vvvv as a language

- a critique of vvvv -

Sebastian Gregor

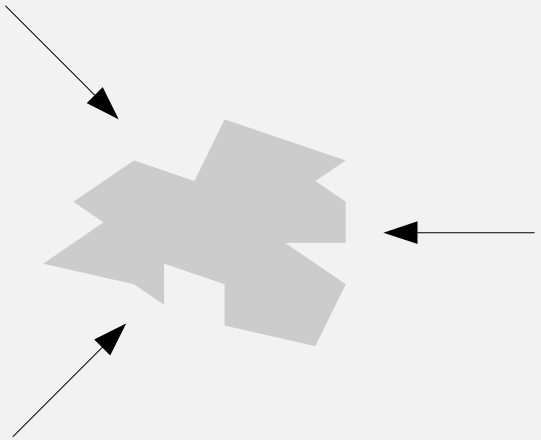
18.11.2010

Node 10

massivly edited version

Part I

The Bigger Picture



Why this talk?

Only with an understanding of

- how the software is used and
- what it actually is

we can

- learn from others
- find the flaws
- create ideas for improvements

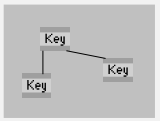
What is vvvv?

- a development environment

code &
patch editors

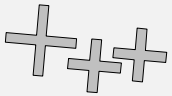
inspector

finder



- a runtime

edit while running



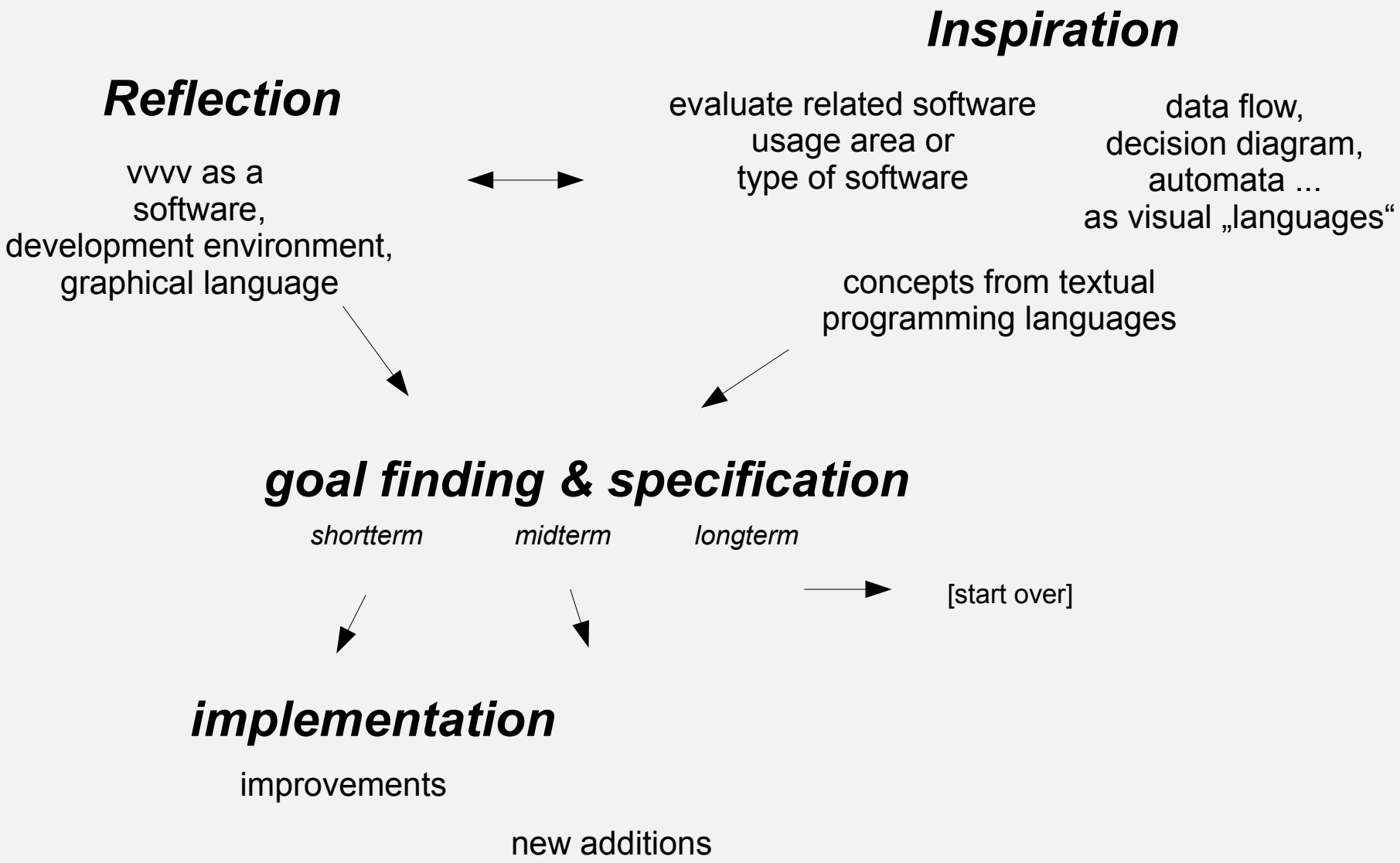
boygrouping

- a library: collection of nodes

- a graphical language

?

How to step forward?



vvvv: Perspectives and Criteria

- vvvv as a software (multipurpose toolkit), defined by
  - node library
  - applications
  - use
- vvvv as an IDE & Runtime
  - running modes
  - tool windows
  - supported languages
  - use
- vvvv as a language

  ?

vvvv45 – a Software Toolkit

- Nodes
  - Ease of use: structuring via name, category & tags
  - Areas of use: Node types (audio,video & general purpose plugins, effects, modules...)
  - Manageability: Collection size and growth
  - Node design: Flexibility vs. High-Level
  - Specific topics and their representation via collections of nodes
  - Central topics and how to keep up to date

DX9?!

- Applications
  - Reactive installations
  - Linear videos, visuals, VJ
  - Is there more?

- Use
  - Web & community
  - Documenation & help
  - Licenses & development strategies
  - Plattform & hardware

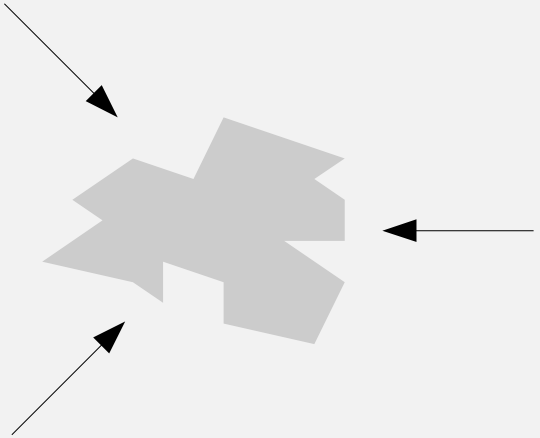
vvvv45 – a Hybrid Development Envirnonment

- Modes
  - Edit = inspect (= debug) = run, profile (CTRL-F9), shutup...
  - Very good prototyping properties
  - No „real“ debugging, no performance boost when edting is done
- Windows
  - Inspektor, Timeliner, Patch Hierarchie Viewer (Finder), Project Explorer
  - Patch & Code Editors
- Language support
  - vvvv, HLSL, C#
  - No F#, C++...
  - No GPU patching
- Use
  - Look & feel: design & responsiveness
  - Plug & play
  - Stability & threading

vvvv45 – a Graphical Language

Part II

Inspiration &
The Search For The Language



Related Software

- vvvv
  - visual, data flow oriented, frame based, graphics processing
- max, pd
  - visual, data flow oriented, events - no frames, audio processing
- field
  - textual programmable nodes, visually enhancable scripts, programmable GUI nodes, graphics
- touch designer
  - visually programmable GUI nodes, graphics
- ventuz
- open frameworks
- sharpdevelop
- ...

Visual Languages

- Fabrik (1988)
  http://portal.acm.org/citation.cfm?doid=62084.62100
  - dataflow with bidirectional links and GUI components (also see ThingLab)
- Prograph / Marten
  http://www.mactech.com/articles/mactech/Vol.10/10.11/PrographCPXTutorial/
  - data flow with mutable objects
- ACME
  http://www.cs.cmu.edu/~acme/docs/language_overview.html
  - Architectural Description Language based on components (services), connectors and constraints.
- esterel SCADE
  http://de.wikipedia.org/wiki/SCADE
  - „Integrated Deterministic Data Flow and Safe State Machine Notations“
- Subtext
  http://subtextual.org/subtext2.html
  - Schematic Tables: deciding horizontally, doing vertically

Visual Languages & Topics

- **data flow** = nodes are *verbs* (functions, operations on data)
  - synchronous, framebased vs. events. → combinable?
  - bidirectional links? (fabrik)
  - (mutable) objects in combination with flow control (prograph)
  - what's the role of the environment of a subgraph? (SCADE, Subtext)
- **state machines** / automata / decision diagram = nodes are *states*
  - graph describes control flow
  - combinable with data flow? (SCADE)
- **components** & systems = nodes are *objects*
  - (ACME)
  - events trigger connectors
  - connectors trigger setters on other components (respecting constraints)

Visual Language Paradigms
& some value judgements

- verb nodes, data links
  - *functional, data centric*
  - verbs + mainloop = reactive
  - + prototyping, highly modular, transparent
  - + good control over simultanious actions
  - - big systems with many independantly communicating components can't be mapped properly
- state nodes, condition & transition links
  - *imperative, state oriented*
  - + logic programming, control flow
  - - data processing needs to be added otherwise
- object nodes, event & message links
  - *imperative, object oriented*
  - + enables mapping of complex systems
  - - huge overhead.

Textual Languages

- Lustre, Lucid Synchrone
  - reactive, data flow oriented, different clocks
- J
  - functional, implicit multi-dimensional lists
- Haskell
  - purely functional
- F#, ML, OCAML
  - functional, object oriented
  - F# : meta programming
- Java, C#, Delphi
  - imperative, object oriented

textual ↔ visual

- #todo: detailed discussion of graphical representations of language concepts known from textual languages
  - is there a representation independant form of semantics?
    - which features of a programming language are in any way related to their textual or visual representation
    - evaluate relevance of syntactic expressions inside operational semantic expressions
    - …
  - compare by example
    - e.g. Prolog should have a straight forward visual representation (adding another class of visual paradigm with links establishing rules)
    - study visual haskell
    - ...
  - which languages have a better visual than textual representation?
  - what guidance do textual & graphical editors allow?

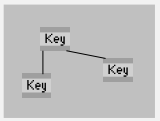
What is vvvv again?

- a development environment

code &
patch editors

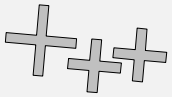
inspector

finder



- a runtime

edit while running



boygrouping

- a library: collection of nodes

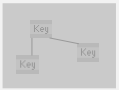
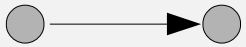
- a graphical language

what's left?

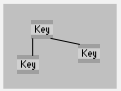
Part III

The Language vvvv

The language vvvv
and its topics

- Semantics (gives Meaning)
  - pins, values, links & data flow 
  - types & subtypes 
  - spreads & why they are no lists (unfailability) 
  - nodes, modularity & border control 
  - frames, states & flow control 
  - boygrouping
- Syntax (supports Understanding & Ease of Use)
  - graphical representations
  - intellisense: guardance while patching, unfailability

The language vvvv
and its primitives

- everything that *does* something is a node
  - it should be perceived as a closed entity
  - implementation knowledge ( / 

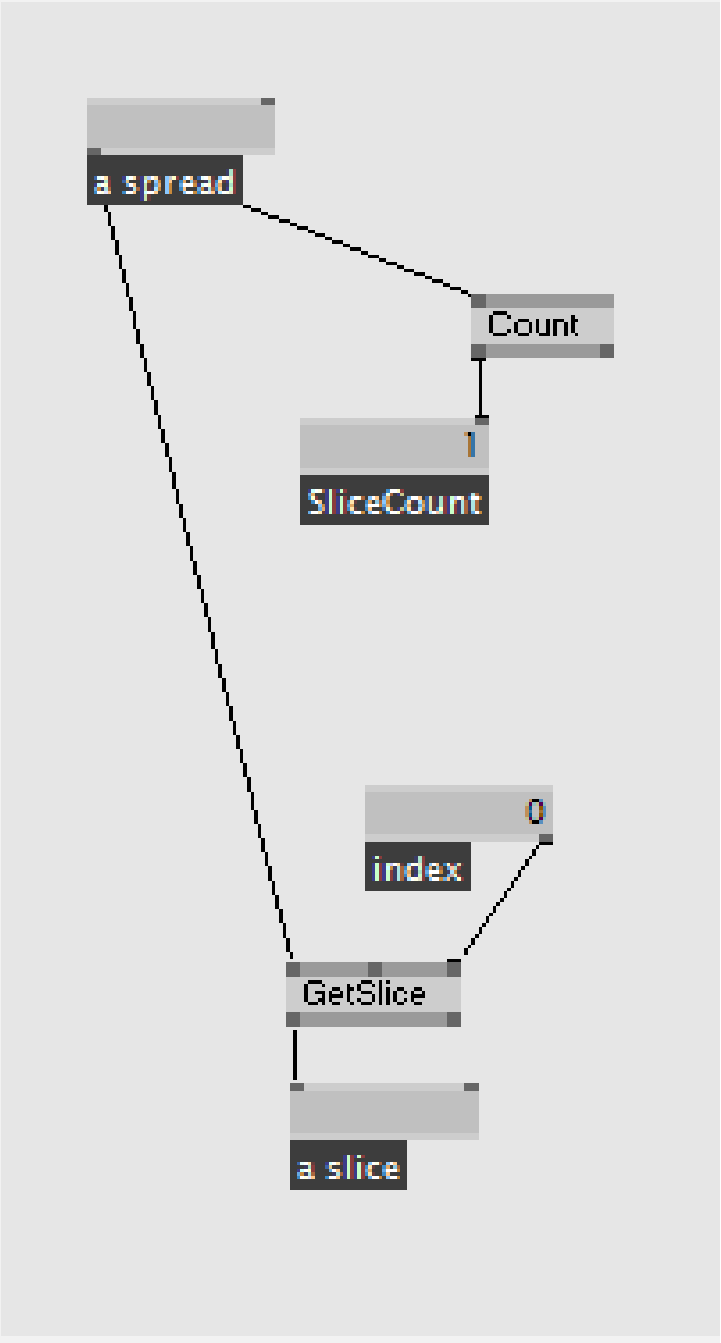
) not needed to understand its usage
- pins
  - hold data
  - are typed / subtyped
- links
  - are the glue in data flow
  - implicitly decide upon control flow
- spreads
  - are basically a data type comparable to lists
  - different lengths are magically no problem
- a patch is just a node

What is a node?

- It has a name
- It has input and output pins
- It evaluates data for the outputs
- It may have „side effects“
- It is a concept of modularisation

What is a spread?

- It is the data
  - stored in pins
  - flowing in links
- It consists of slices
- It has a slice count
- Access slices by index
- The index can be **any** whole number



Multi-dimensional Spreads

- example: ControlPoints of B-Spline (3D Surface)
  - the values in this pin specify a spread of surfaces
  - with each several curves
  - with each several control points
  - with each [x, y, z]

- what should be the meaning of CAR?
  (a) should it output all points of the first surface?
  (b) output all points of the first curve of each surface?
  (c) output the first point of each curve?
  (d) output x of each point?

- How would you want to specify that?

Part IV

The Language vvvv45

Types and SubTypes

- Value (numbers and booleans)
  - subtypes for whole numbers only
  - subtypes for booleans only
  - subtypes for different min, max and default values
- Color
- String
  - subtypes for filenames
- Enum
  - subtypes for the allowed values
- „Node“
  - anything else
  - only few operations are available (GetSlice (Node), Switch (Node) for whole spread)
- Spreads are neither Types nor SubTypes

The genious of implicit spreads

- almost any pin of any node can handle many slices
- this way whole patches end up being spreadable
- no need for the user to distinguish between a color or a *spread* of colors. spreads are always there
- all nodes can handle different spread lengths on their inputs
- there is one rule of thumb that tells how different spreads will be related to each other

  (spreads are endless lists → modulo)
- the biggest spread defines the slice count of the outputs

Problems of implicit spreads

Demo

Calculating Indices
within
B-Spline (3D Surface) module

Multi-dimensional Spreads

- example: ControlPoints of B-Spline (3D Surface)
  - the values in this pin specify a spread of surfaces
  - with each several curves
  - with each several control points
  - with each [x, y, z]
- what should be the meaning of CAR?
  (a) should it output all points of the first surface?
  (b) output all points of the first curve of each surface?
  (c) output the first point of each curve?
  (d) output x of each point?

How would you want to specify that?

- *BinSize & VectorSize* pins for reinterpretation of the raw data
  - how to get the right bin and vector sizes for the different cases (a..d)?

Multi-dimensional Spreads

Spreads in dynamic plugins are a pleasure...

We even can work with spreads of spreads.
A feature not present in the native graphical language...

Implicit Spreads:

we need additional BinSize (& VectorSize) pins to reinterpret the well known (implicit) spread as not to be 1D...

Explicit Spreads:

use a spread when you need it.
Spreads of spreads are most elegant.

```
ISpread<ISpread<Vector3d>> ControlPoints;
```

Polymorphy & ISpread<*T*>

- we don't have polymorphy yet
- nodes that *should* operate on *spreads of any type*:

  CAR, CDR, CONS, Queue, Ringbuffer, GetSlice, SetSlice, S+H, r, s, Switch (input), Switch (output), Stallone, GetSpread, Scroll, Select, Swap, Unify (Set), ...

- other sorts of polymorphy
  - element types that are comparable:

    Substitute, =, <, >, <=, >=
  - element types that are addable, negatable ...

Flow Control

- We have one main loop that ensures that the whole patch system (graph) is evaluated within descrete logical units in time (frames)

...

  - This ensures that many timing problems just can't occur. The user just doesn't have to know the exact evaluation order of nodes.
  - However it is useful to know that the order of evaluation respects the dependancy of nodes: a node that needs data from another nodes' output will be evaluated later than the node it depends on → downstream nodes are *pulling* data from upwards connected nodes
  - Since cyclic loops within one frame are not allowed, each frame is a consistent dependency chain and can be understood without knowing evaluation oder. In logic programming you just can trust on the idea that nodes are evalauted „at the same time“ (frame).

- No further flow control seems necessary

States & Dynamic Spreads

- node states are hidden within nodes
- think of a Damper which tracks/generates movement of individual particles
- there is no way of telling the Damper that a certain particle (slice 17 within a bigger spread) died and that in the next frame slice 17 corresponds to former slice 18...
- → dynamicly changing spreads are not compatible with nodes that have states

- demo of how to get around the problem:

  see folder id buffer

Nodes & Modularity

- Input & Output pins form the interface of a node
- It seems like a perfectly closed entity
- However:

  Solving a problem once doesn't mean that you solved it once and forever:
  „Is that node already spreadable?!“
  - How can that be, when it is a perfectly closed unit that you just want to use several times?
  - → missing spreadability is a problem that comes from implicit 1d spreads

- what we would need is the ability to „close“ the node over pin types:
  - e.g. a pin has type
    - `Color or Spread of Colors ...`
    - *`T`* `in general`
  - spreading a node (feeding a `Spread of` *`T`*, where *`T`* is expected) would just result in „using“ the node several times
- → we have code modularity, we additionaly need data modularity via explicit types (like ISpread<T>)

vvvv45 – a Graphical Language

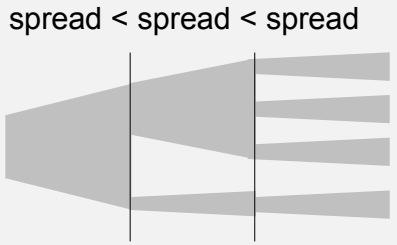
- Spreads & Types
  - + implicit recombination when dealing with spreads of different lengths → easy prototyping
  - - no real „subspreads“→ indexing „GAU“ (worst case scenario)
  - - no Polymorphy → nodes operating on spreads are implemented more than once
  - - no own types (objects, functions)
- flow control
  - + mainloop = synchronous data flow → no need to specify evaluation order
  - - no alternative loops / clocks → some tasks (Spectral nodes, fast audio graph...) are not patchable
  - - automata not integrated nicely→ Logic programmierung is tedious
  - - asynchronous/parallel evaluation not possible
- states
  - + nodes with states are easy to use, Framedelay allows to do own states
  - - States are incompatible with dynamic spreads. → some tasks (e.g. particle systems) are not easy to patch

Part V

Ideas for a future version of the language vvvv

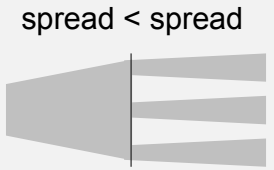
Multi-dimensional Spreads

- example: ControlPoints of B-Spline (3D Surface)
  - the values in this pin specify a spread of surfaces
  - with each several curves
  - with each several control points
  - with each [x, y, z]
- what should be the meaning of CAR?
  (a) should it output all points of the first surface?
  (b) output all points of the first curve of each surface?
  (c) output the first point of each curve?
  (d) output x of each point?



How would you want to specify that?

- **explicit** typing á la `ISpread<ISpread<..>>`



  - should yield only one meaning of CAR:
    first slice of outermost spread

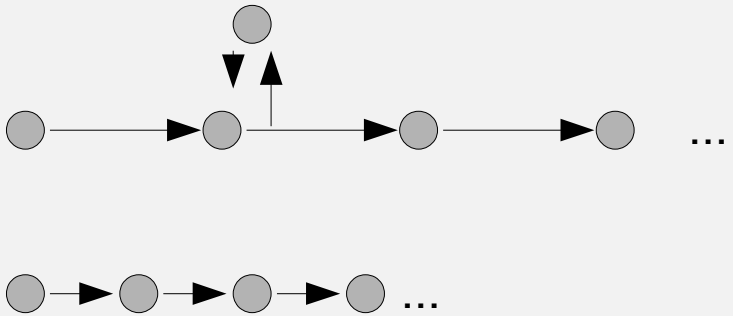
    if you want anything else, other helper nodes should
    get involved to access other levels within the spread of spread (…)

  - alternative: picking the right level (0..3) of an explicit spread of spreads of spreads of floating numbers^^

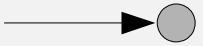
Flow Control

- Frame based „simultanious“ evaluation by the mainloop is a key feature of vvvv. The mechanism behind is called a main *loop* that seperates one frame frome the other and - within each - ensures evaluation of each node that is subscibed to it or that delivers data for those subscribed.
- However alternative *loops* would make sense:
  - To be able to iterate a spread (Spectral nodes!)

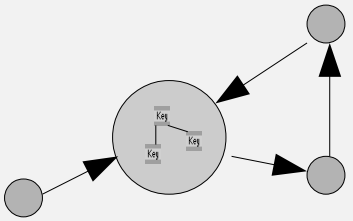
    (e.g. to patch a „+ (Spectral)“ when you already have normal „+“)
  - To be able to have different evaluation speeds for different part of your program



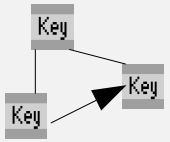
- You might want to react only once on an event. So no loop at all. A patch that gets evaluated only under certain circumstances...



- You might want to integrate a data flow patch within a state patch of a surrounding state system. The patch would only be evaluated when the surrounding state machine is within the specific state.



- In seldom cases (side effects) you might want to give hints to which node should evaluate earlier than another.
- You might want to run a patch in a different thread...



Function types

- would allow to abstract over functionality, where now we only can abstract over data

  e.g. the inlets of a patch abstract over some parameter data, so that it can be used with different input data

  - if we could abstract over functionality, we could
    - patch a „+ (Spectral)“ by combining a „fold“ with the functionality of the „+“ node
    - feed the functionality of comparison („>“) into a sort node, to sort the data, that can be compared by the comparison node
- how?
  - strike out input pins yields a functionality
    - the node itself now is connectable
    - outputs dissapear
  - that functionality now can be fed into a node that can apply the functionality
    - e.g. an „apply“ node, that after knowing the type of functionality, offers inputs for accepting the missing data. This appyl node then would have the outputs that dissapeared upstream.
    - any other node that can apply the functionality otherwise (e.g. Sort, fold, map)
      see functional programming

Patch states

- persistent data by introducing „patch fields“
  - the last state of a patch field can be accessed like you access data of an patch inlet
  - the state of a patch field can be written like you pass data to an patch outlet
  - data gets fed back from the system without the need of a framedelay
  - if a node has states, it has - by default hidden - in & outputs for these states, so you can grab them and manage states in lists
  - built in nodes would also offer pins to grap theor state
  - by that particle animations can be done much easier

Expressivness

- You can sum up all suggestions with a demand for more expressivness.
- Expressivness means more explicit constructs:
  - explicit (spread) types to ensure modularity
  - explicit flow control to allow more applications
  - explicit node state representations to be able to treat slices like objects independant of their position in a spread

- Expressivness doesn't mean that you always want to be explicit
  - You still want the ability to feed a spread of colors where only a color is expected → implicit map
  - You still want an easy implicit mainloop that is basis of a typical application → root template
  - You still want nodes that know their past (have states), like filters, behave as known, without the need of explicit state handling → implicit feedback, if state input not connected in parent patch

vvvv – a language

The limits of my language mean the limits of my world

Wittgenstein

so the language allows to express ourselves

or prevents expression

in terms of vvvv

the language sets the rules and
therefore decides which problems can be solved with patching

Thanks!

Vera Siegmund

Elias Holzer (elias)
Joreg (joreg)
Tebjan Halm (tonfilm)

Woeishi Lean (woei)
Julien Vulliet (vux)
Elliot Woods (sugokugenki)
Sebastian Oschatz (oschatz)

Eric Meijer
Simon Peyton-Jones
Benjam C. Pierce

and all that publish their knowledge on the web

channel9, stackoverflow, wikipedia …

free software and open source contributors

haskell for keeping it real