

# Design and Concepts of GridFlow

Mathieu Bouchard <matju@artengine.ca>

September 24th, 2004 (2nd draft)

## Abstract

GridFlow has been developed since 2001 with a dual goal: on one side, of making video processing available to jMax; and on the other side, of applying powerful generic array-oriented techniques to video processing. In this paper we will explore the notions that are characteristic to GridFlow and the motivations behind the introduction of those notions into GridFlow's model.

## Contents

|          |                                    |          |
|----------|------------------------------------|----------|
| <b>1</b> | <b>Grid</b>                        | <b>2</b> |
| 1.1      | Floats . . . . .                   | 2        |
| 1.2      | Lists . . . . .                    | 2        |
| 1.3      | Signals . . . . .                  | 2        |
| 1.4      | Grids . . . . .                    | 2        |
| <b>2</b> | <b>Flow</b>                        | <b>3</b> |
| 2.1      | Message . . . . .                  | 3        |
| 2.2      | Packet . . . . .                   | 3        |
| <b>3</b> | <b>Polymorphisms</b>               | <b>4</b> |
| 3.1      | Dispatch on Type . . . . .         | 4        |
| 3.2      | Shape Polymorphism . . . . .       | 4        |
| 3.3      | Protocol Polymorphism . . . . .    | 4        |
| 3.4      | Isomorphism . . . . .              | 5        |
| <b>4</b> | <b>#out</b>                        | <b>6</b> |
| 4.1      | Pixel . . . . .                    | 6        |
| 4.2      | Picture . . . . .                  | 6        |
| 4.3      | Video . . . . .                    | 7        |
| <b>5</b> | <b>Ruby</b>                        | <b>7</b> |
| 5.1      | jMax . . . . .                     | 7        |
| 5.2      | Automated Testing . . . . .        | 8        |
| 5.3      | Development of Externals . . . . . | 8        |

# 1 Grid

## 1.1 Floats

PureData's basic computational objects involve messages of type *float*. For example, the `[+]` object has one right-inlet method that stores a float for later use, and one left-inlet method that computes the sum of a float with the stored float, and outputs that result on the outlet. It's technically possible to add vectors using `[+]`, but it doesn't mean it's fun to do so.

## 1.2 Lists

A list is a message that may contain any number of elements. In that sense it is useful to represent a collection of values as a single entity. However there are not many operations that are available on lists. Can you name the object required to add two lists (of floats) together? Also, lists are still made of atoms so there is overhead associated to each element as for each element it must be checked that it is a float upon reading, and for each element it must be specified that it is a float upon writing. It's much less overhead than routing many single-float messages but it's far from efficient in large-scale computations.

## 1.3 Signals

A signal is a non-message way of synchronously sending a stream of floats from object to object. It has a fixed speed and a semi-fixed block size. The `[+~]` object adds two signals together in a quite efficient way. Signals are normally used for audio, but in VideoDSP[3] they are used for RGBA video.

In that kind of implementation of video, the framerate, width, height and color model are all fixed. It can be likened to common TV-station and video-edition hardware that does everything in one standard format, be it NTSC or PAL. With that kind of hardware, each machine does its own real-time processing so naturally when adding more effects you add more hardware. In the case of PureData all effects are sharing a same CPU and it's technically easy to deal with several video specs and convert between them, so there would be a benefit to handling several different data-rates at once, so that computationally expensive effects may run at lower data-rates while other things still are handled at high data-rates at once.

In the case of audio the same argument could be made, so there would be an advantage for Pd to handle 11 and 22 and 44 kHz signals all at once, but the issue is not nearly as much exacerbated as it is in video, which is usually of much higher bandwidth.

## 1.4 Grids

A grid is a message that may contain any number of elements, and that number can be factored into any number of dimensions. For example, a (4,4)-grid is a

four-by-four “matrix” of numbers, and has 16 elements, just like a (1,16)-grid (row matrix) or (16,1)-grid (column matrix) or (2,2,2,2)-grid (a 4-dimensional data structure). A grid has a uniform element type just like a signal, for efficiency. The numbers within parentheses are called the *shape* of the grid (or the *dim*). The operators may behave differently depending on the shapes of the incoming grid. For example, `[#fold +]` sums together elements along the last dimension, so that a (4,4)-grid is transformed to a (4)-grid, but a (2,2,2,2)-grid is transformed to a (2,2,2)-grid.

The main inspiration for GridFlow’s Grids has been APL[5]. Many popular languages support some kind of multidimensional array structure (Fortran, BASIC, CommonLISP) or uniform nested arrays (C/C++, ML, Haskell), but what makes APL special, and what makes the GridFlow-APL link so strong, is that APL defines high-level operators that operate on whole arrays instead of using bunches of for-loops. For example, `[#fold +]` is quite directly the same as APL’s `+/` operator, And `[#inner]` is quite directly the same as APL’s `+.×` operator. Among the early names of GridFlow was “Visual APL”, though it was more of a nickname than anything else.

## 2 Flow

### 2.1 Message

A message is the basic unit of communication and behaviour in Pd. It is useful that a grid is considered to be exactly one message because that way many objects like `[spigot]` and `[shunt]` and `[bang]` may be used with grids in the same way as they are used for other kinds of messages. For this reason, GridFlow, which used to consider a grid as a certain sequence of messages, was changed (in 0.6.5) so that a grid is one message.

### 2.2 Packet

A grid is nevertheless often broken down into several packets to benefit from RAM cache locality, because computers work faster if they operate on small data sets at once. This is not the same thing as a packet in PDP. This is quite similar to Pd’s signal block sizes except that it doesn’t have to do with latency reduction and that GF’s packet sizes are much more variable, depending on how one object likes to produce its data and how the object that receives it likes to receive it.

This is the reason why GridFlow doesn’t completely work as expected regarding message order. GridFlow respects the concepts of hot/cold inlets in most cases, but due to packet order the data may not have been received when it is needed for processing. Currently there is a workaround using `[#finished]` and `[#store]` and `[t a a]`, but it would be better if regular message semantics would be fully respected, even if it means queueing a bunch of packets in a left inlet. It is expected that version 0.8 or 0.9 would solve this issue.

## 3 Polymorphisms

Polymorphism is when several different things are called the same way and then differentiated using context. This is an extremely useful concept in programming (and linguistics). Here's my personal classification of polymorphisms which I elaborated while working with Pd/GridFlow.

### 3.1 Dispatch on Type

This is when, for example, a same object can react differently to different kinds of data. This may be done in several ways, e.g.:

1. finding the appropriate method in a class. e.g. float vs symbol vs list messages
2. finding the appropriate class for a method name (if several object-classes share methods together)
3. or using a branch, such as **if** or **switch** in C/C++ or **[t]** or **[route]** in an abstraction.
4. or using PDP or GEM (especially greyscale vs RGB vs YUV video)

In the case of GridFlow this happens mostly around the issue of supporting several number types. GridFlow is usually compiled with support for four kinds of integers and two kinds of floats. Many GridFlow objects have six versions of the same code selected at runtime using a combination of C function pointers and C++ template types.

### 3.2 Shape Polymorphism

This is when one object class works with many different shapes of grids. This way **[# +]** can add floats the same way it adds vectors, matrices, pictures. This is also when one object class reacts differently to different shapes of grids, e.g. the **[fold +]** example above. This is the word for the fact that an algorithm isn't hardwired to a fixed number of elements, or often not even a fixed number of dimensions. This can also be applied to any type other than GridFlow grids as long as some concept of shape exists in it and that sometimes the shapes may be sometimes considered as subtypes and sometimes not. However I can't think of any example except matrices. If signals had multiple data rates and/or could be stereo, then that would be a good example.

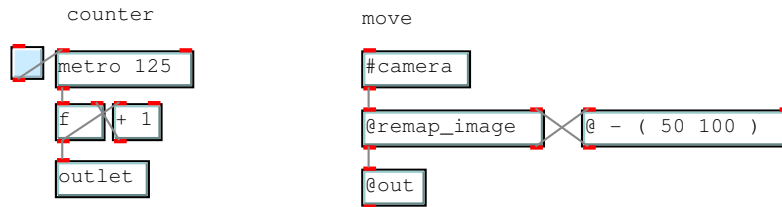
### 3.3 Protocol Polymorphism

This is when one object can be used instead of another because both objects behave in sufficiently similar ways. A set of conventions for the behaviour of objects is often called a *protocol* (Smalltalk) or an *interface* (Java) or a *contract* (Eiffel) or a *(completely) abstract class* (C++). The expectations of the object

are also called *preconditions* and the guarantees offered by the object are also called *postconditions*.

For example, a precondition might be that a *play* message works only if a movie has been open first, and a postcondition is that the movie will be playing. Another good example of postcondition would be the right-to-left message order: **[unpack]** guarantees that it produces messages right-to-left when the hot inlet is activated, and it guarantees that values in cold inlets get remembered.

Bidirectional protocols between two objects can be semantically much richer than unidirectional ones. This is why the side-cross-wiring pattern is so useful. In GridFlow the **[#remap\_image]** abstraction is an impressive example of this, but everyone's made a counter in Pd before too. Patterns involving more than two objects can usually be analysed in terms of just two objects by using subpatchers (or imagining them).



### 3.4 Isomorphism

This is when two operations are equivalent in some way.

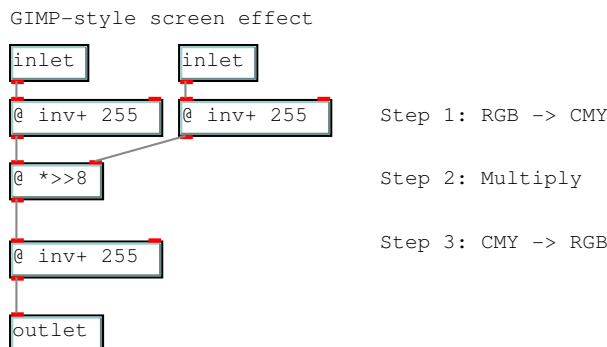
For example, [7]GIMP's *multiply* and *screen* layering effects are isomorphic, because to make one effect you can negate both layers, apply the other effect, and negate the result. Here, the negation of colors is called the *isomorphism*, that is, a conversion used to show the equivalence between the two operations.

A hue shift may be performed on a YUV picture by making a rotation in the UV plane, about its centre (128,128). A hue shift on a RGB picture can be defined by wrapping the rotation using conversions to YUV and back.

Another good example is that  $\log(xy) = \log(x) + \log(y)$ . This means that adding values in a log-space is like multiplying them in a lin-space. This is the characteristic relationship between Hertz frequencies and MIDI (semitone) frequencies, and also between amplitude in Volts and amplitude in dB.

In many circumstances, the conversion between the two systems can also be nothing at all. In such a case, what changes is the way we understand the data. For example, **[# +]** can mean simply adding numbers, or it can mean additive synthesis of pictures, or it can mean translating a polygon or adding rotation

angles. It's just a matter of what is the meaning attached to the numbers in a given context. In GridFlow, this kind of isomorphism is crucial and is an enormous boost of usefulness. The fact that there is not a type tag attached to each grid allows a same object to support both polygons and pictures without inserting any converter objects and without writing more C++ code.



## 4 #out

### 4.1 Pixel

A pixel is one or several values denoting usually a colour. For example the `[#color]` object produces a pixel according to three slider values and provides a preview of the colour, assuming a RGB interpretation of those values. A pixel could also be interpreted into colorspace as diverse as RGBA, YUV, YUVA, Y (greyscale), YA. The ranges of values may also be diverse, for example RGB could be done with a maximum value of 65535 instead of 255, or Y can also be done with a maximum value of 1, or YUV can be done with UV in the 0..255 range as well as the -128..+127 range. All those distinctions are essentially up to the user of GridFlow, except for some specialized objects that expect some specific number ranges, usually RGB from 0 to 255.

### 4.2 Picture

The `[#out]` and `[#peephole]` objects expect a (rows,columns,channels)-grid and accordingly the `[#in]` and `[#camera]` objects produce such grids. They deal with one or more of the six aforementioned color spaces, in the 0..255 range. Those few objects are the main reason GridFlow can be called a video plugin because otherwise it doesn't have much to do with video. The rest is a matter of interpreting numbers and vectors as colours, and grid dimensions as spatial dimensions, which is something the I/O objects deal with but, for example, `[#+]` and `[#inner]` don't know nor care about, although those latter objects are extremely useful in manipulating video. In fact, most video-specific objects are

actually abstractions made using non-video-specific objects (eg: `[#inner]` is central to the implementation of `[#rgb_to_yuv]`).

In math terms, a picture could be defined as a function:  $(row, column) \rightarrow (red, green, blue)$ . A grid could also be defined as a function:  $(...) \rightarrow number$ . The problem with matching those two is that the former has several outputs; but the former may be transformed such that it has only one output, by adding an input dimension that selects which kind of output is wanted, which turns it into  $(row, column, channel) \rightarrow number$  and is then compatible with the definition of grid, so that, in this model of things, a picture is a kind of grid. A model supporting multiple output values was later implemented under the name Jitter[6], in which the output dimensions are called *planes*, and a grid is called a *matrix*. GridFlow's model might be more versatile because more uniformity means more polymorphisms for free, but it doesn't necessarily match with the beginner's view of channels as *not* being a dimension.

### 4.3 Video

As for most video plugins, one video frame in GridFlow is a message, so that the framerate of the video may be controlled using standard objects such as `[metro 33.3667]`. (yeah, that's it.)

## 5 Ruby

GridFlow is not really a plugin for Pd. You may have noticed that `grid-flow.pd_linux` is a very small file. Actually that is an external that could be better called "rubyext" just like there is "pyext". It enables one to create a Pd object class using the Ruby language, either by writing the object in the Ruby language, or by writing it in another language that can be interfaced to Ruby. It was introduced in GridFlow for several reasons.

### 5.1 jMax

For two years long I have tried making GridFlow work in both Pd[1] and jMax[2]. This is now officially terminated, GridFlow 0.8 not supporting jMax anymore. When we began working with jMax it was not yet clear that Pd was going to become a better choice. When I started porting to Ruby and Pd, it was not clear that the switching to Pd was going to be worth the trouble of switching. At the end of 2003 we finally got used to the idea enough that within a few weeks we were not using jMax anymore despite our background of several years of patching in jMax. Ruby was the language I was writing the most code in at the time, and although combining Ruby with live video seemed like a very wild thing to me in 2001, GridFlow was rewritten around Ruby during the summer of 2002.

## 5.2 Automated Testing

Ruby is an excellent language for conducting automated testing of functionality. This is how GridFlow’s “make test” works and, although GridFlow doesn’t have a test suite nearly as complete as is mandated by Extreme Programming and Pragmatic Programming methodologies, “make test” catches many bugs well before they reach a release or even the CVS. It allows for the internals of GridFlow to be improved with much less effort. However there are many cases in which it doesn’t help:

1. Testing GUI objects is complex without human intervention.
2. Testing an operation automatically makes the most sense for operations that are hard to do but easy to verify. For many operations it’s quite complicated to make sure the right thing has been done without expressing it in terms of the operation you are trying to test, and without using the same code that you are trying to test.
3. Habits are hard to overcome. It takes discipline to write the tests before the implementation, or even to write tests at all.
4. Bugs in special cases in the implementation can be overlooked in the tests, especially when the tests are created in relationship to the specifications rather than with a specific implementation in mind.

In spite of those drawbacks, automated testing is a successful tool in the making of GridFlow, to the extent that it can be applied, and to the extent that it is applied. Ruby is successful for automated testing because it is a very dynamic language (more so than C or Pd) and because there is a culture of automated testing in the Ruby community.

It would be possible to make those tests from within Pd instead in the future, but not much work has been done in that direction yet, as this is very much not in the culture of the Pd community. Testing functionality from within a Pd patch could be the topic of a whole paper. Is there any object that can trap an error message generated by another object?

## 5.3 Development of Externals

Ruby is an extremely concise language, usually more so than either Perl or Python, although Ruby is in the same general category as those two. That concision has been helpful many times. For example, many jMax objects had to be cloned so that they had compatible equivalents in Pd. That meant writing an external for each object we needed. The typical process of porting our patches (as well as our patching habits) from Pd to jMax was like this:

1. Alexandre asks me (on IRC) how to do such and such in Pd.
2. Mathieu doesn’t know any equivalent either because he doesn’t know the external or abstraction to do it, or because it doesn’t exist.

3. Mathieu answers to Alexandre by one line of code (a long one, but usually less than 200 bytes).
4. Alexandre tries it and often it works although Mathieu hasn't even tested it.

This process also appears in many circumstances other than patch-porting, so it's still very relevant. About one quarter of GridFlow is written in Ruby, just counting the bytes of code; but usually a given problem will have a shorter solution in Ruby than in C++, so effectively it is more than just a quarter. Notable Ruby-implemented objects are `[fps]` and `[#pack]` and `[#rotate]` and `[#shunt]` and `[#print]`.

## References

- [1] *PureData* by Miller S Puckette et al., since 1997, <http://puredata.info/> <http://crca.ucsd.edu/>
- [2] *jMax* by Francois Dechelle, Norbert Schnell et al., 1997-2003, [http://freesoftware.ircam.fr/rubrique.php3?id\\_rubrique=14](http://freesoftware.ircam.fr/rubrique.php3?id_rubrique=14)
- [3] *VideoDSP* by Christian Klippel and Etienne Deleflie, a video plugin for jMax, 2001-2002, <http://video.mamalala.de/index.html>
- [4] *Ruby* by Yukihiro Matsumoto et al, since 1995, <http://www.ruby-lang.org/>
- [5] *APL* by Ken Iverson, since 1962
- [6] *Jitter* by Joshua Kit Clayton, since 2002
- [7] *Gimp* by Spencer Kimball, Peter Mattis et al., since 1995