Module IN3031 / INM378 Digital Signal Processing and Audio Programming

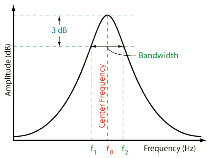
Tillman Weyde t.e.weyde@city.ac.uk

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

Spatial Sound and Hearing

Sound and Hearing in Space

- Speed of Sound (*vs*) 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
  - fp = f vs/(vs-vr)
    with fp: perceived frequency ,
    f: frequency, vr: velocity relative to the listener
    (positive = approaching)
  - Example: car moves with 68 m/s producing a 300 Hz sound. fp 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 inthe previous slide
- Adjusting the minDistance can ensure audibility:

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

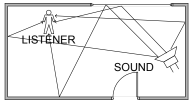

FMOD 3D Modelling and Occlusion

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);
```

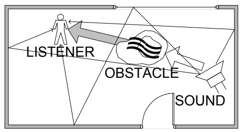
• Update the listener position:

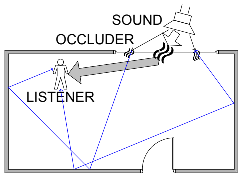
```
system->set3DListenerAttributes(0, &camPos, NULL, NULL,
  NULL);
system->update();
```


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

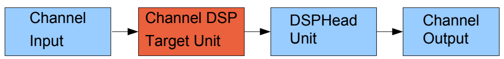
Custom FMOD DSPs

- 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

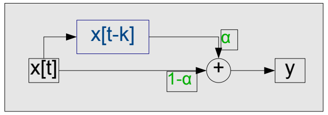
```
FMOD_RESULT F_CALLBACK DSPCallback(FMOD_DSP_STATE *dsp_state,
    float *inbuffer, float *outbuffer,
    unsigned int length, int inchannels, int *outchannels)
{
    for (unsigned int samp = 0; samp < length; samp++)
    {
        for (int chan = 0; chan < *outchannels; chan++)
        {
       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

Example:
2 channel `inbuffer`

0	1	2	3	4	5
L	R	L	R	L	R

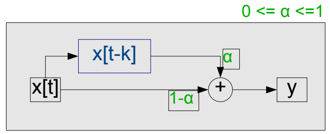
Creating a Delay effect

- Echo adds a **delayed** signal to the original input
- Both delayed and original signal are scaled to stay in range, with 0 <= α <=1
- **Signal flow diagram** of the effect:



Accessing x[t-k]

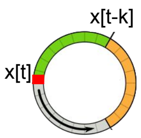
- Buffer the input signal for a time at least equal to the sample delay time k
- Then access x[t-k] from the buffer



Circular Buffer

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

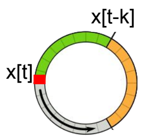
  `pos % buffer_length`

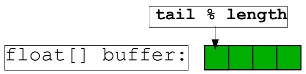


Circular Buffer

- Minimise buffer maintenance costs
- Address the buffer:

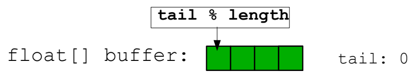
  `pos % buffer.length`
- Use index `tail` to manage the buffer
- `tail % length` points to write position





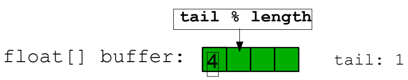
Circular Buffer

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



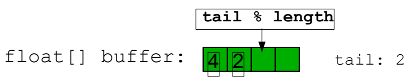
Circular Buffer

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



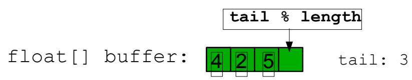
Circular Buffer

```
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);
```

```
                 tail % length
float[] buffer:  4 2 5 7        tail: 4
```


Circular Buffer

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

```
                 tail % length
float[] buffer:  0 2 5 7        tail: 5
```


Circular Buffer

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

```
                 tail % length
float[] buffer:  0 2 5 7        tail: 5
```


Circular Buffer

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

```
                 tail % length
float[] buffer:  0 2 5 7        tail: 5
```


Circular Buffer

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

```
                 tail % length
float[] buffer:  0 2 5 7        tail: 5
```


Circular Buffer

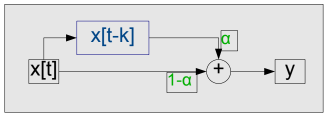
```
cBuffer->AtPosition(2);
  // returns: 5
cBuffer->AtPosition(4);
  // returns: 0
cBuffer->AtPosition(6); // throws Exception! Why?
```

```
                 tail % length
float[] buffer:  0 2 5 7
```


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:

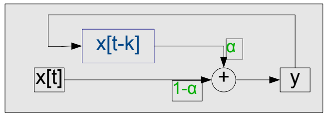


0 <= α <=1

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 Studio Low-level API tutorials on DSP architecture and usage

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

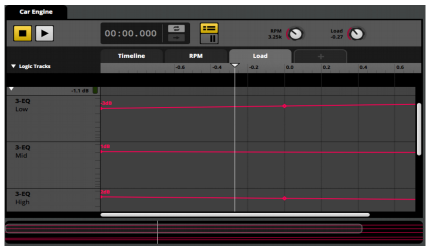
FMOD Studio: Event Editor

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

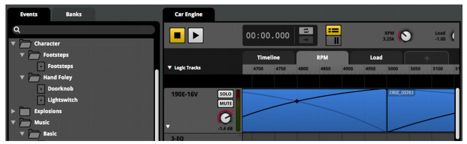
FMOD Studio: Event Parameters

EQ settings



FMOD Studio: Event Parameters

Cross-Fading

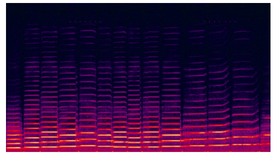


Music in Games

Interactive Music

- Games have no fixed progression of events
  - Music needs to be adaptable. Approaches:
    - Write different pieces of music: only possible to a limited extent
    - Loop parts of the music: common approach
  - Adaptation
    - Horizontal re-sequencing (different sequences of looped sound material)
    - Vertical re-orchestration (different combinations layered sounds)

Musical Structure



- Loops and layers should (normally) create a coherent musical structure in
  - Time (metre and rhythm, horizontal)
  - Frequency (harmony, vertical)

Beat, Metre, and Metrical Hierarchies

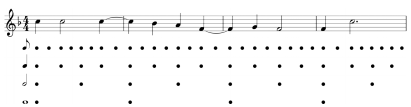
- Introduction
- Beat and Metre
- Metrical Stress Patterns and Time Signatures
- Even, odd, and compound metres
- MIDI Time Signature
- Computing a Metrical Hierarchy

Introduction

Introduction

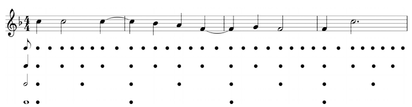
- How do dancers synchronise with the music?
- How do musicians synchronise when they play together?
- This by done by using patterns in time
- Musical beat and metre organise these patterns

Music and Time



- Musical time is structured:
  - the beat (or pulse) creates a (mostly) regular grid
  - Metre creates regular beats groups with internal structure

Beat



- Most music has a perceived *beat* or *pulse*
  - a succession of stressed point in time (beats)
  - beats have approximately equal durations between them (isochronous sequence)
- The frequency of beats is called the tempo
  - tempo is defined in BPM (beats per minute)
  - tempos are typically in the range 50-200BPM

Beat Perception

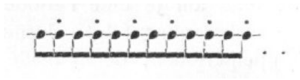
- The beat is inferred in the perception of music
  - a perfectly isochronous sequence of notes evokes a beat unequivocally
  - a very irregular sequence evokes no beat perception
  - composers and musicians use this differently (e.g. classical vs. jazz)
- Beat perception is related to movement (dance music, work songs, ...)

Beats and Musical Organisation

- Notes can occur aligned to the beat, but at a higher or lower rate
- The temporal organisation music is based on stressed and unstressed notes

Common Patterns

- Beats are grouped in patterns
- one stressed (downbeat) one light (upbeat), (stress is indicated by dots over the note)



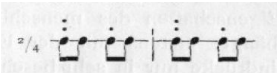
- This is even perceived when a completely uniform isochronous sequence is played

Metre

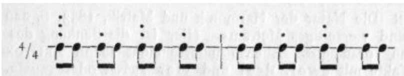
- The distribution of strong and weak times in time is called the metre of the music
- The repeated pattern represents usually one *bar* (brit.) or *measure* (am.), delimited by vertical *barlines*
- In music notation, the metre is usually indicated by the *time signature*
- time signature is written as a fraction x/y
  - x is the number of beats per measure
  - y indicates that a beat has 1/y duration

Common Time Signatures

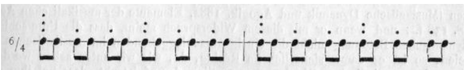
2/4



4/4



6/4



Metrical Levels

- Levels of stress in a time signature

Anacrusis/Upbeat

- The first measure may be incomplete

Well known example

Metrical Organisation

```
                                                        Level
    4/4                4/4            4/4               0
   /   \              /   \          /   \
1/2      1/2      1/2      1/2   1/2      1/2           1
 |      /   \      |        |     |        |
 |   1/4     1/4   |        |     |        |            2
 |  /   \   /   \  |        |     |        |
 | 1/8 1/8  1/8 1/8 |       |     |        |            3
 | |    |   |    |  |       |     |        |
      It's been  a  hard      day's        night
```


Compound meters

- A measure can have irregular subdivisions e.g. 3+2+2 / 8
- Examples:
  - Bulgarian dances
  - Jazz ('Take Five', Latin Rhythms)

Musical Instrument Digital Interface (MIDI)

- A hardware, messaging and file format standard, introduced in 1982
- Binary format, most messages are 8-bit
- Modelled after western music theory
- Now ubiquitous in digital and (some) analogue instruments
- Other messaging formats are available: CV, Open Sound Control, MIDI HD (in development), some ad-hoc solutions

MIDI Time Signature

MIDI Standard Files (0) Meta Message

FF 58 04 nn dd cc bb **Time Signature**

Time signature of the form:
nn/2^dd
eg: 6/8 would be specified using nn=6, dd=3
The parameter cc is the number of MIDI Clocks per metronome tick.
Normally, there are 24 MIDI Clocks per quarter note. However, some software allows this to be set by the user. The parameter bb defines this in terms of the number of 1/32 notes which make up the usual 24 MIDI Clocks (the 'standard' quarter note).

nn	Time signature, numerator
dd	Time signature, denominator expressed as a power of 2. eg a denominator of 4 is expressed as dd=2
cc	MIDI Clocks per metronome tick
bb	Number of 1/32 notes per 24 MIDI clocks (8 is standard)

A MIDI Example

(24 ticks per quarter note)
0 NoteOn 80 127
24 TimeSig 4 2 24 8
24 NoteOn 80 0 <-- (a.k.a NoteOff)
24 NoteOn 80 127
36 NoteOn 60 127
48 NoteOn 60 0
48 NoteOn 60 127
66 NoteOn 60 0
72 NoteOn 80 127
96 NoteOn 80 0

Loops and Metre

- The metrical structure is normally maintained during loop playback.
- Common loop sizes are 4, 8 or 16 bars (although sometimes musical structures have different values, e.g. 'Eleanor Rigby' by the Beatles has a 5 bar structure)

Loops and Harmony

- Harmony describes the sounding of several (pitched) notes together
- In harmonic contexts, some notes sound consonant, others sound dissonant/inappropriate.
- Layered music loops need common harmonic structure (not true for sound loops)
- Each layer in the same harmonic pattern ensures they are musically 'compatible'

MIDI vs Audio

- MIDI representation
  - used mostly in music production
  - used to be applied in Games directly
- MIDI is symbolic representation
  - Advantages:
    - independent tempo and pitch
    - easy to modify for musicians
    - low data volume
  - Disadvantages
    - sound quality (depends on used sound library)

MIDI vs Audio in Loops

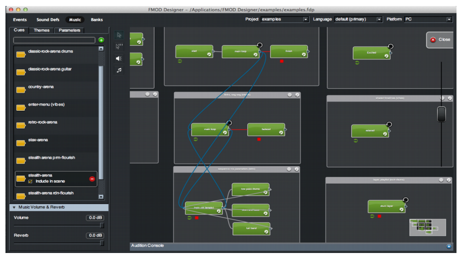
- Audio
  - can have superior quality (e.g. recorded human performance)
  - costly and lossy change of pitch/tempo
  - changing individual notes hardly possible
    → careful planning needed, good for final production
- MIDI
  - very flexible (easy to change tempo, pitch, notes)
    → can experiment, good for developing a soundtrack

FMOD for Game Music

- Supports
  - Loop
  - Synchronisation based on beats and bars
  - Conditional transitions and repetitions
- Used to be separate FMOD Music system, now integrated with Studio Event system

Interactive Music in FMOD

- FMOD Designer interface (a bit like UML)



Interactive Music in FMOD

- The new FMOD Studio interface
  (more like Spaghetti code)