Code Structure Visualization Using 3D-Flythrough

Roy Oberhauser, Christian Silfang, and Carsten Lecon
Department of Computer Science
Aalen University
Aalen, Germany
{roy.oberhauser, christian.silfang, carsten.lecon}@hs-aalen.de

***Abstract*—As software structures grow, it becomes increasingly difficult for those new to any software source code to conceptualize the mostly invisible structure. Historically program structures have been challenging to visualize. This paper contributes a 3D flythrough approach called FlyThruCode for visualizing facets of program structure. A prototype demonstrates its viability, and an empirical study investigates its effectiveness, efficiency, and motivational factors, showing promise for motivating the exploration of larger software structures.**

***Index Terms*—Software engineering, engineering training, computer education, software visualization, program comprehension.**

I. Introduction

More program code is being produced by the software industry every minute. It has been estimated that well over a trillion lines of code (LOC) exist with 33bn added annually [1]. For instance, Google has 2bn LOC accessible by 25K developers [2]. Active open source projects double in size and number in ~14 months [3]. As more program code becomes available, additional programmers are often needed to maintain or evolve this (legacy) code and thus others, besides the original programmer, are required to comprehend its structure. Yet the ability of humans to digest (in terms of understanding) program code directly is limited, as can be seen in the relatively low code review reading rates of around 200 LOC/hour [4]. Assuming reading for comprehension is a factor faster than reviewing for defects, it remains a low rate. Thus, tooling support that can improve the efficiency and effectiveness of structural program comprehension will likely become increasingly relevant.

Brooks asserted that the invisibility of software is an essential difficulty of software construction because the reality of software is not embedded in space [5]. Common formats used for comprehension of code include text or the two-dimensional Unified Modeling Language (UML). [6] provided a vision of walking through a 3D visualization of software architecture using the virtual reality modeling language (VRML). Yet the potential of current graphical information representations and game engine capabilities, including further dimensions and the movement through space, is not as yet prevalent or sufficiently explored in software engineering. Furthermore, support for specific capabilities that can enhance comprehension, for instance personally annotating or tagging code lines (not file versions), is not widely adopted. While program comprehension will remain a cognitively challenging task, visualization capabilities could help. Confronting an unknown software project is cognitively challenging and can be intimidating, especially for beginners with little experience. Currently, students are typically taught to use tools such as Integrated Development Environment (IDE) or Unified Modeling Language (UML) tools to attempt to understand an unfamiliar project. However, it seems that this is not always sufficient. We observe a lack of utilization and adoption of 3D visualization techniques in the industry and academia. Our investigation focused on determining if: sufficient effectiveness and efficiency potential exists to justify further research into 3D visualization of code structures for program comprehension; leveraging current game engine capabilities and certain features removes possible adoption hindrances; and 3D flythrough of software structures is considered intuitive and motivational for users of different competency levels.

In light of these issues, this paper contributes a 3D flythrough visualization approach we call FlyThruCode that offers opportunities for grasping software program structures utilizing information visualization to support exploratory, analytical, and descriptive cognitive processes [7]. A prototype demonstrates the viability of the approach and an empirical study investigates its effectiveness, efficiency, and motivational factors for beginner to more advanced users.

The paper is organized as follows: the next section discusses related work; Section III describes the solution approach, followed by its implementation; and Section V evaluates the solution, followed by a conclusion.

II. Related Work

An overview and survey of 3D software visualization tools across the various software engineering areas is given by [8]. 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. [11] showed a significant increase in terms of task correctness and decrease in task completion time. [12] uses a metaball metaphor 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]. [15] supports the visualization of metrics. White coats [16] provides 3D web visualization of CVS repositories using VRML.

In contrast to the above work, FlyThruCode supports multiple metaphors and the dynamic switching between them, custom and automatic annotation/tagging, and the display of localized contextually-relevant data (code, metrics, UML) in a heads-up display, thereby intermixing 2D data while flying through the 3D space.

III. Solution Approach

The solution approach we call FlyThruCode uses 3D 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 hidden structures.

A. Solution Principles

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

P:Multiple 3D metaphors: The input for a model instantiation is an import of project source code. One of the first issues faced in visualization is how to best model and visualize the program code structures. Because of the lack of any standardization or norms in this area, and to support the spectrum of individual preferences, support is provided for modeling and switching between *multiple visualization metaphors*, analogous to the concept of skins. Our initial model focuses primarily on modeling and visualizing object-oriented packages, classes, and their relationships such as associations and dependencies. Initially we support two metaphors "out-of-the-box" to provide examples of skins, and custom mappings to other objects types are possible. In the *universe* metaphor, each planet represents a class with planet size based on the number of methods, and solar systems represent a package. 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 pipes by default.

P:Cockpit: analogous to an airplane cockpit, this provides information to the user on the border of the screen, and has input fields for searching for a class or method or navigating directly to a class. Buttons can be depressed to indicate preferences. A minimap on the upper right of the screen provides a high-level overview of the entire landscape and one's relative location in a small area.

P:Heads-Up Display (HUD): This provides a transparent class on the screen with additional context-specific information. The type of information displayed can be changed via left/right arrows on the screen edges. The transparency level can be adjusted in the cockpit to provide a less opaque background if desired (e.g., to view code better). Various HUD screens are provided:

1) *Tags*: Automatic and manual persistent annotations/tags
2) *Source Code*: code is shown in scrollable form.
3) *UML:* diagrams are dynamically generated in 2D.
4) *Metrics*: metrics are shown as text because of the density of the number of possible metrics (any of which may be of interest to the user), and graphically displaying a large amount of metric information may be visually disconcerting. Rather, customization supports using metrics to display objects in the given metaphor, such as colors, object height (our default metric of building height is based on number of methods), or font colors to differentiate metrics where a given threshold is exceeded.
5) *Project Management*: can manage the metaphor, load a project record, or import a new project.
6) *Filtering*: this makes, for instance via selectors, only certain packages of interest visible and hides the rest.

P:Flythrough navigation: both mouse and keyboard support for 3D navigation (motion) in all directions is provided, as is autopilot or lockon to navigate to a specific class.

P:Intermixing 3D/2D: support for dynamically generated 2D UML is integrated in the 3D environment, enabling the usage of this standard notation to support the understanding of one particular area of interest.

B. Solution Process

The process consists of 1) *modeling* program code project constructs, structures, and artifacts as well as visual objects 2) *mapping* these to a metaphor of visual objects, 3) *extraction* via tools of a concrete project's structure (via source code import and parsing) and metrics, 4) *visualization* of the model with alternative metaphors, and 5) supporting *navigation* through the model in 3 dimensional space (simulating movement by moving the camera based on user interaction).

IV. Implementation of the Approach

We chose to utilize a game engine for the visualization. Unity was selected for the 3D rendering due primarily to its multi-platform support and popularity.

A. Architecture

Fig. 1 shows the architecture used for the implementation. 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, Shaders (for shading of text in 3D), and Scripts. 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 Applications consisting of General and Java Applications. General Applications currently consist of BaseX, Graph Layout consists of our own version of the KK layout algorithm for positioning objects, Graphviz, PlantUML, and integration with our other software engineering services (SERE). Java

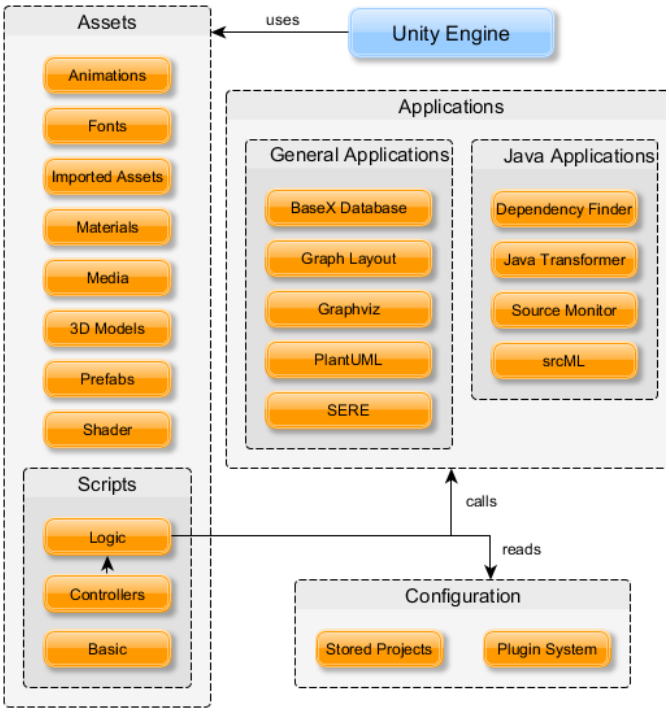

Fig. 2. Architecture of the FlyThruCode Implementation.

Applications consist of Dependency Finder, Java Transformer that invokes Groovy scripts, Campwood SourceMonitor, and srcML. Via the designed Plugin system, additional tools and applications can be easily integrated.

B. Extraction

The existing software structures must be imported and converted to the expected model. To support the greatest amount of interoperability with various existing software development tools, XML was selected as the primary data format. BaseX [17] is used as an XML repository and XQuery used for queries. srcML [18] v.0.9.5 was selected to convert source code (such as Java) into XML documents and provides various code metrics. Campwood SourceMonitor v.3.5 is used because it creates code metrics across multiple programming languages. To determine dependencies such as coupling and inheritance, DependencyFinder was selected, which also provides data on code structure, dependencies, and metrics from binary Java. Furthermore, Groovy scripts were used for the integration of the various tools.

C. Imported project structure

The project structure consists of the following files:

- *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 DependencyFinder.
- *swexplorer-annotations.xml*: contains user-based annotations with color, flag, and text including manual tags placed by the user and automatic tags placed in this file that place the tags where matches occur.
- *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. Graphical User Interface (GUI)

To give an impression of the GUI, we provide some screenshots. In Fig. 2, the terrestrial metaphor can be seen, with labeled glass bubbles that delineate packages and contain buildings (classes), with pipes show associations. As to the cockpit, on the upper right the minimap box can be clicked and expanded and at the bottom right the fields for entering a search word or class is seen, ability to adjust text transparency, and the bottom right shows buttons that can be depressed to indicate what is of interest to the "pilot."

In Fig. 3, the HUD is activated and search results and filtering options presented. In Fig. 4, the HUD shows a collection of metrics from various tools in context, such as various ratios and counts (e.g., number of public or private inner classes, lines of code, etc.). Fig. 5 shows the screen for annotating a class, whereas automatic tags <auto-tag> in the configuration are automatically place annotations based on regular expression pattern matches. To dynamically generate UML diagrams in 2D at runtime, PlantUML and GraphViz are used with an example output shown in Fig. 6. Fig. 7 shows the universe metaphor.

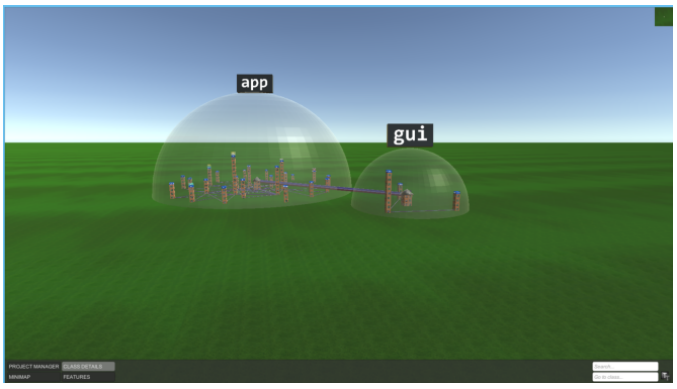

Fig. 2. Cockpit with terrestrial metaphor viewing packages.

Fig. 3. HUD showing project management screen.

Fig. 4. HUD metrics view after loading a record.

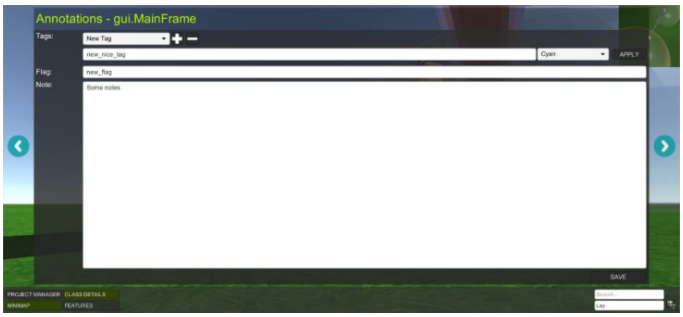

Fig. 5. HUD annotations view.

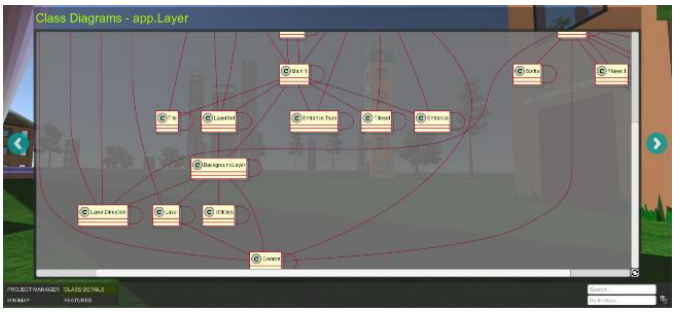

Fig. 6. HUD dynamic UML view.

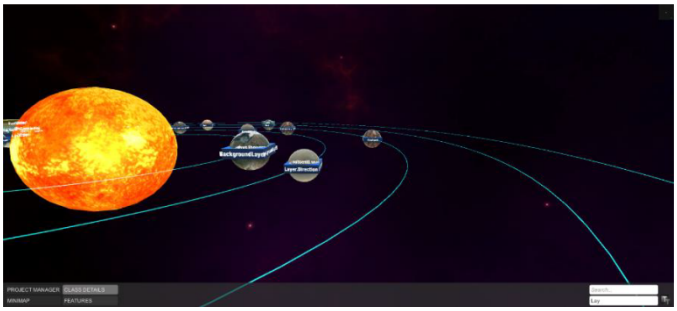

Fig. 7. Cockpit with the universe metaphor.

Separate screens (not shown due to lack of space) include: search, filtering (e.g., inclusion/exclusion of packages), and a minimap (right corner) for orientation. Minimum PC specifications are a CPU supporting Streaming SIMD Extensions 2 and DX9 GPU with Shader Model 2.0. Recommended is DX11 GPU and 1GB video RAM. Java 7 and .NET Framework 3.5 or higher are required.

V. Evaluation

3D visualization has the potential to overload users if it does not take advantage of its strengths and limit its weaknesses [8]. Due to the lack of widespread adoption of 3D tooling for program comprehension, we wished to evaluate if an inherent issue exists with 3D and if 3D flythrough is intuitive for navigating program structures. Thus, our experiment design focused primarily on determining if the FlyThruCode 3D visualization approach is beneficial (relative to the most common tools in use today, namely 2D IDE and UML tools) for comprehension effectiveness, efficiency, and motivation, and does its benefit depend on prior experience? We compared a common text-centric IDE (Eclipse), the 2D UML tool Sparx Systems' Enterprise Architect, and our 3D FlyThruCode approach. Due to resource and time constraints, in future work we plan to evaluate other 3D tools.

A. Research Questions

The following research questions (Q) were investigated:

Q1: Does a flythrough visual experience affect the effectiveness for analyzing certain aspects of software structures?

Q2: Does a flythrough visual experience affect the efficiency for analyzing certain aspects of software structures?

Q3: Does a flythrough visual experience aid in memory retention?

Q4: Does a flythrough visual experience provide a positive experience or motivational factor?

B. Experiment Setup and Process

A convenience sample of 14 computer science students of various academic semesters (2, 4, and 7-9) and prior tool experience (see Fig. 8) was used. Object-orientation is taught in semester 2 and UML in semester 4. A short tutorial on usage of the three tools was given to all participants beforehand; none had prior experience with FlyThruCode.

Two open source projects were used in all tools: Task1 had 720 lines of code (LOC) in one package and four classes, and Task2 had 4677 LOC in 6 packages and 38 classes.

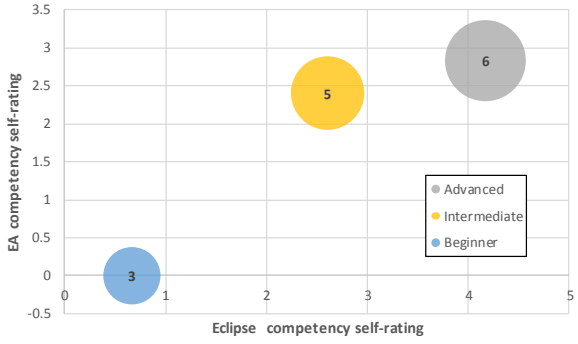

Fig. 8. Average tool competency self-ratings (7-point scale, 6 best).

Participants as a group were given live structural analysis tasks in a supervised PC pool environment at our university and filled out anonymous questionnaires, an extract of which is shown in Fig. 9. Memory recollection questions were answered

without tool access. Sessions were limited to two hours to prevent fatigue and permit working with intensity and motivation. We selected a seven-point Likert scale instead of the supposedly more reliable five-point scale [19] for cultural reasons. It is our impression that people in this particular region tend to have very high internal expectations and have difficulty giving best marks on any questionnaire (this has been supposedly corroborated by auto repair satisfaction surveys showing that our region was consistently rated lower even though the repairs were identical in quality to other regions).

Part 1: Classes
1) How many connections/dependencies does class X have?
2) Which classes are directly connected with the class Y?
3) Name all classes on the shortest path from A to B.

Part 2: Packages
4) How many connections/dependencies does package X have?
5) How many packages are contained in the project?
6) Which package is the largest (without counting)?
7) Which package has the most connections/dependencies?
8) Which package is smallest (without counting)?
9) Name all packages directly connected with package Y.

Part 3: Recollection
10) Diagram the entire project you kept in mind.
11) Name at least one class on the shortest path from class A to B
12) What was the largest package in the project?

Fig. 9. Sample questions from questionnaire (X/Y is project-specific).

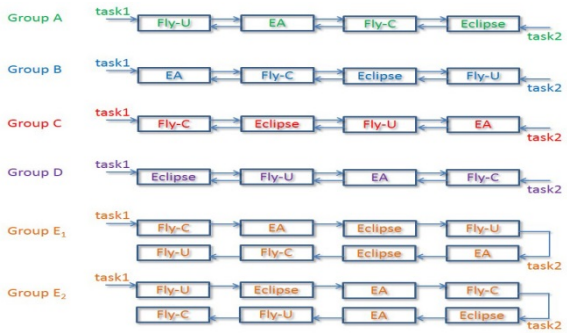

Fig. 10. Sequencing of tool use among groups.

Since questions were answered using tools consecutively by the same person, the order of tool use could affect their perception. To control this factor, we created separate groups with different tool sequence orders (see Fig. 10). Tasks were performed with the three tools consecutively, including universe ('Fly-U'), terrestrial (city) metaphor ('Fly-C').

C. Results

As shown in Fig. 11, a positive impact to effectiveness (Q1) can be observed for FlyThruCode. FlyThruCode improved class, package, and recollection correctness, and was on par with EA for class type questions. Furthermore, as to Q3, we observed better memory retention.

Efficiency (Q2) was evaluated by timing tasks via wall clock time (we had no access to monitoring software) with one participant (an advanced user) following Fig. 10's group E1 sequence and the novice E2. The completion times are shown in Fig. 11 and Fig. 12. We observed that both participants initially had difficulties, the novice E2 with Fly-U and E1 with Fly-C. However, after the "cold-start" effect, E2's performance was on par or faster than that of the advanced E1 for both Fly-C (at Task1 and 2) and Fly-U for the larger Task2. In the future, users will be primed before measuring. As to tool competency effects, while E1 remained faster with the familiar Eclipse and EA across both tasks, the unfamiliar tool Fly-C was within 15% on Task2. As to scaling effects with the larger Task2: E1 with Fly-C was 24% faster than with the simpler Task1, while with both Eclipse (133%) and EA (62%) was slower; and E2 was 42% (Fly-C) and 69% (Fly-U) faster and Eclipse/EA unchanged. Task2 shows metaphor choice can have an efficiency effect.

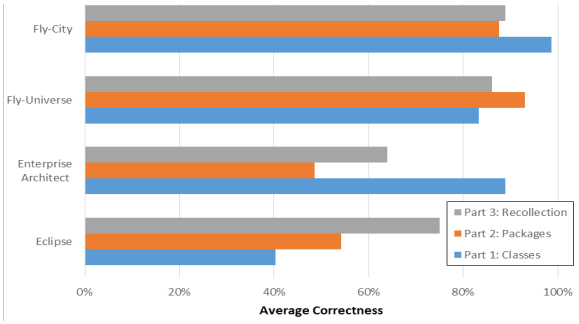

Fig. 11. Averages correct responses for question categories across tools.

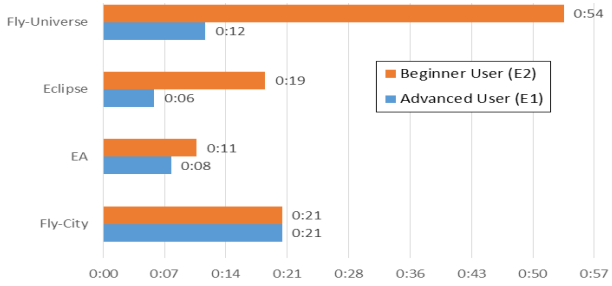

Fig. 12. Task1 completion times (minutes).

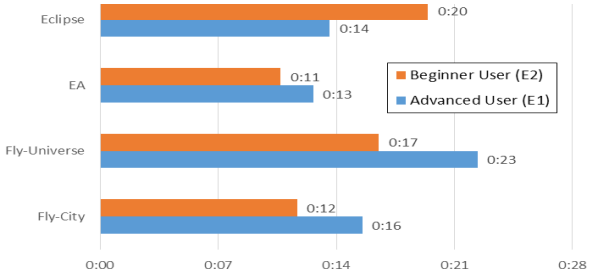

Fig. 13. Task2 completion times (minutes).

To determine if FlyThruCode has any motivational effect (Q4), Fig. 14 compares average responses on suitability, usability, experience, and feeling for each tool on a 7-point Likert scale (6 best). Note that the majority of participants were already familiar with EA and Eclipse. This positive affinity could positively influence the motivation of the students to explore (more complicated or larger) software structures. No participant indicated a cognitive or information overload in his or her usage of the 3D visualization. As to metaphors, we observed that the terrestrial was more intuitive than the 3D

universe. Yet Fig. 14 shows that the universe is preferred to the terrestrial metaphor. One observation is that a non-graphical tool like Eclipse is not well suited for structural analysis, as confirmed by the poorer evaluation result for Eclipse.

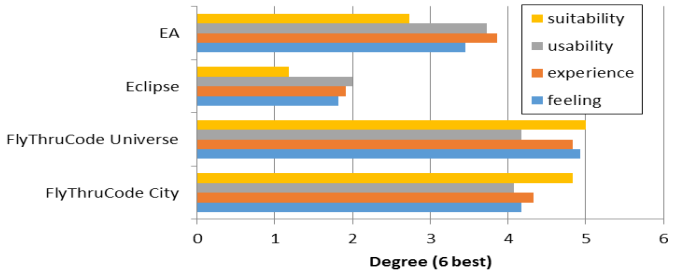

Fig. 14. Averages for personal experience responses on 7-point scale.

Various threats to validity include: the small convenience sample; relatively small project sizes; lack of control of various factors such as participant mental state and side-activities (smartphone interaction); various potential personal and environmental biases; participant self-ratings (versus independent competency assessments); lack of industry professionals; etc. However, we interpret these initial results as a positive indicator that the FlyThruCode approach has potential and should be investigated further in a larger controlled empirical study.

VI. Conclusion

This paper contributes a 3D flythrough visualization approach we call FlyThruCode that provides switchable customizable metaphors, localized contextually-relevant data, and intermixes 2D representations in the 3D landscape. The prototype implementation showed its technical viability and the evaluation provided various insights. These included that the approach is effective in improving the correctness of answers to certain structural questions that are more inclined to be visually determined, especially package level. FlyThruCode was on par with EA for class-level questions, since UML provides detailed info for classes. As to efficiency, for the larger project novice task performance using FlyThruCode equaled that of an advanced user, and while IDE and UML tools took longer, FlyThruCode time decreased. Further, we found that memory recollection was better. We also observed differences in user performance using different metaphors.

The motivational factor based on suitability, usability, experience, and feeling showed a preference for FlyThruCode across both metaphors. This motivation could provide a potential education benefit by encouraging software structure exploration, e.g., in academic settings.

Future work includes investigating differences in metaphors by varying visual objects and utilizing additional metaphors, metadata searching, improving and adding various features, and a comprehensive empirical study utilizing various 3D tools.

Acknowledgment

The authors would like to thank Dominik Bergen and Lisa Philipp for their assistance with the design and implementation.

References

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

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

[3] A. Deshpande and D. Riehle, "The total growth of open source," in *Proc. 4th Conf. on Open Source Systems (OSS 2008)*. vol. 275, Springer Verlag, 2008, pp. 197–209.

[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," *Software Engineering, IEEE Trans. on*, vol. 35, no. 4 2009, pp. 534-550.

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

[6] L. Feijs and R. De Jong, "3D visualization of software architectures," *Comm. of the ACM*, vol. 41, no. 12, 1998, pp. 73-78.

[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 Transactions on*, vol. 15, no. 1, (2009, pp. 87-105.

[9] A. Kashcha. *Software Galaxies* [Online]. 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] C. Mesnage and M. Lanza, "White coats: web-visualization of evolving software in 3d," in *Proc. 3rd IEEE Int. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2005)*, IEEE, 2005, pp. 1-6.

[17] C. Grün et al, "XQuery full text implementation in BaseX," Berlin Heidelberg: Springer, 2009.

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

[19] Likert, R. and Roslow, S. 1934 The Effects Upon the Reliability of Attitude Scales of Using Three, Five or Seven Alternatives, Working Paper, New York University.