Pure Data - Visual data flow programming

Martin Boonk
University of Paderborn
mboonk@mail.uni-paderborn.de

December, 23 2013

Abstract

Pure Data is a visual data flow oriented programming language. Its audio processing capabilities are useful to provide a simulation function for the COSMIC system. This document describes the essential language elements and the underlying working principles of Pure Data focusing on the usage as simulation back-end.

1 Introduction

Computer audio processing can be done in various ways. The synthethization of sounds is a wide field and one of the more intuitive approaches is the modeling of data flow in a graphical interface. An example is the open source Pure Data language which allows users without a background in computer science or training in classical programming paradigms to create digital signal processing applications. Programs are created using a graphical interface and have an intuitive run time behavior. This work describes Pure Data in order to facilitate the development of a simulation back-end for the COSMIC project. After a short history of computer music languages in the following section 2 an introduction into the essential usage and internal working principles of Pure Data is given in section 3. The commercially available system Max/MSP by Cycling'74 has a common history with Pure Data and is widely used by sound engineers and artists. A brief overview on similarities and differences that influence the compatibility and properties for development is given in chapter 4. The conclusion (5) evaluates the feasibility as a simulation back-end base and contains hints for the actual development process.

2 Computer music languages

Since personal computers provide enough processing power for real-time audio generation and manipulation, several specialized languages emerged. The first purely synthetically produced computer music was created by Max Mathews in 1957. The software

"Music 1" led to the development of multiple successors and spawned the creation of other languages like CSound by Barry Vercoe. [6, p. 16]

Following these, Miller Puckette, at that time working at IRCAM [1], developed a visual language named "Max" which was interpreted using a digital signal processing program called "FTS"[2]. The execution of signal processing tasks depended on specialized hardware added to a NeXT computer which was then denominated as ISPW [3]. Pure Data was created by Miller Puckette as rewrite of the Max/FTS program to be released as open source software. The commercial variant of Max, produced by the company Opcode, was sold as a specialized tool for MIDI control and was missing the signal processing capabilities. [6, p. 17] Those functionalities were then added to the Opcode Max version as an add-on named "MAX Signal Processing" created by David Zicarelli who was inspired by Pure Data. [2, p. 69f] The resulting Max/MSP system was further evolved into tools that are today sold and developed targeting the creative sector by the company Cycling'74.

3 Visual data flow programming in Pure Data

Commonly used imperative programming languages like C or Java are used to describe programs as sequential operations. In contrast to those and similar languages, data flow oriented languages such as Pure Data allow the programmer to create programs solely by constructing a graph that depicts the data flow. Visual programmings has its sources in the patch fields analog devices and depictions of signal processing algorithms [6, p. 14].

Pure Data programs are represented as patches which consist of connected elements. As not to limit the user, the initial state of such a patch is completely empty. The user or programmer creates a program by arranging and connecting boxes into a patch. Every patch opened by Pure Data can be in two modes. Changes to a patch are done in edit mode, while the behavior in run mode is locked down to controlling the elements, that offer this functionality. Control and signal calculations are performed independent of the active mode.

3.1 Language elements

The following sections present the different box types, connection possibilities and visual appearance based on the official Pure Data documentation [3, ch. 2.1, 2.2] There are essentially four types of boxes used in Pure Data: object, message, GUI and comment boxes. While all of these carry information such as parameters as text divided by spaces called atoms, only the first three types have an influence on the resulting data and signals. Comment boxes provide solely the possibility to create text in a patch without influencing the results.

[1]Institut de Recherche et Coordination Acoustique/Musique

[2]Faster than sound

[3]IRCAM Signal Processing Workstation

Atoms have two different types, either number or symbol. Every atom that can be parsed as a number will be interpreted as one. Allowed notations for numbers have an optional decimal point and an exponential notation like $1.23e2$ for 123 is allowed. If an atom is not parsable as a number, it will be interpreted as a symbol. The symbols are case sensitive.

Two different kinds of connection exist in Pure Data: control and signal. To create a connection two ports must be linked together. They are called inlets or outlets depending on data flow direction. The visual appearance of these ports shows the type of data they accept. Ports are small rectangles on the top of a box for inlets and on the bottom for outlets. The rectangle is filled if a port accepts or delivers signal data. Control outlets can be connected directly to signal inlets because Pure Data implicitly converts the control data into signal data. The opposite is only possible by using explicit conversion objects. Control data links are depicted by thinly drawn lines while signal connections are being drawn as a thicker line.

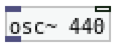

Figure 1: Signal object box

Object blocks are the functional units of the Pure Data language and are displayed as rectangles on the canvas. The first atom in any object blocks text denotes the type of this instance while all following atoms are used as parameters to the function carried out by the object. Objects that have signal ports and as such are capable of some sort of computation on signal data are denoted by a tilde character in their name. The concrete number and types of ports are determined by the implementation of the object, that is selected by the entered type and parameter atoms. They can have an arbitrary number of inputs at which they receive control data or signal data. The results of any object box are delivered to its outputs, which again exist in an arbitrary number. As an example figure 1 is given, which presents an object box of type “osc~” with a single parameter atom which contains the number 440. Its type atom ends with a tilde, by convention depicting a box containing signal processing capabilities.

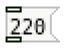

Figure 2: Message box containing a number atom

Message blocks send information to other blocks. Their visual representation is a rectangular shape with an indentation at the right-hand side (cf. figure 2). After creation they contain a message consisting of atoms, that will be send when the block is triggered either by an event such as a click in the user interface or by receiving a message. Almost all other messages can be used as this trigger while there are some special messages, identified by their first atom, that can change the stored message itself (cf. [5])

Boxes allowing control over their contained data at runtime are so called “GUI” boxes. Those graphical user interface elements can have many visual appearances and function-

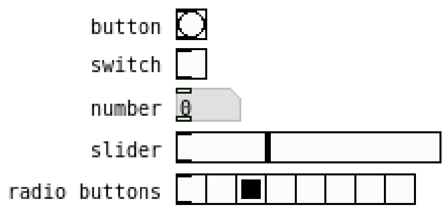

Figure 3: Exemplary selection of GUI boxes

alities (cf. figure 3 for examples) while their main distinction from other object boxes is the ability to visualize the currently stored value at runtime. The number box's internal value for example can be manipulated by clicking and dragging the mouse on the visual representation when the patch is in run mode.

3.2 Example patch

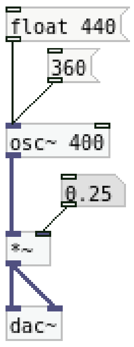

Figure 4: Simple patch generating sine wave audio output

A simple patch showcasing some of the previously described features is shown in figure 4. The end result is a sine wave output to the sound hardware using 3 different frequencies and a modified amplitude. The "osc~" object is a sine oscillator initialized with a frequency of 400 Hz as a creation argument. Connected to its first inlet are 2 message boxes, that can be clicked to change the frequency to 440 respective 360 Hz. For convenience messages containing a number as the first atom instead of a symbol have the "float" selector automatically added making both used notations equivalent. The sine signal leaves the oscillator at its first outlet and enters a signal multiplication object ("*~") that has no parameter setting an initial value. Hence the patch does not directly after loading create audible sound since the default value of the number box connected to the second inlet is 0. Figure 4 shows an adjusted value of 0.25. By clicking and dragging using the "Shift"-key the number box's value can be adjusted with a step size of 0.01. Just clicking and dragging implies a step size of 1. Since the signal oscillates between the values -1 and 1 a multiplication with 1 means full amplitude. After this the signal is copied by using 2 connections into the "dac~" object. This box provides an interface to the sound system of the executing computer and defaults to providing 2 channels, usually mapped to the left and right channel of a stereo output.

3.3 Internal working principles

The following section is mainly based on the official Pure Data manual [3, ch. 2] and will present the internal processes used to calculate the results of a patch.

Pure Data is a modular system that consists of internals, externals and libraries. Internals denote built-in primitives or objects, that can directly be used for the creation of patches. Additional objects can be delivered as externals. These are binaries that are loaded at runtime and can not be distinguished from internals once they are in memory. Any number of externals can be compiled into one binary that then is denoted as a library. [7, p. 4]

Essentially the patch is divided into a control and a signal part for the calculations. Those parts are treated as if they were subgraphs and are independently computed in an interleaved fashion. The signal processing on a block of the configured size (the default block size is 64 samples and can be changed globally or on a per patch basis) is denominated as a DSP-tick. Those ticks run atomically when executed and are not influenced by parameter changes. All messages that are triggered during a DSP-tick are processed sequentially and completely before the next tick calculation is started. [3, p. 2.5.1]

3.3.1 Audio processing

Audio signals are stored as 32-bit floating point data which is interpreted as values between -1 and 1 and for each output clipped to that range [3, ch. 2.4.1].

Tilde objects, as described in section 3.1, carry out audio computations. The scheduler sorts those objects into a linear list which is then processed in blocks of samples. This procedure effectively prohibits loops in the signal graph to prevent the object list becoming infinitely long. [3, p. 2.4.2]

3.3.2 Control processing and messaging

Messages in Pure Data consist of at least one symbol, called the selector, and an arbitrary number of argument atoms. The selector of a message is checked against the class of the receiving block. Each class has the ability to receive a fixed set of selectors it can process. [3, ch. 2.6.3]

The behavior triggered by received messages depends on the type of inlet: it can be either a “hot” or a “cold” inlet. Messages to hot inlets by convention result in new messages sent to the outlets of the object in question. Cold inlets receiving a message change the objects inner state to reflect the received data. Objects send their outgoing messages sequentially and ordered by outlet position beginning at the right most one. [3, ch. 2.3.3]

The order in which messages are delivered is determined by a depth first method. Each message received by an object may trigger the sending of new messages. This results in a tree of messages which is then processed depth first. The processing order of children on the same level is indeterminate. Practically this depends on the creation order of the connections while building the patch. Since loops including multiple hot inlets would create a deadlock situation, such constructs need to be created using delay objects to break the potential infinite loop for the internal scheduler. [3, ch. 2.3.2]

This behavior directly dictates the rules for message merging. If multiple messages arrive at one inlet on the same logical time, there must be some decision how to merge those control streams into one. The right to left order when sending messages from outputs combined with the depth first processing determines the merging for the most cases. The connection creation orders influence on the processing order can create difficulties in result prediction based on the visual patch appearance because the creation order is not visualized.

3.3.3 File format

Although the used file format is officially undocumented there exists an unofficial specification to be found at the projects home page [1]. The following description of the file format gives a short overview of the components and used structure. The Pure Data patch file is a custom text-file format that consists of a set of records. Each record has the form: `#[data];\n\r` where the `[data]` portion is expanded to form a record like `#[type] [element] [parameter]*;\n\r`.

type A single character, defining the record to describe an object (type=X), a new window (type=N) or array data (type=A).

element The name of the Pure Data object to create. An exception are the "connect" records which denote links between two objects.

parameter Parameters needed by the object. Commonly used ones for GUI elements are position and size as integer values.

Records with the type "X" that are not "connect" elements are numbered in order of creation. These virtual numbers are used in the `#X connect` records to reference the objects.

3.3.4 Object implementation

The Pure Data system is written in C. While C is not object oriented, some concept of a class must be defined to represent Pure Datas inherent object oriented structures. As it is common practice to refer to Pure Data object types as "classes", this terminology will be used throughout this work. It therefore denotes a combination of data and methods, that manipulate this data. Pure Data provides an interface for the definition of classes, that is used for creation and instantiation.

At the time a library, for example named “custom_lib”, is loaded, Pure Data invokes a method `custom_lib_setup()` which sets up the included classes by calling the corresponding class setup methods. Those define a name, constructor, inlets and outlets instantiated objects of that particular class should have. The constructor initializes the data space while the method space is constructed by calling methods like `class_addbang()` or `class_addmethod()` to give the class a means to react on messages with fitting selector atoms. [7, ch. 4]

Signal externals performing digital signal processing tasks are normal Pure Data objects using additional methods. The recognition of signal classes works by sending a message with the selector “dsp” to every object. As a positive reaction to such a message the API method `dsp_add()` is called and declares the implementing method for the signal processing as well as the signals data locations. Each signal is handed over as an array of samples. The actual computation on this signal data is then in the hands of the developer. Usually an iterative approach manipulating samples one at a time is chosen. [7, ch. 5]

4 Compatibility with Cycling’74s Max/MSP

The common heritage of Pure Data and Max (cf. Chapter 2) is most prominent in the concepts of use and thus the visual interfaces of both languages. The creation of programs is based on graphically connecting boxes to depict a data flow between functional units.

While some knowledge of one system can be applied to the other there is no true compatibility by design. Patches designed in one system are not automatically portable. [4, p. 15]

Despite this there are several tools and best practices in existence, that facilitate the process of porting patches between Pure Data and Max.

- Clones of objects for the respectively other language (an example is the pdj~ object for Pure Data which is a functional clone of the mxj~ Max object).
- A selection of Max objects is provided in the Pure Data library “cyclone”.
- Max patches in the .pat/.help./.mxt formats can be opened and saved by Pure Data. While newer Versions of Max still work with those file formats they implemented a new JSON-based format using the .maxpat extension which can not yet be used directly in Pure Data.
- Since the message ordering in Max differs from Pure Data, the usage of objects creating an identical behavior in both languages is advisable. One such object is the “trigger” object which copies messages and sends them in a predetermined order.
- As Pure Data stores all number values internally using the float data type, some attention must be used if a Max patches behavior depends on the distinction between integer and float values.

5 Conclusion

The intuitive and relatively simple to use programming environment provided by software like Pure Data and Max is extremely useful to model audio systems of all sorts. For the COSMIC system this provides a means to simulate the runtime behavior of the hardware in a good approximation using proven and widely used software. Additionally an efficient way of prototyping hardware sound components before actually beginning implementation is provided by such software systems. This facilitates the assessment of necessary design efforts.

The open source nature of the Pure Data project and the resulting wide platform compatibility is a major advantage over the commercial Max version. Supporting all major operating systems and multiple sound APIs for each supported platform Pure Data is less limited than Max which currently supports Microsoft Windows and Apple Mac OS. Since the target is the creation of a simulation environment whose concrete feature set is in a state of change and development, the freedom offered by the possibility to adapt the source code of Pure Data if necessary is crucial for the project. Additionally the possibility to redistribute Pure Data free of charge presents a way of delivering a working simulation back-end without depending on an only commercially available tool with considerable cost. The support and existing documentation for Max is far superior. While this could be a major concern for a user, it does not impact the development in the context of COSMIC.

Using Pure Data for the creation of a simulation back-end can be accomplished by generation of patches in the relatively simple text-based format it uses. This gives the developer the possibility to take a generated simulation patch set and inspect it directly using the Pure Data interface.

The COSMIC system sound generation implementation has several similarities with the Pure Data way of calculating patch results. Pure Data classes map directly to the concept of sound components. Despite using a different programming interface for the components, the internal processing workings of especially software based sound components are very similar to the Pure Data implementation. Since COSMIC allows components to be developed on an FPGA-board using gate level hardware, the advantages of hardware can be leveraged for the computational intensive digital signal processing tasks. While the implementation of hardware based components has no direct equivalent in Pure Data they still can be simulated. The universality delivered by the Pure Data language allows for complex simulations only limited by processing power.

To stay compatible with the widely used distributions of Pure Data mostly the already extensive pool of publicly available objects and libraries should be used to create the simulation environment. The same holds for changing the existing source code of Pure Data. Creating custom externals stays a possibility if certain simulation tasks can not easily be achieved by using existing objects but should be avoided.

References

[1] Simon Asselbergs and Tjeerd Sietsma. *Unofficial PD v0.37 fileformat specification*. URL: `http://puredata.info/docs/developer/PdFileFormat` (visited on 12/20/2013).

[2] R. T. Dean. *Hyperimprovisation: Computer-interactive Sound Improvisation*. A-R Editions, June 2003. ISBN: 978-0895795083.

[3] Miller S. Puckette. *Pd Documentation*. URL: `http://www.crca.ucsd.edu/~msp/software.html` (visited on 12/21/2013).

[4] Miller S. Puckette. *The Theory and Technique of Electronic Music*. May 2007. ISBN: 978-981-270-077-3. URL: `http://www.crca.ucsd.edu/~msp/techniques/v0.11/book-html/` (visited on 12/20/2013).

[5] *Pure Data 0.43.1*. Help file: Pure Data/5.reference/message-help.pd.

[6] Fränk Zimmer. *bang. Pure Data*. Wolke Verlag, Dec. 2005. ISBN: 3-936000-37-9.

[7] Johannes M. Zmölnig. *HOWTO write an External for puredata*. Sept. 13, 2001. URL: `http://pdstatic.iem.at/externals-HOWTO/` (visited on 12/22/2013).

List of Figures