H.I.D.E.: A Virtual Reality Debugging Environment

Nicolas Slack
Department of Informatics
University of Sussex
Nicolas.slack@gmail.com

Kate Howland
Department of Informatics
University of Sussex
k.l.howland@sussex.ac.uk

Abstract
This paper presents a prototype virtual reality debugging environment, aimed at novices, that provides a 3D visualisation of code and supports gestural navigation through the visualisation. This prototype is implemented for the HTC Vive, and has the capability to expand into a variety of 3D visualisations, programming languages, and VR platforms. We describe how our design builds on previous research on debugging to provide support for observing, exploring and hypothesizing activities by focusing on flow-of-control and data visualisations. We present preliminary results from pilot user testing and highlight key areas for future development.

1. Introduction
Debugging is an inevitable and challenging aspect of programming, and there is evidence that visualisation tools can provide useful support for users in the task, particularly for novice programmers. With the recent growth in virtual and augmented reality environments, there is potential for development of 3D representations that allow programmers to 'walk through' their code as part of the debugging process. This work explores the potential for alternative methods of displaying data and control flow in a virtual environment, with a view to identifying fruitful future directions for virtual reality debugging support. We present a prototype system, H.I.D.E (Humanised Interactive Development Environment), and some preliminary user testing findings.

The following section gives a brief background on how this work builds on existing research on debugging support. Following this, an overview of H.I.D.E is given, highlighting the key aims and functionality of the system. Finally, we explain our initial findings from pilot testing and their relevance before concluding, and pointing to key areas for future work.

2. Background
Debugging is an arduous task that all programmers at any level must undertake at some point, and a significant quantity of time allocated to it on any project. The motivation behind our system was to explore how a virtual reality environment could provide support to increase debugging efficiency, by building on existing understandings of common behaviours exhibited by programmers. Empirical studies have produced several relevant findings on debugging behaviour. As highlighted by Ko and Myers [1], there is a broad consensus that debugging is an exploratory task that can be broken down into six distinct and interleaving activities: hypothesizing what went wrong; observing runtime data; restructuring data; exploring the restructured data; diagnosing code; repairing code.

Ko and Myers [2] found that around half of the errors novice programmers made with the novice programming environment Alice were due to false assumptions in hypotheses made while debugging existing errors. In later work they reported on the questions asked by novice programmers whilst debugging, noting that "85% of questions were about a single object. The remaining concerned multiple objects' interactions" [1, p. 153]. This set the foundation for our thinking. If programmers are producing incorrect assumptions, and are primarily focussing on single objects or multi-object interactions, we asked how data could be displayed in a fashion to better facilitate their understanding.

Romero and colleagues noted in [3] that "experienced programmers, when comprehending code, are able to develop a mental representation that comprises different perspectives … as well as rich mappings between them". However, this mental representation places a cognitive load upon programmers, increasing the quantity of data they need to process. Ideally, a system should form these mappings in a fashion that can be easily read without inducing additional cognitive load. Such a system would be

especially useful for novices, as they do not have the experience to form these mappings. Romero et al. highlight the 'double challenge' faced by novices:

> *"…. As well as trying to learn abstract concepts about programming, they have to master the decoding, representation coordination and step-and-trace skills required to use debugging environments."* [3, p.993].

Traditionally, debugging environments are deployed onto standard hardware. These environments use the standard interfaces; keyboards, mice, and monitors. However, there are many rapidly evolving technologies that present alternative methods of interfacing with computers. Virtual reality and Augmented reality devices (Such as the HTC Vive and Microsoft Hololens) are but a few examples. These devices present interaction possibilities not present in traditional input devices, such as haptic control, true 3D environments, and depth perception. Exploring how these new possibilities could be used to support novice debugging is the foundation of our system.

3. H.I.D.E Overview

H.I.D.E (Humanised Interactive Development Environment) is a system developed in Unity, deployed to the HTC Vive that visualises LUA scripts in a 3D virtual environment. H.I.D.E focusses on providing support for the observing, exploring and hypothesizing stages of debugging. These activities are the ones most likely to benefit from visualizations, and there is potential to reduce the cognitive load of programmers by making mappings visible.

```
1|    x = 7;
2|    if (example ==
true) then
3|          x = 12;
4|    end
```

Figure 1 – An example LUA script

H.I.D.E represents each lexical token within a program's lexical token tree as a 3D object within the space. When looking directly at, or highlighting a token with an input device, the system will show the data value of that token at that point in the program execution. For example, when looking directly at the token representing `'x'` on the first line, the value shown in the display will be `'7'`. However, when observing `'x'` on the third line, the value shown in the display will be `'12'`. Similarly, when observing the `'if'` statement, if the condition evaluates to true, the system will display `'Is Entered'` to alert the user that this statement's body will be executed.

Figure 2 shows a view from an early build of the HIDE system, displaying an equality test. The user in this instance is looking directly at the equality symbol in the middle, while highlighting the two data values on either side. As can be seen, this equality is testing if `'n == 0'`. The left-hand sphere represents 'n' in this instance, and its current value at this point in the program execution is '1'. As such, the display shows false.

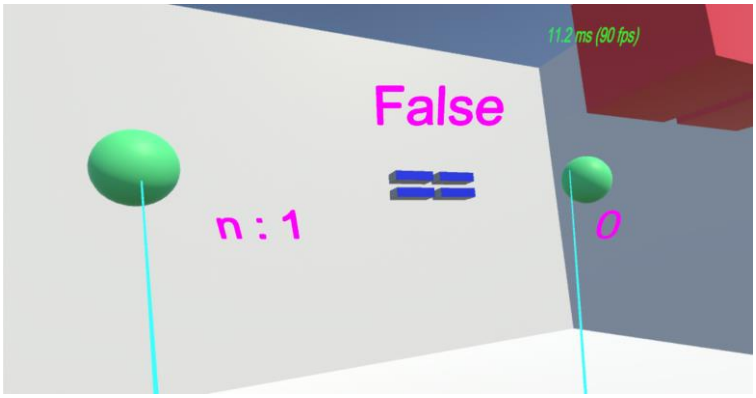

Figure 2– An image captured from an early build of H.I.D.E

Below is an example use case of H.I.D.E:

> *"A programmer is struggling to discover why her factorial function is returning incorrect values. However, while looking at the code alone she cannot discover the error. She loads the program into HIDE, and follows the manipulation of the input variable through in HIDE, using it's highlighting and context-sensitive readouts. She discovers that 5! is returning 121 instead of 120. After following the execution through visually, she quickly spots that the base case is returning 2 instead of 1"*

Though the example is simplified to an extreme, it demonstrates the potential benefit of the tool. HIDE is designed to help programmers spot logical errors or human errors, that are difficult to see when

looking at traditional readouts. Program code can be navigated using the touch pads on the Vive controller, and code objects can be highlighted by pointing a controller at them and holding the trigger button. The system casts rays from the headset and controllers to detect which code object is currently being held in focus, and displays the appropriate readout.

4. Evaluation

We conducted a pilot user test, aimed at exploring how users interacted with the system, and their ability to understand the data displayed. The testing session was designed to assess how quickly participants comprehended what a small script was doing in two different environments – notepad++ and our system.

4.1 Method

Two undergraduate Computer Science students took part in testing. They each had 2-3 years programming experience, but neither had programmed in the Lua language before. Participants were given a brief orientation period, and then asked to debug two different scripts in the two different environments, with the ordering reversed. The time allowed for attempting to debug the short script in each system was 5 minutes. We asked participants to narrate their thought process throughout the experiment, and collected their informal observations about the system by audio recording the session. At the end of the 5 participants were asked to explain what they believed the program did, if they had not tried during the 5-minute period.

4.2 Results

Table 1 shows the time it took each participant to debug the scripts in each environment, whilst Table 2 shows participant feedback comments (transcribed from audio files).

Table 1 – Preliminary test results

Participant	file1	Notepad++ time	file 2	H.I.D.E time
1	test1.lua	4m 33s	test2.lua	Unsuccessful
2	test2.lua	Unsuccessful	test1.lua	End of test

Table 2 – Participant comments

Participant 1	Participant 2
Within a short amount of time, it's easy to understand what is happening	Some parts made sense, but there is a lot going on
Extra-dimensional indentation is useful	Layout is too spread out
Large code bases may cause it to fall apart in terms of readability	Having to look at objects instead of data being explicitly displayed is frustrating
Could be useful in a teaching environment, or a small tech company	It could be difficult to understand
It is another way of sharing or debugging code	Unsure of where to use the system, but could be useful in smaller companies
Is more intuitive than some ways of displaying code	More of a secondary approach to traditional debugging
Difficult to judge what detail is needed	System is an alternative method of viewing code, from a new perspective
Main issues currently are colour and layout oriented. The layout is well presented, but rough around the edges	Vive is a good hardware platform to deploy the system to
It's good fun, and would be a good teaching aide	Colour and transparency was not clear
Good for exploring a code base	

4.3 Discussion

Neither participant successfully found the bug using the HIDE environment. Future versions of this study should allow more time for familiarisation with the system and the debugging task to allow further investigation of whether the system can support bug identification. However, participant comments gave some useful initial feedback. Both subjects described the system as intuitive, especially the dimensionality, which bodes well for developing the idea further. The system naturally leans towards collaborative efforts, as the visualisation could be seen by multiple parties simultaneously which would aide in mutual understanding of information. Colour, however, was highlighted as counter-intuitive, as such it will need further consideration.

The broader implications of this system for virtual reality are apparent. Users found having to look at an object to bring up information frustrating, and that the data was too spread out. This seems to stem from the low resolution of the headset, paired with a narrow field of view. The data can seem overwhelming with such a narrow perspective. In future systems, data must be very carefully filtered and abstracted to prevent cognitive overloading.

5. Conclusion

Our system creates a 3D environment within which users can view program code, to aide in their understanding of program structure. The visualisation aides in the formations of hypothesis regarding program errors, and assists users in comprehending the flow of data and control through a program. The current system presented requires additional refinement at the implementation level.

The system utilises a virtual reality headset to provide a novel approach to viewing program code, and presents opportunities for teaching inexperienced programmers the foundations of programming in a more interactive and engaging way.

The current system is limited to LUA, and does not fully support cross-file dependencies. However, the system has the potential to grow into a multitude of areas. Areas that require data input include; Running a script of function with a set of inputs, taken from natural language input, Modification of program code inside the environment, and integrated automated testing (such as Monte-Carlo or Unit testing). Areas that do not require input that could be expanded upon are; Multiple users in the same environment, Cross-platform compatibility (Such as the Occulus rift and Hololens) and Multi-language support for both programming and natural languages.

The system has identified a promising field for further study regarding alternative methods of displaying data. The inherent ease of use when dealing with multi-sensory environments has great potential for increasing efficiency, as well as communicating data between entities faster.

6. References

1. Ko, A.J. and Myers, B.A., 2004. Designing the Whyline: a debugging interface for asking questions about program behavior. In Proceedings of the SIGCHI conference on Human factors in computing systems (pp. 151-158). ACM.
2. Ko, A.J. and Myers, B.A. 2003. Development and evaluation of a model of programming errors, IEEE Symposia on Human-Centric Computing Languages and Environments, 2003, Auckland, New Zealand, (pp. 7-14).
3. Romero, P., du Boulay, B., Cox, R., Lutz, R. and Bryant, S., 2007. Debugging strategies and tactics in a multi-representation software environment. *International Journal of Human-Computer Studies*, *65*(12), pp.992-1009.
4. "Moonsharp". Moonsharp.org. N.p., 2017. Web. 15 Dec. 2016.