

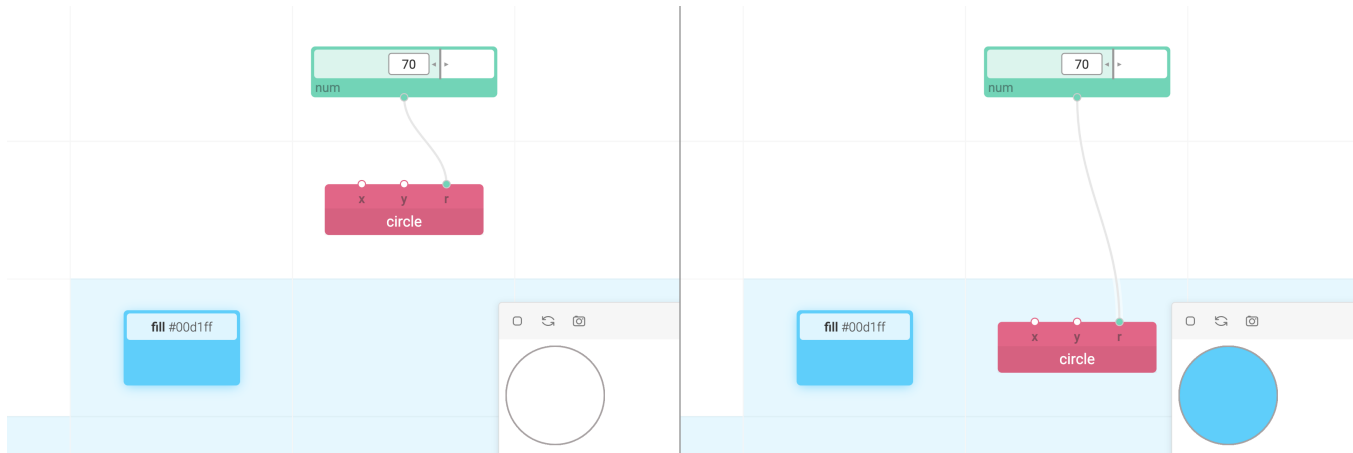
# Positional Control in Node-Based Programming

Peiling Jiang\*

peiling@ucsd.edu

University of California San Diego

La Jolla, California, USA



**Figure 1: Code blocks in b5 and the rendered result on canvas. The circle on the canvas is filled blue when the corresponding circle block is in the *effective range* of the fill block (right), whereas becomes unfilled when the circle block is moved out (left).**

## ABSTRACT

Visual programming languages enable novices to code with a lowered barrier. These languages typically employ one of two popular design approaches – block-based editing (e.g. Scratch), which allows users to control the execution order of code blocks, and node-based editing (e.g. Grasshopper), which enables users to control the data flow through nodes and wires. We propose integrating these two approaches by utilizing positional control in node-based programming to visualize and allow manipulation of both the execution order and data flow. A grid system organizes blocks and determines their sequence. Effect block is introduced, which controls other blocks within its effective range through positional constraints. As relocating blocks is easier than wiring that targets tiny inlets and outlets, we aim to shorten the feedback loop time and encourage exploration. We present b5, a web-based novel visual interface for creative coding, to demonstrate and evaluate this design.

## CCS CONCEPTS

• **Software and its engineering** → **Data flow languages; Visual languages;** • **Human-centered computing** → **Visualization.**

\*Also with New York University.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

CHI EA '23, April 23–28, 2023, Hamburg, Germany

© 2023 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-9422-2/23/04.

<https://doi.org/10.1145/3544549.3585878>

## KEYWORDS

Visual programming, Creative coding, Authoring environment

## ACM Reference Format:

Peiling Jiang. 2023. Positional Control in Node-Based Programming. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems (CHI EA '23)*, April 23–28, 2023, Hamburg, Germany. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/3544549.3585878>

## 1 INTRODUCTION

Visual programming languages and interfaces are widely used for teaching novices basic concepts of coding and computational thinking [23, 45]. They leverage shapes (e.g., code blocks, wires, etc.) as an abstraction of functions and variables, which are common concepts in traditional text-based programming. People ‘program’ by clicking, dragging and dropping, and sliding in a canvas of these graphical representations, instead of typing characters in the text editor. Creative programming, using code to create expressive and interactive media art, designs, games, and experiences, is a major theme of these systems [21, 28, 44].

This approach of authoring code has been proven to be beneficial in multiple ways for people to learn and practice programming, and later move on to more advanced text-based programming languages. Specifically, visual programming interfaces can foster creativity by encouraging experimentation, exploration, and continual evaluation of the code [37, 39, 42]. Being a better metaphor for ‘paper and pencil’ than typing, these systems also facilitate more engagement and reflection of users [32, 37, 40, 49]. On the other hand, poorly designed interfaces may otherwise hinder one’s creativity and lead to undesired coding habits [26, 49].

Two common design approaches of visual programming, which will be briefly reviewed in Section 2, allow direct control over either the execution order of the blocks of code (like Scratch [36]) or the data flow among each individual function node (like Grasshopper [38]). We identify an opportunity to integrate them by introducing positional control to the node-based data flow programming — like prior node-based systems, users can use wires to connect the outlets and inlets of blocks to create a data flow; unlike the prior, the execution order is determined solely by block positions and is independent of their wire connections, which is different from other approaches leveraging block positions [34]. We endeavor to create a better graphical representation of textual programming, where the order of a function being called and its inputs are controlled independently by the line order and function arguments.

*Effect blocks* are introduced based on this new spatial constraint. An effect block controls other blocks by positional relationships instead of wire connections. For example, Figure 1 shows a ‘circle’ block being affected by a ‘fill’ block when placed inside its *effective range*. The effect range concept inherits program ‘context’ available in graphical toolkits (e.g., HTML Canvas Rendering Context and OpenGL State Machine) that save programmers from carrying a lengthy parameter list each time they call a drawing function.

Drag-and-drop becomes an extra degree of freedom of control in node-based programming. Since relocating a block is faster and easier than wiring that targets tiny outlets and inlets, we hypothesize that this new design shortens the feedback loop time and thus encourages more exploration that can foster creativity [1, 2].

We design, develop, and open-source<sup>1</sup> a novel visual programming interface for canvas-based graphics, *b5*, to demonstrate and evaluate this idea (Figure 3).

Thus, this work-in-progress has two contributions: a design and conceptualization of positional control in node-based programming; and *b5*, an open-source web-based interface that employs the concept and enables future experimentation and evaluation.

## 2 RELATED WORK

We review prior research on the block- and node-based design approaches of visual programming, and creative coding, to frame and highlight the contributions of our work.

### 2.1 Block- and Node-Based Visual Programming

Our design inherits both the block-based and node-based editing streams of visual programming.

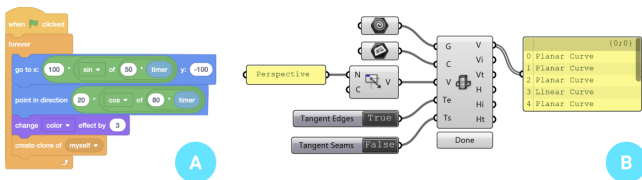


Figure 2: Example code snippets of (A) Scratch [36], and (B) Grasshopper [38].

<sup>1</sup>The source code of *b5* web editor is available at <https://github.com/peilingjiang/b5>.

**Block-based editing** allows the user to edit the blocks instead of letters to form a sequence of instructions [3]. Scratch [36] (Figure 2.A), influenced by Logo [12] and other pioneers of educational programming languages, is one of the most widely taught, used, and socially engaged. Following the principle of ‘low threshold, high ceiling, and wide walls’ [37], it is easy to get hands-on, flexible to create, and powerful enough for various kinds of production in storytelling, game, animation, etc., especially with the Build-Your-Own-Blocks (BYOB) feature introduced by Harvey and Mönig [17, 18]. Lines of program instructions are abstracted into blocks of varied shapes and colors, helping kids and novices avoid trivial typing errors that can easily frustrate them in the early stage of learning [50]. Based on the same design concept, people develop tangible versions, including one of the earliest made by McNerney [25], and more recent ones by Cardoso et al. [7] and Koushik et al. [22]. As visual or tangible blocks are easier to manipulate than text scripts, people also explore increasing accessibility in coding with block-based interfaces [27, 29].

**Node-based editing** allows the user to place individual code blocks on a plane and connect them with wires. Data flow from one executable block to another, visualized through predecessors’ outlets, successors’ inlets, and connecting wires. The design is popularly adopted by software across the fields, like Grasshopper [38] (Figure 2.B) for computational modeling, Pure Data [34] and Max [9] for music and multimedia, Unreal Blueprint [43] for game logic, and many other alternatives for these and even more professional fields, e.g., data processing and visual effects [14].

Attempts have been made to integrate the two approaches. The LEGO EV3-G programming language for its robotics kit uses a block-based design and enables quick parameter binding through wire connections [5]. However, the one-dimensional block placement makes it difficult to distinguish and scale wiring. To address this limitation and other problems in visual programming, effect blocks are introduced to manage the underlying context in programming, utilizing two-dimensional node-based positioning.

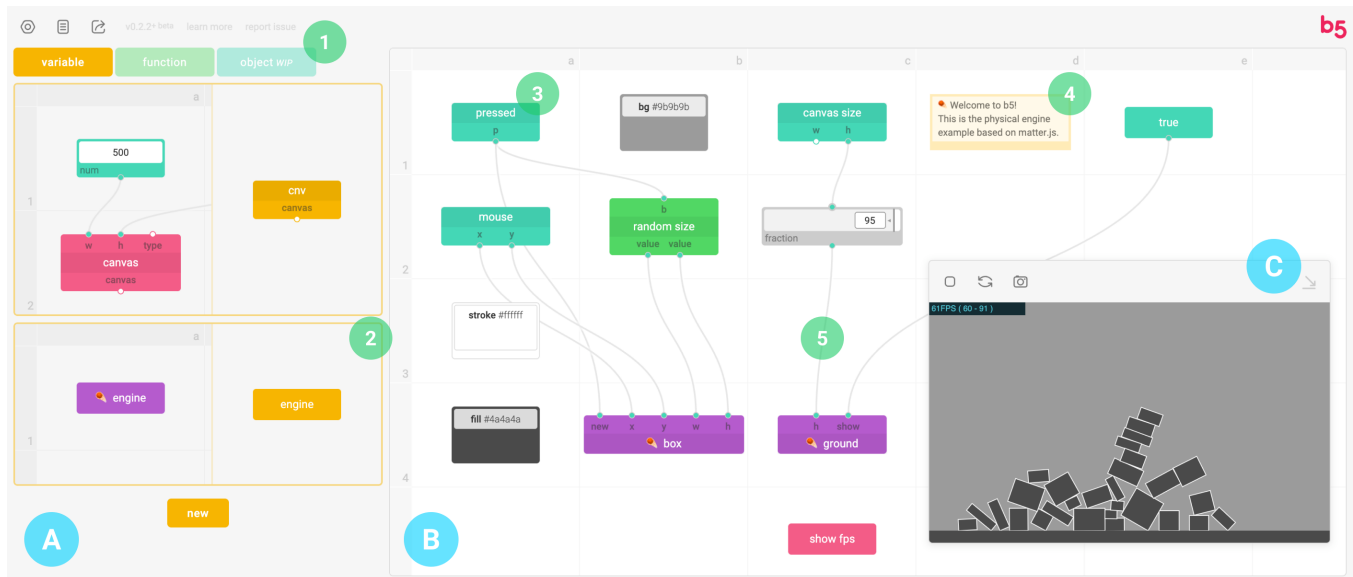
### 2.2 Creative Coding

Creative coding is a common theme for visual and educational programming. Its history can be traced back to the 1960s [28], while the recent development of personal computers and web technology, and the growing body of open-source software, learning resources, and community make it even more powerful and accessible [15]. Besides visual programming interfaces introduced above, Processing [35] is among the most popular text-based ones, which later evolved into p5.js [24] with its web-based editor [41]. Recently, Burgess et al. introduced parallel prototyping by combining a node-based interface with multiple editor-viewer pairs [6, 11].

## 3 B5 INTERFACE

*b5* is a web-based prototype environment<sup>2</sup> for our idea of positional control in node-based visual programming. Various attempts are made to lower its threshold and increase understandability when we design this interface [31]. The entire interface has only one layer with all parts tiled in the browser window. Following other systems for creative coding, *b5* consists of a viewer (Figure 3.C)

<sup>2</sup>A deployed version of *b5* web editor is available at <https://b5editor.app>.



**Figure 3: The web-based interface of b5. The interface consists of (A) Factory Panel for defining new blocks [17], (B) Playground Panel for creating live graphics and interactions, and (C) Preview Window for live previewing the resulting graphics. More specifically, components include (1) tabs to switch between lists of customized variable and function blocks, (2) defining code canvases and the previews side-by-side of each customized block, (3) executable blocks, (4) comment blocks (like sticker notes), and (5) wires connecting inputs and outputs of blocks.**

and an editor, which is further divided into two parts: the Factory Panel (Figure 3.A) for people to define their customized variable and function blocks [17]; and the Playground Panel (Figure 3.B) for creating live graphics and interactions using predefined and customized blocks [13, 35, 41]. The Factory Panel can be folded and hidden completely, and the viewer can be minimized to the right-bottom corner to minimize visual clutter.

Users can place code blocks (Figure 3.3) or comment blocks (Figure 3.4). Each code canvas can be zoomed in or out separately [4, 10, 19, 33]. A code canvas contains an array of cells unfolded on the two-dimensional plane, which is executed in order from left to right in each line, and line-by-line from top to bottom. Blocks can be dragged around but always snap to each cell after release.

Similar to the mechanism of p5.js, b5’s code canvas in the Playground Panel is like the *draw* function, executed 60 times a second (by default) in the determined sequence, and the canvases in the variable section is like being placed in the *setup* function, run once before the first execution of the main canvas in Playground and result in static outputs of the customized variable blocks. On the other hand, the customized function blocks only run after being placed into the Playground code canvas. The customized blocks can be previewed side-by-side with its defining code canvas (Figure 3.2) and spatially duplicated to the main canvas in Playground.

Like other node-based programming languages, wires create data flow between blocks. For example, in Figure 3.5, the wire shows that the output of the ‘fraction slider’ block is passed into the ‘y’ input of the ‘ground’ block. Users can click to select each block (Figure 4.B) or wire (Figure 4.C), which highlights itself and the connected wires or outlets, respectively, as visual feedback.

### 3.1 Implementation

b5 is developed with React with minimal run-time dependencies [16]. We use a customized version of q5.js (a p5.js alternative for efficiency) to render the graphics on canvas [20]. b5 programs are stored in JSON format which can be loaded by the interface later to continue. An independent module, b5.js, interprets and renders the JSON file to canvas graphics, which can work independently of our editing interface to render b5 files on other websites.

## 4 POSITIONAL CONTROL

We propose leveraging positional control in node-based programming by explicitly constraining the block positions with a grid system and using positions to determine their execution order, which is now independent of the connections between the blocks.

This new design introduces new constraints of node-based authoring. The positions a code block can be relocated to are limited by its connections — it has to be placed above all of its successors and below all of its predecessors so that the block inlets always receive outputs from previously executed blocks. For example, in Figure 4, the ‘fraction slider’ can only be placed on the second row. This constraint is visually highlighted when the user relocates a block. This constraint can lead to more organized code styles that help understand, debug, and communicate the code.

Two programs with the same blocks and wire connections can be different due to different block placements, which define execution orders. For example, in Figure 5.A, changing the position of the ‘circle’ block changes the fill color of the circle on canvas; in Figure 5.B, relocating one ‘circle’ block in relation to the others changes its fill color and drawing order at the same time.

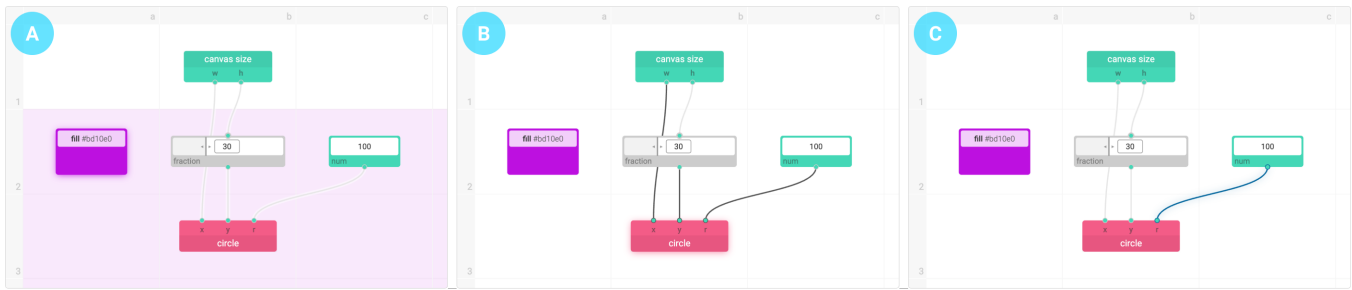


Figure 4: Selecting different components in a code canvas, i.e., (A) an effect block, (B) a regular executable block, and (C) a wire.

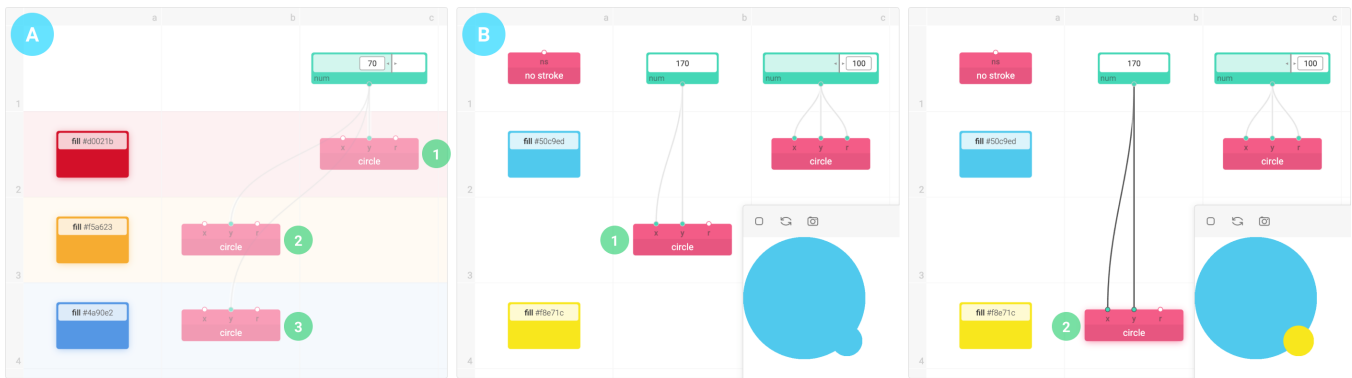


Figure 5: Positional control examples in b5.

Drag-and-drop becomes an extra degree of freedom of control in node-based programming, extending the current wire connection. Since relocating a block is faster and easier than wiring that targets tiny outlets and inlets, we hypothesize that this new way of manipulation shortens the feedback loop time and encourages more prototyping, experimenting, and exploration that can foster creativity and computational thinking [11, 23, 37, 49].

#### 4.1 Effect Block and Effective Range

As blocks are ‘sensitive’ to positions, we introduce a new type of block, *effect block*, that controls other blocks by their positional relationships instead of wire connections. More specifically, certain blocks can be ‘affected’ by the effect block once placed inside its *effective range*, which is visualized once the effect block is selected (Figure 4.A). For example, shape blocks, like the ‘circle’ block in Figure 1, are controlled by effect blocks setting fill and stroke colors.

Effect block is inspired by the mechanism of drawing contexts in graphical toolkits (e.g., HTML Canvas Rendering Context) where users can modify underlying contexts, e.g., fill color and stroke weight, that affect the rendering of the following shapes [30]. Extending this concept, in b5, (1) the effects are beyond the drawing context and apply to other aspects of a program (e.g., whether to disable a range of blocks or computations like ‘quadratic’ block in Figure 6.F, etc.), (2) the effective range of a block in b5 can vary, instead of only after the function, (3) the effective range and affected blocks are visualized on the code canvas, whereas in text-based programming, the underlying drawing context is not visually explicit

to the programmer [8]. This constraint is a 1-to-N mapping, i.e., one effect block can simultaneously affect multiple blocks, which results in a cleaner code canvas compared to the traditional node-based programs where each connection is represented by a wire.

The effective range of different effect blocks can vary. We propose 6 types of them for different effect blocks: ALL CANVAS (Figure 6.A), ALL AFTER (Figure 6.B), ALL BEFORE (Figure 6.C), INLINE (Figure 6.D), COLUMN (Figure 6.E), and AROUND (Figure 6.F). The designs and categorization are subject to further evaluation.

When a code canvas is running, effect blocks are first executed to set the ‘context’ of all other blocks based on their effective ranges. As shown in Figure 7, when multiple effect blocks present with overlapping effective ranges, there are three possible context results, which are visualized differently in the code canvas with colors:

- Bypass (Figure 7.A): two effect blocks control blocks independently, so their ranges do not interfere. For example, in Figure 5.B, the ‘no stroke’ block and the ‘fill’ block control the graphics separately at the same time.
- Overwrite (Figure 7.B): two effect blocks control the same context, so the former range is overwritten. Like in Figure 5.A, each of the first two ‘fill’ blocks’ ranges is cut by the next one and does not affect the following shape blocks.
- Merge (Figure 7.C): two effect blocks control the same context and can affect it concurrently, e.g., when placing two ‘quadratic’ blocks (Figure 6.F) next to each other, placing a number block in the overlapping area of their ranges makes its output value to the fourth power of the original value.

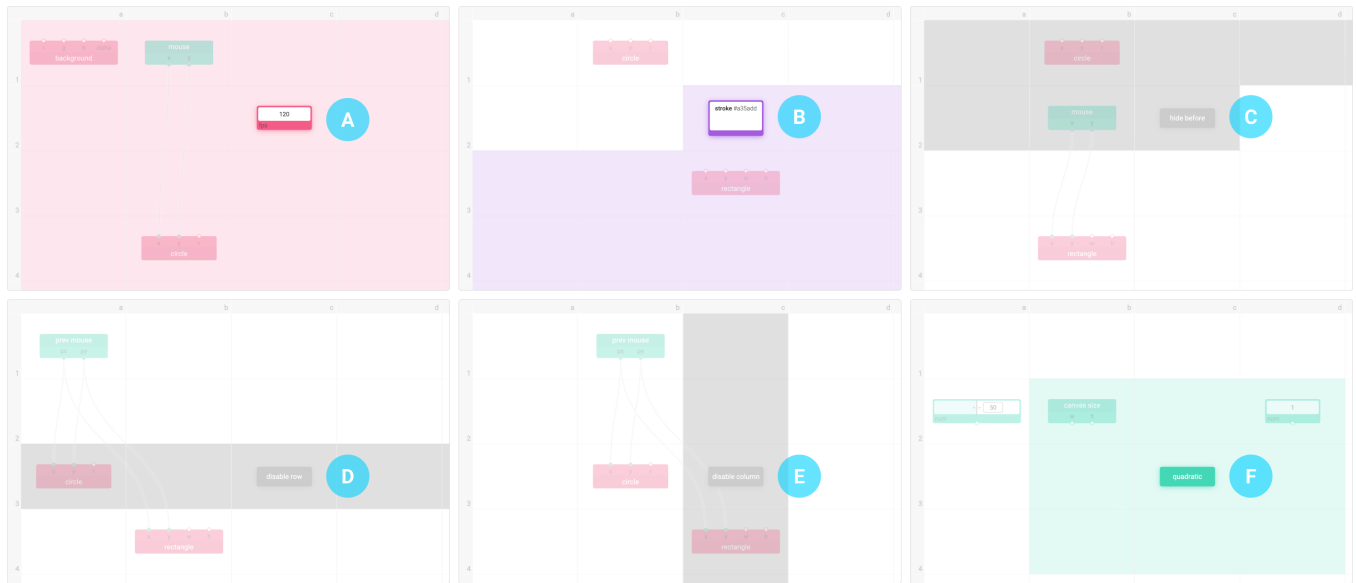


Figure 6: Different types of effective range in b5.



Figure 7: Illustrated relationships between two effective ranges.

## 5 DISCUSSION

Visual programming has been widely used for computing education and production as an alternative to text-based programming. Our idea of utilizing positional control in node-based programming endeavors to provide better control over both execution order and data flow. We hypothesize that this novel interface can make it easier to adjust the code and encourage more exploration and experimentation when learning and prototyping data flow programming.

While the position information has been used by other node-based systems before, our approach considers it as the only source of determination of execution order, visualizing it explicitly to the users alongside data flow. We then introduce effect blocks and present different kinds of effective ranges and the resulting context from their interactions. We hypothesize that this design, promoting drag-and-drop as a new way of control, can lead to more explorations and more organized code styles, and is easier to interpret.

As an early prototype, the current implementation of b5 also has limitations. First, we propose six types of effective range without explicit visualization on the blocks to state this difference. The user can only view the range when the effect block is selected, making it hard to interpret the program as a whole when there are multiple effect blocks with varied types of effective ranges. This categorization may be redundant or confusing to users and is subject to evaluation. Besides, without an organized list of code blocks, currently, users can only search for the block name or description to add new blocks, setting up extra barriers for new users.

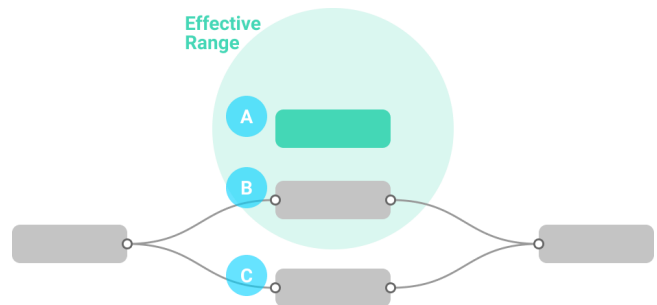


Figure 8: An hypothesized design of effect blocks in current node-based programming interfaces.

Node-based interfaces are commonly treated as ‘scripting’ interfaces instead of programming, as they normally lack control flow primitives, i.e., loop and conditional statements. With effect blocks, we can program them by placing blocks inside corresponding ranges, e.g., conditionally enabling, disabling, or repeating a range of blocks based on the input of the controlling effect blocks.

b5 is our proof-of-concept approach to positional control in node-based programming. However, the idea of spatial constraint and *position* as an extra degree of freedom of control can be independent of the design of code canvas and effect block in b5. As shown in Figure 8, to incorporate this idea in a current node-based programming interface, like Grasshopper, the user may use an effect block

with a certain effective range around itself (Figure 8.A). Placing others close to it, inside this effective range (Figure 8.B), connects them. On the other hand, treating graphic blocks as ‘objects’ with extendable properties may also open up new possibilities [46–48].

## 5.1 Future Work

Our future steps may include evaluating important components of this work, which we compile as the following research questions:

- RQ1: How to optimize the design and visualization of different types of effective ranges, and their combination?
- RQ2: What are the impacts of introducing positional control in node-based programming on programmers’ behaviors?
- RQ3: How do novice and experienced programmers leverage positional control differently, and whether this approach benefits more in educational or production settings?

Two studies, each with novice and expert programmers respectively, and the following analysis will be conducted to evaluate the concept and answer the research questions.

In Study 1 (for RQ1), we will recruit around ten expert programmers with more than eight years of programming experience. Participants will be introduced to the system and the novel approach and asked to recreate one of their existing projects using b5. We will interview participants about the system design and measure how each type of effective range is utilized in their program. We will also code participants’ exploration strategies and their code-structuring choices when they use b5.

In Study 2 (for RQ2), we will recruit around ten novice programmers with none to one year of programming experience. Participants will be introduced to the system and our novel approach to node control and will complete a series of assignment-style tasks using b5 to create simple interactive graphic projects. We will record, code, and report on participants’ usage of positional control of nodes, including any confusion or struggles they experience and the assistance provided.

Finally (for RQ3), we will compare the results from the two studies and analyze how different user groups use the system differently. With these results, we aim to evaluate the effectiveness of our novel approach for programmers of different experience levels, optimize the design of the newly introduced effect block and effective range, and evaluate b5 as a tool for learning and production.

## 6 CONCLUSION

We propose utilizing positional control in node-based programming. Besides connecting inlets and outlets with wires, people can alter their scripts by dragging and dropping code blocks. With the proof-of-concept environment, b5, we demonstrate our design of a code canvas with deterministic execution order, effect block, and effective range. Our system inherits ideas from both block-based and node-based visual programming and extends the mechanisms of drawing contexts in graphical toolkits. We hypothesize that the shorter feedback loop time, thanks to this new control of block relocation, and visualization of effective ranges can help people read and manipulate their code more easily and encourage more exploration. We plan to evaluate this novel interface with novice and experienced users to use b5 as a learning and prototyping platform.

## ACKNOWLEDGMENTS

We thank Daniel Shiffman, Allison Parrish, Shawn Van Every, Denis Pelli, Haijun Xia, and Devamardeep Hayatpur for their valuable feedback and generous assistance. We thank anonymous reviewers for their constructive and insightful reviews. This research received funding from New York University Tisch School of the Arts Undergraduate Creative Research Fund (2021).

## REFERENCES

- [1] Carol R Aldous. 2007. Creativity, Problem Solving and Innovative Science: Insights from History, Cognitive Psychology and Neuroscience. *International Education Journal* 8, 2 (2007), 176–187.
- [2] Tyler Angert. 2021. Replit Apps. <https://blog.replit.com/apps>
- [3] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. 2017. Learnable Programming: Blocks and Beyond. *Commun. ACM* 60, 6 (may 2017), 72–80. <https://doi.org/10.1145/3015455>
- [4] Benjamin B. Bederson and James D. Hollan. 1994. Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics. In *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology* (Marina del Rey, California, USA) (*UIST '94*). Association for Computing Machinery, New York, NY, USA, 17–26. <https://doi.org/10.1145/192426.192435>
- [5] Mark Bell, JAMES FLOYD, and James F Kelly. 2017. *Lego Mindstorms EV3*. Springer.
- [6] Cameron Burgess, Dan Lockton, Maayan Albert, and Daniel Cardoso Llach. 2020. Stamper: An Artboard-Oriented Creative Coding Environment. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) (*CHI EA '20*). Association for Computing Machinery, New York, NY, USA, 1–9. <https://doi.org/10.1145/3334480.3382994>
- [7] Gonçalo Cardoso, Ana Cristina Pires, Lúcia Verónica Abreu, Filipa Rocha, and Tiago Guerreiro. 2021. LEGOWorld: Repurposing Commodity Tools & Technologies to Create an Accessible and Customizable Programming Environment. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) (*CHI EA '21*). Association for Computing Machinery, New York, NY, USA, Article 273, 6 pages. <https://doi.org/10.1145/3411763.3451710>
- [8] Stéphane Conversy. 2014. Unifying Textual and Visual: A Theoretical Account of the Visual Perception of Programming Languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Portland, Oregon, USA) (*Onward! 2014*). Association for Computing Machinery, New York, NY, USA, 201–212. <https://doi.org/10.1145/2661136.2661138>
- [9] Cycling '74. 2020. Max. [cycling74.com/products/max/](http://cycling74.com/products/max/)
- [10] Robert DeLine, Mary Czerwinski, Brian Meyers, Gina Venolia, Steven Drucker, and George Robertson. 2006. Code thumbnails: Using spatial memory to navigate source code. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*. IEEE, IEEE, 11–18. <https://doi.org/10.1109/VLHCC.2006.14>
- [11] Steven P. Dow, Alana Glassco, Jonathan Kass, Melissa Schwarz, Daniel L. Schwartz, and Scott R. Klemmer. 2011. Parallel Prototyping Leads to Better Design Results, More Divergence, and Increased Self-Efficacy. *ACM Trans. Comput.-Hum. Interact.* 17, 4, Article 18 (dec 2011), 24 pages. <https://doi.org/10.1145/1879831.1879836>
- [12] Wallace Feurzeig, Seymour Papert, Marjorie Bloom, Richard Grant, and Cynthia Solomon. 1970. Programming-languages as a conceptual framework for teaching mathematics. *ACM SIGCUE Outlook* 4, 2 (1970), 13–17. <https://doi.org/10.1145/965754.965757>
- [13] Scratch Foundation. 2019. Scratch - Imagine, Program, Share. <https://scratch.mit.edu/projects/editor/>
- [14] Terkel Gjervig. 2017. Awesome Creative Coding. <https://github.com/terkel/awesome-creative-coding#visual-programming-languages>
- [15] Ira Greenberg. 2007. *Processing: creative coding and computational art*. Apress.
- [16] Philip Guo. 2021. Ten Million Users and Ten Years Later: Python Tutor’s Design Guidelines for Building Scalable and Sustainable Research Software in Academia. In *The 34th Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) (*UIST '21*). Association for Computing Machinery, New York, NY, USA, 1235–1251. <https://doi.org/10.1145/3472749.3474819>
- [17] Brian Harvey and Jens Mönig. 2010. Bringing “no ceiling” to scratch: Can one language serve kids and computer scientists. *Proc. Constructionism* (2010), 1–10.
- [18] Brian Harvey and Jens Mönig. 2017. Snap! Build Your Own Blocks. <https://snap.berkeley.edu/>
- [19] Christopher F. Herot, Richard Carling, Mark Friedell, and David Kramlich. 1980. A Prototype Spatial Data Management System. *SIGGRAPH Comput. Graph.* 14, 3 (jul 1980), 63–70. <https://doi.org/10.1145/965105.807470>
- [20] Lingdong Huang. 2021. q5.js. <https://github.com/LingDong-/q5.js> original-date: 2020-09-08T01:20:00Z.
- [21] Caitlin Kelleher and Randy Pausch. 2005. Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice

- Programmers. *ACM Comput. Surv.* 37, 2 (jun 2005), 83–137. <https://doi.org/10.1145/1089733.1089734>
- [22] Varsha Koushik, Darren Guinness, and Shaun K. Kane. 2019. StoryBlocks: A Tangible Programming Game To Create Accessible Audio Stories. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) (CHI '19). Association for Computing Machinery, New York, NY, USA, 1–12. <https://doi.org/10.1145/3290605.3300722>
- [23] Sze Yee Lye and Joyce Hwee Ling Koh. 2014. Review on teaching and learning of computational thinking through programming: What is next for K-12? *Computers in Human Behavior* 41 (2014), 51–61. <https://doi.org/10.1016/j.chb.2014.09.012>
- [24] Lauren McCarthy, Casey Reas, and Ben Fry. 2015. *Getting Started with p5.js: Making Interactive Graphics in JavaScript and Processing*. Make Community, LLC. <https://books.google.com/books?id=iP3GCgAAQBAJ>
- [25] Timothy S McNeerney. 1999. *Tangible programming bricks: An approach to making programming accessible to everyone*. Ph.D. Dissertation. Massachusetts Institute of Technology.
- [26] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. 2011. Habits of Programming in Scratch. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education* (Darmstadt, Germany) (ITiCSE '11). Association for Computing Machinery, New York, NY, USA, 168–172. <https://doi.org/10.1145/1999747.1999796>
- [27] Lauren R. Milne and Richard E. Ladner. 2018. *Blocks4All: Overcoming Accessibility Barriers to Blocks Programming for Children with Visual Impairments*. Association for Computing Machinery, New York, NY, USA, 1–10. <https://doi.org/10.1145/3173574.3173643>
- [28] Monoskop. 2012. Compos 68. [https://monoskop.org/Compos\\_68](https://monoskop.org/Compos_68)
- [29] Cecily Morrison, Nicolas Villar, Anja Thieme, Zahra Ashktorab, Eloise Taysom, Oscar Salandin, Daniel Cletheroe, Greg Saul, Alan F Blackwell, Darren Edge, Martin Grayson, and Haiyan Zhang. 2020. Torino: A Tangible Programming Language Inclusive of Children with Visual Disabilities. *Human-Computer Interaction* 35, 3 (2020), 191–239. <https://doi.org/10.1080/07370024.2018.1512413>
- [30] Mozilla. 2021. CanvasRenderingContext2D - Web APIs | MDN. <https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D>
- [31] Brad Myers, Scott E. Hudson, and Randy Pausch. 2000. Past, Present, and Future of User Interface Software Tools. *ACM Trans. Comput.-Hum. Interact.* 7, 1 (mar 2000), 3–28. <https://doi.org/10.1145/344949.344959>
- [32] Brad A. Myers. 1990. Taxonomies of Visual Programming and Program Visualization. *J. Vis. Lang. Comput.* 1, 1 (mar 1990), 97–123. [https://doi.org/10.1016/S1045-926X\(05\)80036-9](https://doi.org/10.1016/S1045-926X(05)80036-9)
- [33] Ken Perlin and David Fox. 1993. Pad: An Alternative Approach to the Computer Interface. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques* (Anaheim, CA) (SIGGRAPH '93). Association for Computing Machinery, New York, NY, USA, 57–64. <https://doi.org/10.1145/166117.166125>
- [34] Miller S Puckette. 1996. Pure Data: another integrated computer music environment. In *Proceedings of Second Intercollege Computer Music Concerts*. Tachikawa, Japan, 37–41.
- [35] Casey Reas and Benjamin Fry. 2003. Processing: A Learning Environment for Creating Interactive Web Graphics. In *ACM SIGGRAPH 2003 Web Graphics* (San Diego, California) (SIGGRAPH '03). Association for Computing Machinery, New York, NY, USA, 1. <https://doi.org/10.1145/965333.965390>
- [36] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (nov 2009), 60–67. <https://doi.org/10.1145/1592761.1592779>
- [37] Mitchel Resnick, Brad Myers, Kumiyo Nakakoji, Ben Shneiderman, Randy Pausch, and Mike Eisenberg. 2005. Design Principles for Tools to Support Creative Thinking. *Report of Workshop on Creativity Support Tools* 20 (01 2005). <https://doi.org/10.1184/R1/6621917.v1>
- [38] Robert McNeel & associates. 2007. Grasshopper. [grasshopper3d.com](http://grasshopper3d.com)
- [39] Ben Shneiderman. 2007. Creativity Support Tools: Accelerating Discovery and Innovation. *Commun. ACM* 50, 12 (dec 2007), 20–32. <https://doi.org/10.1145/1323688.1323689>
- [40] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Trans. Comput. Educ.* 13, 4, Article 15 (nov 2013), 64 pages. <https://doi.org/10.1145/2490822>
- [41] Cassie Tarakajian. 2019. p5.js Web Editor. Retrieved December 24, 2021 from <https://github.com/processing/p5.js-web-editor>
- [42] Michael Terry and Elizabeth D. Mynatt. 2002. Recognizing Creative Needs in User Interface Design. In *Proceedings of the 4th Conference on Creativity & Cognition* (Loughborough, UK) (C&C '02). Association for Computing Machinery, New York, NY, USA, 38–44. <https://doi.org/10.1145/581710.581718>
- [43] Unreal Engine. 2012. Blueprint Visual Scripting. <https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/>
- [44] Wikipedia contributors. 2021. Creative coding. [https://en.wikipedia.org/wiki/Creative\\_coding](https://en.wikipedia.org/wiki/Creative_coding)
- [45] Jeannette M. Wing. 2006. Computational Thinking. *Commun. ACM* 49, 3 (mar 2006), 33–35. <https://doi.org/10.1145/1118178.1118215>
- [46] Haijun Xia, Bruno Araujo, Tovi Grossman, and Daniel Wigdor. 2016. Object-Oriented Drawing. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) (CHI '16). Association for Computing Machinery, New York, NY, USA, 4610–4621. <https://doi.org/10.1145/2858036.2858075>
- [47] Haijun Xia, Bruno Araujo, and Daniel Wigdor. 2017. Collection Objects: Enabling Fluid Formation and Manipulation of Aggregate Selections. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) (CHI '17). Association for Computing Machinery, New York, NY, USA, 5592–5604. <https://doi.org/10.1145/3025453.3025554>
- [48] Haijun Xia, Nathalie Henry Riche, Fanny Chevalier, Bruno De Araujo, and Daniel Wigdor. 2018. *DataInk: Direct and Creative Data-Oriented Drawing*. Association for Computing Machinery, New York, NY, USA, 1–13. <https://doi.org/10.1145/3173574.3173797>
- [49] Yasuhiro Yamamoto and Kumiyo Nakakoji. 2005. Interaction Design of Tools for Fostering Creativity in the Early Stages of Information Design. *Int. J. Hum.-Comput. Stud.* 63, 4–5 (oct 2005), 513–535. <https://doi.org/10.1016/j.ijhcs.2005.04.023>
- [50] Nesra Yannier, Scott E. Hudson, Kenneth R. Koedinger, Kathy Hirsh-Pasek, Roberta Michnick Golinkoff, Yuko Munakata, Sabine Doebel, Daniel L. Schwartz, Louis Deslauriers, Logan McCarty, Kristina Callaghan, Elli J. Theobald, Scott Freeman, Katelyn M. Cooper, and Sara E. Brownell. 2021. Active learning: “Hands-on” meets “minds-on”. *Science* 374, 6563 (2021), 26–30. <https://doi.org/10.1126/science.abj9957>