Toward a VR-Native Live Programming Environment

Leonard Geier
leonard.geier@student.hpi.uni-potsdam.de
Hasso Plattner Institute
Potsdam, Germany
University of Potsdam
Potsdam, Germany

Clemens Tiedt
clemens.tiedt@student.hpi.uni-potsdam.de
Hasso Plattner Institute
Potsdam, Germany
University of Potsdam
Potsdam, Germany

Tom Beckmann
tom.beckmann@hpi.uni-potsdam.de
Hasso Plattner Institute
Potsdam, Germany
University of Potsdam
Potsdam, Germany

Marcel Taeumel
marcel.taeumel@hpi.uni-potsdam.de
Hasso Plattner Institute
Potsdam, Germany
University of Potsdam
Potsdam, Germany

Robert Hirschfeld
robert.hirschfeld@uni-potsdam.de
Hasso Plattner Institute
Potsdam, Germany
University of Potsdam
Potsdam, Germany

Abstract

Fast feedback loops between performing code changes and seeing their outcome help developers to be productive. For development of virtual reality (VR) applications, developers use a separate device, forcing them to switch devices whenever they want to test their application, thus significantly increasing the length of the feedback loop.

In this paper, we describe a prototypical development environment that allows writing VR applications while inside VR. Unlike previous work in this area that projected traditional 2D editors into the 3D world, we explore the use of direct manipulation in a structured editor for the general-purpose programming language Smalltalk. We present and discuss insights from a preliminary user study with four participants. Our findings demonstrate that the concept does work if users are given prior instructions, especially for smaller features where direct feedback is valuable, but ergonomics of both the hardware and our prototype have to be improved before extended programming sessions are viable.

CCS Concepts: • **Software and its engineering** → *Formal language definitions*; *Visual languages*; • **Human-centered computing** → **Virtual reality**.

Keywords: virtual reality, programming environment, live programming, vr-native

PAINT '22, December 05, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9910-4/22/12.
https://doi.org/10.1145/3563836.3568725

ACM Reference Format:
Leonard Geier, Clemens Tiedt, Tom Beckmann, Marcel Taeumel, and Robert Hirschfeld. 2022. Toward a VR-Native Live Programming Environment. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT '22), December 05, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 9 pages.
https://doi.org/10.1145/3563836.3568725

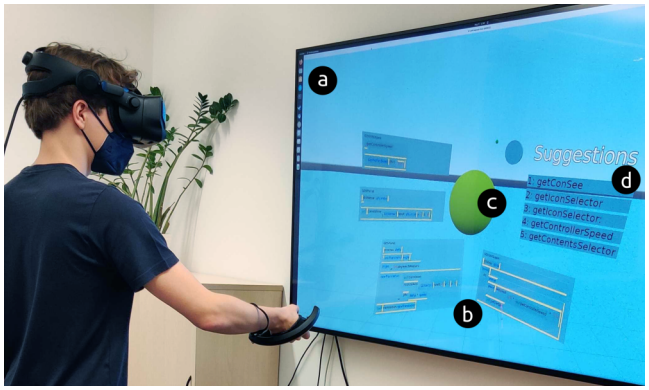

Figure 1. A user of our prototypical VR programming environment, the monitor (a) shows the view the user is seeing through the headset. Four methods (b) are arranged around the VR application the user is editing, a solar system simulation partly visible in the form of the yellow sphere (c). The user is in the process of typing a selector, a popup (d) is offering corresponding autocompletions to select from.

1 Introduction

Live programming techniques [18] support programmers by shortening feedback cycles between making a change and seeing its result. Through reduced context switches,

live programming helps programmers reach a flow state by eliminating much of the overhead that comes with toolchains that require compilation or program restarts.

Live programming techniques can also be applied when developing applications for virtual reality (VR) and thus often allow changes to become live in under a second. However, for VR development, programmers no longer use keyboard and mouse to interact with the resulting program and instead have to change to a VR headset and VR controllers to test their changes. While a change in input and output devices also typically occurs for mobile or console development, the cost of switching between devices is significantly higher for VR, as the devices have to be fitted to one's body each time. Our goal is thus to remove this cost of switching input and output devices to support users in reaching a state of flow, where the result of changes can be observed near-instantly.

In this paper, we contribute a novel means of interacting with a *VR-native* programming editor in VR as shown in Figure 1, as well as a preliminary evaluation and discussion of our design decisions through feedback from a user study. We introduce the term *VR-native* to refer to interactions that make use of dedicated VR hardware such as the controllers and headset, rather than traditional keyboards or a desk setup. More specifically, we investigate the ideas of structured editing and direct manipulation of the program syntax tree to allow programmers to edit their VR applications while they are inside virtual reality. To support a basic live programming workflow in VR with these constraints, we identified as the four major building blocks

1. means to display and edit the syntax tree,
2. means to input text,
3. means to browse code, and
4. means to execute the code.

We aim to eventually provide a programming environment where it is possible for programmers to stay in virtual reality for most of the development of their VR applications. Note that for many people, extended sessions in virtual reality may induce a feeling of simulation sickness, especially if they are not yet used to working in VR. We expect that with advances in VR hardware in the next years this factor will decrease. Additionally, the length of development sessions may be limited as interactions in VR can be comparatively exhausting, especially if they require the user to raise their hands. In the future, this may be remedied by lighter controllers or hand tracking to an extent.

In the following, we will first discuss the VR-native term and existing solutions to program in VR (section 2). Next, we describe our concept for a VR-native development environment (section 3) and briefly aspects of its implementation (section 4). We then present insights we gathered from test runs with users (section 5). We end with a discussion of our concept (section 6), before concluding the paper (section 7).

Figure 2. Picture of the Valve Index controller. The index finger is placed at the trigger button. Middle, ring, and little finger determine the strength of the grip input. Using their thumb, users can control a touchpad, a thumbstick, and several buttons.

2 Programming in VR

In this section, we briefly describe constraints imposed by the current VR hardware, describe the term VR-native, and finally relate the term to prior and related work.

2.1 Hardware

Current consumer VR headsets do not differ greatly in the interactions they imply. The headset tracks the user's movement and displays a correspondingly transformed view into a 3D world. Future developments in headsets relevant to our use case concern primarily comfort in wearing the headset, most influenced by weight, as well as quality of rendering through focus and resolution, to allow displaying text in smaller sizes.

For input, there is one controller for each hand. A picture is shown in Figure 2. Controllers feature a trigger button at the index finger, which is typically the main interaction, corresponding to a left-click using a mouse. The thumb typically has access to an array of buttons, touchpads, or thumbsticks akin to a gamepad. In addition, most controllers have a "grip" input, which is either on the middle finger (e.g., Oculus Touch Controller) or detects strength of the user's grip on the controller of middle, ring, and small finger separately (e.g., Valve Index Controller). Some headsets also feature experimental hand tracking that allows users to perform gestures with their fingers to provide input to applications instead of controllers.

2.2 VR-native Programming

As mentioned in section 1, we use the term *VR-native* to refer to interactions that

1. make use of dedicated VR hardware, rather than traditional keyboards or a desk setup, and
2. do not tie users to a specific physical location for their desired interactions.

Interpreting this concept in the context of a programming environment, we want users to be able to perform changes or investigate problems as and where they happen to support temporal and spatial immediacy [23] when debugging in VR. As such, removing the headset or even walking towards a desk with the headset on may be impractical, as behavior of code in VR often depends on the physical location of the user.

This implies that users do not have access to a physical keyboard, which participants in our study linked to a feeling of productivity when programming. We thus hypothesize that interactions that minimize the amount of text that needs to be entered are more suited to a VR-native setting. These could take the form of direct-manipulation via drag-and-drop of program elements, as is common in some structured editors such as Scratch [19]. As VR controllers are directly tied to the user's hands, drag-and-drop feels natural in a VR environment and is thus also an often used means of interaction in VR applications [5].

Based on observations by the authors, we hypothesize that benefits of VR-native programming could include more immediate feedback for the development of VR applications, a richer display of both static and dynamic information through use of the third dimensions, or intuitive direct-manipulation tools for interacting with code and tools. In this paper, we focus on providing users with immediacy of feedback when developing VR applications; investigation of other potential benefits that may come with a VR-native programming environment is future work not covered by this paper.

2.3 Related Work

For our discussion of related work we refine the term VR-native further to distinguish between *native interactions*, involving the controllers and headset, and *native tools*. Native tools, in this case, refer to tools that make use of 3D space for user interface elements rather than projecting entire 2D applications into 3D space and using the controllers or a mouse as pointers.

Non-native Interactions, Non-native Tools. The simplest means to start programming in virtual reality are desktop mirroring tools that project a user's regular monitor into a 3D scene [11, 21]. Naturally, the physical size and number of monitors is not limited in the virtual space; some approaches also mirror individual applications and allow users to arrange windows independent of a monitor [21]. Here, interactions may occur via controllers and a virtual keyboard but tend to also support input via the user's physical mouse and keyboard.

Non-native Interactions, Native Tools. A number of prototypical programming environments bring programming tools into the VR environment but rely only on text to represent code [2, 4, 10, 13, 20]. Textual code is typically shown in a rectangular plane that can be placed freely in 3D space. Again, input is given either through a fully virtual keyboard to be used with the VR controllers or through a physical keyboard.

Native Interactions, Native Tools. A set of applications embraces both native interactions and makes full use of 3D space. However, the applications tend to be focused on a specific aspect of software development [9, 14]. In particular for software visualization and program understanding, many tools exist that demonstrate benefits compared to their equivalents on a 2D screen. Most attribute this to the additional space available to users and the spatial understanding users form as they interact in 3D space with the domain objects. Other tools reduce the scope from general-purpose programming to specific problem spaces that are well-suited for VR, such as creative coding [22]. These tend to employ node-and-wire visualizations for their domain-specific languages.

3 A VR-native Live Programming Environment

In section 1, we outlined four factors that we consider relevant to allow programming in virtual reality via direct manipulation in a structured editor to enable VR-native editing. In the following, we will describe each concern and our approach to realize the concern in virtual reality.

3.1 Syntax Tree Display and Modification

Our programming environment prototype uses Smalltalk, hosted on top of the Squeak/Smalltalk [12] live programming system. To display Smalltalk code in our 3D world, we take the syntax tree as produced by the system's parser and define a mapping to labels and rectangular cuboids, which we will refer to as blocks. Our mapping resembles a mapping discussed in prior work [1] but adds depth to each level of nesting such that blocks have volume users can aim at when grabbing. As Smalltalk is entirely expression-oriented, a mapping via the jigsaw-puzzle metaphor as used in Scratch [19] or Blockly [8], where a distinction between statements and value expressions is made, is impractical.

The programmer's central means of interaction in our design is a small wand, of which they carry one in each hand. The wand functions as a precise pointing device, with a diameter which tapers towards 2mm at the far end. Whatever object the tip of the wand intersects with is considered as targeted. We tested some different lengths of the wand with users. The shorter the wand, all the way to no wand at all, the more users experienced the interactions with blocks as direct. However, the longer the wand the less movement of users' hands and arms was required to target blocks. After some testing, we opted for a wand of around 20cm in virtual space, which appeared to provide a compromise between both concerns.

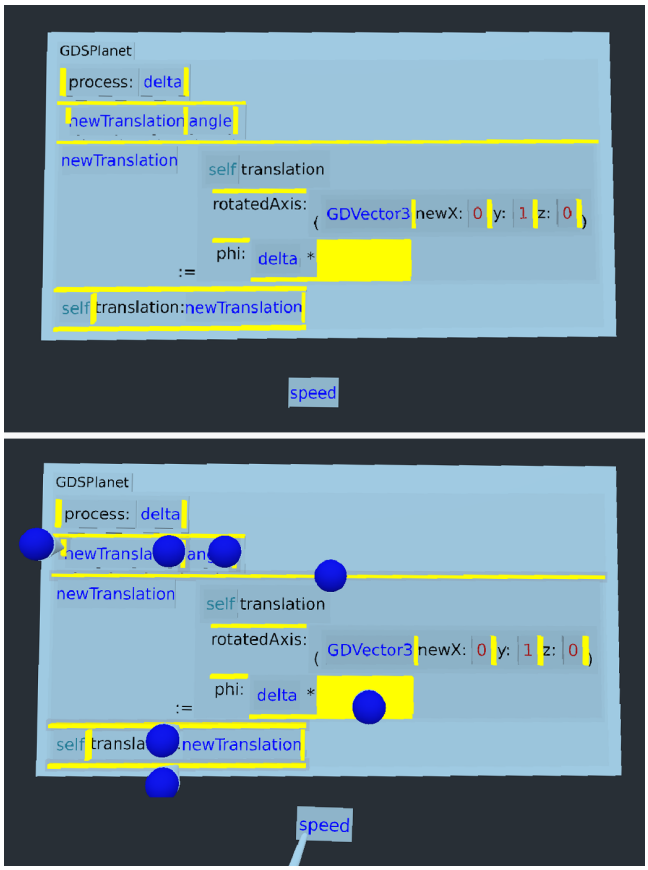

Figure 3. At the top, a collection of blocks forming a method and a single block containing the identifier "speed" are depicted. After picking up the identifier block (below), blue spheres mark possible insert positions in the method.

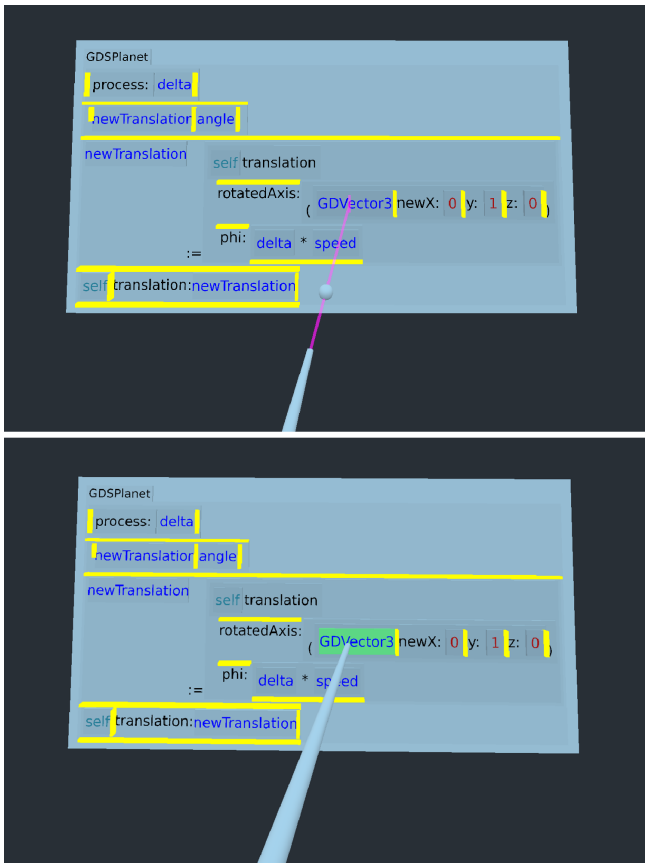

Figure 4. At a distance (top), the wand allows placing a cursor on text while showing a preview of where it is currently aimed at. Once the wand touches a block (bottom), the block is highlighted and may be picked up.

Users have two means to interact with the program blocks. First, akin to drag-and-drop in the desktop metaphor, users can pick up blocks by pointing their wand at them and pressing and holding the trigger button. These interactions mimic those used in block-based programming environments [17, 19]. Once picked up, the selected block and the blocks it contains are removed from the block it was previously attached to and are instead attached to the wand. The user can let go of the blocks in open 3D space to temporarily set them aside; or, they can target gaps between blocks and places where a block is missing which we highlight as shown in Figure 3 to insert blocks.

Second, they can place a text cursor inside a label of a block by touching a label and pressing the trigger button. As both interactions, grabbing blocks and placing a cursor, involve pointing the wand at a block and pressing a button, a means of distinction between the interactions is needed. Options we considered include allowing users to switch between a drag and a text wand; using different buttons for drag and text; eagerly placing the text cursor and only starting a drag if a threshold of movement is exceeded; and segmenting the wand in a text and drag section. The latter is the option we eventually settled on using, as shown in Figure 4: as the wand nears a text field, we display a faint preview of where the cursor would be placed if the user would press the trigger button at the current position. Once the wand begins intersecting a block, we no longer display a cursor preview, as pressing trigger would now begin dragging the block. We found this design to form a good compromise between accuracy and overhead of mode switching. Once a text cursor is placed, users can start character input, which is further described in the next subsection.

3.2 Textual Input

Textual input allows modifying labels in blocks. In addition, it is also the only way for users to create new blocks. Contrarily to block-based editors that usually employ a form of block palette for creating blocks, users in our prototype point at an insert position and begin typing the syntactic

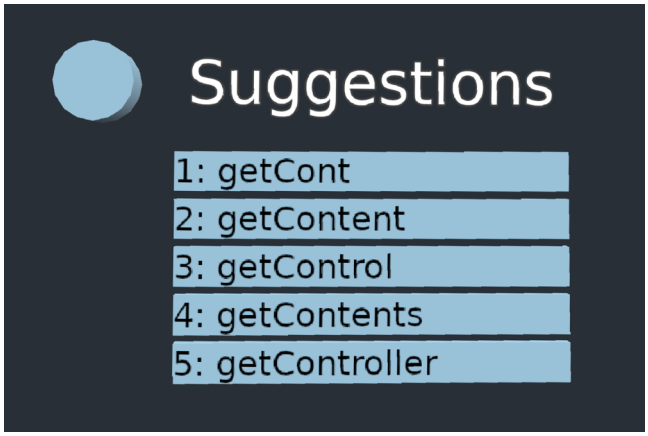

Figure 5. The suggestions menu after the user enters the string "getCont" into a block.

construct they want to create. As soon as their input is syntactically unambiguous, the construct is created as a block at the position. Further, automatic transformations similar to the GrammarCells system for MPS [24] allow users to enter expression such as 2 + 3 exactly as in a text editor; the corresponding number and binary operator blocks are created automatically by the editor.

We consider this last part essential to form a middleground between direct manipulation and efficient creation of blocks. In particular general-purpose programming languages and their APIs tend to require users to combine a large number of various language constructs into into larger expressions. Domain-specific, visual editors, such as Scratch [19] or Unreal Blueprints [7] instead tend to have few, pre-combined, and expressive language constructs, such that selection via a palette becomes feasible.

To aid with input, when entering identifiers or Smalltalk message names, a suggestions menu offers users to autocomplete names. The popup can be freely positioned by the user. A screenshot can be seen in Figure 5.

We experimented with different means to enter characters, which, while we were optimizing for drag-and-drop interactions, tended to be the bottleneck as programmers wanted to enter new identifiers that the autocompletion did not have yet. Our first approach, dubbed "airwrite" allowed users to write simplified letters akin to the PalmOS Graffiti [6] alphabet in the air. As the single stroke letters were finished, a recognition system would match against their shape and input the corresponding character. By default, all letters were entered in lowercase, with the option to type uppercase letters by pushing the thumbstick forward.

As an alternative, we also added support for a typical virtual keyboard similar to the one used in mobile phones. A physical keyboard would not have allowed users to freely move around the room for editing as intended by our VR-native concept, which we thus avoided.

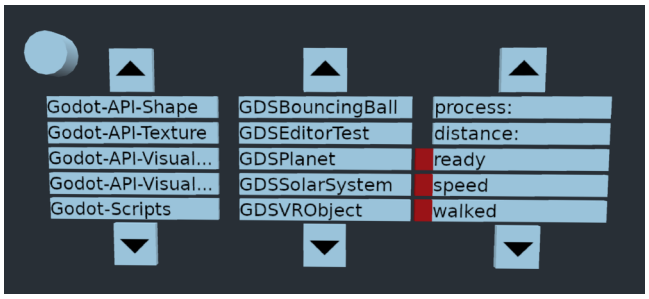

Figure 6. The code browser. The layout is similar to the Squeak/Smalltalk system browser, with code categories on the left, classes in the center, and instance methods on the right. A red button is shown next to subclasses of the VR morph class, allowing users to instantiate the class in the 3D world.

3.3 Code Browsing

For code browsing, we offer an experience similar to the Smalltalk class browser, as seen in Figure 6. Three panes offer the selection of categories, classes of the selected category, and methods of the selected class. By pointing the wand at the labels in the panes and pressing the trigger button, users select the respective element. A selected method opens in 3D space, next to the browser.

Users can reposition methods by dragging their root block. As this often required users to aim well, we added support for directly dragging the entire method, independent of the specific block pointed at, by placing a hand inside the method and gripping the controller. To preserve readability, a block reorients itself after being moved such that any contained text is perfectly horizontal.

3.4 Execution

For execution, we implemented a system comparable to Morphic [16]. Applications are composed of VR morphs. Opening an application corresponds to opening the VR morph designated as the application's toplevel by the application's authors. As of right now, the interactions offered are limited to the necessary minimum, only allowing users to spawn and replace a VR morph in the 3D world via buttons in the code browser that appear next to VR morph subclasses.

Changes to methods are immediately saved if there are no blocks with unfilled slots in the method. As we are building on top of the Squeak/Smalltalk live programming system, changes to methods also immediately become live as they are saved. When errors occur we display these on a virtual screen inside the VR environment. In addition, we cease executing methods of VR morphs that are called every frame once they throw an error. After the user changes such a faulty method, it may be executed again, potentially being removed from execution again right away, if the error persists. In rare cases, this can lead to unexpected behavior, if other parts

of the system depend on side effects of the faulty method, resulting in an inconsistency due to live programming which users have to resolve by restarting the application they are developing.

Coupled with the fact that users can position relevant methods in 3D space directly next to objects of their application, saving on changes allows users to observe the effect of their changes immediately.

4 Implementation

We developed the prototype with Squeak/Smalltalk [12] as a backend for executing user-defined behavior, allowing us to leverage its live-coding features. Additionally, we adapted an existing block-based editor for Squeak/Smalltalk by sending the current layout of the two-dimensional blocks to the VR frontend whenever it changed and derived a 3D layout. Any changes made to the code in VR are synchronized, triggering a corresponding event in the block-based editor in Squeak.

The VR frontend was implemented with the Godot game engine [15], with most of the frontend behavior and visualization being written in the engine-specific GDScript language. The communication between the Squeak/Smalltalk and Godot components runs over a Godot plugin that communicates with Squeak via its foreign function interface. The application was developed with and tested on the Valve Index VR headset.

5 Evaluating The Interaction Design

To evaluate and evolve our interaction design, we settled on a live code editing scenario that we asked test persons to perform in our VR programming environment. Our goal was to validate whether our current prototype allowed users to complete the given task and to identify shortcoming in intuitiveness or usability of our design. We thus asked participants to think-aloud while using the system. Afterwards, we performed a semi-structured interview with each participant on their experience in general and in particular concerning the four aspects we outlined earlier in subsection 2.2.

Method. We decided to split the experiment into two conditions, despite the small number of participants, to gather as much information as possible for future experiment designs. In the first, participants were given no prior instructions on the editor's usage such that we could test its intuitiveness. Due to a technical problem, in the first condition, participants were also not able to execute the program. In the second condition, participants were shown a two minute walkthrough demonstrating the main interactions. In both conditions, to ensure participants could still advance, the instructors set themselves a timer whenever the participant seemed to be lost. If one minute elapsed without progress in either condition, the instructors provided hints.

Editing Scenario. The participants were given a simple VR application which simulates a 3D solar system. It consisted of two Smalltalk classes, one modeling a simple planet as a 3D sphere, and one modeling a solar system, most notably taking care of instantiating the planets. The program amounted to 88 lines of code across two classes, and involved APIs from another three classes, including the relatively large interface of Godot's Vector3 class. We asked participants to perform the first task in the regular Squeak/Smalltalk desktop environment: letting the planets rotate around a common point. Next participants performed a comparable task but while in VR: changing the planet's velocities depending on the velocity of the right controller. Apart from these specifications, the participants were free to choose the exact behavior they implemented and were even encouraged to modify it until they achieved a satisfying result. Changes to the code persisted between the first and second task. Importantly, participants were not given instructions on how the API worked and thus had to browse code of relevant classes.

Participants. We recruited two participants per condition, all male graduate students. All participants were in similar courses as one of the authors, which may introduce bias [3]. Participants reported between 6 and 13 years of programming experience, with 0 to 5 of that in a professional capacity. Experience with the used API from Godot varied from no experience to intermediate, while experience with the host language Squeak/Smalltalk was reported by all participants. All participants had tried VR before but only in the context of games. All participants but one only tried VR a couple of times. The remaining participant owns and semi-regularly uses a VR headset. Only one participant had done development for VR applications before. One participant had insufficient knowledge of 3D transformations to solve the first task, such that the instructor had to provide guidance on API use during the task.

Random assignment resulted in the participants in the first condition having approximately equal experience. In the second condition, one participant was more highly experienced in comparison, and the other less experienced.

Intuitiveness. In the condition where no instructions were provided, participants were not able to complete the task in the VR editor. This was further compounded by technical difficulties where some parts of code entry malfunctioned. Feedback from participants was correspondingly negative but both stated that they imagine a more well-developed version of the editor to be beneficial for VR application development.

Feasibility. In the condition with instructions, both participants completed the task in the VR editor. Both felt less productive than when working with the desktop but appreciated the immediate feedback gained for VR application development. In particular, one participant stated that they

believe the VR editor to be useful during certain phases of development where fast feedback is especially helpful.

Threats to Validity. Given the early stage of the prototype, some technical issues occurred during all runs, where all but one were recoverable, but may still have influenced the participants' impression independent of the proposed interaction design. Further, the scope of the tasks was deliberately chosen as very small, to allow participants to get used to both the API and the novel development environment. As such, it is not clear yet how well our insights would apply in larger scenarios and a setting where participants had more time to get used to the interactions.

A study informed by our described preliminary evaluation should significantly increase the number of participants. Further, the selection of participants should ensure that the domain is sufficiently well known to not be a factor that influences the completion of the tasks. Next to qualitative insights similar to the ones we report here, tracking metrics such as task completion times and ratings of the users on aspect of usability would be valuable to remove possible bias through interpretation in the report of qualitative insights. Finally, the short time frame participants worked with the system may have negatively impacted their impression, as many of the interactions in our prototype have to be performed repeatedly and may thus feel less cumbersome once participants developed a muscle memory.

6 Discussion

Here, we briefly discuss the proposed interaction design based on the insights gained from the user study described in section 5.

6.1 Displaying and Editing the Syntax Tree

The block-based display was unfamiliar to the participants, which they cited as a reason for their difficulties. Participants also found that the readability of the code suffered, mainly due to the bright yellow visualization of possible cursor locations between blocks, which significantly increased the visual noise. Participants reported that they often knew exactly what they wanted to do, but had trouble physically performing the necessary actions.

All of the participants had difficulties positioning the text cursor, especially at the ends of text strings. Accurately placing a cursor required very precise motion due to the small size of the letters; even tiny motions and jitters of the users hand, compounded by the length of the wand, could shift the position. Possible solutions include increasing the size of the text and adding a second way to modify the cursor position after initial placement, similar to arrow keys on a keyboard.

In the same vein, aiming at individual blocks with the tip of the wand proved troublesome. This may be solved by increasing the thickness of the blocks past the current value of 3mm. However, one participant found the removal and insertion interactions very natural when they did work, although they did not feel the need to use them to complete the given task.

The participants generally made use of the suggestion menu. One participant did not notice that suggestions were provided and even expressed confusion over the purpose of the suggestion menu; whenever suggestions would be given, the menu would be outside of the participant's field of view or covered by blocks. This highlights the need for a mechanism that notifies the user of interesting events that occur out-of-sight.

6.2 Textual Input

We alternated between the virtual keyboard and our "airwrite" system, as described in subsection 3.2. For both, participants were successfully using them but mentioned, in particular for the virtual keyboard, that they miss the efficiency gained from typing with ten fingers.

For airwrite, a large poster placed in the environment depicted the mapping between gestures and their corresponding letters. Even though the participants initially needed time learn this mapping at least in part, they found it intuitive and clear to use. Notably, the number of failed input attempts appeared to drastically decrease with time, even in the small time span of our experiment. Similarly, a study on people using the Graffiti system, which our airwrite is based on, demonstrated that expert users tended to be faster with the stroke letters than touch typing [6].

For writing new code, where large amounts of text need to be entered, some participants would have preferred an input method that supports the speeds they are used to—either touch typing or speech-to-text.

6.3 Code Browsing

Finding code through the browsing interface worked well; all participants felt that their familiarity with Squeak's system browser helped. Apart from general usability improvements such as highlighting the correct category, the participants most often desired search functionality. One participant in particular missed the more advanced browsing tools that Squeak/Smalltalk provides, such as displaying implementers and senders of methods.

Multiple participants noted the fact the browser only allowed them to view instance methods of classes, but not class methods or class definitions. Upon inquiry, the instructors provided information on inheritance relationships in the API that helped users advance in the code browsing task. A future version of the programming environment needs to support such features.

One participant began arranging multiple methods in 3D space and mentioned that they believed to profit from the additional available space.

6.4 Execution

Live execution was received well and was described as intuitive and easy to use. Participants enjoyed seeing the effect of their changes immediately, which enabled them to quickly judge whether they were on the right track. One participant saw particular promise for applications that involve VR-specific interactions or generally utilize 3D space more fully.

6.5 General Observations

Due to the prototypical nature of the application, there are many areas in which usability can and needs to be improved. Amongst other things, participants commented on annoyances like methods appearing in the same place upon opening and overlapping each other, and a lack of highlights to indicate the currently selected category and class in the code browser. One participant found that their workflow was not fundamentally different than the one in a traditional desktop development environment—the main difference was just that most actions felt more awkward.

Even though all participants found that a desktop development environment was more mature and suited to their work, the VR environment was not without merits. All participants but one mentioned the large amount of space and additional dimension for arranging their workspace, with one stating that they felt that the available space encouraged them to be more explorative. Live feedback was also repeatedly mentioned as a significant advantage. The integrated nature of the development environment and developed program was also well received, reducing and even eliminating the need to switch between multiple programs.

All participants could imagine using a more mature VR-native live programming environment to develop VR applications in order to boost productivity. For general software development, their stance was generally more reserved, but not averse.

7 Future Work and Conclusion

As described in section 6, our study participants identified a number of areas that need improvement. Notably, we consider as most essential in a future iteration to investigate

- fully featured code browsing,
- higher tolerance for error when selecting blocks and positioning the cursor, and
- a more easily readable representation of code.

In addition, experimenting with different means to provide textual input, such as speech-to-text or one-handed, portable keyboards ("keyers"), or alternatively reducing the need for typing by locating existing identifiers more easily, could be investigated. Alternatively, different means to express behavior than through Smalltalk code, could be investigated, with the hopes of finding a notation that by its design requires less textual input.

In a future user study, we want to both verify that the proposed interactions are feasible for longer sessions and more complex tasks. To evaluate whether the idea of employing direct manipulation of program elements yields a more natural editing experience in VR, a future user study may want to compare our approach to the more common method of projecting a 2D text buffer into the 3D world.

Conclusion. In this paper, we presented a prototypical live programming environment for VR applications that leverages *VR-native* interactions through direct manipulation in a structured editor. In a user study with four participants, we found that participants were able to complete programming tasks in our editor if given prior instructions on the editor's use. Participants stated that they missed the productivity from the desktop setup they are used to, but liked the additional space gained from working in the virtual 3D world and the immediate feedback, as programming environment and application were both running in VR. As such, we hope to provide guidance for future designs, based on aspects that our test users found lacking and others that they validated as feasible for creating a programming environment in VR.

Acknowledgements

We thankfully acknowledge the financial support of the HPI Research School on Service-oriented Systems Engineering (www.hpi.de/en/research/research-schools) and the Hasso Plattner Design Thinking Research Program (www.hpi.de/en/dtrp).

References

[1] Tom Beckmann, Stefan Ramson, Patrick Rein, and Robert Hirschfeld. 2020. Visual Design for a Tree-Oriented Projectional Editor. In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming* (Porto, Portugal) *(<Programming> '20)*. Association for Computing Machinery, New York, NY, USA, 113–119. https://doi.org/10.1145/3397537.3397560

[2] Víctor Stefano Segura Castillo, Leonel Merino, Geoffrey Hecht, and Alexandre Bergel. 2021. VR-Based User Interactions to Exploit Infinite Space in Programming Activities. *2021 40th International Conference of the Chilean Computer Science Society (SCCC)* (2021), 1–5.

[3] Nicola Dell, Vidya Vaidyanathan, Indrani Medhi, Edward Cutrell, and William Thies. 2012. "Yours is Better!": Participant Response Bias in HCI. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Austin, Texas, USA) *(CHI '12)*. Association for Computing Machinery, New York, NY, USA, 1321–1330. https://doi.org/10.1145/2207676.2208589

[4] Anthony Elliott, Brian Peiris, and Chris Parnin. 2015. Virtual Reality in Software Engineering: Affordances, Applications, and Challenges. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 547–550. https://doi.org/10.1109/ICSE.2015.191

[5] Shaghayegh Esmaeili, Brett Benda, and Eric D. Ragan. 2020. Detection of Scaled Hand Interactions in Virtual Reality: The Effects of Motion Direction and Task Complexity. In *2020 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*. 453–462. https://doi.org/10.1109/VR46266.2020.00066

[6] Michael D Fleetwood, Michael D Byrne, Peter Centgraf, Karin Dudziak, Brian Lin, and Dmitryi Mogilev. 2002. An evaluation of text-entry

in Palm OS–Graffiti and the virtual keyboard. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, Vol. 46. SAGE Publications Sage CA: Los Angeles, CA, 617–621.

[7] Epic Games. 2014. Unreal Blueprints. https://docs.unrealengine.com/5.0/en-US/blueprints-visual-scripting-in-unreal-engine/. https://docs.unrealengine.com/5.0/en-US/blueprints-visual-scripting-in-unreal-engine/ [Online, accessed 31 August 2022].

[8] Google. 2020. Blockly. https://developers.google.com/blockly. https://developers.google.com/blockly [Online, accessed 29 August 2022].

[9] Akihiro Hori, Masumi Kawakami, and Makoto Ichii. 2019. CodeHouse: VR Code Visualization Tool. In *2019 Working Conference on Software Visualization (VISSOFT)*. 83–87. https://doi.org/10.1109/VISSOFT.2019.00018

[10] Luke Iannini. 2016. Rumpus. https://store.steampowered.com/app/458200/Rumpus/. https://store.steampowered.com/app/458200/Rumpus/ [Online, accessed 29 August 2022].

[11] Immersed Inc. 2020. Immsersed. https://immersed.com/. https://immersed.com/ [Online, accessed 24 August 2022].

[12] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Atlanta, Georgia, USA) *(OOPSLA '97)*. Association for Computing Machinery, New York, NY, USA, 318–326. https://doi.org/10.1145/263698.263754

[13] jmiskovic. 2020. inDECK. https://github.com/jmiskovic/indeck. https://github.com/jmiskovic/indeck [Online, accessed 29 August 2022].

[14] Pooya Khaloo, Mehran Maghoumi, Eugene Taranta, David Bettner, and Joseph Laviola. 2017. Code Park: A New 3D Code Visualization Tool. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*. 43–53. https://doi.org/10.1109/VISSOFT.2017.10

[15] Juan Linietsky, Ariel Manzur, and Godot Engine contributors. 2014. Godot. https://godotengine.org/. https://godotengine.org/ [Online, accessed 30 August 2022].

[16] John H. Maloney and Randall B. Smith. 1995. Directness and Liveness in the Morphic User Interface Construction Environment. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology* (Pittsburgh, Pennsylvania, USA) *(UIST '95)*. Association for Computing Machinery, New York, NY, USA, 21–28. https://doi.org/10.1145/215585.215636

[17] Mauricio Verano Merino, Jurgen J. Vinju, and Mark van den Brand. 2021. DRAFT-What you always wanted to know but could not find about block-based environments. *CoRR* abs/2110.03073 (2021). arXiv:2110.03073 https://arxiv.org/abs/2110.03073

[18] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2019. Exploratory and Live, Programming and Coding - A Literature Study Comparing Perspectives on Liveness. *The Art, Science, and Engineering of Programming* 3, 1 (2019), 1. https://doi.org/10.22152/programming-journal.org/2019/3/1

[19] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and et al. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (Nov. 2009), 60–67. https://doi.org/10.1145/1592761.1592779

[20] Markus Schütz and Michael Wimmer. 2019. Live Coding of a VR Render Engine in VR. In *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*. 1150–1151. https://doi.org/10.1109/VR.2019.8797760

[21] SimulaVR. 2018. Simula. https://github.com/SimulaVR/Simula. https://github.com/SimulaVR/Simula [Online, accessed 29 August 2022].

[22] Robert Twomey, Tommy Sharkey, Timothy Wood, Amy Eguchi, Monica Sweet, and Ying Choon Wu. 2022. An Immersive Environment for Embodied Code. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI EA '22)*. Association for Computing Machinery, New York, NY, USA, Article 197, 4 pages. https://doi.org/10.1145/3491101.3519896

[23] David Ungar, Henry Lieberman, and Christopher Fry. 1997. Debugging and the Experience of Immediacy. *Commun. ACM* 40 (04 1997), 38–43. https://doi.org/10.1145/248448.248457

[24] Markus Voelter, Tamás Szabó, Sascha Lisson, Bernd Kolb, Sebastian Erdweg, and Thorsten Berger. 2016. Efficient development of consistent projectional editors using grammar cells. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, Tijs van der Storm, Emilie Balland, and Dániel Varró (Eds.). ACM, 28–40. http://dl.acm.org/citation.cfm?id=2997365

Received 2022-09-01; accepted 2022-10-02