Introduction to Pure Data

Pure Data

- Pure Data (Pd) is a visual signal dataflow programming language
- Designed to process sound and MIDI events. Has grown to process video and inputs from a variety of general purpose sensors
- Free alternative to MAX/MSP
- Runs on Linux, Macintosh, and Windows computers
Pure Data Console

Pure Data Test Signal
2.2 How does Pure Data work?

- Data flows between objects connected through cords or wires
- Thin cords carry message data; fat cords carry audio signals
- Objects take in data at inlets, and may send output to outlets; inlets and outlets appear as tabs at the edge of objects
- Types of object names:
  - Object names with ~: Process signals
  - Object names without ~: Process messages

Like a Moog™ modular synth, but digital
Patches

- A collection of objects wired together is a dataflow graph program or *patch*. Name is derived from analog electronic synthesizer modules connected together with patch cords.
- Patches are placed on a *canvas*.
- Patches are navigated by the PD interpreter depth first, from right to left (tries to go as deep as possible in a graph, processing the right-most branch first before a left branch).

Dataflow computation
Pure Data Software Architecture

• Pure Data consists of several programs:
  – pd (main engine): the interpreter, scheduler and audio engine
  – pd-gui: the interface you use to build Pure Data programs
  – pd-watchdog: monitors the main engine and gui, and will attempt to terminate unresponsive pd or pd-gui

LEGO® for sound and video?
Pd basic elements (1)

- Object types by function/appearance:
  - Object (processing)
  - Message (events)
  - GUI (user interaction)
  - Comment (documentation)

- Object types by topology:
  - Source (outlet only)
  - Sink (inlet only)
  - Filter (inlet and outlet)

- Atoms
  - Float, symbol or pointer

Pd basic elements (2)

- Patch = network or graph of data flows
- Connections/Streams:
  - Signals (continuous audio)
  - Messages (sporadic events)
    - Typ. control-oriented
    - Made up of multiple atoms
  - Data streams flow from top to bottom
- Audio I/O: [adc~], [dac~]
- Abstraction
- Editing and Interaction modes
Origins

• Miller S. Puckette
  – PhD in math from Harvard in 1986
  – Currently at CRCA (Center for Research in Computing and the Arts), UCSD

• IRCAM (FR) (1980s) Institut de Recherche et Coordination Acoustique/Musique
  – Was common for technicians to develop systems to support artists
  – Puckette developed Max to enable artists to do it themselves

• Pure Data (Puckette, 1996)
  – Design based on Max
  – Open source
  – New: graphical data structures

Pd’s philosophy and architecture

• Graphical literate programming:
  – Visual appearance of the patch is the program
  – DSP block diagrams are pseudocode
  – Comment objects can be placed anywhere on a patch

• Object-oriented/functional paradigm:
  – Classes and instantiation
  – Message passing
  – Outlets pass data to inlets

• Patch = document = program/subprogram

• Object network must be acyclic
  – But feedback (recirculation of data) is possible using special delay objects

• Data processed in real time
Other design features

• Patches can be edited while running
• Abstraction and re-use of patches
  – Ad hoc, one-off sub-patches
  – External patches (re-usable)
  – All look like objects from the outside
• Data structures: arrays, lists, graphics
• Entire libraries of “externals”
• Help file conventions make objects self-documenting

Implementation details

• All numbers are 32-bit floating-point
  – Audio h/w usually 16-24 bit integer precision
• Primitive objects typ. implemented in C
• Many audio APIs supported:
  – PortAudio, ASIO, MMIO, Core Audio, ALSA, OSS, JACK
• Audio rate processing runs continuously, in blocks
  – Usually driven by audio hardware clock
• Patches stored as plain text, describing topology and layout
• GUI is implemented using Tcl/Tk
Input/Output

- [print], [snapshot~]
- Load/save audio files to/from Pd arrays
- MIDI, OSC
- USB HID-class devices: [hid]
  - Keyboard, mouse, joystick, etc.
- Bluetooth (e.g. Wii™ remote control)
- Network (TCP or UDP)
  - Messages and uncompressed audio
  - Compressed audio, e.g. [oggcast~]
- Local IPC: pdsend/pdreceive
- COMEDI (Linux)
- Video capture

Subtleties

- Using messages for control of audio-rate data
  - Quantisation, low data rate (10-1000 Hz)
  - “Zipper noise”, clicks on toggling, noise
  - Add interpolation ([line], [line~], [vline~])
- Foldover distortion (sampling; Nyquist limit)
- Clipping on audio I/O
- NaN
- Platform-dependent features:
  - Graphics, codecs, tablets, etc.
Thinking in data flows

• Where are the loops? Conditionals? Variables? Assignment operations? Flow of control?

No visible flow of control

• Messages happen virtually simultaneously
• Audio signals processed continuously...
  – But in finite blocks
    • Power-of-2 samples in duration
    • Some latency (1.45 ms typ. @ 44.1 kHz)
    • Interleaved with message processing
• Implicit event loop, effectively
• However, it’s not stateless...
Some procedural counterparts

- Variables (typed)
  - [integer], [float] and [symbol]
  - Store received input values, emit when “banged”
- [until] for iteration
- Expressions
  - Network of objects (inverted expression tree)
  - [expr] (formula in a box)
- [spigot] conditionally enables data flow
- [moses], [select] and [route] resemble CASE or IF as functions
- Numeric messages can be interpreted as Booleans
- Objects for logical and relational operators
  - [&], [||], [==], [<], [<=], et al.

Certain tasks are easier in a data-flow environment

- Real-time, interactive tasks
- Function-oriented tasks
- Dealing with continuous signals (streams)
  - e.g. capture and playback, analysis and synthesis
- Event-driven stuff
  - External triggers, physical devices
  - Timed events (e.g. [metro] (metronome))
Going beyond sound

- 3D: GEM (OpenGL)
- Video capture, processing, compositing, etc.
  - PDP, PiDiP
- Physical modelling
- Physical transducers and other I/O

Light sensors

- Ordinary cadmium sulfide devices
- More light, less resistance
- Some analog pre-processing required before DAC
Drum pads

- Rubber practice pads
- Piezoelectric transducer element
- Suitable for use with Pd’s [bonk~] object
  - Takes audio signal as input
  - Detects “hits”
  - Outputs messages (including intensity)

Wii™ remote controller

- Buttons
- 3-axis accelerometer
- IR camera for tracking reference points
- Vibration
- Speaker
- LEDs
Potential Pd applications

• General signal processing
  – suitable for real-time, audio frequency work
• Data visualisation
• Simulation
  – (damped mass on spring demo)
• Prototyping
  – (simple flight sim in one patch)
• DIY groupware systems
• VJ (video jockey) performance
• Sound design
• Game development
• ...

Potential improvements

• Define aliases for object classes
• Attach comments to specific objects, groups or regions
• An on-demand signal snooper for testing and troubleshooting
• Macro capability?
• Hierarchical namespace for objects?
• UI refinements
Conclusions

• Modularity and generality are great strengths
  – Need abstractions to manage complexity
  – Libraries are important
  – Be willing to DIY
• Literate graphical programming has benefits
• “Everything is a function” works well for audio
• Ability to edit running patches is useful
• Invisible connections ([send]/[receive], etc.)?
  – Undermine graphical approach
  – but avoid clutter on complex graphs