Encouraging Experimentation Through Programming by Proximity

Tom Beckmann
Hasso Plattner Institute, University of Potsdam, Germany

Leonard Geier
Hasso Plattner Institute, University of Potsdam, Germany

Stefan Ramson
Hasso Plattner Institute, University of Potsdam, Germany

Marcel Taeumel
Hasso Plattner Institute, University of Potsdam, Germany

Robert Hirschfeld
Hasso Plattner Institute, University of Potsdam, Germany

Abstract

Exploratory programming involves evaluating a variety of approaches to identify those that advance the problem understanding. For this purpose, we investigated a notation for code designed to encourage experimentation with elements of a program. In our proof-of-concept, we evaluate the idea of program elements connecting by mere proximity through small case studies. We identify multiple constraints to enable connection through proximity and its limitations.

2012 ACM Subject Classification Software and its engineering → Integrated and visual development environments

Keywords and phrases Visual Programming, Proximity, Experimentation Support

Digital Object Identifier 10.4230/OASIcs.Programming.2025.15

Supplementary Material *Software (Source Code)*: `https://github.com/hpi-swa-lab/syntactic-shuffle` [2], archived at `swh:1:dir:2a0397fdd156bf216400c71e2899bb512f25a316`

Acknowledgements Thanks to the reviewers for their extensive and helpful comments. This work was supported by the HPI–MIT "Designing for Sustainability" research program[1], and SAP.

1 Introduction

During exploratory programming [17], programmers work on open-ended tasks where an implementation is not obvious from the start [3]. Consequently, they benefit from exploring multiple approaches to address their problem domain.

To support the exploration of approaches, various methods and tools have been proposed. For instance, Babylonian Programming [14] supports programmers in evaluating their progress in an approach through fine-grained, inline feedback. Variolite [12], Fork It [19], or Exploriants [1] enable microversioning that allows programmers to explore multiple approaches simultaneously.

In this paper, we describe an exploration of a notation for code that is designed to facilitate spontaneous change. The notation's core idea is to express connection between programming elements through proximity. As an effect, many edit operations are reduced to a single drag operation to the rough vicinity of where the programming element should come into effect.

[1] `https://hpi.de/en/research/cooperations-partners/research-program-designing-for-sustainability.html`

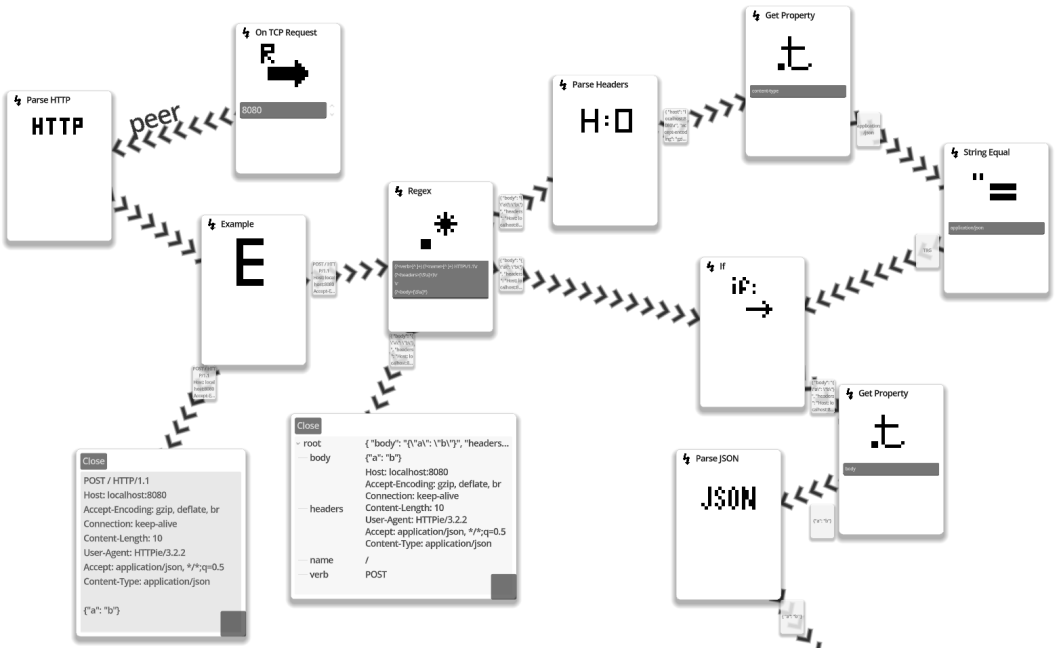

Figure 1 A screenshot of a program for an HTTP server expressed using cards in our proof-of-concept of a programming environment that connects its programming elements through proximity. Note that we moved the program start point, the "On TCP Request" card to the top left for a better fit on the page.

In the following, we will first outline the idea further and what challenges arose during prototyping (Section 2, then describe how our approach addresses these challenges (Section 3) and what programs look like (Section 4), discuss it (Section 6), describe related work (Section 5), and finally conclude the paper (Section 7).

2 Connection Through Proximity

Our concept's goal is to identify a programming environment and notation that reduces *viscosity* [10] to support spontaneous changes to a program. More concretely, we seek to reduce the effort required to start exploring an alternative or elaborate a new thought in the notation such that it can be executed by the system and new implications from the runtime results can be derived.

The core idea is to connect program elements through proximity only. This is in contrast to nodes-and-wires programming, where wires connect specific slots of nodes, or block-based programming, where blocks snap together or contain slots for nested blocks. We approached this idea through both analog and digital prototypes.

2.1 Analog Prototyping

During analog prototyping, we chose blank cards as primitive element. In testing sessions, we defined a rough goal that a program should reach. We then began labeling cards to identify a suitable level of granularity and abstraction for individual program elements. Our constraint was to ensure that we would need to be able to formulate notation semantics that would be able to unambiguously execute the arrangements of cards that we layed out. With

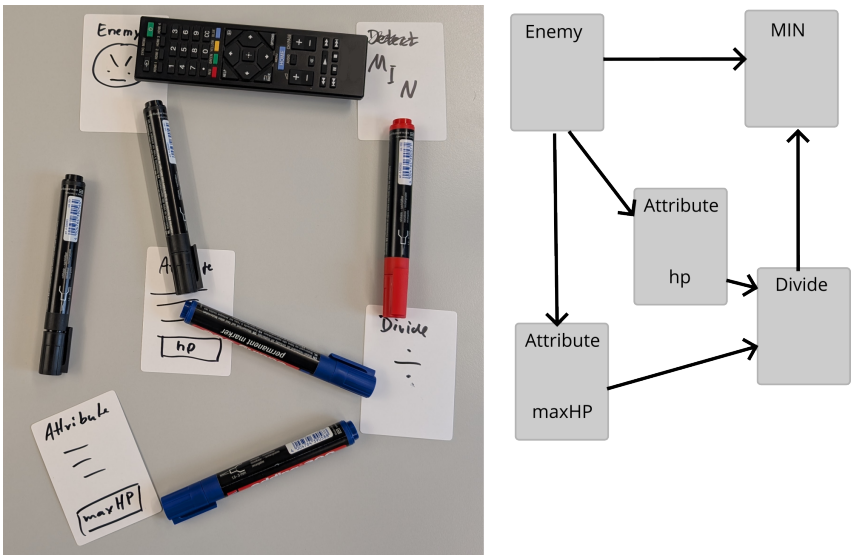

Figure 2 Paper prototype for a program that identifies the object of the "enemy" collection that has the smallest hitpoint (hp) percentage. The pens and the remote between cards are used to disambiguate connections. The card covered by the pen in the center says "Attribute" at the top. To the right, we include a digital recreation for clarity.

this constraint in mind, we tried to formulate programs as close to the problem domain as possible, often summarizing multiple cards into one or adding cards to disambiguate our intent when we were not able to refine the notation semantics to remove ambiguity.

One such program can be seen in Figure 2. The program operates on a collection of "Enemy" objects. The notation semantics we explored here apply the operators connected to the collection on each element of the collection, akin to APL [8]. The figure also demonstrates a limitation of the paper prototyping approach: it was difficult to arrange cards such that it was unambiguous which cards would connect. To disambiguate connections, we often added pens or other objects. This informed three design constraints:

C1. Require Semantic Compatibility Program elements must only connect to their neighbors if they are semantically compatible.

C2. Avoid Accidental Compatibility The protocol to determine compatibility must be expressive enough to match programmers' intent.

C3. Draw Connections Even though program elements connect through proximity, formed connections must be visualized.

As we labeled cards, we often included constants. This informed another constraint:

C4. Reduce visual weight of program elements of minor relevance to the program For instance, constant parameters with unsurprising default values should not carry the same visual weight as operations that are important to the program domain.

2.2 Digital Prototyping

During digital prototyping, we initially experimented with some different shapes for program elements but eventually adopted the same card metaphor used during analog prototyping. This informed the design constraint:

C5. Ensure Large Hit Area As our goal is to support rearranging elements, program elements must present a large hit area.

In the digital space, we began formulating larger programs. Here, the need to group and abstract elements became apparent, as this allowed programs to remain at manageable sizes.

C6. Enable Abstraction It must be possible to extract program elements into a reusable abstraction.

Through abstraction, it then became also possible to formulate new program elements within the programming environment and save those in libraries for reuse.

Finally, as we were composing programs of increasing complexity, we started to encounter more and more expressions that had a natural formulation in text but were cumbersome to express using proximity, such as mathematical expressions.

C7. Support Multiple Notations When an expression is best formulated using text or another notation, it should be possible to use that notation.

3 Cards

Our proof-of-concept is implemented using the Godot game engine [6] for fast, interactive graphics. The implementation comprises a general-purpose live programming environment. It is available open-source[2]. In the following, we describe relevant design decisions and how they relate to the constraints determined in our prototyping process (Section 2).

3.1 Cards

We decided to keep the same card metaphor we used in the prototypes, as they provided a large hit area (C5). We sized cards to allow targeting cards using a mouse at zoom levels where text is still legible.

The user places cards on an infinite canvas that can be panned and zoomed. By drawing a line over multiple cards, the user can select a group of cards, which can then be moved together.

Cards are implemented using other cards through an abstraction mechanism. When a card is double-clicked, it expands in-place to show its implementation. This works recursively: cards can be expanded inside of one another until only primitive cards remain.

3.2 Primitive Cards

Primitive cards are implemented in the underlying host system instead of using other cards. Our proof-of-concept includes five primitive cards.

To facilitate abstraction (C6), there are input and output cards. Input cards allow specifying a signature, further described in Subsection 3.3. A CellCard acts as a generic store for a named piece of data. To integrate with the system's lifecycle, an EnterLeaveCard can be used to react to user actions to add or remove cards from the program.

The most important primitive card is the CodeCard. We derive the need for CodeCards from C7, the support for multiple notations. In our proof-of-concept, users can write textual code of the host system in a CodeCard.

As illustrated in Figure 3, a code card consists of a function declaration, as well as a list of input and output name and signature pairs. The interface of code cards automates creation of inputs and outputs as much as possible, such that, ideally, users need to only

[2] `https://github.com/hpi-swa-lab/syntactic-shuffle/tree/c6a01db8e6c01cd2276fbe758004e4ab`

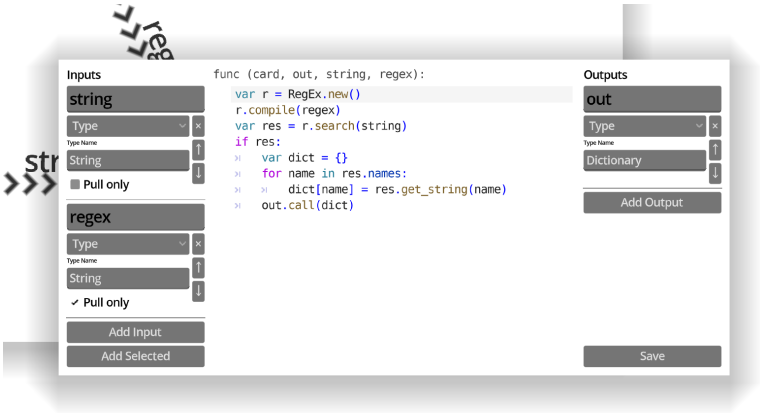

Figure 3 An expanded CodeCard that receives a subject string and a regular expression string. It contains a textual function that extracts the first hit and returns the resulting dictionary of captured values.

instantiate a code card and write the desired textual expression. By making selection of one or multiple cards and pressing the "Add Selected" button, the CodeCard automatically derives signatures and names for the selected cards, and connects them to the CodeCard.

CodeCards thus offer a more concise, textual notation for all parts of the system that are either likely to be static or appear to the system designer as details that they do not want to expose as cards.

3.3 Connections and Signatures

Connections in our proof-of-concept are established through proximity. Specifically, when the distance between two card midpoints falls below a fixed threshold distance, the cards attempt to establish a directed connection. We first try a connection from the card currently being moved by the user to the other, and if no connection can be established in this direction, we try the reverse, i.e., from the non-moving card to the card the user moved.

Two cards can establish a directed connection if the output signatures are compatible to the other card's input signatures, fulfilling C1, require semantic compatibility. The respective signatures of a card are determined by the primitive input and output cards they contain.

To address C2, avoid accidental compatibility, we experimented with a set of signatures that attempt to approach the user's intuition of compatibility. In our proof-of-concept, we introduced the following signature kinds:

Type. A type of the Godot host system, e.g., float or CharacterBody2D. Types are compatible if they are identical or if the one is a subtype of the other, as defined in Godot.

Generic. A generic type that assumes the type of all connected types.

Command. A command is a plain identifier wrapping another signature. It can be used to wrap for example a type signature with an intent, such as "Store" or "Accelerate".

Trigger. A signature that signals an activation without carrying any data.

Iterator. A signature that wraps another signature. When a non-iterator signature is connected to an iterator, it is upcast to be an iterator as well and its operation is applied to the entire collection the iterator is describing.

Cards can contain named and unnamed input cards. This is relevant for operators that require multiple arguments of the same signature, e.g., a subtraction operation that needs to distinguish between minuend and subtrahend.

Cards do not connect when a directed cycle would be formed through the new connection. However, to allow for more flexibility, inputs can be marked as pull-only, in which case a connection to that input is not considered for cycle detection. Using a pull-only input, an increment operation can be implemented: StoreCards in our proof-of-concept emit their value when they are updated. Consequently, an IncrementCard that reads a store's value and then writes an incremented number back to it would create a cycle. By marking the IncrementCard's input as pull-only, it can read the store's current value, write the incremented value, and ignores the pushed value update from the store.

In rare cases, a card may be compatible with multiple of the outputs of a card, especially if the receiving card has a generic signature. For this purpose, our proof-of-concept's standard library includes a FilterSignatureCard. Once connected to another card, it only forwards activations of a pre-configured signature and discards all others.

As another special case, it may sometimes be inconvenient to arrange cards using proximity as connection mechanism, for example if constructing a pipeline of operators that needs to refer to the original value at its end. For this purpose, our proof-of-concept supports a locking mechanism: when a card is held over another, a visual countdown appears that, when elapsed, establishes a locked, distance-independent connection. The locked connection can be unlocked using the same hovering mechanism.

Connections are shown through a directed line, as seen in Figure 1, to fulfill C3, drawing connections. Visualizing formed connections is especially important as data will only be transferred between cards if the signatures match, as further described in Subsection 3.4.

3.4 Execution Semantics

Using the EnterLeaveCard, cards that are added to the program can kick off an execution of a sequence of cards. Commonly, Cards that are starting points for execution will use a SignalCard, which subscribes to some event of the host system's standard library, when they enter the program.

In our proof-of-concept, most cards follow a data-flow-like execution model. When an output card receives data, it iterates over the compatible input cards of all cards within its reach and invokes these by passing the data it received. The input cards, in turn, contain a nested output card that repeats the same process.

CodeCards store a queue of passed values for each of its inputs. When an input card of a CodeCard is invoked, it pushes the value onto the respective queue. The CodeCard then checks if all queues are currently non-empty and, if so, pops each queue and performs its specified function, which may invoke an out card to continue passing data.

Through the queues, we protect against issues where the order of execution at a join point after a fork might prevent a CodeCard from being executed: independent of which path is taken first, the CodeCard at the join point will store the results of completed paths, until all paths have reached the join point.

To support concurrent processes running through the same graph of cards, the queues of a CodeCard are indexed by an Invocation object, which identifies the current execution thread. When a signal is invoked, for example a signal of an EnterLeaveCard or a signal of the host system, a new Invocation object is created. Invocation objects are held weakly in CodeCards but strongly while a CodeCard is performing its operation. Correspondingly, Invocation objects and their associated stored data can be freed as soon as all CodeCards that were reached finished execution. If a CodeCard invokes an asynchronous operation, it also maintains the strong reference until that operation completes.

3.4.1 Pulling Values

While building programs using the above execution semantics, we realized that there were instances where we wished for values to be pulled by CodeCards, as opposed to requiring us to fork execution just to trigger a static data store to emit its value into a CodeCard's queue.

As a convenience, we thus added the possibility for output cards to hold a "remembered value". A remembered value can be set by the output card's containing card to offer a value that can be pulled. Most notably, the primitive CellCards implement this behavior, such that it is always possible to pull value from data stores. Pulling works across boundaries of input and output cards, but not across any other CodeCards, as these would need to trigger side effects when traversed. Correspondingly, pulling a static number is possible, but to use the result of the addition of two numbers, where a CodeCard has to perform the actual addition, a normal push has to occur.

As described in Subsection 3.3, inputs can be marked as pull-only, in which case they ignore pushes and only look for remembered values.

3.5 Liveness and Feedback

As program elements are added to the program, they become immediately active and affect the program execution. The main entry point, as described in Subsection 3.4 is the EnterLeaveCard. The EnterLeaveCard passes in its "enter" signal a reference to the CardEditor object, which encapsulates the programming environment itself.

Using the reference to the CardEditor, we can express meta tools using cards. For example, by subscribing to the CardEditor's "after_edit" signal, we can implement a card that reacts to edits to the program. In Figure 4 we show the implementation of an ExampleCard that stores the last object it received as an input. When an edit of the program occurs, it re-emits that stored value to enable a live programming loop as known from example-based live programming [15].

When a card's output is triggered, it flashes the connection line, if any, to show that it was activated. Further, it places a small feedback card on the connection line that displays a textual representation of the most value that passed through the connection.

Beyond editing, the CardEditor allows other cards to create new cards, perform edits on other cards, or save card state. In this way, the CardEditor allows cards to act as meta tools when their function benefits from scripted editing steps. A future card making use of these facilities could be a regular expression card that parses the entered expression and automatically adapts its output card's signature to reflect the capturing groups in the expression.

3.6 Card Surfaces

The cards' appearance is modeled after common playing cards with a title at the top, an icon in the top half, and a content body in the lower half. The card's icon can be edited using a built-in pixel editor that becomes active by simply zooming close onto the icon, as shown in Figure 5.

To approach C4, reduce visual weight of program elements of minor relevance to the program, nested CellCards display an inline editing widget on its containing card to allow the user to change the value directly, as also seen in Figure 5. Card authors can thus use nested CellCards to allow users to specify static arguments, such as the string to be used for a string split operation.

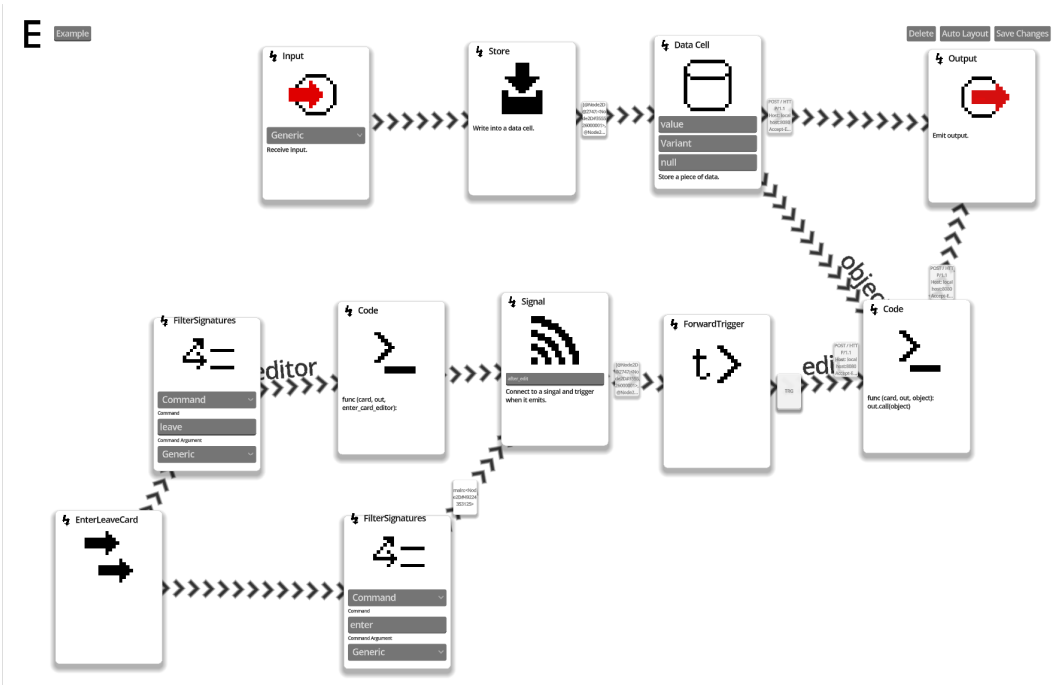

Figure 4 Implementation of an ExampleCard. The upper path receives and stores data. The lower path subscribes to changes in the editor and then emits the stored data.

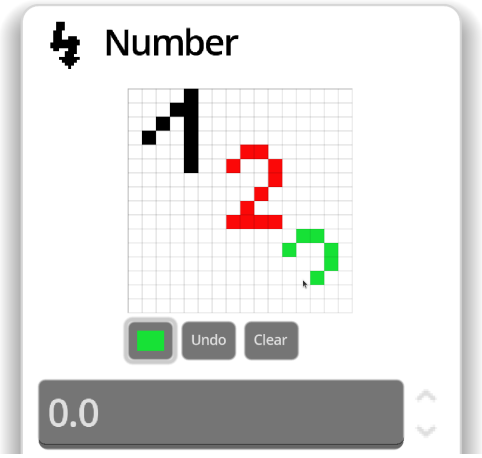

Figure 5 Icon editor active on the NumberCard. The viewport is zoomed in such that the card in the screenshot occupies about a third of the screen, as also evident from the small cursor.

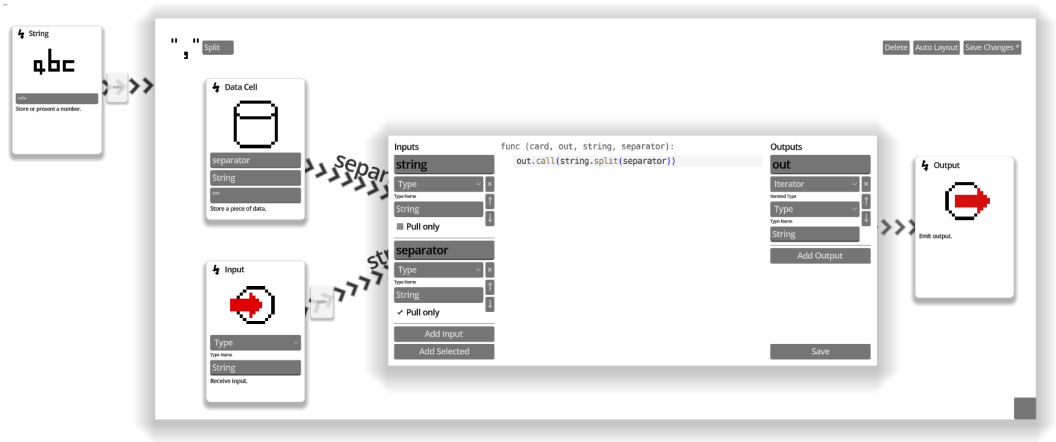

Figure 6 Implementation of a Split String card.

4 Examples

In the following, we outline the formulation of three small usage examples in our proof-of-concept.

4.1 Creating a New Card

As a first example, we want to describe the workflow to create new cards. We assume that the user has already formulated an initial program that reads a file as a string. Next, they want to split the string into lines.

As our proof-of-concept executes live, the user can see that their program is working correctly so far on the last feedback card. They also know that the host platform offers a split function on strings.

Correspondingly, they add a new CodeCard and a CellCard for the delimiter. They name the CellCard, assign it to be of type string, and enter a default value. Next, they select both the last card of their original program that yielded a string and the new CellCard, and click the "Add Selected" button on the CodeCard. The CodeCard automatically receives two new inputs of type string and is connected to the two selected cards. This side-steps the proximity connection mechanism for the specific scenario where previously incompatible cards are now made compatible by the user. An alternative, more laborious, method for connection allows the user to manually specify the expected signatures and then use the regular proximity mechanism to connect the cards.

The user proceeds to rename the inputs to be more descriptive, and implements the function by invoking the split function on the string using the delimiter on the CodeCard. They can immediately see using the CodeCard's feedback card if their implementation is working and proceed working on their larger program.

Later, they want to make this functionality reusable in other parts of the program. They select the two new cards, the CodeCard and the CellCard, and hit the "Create Card" button, which replaces the two cards in-place with a new empty card, akin to an "extract method" operation. Finally, they zoom in to the card's placeholder icon and quickly sketch an appropriate symbol, and name the card "Split String". We can see the final card in Figure 6.

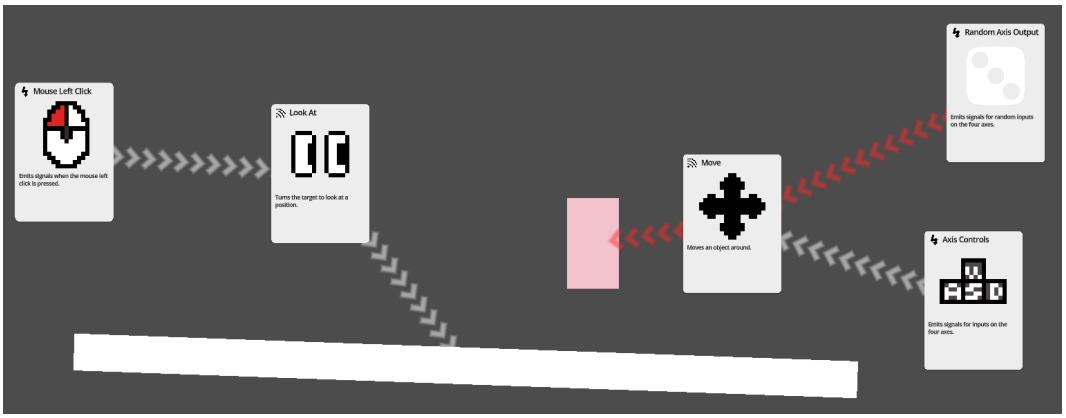

Figure 7 A program in which we experimented with two subsystems for game mechanics. The left cluster rotates the white floor piece to face the mouse cursor on click. The right cluster constantly applies random movement to the pink rectangle, while the user can counteract that movement using the controls card.

4.2 Game Prototyping

As our proof-of-concept is hosted within the Godot game engine, we were able to easily integrate a special connection to game objects of the game engine. Specifically, we extended our system to consider every game object, such as a player character or a wall segment of a level as a card as well. This allows connecting cards to game objects by placing them close to the object.

We found great utility in experimenting with unexpected combinations of cards. In Figure 7, you can see two such experiments we stumbled upon by connecting cards. The right cluster of cards combines an AxisControlsCard, i.e. deriving a 2D direction vector from the pressed arrow keys, with a MoveCard, which in turn is connected to the pink rectangle. We then also connected a RandomAxisCard to the MoveCard, which meant that the user constantly had to "fight" against erratic random movement with their own inputs.

On the left, we combined a LeftClickCard, which emits the mouse cursor's position on click, with a LookAtCard, which rotates a connected object to face a vector. Combined, the card cluster allowed us to rapidly orientate the connected white floor piece toward the mouse cursor.

4.3 Web Server

For a more complex example, we want to create a basic HTTP web server that receives a request, parses it, handles the application/json body content if set, and finally returns a string to the requester.

During its creation, we used the live programming workflow described in Subsection 4.1 of formulating our program using existing cards and successively introducing new cards that group parts of our program. Notably, the purpose here was not primarily reuse but abstraction, such that the final program remains at a manageable size.

A zoomed-out view of the final program can be seen in Figure 1. To give an impression of the split between cards and textual code, beyond utility functionality that would be part of a standard library, such as string split, match regex, or get property, the following functionality is found in CodeCards:

- TCP socket listen and accept
- TCP socket read
- TCP socket write
- JSON parse

The remainder of the program combines these cards together with cards of the standard library to reach the final program. In the screen in Figure 1, you can see the ExampleCard being used to capture the most recent request to allow seeing live changes to program output after every edit, without having to relaunch a request every time. Just underneath the ExampleCard and its subsequent Card, two InspectCards are placed. These act essentially as explorable, opt-in versions of the small feedback cards. In this instance, they visualize the raw request text that is stored in the example and the dictionary that results from using the RegexCard on the request string.

After the regex, the program proceeds to inspect the "Content-Type" header. We grouped parsing in its own card, then extract the "Content-Type" field, compare it to "application/json" and provide this input to an IfCard that conditionally forward its non-trigger inputs. Finally, the visible program excerpt extracts the body text of the request dictionary and passes it to the ParseJSONCard, which is another grouped card that invokes the host system's JSON parse function in a CodeCard.

5 Related Work

Our proof-of-concept closely resembles nodes-and-wires programming environments, such as Unreal Blueprints [7] or Blender's compositing, material, or geometry nodes [4]. It implements important aspects of these, such as the visualization of intermediate values close to each node, or card. It differs in that connections occur through proximity, whereas nodes in nodes-and-wires programming environments have dedicated input and output sockets between which connections are established. In the cited examples, the sockets are also typed, often reinforced through coloring or shapes of sockets, to communicate type compatibility.

Another important inspiration for our proof-of-concept are block-based editors, such as Snap [11] or Scratch [16]. These combine jigsaw puzzle-like blocks by snapping together or nesting blocks. The blocks are typically found on an infinite-scrolling 2d canvas. Blocks can be placed anywhere on that canvas to create subprograms or store blocks while the user is editing their program. Thanks to generously sized areas between blocks where dragged blocks can be dropped, the placement and combination of blocks present little viscosity to recombination, similar to our proof-of-concept. Likely due to the compact layout or as a deliberate decision to reduce visual clutter, block-based programming environments typically do not visualize intermediate results of blocks. Instead, for example in Snap, users can click on a block compound to execute it and see its return value in a speech bubble.

Along the same ideas of encouraging experimentation with programming elements, various tangible programming approaches exist [9, 13, 18]. Most of these focus on teaching an initial understanding of computational thinking by placing tangible elements that often resemble the blocks of block-based programming environments and interpreting their function in the room or on the surface that they are placed on. When embedded in a live setting, these tangible blocks can act similarly to the proximity connection mechanism of our proof-of-concept.

6 Discussion

In this section, we discuss our proof-of-concept, in particular with regard to the constraints we established in Section 2. Where applicable, we make references to relevant cognitive dimensions of notations (CDN) [10] We also outline avenues for future work.

6.1 C4: Reduce Visual Weight of Program Elements of Minor Relevance

We established as a constraint to reduce the visual weight of program elements that are of minor relevance to the program's purpose. This becomes apparent both in the CDN of *visibility* and *diffuseness/terseness*, as important elements are obscured and more space is taken. In our experience with nodes-and-wires editors, a lack in this regard is often a cause for the dreaded "node spaghetti", where the number of wires in the program exceeds what can be easily visually parsed.

We would argue that our prototype insufficiently addresses this constraint: while the need for some cards that merely parametrize a relevant operation is removed through embedded input fields, this is a binary decision for the card author. Either the argument is embedded, or it must be provided as an external card. This both limits the flexibility in which the card can be used (what if I need a computed delimiter for a string split but it must be provided as a constant string?) and also does not allow the user to express that a parameter may actually be of high relevance to the program, such as a speed vector for a movement card.

An alternative to our design might be to allow users to set a card to appear at a minimum size if it is less important to the program flow. Another prominent instance where this would have been helpful is in the web server program in Figure 1. The two GetPropertyCards in the program would have appeared as a simple ".body" in a textual program but appear with the same visual weight as important steps such as "Parse Headers" in the cards layout.

A solution commonly adopted by nodes-and-wires programs to the same problem is to embed widgets for each of its inputs that are present as long as no wire is connected to that input.

6.2 Execution Semantics

A repeated source of confusion when we were designing our own cards was the push vs. pull semantics. The design as described in the paper appears to strike a trade-off between flexibility and convenience. Common use cases support pulling values but every other case requires a push.

We often found ourselves spawning a ClockCard, which emits a push at a fixed interval, to kick off a computation or initialize a remembered value, if it involved a sequence of operators. After initial surprises while using the system, as we were not quite familiar with the consequences of the design yet, we eventually started to appreciate the relative simplicity of the approach. Here, a user study would help in understanding how surprising the mix of push and pull execution for users is in practice and might help reveal an alternative design that makes the semantics clearer in the visual design of the notation.

6.3 Interactions

Our main goal of enabling connections through proximity and thus supporting spontaneous interactions with a program appears to have been demonstrated as feasible for programs of small complexity. Both in the game prototyping example (Subsection 4.2) and in the web

server example (Subsection 4.3), the interactions allowed for fast experimentation, exhibiting low *viscosity* in terms of CDN. The inherently interactive setting and open-ended nature of game prototyping naturally felt like an even better fit to us while testing.

Through C7, support for multiple notations, it was possible to quickly and effectively make use of the host system's functionality without a significant context switch.

The most interesting constraint in this exploration was likely C2, avoid accidental compatibility. We were able to express all programs that we were trying to formulate. However, in a few cases, we had to make use of the FilterSignature card to narrow the returned signatures of a card that returns a generic type. For cards where their input and output signatures are compatible and vice versa, we had to make sure to move the card that was meant to act as output, as the system starts trying to establish connections from the card the user is moving. This caused confusion for us on several occasions, as the symmetric compatibility is not apparent unless you inspect the card signatures and is also cumbersome to fix, as you first have to move the card far enough away to break the accidental connection. As such, the CDN of *error-proneness* and *viscosity* are negatively impacted in these scenarios. In the future, one could consider either tweaking the signature system further to potentially carry roles or a concept of affinity such that the system would establish a connection in the intended direction on the first try. Alternatively, a gesture, such as a shake, might be used to reverse a connection direction.

More generally, the proximity connection mechanism does add a burden on the user to fiddle with card locations until only the desired set of connections is formed. Based on our limited experiments, it is still unclear whether the directness and spontaneity of connection forming will be sufficient to offset the occasional tricky situation, where users have to adjust multiple cards' positions, essentially solving a puzzle within the notation.

It would be possible to put connection forming behind an explicit gesture–for example using proximity to only suggest valid connections–which would introduce more friction in simple scenarios where the intended connections are obvious but would fix tricky cases where the user would otherwise have to solve a card arrangement puzzle. As a future direction, we may explore automatically detecting cases where the user's intent could be unclear and only then switching to an explicit connection mechanism.

As an outlook, we believe this authoring system to be useful in extended reality scenarios, as it can significantly reduce the reliance on keyboard input and relies on very few, coarse gestures. Similarly, exploring creative coding applications, such as live coding [5], in combination with this system appears promising–a card's precise position in relation to another may even become a continuous input that the receiving card can use to direct for example pitch or volume.

6.4 Implications of Spatial Authoring

Similar to textual programming languages, our system implements a default view on the notation. Whereas in text, elements' relationship is determined through adjacency and nesting in the linear text stream, in our system relationships between elements is determined through proximity on a 2D canvas. The 2D canvas can be discretized into separate areas in which the exact position of a card does not matter: within that space, the card can be moved around without affecting the connections and thus the program's effect–it is a form of secondary notation [10]. Textual programming also commonly offers forms of secondary notation through additional whitespace and comments.

Notably, this secondary notation is often tightly coupled to the primary view on the system: whitespace only makes sense for the given linear text layout, the code that comments relate to may become unclear, or the position in 2D space of cards may no longer make sense

in 3D space. Correspondingly, if we want to create an alternative on the system, we will likely discard or reduce secondary notation. An example of this is an outline view that show a structural hierarchy of code. Or, a documentation parser will traverse methods, fetch their signatures, fetch embedded comments, and remove indentation in the comment string for later display.

A reprojection of our cards in 3D space, may imply relayouting cards to make use of the additional dimension. In that process, the position in 2D space that was originally authored may be taken into account, or a completely new position could be derived or authored. Importantly, a lossless change in projection, both for our system and for textual programming, is difficult to achieve.

7 Conclusion

We presented a proof-of-concept of a programming environment designed to explore the idea of establishing connections between programming elements through mere proximity. We were able to successfully formulate programs of small complexity in the environment. The interactions showed promise for use cases that benefit from experimentation and revealing unexpected combinations, as moving cards across the program canvas will cause them to spontaneously connect to neighboring cards and start computing as part of the live environment. We thus hope that with further refinement, a programming environment designed around coarse, easily recombinable elements could become a helpful design metaphor for programming environments in creative scenarios.

References

1 Tom Beckmann, Joana Bergsiek, Eva Krebs, Toni Mattis, Stefan Ramson, Martin C. Rinard, and Robert Hirschfeld. Probing the Design Space: Parallel Versions for Exploratory Programming. *The Art, Science, and Engineering of Programming*, 10(1):5, February 2025. arXiv:2502.20535 [cs]. doi:10.22152/programming-journal.org/2026/10/5.

2 Tom Beckmann, Leonard Geier, Stefan Ramson, Marcel Taeumel, and Robert Hirschfeld. Syntactic Shuffle. Software, swhId: swh:1:dir:2a0397fdd156bf216400c71e2899bb512f25a316 (visited on 2025-08-29). URL: https://github.com/hpi-swa-lab/syntactic-shuffle, doi:10.4230/artifacts.24598.

3 Mary Beth Kery and Brad A. Myers. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 25–29, October 2017. ISSN: 1943-6106. doi:10.1109/VLHCC.2017.8103446.

4 Blender Online Community. Blender - a 3D modelling and rendering package, 1994. URL: https://www.blender.org/.

5 Nick Collins, Alex McLean, Julian Rohrhuber, and Adrian Ward. Live coding in laptop performance. *Organised Sound*, 8(3):321–330, 2003. doi:10.1017/S135577180300030X.

6 Godot Engine. Godot Engine - Free and open source 2D and 3D game engine. URL: https://godotengine.org/.

7 Epic Games. Unreal Engine 4, 2014.

8 Adin D. Falkoff and Kenneth E. Iverson. The evolution of APL. *SIGPLAN Not.*, 13(8):47–57, August 1978. doi:10.1145/960118.808372.

9 Anna Fuste and Chris Schmandt. HyperCubes: A Playful Introduction to Computational Thinking in Augmented Reality. In *Extended Abstracts of the Annual Symposium on Computer-Human Interaction in Play Companion Extended Abstracts*, CHI PLAY '19 Extended Abstracts, pages 379–387, New York, NY, USA, October 2019. Association for Computing Machinery. doi:10.1145/3341215.3356264.

10 T.R.G. Green and M. Petre. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages & Computing*, 7(2):131–174, June 1996. doi:10.1006/jvlc.1996.0009.

11 Brian Harvey and Jens Mönig. Lambda in blocks languages: Lessons learned. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, pages 35–38, October 2015. doi:10.1109/BLOCKS.2015.7368997.

12 Mary Beth Kery, Amber Horvath, and Brad Myers. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 1265–1276, Denver Colorado USA, May 2017. ACM. doi:10.1145/3025453.3025626.

13 Iulian Radu and Blair MacIntyre. Augmented-reality scratch: a children's authoring environment for augmented-reality experiences. In *Proceedings of the 8th International Conference on Interaction Design and Children*, IDC '09, pages 210–213, New York, NY, USA, June 2009. Association for Computing Machinery. doi:10.1145/1551788.1551831.

14 David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. Babylonian-style Programming: Design and Implementation of an Integration of Live Examples into General-purpose Source Code. *The Art, Science, and Engineering of Programming*, 3(3):9, February 2019. doi:10.22152/programming-journal.org/2019/3/9.

15 Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. Exploratory and Live, Programming and Coding. *The Art, Science, and Engineering of Programming*, 3(1):1:1–1:33, July 2018. Publisher: AOSA, Inc. doi:10.22152/programming-journal.org/2019/3/1.

16 Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: programming for all. *Commun. ACM*, 52(11):60–67, November 2009. doi:10.1145/1592761.1592779.

17 D. W. Sandberg. Smalltalk and exploratory programming. *ACM SIGPLAN Notices*, 23(10):85–92, October 1988. doi:10.1145/51607.51614.

18 Hideyuki Suzuki and Hiroshi Kato. Interaction-level support for collaborative learning: AlgoBlock—an open programming language. In *The first international conference on Computer support for collaborative learning*, CSCL '95, pages 349–355, USA, October 1995. L. Erlbaum Associates Inc. doi:10.3115/222020.222828.

19 Nathaniel Weinman, Steven M. Drucker, Titus Barik, and Robert DeLine. Fork It: Supporting Stateful Alternatives in Computational Notebooks. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–12, Yokohama Japan, May 2021. ACM. doi:10.1145/3411764.3445527.