Module IN3031 / INM378 Digital Signal Processing and Audio Programming

Johan Pauwels
johan.pauwels@city.ac.uk

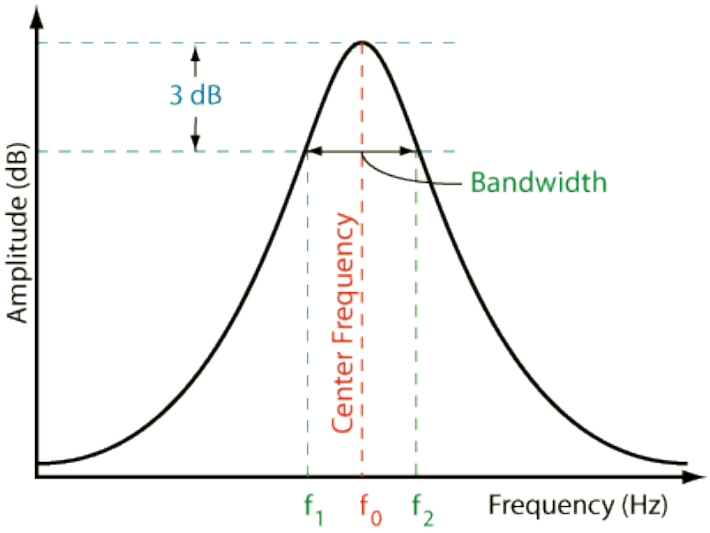
based on slides by Tillman Weyde

3D Audio
Real-time DSP
FMOD Custom DSPs
Game Audio Workflow
Event Model

Spatial Sound and Hearing

Sound and Hearing in Space

- Speed of Sound (v_s) approx. 340 m/s
- Wavelength (λ)
  - wavelength = speed * period = speed / frequency
- Sound intensity decreases with distance (see next slide), called *Distance Roll-Off* in programming
- Air absorbs energy over distance, acting as a low-pass filter
- In games, a band-pass filter is often used to make sounds seem more distant



Directional Hearing and Localisation

- Directional hearing is based mostly on binaural hearing
  - Interaural Intensity Differences (IID)
  - Interaural Time Differences (ITD)
  - IID and ITD vary over frequencies.
- IID and ITD give only information on left-right
  - front-back and high-low are detected through head-related transfer functions (head shape, pinna)
- Room reflection
  - absorbing and reflecting objects give clues for source location

Intensity and Distance

- Intensity is power (energy per time) per area, measured in Watts/Meter^2
- 0 dB Sound Pressure Level defined as 10^{-12} W/m^2 (~threshold of hearing)
- Intensity decreases as the square of distance
- Power increases as the square of amplitude
- Example: 60dB SPL at 1m means 40dB SPL at 10m (10-fold distance -> 100-fold decrease -> -20dB)

Doppler Effect

- Doppler Effect changes frequency for moving sources
  - $f_p = f\, v_s/(v_s - v_r)$
    with f_p: perceived frequency ,
    f: frequency, v_r: velocity relative to the listener (positive = approaching)
  - Example: car moves with 68 m/s producing a 300 Hz sound. f_p for a stationary listener in front of the car is
    300 Hz * 340/(340-68) = 300 Hz *5/4 = 375 Hz

FMOD Audio Programming and 3D Rendering

Loading and Playing a Sound

- Load a sound:

```
FMOD::Sound *sound;
result = system->createSound(filename,
    FMOD_LOOP_OFF, 0, &sound);
FmodErrorCheck(result);
```

- Create a Channel object and play the sound

```
FMOD::Channel    channel = null;
result = system->playSound(sound, NULL, false,
    &channel);
```

- `channel` now has the channel where `sound` is played

```
// set the Volume
result = channel->setVolume(0.8f);
```

Creating DSP Objects

- To create an oscillator and set its parameters:

```
FMOD::DSP *osc;

system->createDSPByType(FMOD_DSP_TYPE_OSCILLATOR, &osc);

osc->setParameterFloat(FMOD_DSP_OSCILLATOR_RATE, 440);

// 0 = sine. 1 = square. 2 = sawup.
// 3 = sawdown. 4 = triangle. 5 = noise.
osc->setParameterInt(FMOD_DSP_OSCILLATOR_TYPE, 1);
```

Using DSP Objects

- To play an oscillator:

```
system->playDSP(dsp, NULL, false, &channel);
```

- ... or to insert a DSP processor into the signal flow sequence at the DSP chain head :

```
channel->addDSP(FMOD_CHANNELCONTROL_DSP_HEAD,
 dsp);
```

3D Sound with FMOD

- Set up the FMOD systems 3D settings:
  system->set3DSettings(doppler, distFactor, distRolloff);
- Where:
  - doppler scales the intensity of the doppler effect
  - distFactor determines the length of an FMOD unit (1 means 1m)
  - distRolloff scales the distance roll-off (1 is like real world)

Transforming and Updating 3D Positions

- Transformation of OpenGL vectors to FMOD vectors:

```
FMOD_VECTOR fmodVec = FMOD_VECTOR();
fmodVec.x = glVec3.x; fmodVec.y = glVec3.y; fmodVec.z = glVec3.z;
```

- Make this ^ into a converter function for ease: e.g.

```
void ToFMODVector(glm::vec3 &glVec3, FMOD_VECTOR *fmodVec)
```

- Transform position vectors

```
ToFMODVector(camera->GetPosition(), &camPos);
```

- Update the listener position:

```
system->set3DListenerAttributes(0, &camPos, NULL, NULL, NULL);
system->update();
```

Positioning 3D Sound

- Create position and velocity vectors:

```
FMOD_VECTOR pos1 = FMOD_VECTOR();
pos1.x = -10f; pos1.y = 0f; pos1.z = 0f;

FMOD_VECTOR vel1 = new FMOD_VECTOR();
vel1.x = 0f; vel1.y =  0f; vel1.z = 0f;
```

- And set channel attributes:

```
channel->setMode(FMOD_3D);
channel->set3DAttributes(&pos1, &vel1);
```

3D Sound Playback

- Create a sound:

```
system->createSound(filename,
FMOD_LOOP_OFF,0,&sound);
```

- ... play it:

```
system->playSound(sound,NULL,false,&channel);
```

- ... and set the channel's 3D attributes as in the previous slide

DSP Playback in 3D

- Start playing the oscillator (preferably paused):

```
system->playDSP(dsp, NULL, false, &channel);
```

- Then assign 3D properties to the channel as in the previous slide

- Adjusting the minDistance can ensure audibility:

```
channel->set3DMinMaxDistance(200f, 100000f);
```

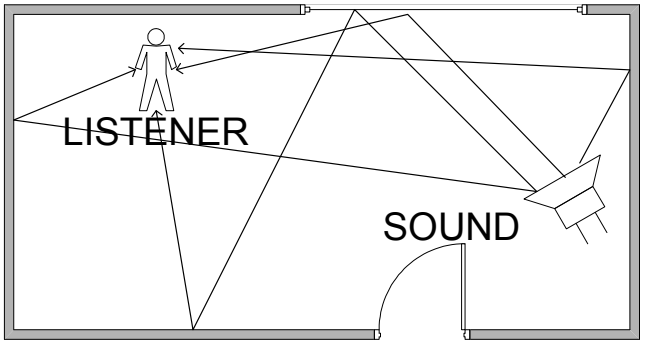
FMOD 3D
Modelling and Occlusion

3D in FMOD

- Recap:
  - FMOD has its own 3D coordinate system and model
  - Need to coordinate
    - listener positions and velocities
    - sound source positions and velocities
    - objects in 3D that occlude or obstruct sound

Simulating Acoustics

- Reflection and conduction change sound on its way from source to listener.



- Reflection: Echo, Reverb
- Conduction: Distance roll-off
- Complex effects: Occlusion, Obstruction

Positioning 3D Sound

- Create position and velocity vectors:

```
FMOD_VECTOR pos1 = {-10.0f, 0.0f, 0.0f};

FMOD_VECTOR vel1 = {0.0f, 0.0f, 0.0f};
```

- And set channel attributes:

```
channel->setMode(FMOD_3D);
channel->set3DAttributes(&pos1, &vel1);
```

Transforming and Updating 3D Positions

- Transformation of OpenGL vectors to FMOD vectors:

```
FMOD_VECTOR fmodVec = FMOD_VECTOR();
fmodVec.x = glVec3.x; fmodVec.y = glVec3.y; fmodVec.z = glVec3.z;
```

  - Make this ^ into a converter function for ease: e.g.

```
void ToFMODVector(glm::vec3 &glVec3, FMOD_VECTOR *fmodVec)
```

- Transform position vectors

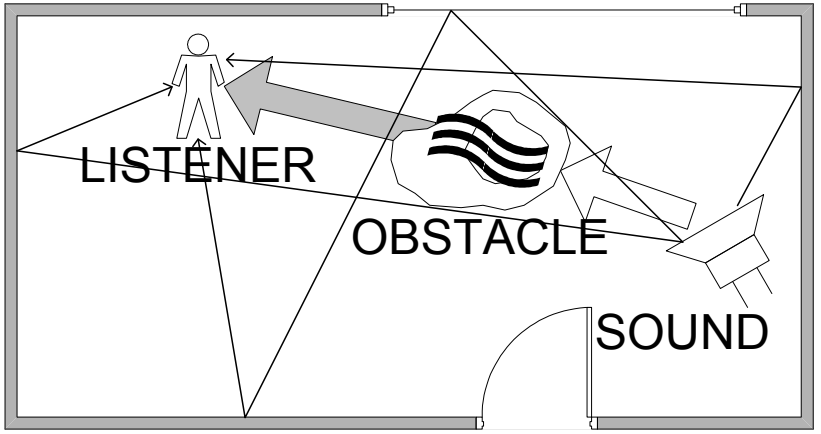
```
ToFMODVector(camera->GetPosition(), &camPos);
```

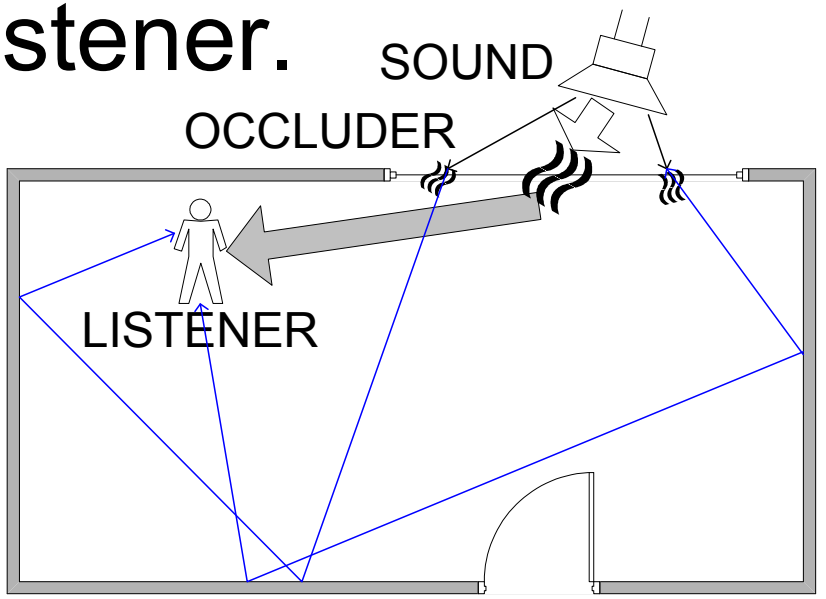
- Update the listener position:

```
system->set3DListenerAttributes(0, &camPos, NULL, NULL, NULL);
system->update();
```

Occlusion

- Obstruction: obstacle blocks direct path between sound source and listener.
- Occlusion: occluder blocks all paths from the source to the listener.





Creating 3D Geometry Objects for Occlusion

First, convert your polygon to FMOD format:

```
FMOD_VECTOR wallPoly[4];
ToFMODVector(v1, &wallPoly[0]);
ToFMODVector(v2, &wallPoly[0]);
ToFMODVector(v3, &wallPoly[0]);
ToFMODVector(v4, &wallPoly[0]);
```

Creating 3D Geometry Objects for Occlusion

Then, create the object in FMOD's system

```
FMOD::Geometry *geometry;

system->createGeometry(1, 4, &geometry);

int polyIndex = 0;

// these numbers control direct, and reverb occlusion settings (0-1)

geometry->addPolygon(1.0f, 1.0f, TRUE, 4, wallPoly, &polyIndex);

FMOD_VECTOR wallPosition;

ToFMODVector(position, &wallPosition);

geometry->setPosition(&wallPosition);

geometry->setActive(TRUE);
```

Real-time DSP
Circular Buffers
FMOD Custom DSP Programming

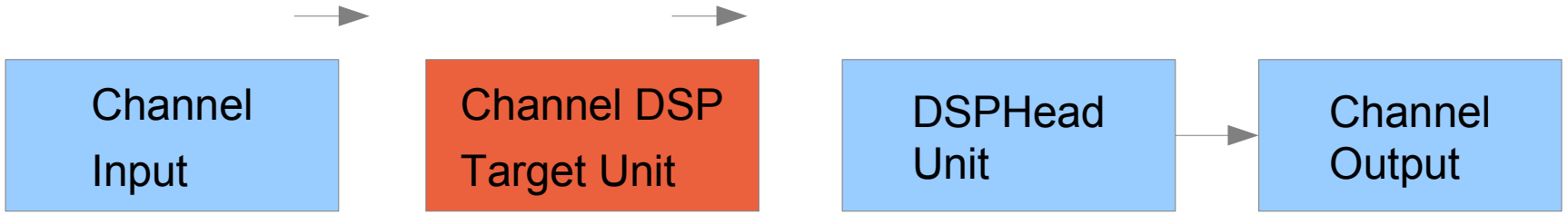
Custom FMOD DSPs

- DSP inserts
  - For whole system (all channels):

```
system->addDSP()
```

  - For specific channel

```
channel->addDSP()
```



Creating a custom DSP

```
// Create a DSP descripton
FMOD_DSP_DESCRIPTION dspdesc;
memset(&dspdesc, 0, sizeof(dspdesc));

strncpy_s(dspdesc.name, "My first DSP unit",
    sizeof(dspdesc.name));
dspdesc.numinputbuffers = 1;
dspdesc.numoutputbuffers = 1;
dspdesc.read = DSPCallback;


// Create your new DSP object
result = system->createDSP(&dspdesc, &dsp);
FmodErrorCheck(result);
```

Custom DSP callback

```
FMOD_RESULT F_CALLBACK DSPCallback(FMOD_DSP_STATE *dsp_state,
       float *inbuffer, float *outbuffer,
       unsigned int length, int inchannels, int *outchannels)
{
   for (unsigned int samp = 0; samp < length; samp++)
   {
    for (int chan = 0; chan < *outchannels; chan++)
    {
        outbuffer[(samp * *outchannels) + chan] =
               inbuffer[(samp * inchannels) + chan];
    }
   }
};
```

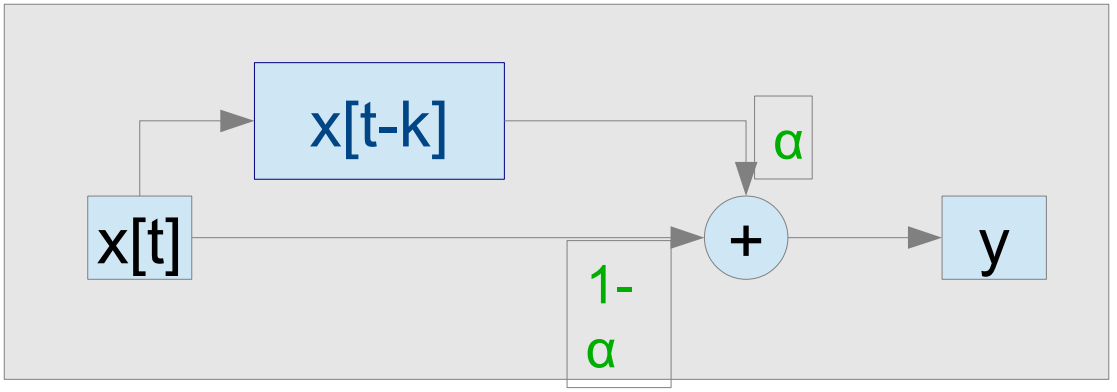
Example:

2 channel `inbuffer`

Creating a Delay effect

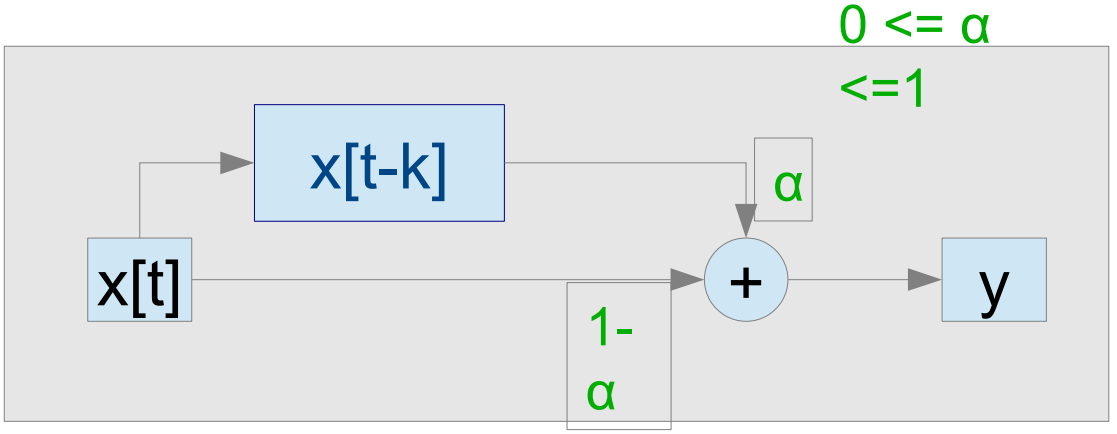
Echo adds a **delayed** signal to the original input
Both delayed and original signal are scaled to stay in range, with $0 <= \alpha <= 1$

Signal flow diagram of the effect:



Accessing x[t-k]

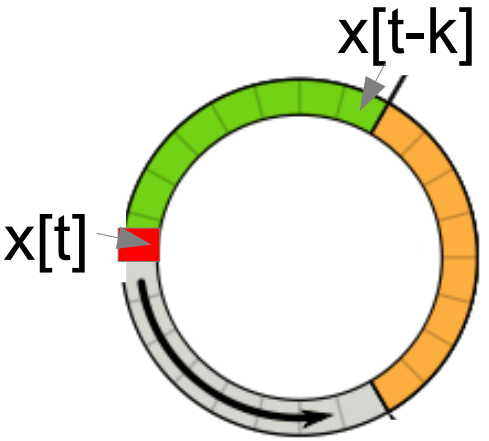
- Buffer the input signal for a time at least equal to the sample delay time k
- Then access x[t-k] from the buffer



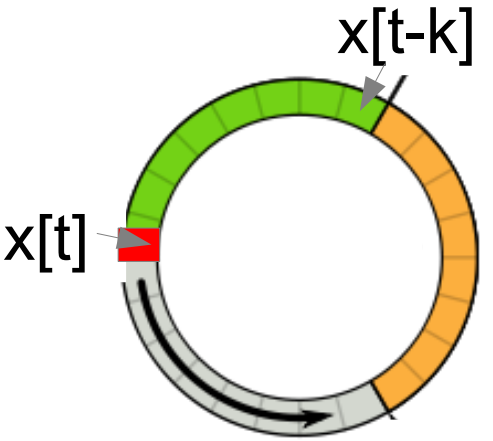
Circular Buffer

- Goal: continuous buffering of incoming data in linear array
- Address the buffer:

```
pos % buffer_length
```



Circular Buffer



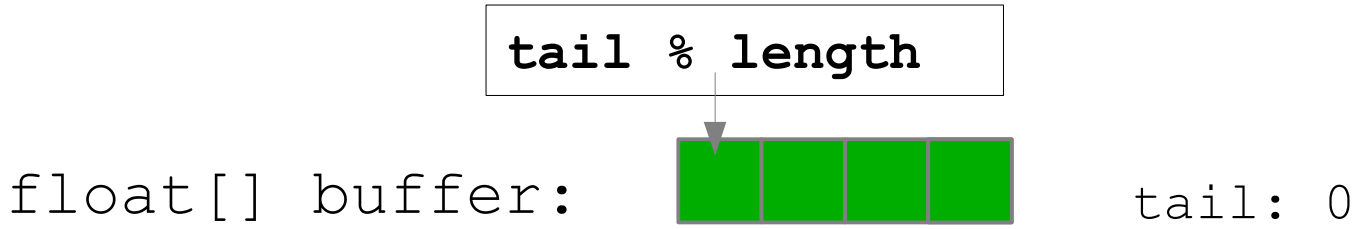
- Minimise buffer maintenance costs
- Address the buffer:

```
pos % buffer.length
```

- Use index `tail` to manage the buffer
- `tail % length` points to write position

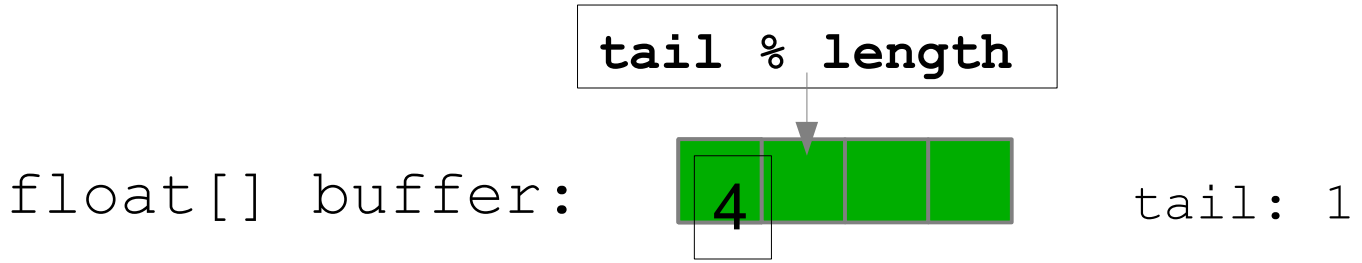
Circular Buffer

```
CircBuffer *cBuffer = new CircBuffer(4);
```



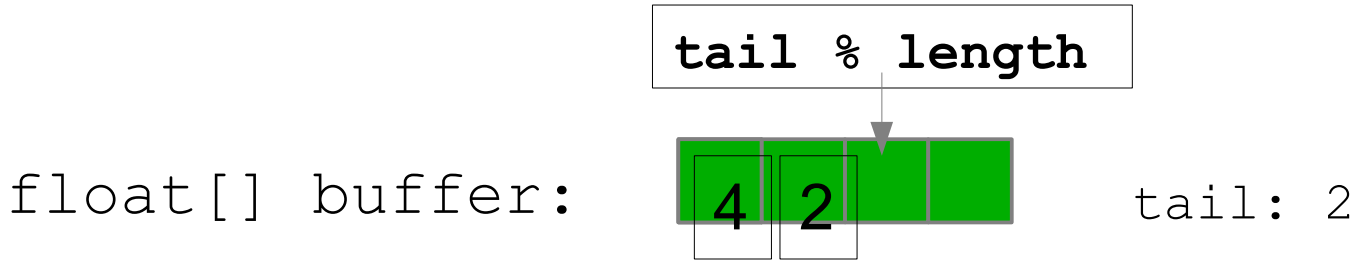
Circular Buffer

```
CircBuffer cBuffer = new CircBuffer(4);
Cbuffer->Put(4.0);
```



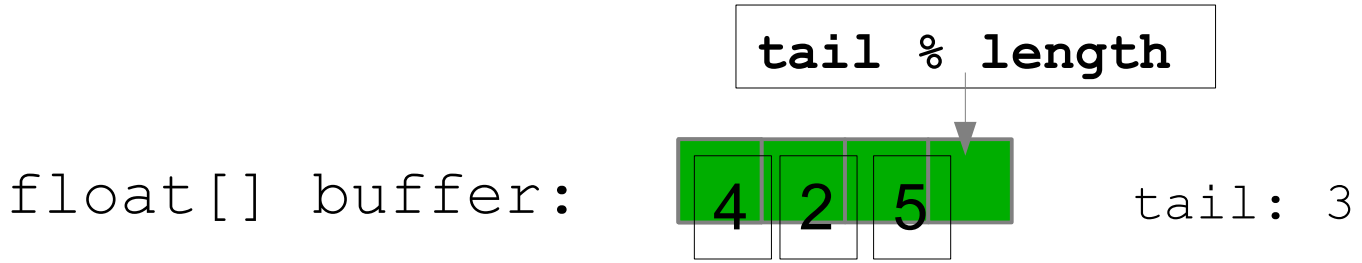
Circular Buffer

```
CircBuffer cBuffer = new CircBuffer(4);
Cbuffer->Put(4);
Cbuffer->Put(2);
```



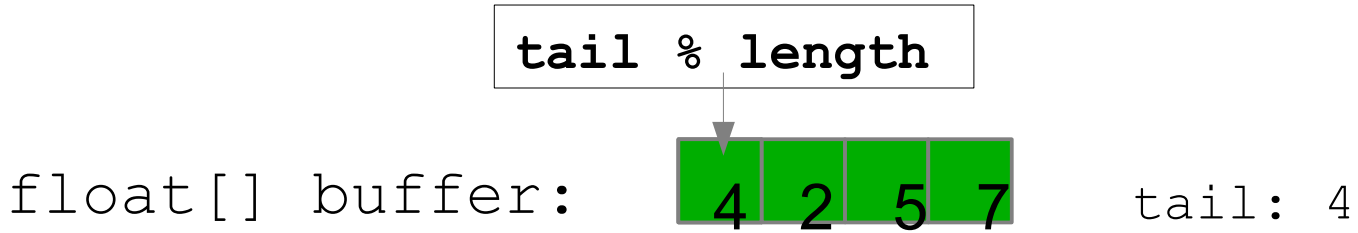
Circular Buffer

```
CircBuffer cBuffer = new CircBuffer(4);
Cbuffer->Put(4);
Cbuffer->Put(2);
Cbuffer->Put(5);
```



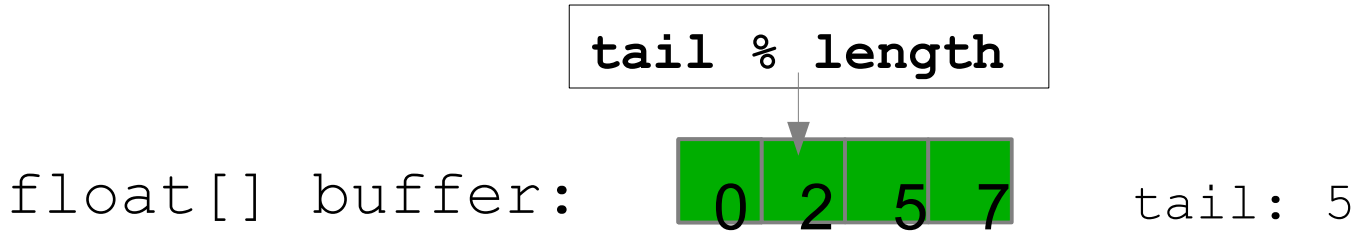
Circular Buffer

```
CircBuffer cBuffer = new CircBuffer(4);
Cbuffer->Put(4);
Cbuffer->Put(2);
Cbuffer->Put(5);
Cbuffer->Put(7);
```



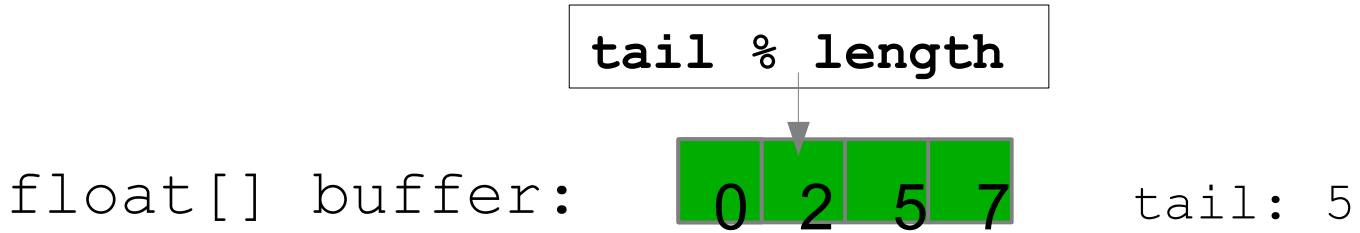
Circular Buffer

```
CircBuffer cBuffer = new CircBuffer(4);
Cbuffer->Put(4);
Cbuffer->Put(2);
Cbuffer->Put(5);
Cbuffer->Put(7);
Cbuffer->Put(0);
```



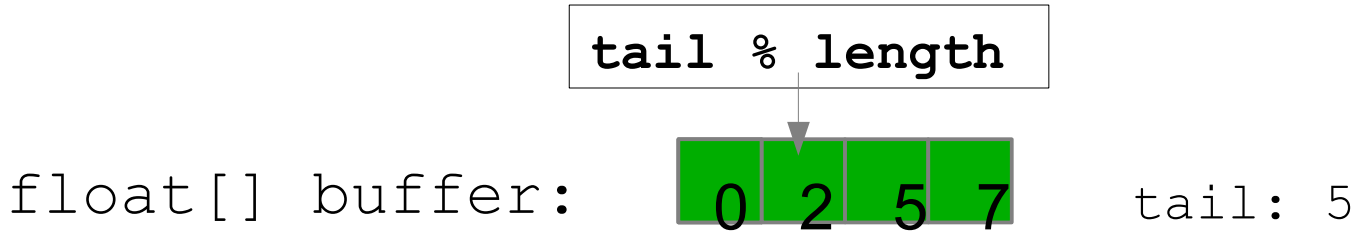
Circular Buffer

```
cBuffer->AtPosition(2);
  // returns: 5
```



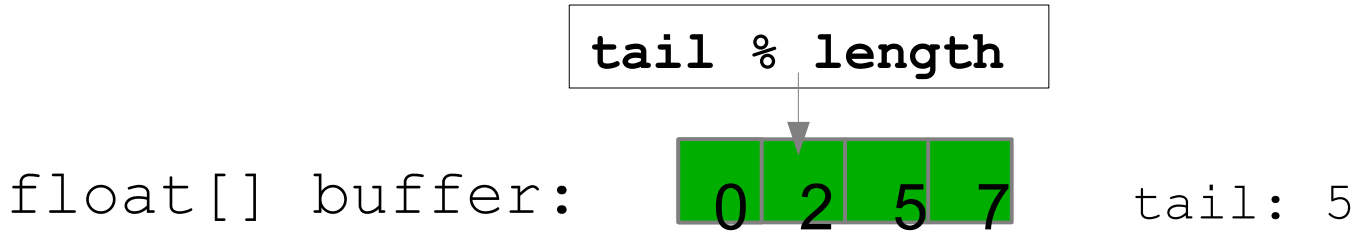
Circular Buffer

```
cBuffer->AtPosition(2);
  // returns: 5
cBuffer->AtPosition(4);
  // returns: 0
```



Circular Buffer

```
cBuffer->AtPosition(2);
  // returns: 5
cBuffer->AtPosition(4);
  // returns: 0
cBuffer->AtPosition(6);
```



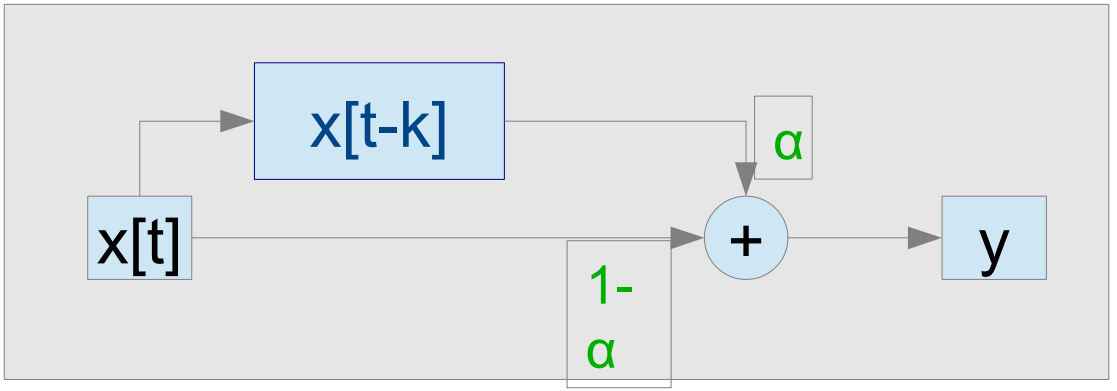
Circular Buffer

```
cBuffer->AtPosition(2);
  // returns: 5
cBuffer->AtPosition(4);
  // returns: 0
cBuffer->AtPosition(6); // throws Exception! Why?
```

Echo effect

Echo adds a **delayed** signal to the original input
Both delayed and original signal are scaled to stay in range

Signal flow diagram of the effect:

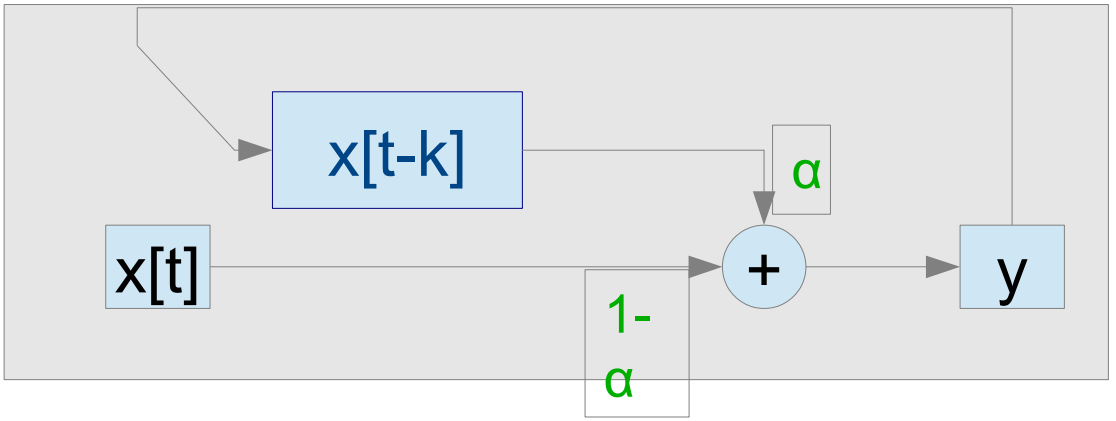


0 <= α <=1

Feedback delay

There is also a feedback version of the delay:
The output is **fed back** into the buffer!

Simplified signal flow diagram of the effect:



0 <= α
<=1

Frequency analysis in FMOD

```
// create FFT DSP object
DSP * fft;
system->createDSPByType(FMOD_DSP_TYPE_FFT, &fft);
// define spectrum length and window
fft->setParameterInt(FMOD_DSP_FFT_WINDOWSIZE, 1024);
Fft->setParameterInt(FMOD_DSP_FFT_WINDOWTYPE,
       FMOD_DSP_FFT_WINDOW_HANNING);

// get spectrum data
FMOD_DSP_PARAMETER_FFT *fftData;
fft→getParameterData(FMOD_DSP_FFT_SPECTRUMDATA,
       (void **)fftData);
```

Reading

- FMOD Core API tutorials on DSP architecture and usage
- Core API example

  `api/core/examples/dsp_custom.cpp`

  check especially `FMOD_DSP_DESCRIPTION` and the different callbacks to learn how to change parameters and pass data between multiple invocations

Game Audio Workflow

Game Audio Workflow

- Audio producers → audio programmers
- Audio producers:
  - recording engineers
  - sound designer, voice artists, composers and musicians
- Interface
  - FMOD Studio (Designer) for producers
  - FMOD Studio low level API for programmers

FMOD API Event Model

FMOD Event Model

- High level model to support sound and interaction design for games
- FMOD Studio supports to
  - define, manage, package, test and optimise sound 'events' with parameters
- FMOD event model
  - high level interface to programmers, leaving audio matters mostly to the sound designer

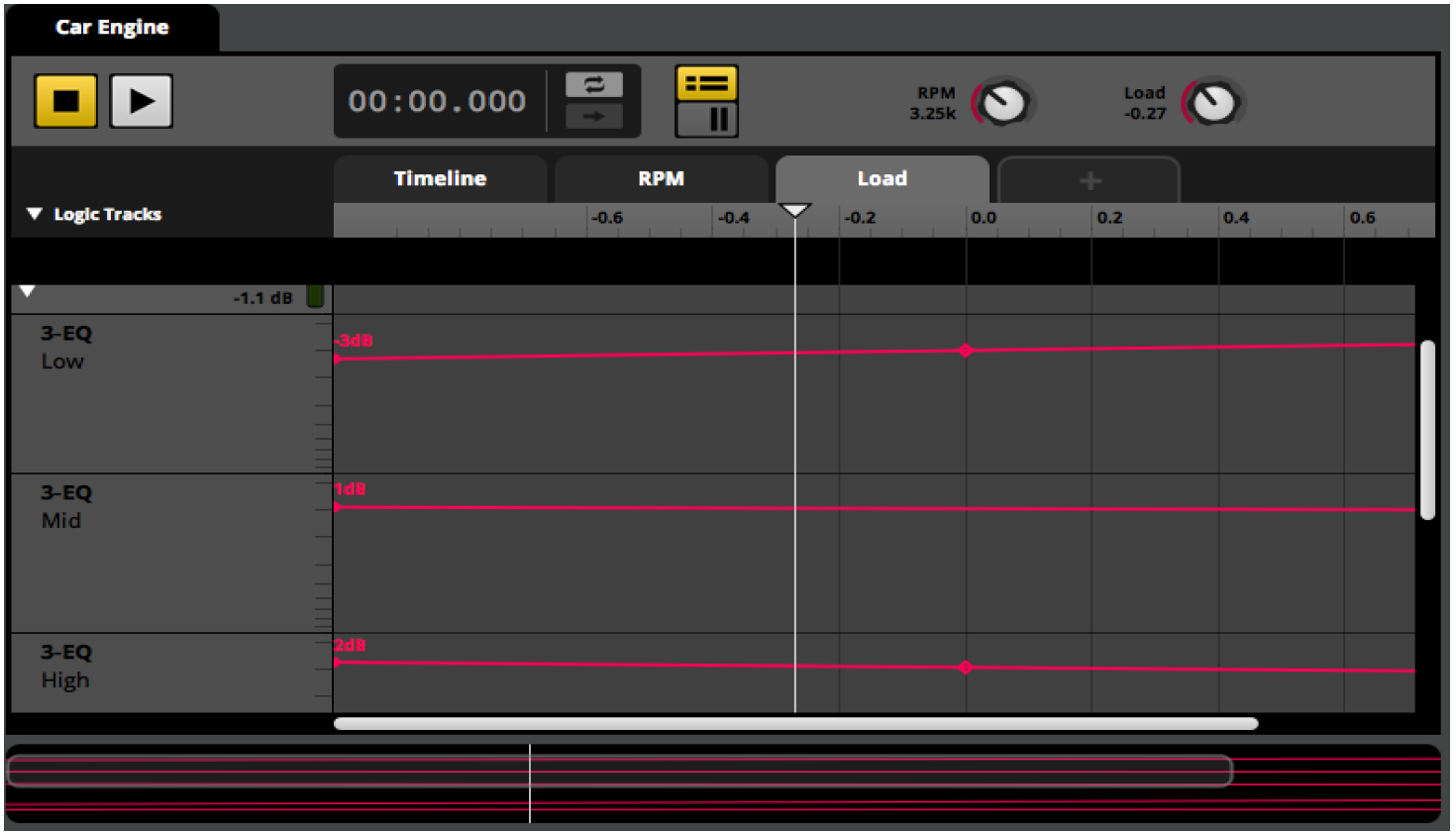
FMOD Studio: Event Editor

FMOD Event Parameters

- Parameters control the sound
- Parameter values are set by the game
- Designer defines event's reaction to parameters
  - Sound mix/crossfades
  - Effects: Volume, Pictch, Reverb, Chorus, ...
  - Auto-pitch (pitch changes)
  - Randomisation

FMOD Studio: Event Parameters

EQ settings



FMOD Studio: Event Parameters

Cross-Fading