



A Code-Free, Direct-Manipulation Interface for Constructing Boolean Expressions

Andrin Gasser*
ETH Zurich
Zurich, Switzerland
angasser@student.ethz.ch

April Wang
ETH Zurich
Zurich, Switzerland
april.wang@inf.ethz.ch

Sverrir Thorgeirsson*
ETH Zurich
Zurich, Switzerland
sverrir.thorgeirsson@inf.ethz.ch

Zhendong Su
ETH Zurich
Zurich, Switzerland
zhendong.su@inf.ethz.ch

Abstract

Boolean algebra is foundational to programming, yet the terse textual syntax of boolean expressions does not map clearly onto the way that students reason about logical conditions. To help bridge this gap, we introduce Boolean Canvas, a direct-manipulation interface that lets learners construct visual boolean diagrams while the corresponding Python code is generated in real time. We report on a within-subjects study with 29 tertiary-level students who solved boolean-logic tasks in Python with and without Boolean Canvas. Task success, cognitive load, system enjoyment, and perceived usability were recorded. Results show that Boolean Canvas performs similarly to a traditional code editor across objective and self-reported measures. We reflect on the design and study outcomes, identifying which features supported learning, which did not, and why, and offer evidence-based recommendations for instructors and tool builders.

CCS Concepts

• **Social and professional topics** → **Computing education**; • **Human-centered computing** → *Empirical studies in HCI*.

Keywords

boolean logic, visual programming, direct manipulation, computer science education, programming by demonstration

ACM Reference Format:

Andrin Gasser, Sverrir Thorgeirsson, April Wang, and Zhendong Su. 2026. A Code-Free, Direct-Manipulation Interface for Constructing Boolean Expressions. In *Proceedings of the 57th ACM Technical Symposium on Computer Science Education V.1 (SIGCSE TS 2026)*, February 18–21, 2026, St. Louis, MO, USA. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/3770762.3772643>

*Co-primary author



This work is licensed under a Creative Commons Attribution 4.0 International License. *SIGCSE TS 2026, St. Louis, MO, USA*
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2256-1/2026/02
<https://doi.org/10.1145/3770762.3772643>

1 Introduction

In contrast to early programming practices that relied on frequent and unstructured jumps in the code execution, structured programming is a methodology that has long become ubiquitous due to the common belief that it makes it easier to write intelligible, maintainable, and correct code [20]. Relying on a classical programming language theorem that any computable function can be represented with only three broad categories of control flow structures [3, 30], modern structured programming languages make extensive use of a small set of common ideas, for instance, that the execution of subprograms is conditional on boolean expressions. Consequently, almost all nontrivial programs written today feature at least some boolean logic, meaning that software developers must be adept at constructing and interpreting how logical operators, variables, and expressions interact within boolean logic.

However, as for many other computing topics, prior work has documented persistent student misconceptions with boolean logic, e.g., misinterpreting negation and implication, and incomplete enumeration of cases [9, 10, 24]. Furthermore, as boolean expressions grow in size, their complexity increases sharply; for every new variable that is added to an expression, the corresponding truth table becomes twice as large, making it exponentially harder to test it and maintain a correct mental model of the expression. Apart from their size, factors such as the number of negations and frequency of certain operators also make them harder to understand, even for experts [2]. For beginners, the challenge of composing and comprehending boolean expressions is further compounded by their opaqueness; when measured against Bret Victor's guidelines for learnable programming [28], the typical textual representation of a boolean formula fails on most counts. For example, there is no obvious connection between the value of the expression and its components, with each small change to the operators or variables of an expression requiring one to recalculate its value from the ground up. This makes it hard for a learner to maintain a robust mental model of the boolean state, convert their needs into code, and debug a faulty expression.

In this paper, we present a new visual system called *Boolean Canvas* that is designed to address these problems and make it easier for those with little programming experience to work with boolean logic. The underlying idea behind the system is to (1) represent the boolean state visually in a way that we believe is closer to how

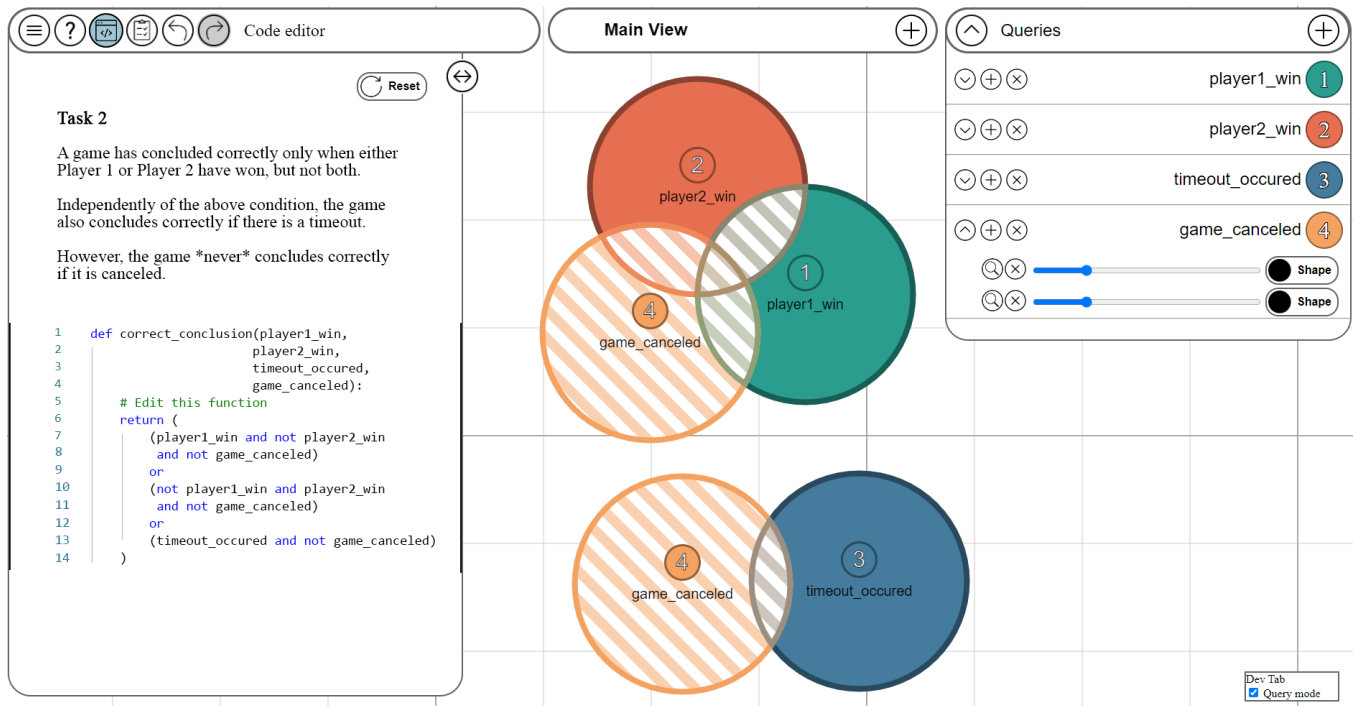


Figure 1: The system used in the study. The center of the image shows an interactive visual representation of the boolean expression that the user is working with on the left. Checkered regions indicate that the corresponding condition is false, while colored regions are true. To create additional boolean variables, the user can use the query tab on the right.

humans understand conditional logic, and (2) allow the programmer to work with this representation using direct manipulation, an interaction paradigm that allows the programmer to construct new expressions by working directly with the visual elements in question. This way, our objective is that the programmer can build complex expressions without writing any code, meaning that only a single state representation is needed for the mind and screen. To achieve this, we borrowed the Venn diagram notation, a widely used and understood visual technique for displaying set relationships and boolean algebra. While the typical passive Venn diagram has limited utility for more complex expressions, Boolean Canvas supports more control and versatility; the programmer can control the size, shape, position, and number of the shapes that are mapped to each boolean variable in a given expression, making it feasible to express some logic that would be impossible otherwise. Furthermore, the system also allows the user to construct subexpressions or variables that are represented as their own shapes. Boolean Canvas will convert the resulting diagram into Python code, making it possible to integrate the visual logic into actual programs.

In addition to presenting the tool, we also present results from a within-subjects deployment study with 29 tertiary-level students in which the system was compared to the textual language Python alone. The participants were asked to complete the same tasks involving boolean logic using a web-based Python environment with typical code editing enhancements like auto-completion, auto-pairing, and syntax highlighting, with and without Boolean Canvas.

Our objective with the study was to answer the following research questions about the system:

- RQ1** What is the difference in the task success rate between tertiary-level students who solve tasks with and without Boolean Canvas?
- RQ2** What was the difference in cognitive load between tertiary-level students solving tasks in Python with and without Boolean Canvas?
- RQ3** What differences do tertiary-level students report in perceived usability between Boolean Canvas and a code-only version of the system?
- RQ4** Is there a difference in enjoyment and interest for tertiary-level students between the Boolean Canvas and code-only conditions?

Our hypothesis was that the Boolean Canvas system would achieve better outcomes across each metric.

2 Related Work

2.1 Visual Boolean Systems

Direct manipulation interfaces reduce cognitive load by letting users operate on visible objects rather than issuing commands [11, 25], and they have been deployed with some success in computer science education [5, 6, 26, 29]. In domains where people must compose boolean conditions, such as information retrieval and database querying, this support can be especially valuable; novice users often struggle to author boolean queries, and even sustained

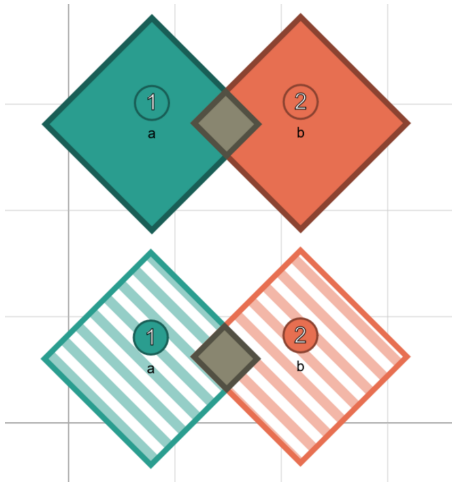


Figure 2: The visual distinction between union (top) and intersection (bottom) as it appears in Boolean Canvas with diamond shapes.

training yields modest accuracy (e.g., roughly 65% after about 14 hours of SQL practice) [21].

A recurrent design pattern in systems that implement visual support for boolean expressions is to restrict the operator palette to the three functionally complete boolean operators, i.e., and, or, and not. Any boolean expression can be constructed from these [7], a convention mirrored in many textual languages, including Python [27, p. 46]. Early visual systems exploited this idea with different composition metaphors. CUPID [16] represented conditions as nodes in a graph and combined them by connecting edges; executing the graph yielded, for example, the conjunction of two queries. Echeverria and Pino [4] proposed “query cards” that stack vertically for and, horizontally for or, with an inversion operation for not. This layout is isomorphic to Disjunctive Normal Form (DNF): multiple conjunctions (stacks) combined by disjunction (rows).

2.2 Visualizing Queries with Venn Diagrams

Venn diagrams are a visual way to illustrate interactions between sets, intuitively revealing the significance of different regions, such as the distinction between intersections and unions (Figure 2). The set of regions can be seen as analogous to DNF where each region represents a single conjunction of literals. Consequently, by toggling any combination of these regions, one can select which conjunctions appear in the DNF. For example, with two queries Q_1 and Q_2 , there exist four potential conjunctions and four regions in the Venn diagram. $(Q_1 \text{ and } Q_2)$ is the intersection between the two queries, $(Q_1 \text{ and not } Q_2)$ and $(\text{not } Q_1 \text{ and } Q_2)$ are the differences of each of them and $(\text{not } Q_1 \text{ and not } Q_2)$ the area outside of both queries. If all regions are enabled, the diagram represents a tautology. Conversely, if all regions are disabled, it represents a contradiction.

Supporting nested and complex boolean expressions in Venn diagrams can be challenging. Michard [17] presents such a system which employs Venn diagrams for visualization. The user can set multiple queries, which will be populated into an interactive Venn

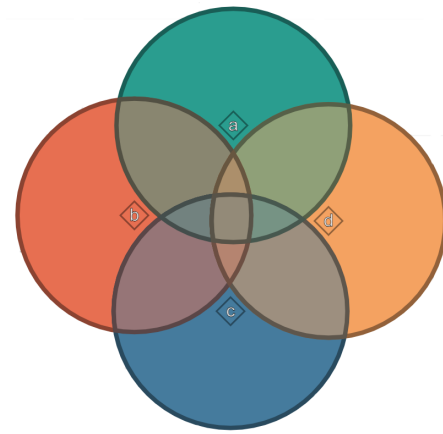


Figure 3: This Venn diagram contains four sets. Notice that two intersections cannot be shown, which is an expressiveness gap that we addressed by supporting arbitrary shapes and sub-variables.

diagram. However, it only supports up to three queries at once. For more complex expressions, the queries must be divided into separate subqueries using a MEMORIZE function. The limitation arises from the challenges of representing queries in a two-dimensional space. With four or more queries with circular shapes, not all interactions can be depicted at the same time. For example, the diagram in Figure 3 is missing $a \cap c \cap \neg b \cap \neg d$ and $\neg a \cap \neg c \cap b \cap d$. Using more complex shapes can address the issue with expressiveness, but at the cost of increased complexity for the user. If the restriction of using circular primitives is lifted, all regions can be displayed again by using a fractal-like outline. The self-similar structures can uniquely overlap with every subset of queries. However, the use of intricate shapes can make it challenging for users to interpret the diagram due to their convoluted outline.

Another approach to address the challenge is through Euler diagrams that show hierarchies and overlaps, as explored in VennDiagramWeb [13]. While Venn diagrams display all 2^n possible regions, Euler diagrams may omit some zero-value regions to simplify the representation. Additionally, instead of being limited to binary decisions it supports proportional Euler diagrams, where regions can have values greater than zero. VennDiagramWeb [13] focuses more on generating clean diagram graphics rather than providing an interactive editing environment. Peter, Flower, and Stapleton [22] add another twist to Venn and Euler diagrams by incorporating the third dimension. In contrast to the two-dimensional case, where only up to three queries can generate a Venn diagram, the 3D approach allows for the inclusion of four queries. However, even when changing the dimensions the maximum amount of queries remains limited. The focus of this paper was not on creating an intuitive user experience but rather on the mathematical conclusions drawn from this shift. Three-dimensional spaces are often considered less comprehensible as not all information can simultaneously be displayed on a 2D screen. For instance, some objects may be covered by other bodies in the foreground.

3 System

Boolean Canvas is a visual programming system that allows students to build and manipulate boolean expressions using direct manipulation, with or without the assistance of textual code (see Figure 1). To do so, one uses drag-and-drop operations on geometric forms that represent boolean variables, which we call *queries*. The queries may vary in size, shape, and number, depending on the needs of the programmer; if need be, a single boolean variable can be represented by multiple distinct visual shapes. To implement boolean logic on the queries, one can (i) change their truth value by a mouse click, and (ii) drag different queries onto each other, creating new areas that represent the intersection of distinct queries, similarly to Venn diagrams, but with booleans instead of sets.

Because Venn diagrams lose clarity with more than three or four sets, our system allows arbitrary shapes and the creation of sub-variables to avoid having to display all 2^n intersections simultaneously. When creating sub-variables, the programmer can use existing booleans as building blocks like in a textual language. This way, the user can avoid redundancy and also create more complex expressions. We also assume that non-overlapping queries on the screen semantically indicate an *or* operation, meaning that if at least one set of overlapping shapes represents a true value, so does the entire expression. Using these strategies, one can create any arbitrary boolean expression: every expression can be converted into disjunctive normal form, i.e., a disjunction of conjunctions of literals. Each disjunction is realized by placing its conjunctive groups in separate, non-overlapping areas, while a conjunction is rendered by overlaying the shapes for its positive literals and suppressing the regions corresponding to negated literals. Repeating this construction for every clause in the normal form shows that the system can depict any boolean expression one might need.

Our system treats global negation differently from a standard Venn diagram. Because many separate query groups can coexist, letting users toggle the “outside” region would be confusing, especially when queries are added or removed. We therefore removed that control and instead interpret “all regions disabled” within a group as the semantic equivalent of toggling the outside area. As a result, two classic Venn states, i.e., tautology (all regions on) and contradiction (all off), are not displayed directly; we emulate them by duplicating a query and letting it interact with itself.

The user interface of *Boolean Canvas* contains three main UI components: the viewport, the heads-up display (HUD), and the code editor. The viewport manages the area where queries can be combined to create Venn diagrams. While the viewport camera can be moved and zoomed in or out, the HUD stays static on the screen, relative to the corners of the display. The HUD enables the user to add, remove, and edit variables or queries. Additionally, a toolbar gives various tools, such as a tutorial, undo and redo functionality, and control of the textual output of the viewport. The code editor would also be subordinated to the HUD but turned into a major logic component due to its functional depth and importance in the system. It lies in the intersection between regular textual code and the visual display of our boolean query system.

In the HUD, the user can also manage the other features of the system. New queries can be created, modified, and deleted. When creating a query, a color and ID are automatically assigned to it

and the user can specify its name. Additionally, the user can also influence the shape (circle, rectangle, rhombus) and size of the primitives representing the query in the viewport. Variables can also be managed in the HUD. They allow one to simplify a boolean expression by splitting the expression into separate viewports. A variable can then be used in other viewports semantically identical to any other query. This can reduce code duplication and make the view more manageable.

The system offers some additional usability tools. The undo history enables the user to fix accidental mistakes and return back to some previous state. The help tab gives an introduction to all of the system’s main features. In the output tab, the state of the viewport is written in natural language, allowing novice programmers to verify the behavior of their constructed expression without needing to entirely understand the underlying Python code. In the menu tab, the user can switch between different tasks, an interactive tutorial, and a sandbox to test the system. Lastly, the code editor outlines the task description together with the corresponding Python code.

The system includes a Python code editor. When the user encounters a boolean expression in the code, they can use the system to verify or modify it. Any changes to the Venn diagram will automatically generate semantically identical textual code in the editor. Variables created in the viewport are also transferred to Python variables. Hovering over the different regions in the Venn diagram highlights the corresponding section in the code, allowing users to seamlessly switch between the different representations.

4 User Evaluation

After receiving ethics approval from our institution, we recruited participants with the help of ETH Zurich’s Decision Science Laboratory, which has a large volunteer pool of students enrolled in universities in Zurich, Switzerland. Qualifying participants were recruited on a first-come-first-served basis until 32 participants were found, of which 29 made an appearance on the day of the study. The inclusion criteria were to be an adult tertiary-level student, have successfully completed at least one course in computer programming, and to be familiar with the Python programming language. Participants were compensated with 25 Swiss Francs per hour.

The study duration was 1.5 hours. The study took place in a computer laboratory dedicated for empirical studies. Each participant was assigned a private booth. No external resources were allowed and students had no access to other websites other than the one designed for the study. The interactions of the participants with the system were logged using screen recordings.

After arriving in the laboratory and signing consent forms, students were asked to fill a survey (pre-test) on their demographic information and their perceptions about their skills in computer programming. The free-form questions on age and gender indicated that the median age of the participants was 23 (minimum 20, maximum 34) and that 16 identified as male and 13 as female. There were 13 participants (45%) pursuing a bachelor’s degree, 12 (41%) pursuing a master’s degree, and four (14%) pursuing a doctoral degree. 23 participants (79%) were working on a degree in a STEM subject (computer science, engineering, mathematics, physics, or biology).

	Mean	Std. Dev.	BF ₋₀
Code SUS	71.90	19.12	
Visual SUS	72.24	20.39	0.198
Code TLX	7.08	3.10	
Visual TLX	8.37	2.99	0.073
Code IMI	4.52	1.17	
Visual IMI	4.39	1.05	0.158

Table 1: Descriptive statistics and Bayes factors (BF₋₀) for system usability (System Usability Scale; SUS), cognitive load (NASA Task Load Index; TLX), and enjoyment/interest (Intrinsic Motivation Inventory; IMI).

4.1 Tasks and Surveys

The tasks designed for the study were intended to represent realistic scenarios of varying complexity that are similar to ones that students might encounter in practice. The tasks were tested with four students in a pilot setting before the study took place and their appropriateness for the purpose of the study was verified by two domain experts. Each task required participants to write boolean expressions to solve problems such as how to control a security system, manage energy usage, or decide when to engage air conditioning. To successfully compose the boolean expressions with code, the participants needed to understand concepts such as operator precedence, the effects of negation before parentheses, and the use of conjunctions and disjunctions to combine multiple conditions. The tasks can be seen in Table 2, including two practice tasks that were used to familiarize the students with the system.

Each participant was asked to solve each set of tasks once for each programming environment: once with the Boolean Canvas visualization module (the experimental condition) and once without (the control condition). To mitigate learning effects, we used counterbalancing, with fourteen randomly selected participants starting with the visual (Boolean Canvas) environment and the remaining fifteen doing the opposite. For each programming environment, the participants could work on the tasks in any order and could navigate back freely to problems they had worked on previously, but they could not switch environments until they had submitted their solutions or until forty minutes had passed.

To measure the cognitive load of the participants, their perceptions about the usability of the system, and their enjoyment of it, we used the following instruments: five of the six scales of the NASA Task Load Index (NASA-TLX) [8] (the physical effort scale excluded), the System Usability Scale (SUS) [1], and the enjoyment/interest scale of the Intrinsic Motivation Inventory [15]. Participants were asked to answer each survey twice, once for each programming environment. Lastly, we asked the participants a 7-item Likert-scale question whether they perceived the visual environment as useful for learning about boolean expressions.

4.2 Analysis

Our methods and results from the study are reported using the ACM SIGSOFT Empirical Standards for Software Engineering [19], a set of criteria that was designed to help contribute to uniformity

in the research methodology and review process [18]. The criteria includes items such as stating the formal hypotheses of the study, justify the use of one-sided hypothesis testing, justify the choice of sample size, and others.

We used Bayesian statistical analysis to compare the task performance, cognitive load, usability and enjoyment across the two environments. To compare the performance difference (RQ1), we used a Bayesian independent-samples Mann–Whitney U test (10,000 samples) with only the performance results from the first environment that each student encountered. To compare the difference between the two environments across the survey instruments (RQ2, RQ3 and RQ4), we used the Bayesian Paired Samples Test (Wilcoxon signed-rank) with 10,000 samples. These nonparametric tests do not assume normality. In both cases, we applied one-sided hypothesis testing. We also calculated the summary statistics of the survey question on whether the participants perceived the system as useful for learning. All of the above was calculated using the JASP statistical analysis software [14].

5 Results

The non-practice tasks were graded on the basis of correctness. A slightly higher proportion of tasks was solved by the experimental group (visual group), with an effect size of 0.38, indicating moderately better performance when using the visual system. However, this difference has a BF₁₀ value of 0.720 after applying the Mann–Whitney U Test, indicating that the results may have occurred due to chance. On average, the participants achieved a score of 0.53 (SD = 0.24) using Boolean Canvas without code, and 0.47 (SD = 0.32) with code.

The participants considered the system useful for learning about boolean expressions; on the 7-item Likert scale with a question about whether they agreed with that statement, they gave an average response of 5.62 (standard deviation 1.50).

The mean survey scores (see Table 1) indicate that the perceived usability of the code-only environment (the control environment) and the experimental environment were very similar and the participants also enjoyed the two environments approximately equally. There was a larger difference in cognitive load, with students experiencing on average higher cognitive load in the experimental environment. The Bayes factors were all well below one, indicating that the evidence for the alternative hypotheses (that the experimental environment would induce lower cognitive load, higher enjoyment, and higher perceived usability than the control environment) was much lower than the evidence for the null hypotheses. For instance, the Bayes factor for the usability (SUS) comparison was 0.198, meaning that the data indicates that the null hypothesis was about five times (1 divided by 0.198) more likely than the alternative hypothesis.

The mean usability scores for both systems (72) were slightly above the “widely reported” [12] benchmark of 68, which is the average value of SUS scores across a large number of studies [23]. A score of 72 indicates that the usability was perceived as “good” [23].

6 Discussion and Conclusion

Overall correctness did not show strong evidence for a difference relative to text-only coding. By a post hoc analysis that combines

Task #	Scenario	Description	Conditions
P1	Security System	Implement a function where the alarm sounds if and only if a person is detected and the door is locked.	2
P2	Energy Saving	Define a function for a smartphone to enter energy saving mode if the battery is below 20% and it is not plugged in.	2
T1	Athletes	Create a function to list athletes participating in exactly two out of fencing, tennis, and volleyball.	3
T2	Air Conditioning	Design logic to turn on the AC if the temperature is in a certain range, with additional conditions when not at home.	4
T3	Game Outcome	Write a function to determine if a game concludes correctly based on wins, timeouts, or cancellations.	4
T4	Raincoat Requirement	Develop a function to decide if a raincoat is needed based on weather conditions and personal items.	4
T5	Picnic	Establish a function to decide going to a park based on weather, availability of friends, holidays, and concerts.	5
T6	Alarm Setting	Construct a function to set an alarm based on work, school, events, health, and day of the week.	6

Table 2: Overview of the eight tasks used in the study, including the two practice tasks.

effect-size estimates with screen-capture logs, we can add some nuance to an otherwise null aggregate result. We observed that the visual condition produced a moderate performance advantage ($d=0.38$) and noticeably fewer wholly incorrect submissions from participants who self-reported lower Python confidence. Screen recordings show that these students made heavy use of the region hovering feature in our system to find corresponding parts of the code. In cognitive-load terms, these features may have reduced the intrinsic load of the task (the load imposed by boolean logic itself) by externalizing otherwise invisible intermediate states, echoing Victor’s “immediate connection” principle [28]. Secondly, bidirectional synchronization between the diagram and live Python may have lowered the “gulf of evaluation” for participants with lower computational proficiency. Our exploratory hypothesis, driven by the data, is that the tool in its current state is primarily useful for novices, and possibly that it would be most useful in secondary education.

We also identified aspects of the system that should be improved. Participants spent non-trivial time panning, zooming, and manually re-arranging shapes to avoid occlusion. This overhead inflates extraneous load that is unrelated to learning goals. Automated placement would reduce manual camera work and free cognitive resources for reasoning. Additionally, offering a miniature truth table on hover for the current selection might improve performance.

6.1 Implications for Instructors

In light of the results from our study, which suggest that the system works best for novices, we speculate that the deployment of Boolean Canvas should precede formal instruction on boolean operator precedence, rather than function as a stand-alone tool. By letting students see precedence rules for themselves instead of memorizing them, the tool can serve as a conceptual bridge and later as a lightweight debugger once they move to text-only tasks. For that

reason, instructors should position activities with Boolean Canvas as formative rather than summative, i.e., assess the quality of students’ reasoning or their final Boolean Canvas artifact, but avoid grading the process itself (such that learners who have internalized the visual workflow will not be penalized when asked to write logic unaided). The shared visual state may also be useful for pair-programming exercises, for example rotating the driver-navigator roles between diagram manipulation and code editing could plausibly help students develop a greater conceptual understanding of the topic.

6.2 Implications for Tool Builders

For our own development, our findings point to a clear next step: embed Boolean Canvas *in situ* inside mainstream code editors rather than hosting it as a stand-alone sandbox. Throughout the study, participants treated the auto-generated Python as the authoritative representation, and since this was already central to their workflow, collapsing the visual layer directly into the editor pane would remove costly mode switches and reduce extraneous cognitive load (as seen by frequent panning and zooming). An in-editor plug-in would also be useful for intermediate students, as they could invoke the visualization selectively and for more complex expressions only. We believe that this suggestion would be consistent with a general principle for tool builders outside of boolean logic, namely that learning aids that live where work already takes place can better support transfer to authentic programming tasks. For Boolean Canvas, this should be corroborated in future studies.

References

- [1] Aaron Bangor, Philip T Kortum, and James T Miller. 2008. An empirical evaluation of the system usability scale. *Intl. Journal of Human-Computer Interaction* 24, 6 (2008), 574–594.
- [2] Aviad Baron, Ilai Granot, Ron Yosef, and Dror Feitelson. 2024. Understanding Logical Expressions with Negations: Its Complicated. In *Proceedings of the 28th*

- International Conference on Evaluation and Assessment in Software Engineering*. 303–312.
- [3] Corrado Böhm and Giuseppe Jacopini. 1966. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM* 9, 5 (1966), 366–371.
 - [4] Luz E. Echeverria and José A. Pino. 1989. An intuitive approach for the expression of Boolean queries. *[Proceedings] 1989 IEEE Workshop on Visual Languages* (1989), 118–123. <https://api.semanticscholar.org/CorpusID:2898089>
 - [5] Patrice Frison. 2014. AlgoTouch: a programming by demonstration tool for teaching algorithms. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education* (Uppsala, Sweden) (ITiCSE '14). Association for Computing Machinery, New York, NY, USA, 334. doi:10.1145/2591708.2602654
 - [6] Maximilian Georg Barth, Sverrir Thorgeirsson, and Zhendong Su. 2024. A Direct Manipulation Programming Environment for Teaching Introductory and Advanced Software Testing. In *Proceedings of the 24th Koli Calling International Conference on Computing Education Research (Koli Calling '24)*. Association for Computing Machinery, New York, NY, USA, Article 2, 11 pages. doi:10.1145/3699538.3699564
 - [7] David Money Harris and Sarah L. Harris. 2012. *Digital Design and Computer Architecture, 2nd Edition*. Morgan Kaufmann, Burlington, MA.
 - [8] Sandra G Hart. 2006. NASA-task load index (NASA-TLX); 20 years later. In *Proceedings of the human factors and ergonomics society annual meeting*, Vol. 50. Sage publications, Sage CA: Los Angeles, CA, 904–908.
 - [9] Geoffrey L. Herman, Lisa Kaczmarczyk, Michael C. Loui, and Craig Zilles. 2008. Proof by incomplete enumeration and other logical misconceptions. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia) (ICER '08). Association for Computing Machinery, New York, NY, USA, 59–70. doi:10.1145/1404520.1404527
 - [10] Geoffrey L. Herman, Michael C. Loui, Lisa Kaczmarczyk, and Craig Zilles. 2012. Describing the What and Why of Students' Difficulties in Boolean Logic. *ACM Trans. Comput. Educ.* 12, 1, Article 3 (March 2012), 28 pages. doi:10.1145/2133797.2133800
 - [11] James Hollan, Edwin Hutchins, and David Kirsh. 2000. Distributed cognition: toward a new foundation for human-computer interaction research. *ACM Trans. Comput.-Hum. Interact.* 7, 2 (jun 2000), 174–196. doi:10.1145/353485.353487
 - [12] Maciej Hyzy, Raymond Bond, Maurice Mulvenna, Lu Bai, Alan Dix, Simon Leigh, Sophie Hunt, et al. 2022. System usability scale benchmarking for digital health apps: meta-analysis. *JMIR mHealth and uHealth* 10, 8 (2022), e37290.
 - [13] Felix Lam, Christopher M. Lalasingh, Holly E. Babaran, Zhiyuan Wang, Stephanie D. Prokopec, Natalie S. Fox, and Paul C. Boutros. 2016. VennDiagramWeb: a web application for the generation of highly customizable Venn and Euler diagrams. *BMC Bioinformatics* 17, 1 (Oct. 2016), 401. doi:10.1186/s12859-016-1281-5
 - [14] Jonathon Love, Ravi Selker, Maarten Marsman, Tahira Jamil, Damian Dropmann, Josine Verhagen, Alexander Ly, Quentin F Gronau, Martin Šmíra, Sacha Epskamp, et al. 2019. JASP: Graphical statistical software for common statistical designs. *Journal of Statistical Software* 88 (2019), 1–17.
 - [15] Edward McAuley, Terry Duncan, and Vance V Tammen. 1989. Psychometric properties of the Intrinsic Motivation Inventory in a competitive sport setting: A confirmatory factor analysis. *Research quarterly for exercise and sport* 60, 1 (1989), 48–58.
 - [16] Nancy H. McDonald and Michael Stonebraker. 1975. CUPID - The Friendly Query Language. In *ACM Pacific*. <https://api.semanticscholar.org/CorpusID:12369458>
 - [17] A. MICHARD. 1982. Graphical presentation of boolean expressions in a database query language: design notes and an ergonomic evaluation. *Behaviour & Information Technology* 1, 3 (1982), 279–288. arXiv:<https://doi.org/10.1080/01449298208914452> doi:10.1080/01449298208914452
 - [18] Paul Ralph. 2021. ACM SIGSOFT empirical standards released. *ACM SIGSOFT Software Engineering Notes* 46, 1 (2021), 19–19.
 - [19] Paul Ralph, Nauman bin Ali, Sebastian Baltes, Domenico Bianculli, Jessica Diaz, Yvonne Dittrich, Neil Ernst, Michael Felderer, Robert Feldt, Antonio Filieri, et al. 2020. Empirical standards for software engineering research. *arXiv preprint arXiv:2010.03525* (2020).
 - [20] Edwin D Reilly. 2003. Structured programming. In *Encyclopedia of Computer Science*. 1701–1708.
 - [21] Phyllis Reisner. 1981. Human factors studies of database query languages: A survey and assessment. *ACM Computing Surveys (CSUR)* 13, 1 (1981), 13–31.
 - [22] Peter Rodgers, Jean Flower, and Gem Stapleton. 2012. Introducing 3D Venn and Euler Diagrams. In *Proceedings of the 3rd International Workshop on Euler Diagrams 2012*, Peter Chapman and Luana Micallef (Eds.). *Proceedings of the 3rd International Workshop on Euler Diagrams 2012* 854, 92–106. <https://kar.kent.ac.uk/30797/>
 - [23] Jeff Sauro and James R Lewis. 2016. *Quantifying the user experience: Practical statistics for user research*. Morgan Kaufmann.
 - [24] Marko Schmellenkamp, Alexandra Latys, and Thomas Zeume. 2023. Discovering and Quantifying Misconceptions in Formal Methods Using Intelligent Tutoring Systems. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (Toronto ON, Canada) (SIGSE 2023). Association for Computing Machinery, New York, NY, USA, 465–471. doi:10.1145/3545945.3569806
 - [25] Sverrir Thorgeirsson, Oliver Graf, and Zhendong Su. 2024. The Hidden Program State Hurts Everyone. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Pasadena, CA, USA) (Onward! '24). Association for Computing Machinery, New York, NY, USA, 266–274. doi:10.1145/3689492.3689813
 - [26] Sverrir Thorgeirsson and Zhendong Su. 2021. Algot: An Educational Programming Language with Human-Intuitive Visual Syntax. *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2021), 1–5. <https://api.semanticscholar.org/CorpusID:240156985>
 - [27] Guido van Rossum. 2003. *An Introduction to Python* (release 2.2.2 ed.). Network Theory Limited, 15 Royal Park, Clifton, Bristol BS8 3AL, United Kingdom. A catalogue record for this book is available from the British Library.
 - [28] Bret Victor. 2012. Learnable programming: Designing a programming system for understanding programs. URL: <http://worrydream.com/LearnableProgramming> (2012).
 - [29] Theo B. Weidmann, Sverrir Thorgeirsson, and Zhendong Su. 2022. Bridging the Syntax-Semantics Gap of Programming. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Auckland, New Zealand) (Onward! 2022). Association for Computing Machinery, New York, NY, USA, 80–94. doi:10.1145/3563835.3567668
 - [30] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. 2008. Reversible flowchart languages and the structured reversible program theorem. In *Automata, Languages and Programming: 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II* 35. Springer, 258–270.