VL Concepts + Patterns I

Tebjan Halm + Elias Holzer NODE17

Intro

- **Top-Down** view on everything
- 6 topics with 30 minutes
    - **Document handling** and how it affects the **node browser**
    - **Definitions and Applications** how **nodes are made and used** in detail
    - **Data types** and how the **type system helps you**
    - **Evaluation and control** flow, **when does stuff happen** and how define it
    - **Generic definitions**, extension of normal definitions, **very practical nodes**
    - **Loops and collections,** spreads not spreading, **simpler slicewise thinking handle dynamic instances**
- Very small and **simple examples**
- Most concepts can be **explained with simpler ones**
- Patch/UI features are **sugar/shortcuts**

Document Handling

Structure

- Starting VL creates a ‘**Session**’ → similar to websites
- The ‘Session’ has a ‘**Solution**’ which is the Root element
- The ‘Solution’ consists of one or more .vl docs → see solution explorer ”All Documents”
- One .vl doc can define any number of nodes → **Whole project in one file**, more convenient faster to navigate
- Doc can **reference** other docs → called **dependency** → keep nice relative structure
- Can also reference **c# dlls and nugets**

Definitions

- Adding a **dependency** means **adding nodes to the node browser**
- Making patches → making definitions → definitions in doc also add to the node browser (registry, marketplace of nodes)
- **Add help text** via CTRL + M
- **‘Forward’** definitions of a dependency → **add definitions to own definitions** if another document references this document → rather used by **library patchers**

VVVV Integration

- Node definitions need to be in **category that starts with ‘VVVV’**

- Drop .vl with vvvv nodes on .v4p → pop-up with available nodes
- **Place in ‘vl’ folder** besides .v4p → available nodes in NodeBrowser

Definitions and Applications

Patch Types

- Document Patch → void, contains definitions
- Group Patch → extends document patch → **like “folder” in file system**
- Util Operation Patch
- Process/Record/Class Patch

Node Definitions

- Every definition **adds to node browser**
- Four **node definition types**:
    - Process
    - Member Operation
    - Util Operation
    - Delegate → **ah hoc operation definition**, not added to node browser, can only be used by **Invoke** → can link stuff into it from enclosing operation
- **Stateful**
    - Process
    - Member Operation
- **Stateless**
    - Utility Operation
    - Delegate
- **VVVV nodes**
    - Any **util** or **process**
    - Category **starts with VVVV**
    - **Non-generic**

Node Applications

- **‘calls’ the operation** with given data
- Nodes get called if containing patch gets called → strict evaluation
- **Calling starts somewhere** → vvvv node gets called by **mainloop**, also **input devices** start a call
- Main patches (vvvv nodes) call their nodes → and so on
- “Uses” has similar meaning → tell compiler to make an application here

- **Regions** have ‘something like operation’ inside that gets (maybe) called somewhen else → delegate is an ‘ad hoc’ operation
- Three (four) node types that call operations:
    - Process node → **needs stateful patch** (other process, record, class), only a **shortcut for pad + ops**, can be turned into a type by placing it into wrapper type
    - Utility Operation node → **can be placed everywhere**, **needs all data put in** (from input pins or pads)
    - Member Operation node → nothing special, **same as util, just has “state” in/out** of defining type, but **also needs all data fed in** (“instance of defining type”)
    - “**Invoke**” is a special operation node → doesn’t call operation from node browser, but from **delegate at input pin**
- **Main questions** as patcher are on **what operation are you currently patching** and **when will it be called?**

Pin Groups

- Of node can be **chained** → **multiple pins**
- **First in and out has to match**, other pins can be anything
- Kind of **easy with member ops** → show in demo patch

Data Types and Data Flow

The Data Tree

- Just to get the **big picture**
- Your program is a big tree of **structured data + operations**
- If nothing gets called the tree stays the same forever
- **Operations change the tree** on every frame/other external input device call
- You **know a few from vvvv** (value, color, string) → **now many more and make your own**
- Each **field (pad, property**) creates a new branch of a given type in the tree → instructing the machine to provide this memory space with this data
- **Process nodes** also build up the tree, even more obvious (**is pad + ops**)
- The **machine automatically takes care of well structured data** → **type system**
- The more types you nest into each other the more powerful this concept gets
- **Inside an operation** you have a contract via the types/signature that makes sure you **don’t have take care of the outside world** when patching in a type
- Allows to **patch everything for just one slice** → that is the most simple case
- **Type system also helps a lot** while patching, **always ask what type is the data?** → gives information of what the data is intended for and what operations are available for it

Data Fields

- Define the **structure of the type** → each field adds to the **memory**
- Description/template/blueprint for **all possible instances of that type** → concrete values → one concrete instance
- **Operations** can **read/write this data**
- **Data of fields comes in via state pins** → show example with one field → differences of record and class
- **Just a shortcut for split/join and util operation**
- All would work with only records without operations + util ops
- But it's **super handy**… → see some basic types like **S+H or FlipFlop**
- Patch simple **MyRect example**

Record vs. Class

- **Record** is located at the actual data hub and **moves around** → operations create **new instances**
- **Class** is located in **one place**, **reference moves around** → all operations **read and modify this place** → rectangle example switch between record and class
- **Record** is always the favorite since **more easy to handle**
- Show patch with change nodes
- Show audio example → **records messages** between threads → **classes managing** objects that stay the same
- **Multiple sinks** order problematic for classes → warning in patch → **see next topic**

Evaluation and Control Flow

Operation Order

- Operation gets **called by the operation that uses it** → if nobody ontop that chain/strain gets called by **mainloop** or **input devices** it **will not run** → there is **no auto-evaluate**
- **Dataflow inherently defines order**
- Show operation patch as basic example
- Show **counter patch** → also **record vs. class topic**

Process Node Order

- “Order” entry in **definition defines order**
- Show comparison patch **Process vs. explicit patched ops**

Conditional Evaluation

- If is very **basic and important** as a concept, it allows to **call different code paths** depending on the **result of a calculation** → route the data flow a different way
- Show IF region
- **In dataflow** it makes sense to provide the **if and else data for the output** → accumulator → if patch gets called on true or pass ‘unchanged’ data to the output on else
- Compare with **vvvv Switch → evaluation differences**
- Show **enable/disable LFO as type patch** → makes much sense to **split GetValue from Update**!
- Enabled pin is **shortcut for most simple if use case**

Regions

- Like the If there are **other regions** → patch in region can be **called independently from surrounding operation**
- **Delegate** → ad hoc utility operation definition → **does not get called by surrounding operation** → only invoke would call it
- Nodes can take **delegate via input pin** and call this ad hoc operation **whenever they want and as often they want** → different evaluation context → can be on another thread like in audio example
- **Reactive nodes → executes** delegate **when event fired** → independent from mainloop and can be on other thread → CraftLie CPUCoreRace
- LINQ examples → **ask user of node for type specific operation** or stuff the **developer does not know** but only the person who uses it → **incomplete operation** inside
- **Loop is very similar**, executes the patch (delegate) inside it **for every slice/iteration**

Generic Definitions

Idea

- Make **one definition** → compiler builds **applications for concrete type**
- Like **definition is a ‘template’ for all possible instances**, **generics extend definitions** in the same way → **template for all possible types**
- **Simplifies patching** because all **operations that don’t care about the data** can be made like this → **Spread, getslice** and so on…
- **Define once** use many times in slightly different ways

Generic Operations

- Easy to write **operation for many types that are similar** (vectors, numbers, collections)

- Resample spread example
- **Using other generic ops** helps of course
- When running into **type specific trouble/questions → delegate** input to handle the questions → make incomplete operation and **complete it with invoke/delegate**

Generic Types

- Same **works for types**
- Point with color example
- **Instances** with **different type parameters are not compatible → actually different types** build by the compiler
- See next topic for most prominent example

Loops and Collections

Spreads vs. Spreading vs. Loops

- **Spreads are collections** that can hold slices of **any type → generic**
- In **vvvv spreading** worked because **every node has build in loop**
- **VL has explicit loops** → make **your own spreading** → no surprises
- Compare **spreading vs loops in patch**
- Check **spread operations**
- **Spread of Spread** handled by the compiler, **endless** → level of spread is number of **nested loops → see patch**

Other Collections

- **Sequence** → the **mother of all collections → simplest collection**, can only iterate thru slices → only **ForEach** works
- **Spread is a Sequence** → show **compatible pins** → also subtype <> supertype
- **Type hierarchy** → everything is an **object**, everything can be connected to an **object input**
- **LINQ loves sequences** (and therefore **spreads** (and List, ...))
- Other collections have additional features like random access
- **SpreadBuilder is a class**, used **when a lot of modifications are needed → spread** is a record **copy on modification**, good for **reading and passing data around** → spread builder helps to **"generate" a spread efficiently** → **ToSpread 'locks' the data** and makes it secure
- See CPUCoreRace example → spreadbuilder gathers all positions → ToSpread send to mainloop

Loops in Detail

- **ForEach** → go thru all **slices via splicer → spread min** when multiple splicers

- **Repeat** → set **iteration count** → fixed call count → nice to **generate spreads**
- Can **link stuff from outside** → fixed value for all iterations
- **Splicer** slice/**element** of the collection **per iteration**
- **Index pin** → iteration number
- I Spread patch
- **Accumulator** passes **data from iteration to iteration → possibly new value per iteration → initial value as input → final value as output**
- Patch examples, odd/even, average
- Recursive tree **combination of loops, splicer, accumulator and delegate**
- Don't forget → **can give names to everything**

Summary

Patch Sugar

- Many concepts are **only a combination of simpler concepts**
- **Process node → field + operations**
- **Member operation → split/join of data fields + utility operation** → data + matching operations with state in/out
- **Region node with delegate input → on the fly operation definition**

Guidelines

- **Start with Process Definition**
- Extract patch parts into **small readable utility operations**
- Only **switch to Record or Class if you need to send the data type over a link** → call operations on an instance in different parts of your patch
- **Record** for small **data 'packages' the are sent around in the patch** (rectangle, spreads, vectors...) → often re-created every frame → needs to store the new instance back to field
- **Class** for more **central types that manage** logic → usually not created/deleted every frame
- Try to keep one path for classes (reference) to have a well defined order → every op changes the same data → **multiple read operations are no problem**