Immersed in Software Structures: A Virtual Reality Approach

Roy Oberhauser and Carsten Lecon
Computer Science Dept.
Aalen University
Aalen, Germany
email: {roy.oberhauser, carsten.lecon}@hs-aalen.de

Abstract - **Program code has been inherently challenging to visualize due to its abstract nature. With the advent of affordable virtual reality (VR) hardware products, the use of VR has become a feasible option for software tools. In this paper, we describe a VR approach for visualizing internal program code structures and evaluate its suitability for selected software development and education tasks. VR efficiency and effectiveness was equivalent to mouse and screen mechanisms, but provided an enhanced user experience for 70% of the subjects, whereas 30% experienced VR sickness.**

Keywords - Virtual reality; software visualization; program comprehension; software engineering; engineering training; computer education.

I. INTRODUCTION

With the ongoing digitalization of society, the amount of program source code produced and maintained worldwide is dramatically increasing. Google alone has at least 2bn LOC accessible by 25K developers [1], and it has been estimated that well over a trillion lines of code (LOC) exist worldwide with 33bn added annually [2]. Aristotle stated, "thought is impossible without an image", and F. P. Brooks, Jr. asserted that the invisibility of software is an essential difficulty of software construction because the reality of software is not embedded in space [3]. Common display forms used in the comprehension of source code include text and the two-dimensional Unified Modeling Language (UML). Program comprehension limitations are evident in the relatively low code review reading rates of around 200 LOC/hour [4].

Feijs and De Jong [5] present a vision of walking through a 3D visualization of software architecture with VRML. Yet the potential of VR and game engines have not been realized in software engineering (SE) tools, and their practicality with off-the-shelf VR hardware remains insufficiently explored.

In prior work, we developed a non-VR (PC display) FlyThruCode (FTC) 3D fly-through approach for navigating software structures [6]. This paper contributes VR-FlyThruCode (VR-FTC), a new VR approach for visualizing, navigating, and conveying program code information interactively in a VR environment to support exploratory, analytical, and descriptive cognitive processes [7]. A prototype demonstrates its viability, with a technical and empirical study investigating effectiveness, efficiency, and user experience (UX) factors for SE tasks and SE education.

The paper is organized as follows: the next section discusses related work; Section III then describes the solution approach. Section IV provides realization details. Section V evaluates the solution, which is followed by a conclusion.

II. RELATED WORK

Teyseyre and Campo [8] give an overview and survey of 3D software visualization tools across the various software engineering areas. Software Galaxies [9] provides a web-based visualization of dependencies among popular package managers and supports flying. Every star represents a package that is clustered by dependencies. CodeCity [10] is a 3D software visualization approach based on a city metaphor and implemented in SmallTalk on the Moose reengineering framework. Buildings represent classes, districts represent packages, and visible properties depict selected metrics, improving task correctness but slowing task completion time [11]. Rilling and Mudur [12] use a metaball metaphor (organic-like n-dimensional objects) combined with dynamic analysis of program execution. X3D-UML [13] provides 3D support with UML in planes such that classes are grouped in planes based on the package or hierarchical state machine diagrams. A case study of a 3D UML tool using Google SketchUp showed that a 3D perspective improved model comprehension and was found to be intuitive [14]. Langelier et al. [15] supports the visualization of metrics (e.g., coupling, test coverage).

As to VR, Imsovision [16] visualizes object-oriented software in VR using electromagnetic sensors attached to shutter glasses and a wand for interaction. ExplorViz [17] is a Javascript-based web application that uses WebVR to support VR exploration of 3D software cities using Oculus Rift together with Microsoft Kinect for gesture recognition.

In contrast to the above work, the VR-FTC approach leverages game engine capabilities to support an immersive VR software visualization environment multiple dynamically-switchable (customizable) metaphors; uses one VR system and controller set (not requiring gesture training) for interaction and navigation; supports tagging, searching, and filtering; and integrates information screens within the VR landscape that dynamically invoke external SE tools.

III. SOLUTION APPROACH

VR-FTC uses VR flythrough for visualizing program code structure or architecture. This inherent 3D application domain view visualization [8] arranges customizable symbols in 3D space to enable users to navigate through an alternative perspective on these often hidden structures. For example, certain information typically not readily accessible is visualized, such as the relative size of classes (not typically visible until multiple files are opened or a UML class diagram is created), the relative size of packages to one another, and the dependencies between classes and packages.

A. Principles

The principles (P:) of the VR solution approach include:

P:Multiple 3D visual metaphors: Analogous to the concept of skins, it models and supports tailoring and switching between multiple code structure visualization metaphors. While our initial implementation focused on modeling and visualizing object-oriented packages, classes, and their relationships, the approach is extensible for other programming languages. Initially, two metaphors are provided "out-of-the-box" while custom mappings to other object types are supported. In the universe metaphor, each planet represents a class with its size based on the number of methods, and solar systems represent a package. Any metric can be used to map to any visual object property (like color). Multiple packages are shown by layer solar systems over one another. In the terrestrial metaphor, buildings can represent classes, building height can represent the number of methods, and glass bubbles can group classes into packages. Relationships are modeled visually as colored pipes.

P:Group metaphor: elements (classes) are grouped and delineated in a way appropriate for that language (packages for Java) and metaphor. For instance, the terrestrial metaphor uses either a glass bubble over a city or a circle of trees at the city border, and the universe metaphor uses solar systems.

P:Connection metaphor: elements (classes) are connected in a way appropriate for that metaphor. For our two metaphors, we chose colored light beams, which often are used to portray networks on a geological background.

P:Fly-in theater screens: analogous to drive-in movie theaters common in the United States, here multiple various relatively large 2D screens are placed directly within the scene and provide menu settings and source-code and SE tool supplied information with which developers are familiar. While our original 2D FTC utilized a cockpit and heads-up display (HUD) metaphor principle simulating semi-transparent glass for providing multiple screens with context-specific information, in VR this turned out to be too close to the eyes and cumbersome for menu interaction.

The screens presented are:

- *Tags*: Setting, searching, or filtering automatic (via patterns) or manual persistent annotations/tags.
- *Source Code*: code is shown in scrollable form.
- *UML*: dynamically generated 2D diagrams.
- *Metrics*: code metrics are displayed textually due to the large number of possible metrics that may be of interest to the user; displaying a large amount of metric information visually may be disconcerting. Customization enables metrics of interest to be utilized in a metaphor (e.g., colors, object height can relate to number of class methods, font colors can indicated a threshold is exceeded).
- *Filtering*: shows elements that match selectors.
- *Project*: change metaphors, load, or import a project.

P:Flythrough navigation: 3D navigation (motion) is provided by moving the camera in space based on controller or motion sensor input. The scenery, however, remains anchored in the scene, allowing users to remember places via their geolocation relative to other elements.

B. Process

The approach process consists of:

1) *Modeling:* modeling generic program code structures, metrics, and artifacts as well as visual objects.

2) *Mapping:* mapping a model to a visual object metaphor.

3) *Extraction:* extracting a given project's structure (via source code import and parsing) and metrics.

4) *Visualization:* visualizing a given model instance within a metaphor.

5) *Navigation:* supporting navigation through the model instance (via camera movement based on user interaction).

IV. Implementation

The Unity engine was utilized for 3D visualization due to its multi-platform support, VR integration, and popularity, and for VR hardware both HTC Vive, a room scale VR set with a head-mounted display and two wireless handheld controllers tracked using two 'Lighthouse' base stations, and Google Cardboard with Android smartphones was used.

A. Architecture

Figure 1 shows the architecture. Assets are used by the Unity engine and consist of Animations, Fonts, Imported Assets (like a ComboBox), Materials (like colors and reflective textures), Media (like textures), 3D Models, Prefabs (prefabricated), Shaders (for shading of text in 3D), VR SDKs, and Scripts. While Google Cardboard does not use SteamVR, for the HTC Vive both are used because we needed a reticle. Scripts consist of Basic Scripts like user interface (UI) helpers, Logic Scripts that import, parse, and load project data structures, and Controllers that react to user interaction. Logic Scripts read Configuration data about Stored Projects and the Plugin System (input in XML about how to parse source code and invocation commands). Logic Scripts can then call Tools consisting of General and Language-specific Tools. General Tools currently consist of BaseX, Graphviz, PlantUML, and Graph Layout - our own version of the KK layout algorithm [18] for positioning objects. Java-specific tools are srcML, Campwood SourceMonitor, Java Transformer (invokes Groovy scripts), and Dependency Finder. Our Plugin system enables additional tools and applications to be easily integrated.

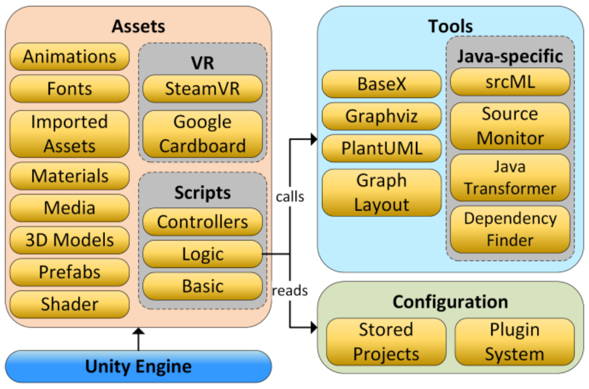


Figure 1. VR-FlyThruCode software architecture.

B. *Information Extraction*

For extracting existing code structure information into our model, srcML [19] is used to convert source code into XML that is then stored in the XML database BaseX, Campwood SourceMonitor and DependencyFinder are used to extract code metrics and dependency data, and plugins with Groovy scripts and a configuration are used to integrate the various tools.

C. *Project structure*

For an imported project the following files are created:

- *metrics_{date}.xml*: metrics obtained from SourceMonitor and DependencyFinder are grouped by project, packages, and classes.
- *source_{date}.xml*: holds all classes in XML
- *structure_{date}.xml*: contains the project structure and dependencies utilizing the DependencyFinder.
- *swexplorer-annotations.xml*: contains user-based annotations (tags) with color, flag, and text including both manual and automatic (pattern matching) tags.
- *swexplorer-metrics-config.xml*: contains thresholds for metrics.
- *swexplorer-records.xml*: contains a record of each import of the same project done at different times with a reference to the various XML files such as source and structure for that import. This permits changing the model to different timepoints as a project evolves.

D. *Virtual reality*

1) VR Interaction: On the HTC Vive the touchpad on the left controller controls altitude (up, down) and the one on the right hand the direction (left, right, forward, backward), which realizes *P:Flythrough navigation* by moving the camera position. The controllers are shown in the scenery when they are within the view field, as shown in the universe (Figure 2) and city metaphor (Figure 3). A virtual laser pointer was created for selecting objects, as was a virtual keyboard (Figure 4) to support text input for searching, filtering, and tagging. Menus and screens showing source code, code metrics, UML, tags, filtering, and project data were placed in the scene, in accordance with *P:Fly-in theater screens* (Figure 5 and Figure 6). To highlight a selected object, we utilized a 3D pointer in the form of a rotating upside-down pyramid (see Figure 5). This was needed because, once an object is selected, after navigating to a screen or menu one may turn around and lose track of where the object was, especially if the object was small relative to its surrounding objects.

2) Metaphor realization: To support *P:Multiple 3D visual metaphors*, a universe (Figure 6) and a city metaphor were chosen. Figure 3 shows the city metaphor where buildings represent classes with a label at the top and the number of stories represent the number of methods in that class. For the universe, *P:Group metaphor* was implemented as a solar system, whereas in the city metaphor a glass bubble. For the *P:Connection metaphor*, in both metaphors colored light beams were used to show dependencies between classes or packages (Figure 3 and Figure 7). To allow the user to remember objects, tagging is supported, which allows any label to be placed on an object (e.g., the Player Tag in Figure 7).

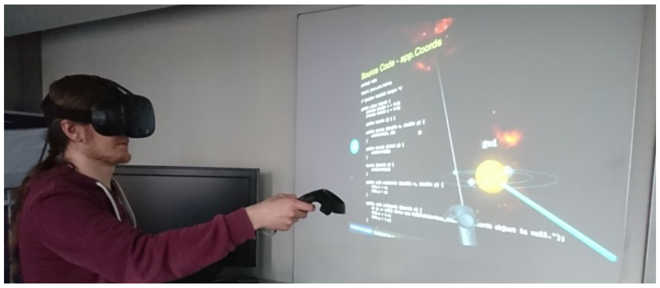


Figure 2. Subject with a Vive headset and controller (visible in a scene making a selection). Note: screen is mirroring what the subject is viewing.

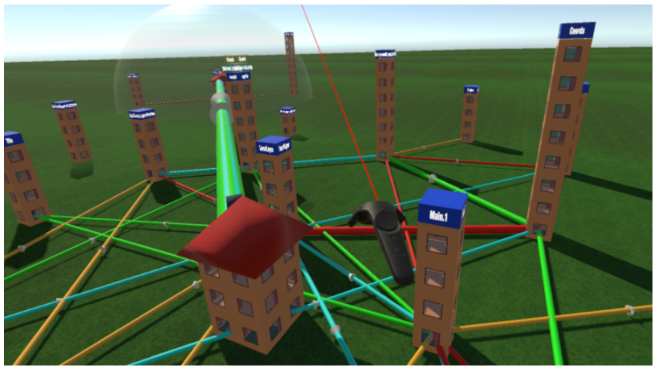


Figure 3. Vive controller visible in city metaphor, buildings as classes.

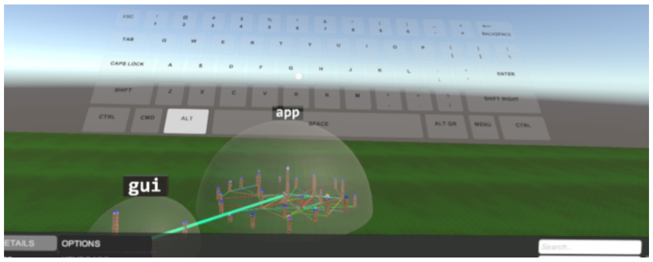
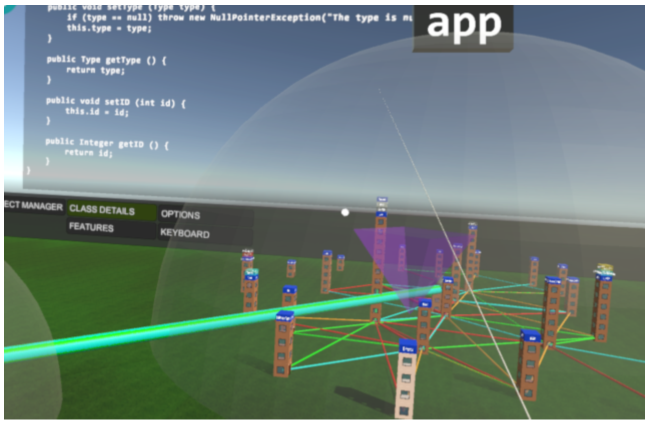


Figure 4. Virtual keyboard button pressed with reticle via controller.



```
    public void setType (Type type) {
        if (type == null) throw new NullPointerException("The type is nu
        this.type = type;
    }

    public Type getType () {
        return type;
    }

    public void setID (int id) {
        this.id = id;
    }

    public Integer getID () {
        return id;
    }
}
```

Figure 5. City glass bubble with pyramid pointer and program code screen.

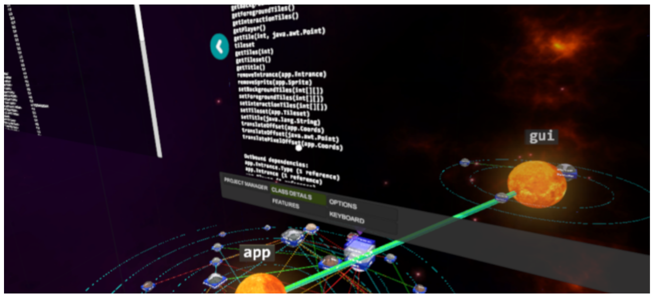


Figure 6. Universe solar systems with pyramid pointer and menu screen.

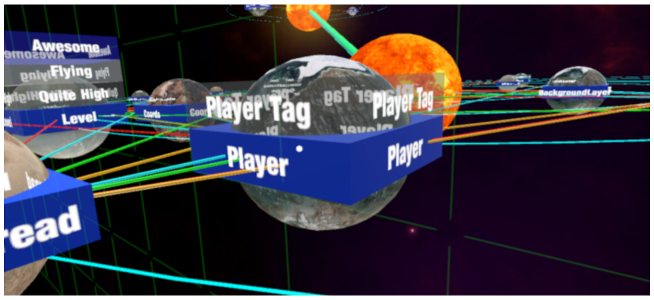


Figure 7. Universe showing tagged class.

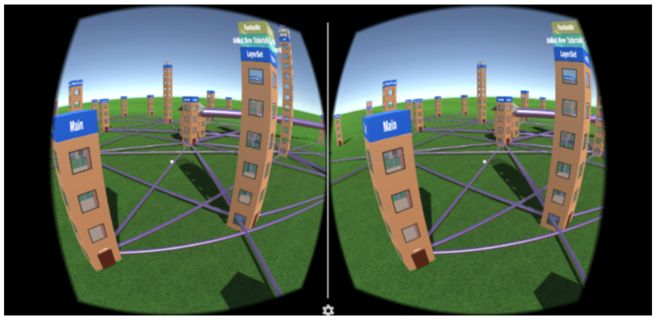


Figure 8. Cardboard stereoscopic visualization shown in the Unity Editor.

3) Google Cardboard: To support VR use with inexpensive hardware, VR-FTC was integrated with the Android SDK and Google Cardboard SDK for Unity and run with a Google Cardboard visor (see Figure 8) and utilize motion sensors within the smartphone. Using the gaze point, the reticle changes from a small closed circle to an open circle to indicate the object can be selected. The external tools were not ported.

V. EVALUATION

A technical evaluation focused on assessing the application's viability on current VR hardware options. Having the same FTC application as a constant in VR-and non-VR modes, the empirical evaluation assessed differences in application usability, effectiveness, and efficiency.

The evaluation utilized an HTC Vive with a 2160×1200 447 PPI resolution, Unity 5.3.5f1 PE, SteamVR 1479163853. The desktop PC had a 4GHz i7-6700K, 32GB RAM, SSD, NVIDIA GeForce GTX980Ti with 6GB GDDR5, Win7 Pro x64 SP1. The notebook was a MSI GS60 2.5GHz i7-4710HQ, 16GB RAM, NVIDIA GeForce GTX870M with 3GB GDDR5, SSD, Win10 Home x64, which did not meet Vive's minimum requirements but allows us to determine if a notebook (popular among software developers) would suffice for our VR application.

A. Technical Evaluation

1) Resource usage: RAM was allocated for a 64-bit implementation was 220MB (with no project), 250MB (project with 27 classes), and 620MB (project with 95 classes). On the notebook, graphics card load was 80% without a project and went to 90% with a loaded project (for the PC 20%). We determined the CPU was the bottleneck, with load on the PC for a large project almost always at 100%. We believe that scripts attached to each visible class invoke their update method for each frame, and plan to optimize this in future work.

2) Google Cardboard: an inexpensive VR alternative (when one already has a smartphone), a Sony Xperia Z2 2.3GHz Quad-core 5" 1080p 423 PPI display and a Samsung Galaxy A5 1.2GHz Quad-core 5" 720p 300 PPI display were used running Android 6.0.1 with Google Cardboard. On the A5 with 720p, the pixels became discernable, and since Google Cardboard lenses magnify the screen, we believe it to be related to the lower PPI. We also observed significant drops in the frame rate. Future work will analyze this further to determine if optimizations can address this or if current smartphone performance is inadequate for this type of application.

3) Frame rate: to determine the performance impact of each metaphor and if a notebook would be sufficient, the Saxon XSLT 2.0 and XQuery processor consisting of 300K lines of code and 1635 classes was loaded and the frames per second (FPS) measured via a custom script.

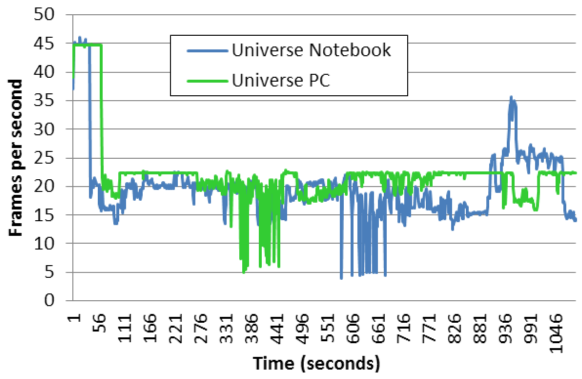


Figure 9. Universe metaphor frame rate over time on notebook and PC.

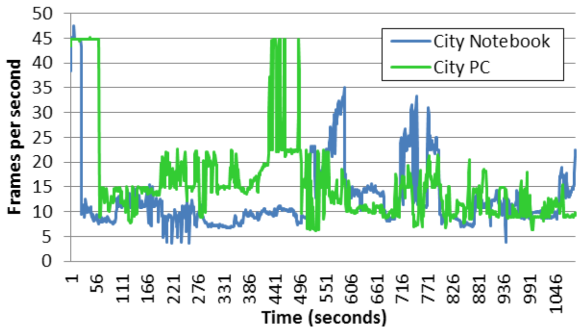


Figure 10. City metaphor frame rate over time on notebook and PC.

Figure 9 and Figure 10 show that the notebook mostly exhibited lower FPS rates than the PC, and that the city metaphor lowered the FPS rate. This is also shown by the average FPS rates for universe (notebook=20.0; PC=22.2) and city (notebook=12.7; PC=16.0). Below 15 FPS is not tolerable (early silent films had 16-20). An initial analysis found the dynamic UML generator - run in a separate process - as a main cause, and this will be addressed in future work. The universe ran better than city, since city included multiple shadows, reflections from the glass bubble, and a terrain. Higher FPS occurred when flying to an outer package such that far fewer objects were in view.

B. *Empirical Evaluation*

Since non-VR FTC was previously evaluated against both IDE (Eclipse) and UML (Enterprise Architect) tools [6], here we focus on comparing VR and non-VR modes keeping the FTC application a constant. For empirical evaluation of SE task suitability, because of the Google Cardboard's lack of controllers and it's the frame rate issue, only the Vive was used. Our hypotheses are (1) that VR mode is on par with non-VR in effectiveness and efficiency for SE code structure analysis tasks and education, and (2) VR mode offers an immersive and UX quality absent in non-VR.

Resource-constraints such as having only one Vive and the time-intensive 2-on-1 supervision of the experiment with a single subject at a time limited our sample size. A convenience sample of 10 computer science students of various academic semesters (1; 3-4; 6-9 grouped respectively as beginner, intermediate, and advanced) participated and self-rated their programming and UML competency (Figure 11). Object-orientation (OO) is taught in the second and UML in the fourth semester. The one first semester student had work experience in the software industry and thus knew OO and UML. Each received a short tutorial on non-VR FTC (three had prior experience). Project A consisted of 2 packages, 27 classes, and 170 methods, while Project B had 5 packages, 95 classes, and 800 methods.

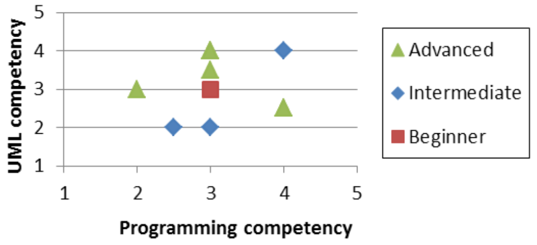


Figure 11. Participant UML/programming self-rating by semester level.

In non-VR mode, project A was loaded in the universe and thereafter the city metaphor, and likewise with B, and the same sequence repeated for VR mode. 8 questions were asked per case dealing with program code structural comprehension requiring navigation (not the same set each time), resulting in 64 questions (see Figure 12); 5 additional general questions followed giving 69 in total. So that the VR glasses need not be removed, and in order not to skew the task durations in non-VR mode, questions were asked and answered verbally and noted by a supervisor.

1) How many connections/dependencies does class X have within the package Foo?
2) How many connections/dependencies does class Y have within the package Bar?
3) Add a tag to the class X
4) Which package is the largest/smallest?
5) How many connections/dependencies does package Foo have?
6) How many connections/dependencies does package Bar have?
7) How many variables are declared in the class Y in package Foo?
8) Which classes are directly connected with the class Y?
9) Name all classes on the shortest path from A to B.
10) How many overloaded functions does the class Z have in package Bar?
11) In what package did you set your tag?

Figure 12. Sample timed task questions and requests.

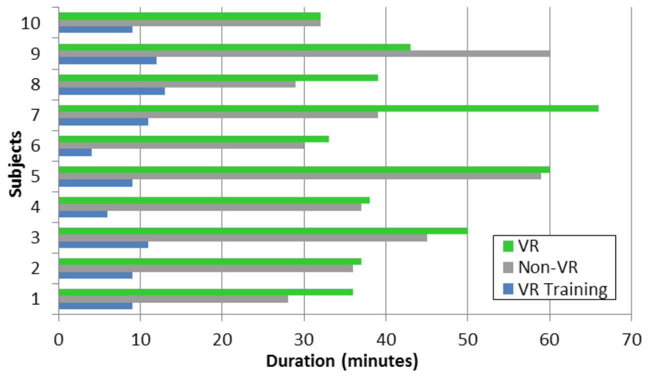


Figure 13. Sum of task durations per subject for VR and non-VR modes.

As to efficiency, on average 92.5 min were needed for the 64 questions, 43.4 in VR mode vs. 39.5 min in non-VR (10% difference), while VR training took 9.4 min. 0shows the sum of the task durations for each mode per subject, whereby subjects 8-10 had prior FTC familiarity. While VR mode was 10% slower, this was their first experience using VR. In addition, in non-VR mode the HUD is instantly available and screens can be switched, while in VR mode navigation to a screen is required. In our opinion, more VR practice would reduce this difference further.

With regard to effectiveness, given 32 questions in each mode across 10 subjects, in non-VR 300 (94%) and in VR 296 (93%) were answered correctly. Considering fatigue, we consider the effectiveness equivalent for both modes.

Subjects considered both FTC application modes suitable for these SE tasks. Comments included liking how information was visually displayed, its closeness to a reality, its clear arrangement, and that head movement could be used for exploring (which non-VR cannot provide). Subjects felt no differently after using non-VR, whereas after VR the feeling was described as impressive for seven of the ten subjects. The other three subjects reported VR sickness symptoms, a type of visually-induced motion sickness exhibiting disorientation. We plan to address the VR sickness in future work, e.g., by increasing the frame rate via optimizations and reducing the speed of camera movement. .

Our empirical hypotheses were confirmed by our results and the feedback from participants. One threat to validity is the order effect of application usage in that non-VR followed VR. Thus, non-VR times include the overhead for gaining familiarity with the application concepts, and VR mode did not have this overhead. However, 2D monitor and mouse-centric interaction was a pre-existing competency, while VR display and navigation was a new interaction paradigm for all subjects. Furthermore, subjects 8, 9, and 10 had prior familiarity with the non-VR FTC via a prior experiment, yet their task duration times did not exhibit any clear trend that prior familiarity sped up the non-VR task durations. Furthermore, the 1% difference in correctness could be attributed to fatigue since VR was done in the second hour. A further threat to validity is that the positive experience is possibly a novelty effect - VR veterans would be needed to be included to assess this factor. For better external validity, the sample size should be larger and more diverse to include professionals. However, the results can be viewed as indicative and the approach as promising if we can address the VR sickness.

VI. Conclusion

As virtual reality devices become more commonplace, the application of VR to SE environments for SE tasks and education will likely gain acceptance. This paper contributes a VR flythrough software structure visualization approach called VR-FTC that immerses users into multiple and customizable virtual reality metaphors for visualizing, navigating, and conveying program code information interactively to support exploratory, analytical, and descriptive cognitive processes. A game engine creates an immersive VR software visualization environment using only one VR system and set of VR controllers for VR interaction and navigation, supports tagging, searching, and filtering, integrates dynamic information screens within the VR landscape that dynamically invoke external SE tools, and supports an inexpensive VR option (Google Cardboard).

A prototype demonstrated its viability even for notebook configurations. The technical evaluation showed suitable resource usage but pointed out frame rate issues, especially for Google Cadboard. The empirical study determined that VR mode was 10% slower on average for tasks for untrained VR users with no significant difference in correctness. UX impressions were that both modes were suitable, and 70% were very impressed with the VR feeling, while 30% experienced VR sickness symptoms. That VR mode user performance was equivalent to the very common non-VR interaction in conjunction with the reported emotional effects indicates VR mode can be an appealing option for SE tools. Also, our VR training took less than 10 minutes, so training investments for VR should be minimal.

Future work includes further analysis and optimizations to address frame rate issues, visualization improvements, addressing the VR sickness symptoms, and a more comprehensive empirical study including professionals.

Acknowledgment

The authors would like to thank Dominik Bergen, Alexandre Matic, Lisa Philipp, and Camil Pogolski for their assistance with various aspects of the design, implementation, and evaluation.

References

[1] C. Metz, *Google Is 2 Billion Lines of Code—And It's All in One Place*. [retrieved: Jan, 2017]. Available from: http://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/

[2] G. Booch, "The complexity of programming models," Keynote talk at AOSD 2005, Chicago, IL, Mar. 14-18, 2005.

[3] F. P. Brooks, Jr., The Mythical Man-Month. Boston, MA: Addison-Wesley Longman Publ. Co., Inc., 1995.

[4] C. F. Kemerer and M. C. Paulk, "The impact of design and code reviews on software quality: An empirical study based on PSP data," IEEE Trans. on Software Engineering, vol. 35, no. 4, 2009, pp. 534-550.

[5] L. Feijs and R. De Jong, "3D visualization of software architectures," Comm. of the ACM, 41(12), 1998, pp. 73-78.

[6] R. Oberhauser, C. Silfang, and C. Lecon. "Code structure visualization using 3D-flythrough," 11th Int'l Conf. Computer Science & Educ. (ICCSE 2016). IEEE, 2016, pp. 365-370.

[7] D. M. Butler et al., "Visualization reference models," in Proc. Visualization '93 Conf., IEEE CS Press, 1993, pp. 337–342.

[8] A. R. Teyseyre and M. R. Campo, "An overview of 3D software visualization," Visualization and Computer Graphics, IEEE Trans. on, vol. 15, no. 1, 2009, pp. 87-105.

[9] A. Kashcha. *Software Galaxies* [retrieved: Jan, 2017]. Available: http://github.com/anvaka/pm/

[10] R. Wettel and M. Lanza, "Program comprehension through software habitability," in Proc. 15th IEEE Int'l Conf. on Program Comprehension, IEEE CS, 2007, pp. 231–240.

[11] R. Wettel et al., "Software systems as cities: A controlled experiment," in Proc. of the 33rd Int'l Conf. on Software Engineering, ACM, 2011, pp. 551-560.

[12] J. Rilling and S. P. Mudur, "On the use of metaballs to visually map source code structures and analysis results onto 3d space," in Proc.. 9th Work. Conf. on Reverse Engineering, IEEE, 2002, pp. 299-308.

[13] P. M. McIntosh, "X3D-UML: user-centred design, implementation and evaluation of 3D UML using X3D," Ph.D. dissertation, RMIT University, 2009.

[14] A. Krolovitsch and L. Nilsson, "3D Visualization for Model Comprehension: A Case Study Conducted at Ericsson AB," University of Gothenburg, Sweden, 2009.

[15] G. Langelier et al., "Visualization-based analysis of quality for large-scale software systems," in Proc. 20th Int. Conf. on Automated Software Engineering, ACM, 2005, pp. 214-223.

[16] J. I. Maletic, J. Leigh, and A. Marcus, "Visualizing software in an immersive virtual reality environment," 23rd Intl. Conf. on Softw. Eng. (ICSE 2001) Vol. 1., IEEE, 2001, pp. 12-13.

[17] F. Fittkau, A. Krause, and W. Hasselbring, "Exploring software cities in virtual reality," IEEE 3rd Working Conference on Software Visualization (VISSOFT), IEEE, 2015, pp. 130-134.

[18] T. Kamada and S. Kawai, "An algorithm for drawing general undirected graphs," Information processing letters, 31(1), 1989, pp. 7-15.

[19] J. Maletic et al, "Source code files as structured documents," in Proc. 10th Int. Workshop on Program Comprehension, IEEE, 2002, pp. 289-292.