Noodle Reasoning

Anton Kotenko

December 28, 2020

Abstract

This paper tells any reader in details, why Noodle project matters for JetBrains and what its competitors lack of.

Contents

1 Core idea

1.1 Functions as nodes

Every function has its arguments and return value (or several ones). Every node in visual programming has its inputs (inlets) and outputs (outlets).

A *pure* function is the function which never changes something that came inside without telling the outside world about it. So, it may have local variables and change them, but without affecting the values given by its arguments. If they are changed, their new state has to be returned from the function. It is the internal ecosystem.

I vote for the fact that any node template in visual programming is a representation of such function.

The definition of a node in the node library is the definition of a function, and the instance of a node in the working patch is the instance/call of such function.

There can be side-effects, if they are isolated.

In Pure Data, for example, one may edit the code that is represented by any `object` node. Actually, it is a one-line expression, a call of a function. If one adds arguments to it, the node automatically and immediately gets another inlet. If the function returns more than one thing, the node gets one outlet more.

1.2 Expressions as node instances

Also, in functional programming, unlike imperative programming, there are no statements / expressions that do not return anything.

Every `if` *has to have* the corresponding `else`. Every pattern matching *has* to cover all the possibilities. It gives advantages not only in the compiler guarantees, but also by allowing you to replace *any* expression by *any* expression, while they both have the same type. A value, a function call, a statement, `let-in` block —literally anything.

Notice the fact that functions may be partially applied when called, so keeping the same type is even easier.

This way, the body of any expression (be it function definition or something else) in pure functional languages, is nesting blocks: it just goes deeper and never goes vertically (unless it is a list of named definitions).

Some frameworks in imperative languages, especially the ones which are connected with art, come up with the similar ideas, consciously or unconsciously.

I vote for the fact that any node instance in visual programming is a representation of such expression.

Meaning that every expression that can be represented as a nested block, may be laid out on a patch workspace as a node and nesting may be emulated by connecting such nodes.

1.3 Streams as the first-class-citizens

Functional Reactive Programming was popular by bringing the concept of streams in the area.

Consider streams as the infinite flows of any data (events, for example) you may subscribe to, and by filtering and combining the data from them, get what you need and react correspondingly. The most important thing, everything in the flow should be immutable or else any such handling turns out to be a mess.

FRP concepts were brought to many imperative languages since, but most natively and intuitively they still look in pure functional programming, where such stream is just another Monad :).

But we're talking about visual programming now.

Most of the programs nowadays listen for some events and react to them. The struggle was always to make events friendly and be able to say "Fire this event every 5 seconds" or "this happens infinitely, but when I trigger this, the value changes, but still continues to live" or many other combinations (or *combinators*, if you let me).

If you look closely, both in art and in math/physics, both in data-processing and machine-learning, such streams is one of the core pillars.

- For math, it is a live plot of some function when incoming data changes, the plot changes live;
- Or, the progression of values (constant in time or not);
- For physics, it is the values of forces, they exist always, sometimes they are just 0 or negative;
- Or the vectors of objects in space;
- For IoT, the movements the robot hand should perform;
- Or the lightning in the room for a camera/HomeKit to adjust it to the proper color/amount;
- For music, it is the MIDI stream of notes, or just the endless sound stream;
- For generative music, this stream is produced by oscillator and could then be filtered by LFO etc. as a *combinator*;
- For art, it is the animation timeline, and the lifetime of every object in the animation, can be split in tweens, for example;
- For data processing, it is the live data of COVID-19 patients, for example;

- For machine-learning, the data the obstacles the car "sees", is an infinite stream of data;
- For neural networks, it is the values that pass between the neurones and functions on the neurones are the *combinators* of such values;
- For programming, it is the `EventBus` in MVC or the flow of Actions in Redux, every framework/approach nowadays has a similar concept;
- ...can be continued indefinitely, as a stream of ideas itself;

In FRP, it is possible to define the list/array of values as a stream, producing the values at a single point in time.

In PureData, there are special connectors and links that represent the infinite stream of sound or, actually, any data, the bold ones:

I vote for the fact that any stream between the connected outlet and inlet can be an instance of some infinite stream, and the nodes may act as the combinators on it.

1.4 ADTs to control types

Algebraic Data Types allow to describe much, much more concepts rather than simple type systems. Recursive or not, functional or not, primitive or not. They help in finding generic abstractions between the concepts, and it's not just philosophy and belief, it turns out to be pure algebra indeed (hence the name).

In visual programming, it is important to connect specific outlets to specific inlets. So to say, the outlet producing sound has more sense to be connected to the inlet receiving sound. It is also important to make such ports distinct visually.

ADTs allow to wrap any type in any other type, so there could be a matrix of sound streams represented as type—and it has sense in visual programming as well, say you want to show a table of seismic spectrograms in a body of a node, where every spectogram in this table is *updating live*.

I vote for the fact that any data flowing between nodes is better to be stored in a flexible type system.

1.5 Resume

Those above are not theoretic and exclusive conclusions, seems many developers nowadays come closer and closer to the similar ideas.

I think they are the reasons why `luna-lang.org` has appeared and had some "hype" and still a lot of programmers are waiting for it to be stable and be released. `luna-lang` back-end is written in Haskell, and the front-end was re-written from JS to Haskell-to-JS, since alpha had many-many front-end bugs, mostly thanks to JS dynamic typing etc.

But the idea they have—every node is a pure function, is a same concept I postulate above. They have taken it literally and it's a Haskell function indeed, including function composition as a constructors for such nodes. I think they have proven that these ideas do work at least for data processing. If the implementation works as well, we'll see.

Haskell is not the only functional language that fits these ideas, there is Elm, there is Racket and lots more.

My primary goal is not to develop the tones of nodes as a library that would prove perfectly the ideas listed above, but to develop the core of a visual language that would allow to easily build such libraries, with a little-to-no coding. May be using the Noodle itself. It is the secondary goal.

The VVVV project could be the illustration of that it possible and not a fantasy. They have all kinds of core blocks like loops and boolean operators and list producers, and the community has built the library on top of it thanks to the fact there was a core.

One of the core principles of functional programming is *re-use as much as you can*. Since the set of building blocks (nodes) is minimal, it is easy to find patterns in their combinations and re-use them later. By creating templates (a.k.a. *subpatches*), for example.

Figure 1: Pure Data. Notice bold lines for signals (~ means output is a signal) and nodes being functions (some, partially applied). Operators are functions of two arguments. Also notice the node with the spectrum of sound.

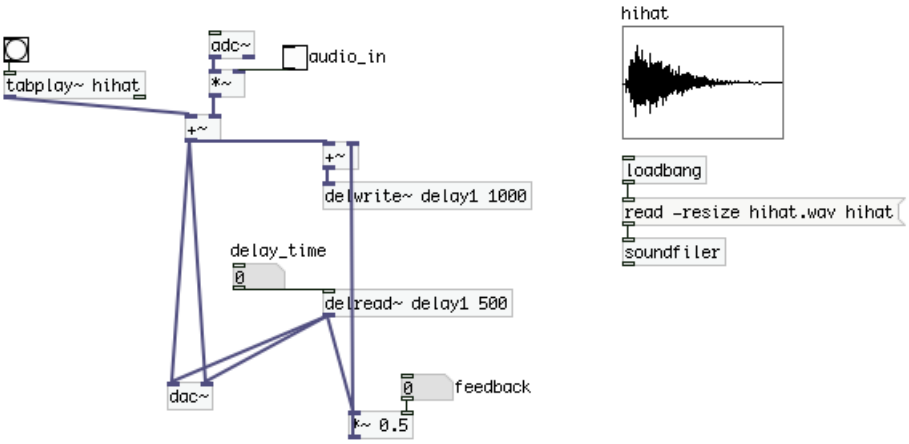


Figure 2: Max/MSP. All the same applies, as the above. But here partially applied functions are having empty plugs for arguments. Plus, more nodes pretending to be controls.

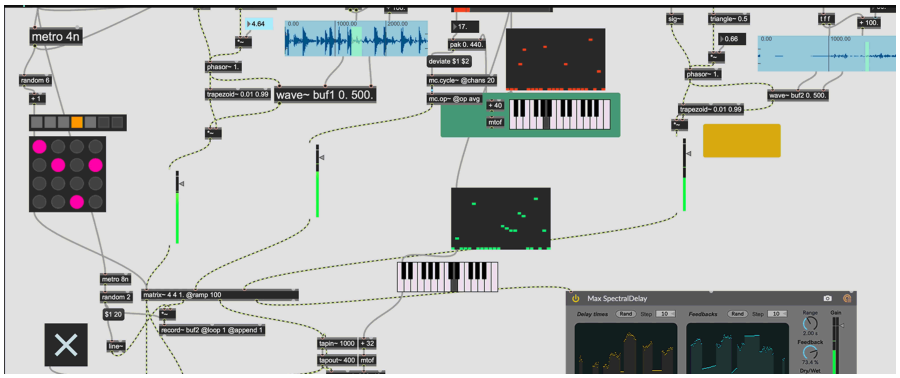


Figure 3: The paused video from Improving Declarative APIs for Graphics with Types article. Using Elm-like language as a reference, it shows how pure functions are transformed in Scratch-like blocks. So can be the nodes.

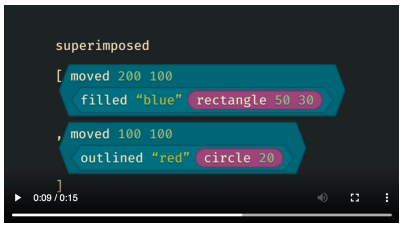


Figure 4: The figure from Elm and Algebraic Thinking to K-8 Students paper. Same thing, pure functions are blocks to build graphics.

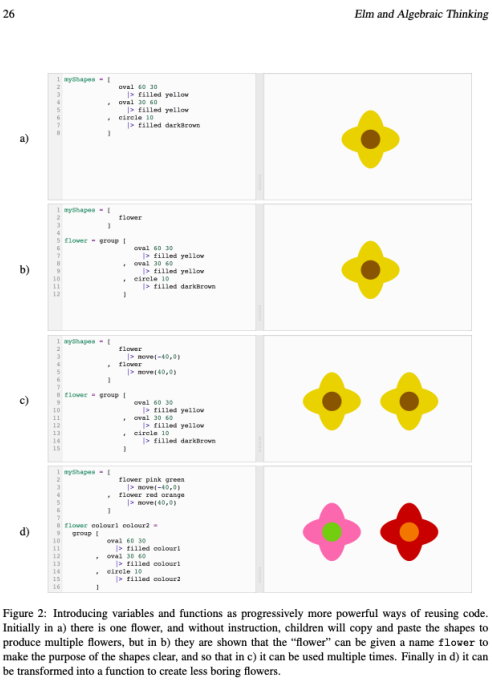


Figure 2: Introducing variables and functions as progressively more powerful ways of reusing code. Initially in a) there is one flower, and without instruction, children will copy and paste the shapes to produce multiple flowers, but in b) they are shown that the "flower" can be given a name `flower` to make the purpose of the shapes clear, and so that in c) it can be used multiple times. Finally in d) it can be transformed into a function to create less boring flowers.

Figure 5: Hydra. It is the generative art tool in JavaScript. Still there are functions and their composition used to produce some combined effect. Any function here could be represented with a node. Be it oscillator or blending operator. Notice the fact that mostly just these 44 nodes are used to produce tons of different scenes.

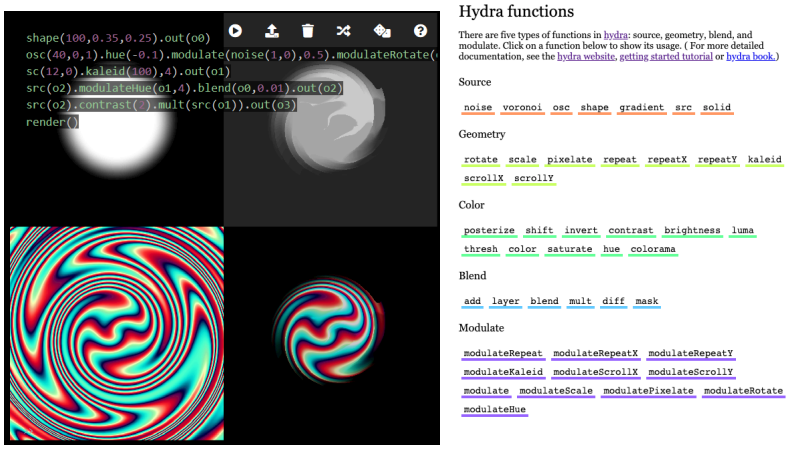


Figure 6: The `fluxus` application, using Racket language to generate graphics using functional programming.

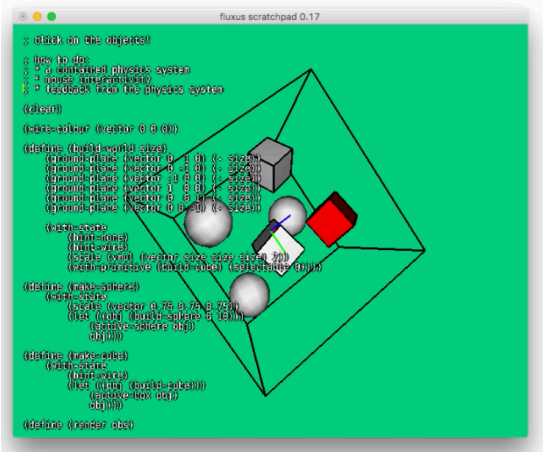


Figure 7: Luna Lang. Screenshots from the demo video. They even don't hide the fact that every node is a functions or a composition of functions. . operator in Haskell represents the function composition.

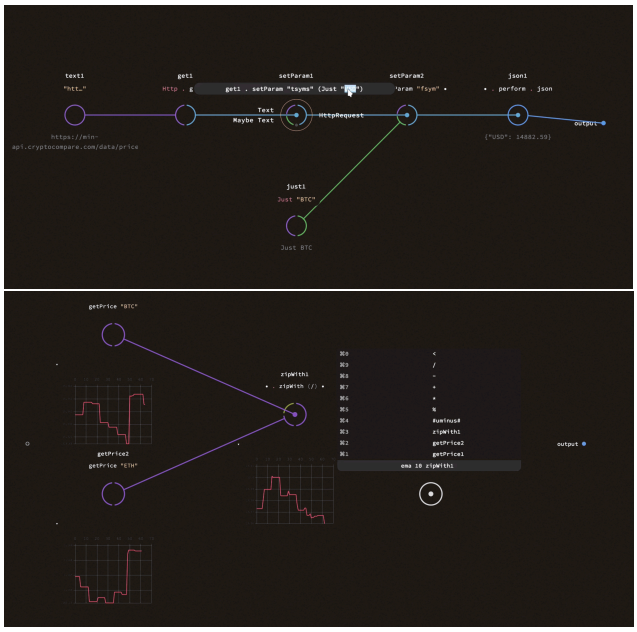


Figure 8: Screen from TouchDesigner. Source image. Notice how every node is the minimal operator with its own preview. Compare with code from Hydra, Elm and other examples above and below.

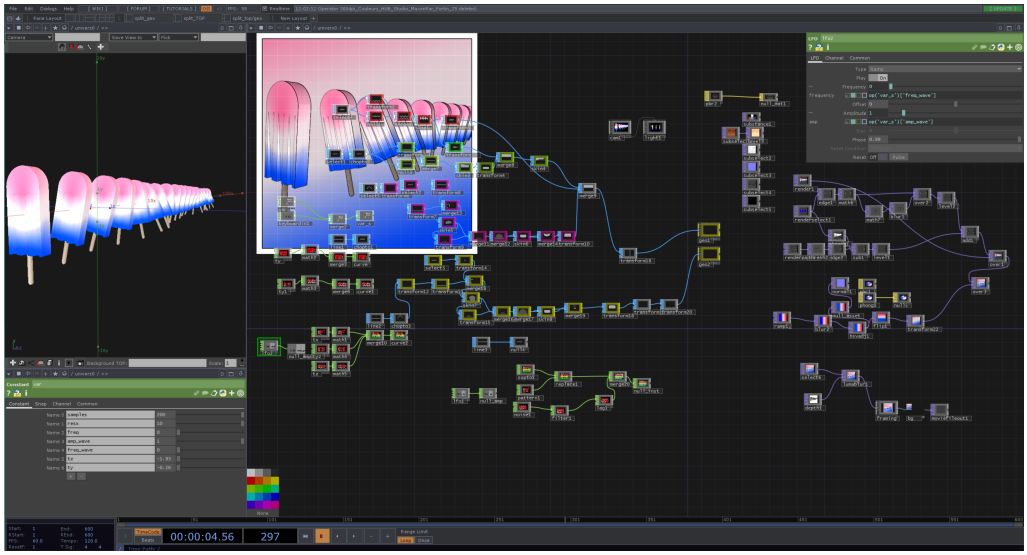
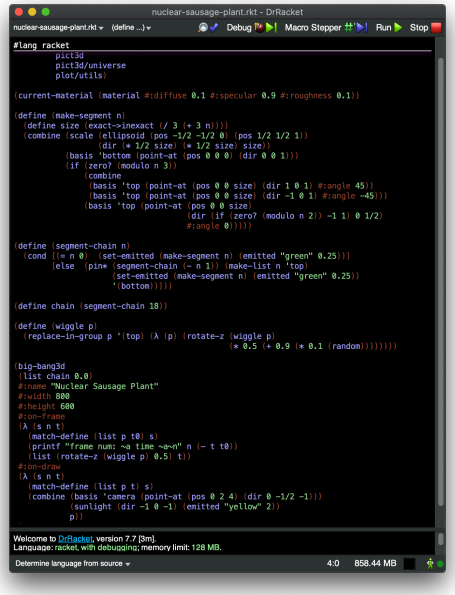
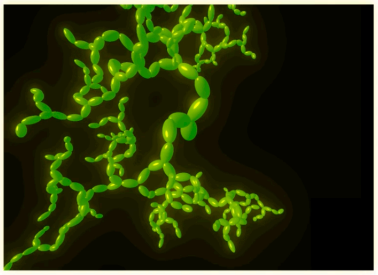
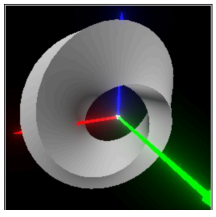


Figure 9: The examples from Pict3D library for Racket. The complete *Nuclear Sausage Plant* producing code is shown on the picture. Notice how pure functions act as the building blocks.





```
> (combine
   (deform (tessellate (rectangle (pos 0 0 2) (dir 1/8 1/4 2))
                       #:segments 48)
     (smooth-compose
      (move-x (/ -2 pi))
      (bend 360 (interval 0 4))
      (twist 45)
      (rotate-z 45)))
   (basis 'camera (point-at (pos 1/2 1 1/2) (pos 1/8 0 1/8))))
```



2 Why JetBrains would be interested

Since it seems to be a one-man-project, at least for a moment, it is a risky situation, no doubts here.

On the other hand, JetBrains releasing a good visual language + its IDE, while there are mostly only parts of such in different IDE-like software, would be a huge news.

2.1 Kotlin?

I am not for using Kotlin at this point *only because* all the core ideas are much harder to be expressed in imperative/semi-imperative language, be it Kotlin or Scala or anything else. When all the core types and logic is polished and stable, it would be much easier for me to re-implement them fully in Kotlin or wrap in Kotlin external API later.

2.2 Another language

Rather it seems to be another language, a simpler one, and a visual one. I consider Elm, for example, as a simple language, since they have a very limited list of possible statements and declaration (like, 20 different building blocks and a *single A4 page* describing syntax). They removed all the complexity Haskell has and left only the primitive functional Lego Blocks, and I think it is for a great good! *Primitive & Simple* doesn't mean that one doesn't have to learn anything new, but in this case, just a little.

If that building blocks would be presented visually, and the code would be hidden (may be editable, but hidden by default), learning and using such language would be much easier.

Consider it as Scratch for adults.

So, since JetBrains has one language developed, why there couldn't be another? Even not a competing one, but the one covering completely another area. Like JetBrains Sans and JetBrains Mono both (will) exist and cover two different areas digital design and coding.

2.3 Another IDE

I don't believe in prevailing extremes. I don't believe that most of our customers/users have nothing artistic in heart. Many of us programmers wanted to develop games and seriously thought about it as a career.

Alan Kay, in late 1980s was trying to implement visual programming interfaces.

The ideas of Bret Victor changed how some IDEs, like XCode, look nowadays, and the fact there exist Python Notebooks. May be it's not the ideas of Bret in particular, but still the constant feeling that for code there should be more than letter symbols and sometimes programmer wants to try different values in real time and wants to have check on how program performs with different data in real time.

Of course, not everyone would like to move in visual programming or do the move and never return. It is not a competition, letters vs nodes, it is just the fact that in certain areas visual programming could be, or even being, more useful than "non-visual" one, and so why would not JetBrains cover both?

Mostly in games. I mean Unity. But also in data processing. Also for proving theorems. For research and for science. Some modern scientists learn and use functional programming because it is closer to math and algebra for them. Some modern scientists build visual tools for themselves to help in research. For example, the suicide analysis, they like interactive graphs: meet Lineage. For art, yes. For configuring IoT devices.

I think, a well-done visual IDE would easily sell, even without having a huge library of nodes in the package. The similar products (they don't position themselves as IDEs though) do sell nowadays, and they sell libraries of nodes separately. See *Friends and competitors* section. If the visual language allows to easily build your own node library and motivates to do it, any programmer with an artistic heart, even a small one, would buy it.

2.4 Another try

I am aware that in JetBrains there were numbers of tries to develop something like that, visual programming or similar. I can be not aware about the exact number of tries. As a follow-up, it's certainly not an alien idea to JetBrains.

I don't consider those tries as failed or having no sense. Sometimes it is not the right time for something, sometimes it is all because of marketing, sometimes it is not the idea what failed, but the way it was implemented (UX). As for `luna-lang`, for example, their first alpha version was built on Electron (not that bad these days) and was just popping JS errors to everyone who launched it. Beta front-end is in much more stable, but people remember. Still, almost anything in development can be re-used and re-though to build something better.

I would:

- Focus on simplicity;
- Focus on stability;
- Focus on solving well-known problems with visual programming;
- Focus on not producing new ones;
- Focus on solving previous failures;
- Focus on not producing new ones;
- Focus on keeping the rational mind;
- Focus on keeping the irrational soul;
- Listen to the others who tried (unless they dissuade to try again);
- Welcome them to try again with me;
- Listen to the UX specialists;
- Listen to the QA specialists;
- Listen to the designers;

I am not sure anything of it would help not to fail, since I don't know where success can be guaranteed; If it's about fitting people needs, then there's a higher chance to success, probably.

I feel the need for laconic and simple visual language in programmers, or else they wouldn't try to implement it again and again.

2.5 Solving problems

One of the core problems with visual programming is, usually, the amount of space needed for a visual developer to have a bird-eye view on a project. The more complex the thing is, the more nodes it is needed.

- The problem can be solved with *subpatches* it is the named part of the whole network which is "collapsed" in one node and can be entered with double-click on such node, for example; This node may have inputs and outputs connecting the outer nodes to the inner nodes; That kind of subpatch may be re-used as a template for another instance;
- Sometimes it is solved with easy zooming; Still, the developer has to jump out and in many times;
- But the best solution of this problem, in my opinion, is detachable nodes: every node in a patch could be taken away and moved to another tab in the browser or another screen or another device, like iPad, for example; This is easily achieved with WebSockets+URLs and some uniform format for sending the updates between instances;
- If we move and can render UI in Virtual Reality, there's the infinite space to place nodes. It's not that hard to render to Virtual Reality nowadays, with frameworks like a-frame. without the zoom, though, the user still would have to run a lot;
- Everything of the listed, combined, would help user to focus on the important things without distraction; In letter-driven programming we have modules, classes, files and directories, which all communicate and bound in the single application still there were found ways to help the developer focus on one thing at a time without completely forgetting of others;

...This is not a finished chapter. There will much be more problems to solve than this single one, used as an example. Some are known from experience, some not.

3 Friends and competitors

I do not consider any of the listed projects as competitors, not because Noodle is definitely better just by braving, but because Noodle is planned to be built on their ideas, combine them and improve, solve some problems they have, and only by that, be factually a bit better.

3.1 What could distinguish from others

Noodle is planned to combine: minimalistic approach of Pure Data. Which can be, but not required to be, with the help of community, extended to the possibilities of VVVV/nodes.io.

- Works everywhere concept;
- Minimal set of nodes / operators, so that it is easy to create libraries;
  - For that to work, one node could represent one pure function and so, like in PureData, be modified/replaced on the fly; So that one would not be required to define node in its full, rather register some function as a node;
  - There should be a node that visualises any data (as well as stream) as a graph;
  - There should be a node that visualises any data (as well as stream) as a table;
  - ... some more items ...
  - It would seem that it's not enough to produce both graphics and audio and be able to process any data; But notice a limited number of nodes *as a core blocks* many of the competitors have and still could produce truly a lot; So we just should find a subset of this limited number of nodes or extend it just a little;
- Subpatches navigation using URLs; It exists almost nowhere, but very useful, to place parts of the very complex UI on different devices; We tried that in Tron with WebSockets and it works;
- VR interface—the infinite space where one may put and organize nodes;
- Type safety and stability thanks to functional programming;

3.2 nodes.io

This project is the general inspiration for me. Actually they released few weeks ago, before it was just a page with motivation. The motivation, very very similar to mine. This motivation story is still there.

Now, they even have previews on nodes. WebGL demo.

One may enter the node code and see that the

Pros	Cons	In Noodle
-	JavaScript	No
-	No F-prog. advantages	Yes
Web	-	Yes
Seems stable	-	Planned
Rich library (somewhat)	-	Not yet
Node Inspector	-	Planned
WebGL support	-	Planned
-	No Audio support (yet)	Planned
-	No IoT support (yet)	Planned
Subpatches	-	Yes
Zoom	-	Planned
-	No detachable subpatches	In work
Inlets are streams	-	Yes
-	Outlets are not streams	They are
-	Node != expression	==
Quick node creation	-	Planned
Code editor	-	Yes, but...
Preview in the node body	-	Yes
-	Community: not yet	Not at all

3.3 vvvv

Gamma version.

Website + Propaganda.

This project is the second general inspiration for me. They've released Gamma version recently. It doesn't means it's pre-release, they just use greek letters for the release names. And, they say, *it's rewritten from scratch.*

Pros	Cons	In Noodle
-	Only Windows	Web
Very stable & fast	-	Planned
Rich library	-	None yet
Friendly documentation	-	Planned
Optional node Inspector	-	Planned
Spreads	-	Planned
Multi-threading	-	Not planned :(
3D support	-	Planned
Audio support	-	Planned
IoT support	-	Planned
Subpatches	-	Yes
Zoom	-	Planned
-	No detachable subpatches	In work
Inlets can be streams	-	Yes
Outlets are not streams	-	They can
-	Node != expression	==
Quick node creation	-	Planned
-	No code editor (ext. C#)	Yes, but...
Preview in the node body	-	Yes
Huge community	-	Not at all

3.4 Pure Data

Website.

This project is very-very old, completely free, and exclusively intended for writing generative music.

But it has several nice ideas I would like to borrow. May be it even was inspirational for all the projects in the list:

- Minimal number of nodes in the library still allow building very complex things;
- Mostly, thanks to the `object` node which is a function and expression and can be edited.
- Sound as a separate type of connection b/w nodes;
- Subpatches;

Pros	Cons	In Noodle
All Platforms, Desktop	-	Web
Stable & fast	-	Planned
Doesn't require rich library	-	Planned
Friendly documentation	-	Planned
Optional node Inspector	-	Planned
-	No 3D Support	Planned
Audio support	-	Planned
-	No IoT support	Planned
Subpatches	-	Yes
-	No detachable subpatches	In work
Inlets can be streams	-	Yes
Outlets are not streams	-	They can
Node == expression	-	Yes
Quick node creation	-	Planned
Code editor	-	As planned
Graph in the node body	-	Yes
Huge community	-	Not yet

3.5 TouchDesigner

Website.

TouchDesigner is an example of a very successful visual programming product. On the other hand, it is used only (but massively) for producing 3D and 3D effects. Looks like it is far from actual programming. No that much, they have function-like operators and combinators. They, even, have a `null` object, which represents nothing in 3D space, though.

Pros	Cons	In Noodle
All Platforms, Desktop	-	Web
Stable & fast	-	Planned
Has rich library of operators	-	Planned
Friendly documentation	-	Planned
Optional node Inspector	-	Planned
3D Support	-	Planned
Audio support	-	Planned
-	No IoT support	Planned
-	No subpatches	Yes
-	No detachable subpatches	In work
Inlets can be streams	-	Yes
Outlets are not streams	-	They can
Node == expression, kinda	-	Yes
Quick node creation	-	Planned
Code editor	-	As planned
Graphics in the node body	-	Yes
Huge community	-	Not yet

3.6 Others

- Natron : only visual effects;
- Nuke only visual effects;
- AngryAnt Behave : only AI;
- Grasshopper : only 3D;
- FlowCode : only microcontrollers, rather scratch-like than node-like;
- Nodal : only music; cool concepts; was awarded the Eureka Prize for Innovation in Computer Science;
- Softimage : only visual effects;
- Max/MSP : only music only in Ableton;
- NodeRed : only IoT;
- Visuino : only IoT;
- Blender Node Editor : only 3D;
- Orange : only data-mining;
- Cameleon : very similar idea, has 3D, but implemented in QT and uh...;

4 Literature

Every document like this contains some links to the scientific papers and convincing internet articles. This one is not an exception.

- Improving Declarative APIs for Graphics with Types by Philipp `matheusdev23`. July 28th, 2020.
- Using Elm to Introduce Algebraic Thinking to K-8 Students. Curtis dAlves, Tanya Bouman, Christopher W. Schankula, Jenell Hogg, Levin Noronha, Emily Horsman, Rumsha Siddiqui, Christopher Kumar Anand. McMaster University Hamilton, Ontario, Canada. 14 May 2018.
- The Rise and Fall and Rise of Functional Programming by Eric Elliott. Feb 19, 2017.
- Functional Reactive Programming with Elm by Jan-Patrick Baye. Programming Languages and Compiler Construction Department of Computer Science Christian-Albrechts-University of Kiel. 2014/2015. *Contains information on signals and how they are bound to graphics.*
- Graphics Programming in Elm Develops Math Knowledge & Social Cohesion. John Zhang, Anirudh Verma, Chinmay Sheth, Christopher William Schankula. McMaster University. October 2018.
- 10 Advantages of Elm: Moving to Functional Programming in the Frontend. Admin @ Doing Software Right. February 21st, 2020. N
- IBM releases Elm-powered app. *Elm is really bullet proof, its not [false] advertisement.* October, 2018.
- Asynchronous Functional Reactive Programming for GUIs. Evan Czaplicki, Stephen Chong. Harvard University, 2013.
- The Next Mainstream Programming Language: A Game Developers Perspective. Tim Sweeney, Epic Games. 2005.
- The Next Paradigm Shift in Programming. Talk by Richard Feldman. 2020.