H.I.D.E

Humanised Integrated Development Environment

Candidate Name: *Nicolas Slack*

Degree Course: *Game and Multimedia Environments (G.A.M.E)*

Department: *Informatics*

Candidate Number: *132236*

Supervisor: *Dr. Kate Howland*

Submitted 2017

Statement of originality

This report is submitted as part requirement for the degree of Game and Multimedia Environments (*G.A.M.E*) at the University of Sussex. It is the product of my own labour except where indicated in the text. The report may **NOT** be freely copied or distributed without my explicit, written permission.

Nicolas Slack – 13/04/2017

Acknowledgements

DR KATE HOWLAND

For answering my questions and providing invaluable guidance.

DR PATRICK HOLROYD & DR PHIL WATTEN

For advising me on user interface design procedure, even when the headset made them feel sick.

DR SRIRAM SUBRAMANIAN

For generously purchasing a development machine for myself and another HTC Vive student, and allowing my usage of his HTC Vive.

Summary

This paper presents a prototype virtual reality debugging environment that provides a 3D visualisation of code and supports gestural navigation through the visualisation.

This prototype is implemented for the HTC Vive, but has the potential to expand to any device with sufficient processing power such as a tablet, augmented reality device, or embedded system. The system also has support for changing the interpreted code base with minimal adjustment, as well as adapting the display of data.

The researcher describes how his design builds on previous research on debugging to provide support for tasks such as observing, exploring and hypothesizing activities. The system achieves this by focusing on flow-of-control and data visualisations.

The researcher presents preliminary results from a small pilot user test, highlighting key areas for future development. He also expands upon where system development could continue, as well as potential deployment areas for a fully implemented system.

Table of Contents

1 - Introduction

This project is aimed primarily at the development of a proof-of-concept debugging solution which utilises gesture controls in a virtual reality environment.

1.1 - Problem Area

One large time sink when developing any piece of software, is the quantity of time set aside to test and refine the program. On a normal software project, a large amount of the time expended is spent on testing, configuration management, and debugging.

Regarding debugging itself, user research studies suggest it is an exploratory task that can be broken down into six distinct and interleaving activities (as summarised by Ko and Myers in [1]):

1. *Hypothesizing* what went wrong
2. *Observing* runtime data
3. *Restructuring* data
4. *Exploring* the restructured data
5. *Diagnosing* code
6. *Repairing* code

Each of these tasks is critical to the success of any debugging endeavour. The software solution created focused on *Observing*, *Exploring* and *Hypothesizing*. The chosen areas are those that can benefit most from a visualisation of program code, and are crucial steps towards fixing a given bug. There is potential for debugging activities to introduce further errors, particularly whilst hypothesising. In a study of the programming environment Alice, Ko and Myers[2] found that *"50% of all errors were due to programmers' false assumptions in the hypotheses they formed while debugging existing errors."* To provide better support for observing, exploring and hypothesizing, the software focuses on flow-of-control and data visualisations.

1.2 - Objectives

The project's objectives were broken down in to three sub-categories, based on the perceived time investment required to complete them. Core objectives were those identified as essential to complete and extension objectives were those potentially within scope. Future objectives were identified to provide a sense of how the project could grow with additional resources, but were not attempted during the development cycle.

1.2.1 - Core Objectives

1. Visualisation of program code in a 3D virtual space
2. Gestural interaction with code visualisations
3. Ability to "zoom" between scope levels in the visualisations
4. Functional VR file browser built into the system
5. Multi-file support

1.2.2 - Extension Objectives

6. Construction of an execution timeline, allowing any point in the program to be simulated at any time
7. Ability to run files with user specified inputs
8. Context-sensitive debugging data, relative to position in the program
9. Multi-thread support
10. Basic Monte Carlo automated testing
11. Basic Type Conformance automated testing
12. Basic programming interface (to allow users to directly create new program code, instead of manipulating existing code)

1.2.3 - Future Objectives

13. JIT compiler, to re-compile edited segments of code
14. Execution timeline interpreter, to prevent committing the entire program to disk again when a segment is re-compiled
15. Program state streaming from hard disk, to allow saving of example program states

1.3 – Report Structure

In the following section, the professional considerations for the project are discussed, regarding the BCS Code of Conduct and the ethical compliance procedure. Section 3 gives an overview of the development process, including the requirements analysis, design and implementation. Section 4 summarises the salient points of the testing and evaluation performed in section 3, and contains the conclusive points obtained from the project.

2 - Professional Considerations

2.1 - Ethical Considerations

As the researcher conducted a questionnaire to gather user information and requirements, his compliance with the University Ethical standards was required.

(visible at: https://www.sussex.ac.uk/webteam/gateway/file.php?name=ethical-compliance-form-for-ug-and-pgt-projects.pdf&site=356)

A signed copy of the ethical compliance form is scanned and provided in the Appendix[6.2] for viewing.

The software was also tested repeatedly on programmers. As the HTC Vive is not classified as standard hardware under University policy, it does not comply with the University Ethical standards, so a full ethical review was conducted.

The full ethical review can be viewed in the Appendix[6.3]

To summarise the potential risks to using the HTC Vive; nausea and disorientation are the most prominent risks. The researcher ensured that no test subject was using the hardware for more than thirty minutes, and should any of the subjects have wished to stop the test early due to medical concerns or feeling unwell, the test would have been stopped immediately.

https://www.sussex.ac.uk/webteam/gateway/file.php?name=ethical-review-guidance-for-ug-and-pgt-projects.pdf&site=356

2.2 - BCS Code of Conduct

As is required of him as a member of the BCS, the researcher ensured that the project has conformed to the BCS code of conduct. Regarding the code, the following points are especially relevant to this project:

1a) have due regard for public health, privacy, security and wellbeing of others and the environment.

As the project is using new hardware that has the potential to cause slight discomfort to users, the researcher paid special attention to the test subjects' health and wellbeing. To ensure that no strain was caused, the hardware was not used for more than the designated safe period.

1b) have due regard for the legitimate rights of Third Parties

As the researcher was using 3rd party hardware, he took responsibility for using the hardware correctly, not damaging equipment, or misusing any part of said equipment without the 3rd parties' written consent

2c) develop your professional knowledge, skills and competence on a continuing basis, maintaining awareness of technological developments, procedures, and standards that are relevant to your field.

This project focused on bleeding edge technology and practices, and as such the researcher remained updated on all relevant materials to the project. Without doing so, he would have quickly fallen behind the curve.

2e) respect and value alternative viewpoints and, seek, accept and offer honest criticisms of work.

Throughout the development of this project, the researcher constantly sought opinions and advice from potential users/his superiors. As he aimed to create a powerful debugging tool for programmers, he felt it would have been inappropriate not to ask them constantly if the project is useful or fulfilling its role.

2g) reject and will not make any offer of bribery or unethical inducement.

The researcher did not bribe any potential participants to test his software or answer his questionnaires. This would have been counter-productive, as it would have biased the results of said tests, rendering them useless.

*3d) **NOT** disclose or authorise to be disclosed, or use for personal gain or to benefit a third party, confidential information except with the permission of your Relevant Authority, or as required by Legislation*

As he collected a small quantity of personal information through a questionnaire, the researcher ensured that none of that information was given to him without consent. He also ensured that the information was not used inappropriately or shared without express permission.

4f) encourage and support fellow members in their professional development

This project was designed to assist fellow programmers in their jobs, and increase their productivity. As such, the researcher hopes that the project will assist all those involved in furthering their professional development.

3 - The Process

3.1 - Requirements Analysis

3.1.1 - Questionnaire Results

To build a better understanding of programmers' debugging experience a questionnaire was designed and circulated amongst Informatics students at the University of Sussex. The questions covered programming and VR experience, and open questions examined debugging experiences, including challenging areas, as well as ideas and requests for debugging support tools. There were six responses from students. The full questions and responses can be found in Appendix[6.1]. In summary:

The researcher interviewed 6 programmers about their debugging and programming experience. In summary:

- The average quantity of programming experience was 4 years
- Common languages across all interviewees were; Java, Python and C#
- Average project length was 1000-5000 lines of code
- Average debugging time was 1-3 months

Regarding the long answer questions, half of the programmers described debugging as "tedious", only one programmer stated that it was "not too bad", though with the caveat that the code in question was well-structured. The common consensus was that the biggest challenge when debugging was finding the error/bug in the code. A majority also stated that the most time-consuming part of debugging was finding the error in the code.

Responses explaining these programmers would find useful as additional functionality varied:

- *"Well-commented, modular code that is easy to follow."*
- *"To be able to debug while running the project."*
- *"For multi-threaded programs, some graph-like display which shows which threads might be paused or running. The ability to change what the variable might have mid step through might be useful."*
- *"An interactive diagram of actions and state before and after my current point in the diagram."*
- *"Knowing what has changed in your code around the time the bug started."*
- *"These are the tools I am used to using and haven't heard of anything that sounds more helpful than these."*

The most interesting thing to note about these responses is that all the extra functionality can be emulated/created using step through, breakpoints and write line statements (the foundation of modern debugging). However, it is quite a long process to set these up using traditional tools. From this, the researcher extrapolated that these users wish for more streamlined tools to create more sophisticated debugging environments.

For the full results of the questionnaire, please see the Appendix[6.1].

From these results, the researcher drafted two personae for target users:

3.1.2 - Target User Persona - Student

Name	Charlie Hobb
Age	20
Work Location	Computing Laboratory
Technical Skill	Moderate (4-5 years' experience in more than 3 languages)

Background
A programmer who has spent at least three years in academic programming, Charlie looks to debug his programs quickly. Like most amateurs, he doesn't spend nearly as long on testing as he should. As such, his programs are prone to logical errors and poor structuring (due to not spending enough time properly sorting and refactoring his code base).

Goals
Charlie wants to get a program that works first time. He doesn't want to spend lots of time debugging, as he just wants to finish the work as quickly as possible so that he can go home. For him, the most important factors are a clean interface and a fast execution time.

Existing methods
Currently, Charlie uses his IDE's standard debugger, which has: A watch list, Breakpoint functionality, and step-through. He also writes write-line statements to print out to console when required, and occasionally uses the built-in IDE logger.

3.1.3 - Target User Persona – Industrial Programmer

Note: Although no industrial programmers completed this questionnaire, the researcher is well acquainted with several industrial programmers, and based this persona on conversations with them

Name	Steven Smith
Age	32
Work Location	Office
Technical Skill	High (At least 12 years' experience in more than 6 languages)

Background
Steven works in a normal 9-5 job, occasionally working weekends when a deadline is approaching. He works in the test department, and spends his days debugging and testing his team's code base. Because of this, he is intimate with the code base, but sometimes has trouble understanding the code, as he didn't write it.

Goals
Steven wants to speed up his workflow, and aims to achieve this by improving his understanding of the code base he's working with. Steven also wants to be able to find errors in the code base faster, so that he can report them to the code team and save everybody time.

Existing methods
Currently, Steven has access to a full set of industrial grade debugging tools, such as; run-time thread analysis, memory monitoring, bespoke environment loggers, and all normally available tools (breakpoints, etc.)

3.1.4 – Existing Research

A brief review of existing research revealed a myriad of useful insights that should be taken in to consideration when designing the system.

Romero et al. [3] highlights how information tends to be broken up in a debugging interface, along with the varying shortfalls of each "classification" of debugger.

Information within a debugger has varying properties, such as:

- *Information Modality*: The form of expression used to present information
- *Programming Perspective*: The view(s) that can be used to look at a code domain

These properties will be imperative when designing our own debugging interface, as we must provide useful perspectives to the user, without sacrificing the strength of expression by utilizing an incorrect modality. As is stated within the paper;

> *"There has to be a balance between a representation highlighting as few information types as possible, to keep it simple for easy information extraction, and presenting the programmer with only a few additional representations, so that problems of representation co-ordination do not over complicate the task." [3]*

Romero et al. [3] also explain the concept of *Cognitive Dimensions* framework. This framework is used to analyse the notational design of information interfaces, such as debuggers. These dimensions contain three component parts; *Dimension*, *Activity Types* and *Environment*.

Regarding the *Dimension*, there are five types;

1. Closeness of mapping – how far internal and external representations are apart
2. Role-expressiveness – ease of assembly/dis-assembly of structures in the notation
3. Hard mental operations – how much mental load the information poses
4. Diffuseness/terseness – quantity and variation of data representation
5. Hidden Dependencies - visibility of dependencies between components

These dimensions can be used to measure the effectiveness of a debugger. A debugger should have a close mapping, and a high role expressiveness, place very few hard-mental operations on the user, be terse, and reveal as many dependencies as possible. However, these traits conflict with each other, so a balance must be struck between them.

Regarding *Activity Types*, there are four types;

1. Incrementation – adding a new element to the system
2. Transcription – transferring information from another notational system to the current system
3. Modification – changing something within the current notational system
4. Exploratory Design – an activity where the outcome is discovered by performing the task

In the ideal debugger, Incrementation, Transcription and Modification would be fast and simple. Exploratory design activities would be reduced as much as possible, as all actions would have known consequences when debugging.

Regarding *Environment*, this merely refers to the current notational system and software/tools used for it. A change in the environment will likely change the notational system and tools, as such it is best to contain all activities within a single environment if possible.

3.1.5 - Requirements

After considering the questionnaire results and the review of existing research, the system requirements were identified as follows:

Functional Requirements	**Non-Functional Requirements**
Must provide a visual representation of program code in a clear, well laid out manner	Must NOT place a heavy mental load on the user
Must allow users to change between scope levels	Must NOT hide more dependencies than is deemed necessary
Must allow users to perform minor code changes in-system	Must NOT cause users to become nauseous or disorientated
Must interpret C# code correctly into a visualisation	Must NOT run at lower than 90 fps
Must accept code input from Text files or .cs files	Must utilise at least 100 degrees FoV
Must process input code in an adequate time	Must increase the user's debugging efficiency/ reduce debugging time
Must provide basic unit testing (such as type testing/ montecarlo testing)	Must accept Gestural inputs and have a 95% recognition rate

3.2 – Design

3.2.1 - Stage One – Drafting

3.2.1.1 - Iteration I

The above initial sketch was drawn up early in the project cycle, and is the researcher's first attempt at drafting a visualisation that could support debugging.

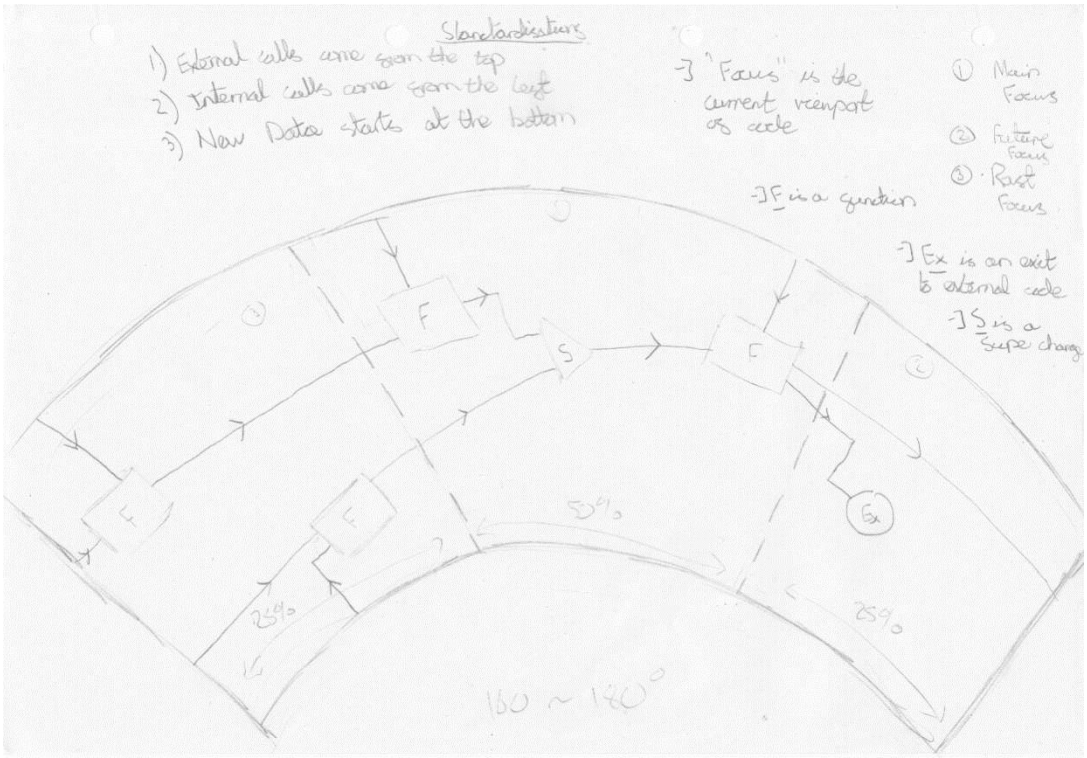


Figure 3.2.1.1.1 – Initial paper design of the interface

This design was merely placing thoughts on paper, though the key points were that the viewport was split into three distinct sections – Past, Present and Future. Each section of the viewport was a specific *Focus*, and served a purpose:

- *Past* was used to view the section of code most recently investigated, and to "scroll" backwards through code
- *Present* was used to see the current code being executed/investigated
- *Future* was used to see upcoming code manipulation, data inputs, etc. Also, used to "scroll" forward through code

As well as this, the topic of "data lines" first emerged in this design. *Data Lines* are the core concept of the software, and are a visual representation of data flow. These lines are to be manipulated and observed, to clarify how data is being used in the program.

3.2.1.2 - Iteration II

This iteration incorporates insights from the previously discussed 'WhyLine' design and evaluation [1], and adds some additional functionality to the first design iteration.

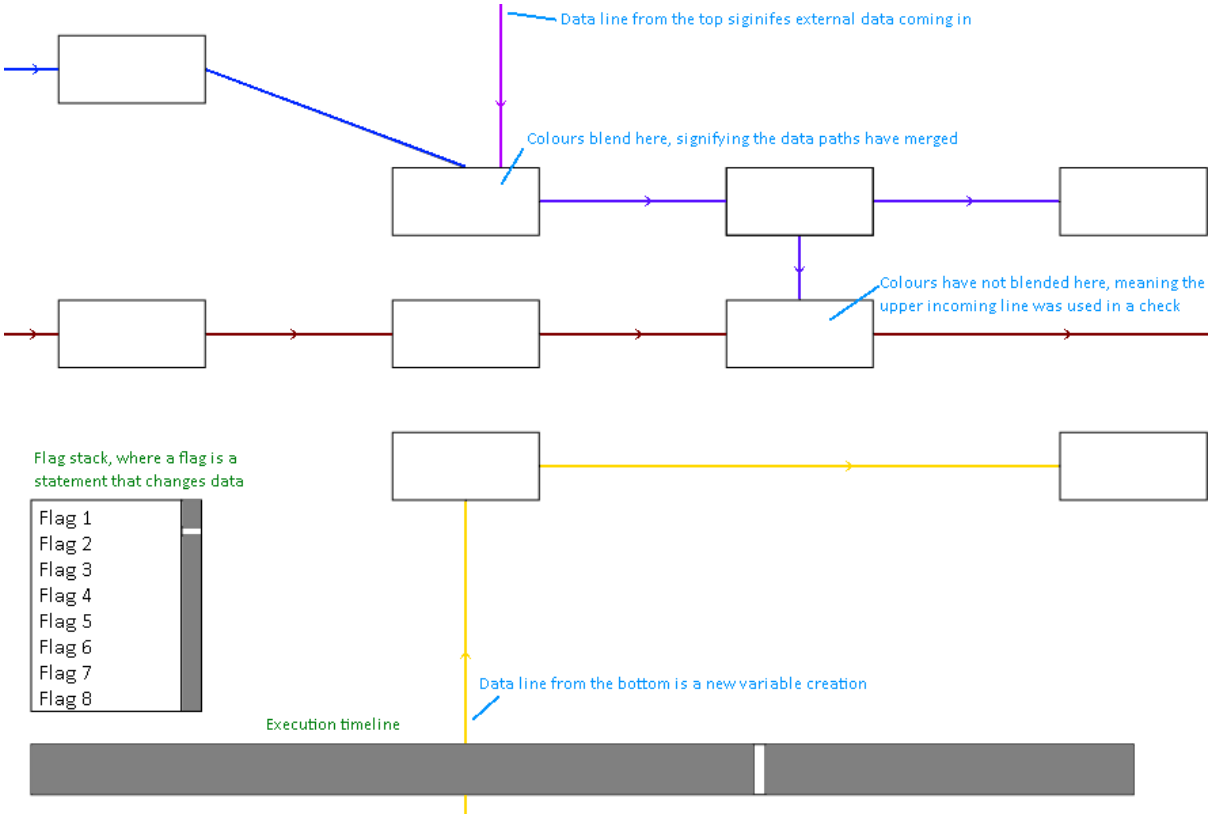


Figure 3.2.1.2.1 – the second UI design iteration, annotated

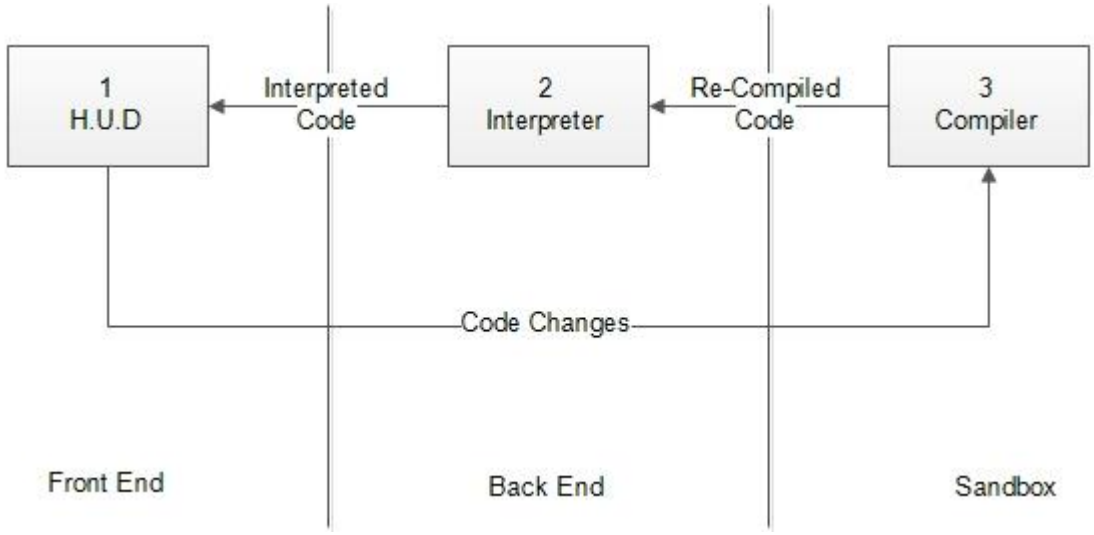
This iteration incorporates two additional functionalities that the *WhyLine* paper mentions;

- *The Execution Timeline*: Allows users to "scrub" the execution of the program, to repeatedly go over a section of code without having to re-execute the program up to a certain breakpoint
- *Flag Stack*: Although a slight twist, the flag stack shows all "important" statements the code base has executed, and allows the user to jump the execution timeline to that position by gesturing to the list

3.2.2 - Stage Two – Modelling

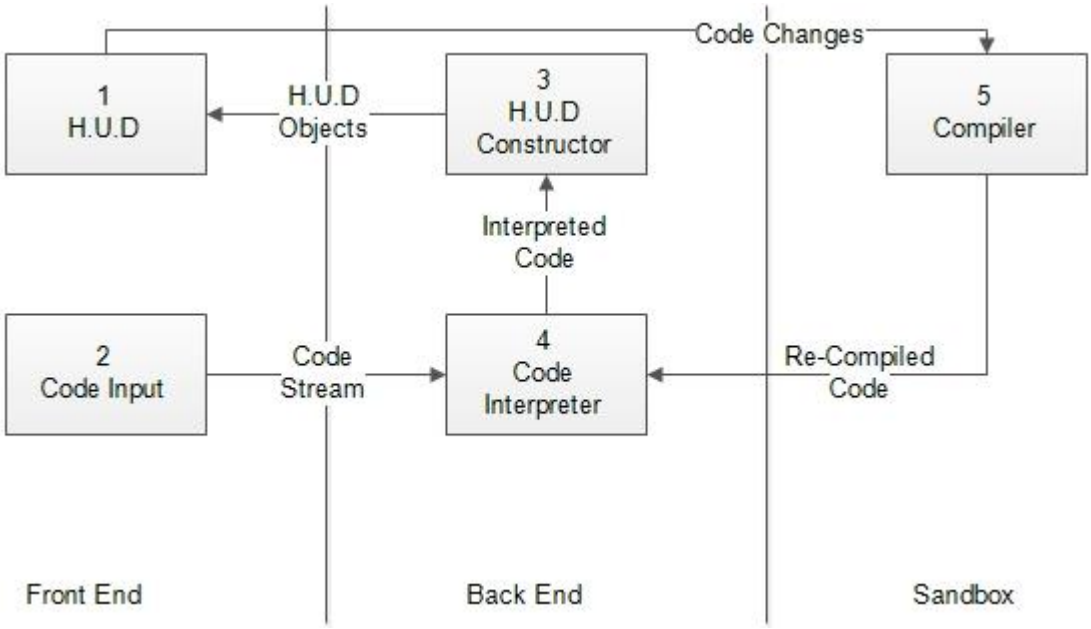
After the concept designed for the user interface, the researcher started to model the internal workings of the system. The first three level drafts are shown below:

3.2.2.1 - Tier 1 Design – Software Overview



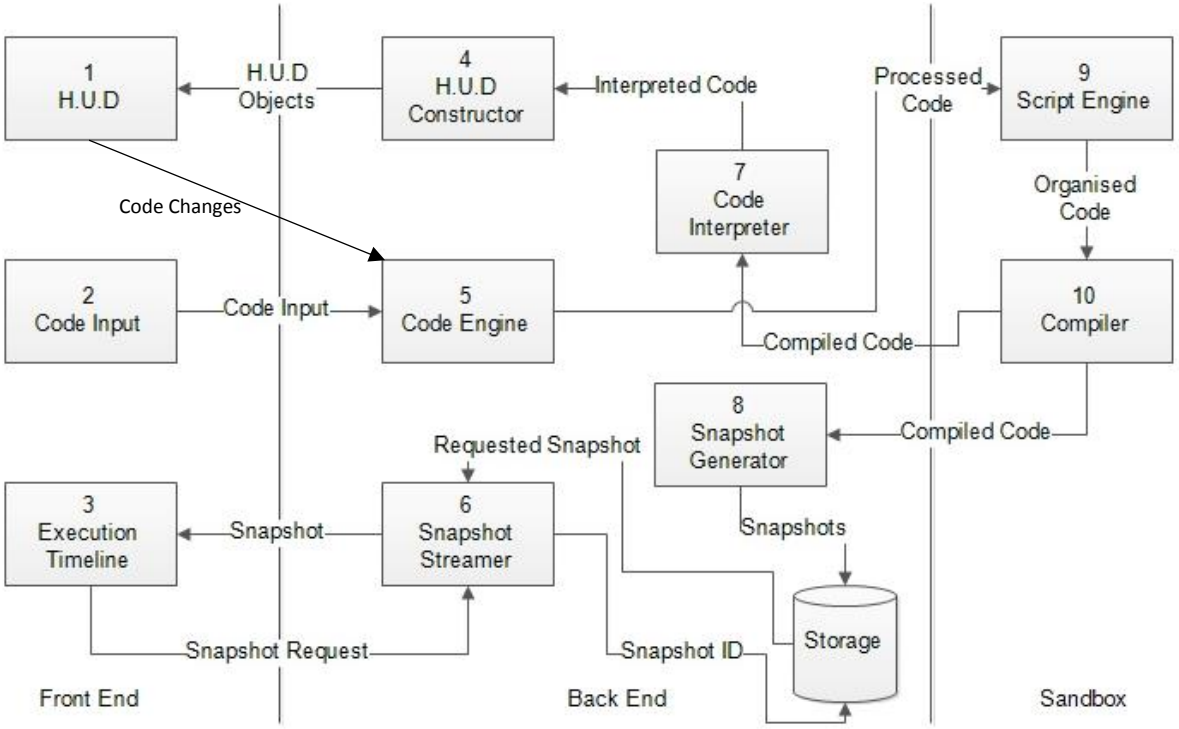
1) H.U.D – Heads Up Display. The H.U.D. contains all the visualisation data and objects, as well as presenting them to the user
2) Interpreter - this interprets input code/compiled code from the compiler in a fashion that can be understood by the H.U.D.
3) Compiler - a sandboxed compiler that compiles input code, and code changes made by the user in the H.U.D

3.2.2.2 - Tier 2 Design – Systems Overview



1) *See 1 Above*
2) Code Input – A module that accepts .txt or .cs files for interpretation
3) H.U.D. Constructor – This module constructs H.U.D. objects such as variable lines and function boxes as per the input interpretation of code
4) *See 2 Above*
5) *See 3 Above*

3.2.2.3 - Tier 3 Design – Functional Overview



1) *See 1 Above*
2) *See 2 Above*
3) Execution Timeline – an optional extra, a widget in the UI that would allow movement of a timeline through the program's execution states.
4) *See 3 Above*
5) Code Engine – passes input code to the script engine if it needs interpreting, also sorts which parts on the program need re-compiling after a code change from the H.U.D.
6) Snapshot Streamer – processes snapshot requests and returns required snapshots of the program to the execution timeline
7) *See 4 Above*
8) Snapshot Generator – runs the program in a sandboxed environment, snapshotting all relevant information (such as variable values, program counter, memory status etc.) for every CPU cycle of the program to create an Array of program snapshots. This Array is then saved to disc.
9) Script Engine – manipulates the sandboxed compiler to compile relevant pieces of program code
10) *See 5 Above*

3.3 – Software Construction

After the designing modelling phase was completed, the researcher moved into making proof-of-concept code bases, to experiment with which approach would yield the most time-optimised result.

He chose to follow an AGILE development philosophy, due to these three constraints:

1. The project is Time Boxed; as such the predictable schedule from an AGILE methodology is ideal
2. The requirements are evolving rapidly; AGILE has the reaction speed to keep up with the changes
3. The project is heavily user focussed; since the key goal is to be of high usability to users, feedback on a regular basis is essential.

3.3.1 - Exploratory Development

3.3.1.1 - Bespoke C++ Lexer { v1.x }

The researcher's first foray was to start completely from scratch. He began building his own bespoke C++ Lexer (also written in C++) to handle code inputs.

This approach yielded results after around two weeks. The Lexer built could stream out a token tree (of a select few tokens) correctly, and the data was formatted into a list of token objects. This led to easy manipulation of the data further downstream.

Though results were optimal in terms of customisation of the lexer itself, the estimated time required to complete a complete C++ Core library compiler was just too large. This branch did teach valuable lessons about what kind of output data stream was needed.

3.3.1.2 - Toy Language Interpretation { v2.x }

The second branch of exploratory development led the researcher to using a pre-built, open-source compiler, written as a C++ executable. This compiler functioned on a small subset of C, which was modified to make it easier to understand for novices.

This route was abandoned as the toy language did not include object orientation, which the researcher felt should be in the software to properly reflect modern paradigms. He also had no efficient way of extracting the tokenised output from this compiler. Finally, the quantity of adaptation required to make this program function as part of another system was infeasible.

3.3.1.3 – Interlude of Thought

In short, the researcher quickly arrived at the conclusion that it was inefficient to create a 3D visualisation environment from scratch. As such he began to search for good ways of displaying his data visually. It may seem strange to begin the search here instead of earlier, however without fully comprehending the practicalities of the data which needed to be displayed, he would have been left without a firm grasp of what exactly he was looking for.

After some investigation into the various front ends that could be used, the researcher selected Unity. Unity was sufficiently open to code level alterations that the output of the chosen back end could be easily fused into it, and came with built-in virtual reality support.

3.3.1.4 - Unity Integrated C# Interpretation { v3.x }

The researcher attempted to use Unity instead of merging with a separate back-end compiler, since Unity has built-in C# and JavaScript compilers. He attempted to write a custom .dll to re-route the tokenised form of certain in-engine scripts back into the engine itself in a script readable format.

However, the level of coupling within the Unity engine relating to the scripting platform was too high to unpick neatly or understand easily. As such, it was not clear how to implement such a system properly, and the time required to study Unity to create such a system was deemed unacceptable.

3.3.1.5 – Unity Integrated LUA Interpretation

The researcher found a plugin called MoonSharp. MoonSharp is a Unity plugin that contains a stand-alone LUA interpreter, and is open source. This was ideal for his purposes, as it allowed opening the interpreter and extracting whatever data was required from it.

3.3.1.6 – Exploratory Conclusions

This last foray into Unity was the final exploratory development branch. The researcher surmised that he had found the correct front end platform, and had found the back-end platform (MoonSharp).

3.3.2 – Concrete Development { v4.x+ }

The researcher began working on the main development branch of the project. This concrete branch houses all the middleware he wrote as a pathway between MoonSharp and Unity, as well as the visualisation assets created. The following is the progression of the project through the various iterations of version 4.

Note: Each version is built upon the code of the previous version, resulting in a cumulative code base.

3.3.2.1 – MoonSharp Integration { v4.0 }

Version 4.0's development was heavily centred around integrating MoonSharp into Unity, and extracting the data from it.

The researcher wrote the following class to gain access to the data he required:

```
publicstaticclassVisualiser_Script{
	privatestaticList<Token> input_tokens =newList<Token>();
	/// Add a token to the input stream
	/// <param name="t">Token to add to list</param>
	publicstaticvoid addToken(Token t){
		input_tokens.Add(t);}
	/// Accessor for the 'input_tokens' list
	/// <returns>list of Token objects bfrom input</returns>
	publicstaticList<Token> getTokens(){
		return input_tokens;}
	/// Empty the token list
	publicstaticvoid Refresh(){
		for(int i =0; i < input_tokens.Count; i++)
		{
			input_tokens[i]=null;
		}
	  input_tokens.Clear();}
	/// Prints the token list
	/// <returns>String conversion of the input</returns>
	publicstaticstring toString(){
		string output ="";
		foreach(Token t in input_tokens)
		{
			output += t.Text;
		}
		return output;}}
```

Figure 3.3.2.1.1 – the Visualiser_Script *class, which creates a manipulatable tokenised output from a LUA script interpreted by MoonSharp [please note – comments have been adjusted to conserve space]*

As the MoonSharp interpreter iterates over the script, each time a token is found it is added into the list of input tokens. The list of tokens is made publicly available to all scripts in the environment, such that any script at any point in time may access it. This is imperative as multiple parts of the visualisation need to access the input list at different times, and may need to access it before they are instantiated.

The method **`Refresh()`** is used when another file is being placed into the system, or a file reload has been called.

3.3.2.2 – VR File Browser { v4.1 }

Upon completion of the MoonSharp integration, the researcher moved onto creating a way of loading files directly into the environment. This process involved creating a functional file browser within the environment. He created a simple file browser from scratch

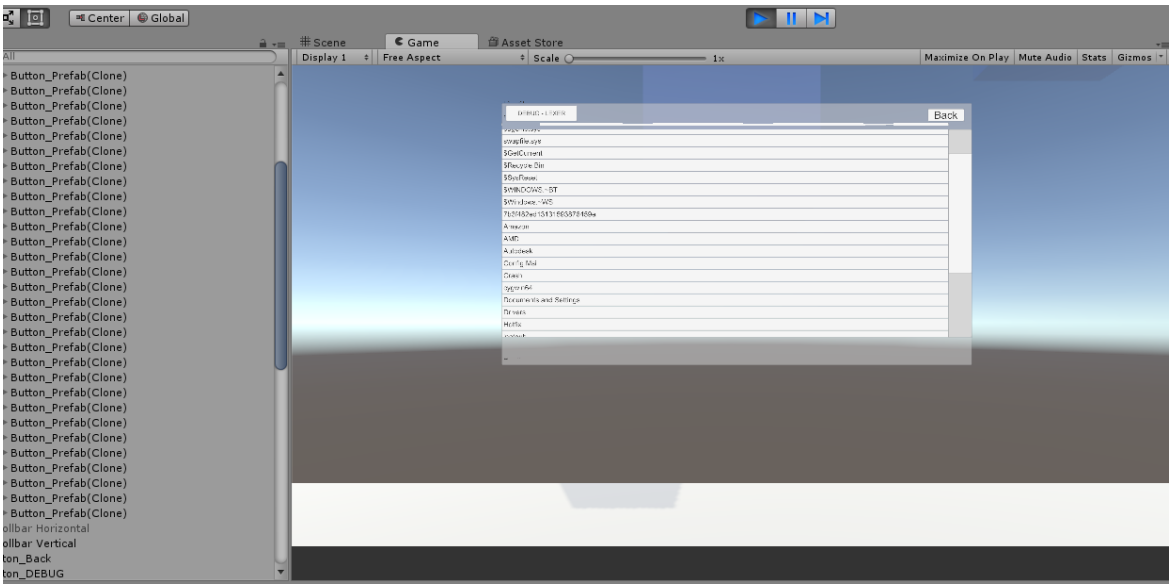


Figure 3.3.2.2.1 – the simple file browser, in-engine

He could not use Unity's .GUI framework, as this was screen-space overlay. This means that it would be as close to the eyes as possible.

Imagine sticking a piece of paper directly over your eyes, and trying to read what's written on it – this is a screen space UI. It is possible, but extremely uncomfortable, and it obstructs your vision. Reading text from a nearby signpost is an example of a world space UI.

As such, the researcher needed to create a world-space user interface. A diagram demonstrating the difference is shown overleaf:

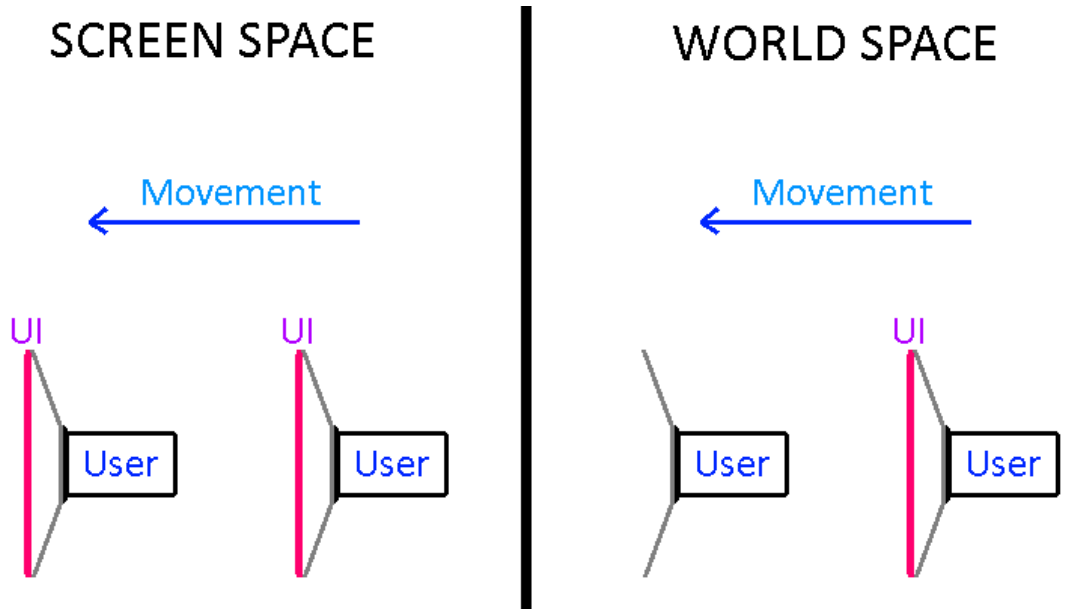


Figure 3.3.2.2.2 – Screen Space UI Vs. World Space UI

Each time the directory is changed, the file browser renders a new set of buttons, and links them up to the directory structure.

However, if the researcher just created and destroyed buttons each time it would cause a memory leak. So, he implemented a pooling system, such that each new button was retrieved from a pool of already existing buttons. If the pool was empty it would create a new button. This meant that the maximum number of button clones in memory was never greater than the largest number of items in any given directory on the machine.

3.3.2.3 – Processing the Interpreter's Output { v4.2 }

Once the file browser was set up, we moved on to implementing the visualisation. This first implementation relied upon a few classes, called **`VisualToken`**, **`Variable`**, and **`GameVisualisation`**.

- **`GameVisualisation`**: The overall controller for the visual representation, this class acted as an independent interpreter for the tokenised output from MoonSharp. It processed the incoming tokens, and created **`VisualToken`** objects relevant to the input token types.

- **`VisualToken`**: A class to store and represent the data of normal statements within a programming language (such as "*if*", "*or*", and "*while*"). **`GameVisualisation`** stores a list of these objects.

- **`Variable`**: A separate data class, used to store variable and function declarations. **`GameVisualisation`** has an internal list of these as well as tokens, and **`VisualToken's`** reference this class if the reference a variable in the source code.

Given overleaf is an excerpt of code form **`GameVisualisation`**:

```
/// <summary>
/// Sets up the list of VisualToken game objects
/// </summary>
privatevoid convertToVisualToken()
      {
      int i =0;
      //Token processing
      while(i < input_tokens.Count)
      {
            //--- V2 :: USED FOR VISUAL PREFABS ---
            // If not already checked, add to list
            if(!input_checked[i])
            {
                  GameObject g = checkToken(input_tokens[i], i,0);
                  try{ g.name = g.GetComponent<VisualToken>().TID;
                  catch{}
                  try{ g.name = g.GetComponent<Variable>().TID;}
                  catch{}
            // Second check is so that post-position argument
               expressions are not added twice
            // As checkToken can cause the current position to
               become checked during execution
            if(!input_checked[i])
            {
                visual_Tokens.Add(g);
                confirmPosition(i);
            }
            }
            i++;}}
```

Figure 3.3.2.3.1 – The main loop of class GameVisualisation*,* convertToVisualToken()

This method is the main processing loop of the **GameVisualisation** class. At the initialisation of a file, a new **GameVisualisation** object instantiates, and processes the tokenised output from MoonSharp. This method processes the input per an internal set of interpretation rules. To prevent tokens from being scanned multiple times, a confirmation list is created. As each token is processed and added to the **visual_Tokens** list, its position in the confirmation list is set to true.

The following is an excerpt from the internal interpreter:

```
caseTokenType.Name:
      {
            g = createVariable(t, t.Text,current_parents.Last(), scope,
             local);
            break;
      }
caseTokenType.And:
      {
            g = createDoubleExp(t, position, callLevel,"And");
break;
      }
```

```
/// Processing method for double adjacency expressions
/// <param name="g">Game object to be configured</param>
/// <param name="position">Position in the token list</param>
/// <param name="callLevel">recursive call level of the current
      checkToken() call</param>
/// <returns>DoubleExp formatted gameObject</returns>
privateGameObject createDoubleExp(Token t,int position,int
callLevel,string name)
      {
      GameObject g = loadPrefab(name, t, current_parents.Last(), scope);
      // Arg1 -- Left hand side
      // Arg2 -- Right hand side
      int arg1 = position;
      int arg2 = position +2;

      visual_Tokens.Add(g);
      confirmPosition(position);
      GameObject arg_2 = checkToken(input_tokens[arg2-1], arg2-1,
      callLevel +1);
      visual_Tokens[position+1].GetComponent<VisualToken>().addArgs(
      visual_Tokens[arg1], arg_2);

      //Add self to parent list, and set parents of arguments as self
      current_parents.Add(visual_Tokens[position+1]);
      visual_Tokens[position+1].GetComponent<VisualToken>().Arg1.
      GetComponent<Variable>().Parent = current_parents.Last();
      visual_Tokens[position+1].GetComponent<VisualToken>().Arg2.
      GetComponent<Variable>().Parent = current_parents.Last();
      removeLastParent();

      //Return expression token, at Last-1
      return visual_Tokens[position+1];
}
```

Figure 3.3.2.3.2 – an Excerpt from the internal Interpreter, demonstrating a double expression [please note – comments have been adjusted to conserve space.]

The token type is stored as an **`Enum`**, which is the input for the switch. **`createDouleExp()`** creates a two-sided statement object (such as "*And*") and sets up the parameter references. These parameter references are used later to create the context-sensitive debugging information that is displayed to the user.

3.3.2.4 – Displaying the Visualisation { v4.3 }

It was at this point that the researcher stepped aside from the code to create the 3D models. He had created a set of models for each syntactic construct in the core library of LUA. The models had normal maps and were ready to be imported into Unity. Given below is an example of a model:

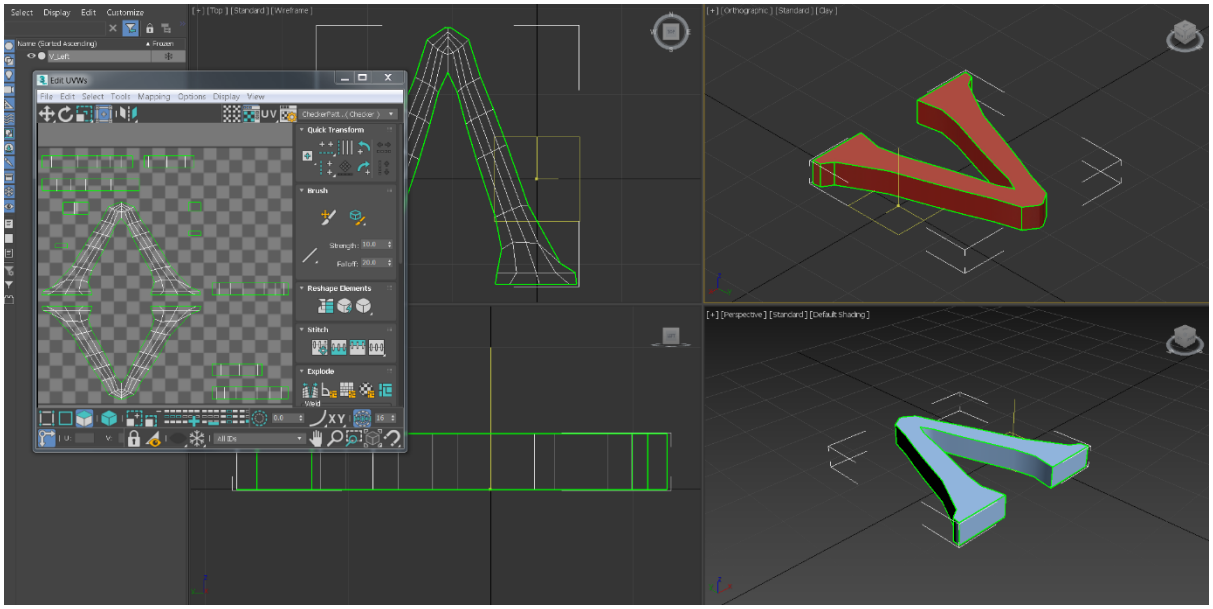


Figure 3.3.2.4.1 – the "And" model, including normal mapping

He designed each model to obey the following principles:

1. Each model must have a distinct shape such that if looked upon as a silhouette they are each distinguishable from each other
2. Each model must be vertex-optimised so that it does not impact performance
3. Each model must be normal mapped, such that textures or colour can be changed at runtime

These principles consider usability issues some users may have. For example, if a user were colour blind or visually impaired in some fashion, the system would still be fully functional and equally expressive.

At this point, the system could output a visual format of the program code in a bare-bones format. An example shot is below:

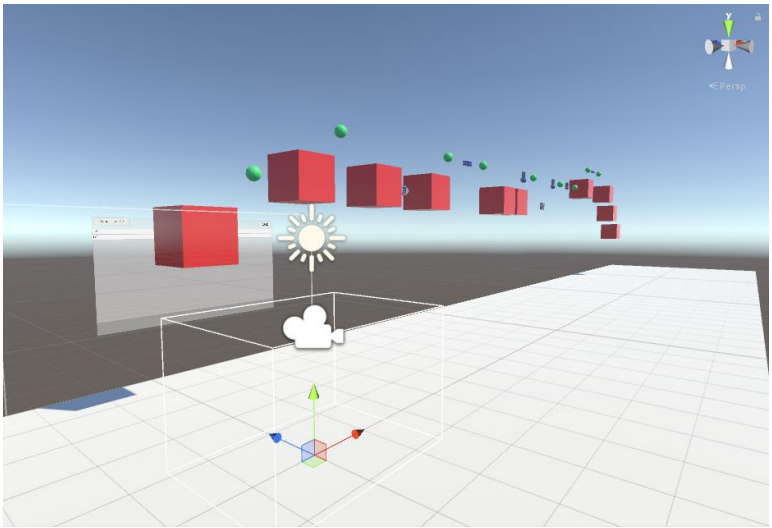


Figure 3.3.2.4.2 – Version 4.3's visual output

Red boxes represent scope switchers (such as "*then*", brackets and "*function*"). Green spheres represent variables, and each blue token represents its respective keyword. The verticality of the display represents the scope of a given statement. The higher in the Y-axis an object is, the lower the object scope. As such, higher objects cannot access objects below them.

This system was counter-intuitive, but due to some physics engine issues we needed to invert the Y-axis at this point in the development cycle. Below is a code excerpt of how objects were positioned:

```
/// <summary>
/// Sets the positions of all tokens in the input list
/// </summary>
privatevoid setPositions()
{
      x_current = x_initial;
      y_current = y_initial;
      z_current = z_initial;

      Vector3 new_Position =newVector3(x_current, y_current,z_current);

      for(int i =1;  i < visual_Tokens.Count; i++)
      {
            // X increment by parent
            x_current = x_current + x_increment;
            // Y Increment by scope
            // SET TO '+' for phyEngine debugging
            try{ y_current = y_initial +(y_increment *
            visual_Tokens[i].GetComponent<Visual_Object>().Scope);}
            catch{}

             new_Position =newVector3(x_current, y_current, z_current);
             visual_Tokens[i].transform.localPosition = new_Position;
      }
}
```

Figure 3.3.2.4.3 – method to re-position all visualToken *objects in the environment post setup*

This method is called after the completion of the **`convertToVisualToken()`** method. It aligns all visual objects to a pre-determined set of co-ordinates, relative to the size of the program.

3.3.2.5 –Processing Optimisation Pass { v4.4 }

The researcher re-wrote the entire visualisation code, as the previous version did not give the expressive ability required. The rewrite followed this structure:

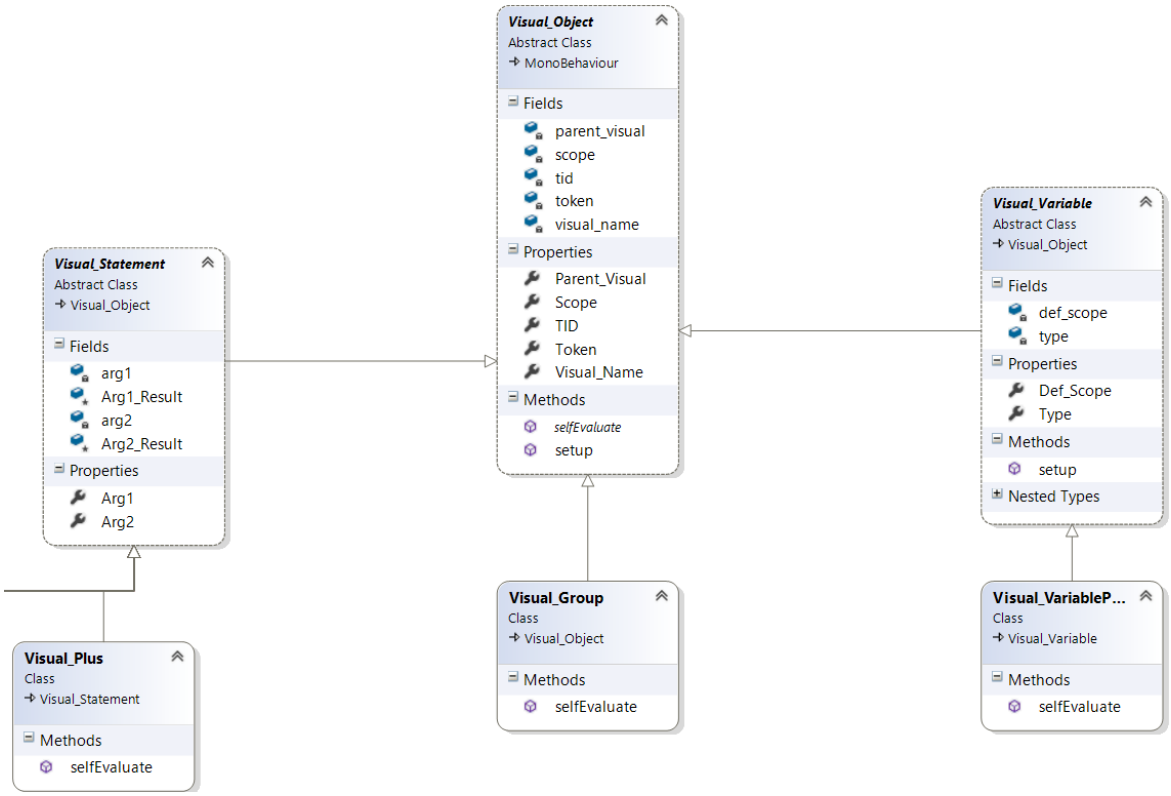


Figure 3.3.2.5.1 – abstract classes used in the newest implementation of the visualisation

This division of labour between three abstract classes allowed greater precision in the behaviours required. As can be seen, all concrete implementations of the top level **`Visual_Object`** class inherit **`selfEvaluate()`** and **`setup()`**. These abstract methods are overridden in each individual token (such as “And”, “For” and “If”) to provide the desired behaviours.

Variables are defined primarily by the two **`enum`** that are stored within them; **`def_scope`** and **`type`**. These two **`enum`** are shown below:

```
publicenumvar_Type{FUNCTION,VARIABLE,LABEL }
publicenumvar_Scope {LOCAL,NORMAL,GLOBAL }


// Type of this variable
privatevar_Type type;
publicvar_Type Type{get {return type;}}

// Was this variable declared as Local?
privatevar_Scope def_scope;
publicvar_Scope Def_Scope{get {return def_scope;}}
```

Figure 3.3.2.5.2 – Variable object Enums*, extracted from the* Visual_Variable *class*

The **`var_Type enum`** stores whether this current variable token is a function declaration/call, a local variable (such as "*n*" or "*hello*") or a defined label for **`GoTo`** statements. This distinction is important, as in LUA all the above get stored as tier-1 data objects in the global environments table. This makes it hard to tell them apart in-context from the token information alone.

The **`var_Scopeenum`** stores the defined scope of this variable. **`NORMAL`** scope respected the scope hierarchy (cannot access object in scopes above, can access objects below), **`LOCAL`** allows only objects in the same scope level to be accessed, and **`GLOBAL`** allows any object to access this value.

By combing the values of these two **`enum`** with the token data, it is possible to determine exactly how a variable will behave in the execution cycle. This is important for our context sensitive debugging, which is implemented in a later version.

3.3.2.6 – VR Optimisation pass { v4.5 }

Although the system at this point was functional, it had a plethora of small issues that prevented it from functioning well on an HTC Vive.

The first series of issues we resolved was centred on the file browser. As the researcher attempted to attach the current file browser to the headset in-engine, he found that the browser was simply too large to easily read. Text was also too small, and would strain the eyes. As such, he re-designed the browser and brought it to a size that was readable:

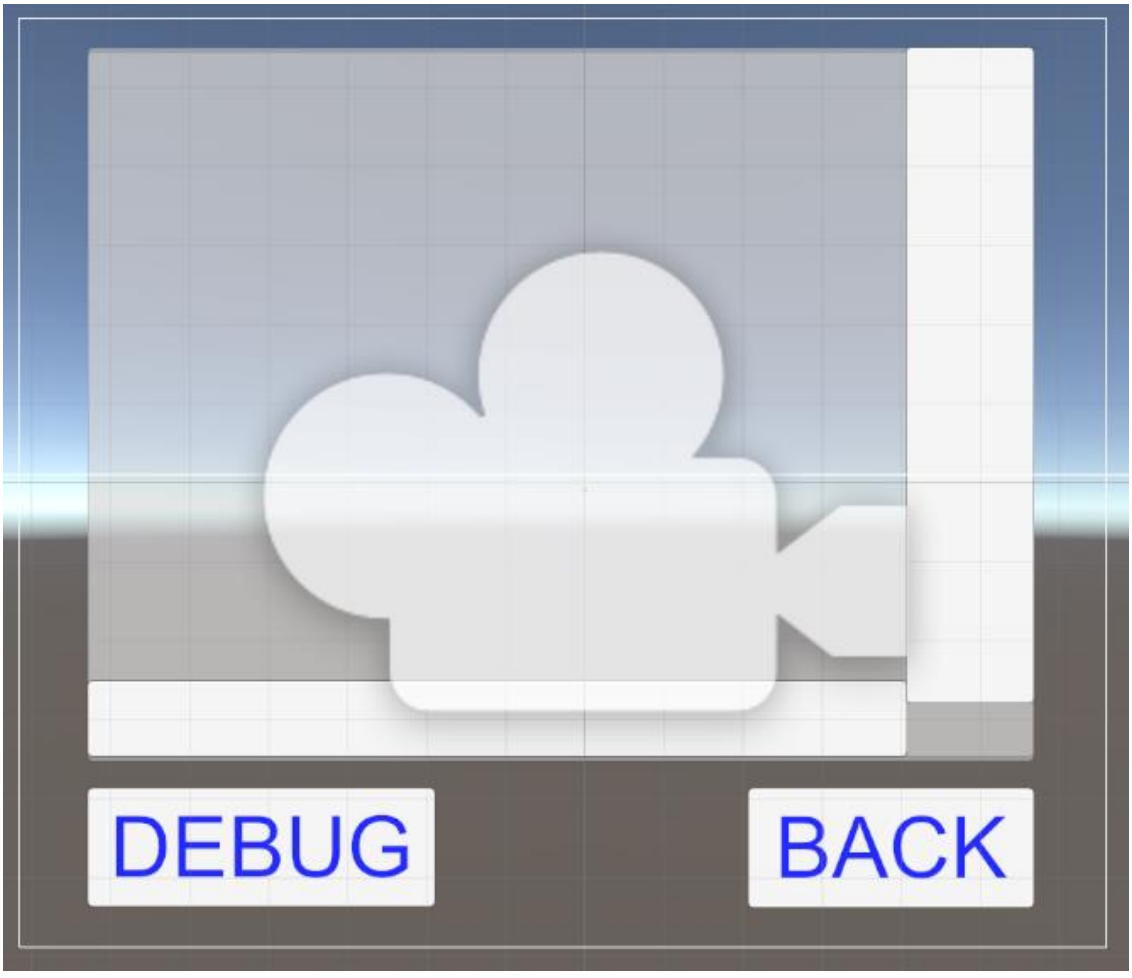


Figure 3.3.2.6.1 – the second implementation of the file browser, now attached to the headset

"*Attached to the headset*" means that the file browser will follow the user and always be in front of them, if active.

The physics engine issues were fixed in this version. Previously, code objects would collide with everything in the environment (including themselves) which led to undesired visual behaviour.

3.3.2.7 – Variable Referencing { v4.6 }

Advanced behaviours in the **`Visual_Object`** hierarchy were implemented. This resolved variables not providing the correct context-sensitive output. The outputs were not connected to a display in this version, but the console. An example of the change is given below:

```
publicclassVisual_Assign :Visual_Statement
{
	override
	publicstring selfEvaluate()
	{
		Arg1_Result =
		this.Arg1.GetComponent<Visual_Object>().selfEvaluate();
		Arg2_Result =
		this.Arg2.GetComponent<Visual_Object>().getValue();
		return(Arg1_Result +" <- "+ Arg2_Result);
	}

	override
	publicstring getValue()
	{
		return selfEvaluate();
	}
}
```

Figure 3.3.2.7.1 – added functionality to the Visual_Object *hierarchy, shown in the* Visual_Assign *class*

The researcher separated the functionality of **`selfEvaluate()`** out into two separate methods; **`getValue()`** and **`selfEvaluate()`**. The **`getValue()`** method returns the default value of an object, and **`selfEvaluate()`** returns the context-sensitive information for that object. In **`Visual_Assign`**, **`getValue()`** is never properly utilised, as the keyword itself has no default value. However, in the **`Visual_VariablePoint`** class, this method is fully utilised. The **`Visual_VariablePoint`** class is shown overleaf:

```
publicclassVisual_VariablePoint:Visual_Variable{
/// Returns a string to display this variable
/// <returns>Name and Value of this variable</returns>
publicoverridestring selfEvaluate()
{
      if(Type ==var_Type.VARIABLE)
      {
            if(transform.parent
                  .GetComponentInParent<GameVisualisation>()
                  .variableRefs.ContainsKey(Visual_Name))
            {
                  string value = getDictValue();
                  if(value !=null)
                  {
                        if(value.Contains("ERR"))
                        {return Visual_Name;}
                        Else
                        {return Visual_Name +" : "+ value;}
                        }
                  else{returnToken.Text;}
                  }
            else{returnToken.Text;}
            }
      else{returnToken.Text;}
}
      /// Returns this variable's value ONLY
      /// <returns>String evaluation of this variable</returns>

override
publicstring getValue()
{
      string value = getDictValue();
      if(value !=null)
      {
            if(value.Contains("ERR"))
            {return Visual_Name;}
            else{return value;}
            }
      else{return Visual_Name;}}
      /// Method that extracts the value of this variable from the
       refrence dictionary
      /// <returns>Value of variable in Dictionary</returns>
privatestring getDictValue()
{
      int index =
            transform.parent.GetComponentInParent<GameVisualisation>()
            .Visual_Tokens.FindIndex(o => o == gameObject);
      if(transform.parent.GetComponentInParent<GameVisualisation>()
            .variableRefs.ContainsKey(Visual_Name))
      {
            string value =
            transform.parent.GetComponentInParent<GameVisualisation>()
            .variableRefs[Visual_Name][index].VarValue;
            return value;
      }
      else{returnnull;}
      }}
```

Figure 3.3.2.7.2 – the Visual_VariablePoint *class, demonstrating the context-sensitive value analysis [NOTE: Comments, brackets and white space adjusted to save space and improve readability]*

The **`Visual_VariablePoint`** utilises a new piece of functionality added in this version, the context-sensitive value analysis. This analysis detects the value of a given variable *at that point* in the program code. For example, if on line 10 of a program you assign {**`n := 17`**}. Further on, on line 23, you assign {**`n := 21`**}. The analysis code would ensure that when you look at the definition of ***n*** on line 10 it would read as 17 on the display, and when looking at line 23, it would display ***n*** as 21.

The context-sensitive analysis data is generated by parsing the list of **`Visual_Objects`** after they have been processed, and storing the data in a bespoke dictionary. The data structure is given below:

```
// List that stores a reference to all gameObjects clasified as
variables
// KEY: string NAME
// VALUE: Dictionary of References to NAME: [INDEX, REFERENCE]
publicDictionary<string,Dictionary<int,Variable_Ref>> variableRefs;

/// Class that stores all relevant data for a variable reference
publicclassVariable_Ref {

/// GameObject reference for this variable
privateGameObject varObj;
publicGameObject VarObj
{ get {return varObj;}}

/// Index reference for master list for this variable
privateint varIndex;
publicint VarIndex
{ get {return varIndex;}}

/// Value reference for this variable
privatestring varValue;
publicstring VarValue
{ get {return varValue;} set { varValue = value;}}

/// Constructor for class Variable_Ref
    /// <param name="obj">GameObj Reference</param>
    /// <param name="ind">Index Reference</param>
    /// <param name="val">Value Ref</param>
publicVariable_Ref(GameObject obj,int ind,string val)
{
        varObj = obj;
        varIndex = ind;
        varValue = val;
}
}
```

Figure 3.3.2.7.3 – the context-sensitive analysis dictionary and data class [NOTE: Comments, brackets and white space adjusted to save space and improve readability]

The string key in the outer dictionary provides the variable name, and the inner dictionary stores all **`Visual_Object's`** that are a matching variable name. This name is taken from the Token objects from the lexer, so there's no chance of mixing names up between similarly named variables. The inner dictionary stores an integer key, which is the line that specific variable instance appears on, and a **`Variable_Ref`** object. The **`Variable_Ref`** object stores the variable's information as given at that line of the program. The method used to create these references is given below:

```
/// Method to create a double-layered dictionary of variable references
publicvoid createReferences(){
      string name;
      for(int i =1; i < visual_Tokens.Count; i++)
      {
      if(visual_Tokens[i].GetComponent<Visual_Object>()!=null){
            name =visual_Tokens[i].GetComponent<Visual_Object>()
            .Token.Text;
      // IF is variable
      if(visual_Tokens[i].GetComponent<Visual_Variable>()
            !=null){
      // IF not local
      if((visual_Tokens[i].GetComponent<Visual_Variable>()
                  .Type ==Visual_Variable.var_Type.VARIABLE)&&
                  (visual_Tokens[i].GetComponent<Visual_Variable>()
                  .Def_Scope !=Visual_Variable.var_Scope.LOCAL)){
      // IF refs does not have a reference
      if(!(variableRefs.ContainsKey(name))){
      Dictionary<int,Variable_Ref> refs =
                        newDictionary<int,Variable_Ref>();

                        refs.Add(i,newVariable_Ref(visual_Tokens[i],
                        i,"ERR_NOT_ASSIGNED"));
                        variableRefs.Add(name, refs);}
      else{
            variableRefs[name].Add(i,new
            Variable_Ref(visual_Tokens[i], i, "ERR_NOT_ASSIGNED"));
      }}}}}
      assignReferences();}

/// Method that assigns values to variable references
privatevoid assignReferences(){
      string name;
      for(int i =1; i < visual_Tokens.Count; i++){
            if(visual_Tokens[i].GetComponent<Visual_Object>()!=null){
            if(visual_Tokens[i].GetComponent<Visual_Assign>()!=null){
                name = visual_Tokens[i -1].
                  GetComponent<Visual_Object>().Token.Text;
            string value = visual_Tokens[i +1].
            GetComponent<Visual_Object>().Token.Text;
            // Assign value of [i-1 -> end] as value of [i+1]
            foreach(KeyValuePair<int,Variable_Ref> kvp in
                        variableRefs[name]){
            if(kvp.Key >=(i -1)){
                        variableRefs[name][kvp.Key].VarValue = value;
}}}}}}
```

Figure 3.3.2.7.4 – methods that create and assign variable references to the context-sensitive analysis dictionary [NOTE: Comments, brackets and white space adjusted to save space and improve readability]

These methods will first create an empty reference for each instance of a variable in program code, then fill that reference per assignments made within the code. This is performed after the **`Visual_Object`**'s have been initialised, so that if an error occurs in this step, it can still display the code in the visual environment.

3.3.2.8 – Fixing the Nausea Issues { v4.7 }

The researcher found that after approximately thirty minutes of headset usage, he began to feel ill. He performed some additional experimentation at this point to figure out how to alleviate his sickness.

His finding was simple: he needed a physical environment so that his eyes had a point of reference for all activity. So, he created a simple environment for the user, to provide said reference points; see below:

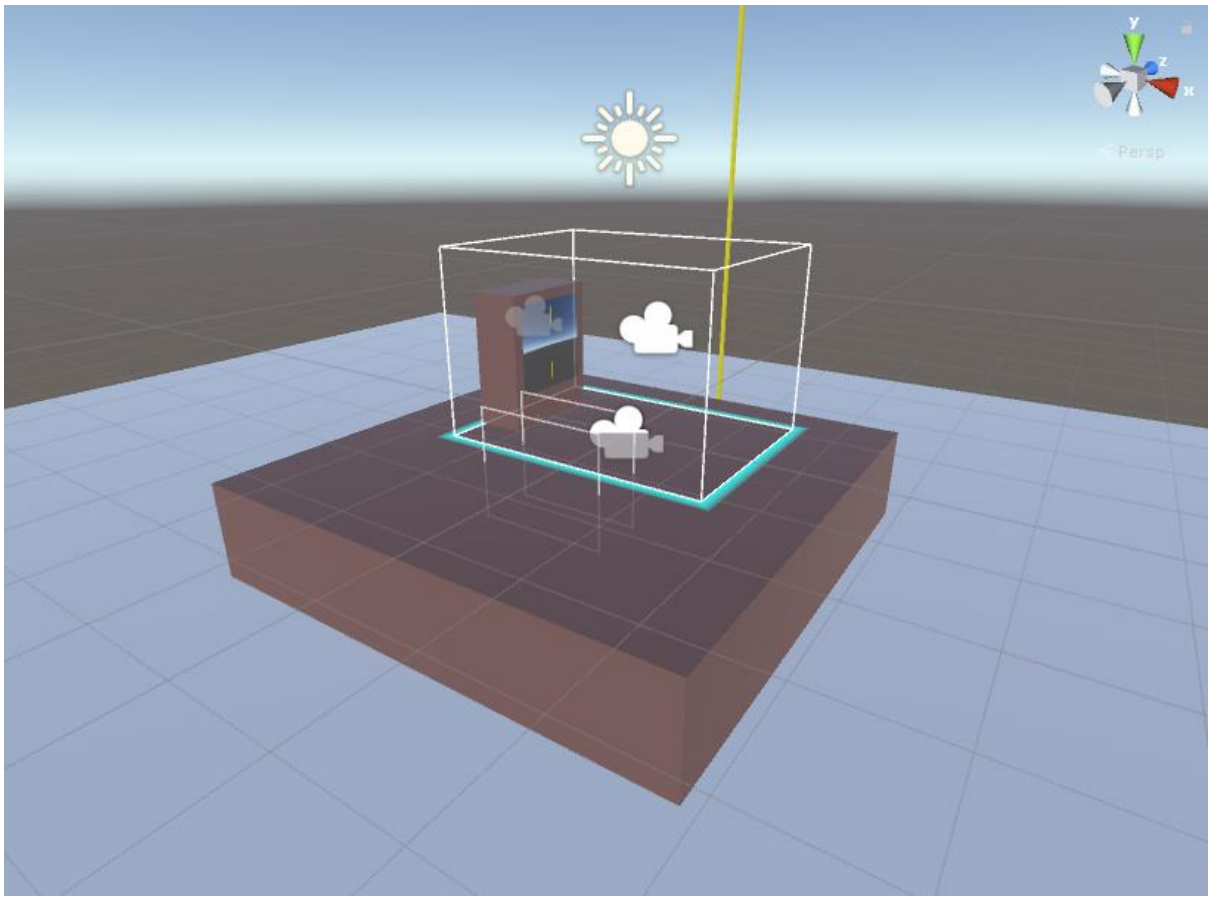


Figure 3.3.2.8.1 – the simple user environment created to ease nausea

Just by adding this platform and lighting, he managed to alleviate some of the nausea.

The monitor seen in the image above is part of the functionality he added this build – the CodeGuide camera. The CodeGuide is a viewpoint that provides the user an overarching view of all program code loaded into the environment. This feature prevents the user from "Getting Lost" in their own code.

As part of the CodeGuide initiative, he implemented the ability to move a group of code objects around in the environment by utilizing the trackpads on the Vive controller. By moving thumbs around on the trackpads, the code would move in the XY plane.

3.3.2.9 – Code Interactions { v4.8 }

Once the code could be reliably loaded into the environment, the researcher focussed on interaction with said code. This involved three key steps:

Context Sensitive Information Display

This involved adding a basic H.U.D. to the helmet display, which included a crosshair and three text displays. These text displays were angled to provide a more comfortable viewing experience.

Ray Cast-Based Detection Code

By adding a ray casting source to each controller and the headset, he could obtain the string output of **`selfEvaluate()`** from each object the ray cast hit. This was done using the following code structure:

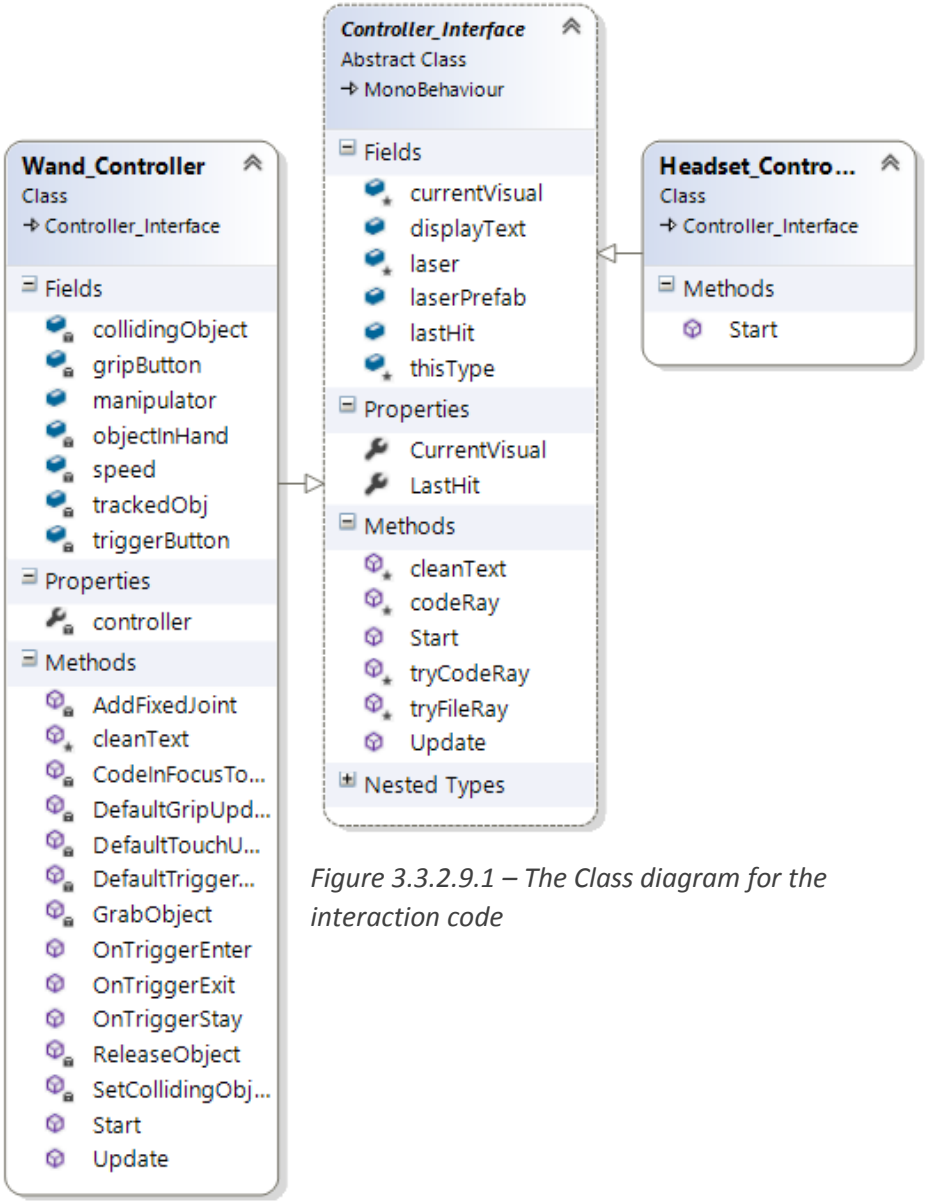


Figure 3.3.2.9.1 – The Class diagram for the interaction code

`Controller_Interface` is the class that stores the default behaviour of any control interface used to interact with the environment. Within it, there are ray cast methods appropriate for hitting and valid object type (such as code objects). The variable **`thisType`** is an **`enum`**, which stores whether a specific instance of this class is a controller or a headset. This is important, as headsets and controllers behave differently, but utilize the same basic ray cast code.

Wand_Controller is used for the Vive controllers, and **Headset_Controller** for the headset. The headset only displayed code on the central text display when looking at a valid code object, whereas the controllers behave based on button presses.

Holding the grip button on the Vive controller summons a cyan laser, which when pointed at a valid code object will display the context sensitive readout in the relevant controller display. When this laser is pointed at a button or UI element, it will interact with the button or UI element. Holding the trigger of the controller will summon an orange laser, which can grab code objects and move them around. An excerpt from **Controller_Interface** is given below:

```
// Update is called once per frame
virtual
publicvoid Update(){
Ray C_raycast =new Ray(transform.position, transform.forward);
RaycastHit C_hit;
bool bHit =Physics.Raycast(C_raycast,out C_hit);

if(bHit){
GameObject result = codeRay(C_hit);
if(result !=null){
if(!(result.name =="noValidTarget")){
                lastHit = result;}}
}else{
        cleanText();
        lastHit =GameObject.Find("noValidTarget");}}
/// Method that summons a codeRay cast behaviour
/// <param name="hit">object hit</param>
/// <param name="cast">type of device causing the cast</param>
/// <param name="pointer">pointer object (if there is one)</param>
/// <param name="origin">origin of the pointer object</param>
/// <param name="display">Text display to update</param>
/// <returns>Result of raycast</returns>
protectedGameObject codeRay(RaycastHit hit){
if(thisType ==CAST_TYPE.CONTROLLER){
        laser.transform.position =Vector3.Lerp
            (transform.position, hit.point,.5f);
        laser.transform.LookAt(hit.point);
        laser.transform.localScale =newVector3(
            laser.transform.localScale.x, laser.transform.localScale.y,
             hit.distance);
}
switch(hit.collider.tag){
      case"File_Button":{
            if(thisType ==CAST_TYPE.CONTROLLER){
                    tryFileRay(hit);
            }
            return null;}
      case"Code_In_Focus":{
            return tryCodeRay(hit);}
      default:{
            return GameObject.Find("noValidTarget");}}}
```

Figure 3.3.2.9.1 – excerpt from class Controller_Interface*, demonstrating the ray cast behaviour [NOTE: Comments, brackets and white space adjusted to save space and improve readability]*

3.3.2.10 – Environment Rework and Optimisation Pass { v4.9 }

The next version of the software was centred on environment optimisation, to further reduce nausea, and modifying the file browser.

The environment was further enhanced with lighting changes, bounding walls and additional interfaces. These interfaces displayed the File browser (which had been moved onto a fixed screen), the CodeGuide camera, and a list of all currently loaded files.

This list of currently loaded files, along with the functionality required to add, remove and modify the display order of files, was added to pave the way for multi-file interactions.

3.3.2.11 – Multiple File Loading { v5.0 }

This build of the project aimed to introduce a basic API for multi-file loading, and cross file interactions. This consisted of creating a master environment controller that stores a dictionary of all currently loaded files. This dictionary is keyed by the file name, so that function calls within files know where they are defined.

Due to the limitations of the interpreter used, cross-file referencing was not fully implemented. This was due to no way of distinguishing the prototype of a function from the token list. As such, the API was left implemented, but unused such that future developments with a different interpreter had some of the foundation work already completed.

3.3.2.12 – Finalisation and Bug Fixing { v5.1– v5.5}

The final four builds were centred on sprints to address user feedback, bug fixing and optimisation. The key areas addressed were:

1. Colour coding – based upon iterative user feedback from a few volunteers, the researcher constantly tweaked colour levels and opacity in the environment to provide an optimally-readable display
2. Object layout – this section focused on optimising the 3D layout of code object, and condensing them down into a more readable format
3. Token testing – this involved creating several sample scripts, and testing each syntactic construct in LUA. Time was short at this point, so testing exercised approximately 60% of LUA's core syntax. This left enough tokens to create valid testing programs.
4. Control input – this was optimised controller input, as the initial configuration was too difficult to use for people who had never used an HTC Vive before.

There were also a number of small bug fixes, too many to count. Although not all were fixed, see Evaluation for a further explanation.

3.4–Evaluation and Testing

3.4.1 User Testing

3.4.1.1 – Methodology

The testing was aimed at discovering how users interacted with the interface presented to them, and their ability to understand it.

In short, the test conducted was how quickly subjects could figure out what a given piece of program code was doing in two separate environments – The software system, and a more traditional environment (Notepad++).

3.4.1.2 - Participants

A total of 13 subjects were gathered as potential candidates. These participants were selected based on interest in the project brief, from other student groups. All participants were between the ages of 18 and 24. Most participants were male.

3.4.1.3 – Materials

The following materials were used during testing:

3.4.1.3.1 Test Environments

The testing took place in a secluded laboratory, where users were presented with an HTC Vive running the software, as well as a normal machine running Notepad++.

3.4.1.3.2 Pre-test Data

Before the test, subjects were asked to complete a short questionnaire to give a little perspective to their programming background. The questionnaire can be seen in the Appendix [6.5].

A results table (personal information omitted) can be viewed in the Appendix [6.6].

3.4.1.4 Procedure

Participants first received 5 minutes with a small LUA script, and were asked to familiarise themselves with the language. During this period, they were free to ask any questions pertaining to LUA or its implementation.

After this period, the first stage of testing began. Participants were given one of two scripts in Notepad++, and given 5 minutes to analyse the script to figure out what it is doing. Participants were asked to think aloud, and were recorded. If the participants believed they had found the purpose, they were asked to explain it to the researcher. If correct, the time was recorded and the test was stopped.

If a participant had not discovered the meaning during the 5-minute period, they were asked at the end of the test to explain their best guess. If their guess was correct, it was recorded that they had figured it out by "*the end of the test*", else they were recorded as not having understood.

Next, participants were given the HTC Vive and the software system, loaded with the same orientation script from the earlier stage. Participants were given 5 minutes to acclimatise themselves to the HTV Vive and the software system, and had the option of completing the Vive's basic interface tutorial at this stage.

Participants were then given the second of the two scripts loaded into the Vive environment, and asked to repeat the same process as above.

After the Vive test, participants were invited to give verbal, informal feedback about the system, as well as discuss where they thought the system would be useful.

Each participant received a different script first, such that there was no bias in script design. All three mentioned scripts can be seen in the Appendix [6.7]

3.4.1.5 – Data Collection

Data was collected from participants through an initial questionnaire, which participants filled out on a separate computer. During the test, data was collected in the form of audio recordings, which were split by section of the test currently in progress. As well as audio recordings, observations were made of the participants.

3.4.2 - Results

3.4.2.1 – Results Gathered from the Participants

ID	*file1*	*Guess time*	*file 2*	*Guess time*
128249	test1.lua	4m 33s	test2.lua	N/A
258978	test2.lua	N/A	test1.lua	End of test

Given below are transcript summaries of the subject's informal interview after the tests:

3.4.2.2 – Subject 128249's Informal Observations

1. Within a short amount of time, it's easy to understand what is happening
2. Extra-dimensional indentation is useful
3. Large code bases may cause it to fall apart in terms of readability
4. Could be useful in a teaching environment, or a small tech company
5. It is another way of sharing or debugging code
6. Is more intuitive than some ways of displaying code
7. Difficult to judge what detail is needed
8. The layout is well presented, but rough around the edges
9. Main issues currently are colour and layout oriented
10. It's good fun, and would be a good teaching aide
11. Good for exploring a code base

3.4.2.3 – Subject 258978's Informal Observations

1. Some parts made sense, but there is a lot going on
2. Layout is too spread out
3. Having to look at objects instead of data being explicitly displayed is frustrating
4. It could be difficult to understand
5. Unsure of where to use the system
6. More of a secondary approach to traditional debugging
7. System is an alternative method of viewing code, from a new perspective
8. Vive is a good hardware platform to deploy the system to
9. Colour and transparency was not clear

3.4.3 – Discussion

Unfortunately, due to hardware failures out of the researcher's control, he could only test two of the candidates at the time. However, their insights are enough to begin a discussion.

The common consensus from these two subjects can be summarised in a few key points:

1. Functionality of the system is good, and has potential uses industrially
2. Presentation of the system is lacking, and requires refinement
3. Layout required adjusting, as it obscures detail

Alongside these obvious observations, the researcher also noticed that there was some obvious confusion regarding the Vive inputs while demonstrating the project at a public event. The common misconceptions were:

1. Users would attempt to swipe the trackpad like a tablet monitor, instead of using it like a pressure sensor
2. Some users complained that the movement of code was inverted, compared to their expectations
3. Very few users managed to properly utilize the grip buttons on the side of the controller
4. Nobody who tested the system could aim the Vive remotes correctly, everyone assumed they were “wands”, whereas they behave like a remote control
5. Most users struggled to aim the controllers properly at objects, even with the haptic feedback given when they were successfully aiming at something

The majority of the issues mentioned above stem from a lack of experience with the deployment platform, not necessarily an issue with the system itself.

3.4.4 –Evaluation

As a whole, the system created is still bare bones. It is missing any form of user-facing documentation (such as tutorials, guides or tips) internally or externally. The only documentation currently available are in-code comments, which primarily focus on explaining the code itself, rather than the logic running the program. The system is not exported as an executable or binary file, and still requires Unity installed on a host machine to run. The system will only function in Windows, and only load in the LUA programming language.

At the code level, the project suffers from high coupling and low cohesion. Though it has a simple abstract hierarchy, lots of classes and methods are over-burdened with multiple types of behaviour. This could be better split out, forming a more complex but less coupled hierarchy. Due to this high coupling testing the system is quite difficult, and tracing errors in the system is challenging. An unintended side effect of this is that the code base itself is very difficult to understand from an outsider’s perspective, even with the copious quantity of comments.

The project’s code is poorly managed. This is a side-effect of the learning process undertaken while writing the system. As the researcher progressed, he learned more efficient methods of implementing the same functionality, however he had no time to go back and re-implement the previous systems. Additionally, to minimise time costs and avoid releasing the code into the public domain, the researcher had to use manual version control. This is because the freely available options would have pushed program code into the public eye, which he wished to avoid.

The system itself also has limitations – artificial restraints imposed upon it, primarily to prevent leakage of incorrectly structured API code that is in place for future adaptability. The key artificial restraints are:

1. The system can only process Integers and Boolean values and logic
2. All assignment statements that have more than a single token on the right-hand side of the assignment must be contained within round brackets
3. The keyword `Local` is non-functional, as it prevents the system’s in-context-analysis of variables from functioning correctly

These artificial restraints were imposed upon the system to force a state of functionality and stability. Without these restraints, the system would have been too unstable to conduct user testing. The code-related limitations are as follows:

1. Functions may only be called or declared with a single parameter
2. Functions cannot be distinguished by prototype, and each function must have a distinct name that does not collide with any other function currently loaded into the system

These code limitations are relics of previous iterations of the software. Given sufficient development time, they could be removed with relative ease. However, they are also linked to the language being interpreted (LUA). In this case, a different base language would be required by the system.

The researcher sees that this software has many applications in education, especially for younger generations. The appeal of wearing a piece of “fun” equipment and “walking” through your own work adds a level of interaction and visual understanding not commonly found when teaching programming skills. The system also has potential for collaborative work, should multiple users all don headsets and walk through code together – these users could be in different geographical locations and on different devices, since the system could be easily adapted for use on alternative devices.

4 – Conclusion

4.1 -Evaluation Summary

The current system has many limitations, some of which are imposed by the researcher. The system is plagued by poorly managed code, however it still functions (within its boundaries) well. The adaption and learning rates for the system are high, and the researcher can already see the high potential offered by this approach to looking at program code in various fields.

4.2 - Testing Summary

The findings were explicit; the layout was difficult to navigate, and the colour coding was not properly implemented. Alongside this, the Vive platform itself caused issues among users, and they found it difficult to use the hardware for the first time. This could be mitigated with in-system tutorials, and proper training to use the device.

4.3 – Possible Expansions

Regarding possible expansions to this piece of software, the researcher has drafted a list, split by whether they need keyboard emulation or not:

Keyboard Emulated Expansions

1. Run an input script file or function with specified inputs
2. Allow for renaming of files in the environment
3. Add colour changing interface, for code objects, the UI, etc.
4. Add a search bar to the file browser and loaded file interfaces

Other Expansions

5. Refactor button-based code into a single hierarchy, thus streamlining Ray Cast hit detection
6. Program the CodeGuide window to display the debug information of the object directly in front of the camera
7. Make selection a toggle-select instead of a hold-select
8. Create Mathematical and Logical token hierarchies, to abstract out common behaviours
9. Fully streamline the codebase, ensuring comments and variable names are all up-to-date with the most recent code branch
10. Attach particle effects to selected objects
11. Create a class hierarchy for conditional statements, to streamline conditional tokens more efficiently

Alongside these advancements, the obvious addition is to change the interpreted language away from LUA to a more structured language, such as C# or Java. This will remove a lot of inherited limitations due to the language, and promote advanced functionality.

4.4 – Closing Thoughts

Overall, the researcher is pleased with the outcome of the project. The system developed has outstanding potential in the educational sector (which was not originally anticipated) and in industrial environments. The system is far from complete, and needs a significant amount of work to be completed. However, he is satisfied with this proof of concept and is looking forward to how the system grows in the future.

5 - 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.