FMOL Architecture

Sergi Jordà

sergi.jorda@iua.upf.es
FMOL (F@ust Music On Line) 1997-2001

- Project for la Fura dels Baus, F@ust 3.0 show (based on Goethe’s)
  Given the important role symbolized by the Internet in our play, we would like part of its music be composed by cybercomposers around the world

- Excerpts from the FMOL Decalogue (S.Jordà & C.Padrissa, 1997)
  - We don’t want MIDI music
  - We don’t want notes, we want Sound
  - We don’t want Keyboardists, we want Sound Sculptors
  - We won’t demand composers more gear than a 50$ multimedia soundcard and an Internet connection
  - Music will be composed in real-time with the mouse
  - New composers will be able to modify, process or distort previous compositions.
  - FMOL output will literally be visual music
Musical & Social Implications

? FMOL 1.0 (18 Jan – 16 April 1998) : + 1,200 20”-compositions from + 50 composers
  - 50 compositions selected and included in F@ust 3.0 soundtrack
  - CD with “remixes” by 12 authors, late 1998

? FMOL 2.0 (1 Sept – 8 Oct 2000) : + 600 60”-compositions
  - Selected ones constitute the electronic parts of the (otherwise orchestral) opera DQ (premiered at the Gran Teatre del Liceu of Barcelona, Nov. 2000)
FMOL (F@ust Music On Line) 1997-2001

Primary goals

- Collective composition on the Net
- Introduce newcomers into experimental electronic music
- “Cheap” (free) and available (no special hardware)
- “Sound over notes” priority (micro+macro control and NO General MIDI !)
- Attractive to both trained and non-trained electronic musicians (i.e. intuitive but intricate...)

Technical solutions

- Mouse driven software for RT synthesis and RT composition with peculiar “visual feedback” interface
- Standalone C program with HTTP facilities (on v. 1.0)
- Restricted to MS Windows (Direct X libraries)
- Non real-time collaboration (half-duplex: first listen then modify)
Direct X and Direct Sound

Windows, 3 ways for real-time audio:

- MME API (Microsoft Multimedia Extensions, older, bigger latency, uses winmm.lib)
- DirectSound API (part of Microsoft DirectX, medium latency, supports ActiveX plugins), does not work on NT 4.0 or <, works on W95,98,ME,2000 and XP, uses winmm.lib + DSound.lib
- Asio (Steinberg, smaller latency, not all cards provide drivers)

FMOL was made with DX 3.0 (currently DX 8.0 but back compatibility) and also uses DirectDraw for the GUI (DDraw.lib)
The Microsoft® DirectSound® application programming interface (API) is the wave-audio component of the DirectX®. DirectSound provides low-latency mixing, hardware acceleration, and direct access to the sound device. It provides this functionality while maintaining compatibility with existing device drivers.

DirectSound enables both capture and playback of wave sounds. It also supports property sets, which enable application developers to take advantage of extended services offered by sound cards and their associated drivers.

- Querying hardware capabilities at run time to determine the best solution for any given personal computer configuration [x]
- Using property sets so that new hardware capabilities can be exploited even when they are not directly supported by DirectSound
- Low-latency mixing of audio streams for rapid response (not used)
- Implementing three dimensional (3-D) sound
- Capturing sound [x]

[x] used in FMOL
Direct Sound (and 2)

- The DirectSound buffer object represents a buffer containing sound data. Buffer objects are used to start, stop, and pause sound playback, as well as to set attributes such as frequency and format.
- The primary sound buffer holds the audio that the listener will hear. Secondary sound buffers each contain a single sound or stream of audio. DirectSound automatically creates a primary buffer, but it is the application's responsibility to create secondary buffers. When sounds in secondary buffers are played, DirectSound mixes them in the primary buffer and sends them to the output device. Only the available processing time limits the number of buffers that DirectSound can mix.
- The FMOL application is responsible for the mixing into a “principal” secondary buffer.
Threads and Synchronicity

? MME uses a Win32 thread. Writes blocks of data to the HW device and waits for events that signal buffer completion.

? DirectSound Uses a timer callback for the background "thread". Polls a circular buffer and writes blocks of data to keep it full.

(information taken from Phil Burk’s and Ross Bencina’s PortAudio)

In FMOL a unique clock controls the application main loop. At each frame:

1. Checks inputs (from mouse, keyboard and MIDI IN)
2. Maps input info into control values, generates new values according to LFOs, [sequences these control values], [outputs them to MIDI OUT], [read more values from scheduler]
3. Redraws horizontal lines according to the inputs and the control values
4. Redraws 6(8) strings after the 6(8) buffer contents (prev. frame)
5. Applies all control values to the synth engine
6. 6 (8) tracks are calculated (synthesized, processed...) with these values (in 6 – 8-independent “for” loops)
7. 6 (8) tracks are mixed into main secondary buffer

NB. But the program uses the Multithreaded run-time library.
8 independent stereo tracks (2-sec. each)

Mixed synchronously into a DS stereo secondary buffer (also 2-sec)

NB. 25 Hz is not true anymore. Now configurable in a text file (bambu.ini). With a PIII almost 100 Hz can be achieved with no gaps.
FMOL’s Synth Engine

- 6 (8) audio channels (independent or not)
- For each channel: 1 generator & 3 processors (selectable from more than 100 algorithms or variations)

For each generator-processor: 4 of its parameters are controlled by independent LFOs (total of 6x4x4=96 simultaneous LFOs)

For each LFO, dynamic control on frequency, amplitude & shape (sin, square, saw, triangle, random)

LFOs are fundamental for all FMOL time evolution (no use of prerecorded sequences)
Clock Code Fragments

FMOL.cpp [WinMain()] \(\rightarrow\) see FMOLApp.Run();
FMOLApp.cpp [:Run()] \(\rightarrow\) see rti.Run();
RTI.cpp [:Run()]

The main loop is in BambuLoop.cpp [AppUpdate()]
```c
int WINAPI WinMain (HINSTANCE hThisInstance, HINSTANCE hPrevInstance,
                   LPSTR lpszCmdParam, int nCmdShow)
{
    MSG msg;
    lpszCmdParam = lpszCmdParam;
    hPrevInstance = hPrevInstance;

    if (!FMOLApp.Init(lpszCmdParam, nCmdShow))
    {
        FMOLApp.Close();
        return FatalError(__APP_NAME__, "Init error");
    }

    //
    // Inner loop
    //
    while (true)
    {
        if (PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE))
        {
            if (!GetMessage (&msg, NULL, 0, 0))
            {
                // WM_QUIT
                FMOLApp.Close();
                FMOLApp.Close();
                return msg.wParam;
            }
            TranslateMessage (&msg);
            DispatchMessage (&msg);
        }
        FMOLApp.Run();
    }

    return msg.wParam;
}
```
void AppQUpdate()
{
    theSeq.Update();
}

int CFMOLApp::Run()
{
    if (theSeq.MIDIOutDrv.bSync) AppQUpdate();

    // Gráficos
    if (rti.Run())
    {
        if (! theSeq.MIDIOutDrv.bSync )
            AppQUpdate();

        DirectDraw.fSoftClear();
        DirectDraw.SetCurrentContext();
        DirectDraw/DDFlip();
        DirectDraw/DDDSErase();
        Display();
        AppUpdate();
    }
    return 1;
}

void CEventsSeq::Update()
{
    MIDIOutDrv.Update();
    if (MIDIOutDrv.bNextBlock)
    {
        if (bMode || pReadSeq || bModeLite)
            ReadEvents(); // play or record
        MIDIOutDrv.NextBlock();
        MIDIOutDrv.bNextBlock=FALSE;
    }
}

void CTracker::NextBlock()
{
    for (BYTE i=0; i<CT_NUMTRACKS; i++)
    {
        track[i].NextBlock();
        track[i].RunPlugIns();
    }
    mix.NextBlock();
    Mix();
    mix.RunPlugIns();
    CopyMemory(pBuffer, mix.pBlock, dwBlockSize);
    bNextBlock=FALSE;
class CTracker: public CSoundDriver {

public:

  CPlugTrack track[CT_NUMTRACKS];
  CPlugTrack mix;
  BYTE bMixDiv;  // Numero divisor de la mezcla

  BOOL Init(LPVOID phwnd, char* szCfgFile="tracker.ini");
  void Done(void);
  void Clear(void);

  void Update(void);
  void Mix(void);

  void NextBlock(void);

  void PlugIn(CSPlugIn *PlugIn, BYTE cTrack, BYTE cNum);  // Num=posición
  void MixPlugIn(CSPlugIn*PlugIn, BYTE cNum);

};

void CPlugTrack::RunPlugIns() {

  for(BYTE p=0; p<CPT_MAX_PLUGINS; p++)
  {
    if(PlugIn[p])
    {
      dwCurWrite=dwBuffOffs;
      PlugIn[p]->NextBlock();
    }
  }

}
FMOL Demo 1

1. Instruments configuration
2. Pitch and amplitude control
3. Showing different algorithms
4. Effects control
5. Arpeggiator
6. LFOs
7. Generators and processors
FMOL Configuration window and graphical interface

Synthesis instruments / Sampler instruments / Filters / Processing Instruments (2+2+2+2)...

orch. has 8 snapshots: [F] [G] [H] [J] [K] [L] [N] [ ]

? Filtros o efectos disponibles para asignar a cualquiera de las líneas horizontales.
The main MFC applications shows the instrument list reading the configuration file (*.cfg)
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Type</th>
<th>Param. 1</th>
<th>Param. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BASIC OSCILLATORS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sine Wave</td>
<td>Generator</td>
<td>MS Pitch</td>
<td>Amplitude</td>
</tr>
<tr>
<td>Square Wave</td>
<td>Generator</td>
<td>MS Pitch</td>
<td>Amplitude</td>
</tr>
<tr>
<td>Sawtooth Wave</td>
<td>Generator</td>
<td>MS Pitch</td>
<td>Amplitude</td>
</tr>
<tr>
<td>Triangular Wave</td>
<td>Generator</td>
<td>MS Pitch</td>
<td>Amplitude</td>
</tr>
<tr>
<td>Pulse Train</td>
<td>Generator</td>
<td>MS Pitch</td>
<td>Amplitude</td>
</tr>
<tr>
<td>White Noise</td>
<td>Generator</td>
<td>NO</td>
<td>Amplitude</td>
</tr>
<tr>
<td><strong>SAMPLE PLAYERS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sample Player</td>
<td>Generator</td>
<td>MS Pitch</td>
<td>Amplitude</td>
</tr>
<tr>
<td>Waveta.Samp.Pl</td>
<td>Generator</td>
<td>Wav.Sound #</td>
<td>Amplitude</td>
</tr>
<tr>
<td>Scratch Samp.Pl.</td>
<td>Generator</td>
<td>Frequency</td>
<td>Amplitude</td>
</tr>
<tr>
<td><strong>BASIC MODULATORS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ring Modulation</td>
<td>Processor</td>
<td>MS Pitch</td>
<td>Modul %</td>
</tr>
<tr>
<td>Amplitude Modul.</td>
<td>Processor</td>
<td>MS Pitch</td>
<td>Modul %</td>
</tr>
<tr>
<td><strong>FEEDBACK ALGORITHMS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Karplus Strong</td>
<td>Generator</td>
<td>MS Pitch</td>
<td>Amplitude</td>
</tr>
<tr>
<td>Binary Modul.</td>
<td>Generator</td>
<td>Primary Pitch</td>
<td>Amplitude</td>
</tr>
<tr>
<td>K.Str. Processor</td>
<td>Processor</td>
<td>MS Pitch</td>
<td>NO</td>
</tr>
<tr>
<td>B.Mod.</td>
<td>Processor</td>
<td>Primary Pitch</td>
<td>NO</td>
</tr>
<tr>
<td>Processor</td>
<td>Processor</td>
<td>Cutoff Freq.</td>
<td>NO</td>
</tr>
<tr>
<td>Resonant Filter</td>
<td>Processor</td>
<td>Res. Freq.</td>
<td>Q</td>
</tr>
<tr>
<td>Comb Reverb</td>
<td>Processor</td>
<td>Feedback</td>
<td>Delay</td>
</tr>
<tr>
<td>Comb Eco</td>
<td>Processor</td>
<td>Feedback</td>
<td>Delay</td>
</tr>
<tr>
<td>Comb Delay</td>
<td>Processor</td>
<td>NO</td>
<td>Delay</td>
</tr>
<tr>
<td>Comb Comb</td>
<td>Processor</td>
<td>Gain</td>
<td>Delay</td>
</tr>
<tr>
<td><strong>LINEAR DIFFERENCE EQUATION FILTERS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>High-Low Filter</td>
<td>Processor</td>
<td>Cutoff Freq.</td>
<td>NO</td>
</tr>
<tr>
<td>Resonant Filter</td>
<td>Processor</td>
<td>Res. Freq.</td>
<td>Q</td>
</tr>
<tr>
<td>Comb Reverb</td>
<td>Processor</td>
<td>Feedback</td>
<td>Delay</td>
</tr>
<tr>
<td>Comb Eco</td>
<td>Processor</td>
<td>Feedback</td>
<td>Delay</td>
</tr>
<tr>
<td>Comb Delay</td>
<td>Processor</td>
<td>NO</td>
<td>Delay</td>
</tr>
<tr>
<td>Comb Comb</td>
<td>Processor</td>
<td>Gain</td>
<td>Delay</td>
</tr>
<tr>
<td><strong>OTHERS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pitch Shift</td>
<td>Processor</td>
<td>Pitch Shift</td>
<td>NO</td>
</tr>
<tr>
<td>Panning</td>
<td>Processor</td>
<td>Angle</td>
<td>NO</td>
</tr>
<tr>
<td>Line In</td>
<td>Generator</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>

*In FMOL 1.0 (on-line 98) the 2+2+2+2 structure of the program forced the collaborative approach. FMOL 2.0 (on-line 2000) added new playing and overdub features and keys ➔ less collaboration!*
FMOL Class library

FMOL is very object oriented

- CSPlugin is the base class for all sound generation processes:
- CSPSource is the base class for classes that generate sound - fill buffers
- CSPSample or CSPSeno derive from CSPSource
- audio input is carried by CSPLineln also derived from CSPSource
- CSPFilter is the base class for classes that process sound - modify buffers
- CSPReverb or CSPComb derive from CSPFilter
- CSPSource and CSPFilter both derive from CSPlugIn
CSPlugin Overview

CSPlugIn has (among others) the following properties:

- CSoundTrack *pST; //write to
- CPLugTrack *pSTIn1, *pSTIn2; //read from
  (CPlugTrack derives also from CSoundTrack)

Some important methods:

- virtual BOOL NextBlock(); //main sound loop
- virtual void On(void); //called when plugin starts
- virtual void Off(void); //called when plugin stops
- virtual void Change(BYTE nParam, LONG iValue); //called on control changes

CSPSource adds important methods such as:

- void FillFreqMap(double freqMin, double freqMax, WORD iMin, WORD iMax);

Or redefines others as:

- Change(BYTE nParam, LONG iValue);

CSPFilter does not add anything (only zeroes reading tracks)

- pSTIn1 = pSTIn2 = NULL;
class CSoundTrack {

    public:
        // Propiedades
        //////////////////////////////////////////////////////////////////////////////////
        SHORT *pBuffer;    // Memoria
        SHORT *pBlock;     // Ventana actual en memoria
        DWORD dwMemSize;   // Tamaño de memoria (bytes)
        DWORD dwBuffOffs;  // Posición del bloque actual en la memoria
        DWORD dwBlockSize; // Tamaño del bloque
        DWORD dwBlockSmps; // Tamaño del bloque en muestras
        DWORD dwCurWrite;  // Posición actual de escritura
        DWORD dwAmp;       // Suma de amplitudes en el ultimo bloque (ABS)
        WORD wFreqMu;      // Frecuencia de muestreio (global)
        BOOL fMute;        // Si muted no se mezcla

        float lfMixAmp;    // Volumen de mezcla
        // Métodos
        //////////////////////////////////////////////////////////////////////////////////
        CSoundTrack() { pBuffer=pBlock=NULL; fMute=FALSE; }
        virtual BOOL Init(WORD dwFreq,WORD uTime,DWORD _dwBlockSize);
        virtual void Done(void);
        virtual void NextBlock(void);

        // Tratamiento secuencial de las muestras
        SHORT In(int idx);  // Lee muestra de la memoria (índice relativo)
        void Out(SHORT value); // Escribe muestra

        void NextPos(void) { dwCurWrite+=2; };       // Siguiente offset de lectura/escritura
        void Clear(void) {/*memset(pBlock,0,dwBlockSize)*/memset(pBuffer,0,dwMemSize); };    //NEW SJ
};
The `NextBlock()` Method

A for loop for the size of a frame - Simplest example: Delta pulse synth

```c
BOOL CSPPulse::NextBlock()
{
    if (!CSPSource::NextBlock()) return FALSE;

    SHORT val=(SHORT)wAmp;
    for (DWORD i=0; i<pST->dwBlockSmps; ++i)
    {
        --n;
        if (n<=0)
        {
            pST->Out(val>>ATTENUATE);
            n = wPeriod; //wSlidePer;
        }
        else pST->Out(0);
    }
    return TRUE;
}
```

pST is a member of CSPlugin (it is a CSoundTrack*)

cfg. practica-4.pdf, practica-5.pdf and practica-6.pdf for additional information about creating new derived instruments (tutorials for 3rd year ESUP students)
NextBlock for the superclasse and for the simplest case (Pulse generator)
The :NextBlock() Method (2)

? pST->Out(short) writes to the track buffer and advances the index.
? pST->In(-index) return previous values
  e.g. a line of Karplus-Strong algorithm:
  `val = (pST->In(-1) + pST->In(-wPeriod))>>1;`
? A generator that is a processor (e.g. Ring Modulator) uses pST1->In() and optionally pST2->In()

Other Methods:
? On() called when sound or a new note starts (see CSPPulse)
? Off() called when sound stops (automatic decay)
? Change() called when parameters change
On method for the (1) superclass, (2) a simple case (Square wave) and (3) how ramps are added at the beginning and end of any sound, with the fSubStatus (assigned in CSPlugin but applied in every NextBlock). The code for SetAttack & SetDecay (from CSPlugin - see next slide) does never need to be overwritten.
void CSPlugIn::SetAttack(void)
{
    double temp1, temp2;
pST->pBlock[0] = 0;
pST->pBlock[1] = 0;

    for (short i=1; i<64; ++i)
    {
        temp1 = pST->pBlock[i<<1];
temp2 = pST->pBlock[(i<<1)+1];
temp1 *= lfAttDec[i];
temp2 *= lfAttDec[i];
pST->pBlock[i<<1] = (short)temp1;
pST->pBlock[(i<<1)+1] = (short)temp2;
    }

    fSubStatus = SS_ON;
}

void CSPlugIn::SetDecay(void)
{
    double temp1, temp2;
    DWORD n = pST->dwBlockSmgs;

    for (short i=63; i>=0; --i)
    {
        temp1 = pST->pBlock[n-(i<<1)-1];
temp2 = pST->pBlock[n-(i<<1)];
temp1 *= lFAttDec[i];
temp2 *= lFAttDec[i];
pST->pBlock[n-(i<<1)-1] = (short)temp1;
pST->pBlock[n-(i<<1)] = (short)temp2;
    }

    fSubStatus = SS_OFF;
    Change(1,0);
}
Embedded LFOs

- The class CSTLfo includes 4 LFOs
- CSTLfo is a member of CSPlugin

Each FMOL instrument has 4 LFOs that can make oscillate any of the instrument 4 main parameters
FMOL Demo 2

1. Arpeggiator
2. LFOs
The main MFC applications shows the instrument list reading the configuration file (*.cfg)
## FMOL 1.0 main algorithms and their two primary parameters

In FMOL 1.0 (on-line 98) the 2+2+2+2 structure of the program forced the collaborative approach. FMOL 2.0 (on-line 2000) added new playing and overdub features and keys ➔ less collaboration!

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Type</th>
<th>Param. 1</th>
<th>Param. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BASIC OSCILLATORS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sine Wave</td>
<td>Generator</td>
<td>MS Pitch</td>
<td>Amplitude</td>
</tr>
<tr>
<td>Square Wave</td>
<td>Generator</td>
<td>MS Pitch</td>
<td>Amplitude</td>
</tr>
<tr>
<td>Sawtooth Wave</td>
<td>Generator</td>
<td>MS Pitch</td>
<td>Amplitude</td>
</tr>
<tr>
<td>Triangular Wave</td>
<td>Generator</td>
<td>MS Pitch</td>
<td>Amplitude</td>
</tr>
<tr>
<td>Pulse Train</td>
<td>Generator</td>
<td>MS Pitch</td>
<td>Amplitude</td>
</tr>
<tr>
<td>White Noise</td>
<td>Generator</td>
<td>NO</td>
<td>Amplitude</td>
</tr>
<tr>
<td><strong>SAMPLE PLAYERS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sample Player</td>
<td>Generator</td>
<td>MS Pitch</td>
<td>Amplitude</td>
</tr>
<tr>
<td>Waveta.Samp.Pl</td>
<td>Generator</td>
<td>Wav.Sound #</td>
<td>Amplitude</td>
</tr>
<tr>
<td>Scratch Samp.Pl.</td>
<td>Generator</td>
<td>Frequency</td>
<td>Amplitude</td>
</tr>
<tr>
<td><strong>BASIC MODULATORS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ring Modulation</td>
<td>Processor</td>
<td>MS Pitch</td>
<td>Modul %</td>
</tr>
<tr>
<td>Amplitude Modul.</td>
<td>Processor</td>
<td>MS Pitch</td>
<td>Modul %</td>
</tr>
<tr>
<td><strong>FEEDBACK ALGORITHMS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Karplus Strong</td>
<td>Generator</td>
<td>MS Pitch</td>
<td>Amplitude</td>
</tr>
<tr>
<td>Binary Modul.</td>
<td>Generator</td>
<td>Primary Pitch</td>
<td>Amplitude</td>
</tr>
<tr>
<td>K.Str. Processor</td>
<td>Processor</td>
<td>MS Pitch</td>
<td>NO</td>
</tr>
<tr>
<td>B.Mod.</td>
<td>Processor</td>
<td>Primary Pitch</td>
<td>NO</td>
</tr>
<tr>
<td>Processor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>LINEAR DIFFERENCE EQUATION FILTERS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>High-Low Filter</td>
<td>Processor</td>
<td>Cutoff Freq.</td>
<td>NO</td>
</tr>
<tr>
<td>Resonant Filter</td>
<td>Processor</td>
<td>Res. Freq.</td>
<td>Q</td>
</tr>
<tr>
<td>Comb Reverb</td>
<td>Processor</td>
<td>Feedback</td>
<td>Delay</td>
</tr>
<tr>
<td>Comb Eco</td>
<td>Processor</td>
<td>Feedback</td>
<td>Delay</td>
</tr>
<tr>
<td>Comb Delay</td>
<td>Processor</td>
<td>NO</td>
<td>Delay</td>
</tr>
<tr>
<td>Comb Comb</td>
<td>Processor</td>
<td>Gain</td>
<td>Delay</td>
</tr>
<tr>
<td><strong>OTHERS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pitch Shift</td>
<td>Processor</td>
<td>Pitch Shift</td>
<td>NO</td>
</tr>
<tr>
<td>Panning</td>
<td>Processor</td>
<td>Angle</td>
<td>NO</td>
</tr>
<tr>
<td>Line In</td>
<td>Generator</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>
The Generator & Processor classes
Constructing Instruments

? (Veure informació en el enunciat de la pràctica)
Bamboo GUI (2)
FMOL (2.0) keys allow to

- Sustain strings [1-6] (already in 1.0)
- Mute/unmute strings [Z-N] (already in 1.0)
- Record and retrieve mouse gestures [A,S] (already in 1.0)
- Record [D] and retrieve up to 8 snapshots [F-Ç]
- Change LFO wave [TAB,CAPS,SHIFT,CTRL]
- and more in FMOL 3.0 (private FMOL trio version) …
MIDI Control and Mappings
BYTE cChn = lpMIDIEvent->cStatus & 0x0F;

switch(lpMIDIEvent->cStatus & 0xF0)
{
    case MIDl__NOTE_ON:
    {
        BYTE cPitch = lpMIDIEvent->cData1;
        BYTE cVel = lpMIDIEvent->cData2;

        /* La altura de la nota ajusta el valor 0 del PlugIn 0
        La velocidad ajusta el valor 1 del PlugIn 0
        Si es 0 hace solo C1,
        Si es >0, hace CC0, CC1 y NOTE_ON */

        if (track[cChn].PlugIn[0])
        {
            if (cVel)
            { // NOTE ON
                track[cChn].PlugIn[0]->Change(1, cVel);
                track[cChn].PlugIn[0]->Change(0, cPitch);
                track[cChn].PlugIn[0]->On();
                LastOnState[cChn][0]=1;

                for (BYTE p=1; p<CPT_MAX_PLUGINS; ++p)
                    if (track[cChn].PlugIn[p]->fStartOn)
                        { track[cChn].PlugIn[p]->On();
                            LastOnState[cChn][p]=1;
                        }
            }
        }
    }
    else
    { //track[cChn].PlugIn[0]->Change(1,0);//NEW SJ
        //track[cChn].PlugIn[0]->Change(1,10);
        track[cChn].PlugIn[0]->fSubStatus=SS_DEC;
        //LastOnState[cChn][0]=0;
    }

    if (theSeq.fNetOn && theSeq.fStrings[cChn])
        midiOutShortMsg(theSeq.hMidiOut, MidiEventToCWord(lpMIDIEvent));
```cpp
void CSPlugIn::Change(BYTE nParam, LONG iValue)
{
    iParam[nParam] = iValue;

    // modifica las 4 posibles frecuencias de LFO
    // (vienen del mouse y han pasado por el MidiOut)
    if (nParam >= 4 && nParam < 8)
    {
        LFO.SetFreq(nParam - 4, iValue * 0.03937); // 5/127 = 0.03937 NB. 5 es la max freq.
        LFO.bRampStep[nParam - 4] = (signed char)iValue;
    }
    // modifica los 4 posibles rangos dinamicos de LFO
    else if (nParam >= 8 && nParam < 12)
    {
        LFO.SetAmp((BYTE)nParam - 8, (BYTE)iValue);
    }
    else if (nParam == 21)
    {
        bNPulsesOn = bPulsesOn = (BYTE)iValue;
        // Pulsos On de Source (si 0 no hay pulsos -> continuo)
    }
    else if (nParam == 22)
    {
        bNPulsesOff = bPulsesOff = (BYTE)iValue;
    }
    else if (nParam == 25)
    {
        SetOscilMode((BYTE)(iValue >> 4), (BYTE)(iValue & 0x0F));
    }
}```
Snapshots and Sequencing
File formats
Communication with the main MFC program
Speed Tricks
Additional Comments about Algorithms
Net features (non-real-time)
Collective Compositional Models

Options:

- Free
- Horizontal (i.e. “exquisite-corpses”)
- Vertical (multitrack-overdub)

New composers can add new layers, and process/distort existing ones
(they can also start new pieces from scratch)

Tree like database for storing uploaded pieces (synthesis engine and GUI are also very tightly related to this concept)

FMOL 2.0 database is now accessible from a web browser
http://teatredigital.fib.upc.es/DQ/eng/fmol/database.htm
FMOL behaves now like a browser plug-in (clicking a node downloads its scorefile and runs the program)

Register, login, upload, query... features are done from the HTML page

Lighter program

Database is accessible to curious surfers
FMOL 1.0 Authorship Tracking

1. At La Fura’s request, the SGAE sponsored the FMOL 1.0 project
2. Users registered (alias, e-mail...) within the system before being able to upload files
3. Scorefiles tracked their multiple authors identities
4. SGAE facilitated registration proceedings of selected and non-associate authors
5. SGAE tracked their rights
6. Several selected Internet authors received more than 1.000$ during F@ust 3.0 tour (1998-2000)
Musical & Social Implications

• FMOL 1.0 (18 Jan – 16 April 1998) : + 1,200 20”-compositions from + 50 composers
  ▪ 50 compositions selected and included in F@ust 3.0 soundtrack
  ▪ CD with “remixes” by 12 authors, late 1998

• FMOL 2.0 (1 Sept – 8 Oct 2000) : + 600 60”-compositions
  ▪ Selected ones constitute the electronic parts of the (otherwise orchestral) opera DQ (premiered at the Gran Teatre del Liceu of Barcelona, Nov. 2000)
Net features (real-time): Net-jamming

? Allow real-time multi-user interaction

? Constraints imposed by current internet technology
  - High latency will cause delays of 500-1000ms to music played on a computer and listened on another
  - Lack of a global reliable synchronism reference for the distributed synthesis engines

? Due to the timbrical nature of the synthesis engine, FMOL music has a high amount of robustness towards the internet constraints

(Atau Tanaka: Internet’s latency can be seen as cyberspace’s acoustic)
FMOL net-jam session server

- Asynchronous and multipoint real-time messaging server (probably based on Phil Burk’s Transjam protocol)
- Server listens to periodically incoming messages from the client
- At every frame (48 times/sec) each client sends the generated events to the server
- Real-time typical data rates of 60-180 bytes/second
- Server redistributes all the generated messages (except client’s own) to all the clients at same framerate
Open problem of net-jamming

• Each client listens to a slightly different mix
• The session server listens to yet another mix
• Which is the “good” one then, and which one (ones) should be stored?