Using Interactive Animations to Analyze Fine-grained Software Evolution

Carmen Armenti*, Michele Lanza*

*REVEAL @ Software Institute – USI, Lugano, Switzerland

Abstract—Understanding the evolution of software systems is a challenging task, due to their sheer size and complexity. Several visualization approaches have been presented over the years, using both 2D and 3D depictions. The vast majority of the approaches is geared towards understanding the "big picture", facilitating the comprehension of the overall evolution. However, when it comes to understanding the basic building blocks of software evolution, *i.e.,* the commits performed by the developers, visualization seems to fall short in favor of the de facto standard of textual diff views.

We present an approach, implemented in a custom tool, to depict commits using interactive animations which allow the viewer to inspect and dissect the intricacies of one or multiple commits. We illustrate our approach on a number of case studies, showing its potential benefits.

All videos are accessible on YouTube ▶.

*Index Terms*—software animation, program comprehension

I. INTRODUCTION

As Lehman stated half a century ago, software systems increase in complexity and scale as they are adapted to changing requirements [1]. Understanding software evolution is a complex mental activity that demands technical knowledge and abstraction skills [2]. Although software consists of *code*, the development process involves more than writing text [3]. Developers spend a significant amount of time reading and understanding *source code* [4]–[6], with the aim of building a *mental model* of the system they are working on [6]–[8]. While textual information is useful, it can lead to comprehension issues [9]. Text processing relies on less efficient cognitive processes [10], making software difficult to understand when viewed in its textual form [6], [11].

Software visualization aids in understanding software systems [12] by mapping software entities, behaviour, and evolution to visual metaphors, thereby reducing the perceived complexity and providing clearer insights and understanding [10], [13]. Over the years, various visualization approaches have been developed, including 2D (*e.g.,* [14]–[17]) and 3D depictions (*e.g.,* [18]–[20]), as well as immersive environments (*e.g.,* [21], [22]). Animated visualizations have also been utilized for educational purposes [11] and to visualize software-related aspects, such as program execution [11], [23] and the history of software repositories [24], [25].

Most visualizations represent coarse-grained structures of software systems, facilitating the comprehension of the overall evolution. However, when it comes to understanding the fundamental building blocks of software evolution – embodied by the commits performed by developers – visualization falls short, and textual diff views remain the *de facto* standard.

According to Weinberg *"The reading of programs is still the key to understand how we make programs"* [3]. Source code remains the natural and primary focus for understanding both a software system and its evolution [7], [26], [27]. During code reviews activities and when accepting pull requests, developers have no other choice than relying on (code) textual diffs, which are the primary source of information they seek, as code-diffs log the parts of the system that have been changed. In fact, code-diffs are the first point of reference developers look into when commit messages are unclear [9].

Many software engineering tasks require developers to understand the history and evolution of source code [6]. Studies have shown that resolving questions about the rationale behind source code changes (*i.e.,* who did what, why, and when) is time-consuming, and that methods providing the creation and evolutionary history of code can help developers understand the design rationale and the context of the changes [4], [28].

Understanding the evolution of software systems is a challenging problem due to the extensive, time-based, and heterogeneous nature of change sequences tracked by the commits of Version Control Systems (VCS). While the field of software visualization has developed powerful techniques to synthesize and simplify complex information, these methods typically face limitations such as *scale* and *modularization*. Also, they usually lack *temporal context* and struggle to represent *fine-grained* information. On the other hand, history exploration tools (*e.g.,* GitGraph (see online ⊕), CHRONOS [29], *Deep Intellisense* [30], etc.) and code reviews tools (*e.g.,* Gerrit (see online ⊕), GitHub interface, etc.) allow one to browse throughout the evolution (*i.e.,* time) of software systems and answer very specific questions engrained in the history of a piece of software (*e.g.,* what is the last revision in which this line of code was modified?), albeit exercising a higher cognitive load, since they mostly rely on textual information.

We present an approach, implemented in a tool, to inspect and dissect one or multiple commits through interactive animated visualizations, where the changes between two or more sequential commits, are represented in their dynamic nature. The interactivity of the animations we propose plays a major role: The approach allows to investigate and further inspect each file version pertaining to the commit at any moment in time during the animation. Basic features such as pausing, rewinding or replaying are provided. We present the tool supported by the approach, that we describe and illustrate through a number of case studies showing the potential benefits of our idea, and conclude with a reflection on the findings.

II. RELATED WORK

Understanding Commits. Commits are atomic units of change of software systems: By tracking information about the evolution of systems they are a valuable resource to support software evolution research. The problem of *understanding commits* has long been explored by researchers. Hattori *et al.* [31] proposed a size segmentation of commits based on the number of files they contain. Alali *et al.* [32] also aimed at characterising commits, including file and hunk diffs. Commits adhere to some rationale [33]: Tao *et al.* [34] found that the most important information needed to understand commits is logic. The logic of a piece of software, documented by commits, is conveyed by textual code-diffs. However, reading and understanding code-diffs remains a demanding task [35].

Software Evolution. Several studies leveraged visualization to understand system evolution. Revision Towers [25] represented by whom and to what extent a file has been changed. The RepoGrams tool provides a metric-based visualization model to understand the evolution of metrics during the history of a software project [36]. Researchers have used a two-dimensional matrix structure as the baseline for their visualization to (1) present metrics changes and (2) improve the software evolution comprehension [37]–[39]. Aghajani *et al.* proposed the *Code Time Machine*, a plugin which leverages visualization techniques to represent the history of files, which can be augmented with meta-information mined from the underlying versioning system [40]. Zimmermann *et al.* applied data mining techniques to version histories and proposed a tool (ROSE) to detect changes and build prediction model to suggest future changes to developers [41]. Kim *et al.* presented an interactive visual analytics system to represent a large Git graph in a scalable manner, to allow developers to effectively understand the context of development history through the interactive exploration of Git metadata [42].

Program Animations. In 1990, Stasko claimed that programs are difficult to understand when viewed in textual form: The meaning, methodology and purpose of a program are better explained by algorithm animations than program's textual representation. He proposed an approach to convey the meaning and purpose of programs, which is based on animated graphical views [11]. Animations were also employed in other disciplines other than software engineering [43]–[45].

Although algorithm animations have been used primarily for instructional purposes, some frameworks have been developed for industrial prototyping and simulation as well [46]. Animations, by dynamically displaying a process or a procedure, should be able to compensate for a student's scarce aptitude or skill to imagine motions [47]. Hoffler *et al.* conducted a meta-analysis of 26 primary studies, yielding 76 pairwise comparisons of dynamic and static visualizations, which revealed an overall advantage of instructional animations over static pictures [48]. Dynamic visualizations help in visualizing a process resulting in a reduction of cognitive load compared to a situation in which the process or the procedure has to be reconstructed from a series of static pictures [49].

The rationale for animations is based on the *cognitive load theory*, which provides a theoretical foundation to explain the superiority of animations over static graphics [50]–[52].

Visualization tools, such as Gource (see online 🌐), provide visualizations of the activity of a Git repository in an animated and aesthetic fashion, but the granularity depicted is coarse-grained and the resulting animation does not allow for interaction and inspection of the changes being displayed.

Learning with static visual representations requires information integration and inferential reasoning. On the other hand, animations are made up of consecutive visual representations (frames), which can serve to facilitate the understanding of dynamic systems and represent their changes over time [53]. Our work proposes an approach that leverages interactive and animated customized visualizations to represent fine-grained information about the evolution of software systems, providing a dynamic visual approach for exploring software histories.

III. SOFTWARE EVOLUTION COMPREHENSION WITH INTERACTIVE ANIMATIONS

Commits are usually depicted as text-based "diff representations," which neglects the actual dynamics of file transitions between versions. Represented as text, code changes in files are typically reviewed individually, despite files in the same commit changing together. It also has limitations with respect to *time*, as it squashes the time period during which changes occur, discarding important details [31], [54], [55] and hindering the comprehension of source code changes *rationale* [6]. The core idea of our approach is to represent the evolutive nature of file version changes through *animated visualizations*. Figure 1 shows the overall approach.

A. Animation Assembly

An animation is composed of 5 phases (④ in Figure 1): *pre-commit*, *prologue*, *corpus*, *epilogue*, and *post-commit*.

1) **Pre-Commit.** Each file involved in the commit is visualized, along with its name and a bar, whose length encodes the lines of code of the file version *before* the commit. Files removed during the commit are colored red. File added during the commit are not displayed at this stage. Modified files are color-coded: Light green if they grow because of the commit and light red if they shrink.

2) **Prologue.** This phase is dedicated to showing the appearance of newly added files throughout the commit, using for each one of them a fade-in effect. The newly added files are color-coded as green bars, and their length is kept to a minimum value until the next phase: Their growth is depicted during the main *corpus* stage.

3) **Corpus.** This is the main phase of the animation, and takes up most of the time. Files that grow or shrink as a result of the commit grow/shrink accordingly. Files that are changed but do not grow or shrink (i.e., lines are substituted with other lines) are shown as "activated": Their color changes following a fading color scheme that ranges from light grey (only one line is substituted) to deep blue (all lines are substituted).
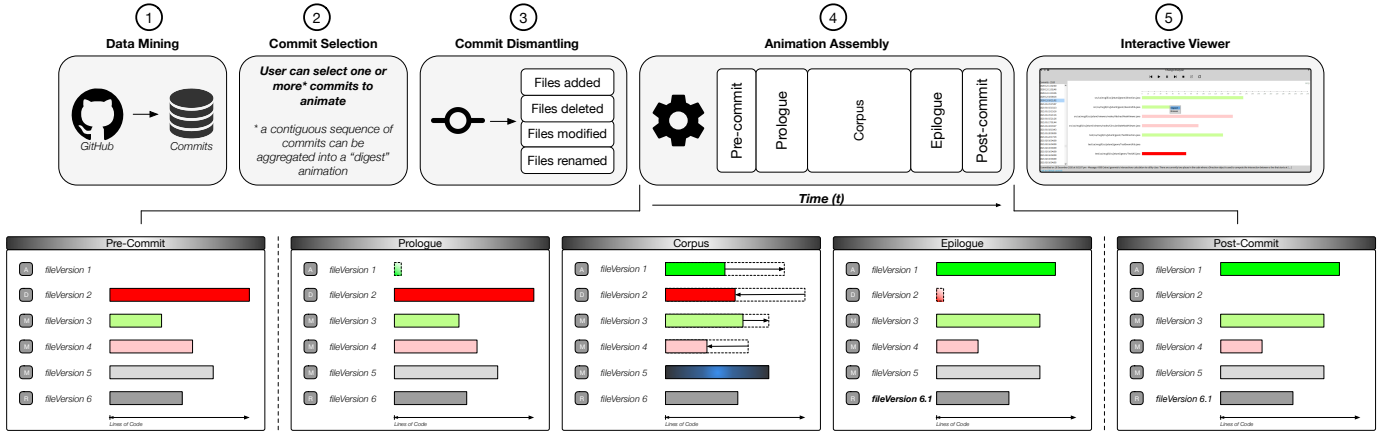
Fig. 1. Overview of the approach: The processing is depicted at the top, and the animation phases are detailed at the bottom.

4) **Epilogue.** This phase displays the disappearance of files deleted during the commit, each with a fade-out effect. This allows users to understand at one glance which files have been deleted. For files that have been renamed during the commit, this phase will fade out the old name and then fade in the new name.

5) **Post-Commit.** This phase depicts the status of the files participating to the commit after the commit has been performed. The boxes representing deleted files are no longer displayed; the length of each file bar represents the number of lines they contain after the commit, the colors continue to indicate the type of change the files have undergone as a result of the commit.

B. Commit Selection

Commits are our unit of observation and its constituent parts are the files that have been added, deleted, renamed, and modified. Our tool allows users to pick (a) one single commit or (b) a sequence of commits they want to animate. The **single-commit animation modality** allows to answer questions such as *"what was the impact of this commit on the system?"*, while observing how the single commit transposed file versions from their prior state to the one tracked by the commit under observation. The **commit digest modality** animates a sequence of commits and commits related to each other (*e.g.,* in the case of commits pertaining to a pull request) and makes possible to answer questions like: *"what happened yesterday?"*, *"how did the system evolve between the opening and closing of a GitHub issue?"*, or *"what is the summary of a pull request?"*. The animation modalities are described in Section IV.

C. Interactive Viewer

The user can set animation-specific parameters, such as frame rate and duration animation. Also, the Viewer allow to specify how to spread the whole duration over the phases of the animation. Experience values range between 5 and 10 seconds for the whole animation, most of which is taken up by the corpus.

The animation is fed to a tool we implemented (Figure 2), which allows users to select the commit or the sequence of commits they want to animate (A), and provides several means to interact with the animation (B), such as pausing, rewinding, stepping, etc. Moreover, it allows further inspections on file versions (D), (E), and one-click direct access to the GitHub diff viewer (D). Popups on each bar provide a summary on the type of change the file version underwent with its value before and after the commit (C). At the bottom of the Interactive Viewer (F), the commit message and the number of files touched by the commit selected are available, together with the link to GitHub Commits view. Additional features of the Viewer also include the possibility to export the animation as a video file or a sequence of stills (sets of pictures).

IV. INTERMEZZO

Data Mining. We mined the data of the 10 most popular software repositories from GitHub, ranked from the most starred to the least. The number of stars is the most visible indication of the popularity of open source projects, as starring repositories is a way to bookmark a specific project and to show appreciation in the GitHub ecosystem. Table I reports the list of projects collected and shows statistics such as the number of commits and stars as of May 28, 2024.

TABLE I
THE TOP 10 MOST POPULAR OPEN SOURCE GITHUB PROJECTS

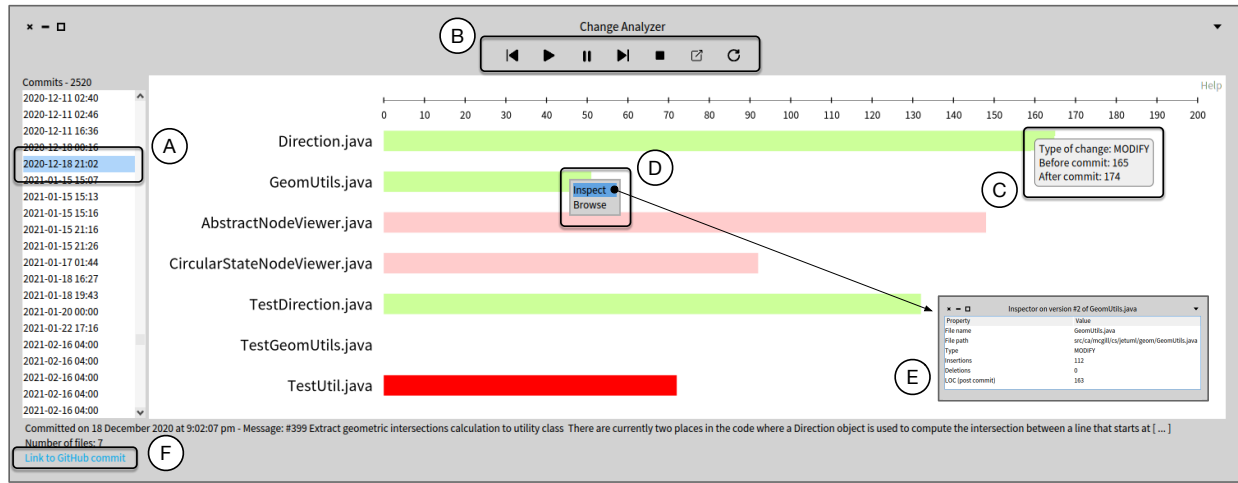| Name	#Commits	#Stars	Explored
freecodecamp	35,430	389,886	**Yes**
developer-roadmap	4,175	275,869	**Yes**
react	18,716	223,064	**Yes**
vue	3,590	207,037	**Yes**
tensorflow	163,529	182,509	No
linux	1,275,250	171,548	No
ohmyzsh	7,235	169,475	**Yes**
bootstrap	22,800	167,621	**Yes**
flutter	40,795	162,153	**Yes**
vscode	121,425	158,513	No

Fig. 2. Interactive Viewer on a commit of the JetUML repository, performed on December 18, 2020.

We did not explore 3 (out of 10) software repositories because their size made the data collection impractical for the purposes of the this study. The column "Explored" indicates whether the project was considered or not.

Example Repository. Using two examples from JetUML (see GitHub ○), we show how we utilize animations to understand commits. Our tool represents an animation as an interactive "movie" that can be paused, rewound, etc., where any entity is available for inspection at any moment.

Animations are difficult to illustrate on paper. Both examples and case studies animations are available as videos on YouTube (links provided in figure captions). For discussion purposes, we disassemble the animation into a series of frames, showing only main frames in a top-down order. The animation is composed of many more frames, whose number depends on its duration and frame rate. For example, a 5 seconds animation at 30 FPS will generate a total of 150 frames.

A. Commit Animation Example

Figure 3 depicts a commit versioned on January 22, 2015 ○, whose message is *"Code cleanups"*. The file versions are ordered alphabetically (as in the GitHub interface). In this commit, the animation mainly operates in the *corpus* phase ③-⑤, as all the files involved gets modified. While *Direction.java* increase in its size of 2 lines of code, *FormLayout.java* and *GraphFrame.java* undergo lines substitutions: In the first file 67 lines out of 100 are modified, in the latter 2 out of 133. The different magnitude of modifications determines the different activation color, which in the first instance is indigo and in the second case is a somewhat deeper shade of grey than the starting color. *PrintDialog.java* gets deleted: This action is depicted not only during the corpus but also in the epilogue phase ⑥, where it fades out. Pre- ① and post-commit ⑦ frames depicts the state of the file versions involved before and after the commit. There is no activity in the prologue phase ②, as no file was added.
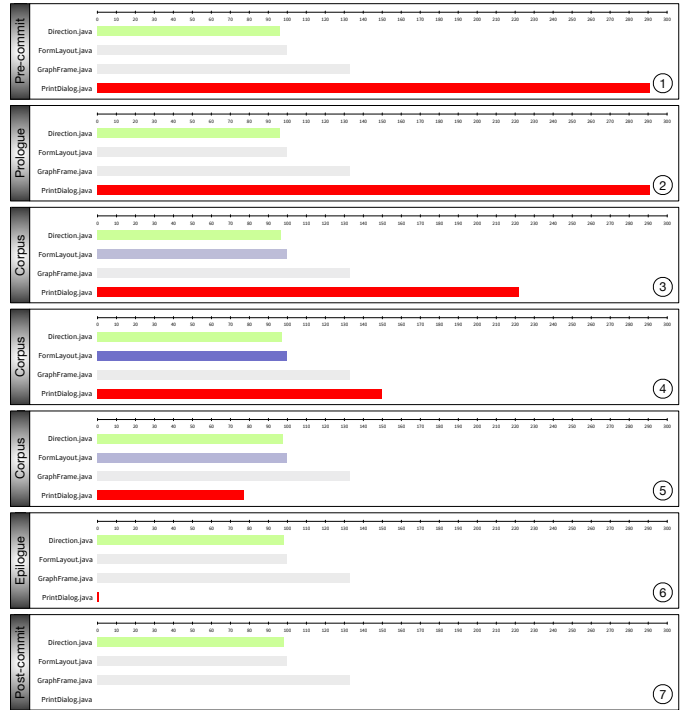

Fig. 3. Commit Animation Example: January 22, 2015 – JetUML. Watch on YouTube ▶

B. Commit Digest Example

Figure 4 depicts the commits versioned in 2017, on December 11 ○ and 12 ○.

The file versions are ordered alphabetically (inter single commit) and chronologically (intra commits).

Assuming there are 2 commits with 2 files versions each, the first 2 glyphs (boxes and labels) are ordered alphabetically between the two of them, and appear before the 2 files versions of the second commit.

In a commit digest, the transition between $n + 1$ states of parts of a system is represented (n is the number of commits).

Fig. 4. Commit Digest Example: December 11&12, 2017 – JetUML. Watch on YouTube ▶

The structure of the animation, *i.e.,* the sequence of *prologue*, *corpus*, and *epilogue*, summarizes the chain of the commits animated: The change-phase mapping is preserved and the history between commits is represented in the *corpus* phase, which has as many rounds as the digest magnitude.

At the beginning of the animation all file versions are depicted ①, and newly added files, which appear fading in during the *prologue*, are represented only by their names. The single commits changes are represented during the *corpus* phase ③-⑦ and are animated *in sequence*: Frame ⑤ in Figure 4 is the pivot frame, *i.e.,* the conjunction frame between commits. In the *epilogue* deleted versions disappear ⑧, and the *post-digest* represent the result of the changes applied to

all files versions in the commit ⑨, especially if compared to the *pre-digest* ①. *Pre-digest* and *post-digest* phases are analogous to the single commit animation's *pre-commit* and *post-commit* phases.

When files are touched by multiple and sequential commits, the animation highlights the chain of changes those files underwent. Figure 4 exemplifies such a case: The animation conveys that *Context.java* is first added and then deleted, and that *GraphElement.java* and *AbstractNode.java* first grow and then shrink. *NamedNode.java* also slightly changes over all commits (from ⑤-⑧ it does not change much, indeed 2 lines of code out of 76 were substituted). All other file versions are added or grow in size within a single commit.
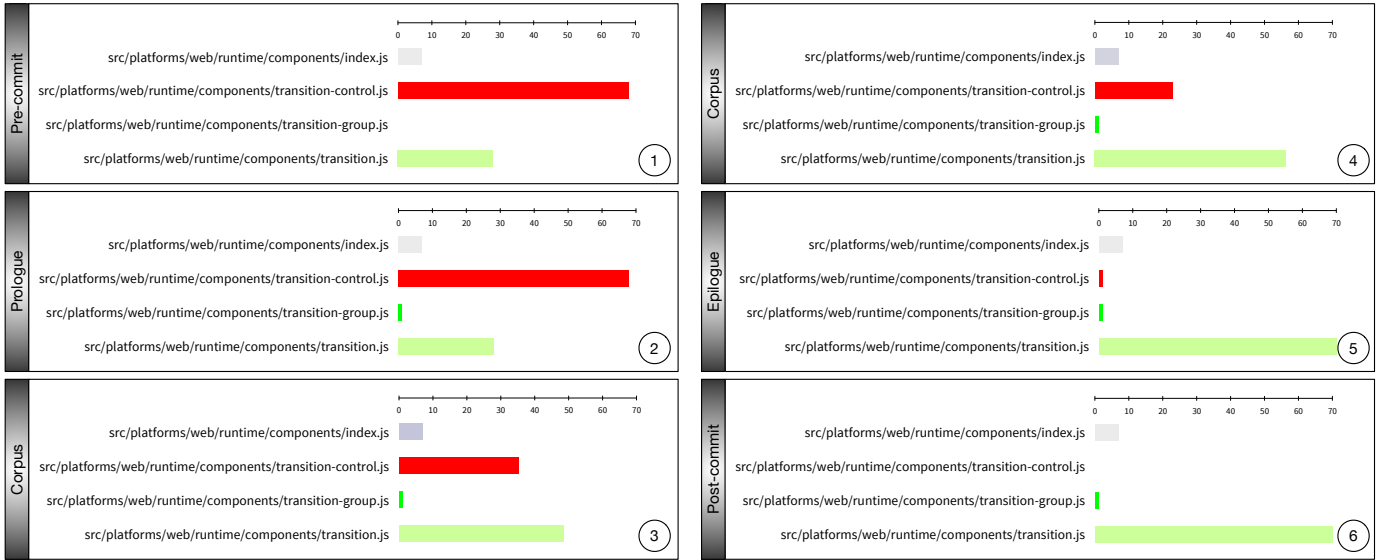
Fig. 5. Vue: Single commit animation of tangled code changes from July 13, 2016. Watch on YouTube ▶

V. CASE STUDIES

To evaluate the usefulness of our approach, we explored 7 (out of 10) repositories and report 3 case studies from 3 different projects, pertaining to:

- Tangled code changes identification;
- Chronological settlement of idem-timestamp commits (*e.g.,* commits in pull-requests or GitHub issues);
- Summary of tiny commits (defined by Hattori *et al.* as those with 5 files or fewer [31]);
- Overview of development activities over a period of time.

The first case study (V-A) illustrates the benefits that our animations can provide while analyzing a **single commit**; the second (V-C) and third (V-B) depict **commits digests**.

A. Vue – Tangled Code Changes

Vue is a JavaScript framework for building UIs, that builds on top of standard HTML, CSS, and JavaScript. Vue, specifically Vue 2, is the 5th most popular software repository on GitHub by number of stars. Vue 2 has reached end of life on December 31, 2023 and has been succeeded by its next version, yet the repository is still available and maintained.

The commit we picked as a case study dates back to July 13, 2016. The first commit happened on April 4, 2016, thus at that time the repository was at the beginning of its evolution.

Figure 5 dissects the commit animation into 6 frames and represents the main frames of the animation that involves 4 file versions. Thanks to the use of 2 different shades of green, it is easy to distinguish files that are newly added from those who only incremented in their size.

Most importantly, animating single commits allows to grasp tangled code-changes. In the case study proposed it appears that the content of *transition-control.js* was transmitted to *transition-group.js* and *transition.js*.

In fact, while the bar of *transition-control.js* shrinks to its minimum size during the corpus phase ②-⑤ and disappears during the epilogue ⑤, on the other hand *transition.js* gets increased ②-⑤.

Accordingly, in *index.js* 2 out of 7 lines of code were changed. Indeed, as stated by the commit message *"transition-mode was merged into transition"*. The file *transition-group.js* gets created with 1 line of code only.

This animated visual depiction facilitates the gathering of possible dependencies between file versions. In the case of such commit, since changes in *transition.js* led to changes in *index.js*, one could expect also in the future to find this two files changing in the same context.

B. Oh My Zsh – A Pull Request of Tiny Commits

Oh My Zsh is a community-driven framework for managing zsh configuration; it is the 7th most popular project by number of stars. As of May 28, 2020, it has 7,235 commits.

As stated at the beginning of this section, the second and third scenarios that we believe might benefit from our technique are: (a) the chronological settlement of idem-timestamp commits and (b) the summary of tiny commits. These benefits are leveraged in the **commits digest modality**, which enables to animate a sequence of commits possibly tied to each other, such as those part of the same pull request (the latter comes under the scenario of idem-timestamp commits).

One disadvantage of VCS, such as Git, is that there are cases when the historical information is rubbed out, *e.g.,* when a pull request is ready to be merged, the commit order and authorship information can be preserved or not, depending on the merge strategy adopted [56]. Also, still in the case of commits belonging to pull requests, VCS compress the time span during which code changes occur, discarding relevant details: In such case, the commits have all the same timestamp, even if they were performed at separate times.
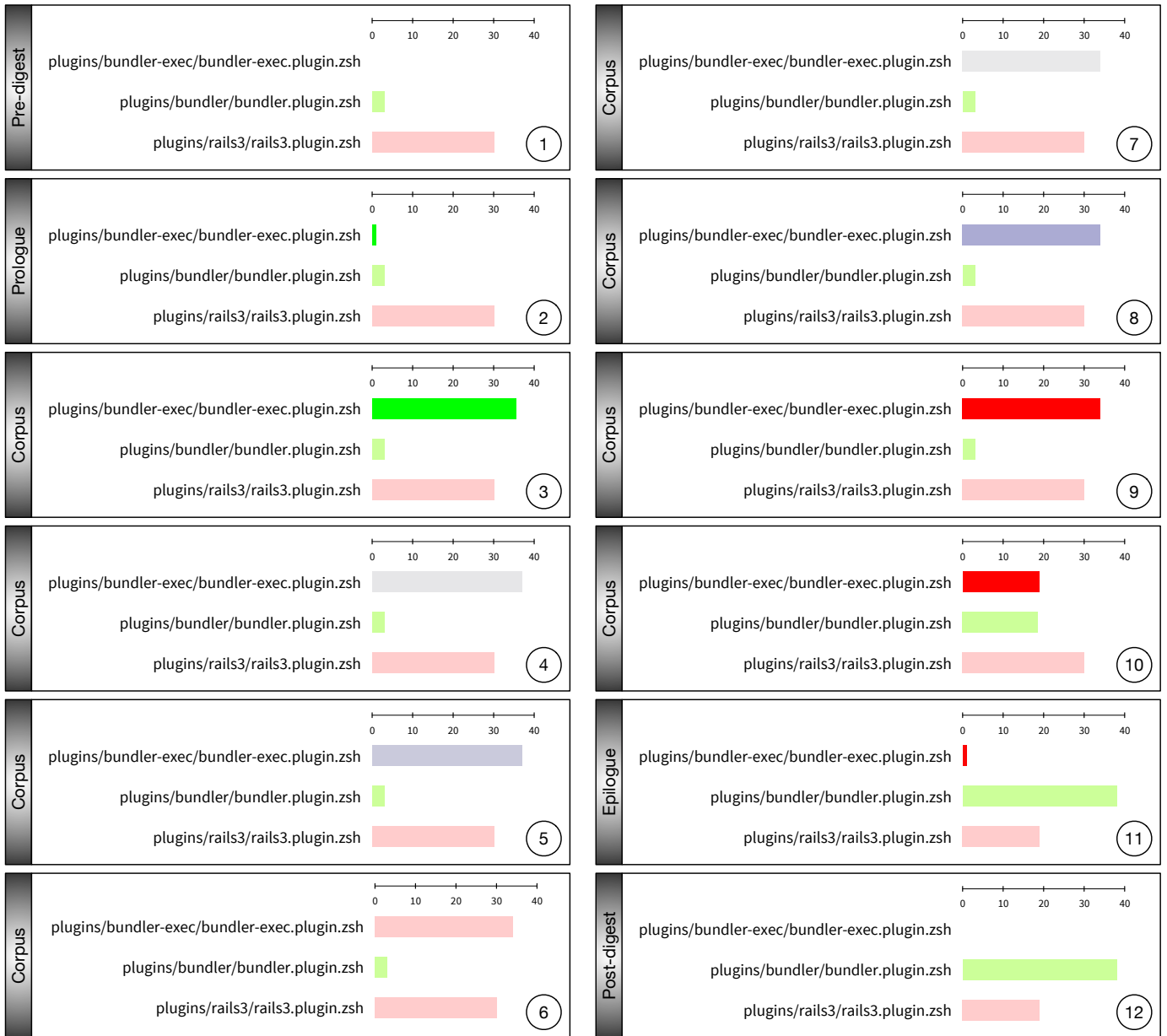
Fig. 6. Oh My Zsh: Commit digest of 10 idem-timestamp and tiny commits from a pull request on July 13, 2011. Watch on YouTube ▶

Analyzing a software system's commit history by browsing the commit diff views provided by modern tools is a time-consuming task. It is considerably more difficult if the commits include just a fiew files. The history of Oh My Zsh has a large amount of tiny commits (6,275 out of 7,257).

To demonstrate how our approach might be useful in the two previously described cases, we picked 10 tiny commits from the same pull request on July 13, 2011.

The resulting animation depicted in Figure 6, not only allows one to view the animated transition between different states of file versions all at once (which would normally be analyzed separately), but it also recreates the chronology of their changes, which is typically flattened by VCS.

The file *bundler-exec-plugin.zsh* undergoes 5 changes: Firstly it gets created (1)-(2); 3 out of 37 of its lines of code are modified (4)-(5) and it shrinks as 3 lines of code are deleted from it (6). It is modified again (6)-(7), this time more lines of code are substituted (15 out of 34). Then it is deleted (9)-(12). On the other hand, *bundler.plugin.zsh* and *rails3.plugin.zsh* only grow and shrink respectively (3)-(8).

C. Bootstrap – May 2012

Bootstrap is a "powerful, extensible, and feature-packed frontend toolkit" at the 8th place amongst the 10 most popular GitHub software repositories, based on the number of stars. When we collected the data, the repository was composed by 22,813 commits (on May 28 they were 22,800).

Fig. 7. Bootstrap: Development activity digest from May 14 to May 18, 2012. Watch on YouTube ▶

We selected 16 commits performed on May 2012 and represented them in a commit digest: The main frames of the animation are depicted in Figure 7. For the sake of clarity, we excluded from all commits made in May 2012 those with file versions large enough to skew the whole view.

The **commits digest modality** allows to aggregate sequential commits into a commit digest. This animation modality summarizes the selected commits, leveraging software evolution comprehension when a period of time of development, *e.g.,* a month, a week, or even a day, needs to be explored.

A way to watch sequential changes using visual aids has the potential to enhance code comprehension tasks as well: It can help to understand the context of the changes made, and identify files that are regularly modified. The motivation of our animated commits digest is to make software evolution more accessible by grouping sequential commits.

Typically, when developers review code, they are faced with textual code-diffs that do not readily allow them to traverse the history previous to the specific diff view they are looking at, which in fact might be beneficial to discover when a certain feature or defect was introduced. To the best of our knowledge, the only way to achieve this is to manually browse the (textual) commits history. Reading textual diffs and browsing software development history are both time-consuming processes.

The commits digest in Figure 7 provides a visual overview of the sequence of changes tracked by 16 commits. The digest illustrates how files change throughout the month of May, 2012. The first file undergoing changes is *js/bootstrap-tooltip.js*, that increases ①-⑨ and decreases ⑩-⑫. The files *navbar.html*, *docs/examples/navbar.html*, and *forms.html* get created in 3 subsequent commits and are incremented until they reach quite the same size ③-④, before *forms.html* starts being deleted ④. 8 commits after its creation, *navbar.html* gets deleted ⑧-⑫; on the other hand *docs/examples/-navbar.html* is renamed in *tests/tests/navbar.html* ⑦. The files *docs/templates/pages/index.mustache* and *docs/index.html* undergo lines substitutions during the 10th commit of the digest: Only 1 line is changed in both files. The files *js/bootstrap-dropdown.js* and *less/type.less* remain subtractive during the whole corpus phase ③-⑩; on the opposite *js/tests/unit/bootstrap-tooltip.js* and *README.md* linger as additive files. In the file *less/component-animations.less*, 3 lines out of 20 are substituted ⑩.

VI. DISCUSSION

The comprehension of commits is key in program comprehension activities, and in practical scenarios where developers need to review past changes. Understanding the intents of the implementation behind commits is fundamental as well. Static visualizations are successful in many contexts, yet they fall short when it comes to representing dynamic events, such as code changes. We proposed a novel means to understand fine-grained software evolution using animated and interactive visualizations. However, the approach has a number of shortcomings, still.

Limitations. Representing large commits or commits with one or more file versions containing a large number of lines of code distorts the visualization and makes it difficult to spot the changes in smaller files. While one could think of applying logarithmic scaling, the question that should be asked is why there are very large files in the first place.

At this time we filter out binary files from the commits of the projects under examination. While they do not provide much new knowledge regarding coding activities (they can only be added or deleted), they remain part of the codebase if present, and should be presented as well.

Next steps. We will refine our animated visualizations, by providing metaphors other than the one used which is based on polymetric views [57]: The idea is to provide means to understand *where* in a file lines are added, deleted, or substituted. We will take inspirations from the SeeSoft tool [14] and the Microprints [58], which are capable of depicting the code down to the level of single characters. Last, but not least, we plan on performing a controlled experiment with human subjects, on the one hand to gather feedback to ameliorate the animations, on the other hand to evaluate to what extent and in which contexts our animations prove to be beneficial.

VII. CONCLUSION

Software systems evolution is a difficult domain to understand in and of itself, as it is composed of complex and heterogeneous information, coupled with scarse documentation: For example, commit messages are often of poor quality [59] and almost 40% of all pull requests are not tracked as merged even though they were [56]. One of the promises of software visualization is to tackle these complexities.

We proposed an approach, implemented in an interactive animation tool, whose fundamental idea is to provide a new way to understand and look at code changes. The animated visualizations we propose seek to facilitate program comprehension and propose a fresh approach to analyze fine-grained software evolution.

BONUS

For recreational purposes, we created a digest of all the commits from the repository we used to write the paper (Figure 8). The digest animation illustrates the benefits of extending the deadline, which resulted in nearly 40 additional commits on the files depicted in Figure 8. The digest is available on YouTube ▶.
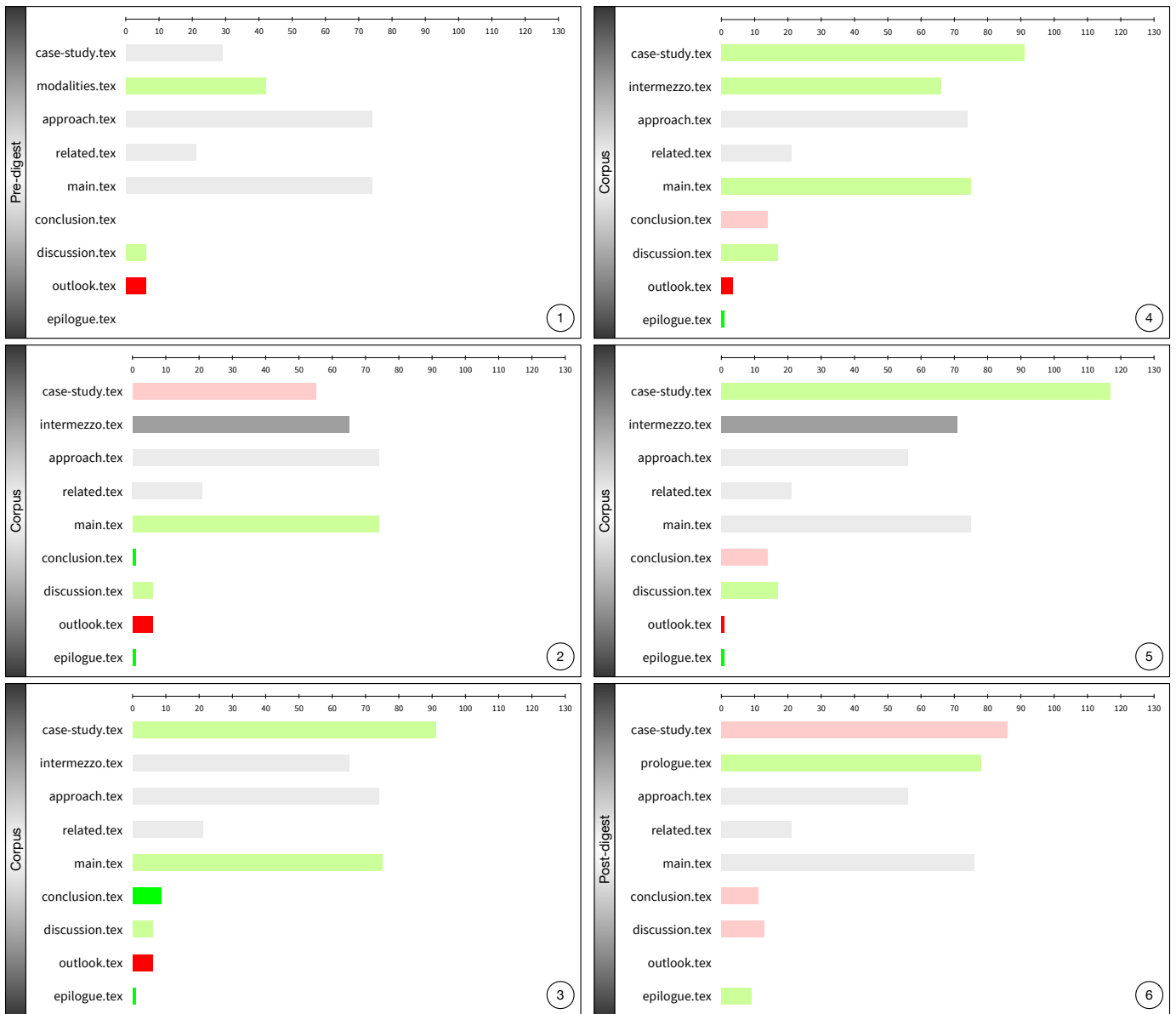
ACKNOWLEDGMENTS

Fig. 8. Frame 1 is the status of the file versions depicted on June 15. Frames 2-5 show the activity during the days 15-18 June. Frame 6 is the status of the file versions involved in the last changes at submission time.

REFERENCES

[1] M. M. Lehman, "Laws of software evolution revisited," in *European workshop on software process technology*. Springer, 1996.

[2] R. D. Banker, G. B. Davis, and S. A. Slaughter, "Software development practices, software complexity, and software maintenance performance: A field study," *Management science*, vol. 44, no. 4, pp. 433–450, 1998.

[3] G. M. Weinberg, *The psychology of computer programming*. Van Nostrand Reinhold New York, 1971, vol. 29.

[4] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *Proceedings of ICSE 2007 (International Conference on Software Engineering)*. IEEE, 2007, pp. 344–353.

[5] R. Minelli, A. Mocci, and M. Lanza, "I know what you did last summer-An investigation of how developers spend their time," in *Proceedings of ICPC 2015 (International conference on Program Comprehension)*. IEEE, 2015, pp. 25–35.

[6] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits," in *Proceedings of ICSE 2006 (International Conference on Software Engineering)*. ACM, 2006.

[7] V. Singh, L. L. Pollock, W. Snipes, and N. A. Kraft, "A case study of program comprehension effort and technical debt estimations," in *Proceedings of ICPC 2016 (International Conference on Program Comprehension)*. IEEE, 2016, pp. 1–9.

[8] M.-A. Storey, "Theories, methods and tools in program comprehension: Past, present and future," in *Proceedings of IWPC 2005 (International Workshop on Program Comprehension)*. IEEE, 2005, pp. 181–191.

[9] E. Fregnan, L. Braz, M. D'Ambros, G. Çalıklı, and A. Bacchelli, "First come first served: The impact of file position on code review," in *Proceedings of ESEC/FSE 2022 (Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering)*. ACM, 2022, pp. 483–494.

[10] D. Moody, "The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering," *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 756–779, 2009.

[11] J. T. Stasko, "Tango: A framework and system for algorithm animation," *ACM SIGCHI Bulletin*, vol. 21, no. 3, pp. 59–60, 1990.

[12] S. Diehl, "Software visualization," in *Proceedings of ICSE 2005 (International Conference on Software Engineering)*, 2005, pp. 718–719.

[13] S. Benford, C. Brown, G. Reynard, and C. Greenhalgh, "Shared spaces: Transportation, artificiality, and spatiality," in *Proceedings of CSCW 1996 (Conference on Computer Supported Cooperative Work)*. ACM, 1996, pp. 77–86.

[14] S. Eick, J. Steffen, and E. Sumner, "Seesoft-A tool for visualizing line oriented software statistics," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 957–968, 1992.

[15] M. Pinzger, H. Gall, M. Fischer, and M. Lanza, "Visualizing multiple evolution metrics," in *Proceedings of SOFTVIS 2005 (ACM Symposium on Software Visualization)*. ACM, 2005, pp. 67–75.

[16] W. Scheibel, C. Weyand, and J. Döllner, "Evocells – A treemap layout algorithm for evolving tree data." in *VISIGRAPP (3: IVAPP)*, 2018, pp. 273–280.

[17] D. P. Tua, R. Minelli, and M. Lanza, "Voronoi evolving treemaps," in *Proceedings of VISSOFT 2021 (Working Conference on Software Visualization)*. IEEE, 2021, pp. 1–5.

[18] C. Knight and M. Munro, "Virtual but visible software," in *Proceedings of VIS 2000 (Conference on Information Visualization)*. IEEE, 2000, pp. 198–205.

[19] F. Pfahler, R. Minelli, C. Nagy, and M. Lanza, "Visualizing evolving software cities," in *Proceedings of VISSOFT 2020 (Working Conference on Software Visualization)*. IEEE, 2020, pp. 22–26.

[20] R. Wettel and M. Lanza, "Visualizing software systems as cities," in *Proceedings of VISSOFT 2007 (Working Conference on Software Visualization)*. IEEE, 2007, pp. 92–99.

[21] A. Hoff, L. Gerling, and C. Seidl, "Utilizing software architecture recovery to explore large-scale software systems in virtual reality," in *Proceeding of VISSOFT 2022 (Working Conference on Software Visualization)*. IEEE, 2022, pp. 119–130.

[22] D. Moreno-Lumbreras, R. Minelli, A. Villaverde, J. M. González-Barahona, and M. Lanza, "Codecity: On-screen or in virtual reality?" in *Proceedings of VISSOFT 2021 (Working Conference on Software Visualization)*. IEEE, 2021, pp. 12–22.

[23] M. F. Kleyn and P. C. Gingrich, "GraphTrace—Understanding object-oriented systems using concurrently animated views," in *Proceedings of OOPSLA 1988 (Conference on Object-Oriented Programming Systems, Languages and Applications)*. ACM, 1988, pp. 191–205.

[24] G. Occhipinti, C. Nagy, R. Minelli, and M. Lanza, "Syn: Ultra-scale software evolution comprehension," in *Proceedings of ICPC 2023 (International Conference on Program Comprehension)*. IEEE, 2023, pp. 69–73.

[25] C. M. Taylor and M. Munro, "Revision towers," in *Proceedings of VISSOFT 2002 (International Workshop on Visualizing Software for Understanding and Analysis)*. IEEE, 2002, pp. 43–50.

[26] M. D. Storey, F. D. Fracchia, and H. A. Müller, "Cognitive design elements to support the construction of a mental model during software exploration," *Journal of Systems and Software*, vol. 44, no. 3, pp. 171–185, 1999.

[27] T. Dal Sasso, R. Minelli, A. Mocci, and M. Lanza, "Blended, not stirred: Multi-concern visualization of large software systems," in *Proceedings of VISSOFT 2015 (International Conference on Software Visualization)*. IEEE, 2015, pp. 106–115.

[28] T. D. LaToza and B. A. Myers, "Hard-to-answer questions about code," in *Evaluation and usability of programming languages and tools*, 2010, pp. 1–6.

[29] F. Servant and J. A. Jones, "History slicing: Assisting code-evolution tasks," in *Proceedings of SIGSOFT 2012 (International Symposium on the Foundations of Software Engineering)*, 2012, pp. 1–11.

[30] R. Holmes and A. Begel, "Deep intellisense: A tool for rehydrating evaporated information," in *Proceedings of MSR 2008 (Working Conference on Mining Software Repositories)*. ACM, 2008, pp. 23–26.

[31] L. P. Hattori and M. Lanza, "On the nature of commits," in *Proceedings of ASE 2008 (International Conference on Automated Software Engineering)*. ACM, 2008, pp. 63–71.

[32] A. Alali, H. Kagdi, and J. I. Maletic, "What's a typical commit? A characterization of open source software repositories," in *Proceedings of ICPC 2008 (International Conference on Program Comprehension)*. IEEE, 2008, pp. 182–191.

[33] K. A. Safwan and F. Servant, "Decomposing the rationale of code commits: The software developer's perspective," in *Proceedings of ESEC/FSE 2019 (Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering)*, 2019, pp. 397–408.

[34] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes? An exploratory study in industry," in *Proceedings of FSE 2012 (International Symposium on the Foundations of Software Engineering)*. ACM, 2012, pp. 1–11.

[35] K. Yamauchi, J. Yang, K. Hotta, Y. Higo, and S. Kusumoto, "Clustering commits for understanding the intents of implementation," in *Proceedings of ICSME 2014 (International Conference on Software Maintenance and Evolution)*. IEEE, 2014, pp. 406–410.

[36] D. Rozenberg, I. Beschastnikh, F. Kosmale, V. Poser, H. Becker, M. Palyart, and G. C. Murphy, "Comparing repositories visually with repograms," in *Proceedings of MSR 2016 (Working Conference on Mining Software Repositories)*, 2016, pp. 109–120.

[37] M. Lanza and S. Ducasse, "Understanding software evolution using a combination of software visualization and software metrics," *Obj. Logiciel Base données Réseaux*, vol. 8, no. 1-2, pp. 135–149, 2002.

[38] M. Lanza, S. Ducasse, H. C. Gall, and M. Pinzger, "CodeCrawler: An information visualization tool for program comprehension," in *Proceedings of ICSE 2005 (International Conference on Software Engineering), 15-21 May 2005, St. Louis, Missouri, USA*, G. Roman, W. G. Griswold, and B. Nuseibeh, Eds. ACM, 2005, pp. 672–673.

[39] M. Lanza, "The evolution matrix: Recovering software evolution using software visualization techniques," in *Proceedings of IWPSE 2001 (International Workshop on Principles of Software Evolution)*. ACM, 2001, pp. 37–42.

[40] E. Aghajani, A. Mocci, G. Bavota, and M. Lanza, "The code time machine," in *Proceedings of ICPC 2017 (International Conference on Program Comprehension)*. IEEE, 2017, p. 356–359.

[41] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on software engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[42] Y. Kim, J. Kim, H. Jeon, Y.-H. Kim, H. Song, B. Kim, and J. Seo, "Githru: Visual analytics for understanding software development history through git metadata analysis," *IEEE Transactions on Visualization and Computer Graphics*, vol. 27, no. 2, pp. 656–666, 2020.

[43] L. P. Rieber, "Using computer animated graphics in science instruction with children." *Journal of Educational Psychology*, vol. 82, no. 1, p. 135, 1990.

[44] D. L. Sonnier and S. L. Hutton, "Enhancing visual aids through the use of animation," in *Proceedings of MSCCC 2004 (Conference on Mid-South College Computing)*. Mid-South College Computing Conference, 2004, p. 155–164.

[45] E.-M. Yang, T. Andre, T. J. Greenbowe, and L. Tibell, "Spatial ability and the impact of visualization/animation on learning electrochemistry," *International Journal of Science Education*, vol. 25, no. 3, pp. 329–349, 2003.

[46] R. L. London and R. A. Duisberg, "Animating programs using Smalltalk," *Computer*, vol. 18, no. 08, pp. 61–71, 1985.

[47] G. Salomon, *Interaction of Media, Cognition, and Learning: An Exploration of How Symbolic Forms Cultivate Mental Skills and Affect Knowledge Acquisition*, 1st ed. Routledge, 1994.

[48] T. N. Höffler and D. Leutner, "Instructional animation versus static pictures: A meta-analysis," *Learning and instruction*, vol. 17, no. 6, pp. 722–738, 2007.

[49] M. Bétrancourt and B. Tversky, "Effect of computer animation on users' performance: A review," *Le travail humain*, vol. 63, no. 4, p. 311, 2000.

[50] F. Paas, A. Renkl, and J. Sweller, "Cognitive load theory and instructional design: Recent developments," *Educational psychologist*, vol. 38, no. 1, pp. 1–4, 2003.

[51] W. Schnotz and C. Kürschner, "A reconsideration of cognitive load theory," *Educational Psychology Review*, vol. 19, pp. 469–508, 2007.

[52] J. Sweller, J. J. Van Merrienboer, and F. G. Paas, "Cognitive architecture and instructional design," *Educational Psychology Review*, vol. 10, pp. 251–296, 1998.

[53] S. Berney and M. Bétrancourt, "Does animation enhance learning? A meta-analysis," *Computers & Education*, vol. 101, pp. 150–167, 2016.

[54] M. D'Ambros, M. Lanza, and R. Robbes, "Commit 2.0," in *Proceedings of Web2SE 2010 (1st International Workshop on Web 2.0 for Software Engineering)*. ACM, 2010, pp. 14–19.

[55] N. Tsantalis, A. Ketkar, and D. Dig, "RefactoringMiner 2.0," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, 2020.

[56] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining GitHub," in *Proceedings of MSR 2014 (Working Conference on Mining Software Repositories)*. ACM, 2014, p. 92–101.

[57] M. Lanza, "Codecrawler - Polymetric views in action," in *Proceedings of ASE 2004 (International Conference on Automated Software Engineering)*. IEEE, 2004, pp. 394–395.

[58] S. Ducasse, M. Lanza, and R. Robbes, "Multi-level method understanding using microprints," in *Proceedings of VISSOFT 2005 (International Workshop on Visualizing Software for Understanding and Analysis)*. IEEE, 2005, pp. 33–38.

[59] Y. Tian, Y. Zhang, K.-J. Stol, L. Jiang, and H. Liu, "What makes a good commit message?" in *Proceedings of ICSE 2022 (International Conference on Software Engineering)*, 2022, pp. 2389–2401.