Pure-Data
Una piattaforma per la sintesi e l'elaborazione audio in tempo reale

Federico Avanzini

<avanzini@dei.unipd.it>

http://www.dei.unipd.it/~avanzini

Corso di *Sistemi di elaborazione per la musica*

Sound modeling & design

Passi principali nello sviluppo di un modello

- Equazioni del modello
  - ⇝ Scelta del paradigma di modellizzazione, semplificazioni, . . .
- prototipazione dell'algoritmo
  - ⇝ Matlab: facilità di debugging, toolbox signal processing, funzioni di lettura/scrittura audio . . .
- design ad alto livello
  - ⇝ Esplorazione dello spazio dei parametri: serve un modello in real-time. Csound, Max/msp, PD . . .

Il “paradigma Max”

M. Puckette, “Max at Seventeen”, *Comp. Music J.* 26:4, pp. 31–43 (2002)

- Attualmente: *Max/MSP*, *jmax*, *Pd–PureData*
- Miller Puckette è o è stato il principale ideatore e programmatore di tutte e tre
- Nome scelto in onore di Max Mathews
- Sviluppo principale nel decennio 80–90, MIT Experimental Music Studio, IRCAM, UCSD
  - Fine ’80: versione commerciale (poi *Max/MSP*)
  - All’IRCAM *Max/FTS* (Faster Than Sound, poi *jmax*)
  - Dal ’94 alla UCSD Puckette sviluppa *Pd*

Il “paradigma Max”

Gestione del controllo real-time

- Collezione di task che girano in parallelo
- Timing dei task regolato da *wait function* e *trigger*
- I trigger possono essere esterni (ad es. master keyboard) o essere accesi da altri task
- Ad es. pianoforte e’ composto da 91 task (88 tasti + 3 pedali) triggerati dall’esecutore
- Idee che precedono la nascita del MIDI

L’interfaccia grafica –GUI– è costruita usando un linguaggio visuale di patching

Pd vs. *Max*

- Open Source

  ⇝ `http://www.crca.ucsd.edu/~msp/`

  Software e libro su sound synthesis
- Architettura “plugin-like”. Moduli aggiuntivi possono essere scritti, compilati, e caricati runtime.
- Numerose estensioni e librerie aggiuntive
  - In particolare Gem per rendering simultaneo audio e grafico
- Tool per la definizione e la gestione di tipi di dati strutturati

L'interfaccia di *pd*

- Finestra principale: controllo picchi, elaborazione audio on/off, menu principali
- Ampia ed istruttiva documentazione (*help* per i moduli nativi)
- Ambiente grafico: moduli elementari organizzati in *patch*
- Patch salvati in formato testuale, ma editati attraverso la GUI.
- Connessioni tra i moduli attraverso le rispettive *inlet* e *outlet*
- Due modi di lavoro: *edit* (elaborazione del patch) e *run* (interazione con oggetti)

Moduli, dati

Tipi di moduli (diverse forme di box):

- Oggetti (controllo/audio)
- Messaggi (interattivi in modo *run*)
- Numeri (interattivi in modo *run*), Simboli
- Commenti

Moduli, dati

Tipi di moduli (diverse forme di box):

- Oggetti (controllo/audio)
- Messaggi (interattivi in modo *run*)
- Numeri (interattivi in modo *run*), Simboli
- Commenti

Tipi di dati:

- Float (Max/msp usa anche int)
- Symbol (*Stop*, *Clear*, . . .)
- Bang (trigger di eventi)
- List

Messaggi

- Trasmessi all'arrivo di un `bang` o quando cliccati
- Piu' argomenti: liste
- Messaggi multipli separati da "`,`" diretti alla outlet
- Messaggi multipli separati da "`;`" direzione esplicitamente indicata
- Variabili di ambiente: $ <n> (riferite agli argomenti dei messaggi entranti)
- Comporre/scomporre liste di messaggi: oggetti `pack` e `unpack`
- Messaggi "wireless": oggetti `send` e `receive`

Flusso

Flusso di messaggi “istantaneo”: messaggi elaborati allo stesso tempo logico della loro generazione.

- Esecuzione dell’albero dei messaggi: strategia depth-first
- Prima inlet “calda” (trigger di eventi), le successive “fredde”
- In uscita da un modulo: outlet ordinate da destra verso sinistra
- Modulo `trigger` per gestire il flusso dei messaggi

Moduli audio

- Convenzione: moduli che elaborano segnali audio sono identificati da "$\sim$"
- Segnali audio: floating point 32bit, tra -1 e 1.

$\rightsquigarrow$ Di solito hardware limita a 16 o 24 bit

- I/O Audio: oggetti `adc~` e `dac~`
  $\rightsquigarrow$sempre in "coda" ad un patch!
- `Audio on/off`
  inizia/termina elaborazione a sample rate
  $\rightsquigarrow$da ricordare!
- Elaborazione audio inizia/termina anche con messaggi `pd dsp 1` e `pd dsp 0`

Gestione audio

- Ciclo di dsp: numero fissato di campioni
  $\rightsquigarrow$ Flag `-audiobuf`
  default 64
- All'interno di un patch si può usare l'oggetto audio `block~`
- Determina anche la window size di `fft~`
- Feedback loop tra moduli audio vengono considerati errori
- Si possono costruire con connessioni non-locali

$\Rightarrow$ ritardo di 1 audio buffer size

Gestione audio

- Scelta della frequenza di campionamento.
  ⇝Flag `-r`
  default 44.1 kHz
- Scelta canali I/O audio.
  ⇝Flag `-inchannels, -outchannels`
  default 2
- Scrittura/lettura: oggetti `writesf~`, `readsf~`, `soundfiler`, `tabwrite~`, `tabread4~` ...
- Oggetto `switch~` permette di spegnere e accendere audio per singoli sottopatch

Audio e messaggi

Elaborazione audio e di messaggi *interleaved*

- Segnali di controllo processati all'inizio di ogni ciclo di dsp. $\Rightarrow 64/44.1$ kHz $\simeq 1.45$ ms
- Cascata depth-first di messaggi elaborata completamente prima del nuovo tick di dsp.
- Messaggi mai passati durante un tick di dsp (determinismo)

$\rightsquigarrow$ Conversione tra segnali audio e di controllo: oggetti `sig~` e `snapshot~`

$\rightsquigarrow$ Problemi:

- Non si può fare controllo a sample-rate
- Non si possono usare eventi a livello audio (ad es. zero-crossing) come trigger

Oggetti per temporizzazione

- `delay <n>`
  ritarda un `bang` di n millisecondi.
- `timer`
  misura l'intervallo di tempo tra inlet destra e sinistra
- `pipe <n>`
  ritarda un messaggio di n millisecondi.
- `metro <n>`
  genera un `bang` ogni n millisecondi.
- `line[~]`
  genera inviluppi a rampa
- `qlist`
  implementa un semplice sequencer.

Oggetti condizionali

- `select`
  Confronta input con argomenti, produce `bang` se coincidono
- `route`
  Simile a `select`. Confronto sul primo argomentoma, gli argomenti successivi vengono passati condizionalmente
- `spigot`
  Inoltra o meno il messaggio in input a seconda del valore di un flag
- `moses`
  Inoltra input sulla outlet destra o sinistra a seconda che sia > o < del suo argomento

Altri oggetti utili

- Oggetti interattivi: sliders, dials, . . .
- Subpatch, astrazioni
- Connessioni audio “wireless”
  `send[~]` e `receive[~]`
- Oggetto `struct` per creazione e gestione di tipi di dati strutturati
- Stampa su standard output
  `print`
- Oggetti MIDI
  note, ctl, pgm, bend, touch, sysex, . . .
  Argomenti: numero di canale, ctl number

Estensioni

Molte estensioni scritte da altri autori

- Oggetti esterni

  ⇝ `http://www.pure-data.org`

- Librerie grafiche
  - GEM (Graphics Environment for Multimedia)
    Rendering 3D (si appoggia a OpenGL)
    ⇝`http://gem.iem.at`
  - XGUI (eXperimental Graphical User Interface)
    Semplice, per animazioni 2D
    ⇝`http://dh7.free.fr`
    Apparentemente non più supportata

Scrivere oggetti

External Una classe che non è *built-in*, e viene caricata a *runtime*. Una volta caricata in memoria non e' distinguibile da qualsiasi altro oggetto.

- PD scritto in `C`, ma orientato agli oggetti
  - Gnu C Compiler (`gcc`)
  - Visual-C++ 6.0
- Ottimo HowTo:
  ⇝`http://iem.kug.ac.at/pd/externals-HOWTO/`
- *Flext*, un C++ wrapper, libreria statica linkata all'external scritto in C++
  ⇝`http://www.parasitaere-kapazitaeten.net/Pd/ext/`

Scrivere oggetti

- External è una nuova classe
- Struct che definisce il suo dataspace
- Insieme di metodi che definiscono l'interfaccia con messaggi
- Costruttore/distruttore della classe

Nel caso di external di tipo "signal" (~)

- Metodo *dsp*, aggiunge una routine *perform* all'albero dsp di Pd
- Metodo *perform*, contiene il ciclo audio