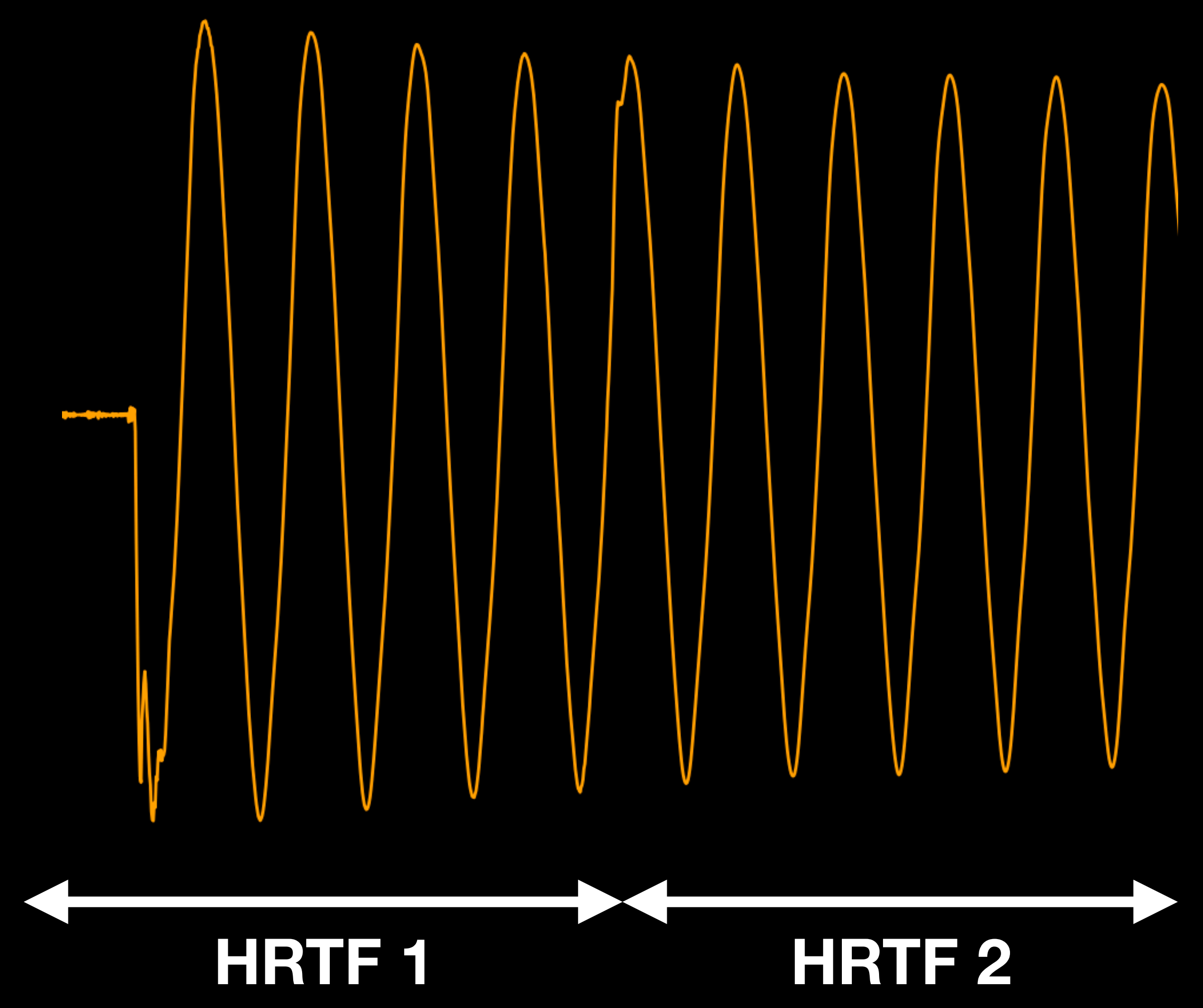
Zipper Noise from HRTF Change



Time Domain Crossfading



Crossfaded Signal

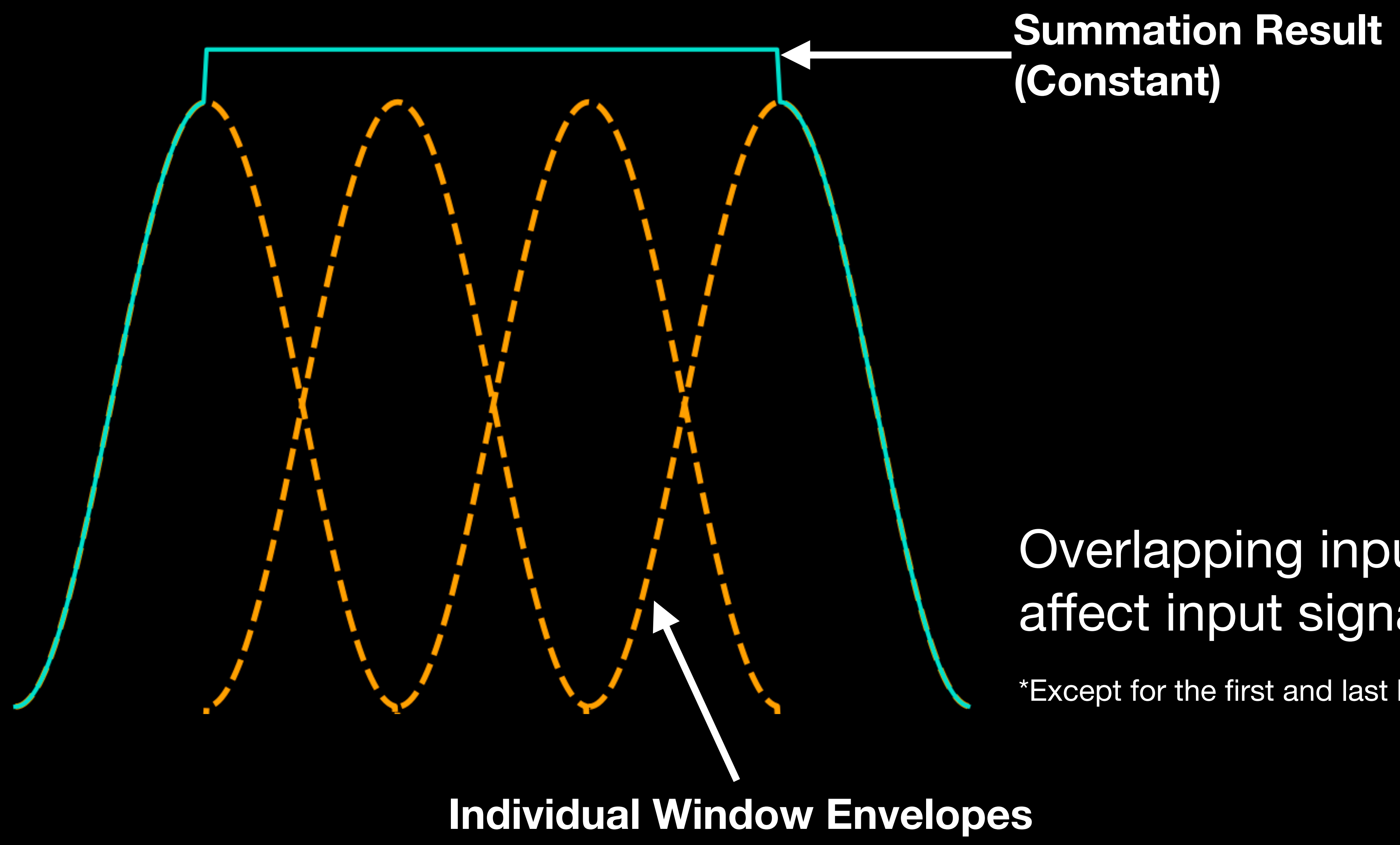


Orbiter

COLA Windowing

- Changing HRTFs means that the FIR filter is *time varying*
- To further reduce artifacts, we need to apply windowing to the input audio
- Need to overlap windowed input audio samples (Constant Overlap and Add)

Overlap and Add with Hamming Windows

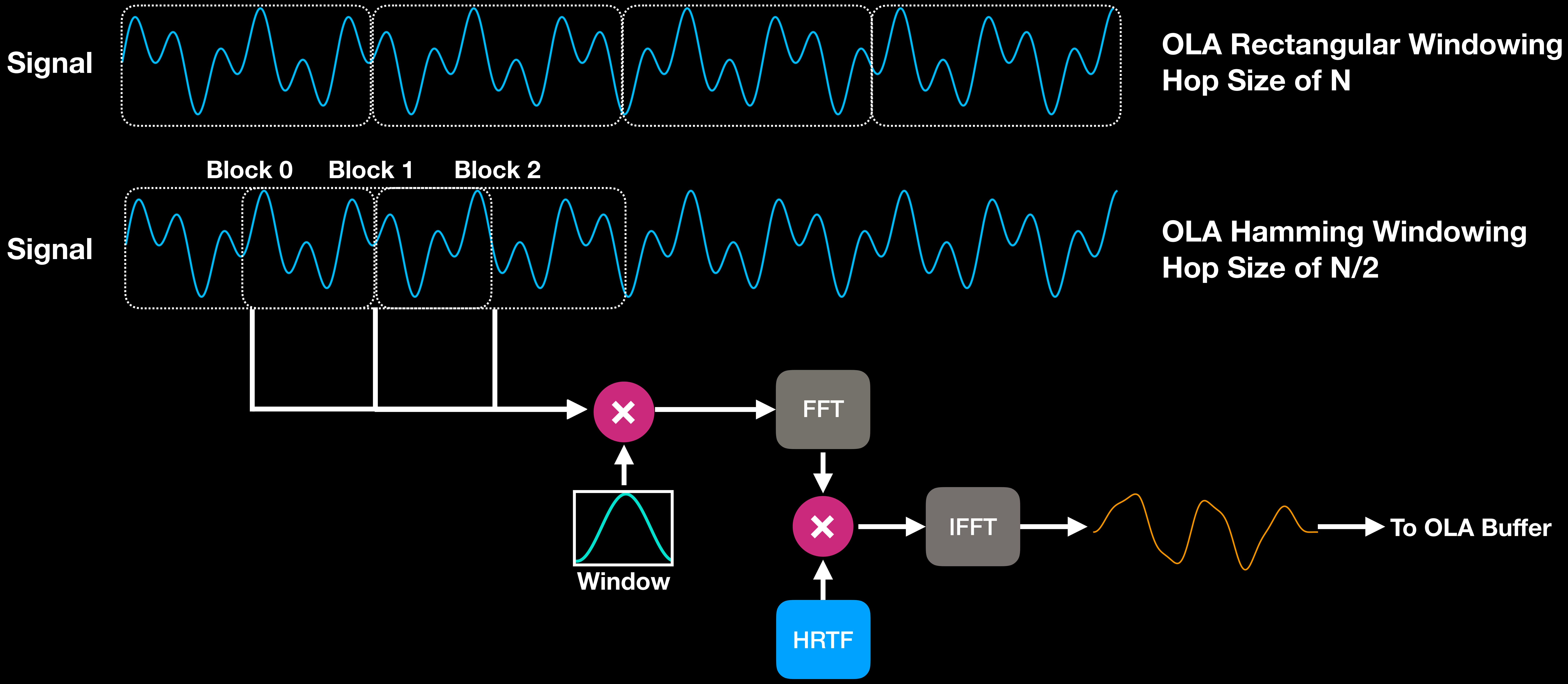


Overlapping input blocks does not
affect input signal integrity*

*Except for the first and last blocks

Orbiter

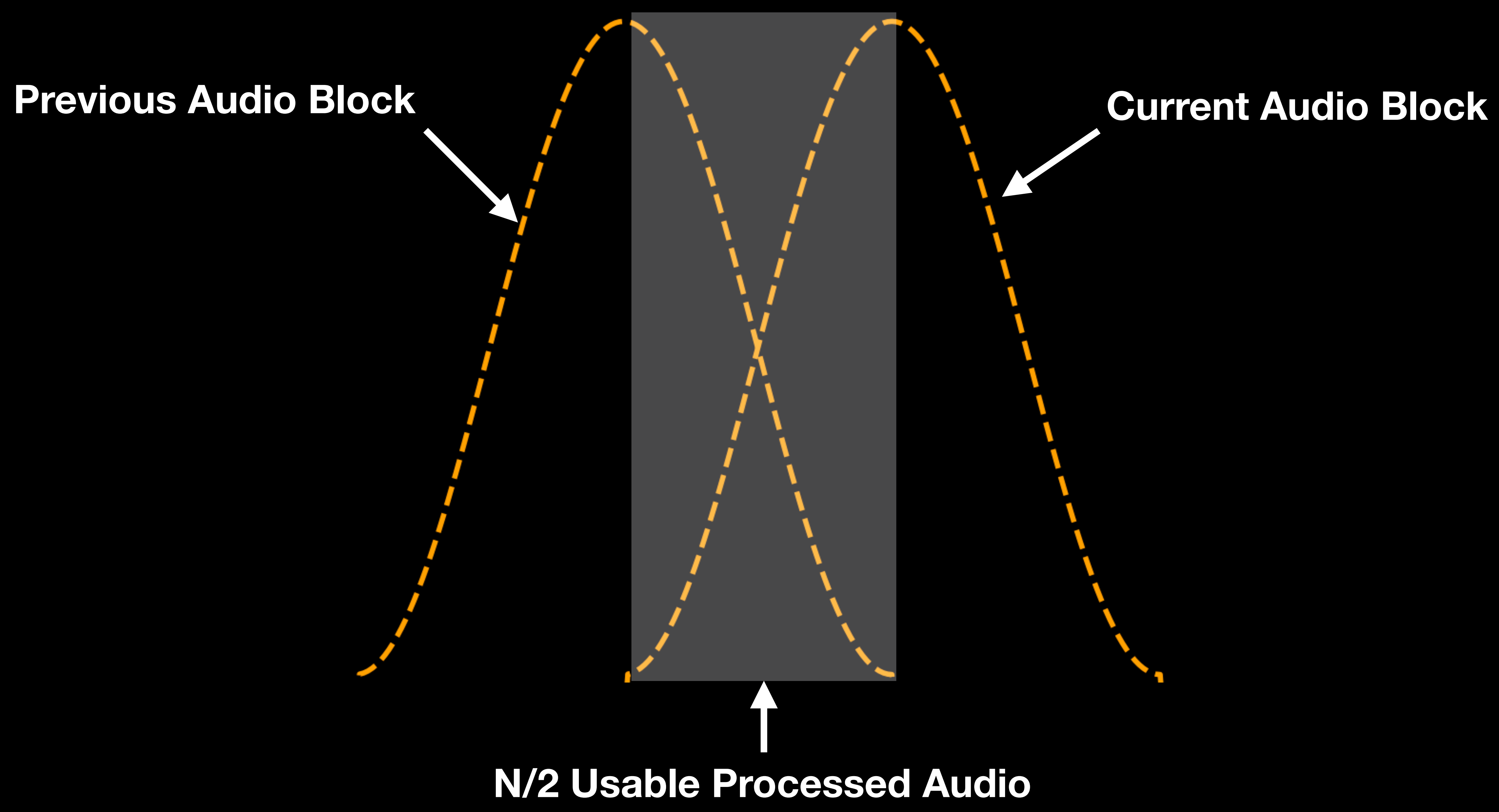
COLA Windowing



Orbiter

COLA Windowing Caveat

- Processing one audio block of length N only outputs N/2* usable output samples
- AudioProcessor::processBlock() expects N output samples
- Need audio input of 2N samples to output N processed samples

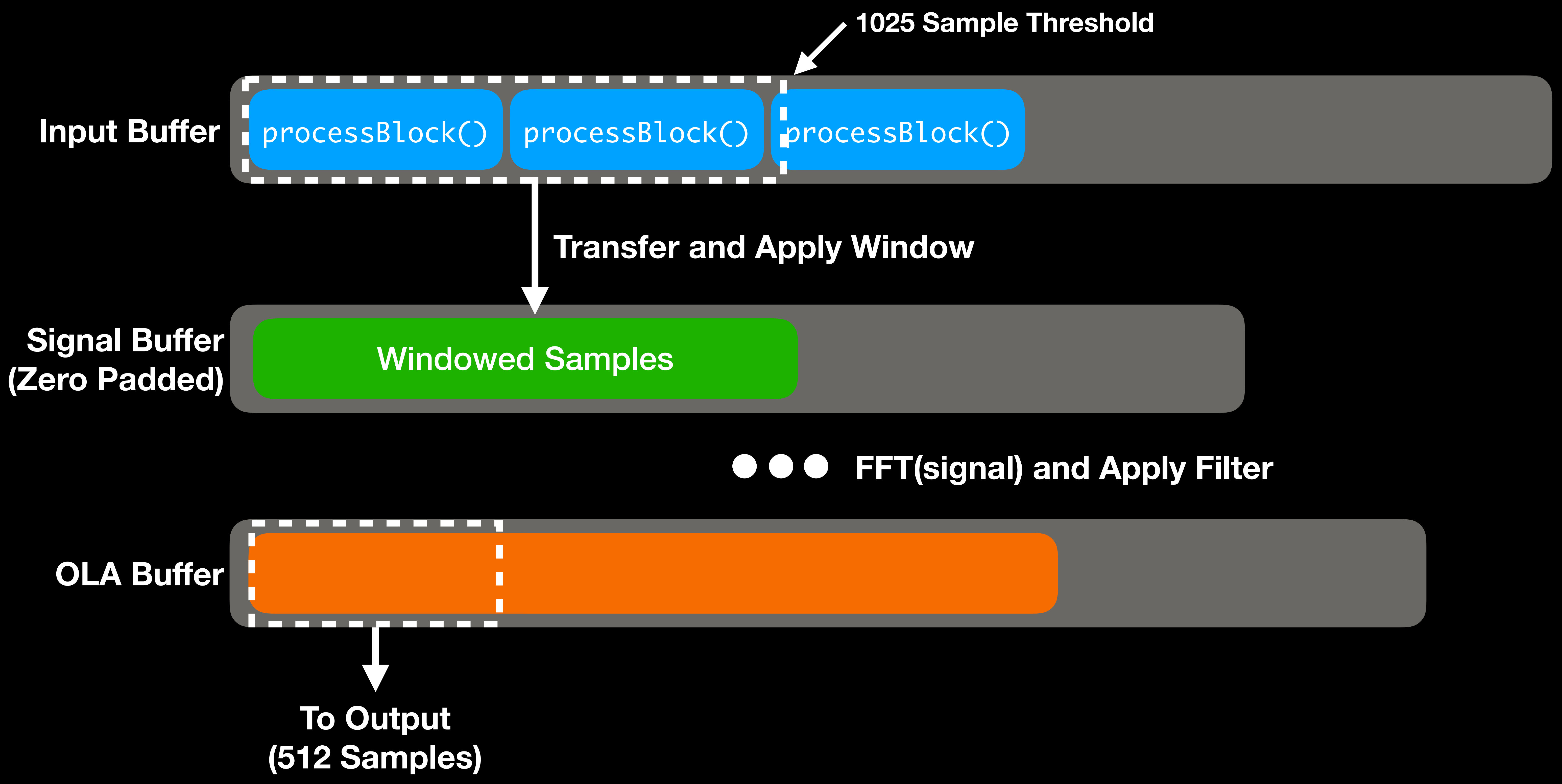


*Other COLA methods can output different number of usable samples

Orbiter

COLA Example

- AudioProcessor::processBlock() gives and requests 512 samples (N=512)
- Input block length needs to be 2N+1 = 1025 samples

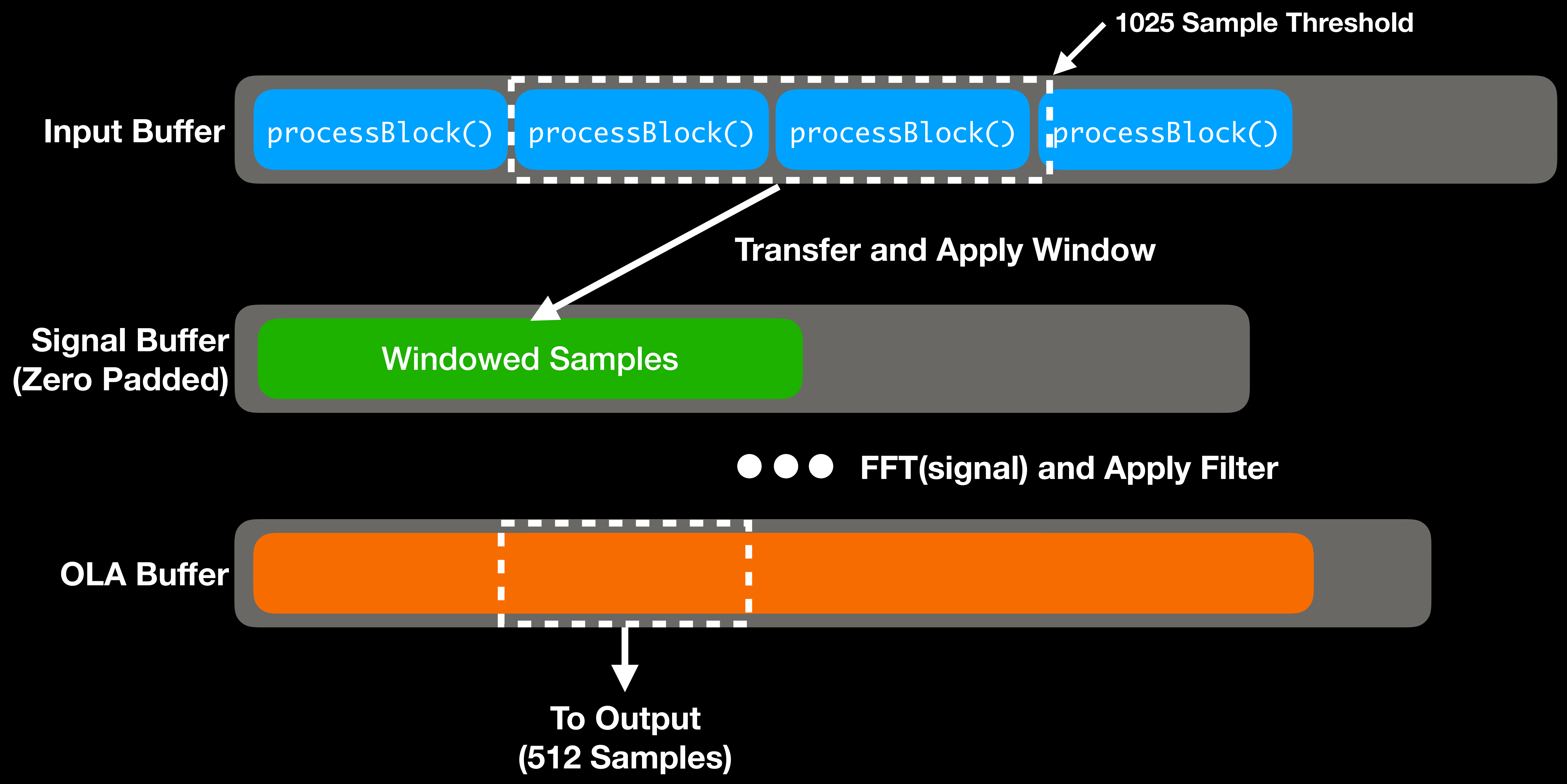


Note that there is an *initial buffering* phase that occurs when the plugin first begins operation

Orbiter

COLA Example (Continued)

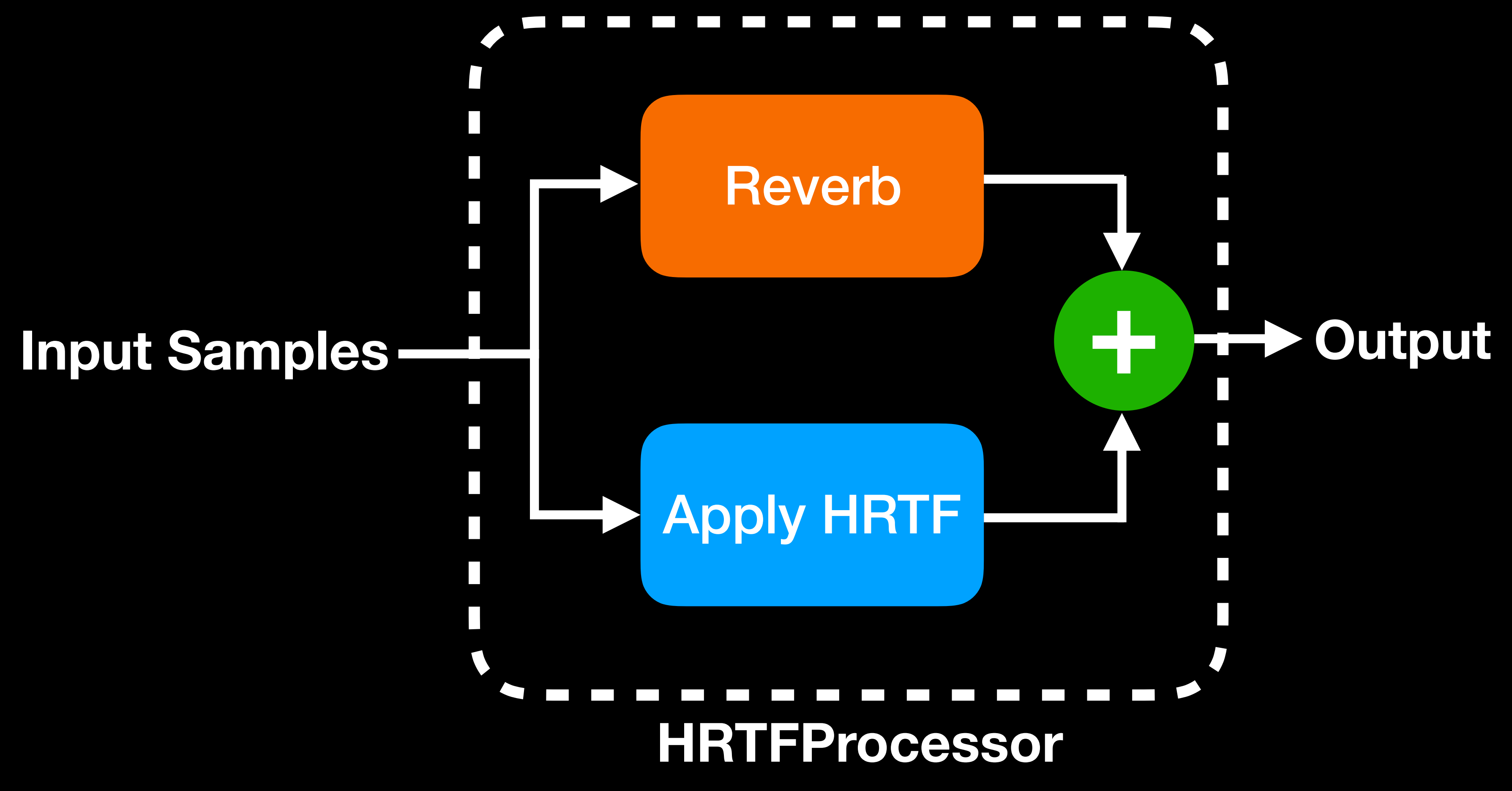
- Another block of inputs is added into the input buffer
- New batch of input data is ready to be processed



Orbiter

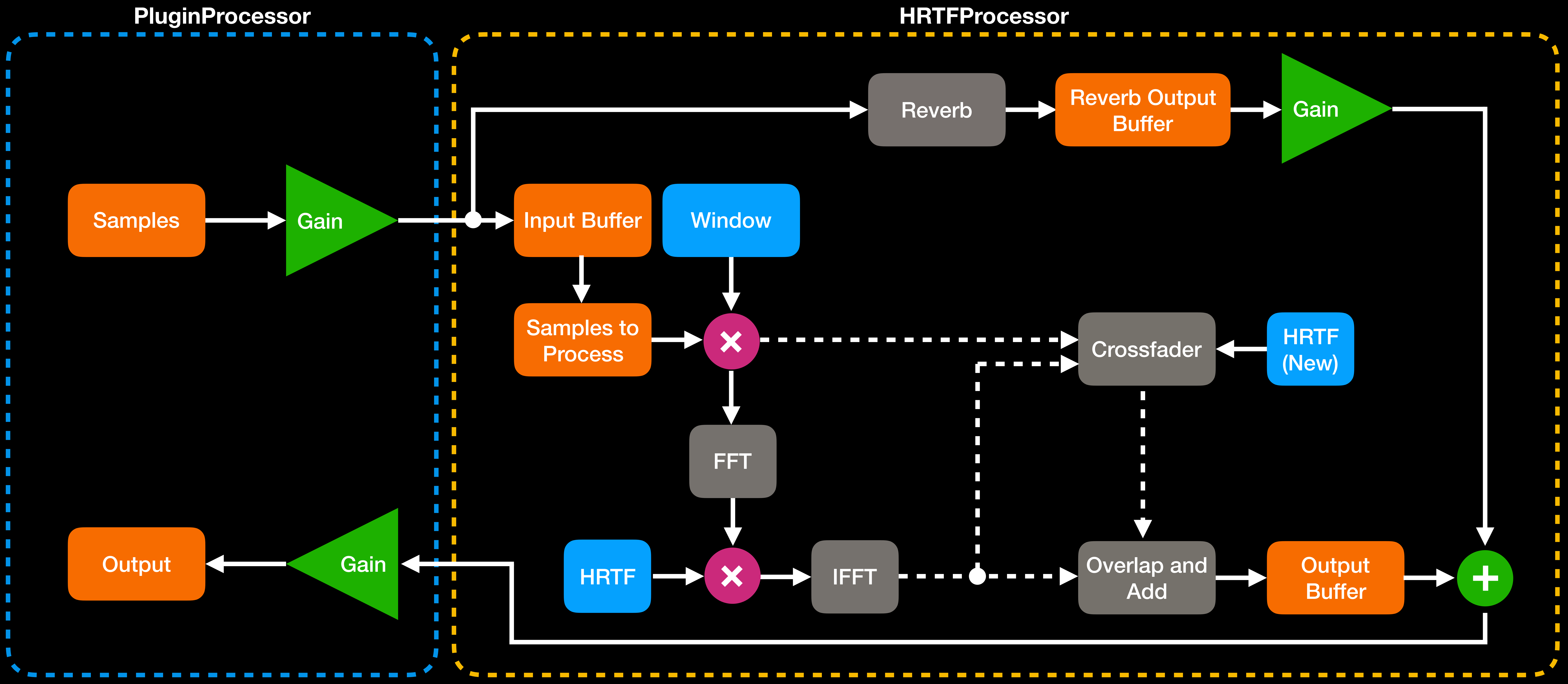
Adding Reverb

- Want a list of early/late reverb cues and their angle of approach
    - Using this list, we can apply the appropriate gain and HRIR
- Or use a BRIR (provided the impulse response is not overly long)
- For simplicity, Orbiter uses the juce::reverb module (Freeverb)



Orbiter

Final Signal Flow



Orbiter

Implementation/SOFA Wrapper

- HRIRs (BasicSOFA Object) and HRTF Processors in a wrapper class, **ReferenceCountedSOFA**
- Facilitates SOFA file changes during plugin runtime

```
class ReferenceCountedSOFA : public juce::ReferenceCountedObject
    {
    public:
        typedef juce::ReferenceCountedObjectPtr<ReferenceCountedSOFA> Ptr;

        ReferenceCountedSOFA(){}
        BasicSOFA::BasicSOFA    *getSOFA() { return &sofa; }

        BasicSOFA::BasicSOFA    sofa;
        HRTFProcessor           leftHRTFProcessor;
        HRTFProcessor           rightHRTFProcessor;

        size_t                  hrirSize;

    private:

        JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR(ReferenceCountedSOFA)
    };
```

SOFA file read by plugin stored in libBasicSOFA instance

HRTFProcessor instances for each ear

Orbiter

Implementation/processBlock

```
void OrbiterAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer, juce::MidiBuffer& midiMessages)
{
    …

    if (sofaFileLoaded)
    {
        ReferenceCountedSOFA::Ptr retainedSofa(currentSOFA);

        for (int channel = 0; channel < 1; ++channel)
        {
            auto *channelData = buffer.getWritePointer (channel);

            …

            retainedSofa->leftHRTFProcessor.addSamples(channelData, buffer.getNumSamples());
            retainedSofa->rightHRTFProcessor.addSamples(channelData, buffer.getNumSamples());

            auto left = retainedSofa->leftHRTFProcessor.getOutput(buffer.getNumSamples());
            auto right = retainedSofa->rightHRTFProcessor.getOutput(buffer.getNumSamples());

            if (left.size() != 0 || right.size() != 0)
            {
                auto *outLeft = buffer.getWritePointer(0);
                auto *outRight = buffer.getWritePointer(1);

                for (auto i = 0; i < buffer.getNumSamples(); ++i)
                {
                    outLeft[i] = left[i];
                    outRight[i] = right[i];
                }
            …
```

Get active ReferenceCountedSOFA Instance

Add samples into the HRTFProcessor Input buffer

Get processed binaural audio

Write processed binaural audio to the AudioBuffer

Orbiter

Implementation/HRTFProcessor::addSamples

```
bool HRTFProcessor::addSamples(float *samples, size_t numSamples)
{
    …
    for (auto i = 0; i < numSamples; ++i)
    {
        //  Add samples into the input buffer and reverb buffer
        inputBuffer[inputSampleAddIndex] = samples[i];
        reverbBuffer[reverbBufferAddIndex] = samples[i];

        inputSampleAddIndex = (inputSampleAddIndex + 1) % inputBuffer.size();
        reverbBufferAddIndex = (reverbBufferAddIndex + 1) % reverbBuffer.size();

        numSamplesAdded++;
    }
```

Add samples into the HRTFProcessor Input and reverb buffer

```
    //  Execute when we have added enough samples for processing
    if (numSamplesAdded >= audioBlockSize)
    {
        numSamplesAdded -= hopSize;
        std::vector<float> x(audioBlockSize);
        auto blockStart = inputBlockStart;

        for (auto i = 0; i < audioBlockSize; ++i)
        {
            x[i] = inputBuffer[blockStart] * window[i];
            blockStart = (blockStart + 1) % inputBuffer.size();
        }

        inputBlockStart = (inputBlockStart + hopSize) % inputBuffer.size();
        calculateOutput(x);
    }
    …
```

When there are a sufficient number of input samples, transfer to signal buffer, window samples and apply HRTFs

Orbiter

Implementation/HRTFProcessor::calculateOutput

```
const float* HRTFProcessor::calculateOutput(const std::vector<float> &x)
{
    …
    std::fill(olaBuffer.begin() + olaWriteIndex, olaBuffer.begin() + olaWriteIndex + hopSize, 0.0);
    olaWriteIndex = (olaWriteIndex + hopSize) % olaBuffer.size();

    std::fill(xBuffer.begin(), xBuffer.end(), std::complex<float>(0.0, 0.0));
    for (auto i = 0; i < x.size(); ++i)
        xBuffer.at(i) = std::complex<float>(x.at(i), 0.0);

    fftEngine->perform(xBuffer.data(), xBuffer.data(), false);

    for (auto i = 0; i < zeroPaddedBufferSize; ++i)
        xBuffer.at(i) = xBuffer.at(i) * activeHRTF.at(i);

    fftEngine->perform(xBuffer.data(), xBuffer.data(), true);
```

Remove old OLA audio data

Take FFT of input signal

Apply HRTF and get time domain output

Orbiter

Implementation/HRTFProcessor::calculateOutput

```
if (hrirChanged)
{
    juce::SpinLock::ScopedTryLockType hrirChangingScopedLock(hrirChangingLock);
    if (hrirChangingScopedLock.isLocked())
    {
        hrirChanged = false;
        crossfadeWithNewHRTF(x);

        std::copy(auxHRTFBuffer.begin(), auxHRTFBuffer.end(), activeHRTF.begin());
    }
}
```

Apply crossfading if needed

```
if(!overlapAndAdd())
    return nullptr;
```

Overlap and add

```
//  Copy outputtable audio data to the output buffer
std::copy(olaBuffer.begin() + olaWriteIndex, olaBuffer.begin() + olaWriteIndex + hopSize, outputBuffer.begin() + outputSampleEnd);
outputSampleEnd = (outputSampleEnd + hopSize) % outputBuffer.size();

numOutputSamplesAvailable += hopSize;
```

…

Transfer usable processed data to output buffer (which is extracted via HRTFProcessor::getOutput()

Orbiter

Implementation/HRTFProcessor::getOutput

```
std::vector<float> HRTFProcessor::getOutput(size_t numSamples)
{
    std::vector<float> out(numSamples);
    if (numSamples > numOutputSamplesAvailable)
        return std::vector<float>(0);

    //  Get reverberated input signal
    reverb.processMono(reverbBuffer.data() + reverbBufferStartIndex, (int)numSamples);

    for (auto i = 0; i < numSamples; ++i)
    {
        out[i] = outputBuffer[outputSampleStart] + (0.5f * reverbBuffer[reverbBufferStartIndex]);
        outputSampleStart = (outputSampleStart + 1) % outputBuffer.size();
        reverbBufferStartIndex = (reverbBufferStartIndex + 1) % reverbBuffer.size();
    }

    numOutputSamplesAvailable -= numSamples;

    return out;
}
```

Apply reverb to (non-binaural) dry input signal

Apply binaural and reverberated signal

Orbiter

Future Improvements

- Add interpolation between HRTFs (smoother transitions)
- Better reverberation model
- More stable SOFA file support
- Headphone compensation
- Use compressed HRTF data files (SOFA files are huge!)

Questions? Comments?

E-mail me!
alee@meoworkshop.org

Twitter
@superkittens

Orbiter and libBasicSOFA Code
https://github.com/superkittens/Orbiter
https://github.com/superkittens/libBasicSOFA

Thank you!