VERSIONING IN INTERACTIVE SYSTEMS

A Thesis Submitted to the College of

Graduate and Postdoctoral Studies

In Partial Fulfillment of the Requirements

For the Degree of Master of Science

In the Department of Computer Science

University of Saskatchewan

Saskatoon

By

Gurjot Singh Bhatti

PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other uses of material in this thesis in whole or part should be addressed to:

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9
Canada

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan S7N 5C9
Canada

ABSTRACT

Dealing with past states of an interactive system is often difficult, and users often resort to unwieldy methods such as saving and naming multiple copies. Versioning tools can help users save and manipulate different versions of a document, but traditional tools designed for coding are often unsuitable for interactive systems. Supporting versioning in interactive systems requires investigation of how users think about versions and how they want to access and manipulate past states. We first surveyed users to understand what a 'version' means to them in the context of digital interactive work, and the circumstances under which they create new versions or go back to previous ones. We then built a versioning tool that can store versions using a variety of explicit and implicit mechanisms and shows a graphical representation of the version tree to allow easy inspection and manipulation. To observe how users used versions in different work contexts, we tested our versioning tool in two interactive systems – a game level editor and a web analysis tool. We report several new findings about how users of interactive systems create versions and use them as undo alternatives, exploring options, and planning future work. Our results show that versioning can be a valuable component that improves the power and usability of interactive systems. The new understanding that we gained about versioning in interactive environments by developing and evaluating our custom version tool can help us design more effective versioning tools for interactive systems.

ACKNOWLEDGMENTS

I would like to express my gratitude to my supervisor Carl Gutwin for his guidance and input throughout my graduate career at the University of Saskatchewan. I also wish to extend my sincere thanks to the faculty and staff of the Department of Computer Science, especially the students and staff of the HCI Lab, for their assistance and friendship.

This thesis is dedicated to my parents and my sister.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

2D	Two Dimensional
3D	Three Dimensional
GUI	Graphical User Interface
PC	Personal Computer
SVN	Subversion
UI	User Interface
VCS	Version Control System

CHAPTER 1

INTRODUCTION

Versioning is integral to recording and maintaining proper logs for the evolution of work. The different editions of books are an example of versioning, where each edition introduces changes and updates to the content of a book. When we apply this concept to digital work, it is referred to as digital versioning. Digital versioning is also crucial for error recovery and data backup in case of inadvertent actions or system failures. Although several versioning approaches currently exist, none of them fully meets the needs of interactive systems in which both the document content and the state of the interface are important elements of the user's interaction history. Interactive systems are the systems that accept input from the user as they run [52], and if these systems also have a graphical interface attached, then they are usually referred to as graphical interactive systems. These systems have two states – document/model state and User Interface (UI) state. The document/model state includes the data associated with the work objects in the application and the UI state includes the data associated with the view/interaction state of the interface elements of the application.

There are multiple ways to handle digital versioning. One approach to handle it is to manually duplicate existing work, e.g., as a new file. Because file-based versioning requires manual effort to duplicate the files, it only works well when there are a small number of project files. As a result, if the project has a significant number of files, this method of manually duplicating files becomes inconvenient to use. The second approach is to leverage software applications' built-in auto-versioning features, which save multiple versions of project files. Blender and 3ds Max, for example, allow for automatic versioning of project files at regular intervals [53,54]. The third approach is to create versions or backups of project files using cloud-based solutions such as Dropbox and OneDrive [55,56]. These services are designed to help users save time and effort by recording and managing multiple versions of the same document. The fourth approach is to use Version Control Systems (VCS) such as Git, SVN, and Mercurial [57–59] to manage revisions of source code, binary files, and digital assets. These tools, however, only handle explicit versioning

(i.e., where the user must choose to save a version), are used externally (not integrated within the interactive environments) and can require substantial configuration to set up. For example, in Visual Studio Code (VS Code) [60], a user can utilize VCS features by accessing the Source Control panel (see **Figure 1.1.1**) and can perform versioning for their document state data. But the VCS system must be installed and configured on users' system before it can be used inside VS Code.

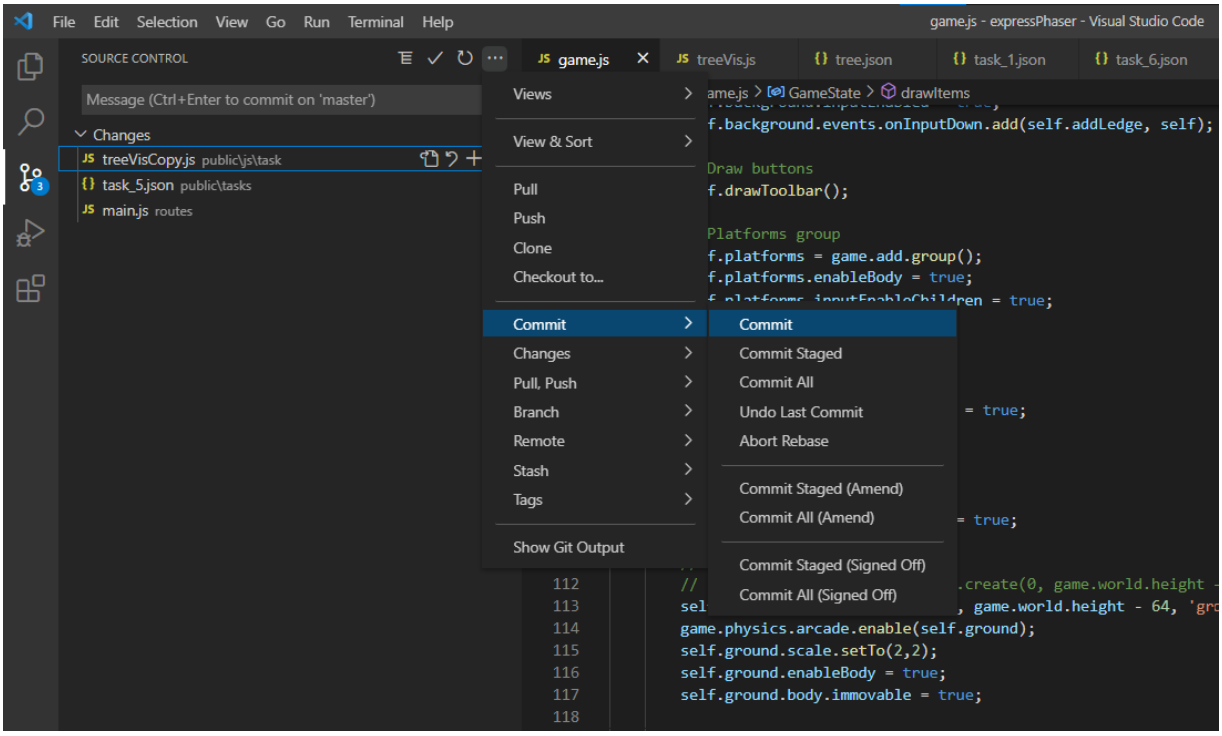
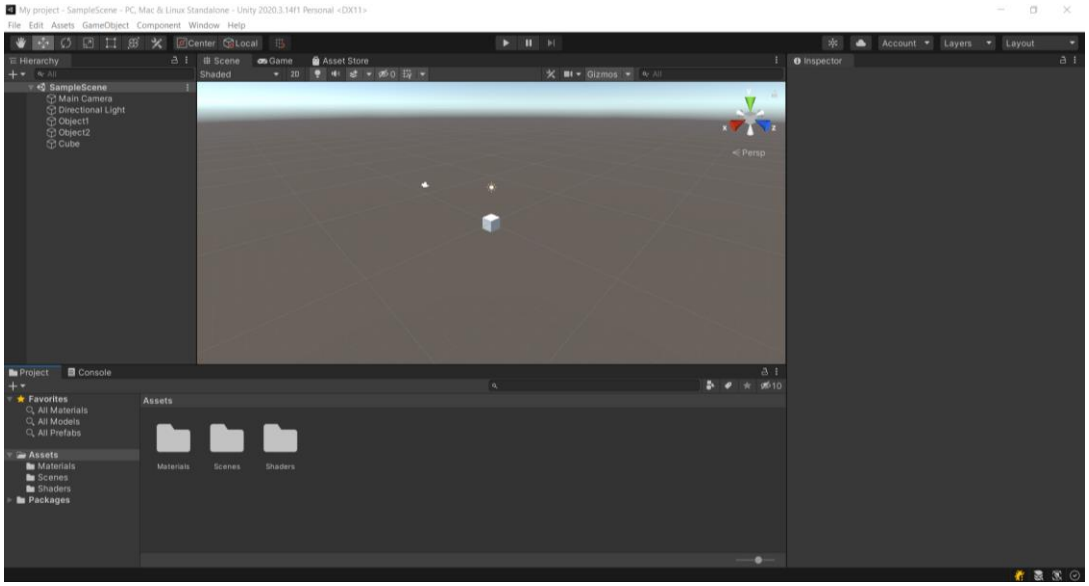


Figure 1.1.1 Git integration inside Visual Studio Code using Source Control Panel.

However, for interactive systems, all four approaches to handle digital versioning have two shortcomings. First, the difficulty of current version tools does not make it easy for users of interactive systems (e.g., visual editors) to create and make use of fine-grained versions. For example, saving a new file or committing to Git every time a user moves an object in a game level editor requires too much effort. Second, current tools only store the state of documents/files and do not store any information regarding past states of interactive elements such as interface controls and widgets, and the system settings or user preferences of an application. For example, if a user is working in an interactive system like a game editor such as Unity [61] or Unreal [62], they might

be able to use versioning to keep track of changes in their document state such as game objects and their properties in their current scene. But any changes related to the UI, such as docking a panel to a different location and browsing to a different folder in the project view, are not stored in any version (see **Figure 1.1.2**).

(a)



(b)

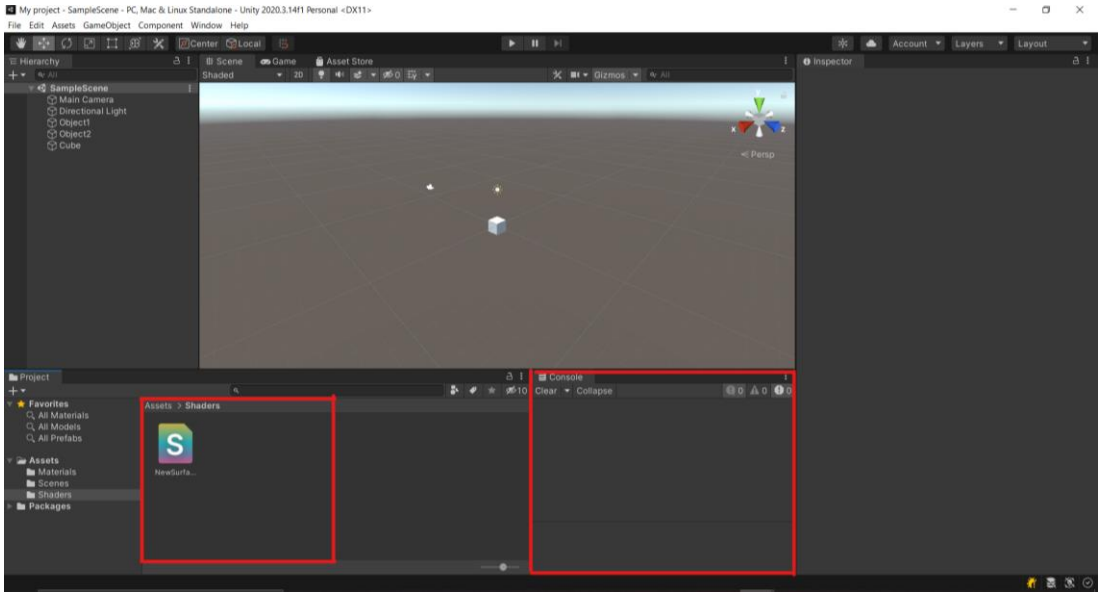
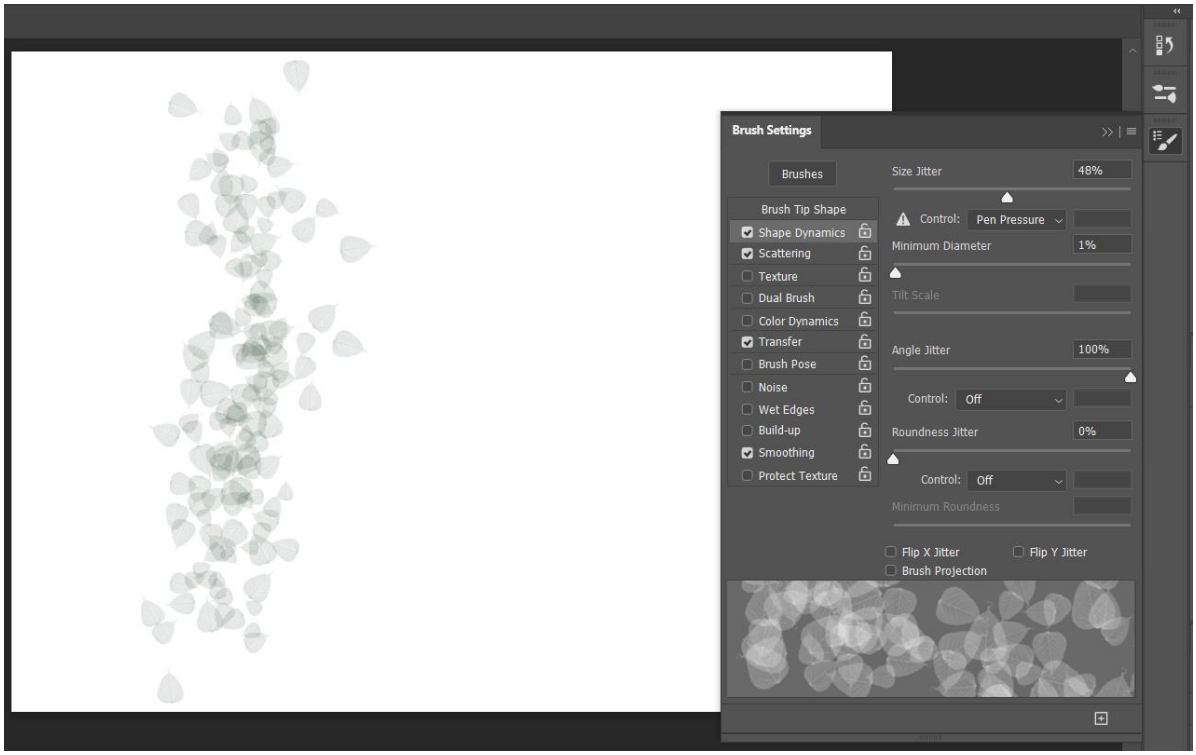


Figure 1.1.2 (a) Unity Editor with default UI view with Project panel showing 'Assets' folder (b) Unity Editor when Console panel is docked to the right side of the Project panel which now shows 'Shaders' folder.

Similarly, in a 3D modeling application such as Blender, changes in global settings that affect the whole system, such as display resolution and anti-aliasing value, are never stored by current versioning tools.

Another example is the changes that a user can make to brush presets in Adobe Photoshop [63]. **Figure 1.1.3 (a)** shows a scenario where a user is drawing leaf patterns using a brush preset ‘A’ with the default settings. If the user makes any adjustments to the parameters of this brush preset in the Brush Settings Panel (see **Figure 1.1.3 (b)**) they will have to manually create a new brush preset to save their adjustments because the moment they switch to a different brush preset ‘B’, they will lose their changes to the brush preset ‘A’. This shows that Photoshop does not store any information about the past states of the UI elements. Therefore, the current tools that will be used to version the Photoshop file will not contain any information regarding the UI state changes.

(a)



(b)

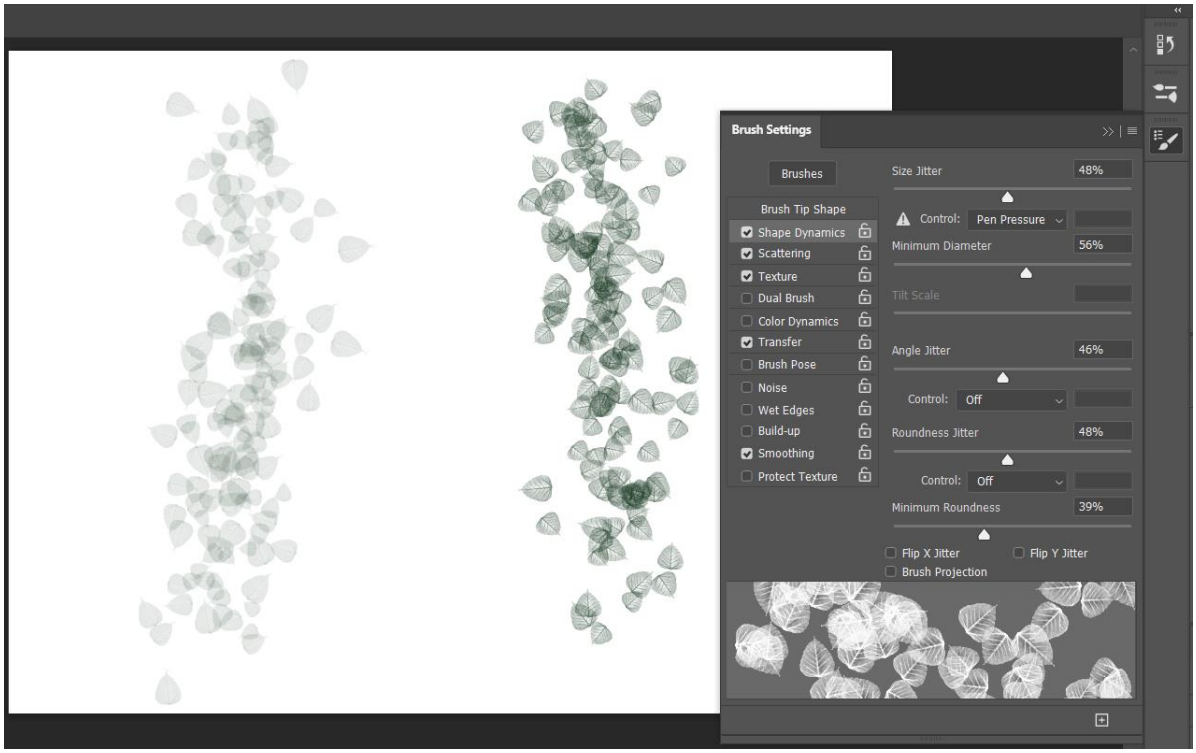


Figure 1.1.3 (a) Photoshop's brush preset with default settings, (b) brush preset with modified settings.

Supporting versioning in interactive systems requires a better understanding of what a 'version' implies in the context of a specific digital system and what constitutes an interaction history. Moreover, knowledge on how users want to access and manipulate past states of the interactions is needed to develop effective versioning tools. For example, past states are important while working in an interactive system for several reasons: users may want to know their interaction workflow (how they reached the current version); they may want to explore possibilities by using previous states as templates; they may want to refer to a previous state to plan future changes; or they may want to use a past state as an undo state to revert in case the new changes are not in line with the requirements.

In this thesis, we gathered new information about versioning in interactive systems and designed a versioning tool that can be integrated into a web-based interactive system and that allows users to record their interaction history without relying on any external tools. We evaluated the tool in two

case studies to see whether people can effectively use integrated versioning in several realistic tasks. The study provides new knowledge about several aspects of interactive versioning, including people's versioning patterns, the use of versions as templates, how people use branching within a version tree, and using versions as an alternative to undo: this knowledge can aid the design of effective versioning tools for other interactive systems.

1.1 PROBLEM

The problem we address in this thesis is the lack of understanding of how to build effective versioning tools for interactive systems - including design issues for developing such tools to deal with past states of interactive work and their ability to save users' interaction history.

All of the existing approaches to handle versioning in interactive systems have drawbacks. Manual file duplication requires substantial effort and does not save the UI state of the application where the file is created. The built-in versioning approach does not work with applications more generally. Cloud-based solutions like Dropbox and OneDrive require explicit actions to save versions and they also do not save any UI state data. The VCS systems are used as external tools that must be set up before they can be used. They also necessitate a significant effort in order to create versions. A user, for example, would need to set up an external repository to store all versions of the project. All the changes for different files in this project would be bundled by a user in a commit (snapshot for current changes). After that, these changes are pushed to the external repository.

These drawbacks imply that a different type of versioning is required for interactive systems. There is a lack of knowledge when we consider designing for such environments. This knowledge can be gained by answering questions such as what must be stored in a version, how to store these versions, are the UI states part of the version, and when and how users create and access these versions. These and other design questions must be answered to create an effective tool to save interaction history in an interactive environment.

1.2 SOLUTION

Our solution is to provide a new understanding about versioning in interactive environments by developing a custom interactive versioning tool. First, we sent out a survey to people over the internet regarding versions to understand how and when they are created, used, and modified. Second, we used the information gathered through our survey to develop a custom interactive versioning tool that can record both fine-grained document/model states either explicitly or implicitly, and can record the interactive state of the UI, in web-based applications. The tool also allowed users to inspect and interact with different versions visualized in a tree to provide an easy way to switch between versions and compare them (visualize changes in the system). Moreover, this tool also created implicit versions on granular trigger events (every 3 seconds, on every change, etc.) to provide even more versions that a user can explore at a later time. Third, we added our versioning tool to two interactive web applications to allow the system to store states of the system (data such as positions of sprites representing a level in a game level editor) or to save user interactions history with UI elements of the application (web-form elements like sliders or dropdowns) as separate versions.

1.3 STEPS TO THE SOLUTION

Our solution has been implemented in three parts as follows:

1.3.1 Understanding versions (survey)

We recruited online users of interactive systems and asked them to complete a survey that asked basic questions about versioning in interactive systems such as under what circumstances users create a new version of any file, how often do they go back to the previous version, and what naming convention do they follow when naming different versions of their files. Answers to these questions helped us gain an understanding of versioning in the context of several digital interactive environments.

1.3.2 Determine the representation for versions

To determine a visual representation for versions, we consulted previous research and other available tools that suitably represent linear and non-linear hierarchies. There are two main candidates – timeline/linear and trees, according to our findings. Timeline/linear representations are ideal for depicting linear changes or events in chronological order, but they are unable to represent branching. As a result, we used tree visualization to describe non-linear hierarchies in which changes can be stored as multiple separate points that diverge from a common point in the hierarchy.

1.3.3 Build the versioning tool

We built our custom version tool, presented in Chapter Three, that allows users to save an application's document/model state data and UI state data as a version. The tool was developed in Javascript and used JSON (Javascript Object Notation) to store the data for each version. All of the versions created were then presented in a version tree within a panel attached to the right-hand side of a web application's interactive interface. Users could access, alter, and explore versions stored in the version tree with our versioning tool.

1.4 EVALUATION

We evaluated how people will use our versioning tool that was developed as a solution to the problem of recording interaction history in an interactive system. We also wanted to see how well they could use our tool during some digital interactive work. Therefore, we added our versioning tool to two web-based applications and observed users who carried out tasks in the applications that involved versioning. The two applications were: game level editor and a web analysis tool. We then carried out a user study of level design tasks for a game to determine how our versioning tool helps users in managing different versions of their game level. This study focused on simple and fine-grained versioning of the document (saving the information of objects in each game level). We observed how users were able to successfully create versions of their levels, easily switch to

different versions when required, and create several branches of versions in a tree. We made additional observations such as using the preview feature to compare different versions and using versions as undo-redo to fix quick mistakes.

We also wanted to test our tool in another interactive environment where the focus was on saving interactive states of the UI. So, we tested our tool in a web interactive system used for performing analysis on microbiome datasets, where the state of UI elements such as fields of a web form could also be saved in a version. We observed what happens when there are a large number of versions generated when analyses are run in bulk mode, i.e., multiple analyses are run sequentially. We discovered how branching versions based on various parameters could make a large number of versions simple to navigate and understand.

1.5 CONTRIBUTIONS

There are three main contributions presented in this thesis: a new understanding of versioning in interactive systems and design issues for developing such tools; development of the versioning tool that can augment interactive systems; and evaluation of the developed tool to observe how people used versioning tool while performing design and analysis tasks, giving us an insight into the key elements such as versioning patterns, templating, and branching that are required to construct functional and helpful versioning tools for other interactive systems.

1.6 THESIS OUTLINE

The content of this thesis is organized into seven chapters.

Chapter Two presents a literature review of the related work that forms the foundation of research done in this thesis.

Chapter Three presents the survey study we carried out to ask people some basic questions about versioning in interactive systems. We discuss the necessity to create versions and the reasons to revert to previous versions.

Chapter Four details the design of our versioning tool and selection of appropriate visual representation for storing different versions in an interactive environment. It also presents our two custom interactive systems that we designed to test our versioning tool.

Chapter Five presents two user studies we did to test our tool in interactive environments – a game level editor and a web analysis tool. We discuss the design and analysis tasks that the participants performed and the observations we made in both studies. We also discuss user feedback and observations for how different version trees are created when versions are saved implicitly.

Chapter Six presents a discussion of the most important results and observations from both the studies from Chapter Five. We present our explanation for each of the findings and how they can be used in improving the design of versioning tools for interactive systems.

Finally, Chapter Seven concludes the research presented in this thesis. It discusses our contributions and provides a summary of our findings. It also highlights future work that can be done in relation to the research presented in this thesis.

CHAPTER 2

RELATED WORK

Augmenting interactive systems with an interactive versioning tool is based on five areas of research. These areas give us insight into versioning, ways to deal with past states of a system, user interactions with interactive history, current systems that implement versioning, and how other interactive environments deal with historical data. In the first section, we introduce and review definitions of versioning and explain why it is a crucial part of digital workflows. In the second section, we discuss ways of dealing with past states of a digital system. In the third section, we discuss how users can interact with interaction history within different applications. In the fourth section, we look at current systems that implement versioning in their workflows. Finally, in the fifth section, we review interactive environments that deal with historical data and versioning.

2.1 VERSIONING

Saving the current state of work is fundamental in a digital environment. Most of the time, people save their work to avoid losing changes made in a file since the last time it was worked on or to explore alternatives in their document. According to Nancel and Cockburn [32], a version is a fixed reference in history that represents a snapshot E_s of an element E at a given state S of history (see **Figure 2.1.1**). In other words, it is a fixed point in history that stores a particular state of the system. In computer science, versioning (or software versioning to be exact) “is the process of assigning either unique version names or unique version numbers to unique states of computer software” [64].

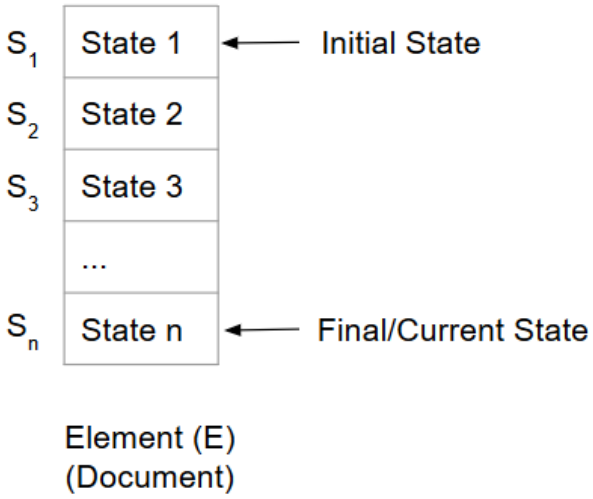


Figure 2.1.1 History Stack with various states S_i of an element E (adapted from Nancel and Cockburn [32].

The most common method of versioning is saving multiple copies of a file as shown in **Figure 2.1.2**. In the figure, three document files are named using the same keyword and suffixed with a number. Each file is a different version of the same document with some modifications to the previous state of file content.

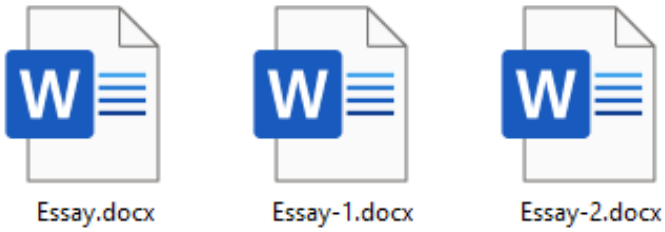


Figure 2.1.2 File with multiple versions.

However, file names that do not adequately describe the version of a document might lead to confusion among users when it comes to correctly identifying the document version they want to access (discussed further in **Section 3.4**).

2.2 DEALING WITH PAST STATES

Storing and retrieving past states is an important part of interacting with digital systems. It's all too easy to make a typing error in a text document, insert an incorrect image into a file, or erase an object from a design diagram by accident. Users require strategies to recover from such blunders and mistakes, which are often unavoidable. One method is to utilize undo-redo functions, a second method is to save frequently and build backups using autosaving and autoversioning, and a third method is to use version management tools.

2.2.1 Undo – Redo

The most common way to recover from errors is to utilize undo-redo, which can be found in most application softwares. Undo-redo allows users to carry out forward and backward error recovery and is usually implemented using the command design pattern [19] in a linear model of system states (**Figure 2.2.1**). Undo allows the user to return to a previous state of the system via one action or one change in a linear model while redo allows them to move back to a future state (only if the undo action was performed previously). This makes it easy to go backward (undo) and forward (redo) in time.

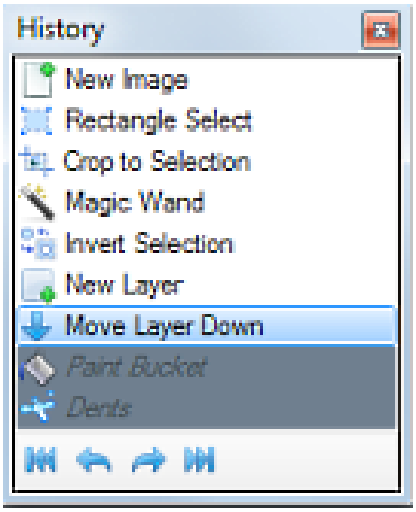


Figure 2.2.1 History window of Paint.net.

However, undo-redo is quite easy to break; if new changes are introduced after performing an undo action, then the previous changes in the redo stack may be lost and the user cannot access those changes. This can be solved to some extent using selective undo [38] that deletes only specific actions in the history without discarding any stored undo-redo commands on the stack [26]. This enables the user to perform any operations stored in a history list and gives them the ability to alter the history with more freedom, unlike regular undo-redo.

The “Three R’s” (Rewind, Repair, and Replay) undo model allows the users to go back in time within a document state history to fix a problem. As a result, the effects of the changes made when fixing the problem are propagated to the present state [3]. In the 'rewind' step, the user restores the system to a previous state, makes changes in the 'repair' step, and then in the 'replay' step, the user replays or re-executes the actions that follow the changed action according to the proposed changes. Altering a part of the history of a document can result in generating temporal paradoxes [29] that can introduce conflicts in the set of actions that follow the alteration. Avoiding such paradoxes requires careful approaches such as rejecting actions that result in conflicts.

Another type of undo model exists called Cascading undo [5] that removes conflicts between changes done through the undo-redo commands by tracking dependencies between user actions. Even with this technique, however, the undo can fail if the capacity of the history list of a system exceeds or when a dependency has changed for the resulting action.

Undo-redo mechanisms are not able to isolate specific versions, and for major changes between versions, stepping through individual undo actions would require substantial user effort.

2.2.2 Autosaving and Autoversioning

Autosave and autoversion are the important features that are helpful for backups and error recovery. Autosave saves the current work automatically, without the need to explicitly perform a save action. Autoversioning automatically creates a separate version of a working file or a project. Some applications such as MS Word, Adobe Photoshop, and Visual Studio Code have an option to auto-save files at a regular interval [53,65–67]. Applications like Blender, 3ds Max, Autodesk AutoCAD, and Zbrush automatically save versions of project files periodically or on every explicit

save [54,68,69]. Besides time-based or explicit saving, there are other cases when an application can autosave a new version of a working document. For example, Adobe Story (Classic) saves versions of a document in cases of changes like online-offline mode switching, session-based changes, collaboration, and overwriting [70].

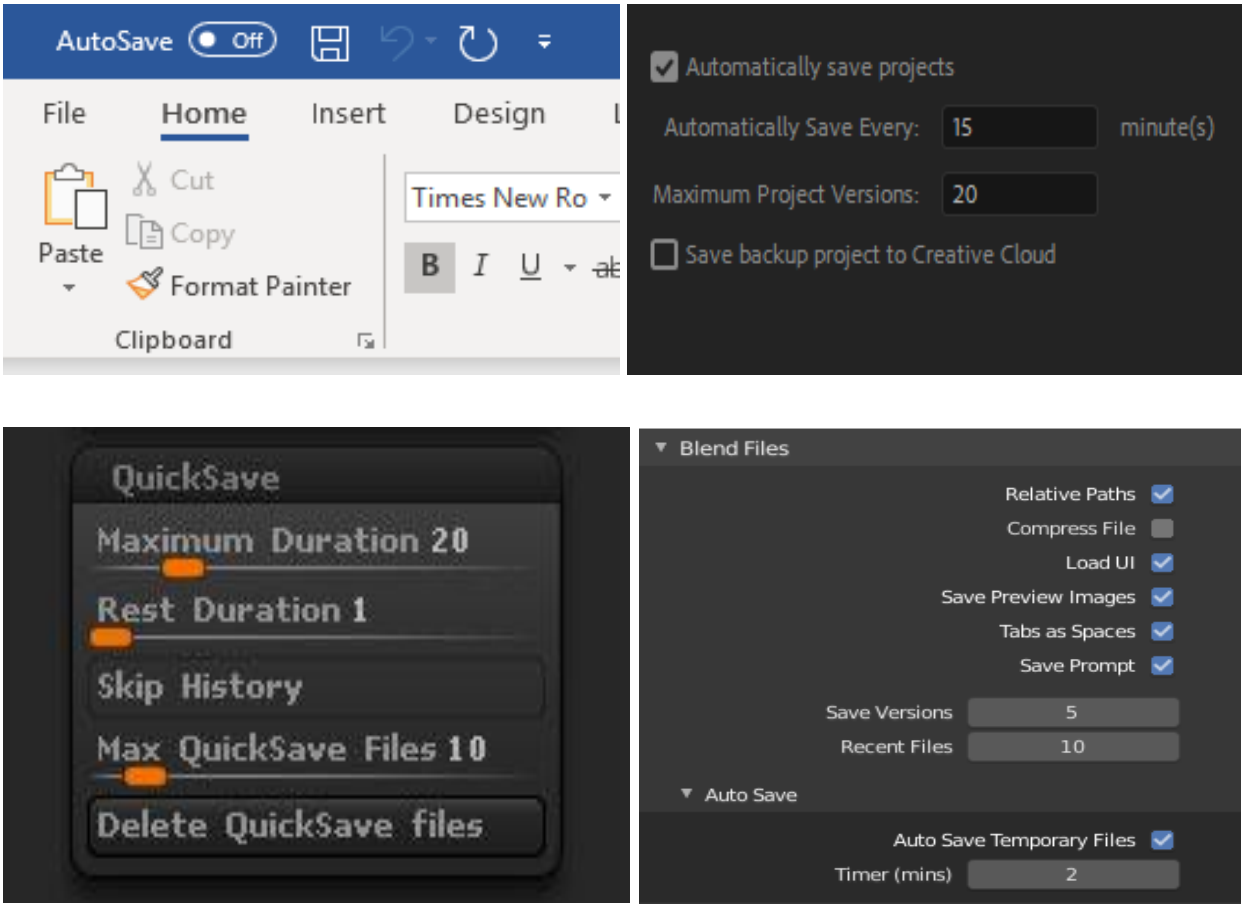


Figure 2.2.2 The top row represents the autosave option available in MS Word (left) and Adobe Premiere (right). The bottom row represents the autoversion settings in ZBrush (left) and Blender (right).

The main difference between these features is that autoversioning creates a duplicate of a file while autosave overwrites the same file. In the latter, however, there is no option to preview the previous stack of changes at different time intervals - the last saved version is the only copy of the work.

Autorecovery is another feature that is available in many applications that utilize autosave and autoversion to recover files. It can be manually set up by the user to avoid losing their work. In MS Word, the users can set up autorecovery for a certain number of minutes, see **Figure 2.2.3**, which allows the software to recover the autosaved file [71]. In Adobe Photoshop, the autosaved recovery files created by the autorecovery procedure are stored at some default recovery location on the hard disk (see **Figure 2.2.4**). These files are not stored for an extended period of time and are removed if Photoshop is opened after a crash or an unexpected failure and is then closed again without selecting the recovered file (unlike MS Word where the user can still get back to the autorecovered file) [72].

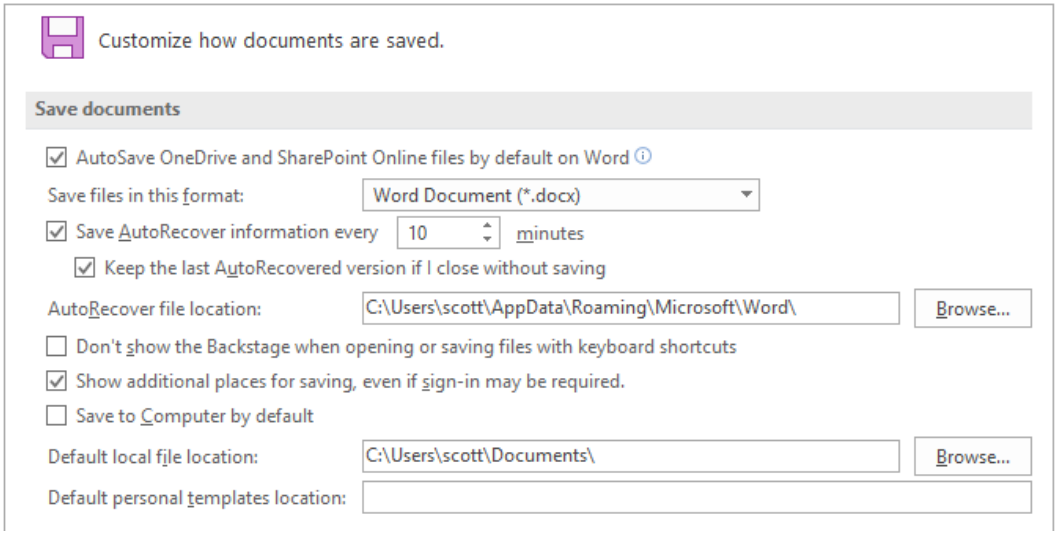


Figure 2.2.3 Autorecovery settings for MS Word.

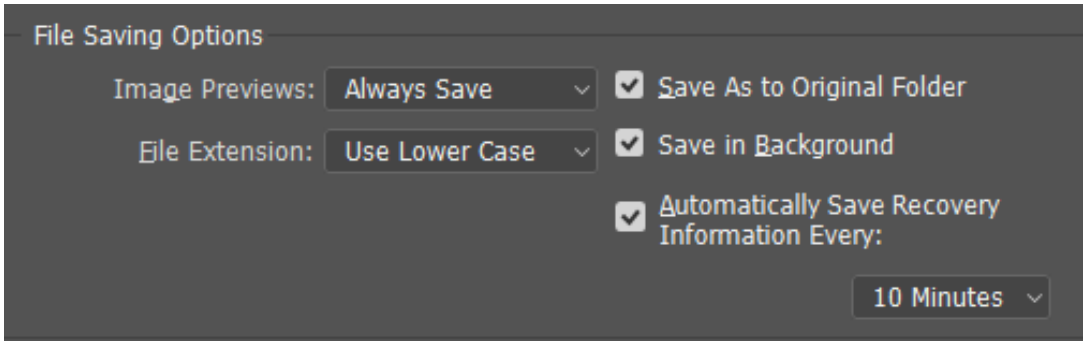


Figure 2.2.4 Autorecovery feature in Adobe Photoshop.

Many other applications perform the autosave operation in conjunction with other actions. For example, compiling the source code or building the current project in an IDE (Integrated Development Environment) also saves the modified or unsaved source code file automatically. This capability exists in IDEs such as IntelliJ, PyCharm, Eclipse, and Codeblocks [73,74]. Even the command line text editors like Vim and Emacs have autosave facilities to save files intermittently [75,76]. The save action can also be triggered on certain events by using hooks (scripts) or by setting some variables in the preferences. These temporary files, also known as swap files or recovery files, can be used to recover the lost work.

The autosave feature is also common in video games [77,78]. Many games have checkpoints which are locations in the game where the player respawns after they die. When the player reaches a checkpoint, the game is automatically saved. Players can also save their progress by manually saving the game, i.e., manually creating a checkpoint, either using a mechanism called 'quick saving' [78] by pressing a single keystroke or with a menu system (demonstrated in **Figure 2.2.5**). Players can overwrite their progress on the same saved game file or can create a new one which is then seen as a separate version of the saved game file.

Figure 2.2.5 Screenshot of a save menu in Assassin's Creed Odyssey [79].

2.2.3 Version Management Tools

In several job domains, version management is crucial for error recovery and backup. Version control systems and tools make it easier to manage the project files' history and modifications. They keep a separate version for each historical state or point of work. Tools like SVN, Git, Mercurial, Bazaar, and others [57–59,80] track changes in a project's files from their previously saved state and store the differences in a changeset (explained in **Subsection 2.2.3.1**) for tracking the changes over time. They keep track of all the files that were added, deleted, or updated in each commit (version), making it easy to see how the project evolved over time.

2.2.3.1 Basic VCS terminology

The following is basic terminology that is used in VCSs.

- *Repository*: The repository is referred to a local or remote server where all current and previous versions of files are stored.
- *Version/Revision*: A version is a state of the repository at any given time. It is also known as *revision* or *snapshot*.
- *Diff*: The difference between two different versions of a file.
- *Commit*: To commit is to write or save the changes to the repository. In some VCSs, this is known as *check-in*.
- *Changeset*: A collection of a set of commits that contains all the changes in a local working copy from the previous revision of the repository.
- *Checkout*: To checkout is to create a local working copy from the repository.
- *Branching/Forking*: A set of files may be branched or forked at a point in time to create a different copy of files independent of each other. An individual copy of a set of files is known as a *branch*.
- *Merging*: To merge changes from one branch to the other branch so that both sets of changes can be combined in a single branch.

2.2.3.2 A brief history of VCSs

There are three generations of version control systems [15]. The first versions of VCSs were all file-oriented, centralized, and mostly lock-based. SCCS (Source Code Control System) [41], developed in 1972 at Bell Labs for Unix systems, pioneered concepts and conventions like version numbers, major and minor versions (revision) separated by periods, which are still used in VCSs

today. RCS (Revision Control System) [48], developed in 1982, operates on revision groups (a set of documents called revisions) which evolved from each other under manual changes. It is a lock-based system, meaning only a single person can have a lock on a file and can make changes to it; the lock is released once the editing is done.

The second generation of VCSs were file-oriented and centralized as well but instead of using a lock-based approach, they used merging which was useful for collaborative development. CVS (Concurrent Versioning Systems) was released in 1986 and was widely used, with its terminology and conventions inspiring later VCSs. CVS uses a client-server architecture where the server stores the current versions of a project and its history while the clients 'check-out' a copy of the project, make changes, and later 'check-in' those changes. Some other second-generation VCS projects are Perforce (1995), Subversion (2000), and Microsoft Team Foundation Server (2005). Subversion also allowed non-text files (unlike all prior VCSs) and even tracked directory structure as each subdirectory of the Subversion working copy behaves like a CVS module and therefore can be checked out and updated individually. The atomic commits (single commit that pertains to one fix or feature), merge-before-commit model, and support for binary files made Subversion more flexible and powerful than other VCSs of the time.

The third-generation systems finally allowed merging and committing work as separate processes. These newer VCSs used decentralized and changeset-oriented approaches. Arch [81] and BitKeeper [82], released in the early 2000s, became influential in terms of features and inspired many new VCSs that are used by millions of users every day. Git, one of the most popular of VCS today [83,84] was developed in 2005 for the development of the Linux Kernel when BitKeeper, a proprietary software that they used to maintain their project at the time, revoked their free-of-charge status [85]. Another project that was created following the BitKeeper pricing issues was Mercurial [59]. Both Mercurial and Git provide high performance and scalability along with robust and fully distributed collaborative development, and advanced branching and merging capabilities. Both of these systems are available as software-as-a-service [86] by many hosting facilities. Git is widely used with platforms like GitHub, GitLab, and CloudForge while Mercurial is hosted at platforms such as Bitbucket, SourceForge, and Perforce. Many open source and private projects are hosted on these platforms with contributors all around the world [87]. Large companies like

Microsoft, Netflix, Reddit, Lyft, and Stackshare use Git for their own projects, and Facebook, Mozilla, and GNU Octave use Mercurial.

2.3 INTERACTION WITH HISTORY

2.3.1 Interaction History

The interaction history refers to a collection of all recorded actions performed by a user in an interactive system. For example, in an image editing software, to resize and color an image, a user executes certain actions in a sequence such as enabling ‘selection box’ to draw a rectangular area around the image, change image size, create a new layer on top of this image, then choose a fill color, and finally add color to the new layer. All these actions and events are stored as interaction history in a list.

Interaction history can help avoid performing the actions that the user may need to execute repeatedly. To automate these actions to perform a particular task, macros can be used which are a series of steps (or interactions) in the interface that can be recorded and performed later [88–90]. Most of the systems that support macros usually do proactive recording [30,42] where recording must be explicitly specified by the user (when to start and stop recording). On the other hand, systems like Smart Bookmarks do retroactive recording [25] where the system automatically stores the user interaction and creates macros from the subset of saved actions. Freeman and Balakrishnan [18] presented tangible actions that allow the system to re-perform the original interactions for a number of objects in the same interaction space using simple user gestures.

2.3.2 Comparing versions

Comparing individual versions is an important aspect in identifying the differences introduced during the project lifecycle and evolution. While text-based differences can be done using line based diff [24], the binary differences are a lot harder. However, doing a visual comparison between versions for complex objects can provide users with easy to understand views. Drucker et. al [14] presented new techniques and tools for doing a visual comparison between multiple

versions of a slide presentation. They identified subsets of slides that were similar across versions and presented an interactive visualization of the multiple versions of the presentation. Their assembly tool allows users to select and copy subsets of slides from one version into a new presentation (see **Figure 2.3.1**), which becomes an important feature in a collaborative environment. On the other hand, Chen et al. [6] presented a view authoring assistance system for 3D models to set views, paths, and surfaces across different model versions. Their comparison view function allows a modeler to view two different model versions in a side-by-side comparison with a version slider to change the version being displayed.

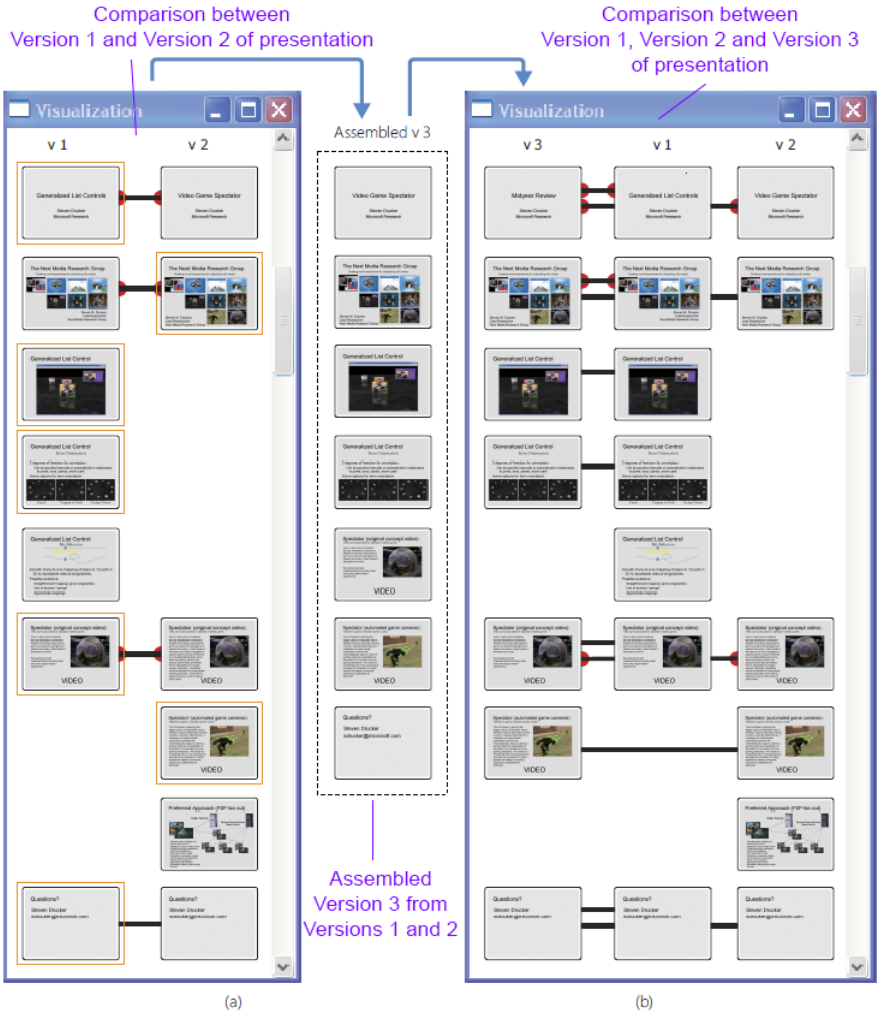


Figure 2.3.1 The assembled presentation from two separate versions of slides and its comparison with the other two versions [14].

2.3.3 Visualizing interaction

Visualizing interaction for individual objects with a technique like Phosphor [1] can help in improving user's ability to understand changes in a user interface. Scented widgets [49] provide visual navigation cues by scenting (augmenting with navigation cues) interactive widgets to navigate information spaces (see **Figure 2.3.2**). A study of this technique showed that scented widgets helped the user make up to twice the number of unique discoveries while navigating unfamiliar data. Guimbretière, Dixon and Hinckley [20] presented an analytical tool called 'ExperiScope' that creates a timeline of interaction histories (capturing data and identifying patterns) to aid the analysis of empirical studies. Cutler, Gadhave, et al. [7] developed a library called 'Trrack' that provides web-based provenance-tracking for the purpose of action recovery (undo/redo), reproducibility, collaboration, and logging. It allows for visualization of and interaction with the non-linear provenance graph using TrrackVis (customizable visualization library to be used with Trrack).

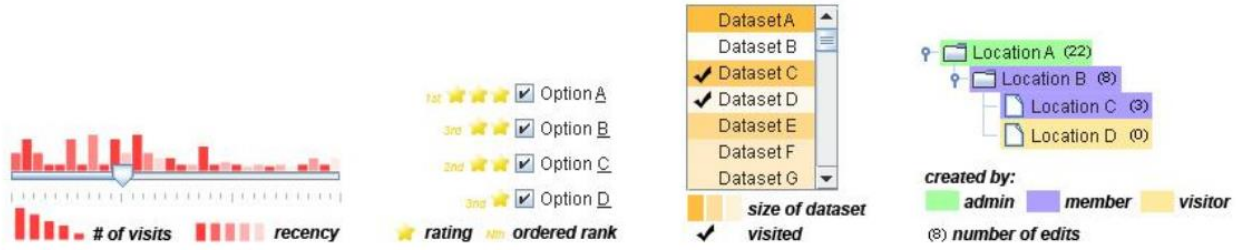


Figure 2.3.2 Examples of Scented Widgets with various encodings [49].

Similar to timelines, storyboards are a great way to visualize the evolution of a design, where each frame of a storyboard can be used to represent a state of the design at a fixed point in time. Zhao, Zhenpeng, et al. [51] developed 'Sketcholution' - a method for creating visual histories for hand-drawn sketches using a bottom-up agglomerative clustering mechanism to group adjacent frames based on perceptual similarity. This allows the designers to visualize the history of a sketch which can provide an insight into the creative process behind the design. Due to the lack of need to support forking the previous states, no hierarchical visual representation was required and therefore, a linear list was used to represent the evolution that creates a storyboard of sketches (**Figure 2.3.3**).

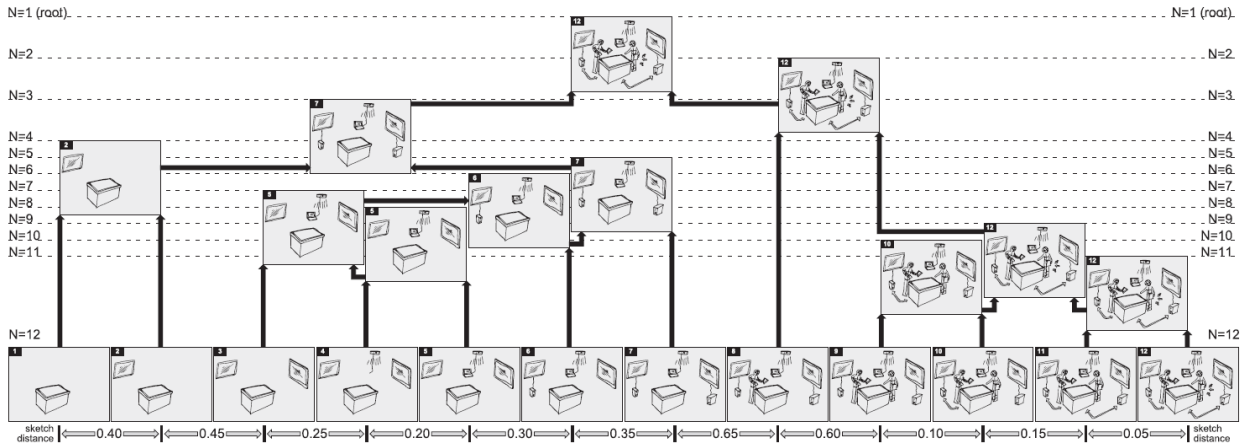


Figure 2.3.3 The evolution of a sketch in a linear layout and the Level of Detail tree created from the dendrogram (resulting from the cluster analysis) [51].

2.3.4 Versioning for Interactive Systems

The revision (version) management of the assets in a 3D environment is critical for artists and designers working by themselves or in a team environment. Doboš & Steed [13] presented a 3D Revision Control Framework that provides asynchronous revision control of 3D assets using NoSQL Database. Their tool allowed version tracking, differencing, and conflict resolution during merging while working with the notion of explicit and implicit conflicts [11]. Another tool that helps in diffing and merging polygonal 3D meshes is MeshGit [10] where the conflicting merges are handled gracefully by allowing users to choose the version that they want to include in the merged version. In **Figure 2.3.4**, the spaceship model (*original*) has two derivative models (*derivative a* and *derivative b*). In *derivative a*, new top part is added to the model, and the base of the spaceship is also expanded. On the other hand, wings are added in the *derivative b*. However, these edits are conflicting when user tries to merge them together. MeshGit resolves this by presenting user with three different versions of the spaceship model. The first version includes merging of only non-conflicting mesh parts, therefore the merged model has only the top part added to the resultant model. The second version consists of all non-conflicting mesh parts and the conflicting part of *derivative a* of the model. And, the third version is a combination of non-conflicting mesh and the conflicting mesh from *derivative b* of the model.

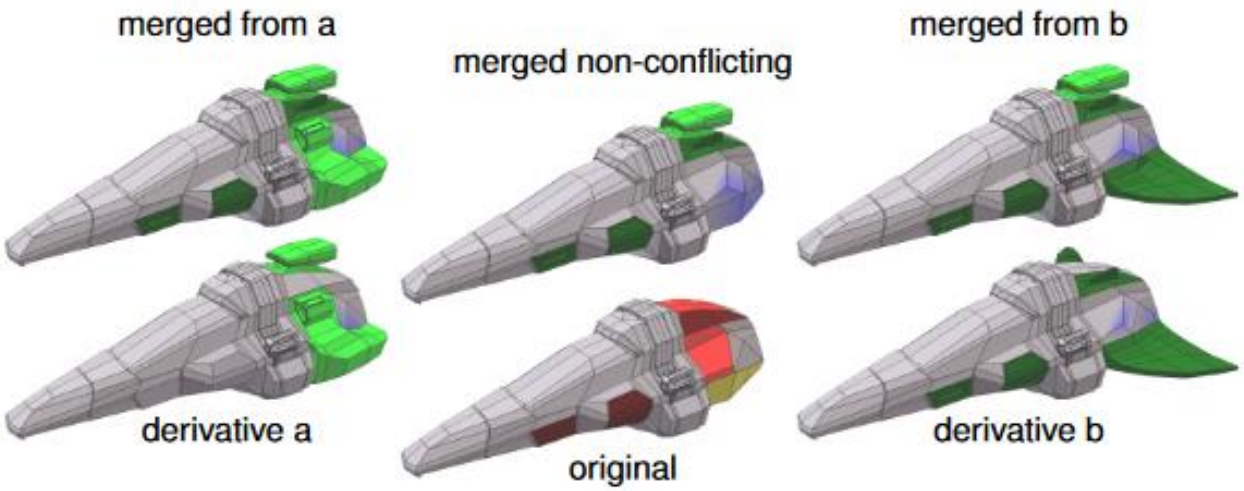


Figure 2.3.4 Bottom row represents the three different versions of the spaceship model. Top row represents how the MeshGit detects non-conflicting and conflicting changes and suggests three possible ways to resolve the conflicted merge [10].

Revisiting the historical states of a system has also been explored by various researchers previously [31,37,40,45]. Systems like Chronicle [4], Sketch-Sketch Revolution [16], and 3D Timeline [12] record workflow histories that can be viewed and played back later, while an interactive system like MeshFlow hierarchically clusters user operations and then ensures that the users can focus on a relevant part of the design process [9]. MeshFlow summarizes the workflow to discrete steps because of fixed clustering rules whereas another workflow tool called 3DFlow works on a continuous summarization and summarizes the workflow down to a single step [8]. With a large number of workflows available, it makes sense to do workflow comparisons. Delta [27] is one such tool that helps the users to identify the tradeoffs between workflows by comparing them based on the set of attributes that users focus on the most.

2.3.5 Exploration of Alternatives

Versions can also help in exploring a parameter space by creating multiple alternatives for a range of parameters that can be used for dealing with some ill-defined problems (problems with poorly defined operators and goals) such as design tasks [33]. There are many systems developed that can

create parameterized alternatives. For example, the Design Gallery interface [28] provides the user with a selection of parameters that can be varied to automatically produce perceptually different graphics or animations. The example interface, shown in **Figure 2.3.5**, shows alternative designs available for setting light parameters such as light type and placement in the scene. These designs were created during a "dispersion" step in which the system selects an appropriate subset of input vectors from a random sample over the input space, i.e., only a subset of total thumbnails is approved that are likely to be of user's interest. Design galleries of animations can also be constructed using a variety of dispersion methods. Similarly, Side Views [46] provides an on-demand dynamic preview of commands applied to the copy of current data allowing users to look at several possibilities without committing to any one course of action. The parameter spectrums display a series of previews across the user-defined range of values for a parameter.

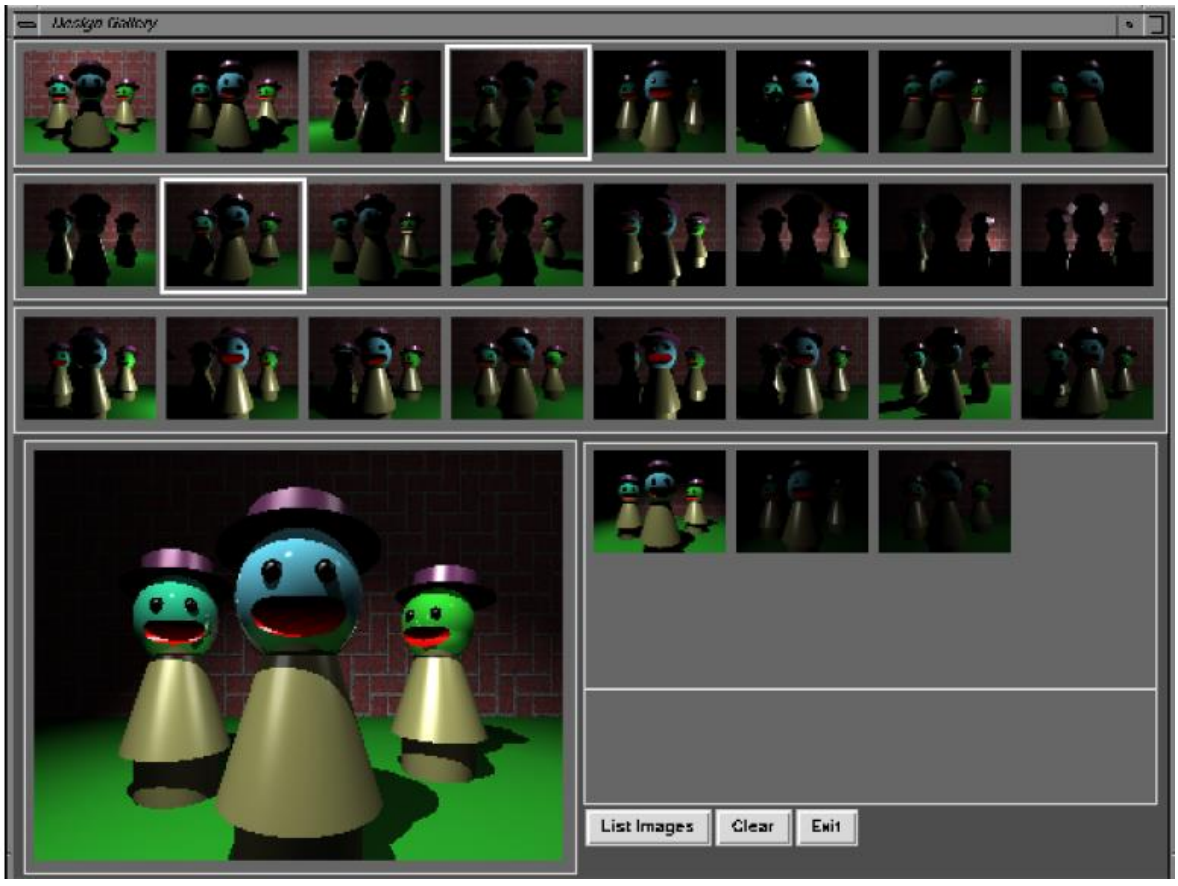


Figure 2.3.5 Design Gallery for Light selection and placement [28].

However, unlike SideViews and Design Galleries, Parallel Paths model of interaction [47] allows users to generate, manipulate, manage and compare multiple solution variations. Users can create new solution variations before, during, and after invoking a command (interaction action that can alter the variations). Since each variation is directly embedded in the same workspace, it allows users to operate on these variations simultaneously or individuals as necessary. Parallel Pies tool, an image manipulation application that implements Parallel Paths' principles, allows a command to be applied on individual variations as well as simultaneous variations (see **Figure 2.3.6**).

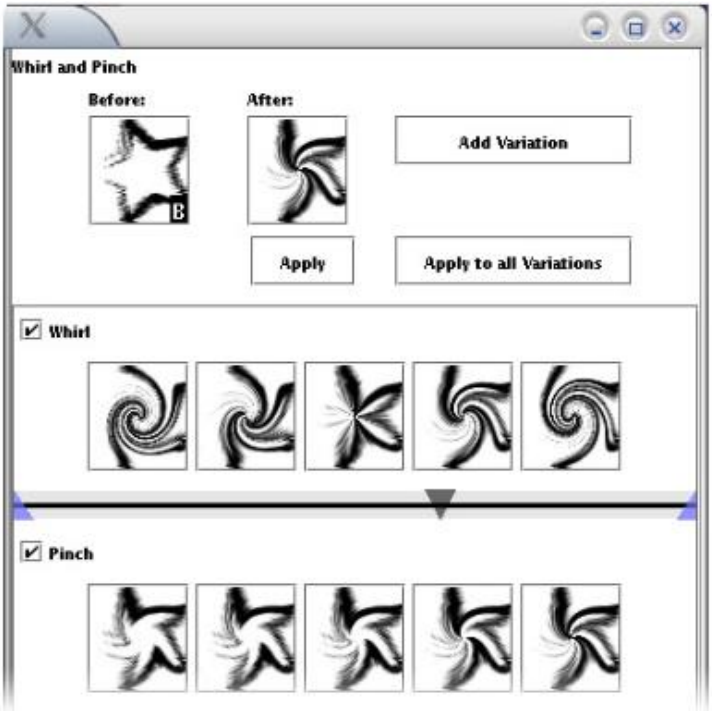


Figure 2.3.6 Parallel Pies tool's command dialog box that can apply current result as a new variation (top right button), or apply commands to all variations (bottom right button) [47].

A similar approach was followed by Zaman, Loutfouz, et al. [50] in their graph-based generative design tool 'GEM-NI' that allows designers to explore different alternatives. Their system supports parallel editing (allowing users to edit multiple alternatives simultaneously), merging alternatives, branching, and comparisons that can help users create multiple alternatives for initial designs. **Figure 2.3.7 (a)** shows the dialog box for creating an alternative from history, where a user can branch out from the currently selected state (in the list on the left-hand side) by pressing a "Creating

Alternative" button that creates a new branch by cloning the history stack using skating (duplicating the current state and return to the previous state non-destructively) [47]. The design gallery, see **Figure 2.3.7 (b)**, shows several possible alternatives based on parameters that can be varied, and their results are obtained based on a Cartesian product.

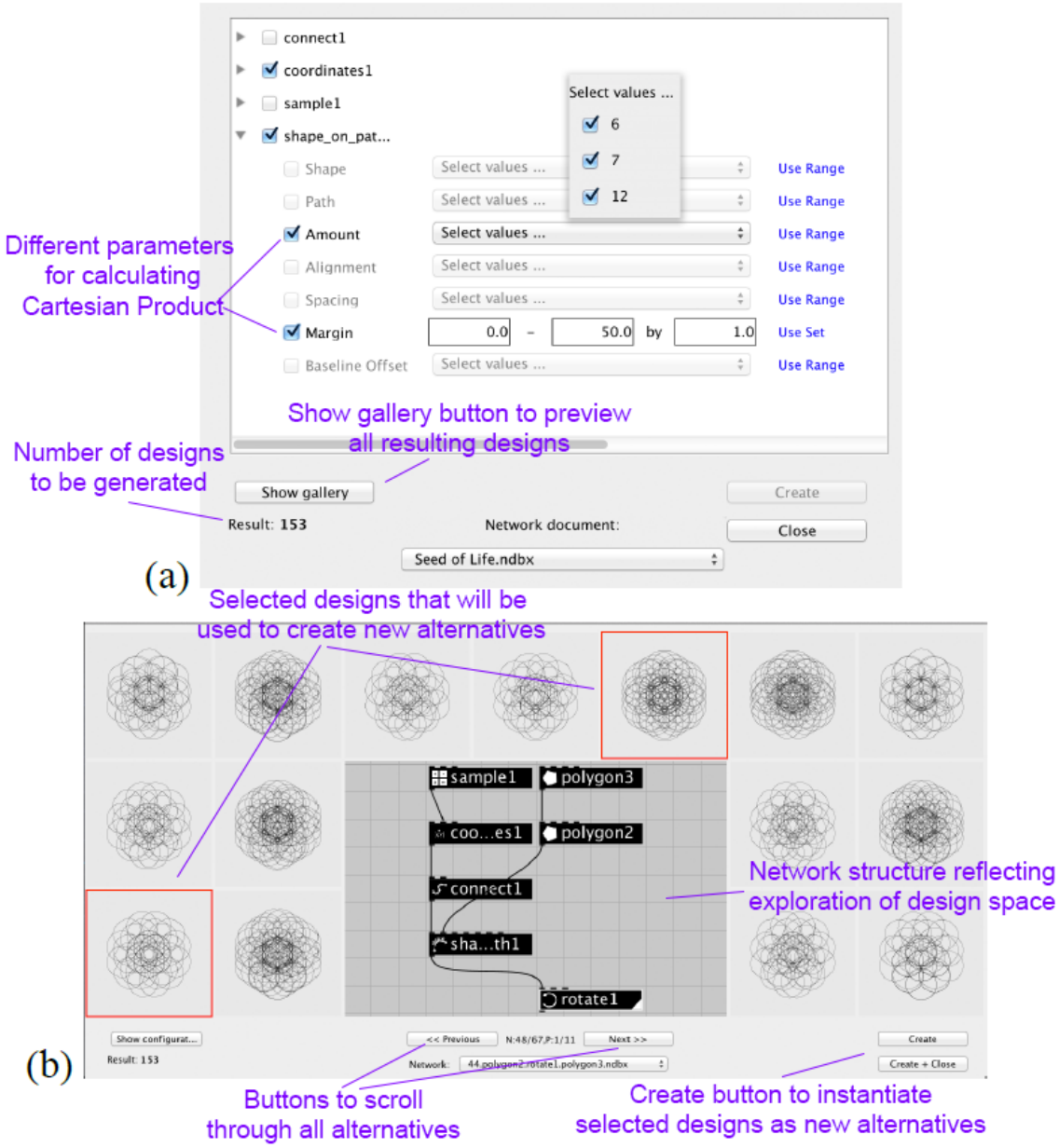


Figure 2.3.7 GEM-NI: (a) Cartesian Product menu for parameter selection, (b) Design gallery for alternatives [50].

O'Donovan, Agarwala and Hertzmann [34] presented a system that provides two types of future versions for interactive layout: refinements and brainstorming suggestions. The system provides refinement suggestions to improve the current layout while brainstorming suggestions help explore alternative designs of the layout (**Figure 2.3.8**). Each alternative layout's style differs from the other according to layout suggestions such as a change in text, size, alignment, symmetry, etc.

Figure 2.3.8 DesignScape Interface: Simple Layout Editor (center), Refinement suggested Layouts (left) and Brainstorm suggested Layouts (right) [34].

2.3.6 Causality Framework

Nancel and Cockburn [32] presented Causality framework which is "a conceptual model of interaction history that keeps track of past states and commands of the edited artifact in the form of a causal system". Causality framework allows safe, flexible, and understandable modifications of history. The commands used in the workflow can be modified, duplicated, and re-ordered, allowing users to easily recover from their previous mistakes or rethink their documents without losing their previous work. Moreover, users do not need to form any strategy for completing their work before they actually begin.

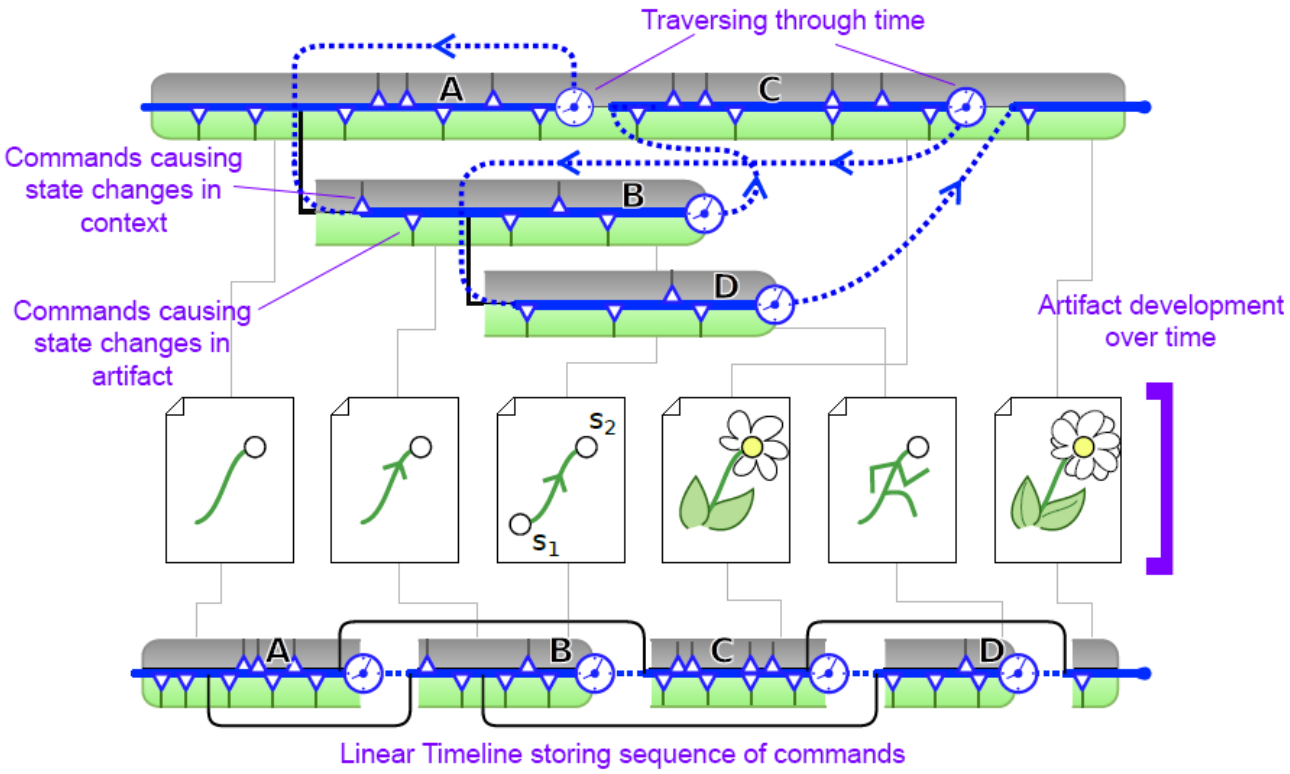


Figure 2.3.9 Causality: Represents branching chronology (top 3 rows), linear timeline (blue lines), application context (grey bar), artifact state (green bar), commands (blue triangles), time-travels (clocks and dotted lines), and branching (black strokes) [32].

There are five main components in Causality that help in modeling temporal interaction in the system: artifacts, context, commands, linear timeline, and branching chronology (see **Figure 2.3.9**). The artifacts are the states of the work object stored at discrete points in time that represent the object's development over time. The context is the state of the application at any given time in history that stores all the system and parameters settings along with the state of the interaction controls. The commands are the actions, that when performed cause changes in the state of artifacts or contexts. Each command has a reference to all the elements of the artifact and the context that it uses and affects. The Linear timeline contains all the commands as a sequential list of events that are executed in real-time during the lifecycle of a project. It even stores the operations that alter the history. The branching chronology is a tree-like virtual chronology of artifacts and contexts

(before a state was modified) where each branch is created when a user traverses through time (using linear timeline) and modifies some artifact or context at a given point in time.

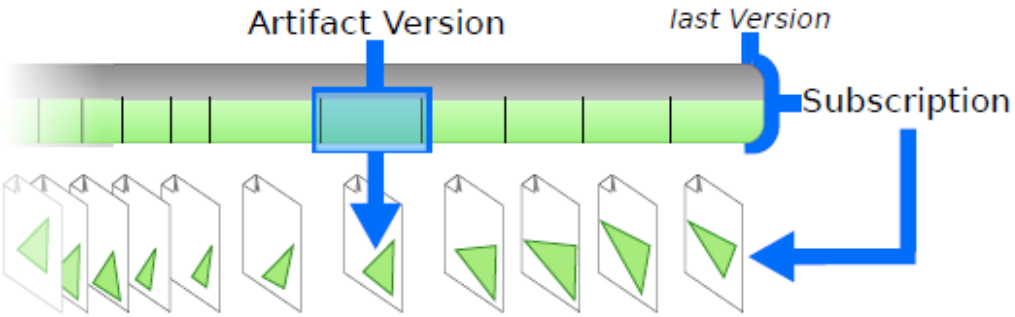


Figure 2.3.10 Artifact versions and subscription in the local history.

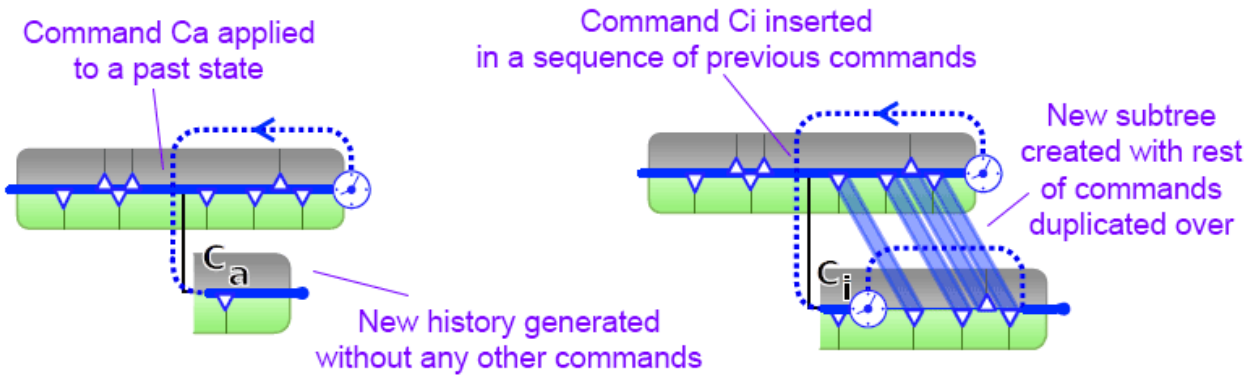


Figure 2.3.11 Applying command to past state generates a new state of history (left) and inserting command in an existing list of commands generates a new subtree (right) [32].

The command objects in Causality use references to the artifacts or contexts instead of their respective copies. These references have been categorized as versions and subscriptions. A version is a fixed position in a history representing the state of a work object (see **Figure 2.3.10**) whereas a subscription is a dynamic link to the current (or latest) version of an object in real-time or history. Any changes to the indicated version will be reflected on the subscription as well. Whenever a new command is applied to a saved state in history, a new child of a node is added to the history tree.

Modifying the historical state generates a new subtree which is a copy of original history but with the intended changes, leaving the original set of history unaltered. For example, as shown in **Figure 2.3.11**, if a command C_a is applied to a past state, a new state of history is generated without the need to copy over the rest of the commands in the sequence after that command. However, if a command C_i is inserted in an existing sequence of commands, a new subtree is generated and the rest of the commands are replicated from the original branch, User can see the effect of this command by traveling to the end of this new branch.

2.4 VERSIONING IN CURRENT SYSTEMS

Current cloud applications like Google Docs, Google Sheets, or Google Slides do auto-versioning of documents [91], while other applications such as Overleaf allow users to save versions of their documents explicitly. There are also other tools such as Dropbox, OwnCloud, and OneDrive that allow users to create backups of their files that can be restored if needed. There are also GUIs for current VCS applications such as SourceTree and Github Desktop for Git, TortoiseSVN and SmartSVN for SVN, and MacHg and TortoiseHg for Mercurial. These are standalone applications that do not come integrated within any software environment and may or may not work well with all interactive applications and their documents.

Google Docs employ a linear model of versioning - a stack-like structure as seen in the right-hand panel called Version History (see **Figure 2.4.1**) while MacHg provides users with an ability to use a non-linear hierarchy to store versions with a graph like structure that represents multiple branches (see **Figure 2.4.2**) for the parallel paths for different versions.

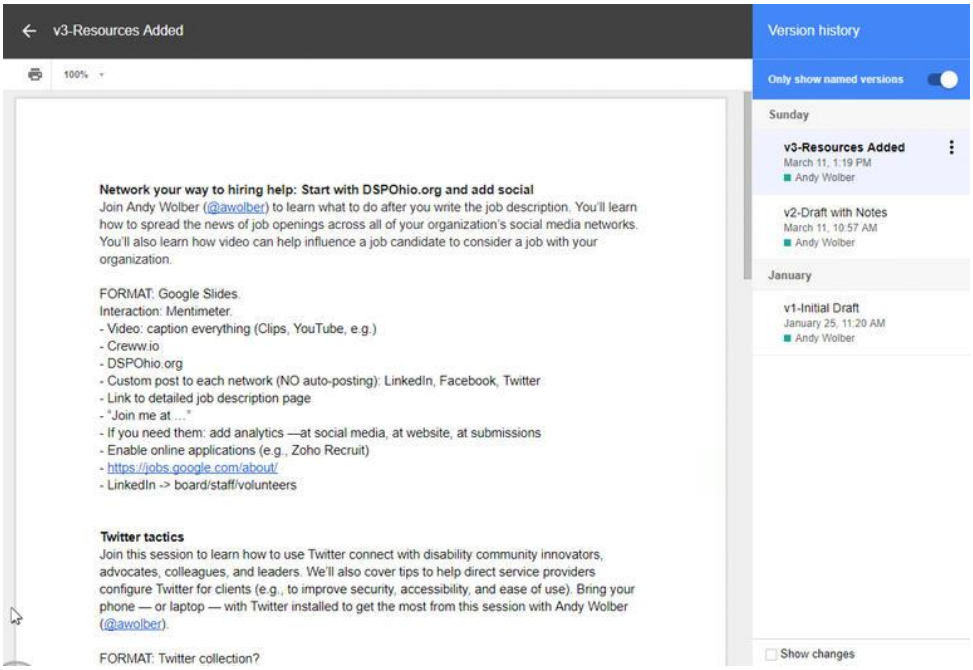


Figure 2.4.1 Version History panel (right hand side) for Google Docs.

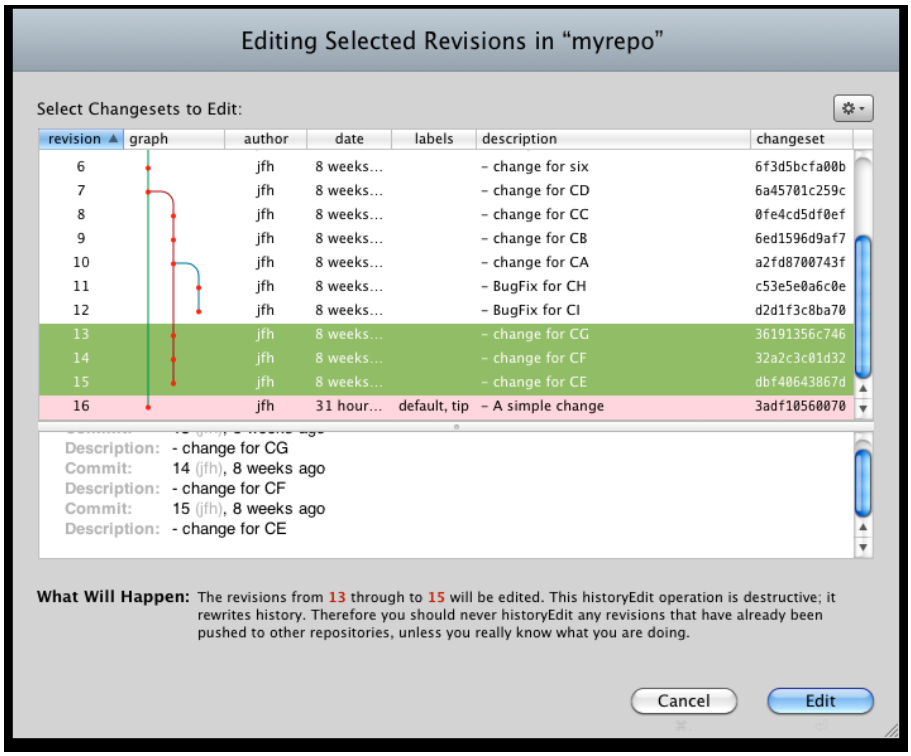


Figure 2.4.2 MacHg screenshot for revision graphs [92].

Another design tool called Abstract [93] is a good example of managing versions and avoiding painful merge conflicts with a focus on collaboration. It keeps track of changes in the sketch file and prompts a user to preview and commit them to the current branch on which they are working. In a collaborative environment, other users can review those changes and can even add their comments. Abstract is another standalone application that may improve and ease users' experience for collaboration and versioning. It does not necessarily ease the process for beginners who are not familiar with the process.

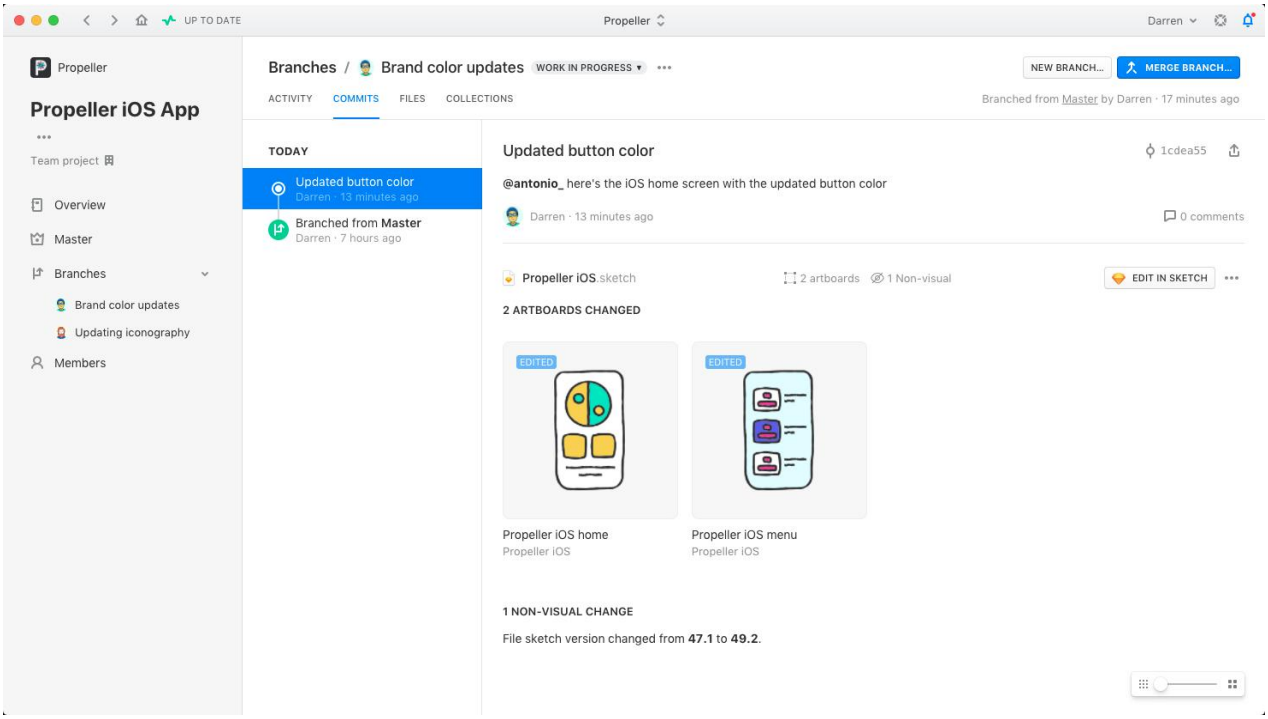


Figure 2.4.3 Abstract Tool screenshot [93].

Visual Studio 2019 introduced Time Travel Debugging (TDD) which is a debugging solution that allows users to record the execution of code in an application or a process and then replay it backwards and forwards [94]. Typical debuggers allow users to go start at a specific point in time and only go forward, however, TDD allows going backward in time to understand the conditions better. The recording of events and data in a discrete time interval creates a timeline that works like a version history that can be accessed at a later stage. The TDD contains the exact details of the execution path that led up to the final failure or bug which is crucial to isolate and identify the

problems that could occur in the production environments. In addition to allowing users to *step forward (p)*, *go forward (g)*, and *trace forward (t)*, TDD allows them to *step backward (p-)*, *go backward (g-)*, and *trace backward (t-)* as shown in **Figure 2.4.4**.

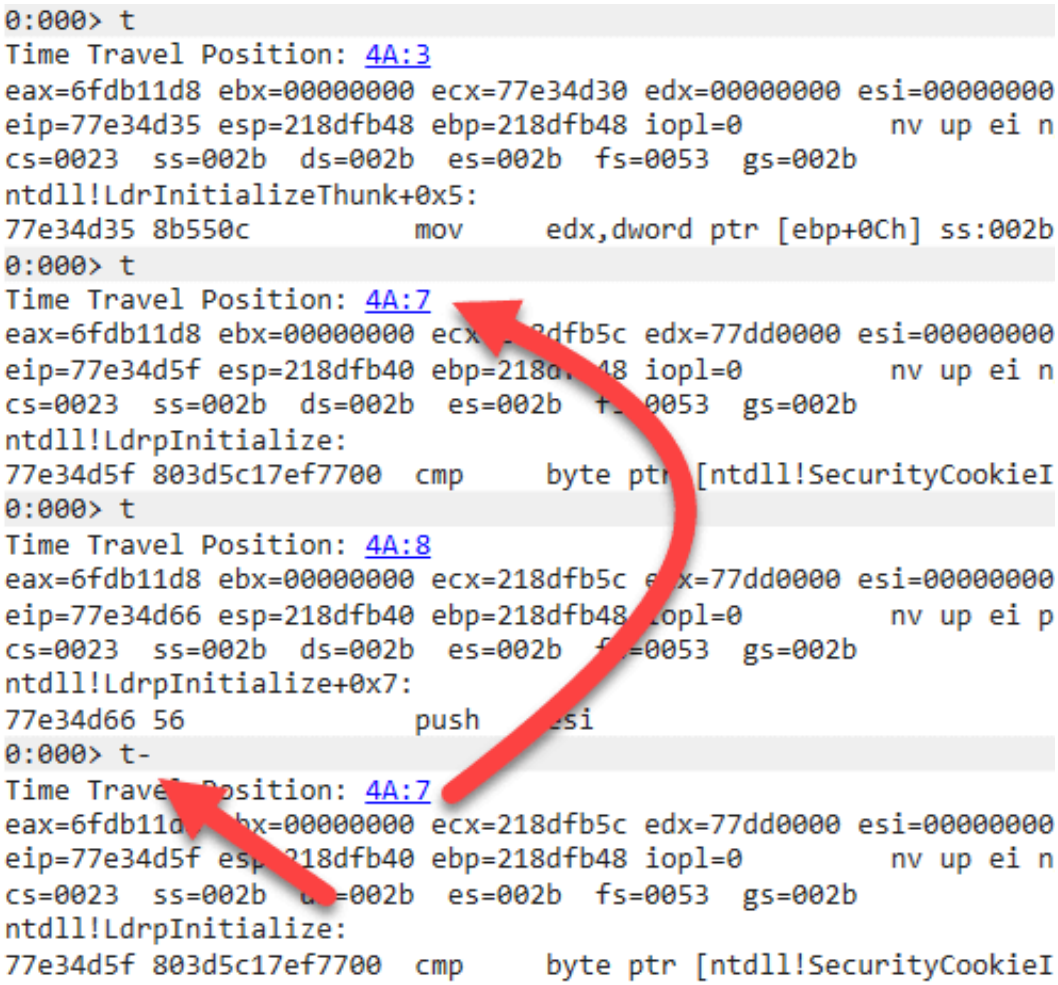


Figure 2.4.4 Time Travel Debugging [95] allows command *t-* to move the debugger back to previous position *4A:7* indicated with red arrows.

2.5 OTHER USAGES

From internet browsers to operating systems, history can be recorded and presented to the user at a later time. Browser history [96] and session management [97] can provide users access to their

past actions and work in web browsers. Using software and tools like Windows Restore Point [98], Time Machine [99], TimeShift [100], etc., users can save a snapshot of their operating systems at any given time to work as a backup solution. Tools like Windows Timeline [101] allow users to view activities they have been working on in the past few days to weeks. Solutions like these are useful in keeping track of interactive history.

We also see creative use of previous states of the system in other types of media like video games. Games like Super Meat Boy [102] and Braid [103] leave historical evidence of the game played by the players. In Super Meat Boy, players make mistakes and their character, red-colored cube, dies a lot of times and with each death, the character leaves a trail of red-colored stains in the environment which gives an indication to the player where they have died previously. While Braid is a puzzle solving platformer game that relies heavily on the ability to rewind time and undo actions even if the player dies. The player can switch to any previous attempt to solve the puzzle and subsequently creates another attempt in the game and even see multiple realities.

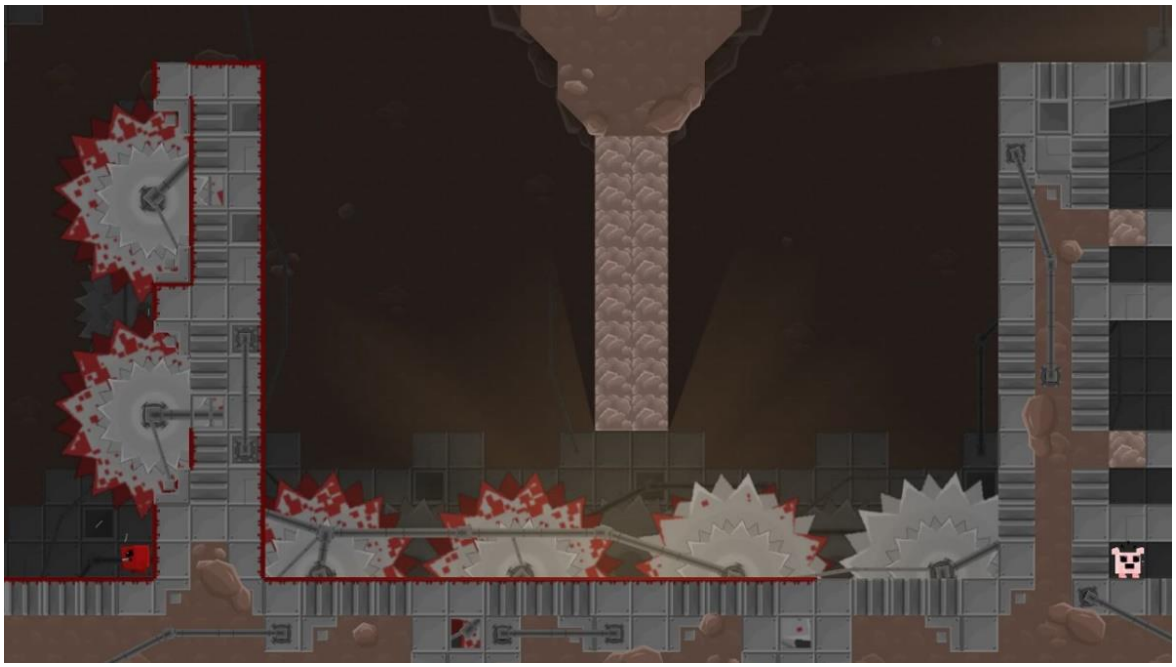


Figure 2.5.1 Screenshot of the gameplay of Super Meat Boy [104].

Figure 2.5.2 Screenshot of gameplay of Braid [105].

Some games like Prince of Persia: The Forgotten Sands, Timeshift, Super Time Force Ultra, Blinx: The Time Sweeper, etc. [106] actually reverse time in the game so the players can solve puzzles or reset their progress if they find themselves in a situation where they die often.

Figure 2.5.3 Screenshot of the gameplay of Prince of Persia: The Forgotten Sands [107].

Figure 2.5.4 Screenshot of gameplay of Super Time Force Ultra [108].

CHAPTER 3

A SURVEY STUDY OF VERSIONING IN INTERACTIVE SYSTEMS

In this chapter, we present a survey we carried out to understand versioning for users of interactive systems. We discuss various circumstances that lead to the creation of new versions and reasons to revisit or revert to previous versions. We also discuss the naming conventions that respondents used for versioning their files that can help keep their versions manageable and comprehensible.

3.1 ONLINE SURVEY

To understand the circumstances for creating new versions and the reasons to go back to previous versions, we designed a questionnaire to ask users about their versioning practices. We deployed a web-based survey (see **Table 1** and **Appendix A3-A5**) asking computer users about their understanding of versions and how they utilize versioning in their projects.

3.1.1 Survey Content

After a set of initial questions about demographics and amount of computer use, the survey asked participants to list the three software programs or tools that they use most frequently. We wanted to see if participants were familiar with existing versioning systems, so we asked them to choose from a list of some common versioning and backup tools. We also asked that they specify the naming convention they use for their versions.

We then asked them three questions related to their versioning practices: *why they create a new version* (circumstances when a new version creation is needed), *why they use previous versions* (reasons that prompt users to revert to previous versions of their work), and *how often they use previous versions*. We left it up to the users to decide whether to consider the nature of their workflow (single-user or collaborative) while answering these questions.

S.No.	Question
1.	Gender
2.	Age
3.	On average, how many hours do you spend on desktop/laptop computers per day?
4.	Specify the top 3 software programs/packages that you use. For instance: MS Word, Photoshop, Autodesk Maya.
5.	Are you familiar with any of the following versioning tools/systems/softwares?
6.	Are you familiar with any of the following backup tools/systems/softwares?
7.	What kind of naming conventions do you follow when you create multiple versions of files? Example: web_design.psd, web_design2.psd, web_design3.psd
8.	In what circumstances do you feel the need to create a version of a file that you are working with?
9.	How often do you go back to the previous version of the file you create when working on a project?
10.	What are the reasons that make you go back to the previous version of the file you created/saved when working on a project?

Table 1 Survey Questions

3.1.2 Participants

We deployed our survey on SurveyMonkey and shared the survey link in several groups (called subreddits) on the social media site reddit.com [109] to recruit participants. These subreddits were

popular groups for designers and artists who work with applications such as Photoshop, Blender, 3DSMax, and various game engines. Along with these creative subreddits, we also posted in a general computer user subreddit that is read by users who work with interactive applications such as MS Office, text editors, and IDEs in their daily work.

The survey was completed by 84 people (see **Table 2**). The participants' age ranged from 18 to 63 years (with a median age of 24). All of the participants reported that they used computers on daily basis for an average of 9 hours. There were 71.43 percent males and 26.19 percent females among the 84 respondents, with the remaining percentage preferring not to disclose their gender. Respondents spent an average of 7 minutes completing the questionnaire.

Total participants (n = 84)	
Age	18 – 24 (50% = 42) 25 – 34 (27.38% = 23) 35 – 44 (17.85% = 15) 45 – 54 (1.19% = 1) 55 and older (2.38% = 2) Undisclosed (1.19% = 1)
Gender	Male (71.43% = 60) Female (26.19% = 22) Undisclosed (2.38% = 2)
Average hours per day spent on computers	1 – 5 (16.66% = 14) 6 – 10 (52.38% = 44) 11 – 15 (27.38% = 23) 15+ hours (3.57% = 3)

Table 2 Participant Demographics

3.1.3 Data Analysis

All participants completed the survey and no participants were discarded. For questions related to participant demographics, we present the results as percentages (see **Table 2**). The responses from the respondents' list of their top three software applications confirmed that at least one of their applications can be augmented with versioning or backup tools.

After examining responses for the questions related to familiarity with available versioning and backup tools, we found that all except two respondents have either worked with or heard about at least one of the tools as shown in **Figure 3.1.1** and **Figure 3.1.2**.

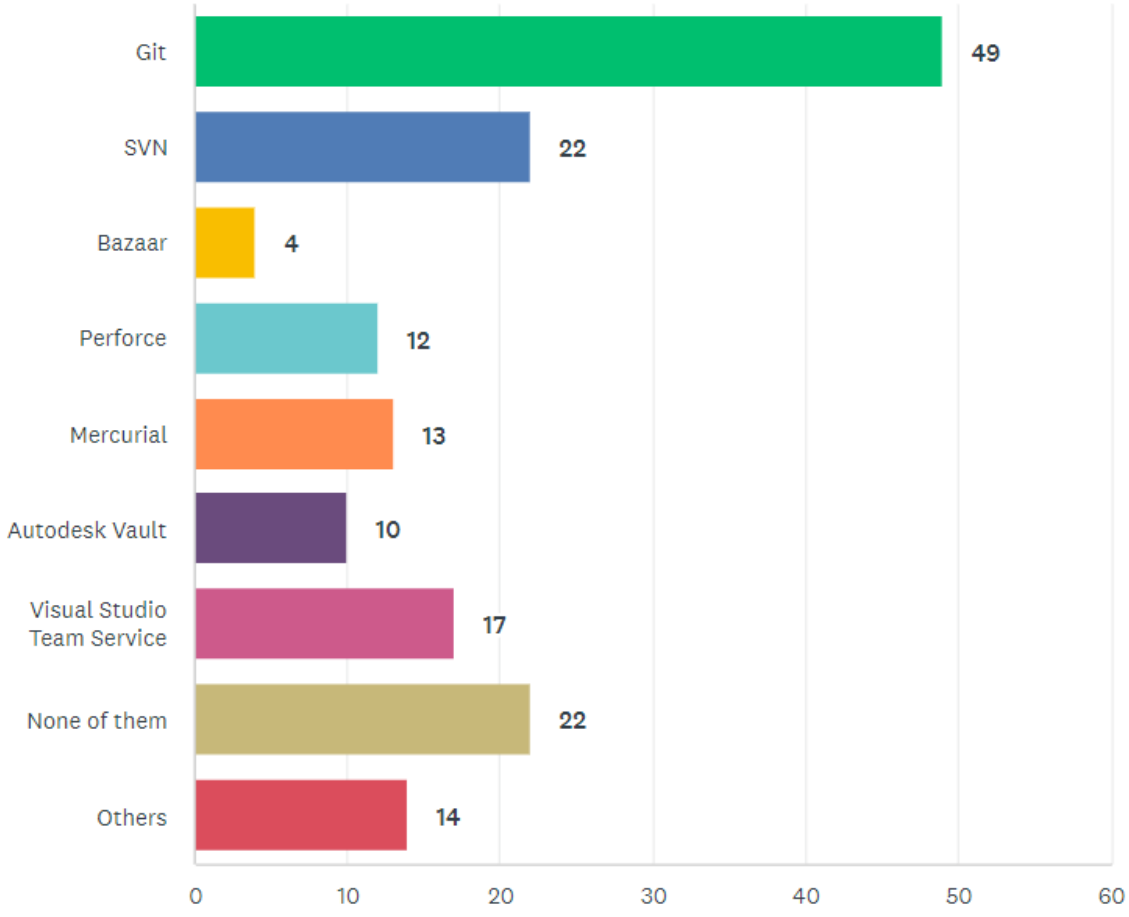


Figure 3.1.1 Number of participants stating that they were familiar with specific versioning tools.

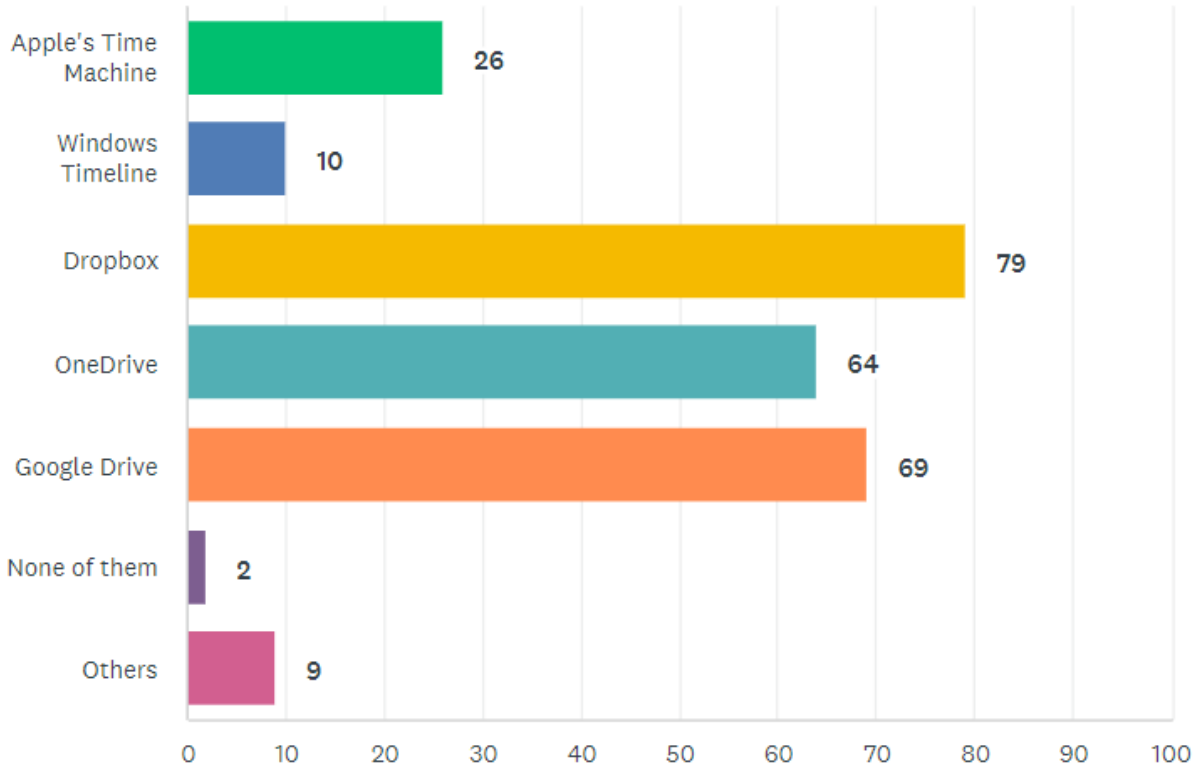


Figure 3.1.2 Number of participants stating that they were familiar with specific backup tools.

We performed inductive thematic analysis [2] on all users' responses for the two important comment-based questions of our survey (see questions 8 and 10 in **Table 1**). First, we went over the data for each question several times to broadly categorize the users' responses by color coding and adding labels to it. This helped us identify initial themes in the data. After thoroughly reviewing these initial themes, we were able to name and define the six main circumstances that lead to the creation of a new version of the current work (**Subsection 3.2**) and seven reasons to go back to previous versions of the work (**Subsections 3.3**). Performing this inductive thematic analysis on our qualitative survey data helped us formulate how users understand and utilize versioning in their daily workflow.

3.2 CIRCUMSTANCES TO CREATE NEW VERSIONS

We found six main circumstances that prompt users to create new versions of their work.

3.2.1 Substantial Changes

Substantial changes are any big or significant changes in a current file that make it difficult to revert to its previous state using simple undo because of the multiple steps required to reach that significant change state. For example, if a user is working on a 3D model of a character, there are various steps involved in the character design process such as modeling, sculpting, re-topologizing (the process of redefining high-resolution 3D models to lower resolution models), UV-mapping (the process of projecting a 2D image to a 3D model's surface), and texturing. Each step could take up to hundreds of actions that are required to be performed by the user. Therefore, having these substantial changes in project files encourages users to create new versions at important points during their project's lifecycle in case they do not like where their project is heading.

31 out of 84 participants mentioned that they create a new version when they plan to make a substantial change in their project or they hit a significant milestone in their project. One of the participants commented on their workflow by stating that,

"*[I create a new version] whenever I'm satisfied with a version, but I have to improve parts of it. For example, a character has a final version of a face, but still needs work on the body."*

3.2.2 Avoid losing work or progress

People create a new version to have a backup of their current file to avoid losing their progress in case of errors and mistakes that can occur due to human or machine faults. They create a copy of the original file in case their current progress becomes irreversible or incorrect. For example, if a user merges two layers in Photoshop and continues to work on this merged layer, then at a later stage, they will not be able to get their original layers back in their unmerged form without losing the newer changes done after the merge operation.

Three participants informed us that they can revert to backed-up versions of their files if they accidentally overwrite their original work with something they were not supposed to do. Twelve participants mentioned that they create versions for their working files to avoid losing their work due to software failure or file corruption. One of the participants noted that,

"*I never work on the original, so the first thing I do is create a copy. I usually work with neurons in a brain slice, so I make a file containing everything, then split it up into individual neurons (each has a file).*"

3.2.3 Testing experimental changes

Experimental changes are changes that are temporarily introduced in a project to test some new functionality or some content that can later be either discarded from a project or accepted and merged in the final stage of a project. For example, during game development, developers may implement a new superpower for a game character that can be playtested and reviewed that may or may not stay in the final game.

Eighteen participants told us that they create new versions when they are unsure about their changes or want to see how their changes affect the project's output. One of the participants mentioned that they would create a new version "*if the old version worked and I continued with improvements which haven't been tested to work well yet.*" Another participant remarked about experimenting with the changes as,

"[I create a new version if] I learned new data since the first version, but am unsure if the new data is valid, so I want to experiment."

3.2.4 Exploring alternatives

Alternatives are duplicates of a document that are slightly different from the original. Nineteen participants mentioned that they create separate versions when they are exploring different options or alternative ideas in their work. When they are creating slight variations to an original design or they are experimenting with different styles or techniques but still want to keep the original work, they create multiple files/versions. For example, a designer working on user interface mockups

might need to create multiple variations of the same design to avoid overwriting the original design. The separate versions not only allow the users to go in different design directions but also enable them to compare these variations. Remarking about changes and the ease to go back to their original work, one participant told us,

"*When the piece doesn't follow a strict layout and there might be many ways to place the content, I'll save a new version every time something moves, making it easy to go back instead of having to redo steps.*"

3.2.5 Collaborating with other users

Collaboration is the process of working on the same project with multiple people. When collaborating on a project with others, eleven participants mentioned that they create a separate copy of their work to keep track of the changes they share. This allows them to easily understand what changes were contributed by others and compare the before and after states of the shared file. One participant mentioned creating a new version to make changes in the file according to the feedback they received. While another remarked that,

"[It allows] incorporating edits/notes/comments from multiple reviewers [and then] branching a file because I'm not sure I'll like the direction it goes."

3.2.6 Time-based versioning

Time-based versioning means creating versions of files periodically or intermittently. Users can save multiple versions of their work periodically or allow the systems to do auto-versioning. Four participants mentioned that they prefer to save their work to a new version at the end of the day or a session while two participants said they do it more frequently (e.g., hourly). Many applications provide an automatic save feature that creates a new version of the current work in a custom time frame, and this was also corroborated by one participant who reported that,

"*I have an autosave feature which increments the version number of a save I'm using in Blender.*"

3.3 REASONS TO GO BACK TO THE PREVIOUS VERSION

Participants reported several reasons for going back to previous versions of their files: user accidents and mistakes, broken or corrupt files, testing new changes, exploring alternatives, referencing or extracting previous information, change in requirements and matching needs, and tracking progress/project history.

3.3.1 User accidents and mistakes

Accidents and mistakes happen due to users' negligence that can interfere with their work. The most common mistake that people make is to change file content that should be left alone or accidentally save over the same file. Unintentional content deletions in a revised file or accidental file deletions are some other common mistakes.

Twenty-three participants mentioned that it is their own mistakes that result in them having to go back to previous versions. One of those participants told us that sometimes while merging separate edits/files, the outcome is not the desired result or some merge conflicts happen in their files, therefore going back to the previous version becomes necessary to fix such issues.

3.3.2 Broken or corrupt files

Files or programs can become unusable due to unexpected computer or application crashes. The files can also become corrupted when being written to a disk. This could happen due to several reasons such as bugs and issues present in the host application, glitches in the operating system, bad sectors on hard drive or storage disk, and viruses or other malwares.

The previous versions act as a backup if any kind of corruption happens in the system. Twenty-seven participants mentioned that they would go back to previous versions in case of broken or corrupt files. Although going back to a previous version is like restoring a previous session or state of work but some users think of it as performing undo actions which helps them to save time in "*adjust[ing things] to work in a reasonable time*".

3.3.3 Testing new changes

While testing the newer changes in a single file or a whole project, the previous versions of the current work can act as a backup in case something goes wrong, like the new version completely breaks due to changes or the project stops working. Twenty-four participants said that they can then switch to the previous version to revert their changes and proceed to work from the last file state. Sixteen of them told us that sometimes the newer changes don't work out as planned and having the backed-up versions can quickly revert those changes.

Two participants mentioned that having previous versions help identify the sections of code that could lead to unexpected behavior in the system. One of them reported that "*If something breaks while working on a newer version, I can try on an older version and see if it will still break.*" And the other participant mentioned applying the identified fix of an issue to the original file as,

"[while] testing changes, [if] a single small change was found to fix the issue, [then] the change is made to the original [file]".

3.3.4 Exploring alternatives

Exploring alternatives refers to when designers switch back and forth between different versions of their designs to see which is better, more aesthetically pleasing, and/or more professional; or when developers test their different implementations of code to determine which works with less memory footprint and/or takes less time to execute.

Nine participants stated that having different alternate versions is quite helpful in making comparisons and one of them noted that it is easier to "*return to a variant design version and continue down that branch of a design*".

3.3.5 Referencing or extracting previous information

Referencing or extracting old information or data to understand how something was implemented or done previously is crucial for developers and designers alike. Consider the case of a developer who added a feature to a project that was later removed because it was no longer needed. Now that

the client's demand has changed and that feature must be built anew, the developer can easily go back to a prior version of the project to extract the required data.

Sixteen participants said that they made use of previous versions in order to retrieve solutions to problems that were already solved in previous versions. Referencing previous versions of their work can also help them recall their earlier workflow or process. Two participants mentioned using older versions to "*extract information from a file that is no longer used*" or "*copy and paste segments that were intentionally removed*".

3.3.6 Change in requirements and matching needs

The requirements of any project can change over time due to several factors such as the client altering their initial requirements, the manager shifting their focus on another feature in their product design, and the developers finding a new algorithm to make certain functionality perform more efficiently. One of the participants summarized this as, "*the nature of designing [a] product is creative and never linear*". Therefore, it is useful to have older versions to fall back on. It is also possible that the newer changes do not meet the intended target/milestone, or the new version turns out not to be what was originally envisioned. Twelve participants had similar thoughts and one of those participants said,

"*When I realize that the new feature I'm working on would have been better implemented in a different way, I go back and redo some work rather than deal with my mistake as I progress further.*"

3.3.7 Tracking progress/project history

Version history keeps track of all the changes made throughout the lifecycle of a project and therefore people use previous versions to check their progress over time. Three participants mentioned the importance of project version history for comparison at different stages of their work while one participant mentioned using previous versions to identify the work done by collaborators and stated,

"*Once a project partner didn't contribute anything, so having an earlier draft that contained basically everything in the project I handed in was very useful as it hadn't been opened/edited since long before my partner claimed to have done said work.*"

There were a couple of participants who mentioned not going back to the previous version and one of them mentioned using emails for retrieving previous versions of work as, "*With coding, I've done it [go back to the previous version] because I mess something up. I use email for prior versions of the papers.*"

3.4 NAMING CONVENTIONS

The survey also asked participants about the naming conventions that they use for their files in their personal and/or professional projects. We discussed why naming conventions are important and what, as our survey revealed, are some common conventions people use to name their files in the following subsections.

3.4.1 Why are naming conventions important?

Saving multiple copies of the same files in the same folder or directory forces the users to give unique names to those files because file browsers do not allow multiple files with the same names. But this mandatory step is not as easy as it seems at first. Substantial effort can go into deciding on appropriate names for the files. The default names that the file browsers give to the copies of a file are suitable only for a certain number of copies after which it becomes difficult to keep the association of a file with its content. **Figure 3.4.1** represents the default file names that the Windows File browser assigns to duplicates of a file. It can be noticed that the pattern of "Copy (n)" (where n is a number) is appended to each incremental copy of the file and as expected, the file names start losing context for creating those duplicate versions of the file and it becomes incomprehensible to recognize which version contains what changes.

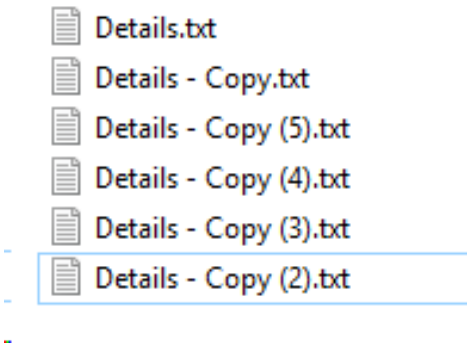


Figure 3.4.1 Default file names of the duplicate files created in Windows Operating System.

Another problem with the default naming scheme is determining which file is the most recent version. At first glance, a file name with the largest number in its name may appear to be the latest version, but that cannot be a certainty. A user could make the latest changes in any of the versions which defeats the purpose of having an incremental number in the filenames. What if the user wants to know what was changed in any of the versions? Just by looking at the names, it is sometimes not easy to answer these questions. There are two ways to make it easier to understand the context of each file: using time references and using proper naming conventions.

Using a time reference for all of the files can be used to sort them according to their date of creation or their date of modification (see **Figure 3.4.2**), making it easier to discover the most recently updated file or the oldest file.

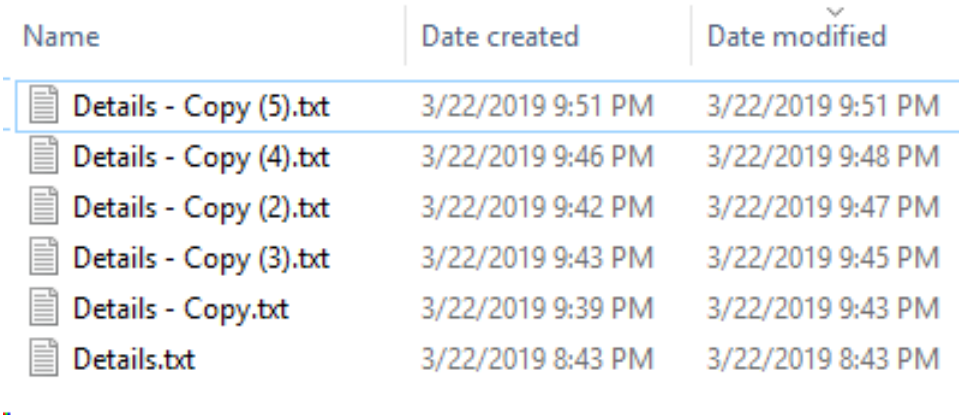


Figure 3.4.2 Files sorted according to Date Modified tag in Windows Operating System.

But using a time reference does not help us figure out what was changed in the files or what was the purpose of creating a new version of that file. We would have to open each file and look at the contents to determine that. This is where using naming conventions helps users easily distinguish between files for the content differences.

There are many keywords that people use with their file names such as ‘draft’, ‘wip (work in progress)’, ‘final’, etc. These keywords help in assigning valuable information to the status of each file. But even then, the naming of different versions of a file may not prove helpful as in the following case depicted in **Figure 3.4.3**.

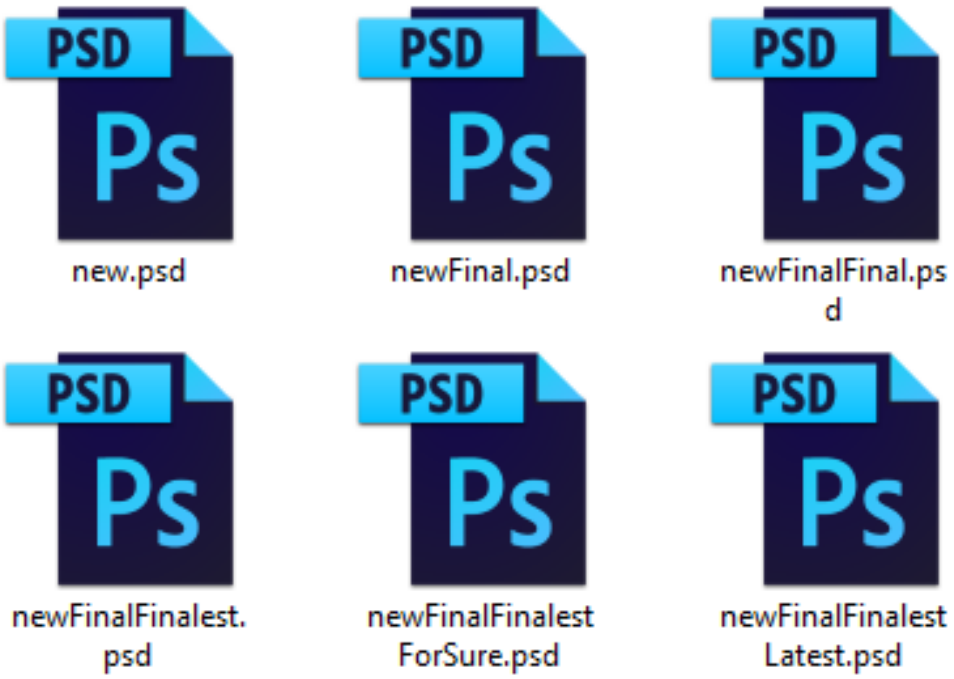


Figure 3.4.3 Confusing file names for different versions.

Here, the user has tried to use the ‘final’ keyword to indicate the last version of their work, but the filenames are still ambiguous. Therefore, it is important to follow a good naming convention to avoid confusion and frustration later, especially when working in a collaborative environment where other users need to understand the state of work at any given point.

3.4.2 Survey result for naming conventions used

There is no standard way to give names to the files, and people use naming conventions that they find easy to understand. We asked people through our online survey about the different conventions that they use to name their files in their personal or professional projects. The survey revealed that people use all kinds of variations for the names to distinguish between different versions of their files. There were many naming conventions that people use, and the most common ones are grouped and listed in **Table 3**.

Convention style	**Examples**
Version number / Sequential	Example.xxx, Example1.xxx Example_1.xxx, Example_2.xxx Example_V1.xxx, Example_V2.xxx
Date and Time / Chronology	Example_YYYYMMDD.xxx, Example_YYYYMMDDhhmmss.xxx
Descriptive	Example_white.xxx, Example_black.xxx
Author name (in collaboration)	Example-Alex.xxx Example-Bob.xxx

Table 3 Naming Conventions

We saw four major convention styles used by people who answered our survey. First is version number or sequential based naming where a number or a letter that represents some sequential

order is appended to the name of the file. For example, a series of 1,2,3, etc., or V1, V2, V3, etc. can be used to convey the order of file created. The second style involves appending date and time to represent a chronology that makes it easy to find files. For example, appending YYYYMMDD or YYYYMMDDhhmmss are some common ways to add a date or chronology information to the file names. The third convention style involves adding a descriptive name to the end of a file that makes it easy to understand what a file may refer to. For example, if there are two logo files named ‘Example_white.jpg’ and ‘Example_black.jpg’, we can easily understand which one refers to a black logo. Lastly, we have a convention style where an author name is added to the end of a file name to easily identify which author has worked on what file when two or more authors are collaborating on a single project. For example, ‘Example_Alex.docx’ and ‘Example_Bob.docx’ filenames can help us identify files associated with their respective authors.

3.5 SUMMARY

In this chapter, we presented a survey we carried out to better understand versioning for users in interactive systems. We have identified several situations when there is a need to create a new version and the reasons to go back to previous versions in a digital software environment. We also saw various naming conventions that are used by people to name different versions of the same files. In the next chapter, we describe the versioning tool we developed to support versioning in interactive systems.

CHAPTER 4

VERSIONING TOOL

In this chapter, we introduce the tool that we built to implement versioning in interactive systems. We discuss what is stored in a version, how versions are stored, the relationship between different versions and how it is represented, and how to interact with the tool. We also introduce the two applications that we used to test the versioning tool.

4.1 VERSIONS AND THEIR RELATIONSHIPS

In the context of an interactive system, a version (the current state of a system) can encapsulate document/model data and parameter space data for that system. A new version is created by making changes to the current state of the system which means that each new version is derived from a previous version, therefore establishing a parent-child relationship. There can be several versions that are derived from a single version, but no single version can have two parent versions, i.e., there is a 1-to-n relationship between a parent version and a set of child versions.

4.2 VISUAL REPRESENTATION FOR VERSIONS

There are two important factors to consider while choosing the graphical representation for versions:

1. Parent-child relationship between each version
2. Bi-directional traversal

Therefore, to represent a parent-child relationship as discussed in **Section 4.1**, we require a structure that has no cycles (a closed path where we start and end at the same vertex), i.e., an acyclic

graph with the ability to traverse in both directions. There are two such structures that can be used to depict versions and their relationships: lists and trees.

4.2.1 Lists

Lists, as shown in **Figure 4.2.1**, are linear structures with just one child at each vertex except the final vertex, which has none. The vertices in this type of structure can be laid out in two different orientations: horizontally and vertically. In this type of structure, there is only one root vertex (colored purple) and one final vertex (colored green).

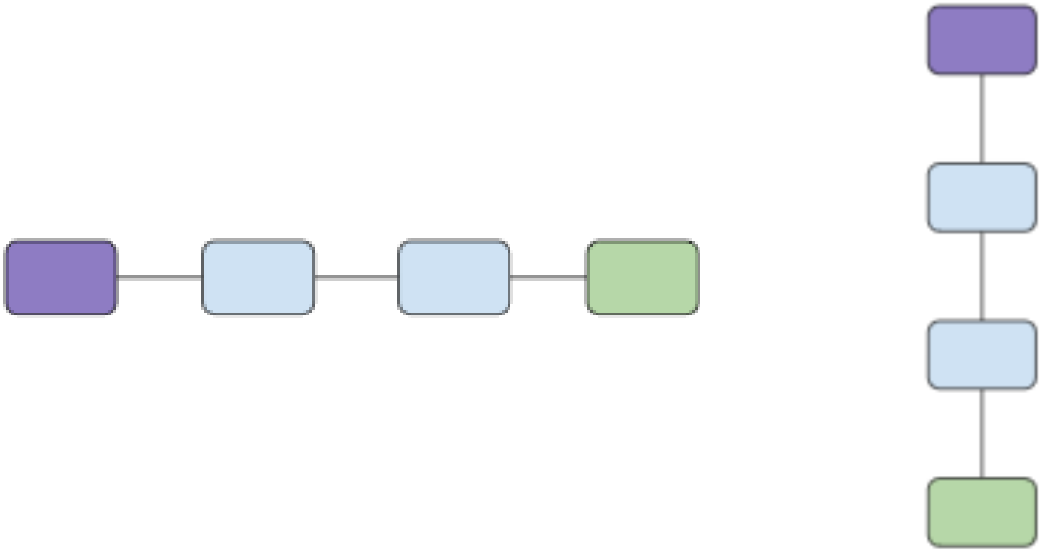


Figure 4.2.1 Lists: horizontal layout (left) and vertical layout (right).

Timelines [110] and storyboards [111] are graphical ways of representing a list of events in chronological order. These types of visualizations are quite effective for storing time-based events where a linear structure is observed.

An undo stack is an example of a linear structure that is implemented in a wide range of applications. Adobe Photoshop's History panel [112] implements the undo stack where each action performed in the document is recorded and displayed in a stack of changes. Users can go to

different states of the document saved in the undo stack by directly selecting an action from the list (see **Figure 4.2.2**).

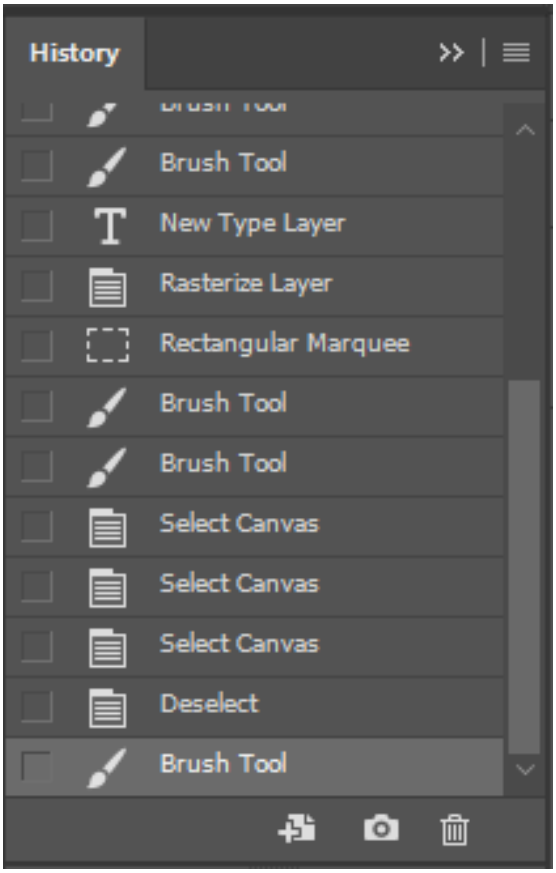


Figure 4.2.2 Adobe Photoshop's History tool representing a linear hierarchy.

4.2.2 Trees

A tree is a non-linear structure that can have vertices with zero or more children, see **Figure 4.2.3**. There are two common ways of organizing a tree structure: (a) each vertex has the same number of children except the leaf vertices (vertices with no children), and (b) each vertex can have any number of children.

A tree is defined as an undirected graph in which any two vertices are connected by a unique edge. The following is some basic terminology [113] that is associated with trees:

- *Root node*: The topmost node of a tree that has no parent node.
- *Leaf node*: A node of a tree that has no child nodes.
- *Branch node*: A node with at least one child node.
- *Ancestor*: A node reached by repeated traversing from child to parent.
- *Descendant*: A node reached by repeated traversing from parent to child.
- *Branch*: The path connecting a root node to a leaf node.
- *Size of a tree*: Total number of nodes in a tree.
- *Degree*: Number of children of a given node.

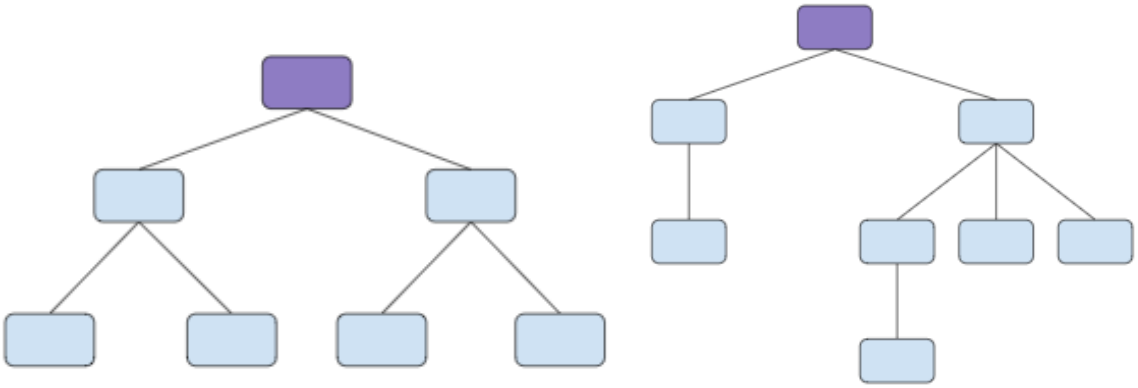


Figure 4.2.3 Tree Structure: number of children are fixed (left) and number of children are not fixed (right).

Tree visualizations can encapsulate information for work diverging in different spatial directions from a common point. A version of work derived from the parent node can be stored in each node of a tree. Unlike lists, where each node can only refer to one neighbor, each node in a tree can refer to multiple neighbors. While both lists and trees allow for many snapshots of the project to exist at the same time, only trees allow users to go to any version and create a new derivative version to save their recent changes without impacting the other versions. This is because trees allow users to add new child nodes to any prior node, whereas lists only allow users to add new nodes to the lists' endpoint. Consider a scenario (see **Figure 4.2.4 (left)**) where a user is working on a project that has three versions. Version 3 is the most recent version of the project, and it is represented by the color green. Now, if a user decides to go back to an earlier version of the project, version 1, and makes some modifications to be saved, a new version will be created in the version history. If the version history is stored as a list (as shown in **Figure 4.2.4** (**middle)**), versions 2 and 3 (colored

red) will be removed from the list, and a new version 2 will be added after version 1 and becomes the current version (colored green). However, if the version history is stored as a tree (as shown in **Figure 4.2.4** (**right)**), a new version (version 4) is added as a child to version 1, creating a new branch in the tree and marking it as the current version. The other versions (version 2 and 3) will not be removed from the history as they will be preserved in a separate branch.

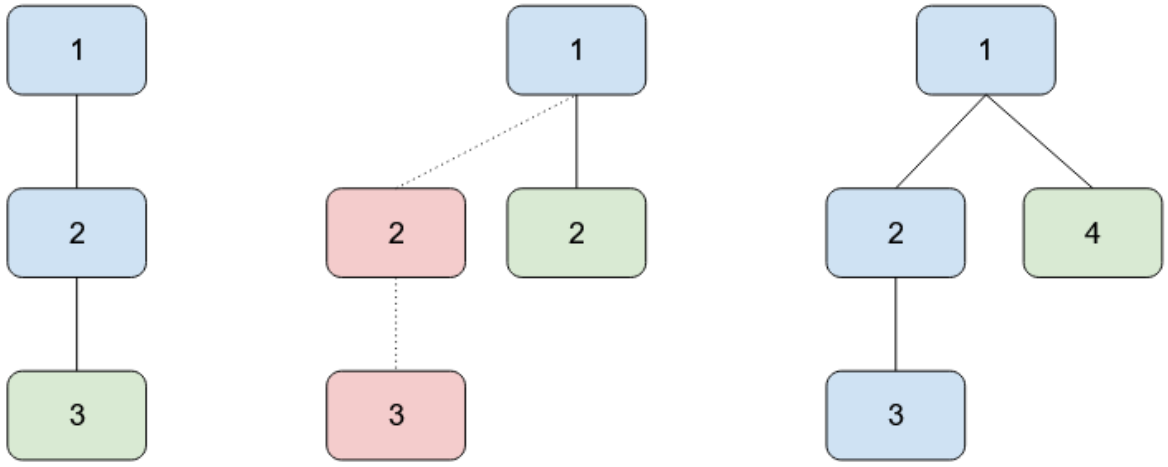


Figure 4.2.4 Set of versions of a project (left), version history stored in a list when a new version is added (middle), and version history stored in a tree when a new version is added (right).

4.3 VERSION TOOL FOR INTERACTIVE SYSTEMS

We developed a custom tool to implement versioning in web-based applications. The tool allowed users to save the document/model data and UI state data of an application as a version, which could subsequently be displayed as an interactive version tree. Users can also access, manipulate and navigate versions stored in the tool. The survey results motivated us to add two features to our version tool – preview feature and auto-versioning. One of the reasons to go back to a previous version was to reference or extract information from it, which informed the design of the preview

feature (discussed in **Section 4.3.3**) that allowed users to see the data saved in past versions without actually switching to them. The respondents of our survey mentioned time-based versioning where they created new versions at different time intervals (e.g., hourly). We leveraged this concept to implement autoversioning feature in our tool, which allowed us to save a new version on various triggers (see **Section 5.1.6.4**).

The version tool was developed in Javascript and Node.js, while the data storage is done using mongoose [114] – an object modeling library for MongoDB. The data for each version, stored in a JSON format, also contained a reference to the parent version to provide a parent-child relationship which helped construct a version tree. The tree visualization was developed using 'vis.js' [115], a JavaScript library for web-based visualization, but it was modified to allow custom node colors and event handling.

4.3.1 What information is stored in a version?

According to the Causality model (as explained in **Section 2.3.6**), artifact and context are two of the five main components that help in modeling the temporal interaction in the system. 'Artifacts' are the main work objects in the application at any given time whereas 'context' is the state of the application at any given time that stores all system and parameters settings as well as interactive elements. Both components comprise our data model which we store in a version.

For an interactive application, we can save data information on every aspect of the application's interactive system. However, we wanted to mainly focus on storing the data of the application in two aspects – document/model state and UI state. Both of these aspects are covered within our two interactive environments (**Sections 4.4 and 4.5**) that we used for testing our versioning tool. In the game level editor, our versioning tool stored the document/model state where data for each version consists of the location of each sprite in the level. On the other hand, our tool in the web analysis tool stored UI state where data for each version consists of the state of the UI that is used to specify parameters for an analysis run.

4.3.2 Visualization of the version tree

The version tree contains different nodes that are connected through curved lines, as seen in **Figure 4.3.1**. Each node in the tree represents a separate state of the system. The version tree always begins with one root node acting as a parent for all the other nodes that are created later on during the process of working with the interactive interface. This root node consists of the initial system state. As new changes are introduced to the system, new nodes are added to the tree beneath the node that is currently active (called the current node). To avoid cluttering up the screen space, by default not all of the nodes are displayed on the screen at once. The scale (zoom level) of the tree view is automatically set to put the current node in focus but a user can change the scale as explained in the next section. Initially, the tree focuses on the latest node in the tree; otherwise, the current node is in focus.

All nodes in the version tree are colored blue except for the current node, which is colored green (see **Figure 4.3.1**). Any other node in the tree that a user hovers over, with their mouse cursor, to preview the version is colored orange.

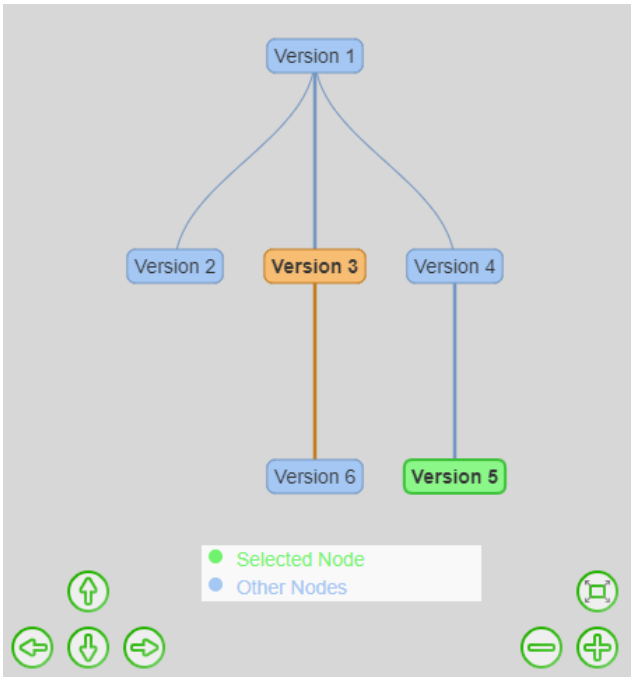
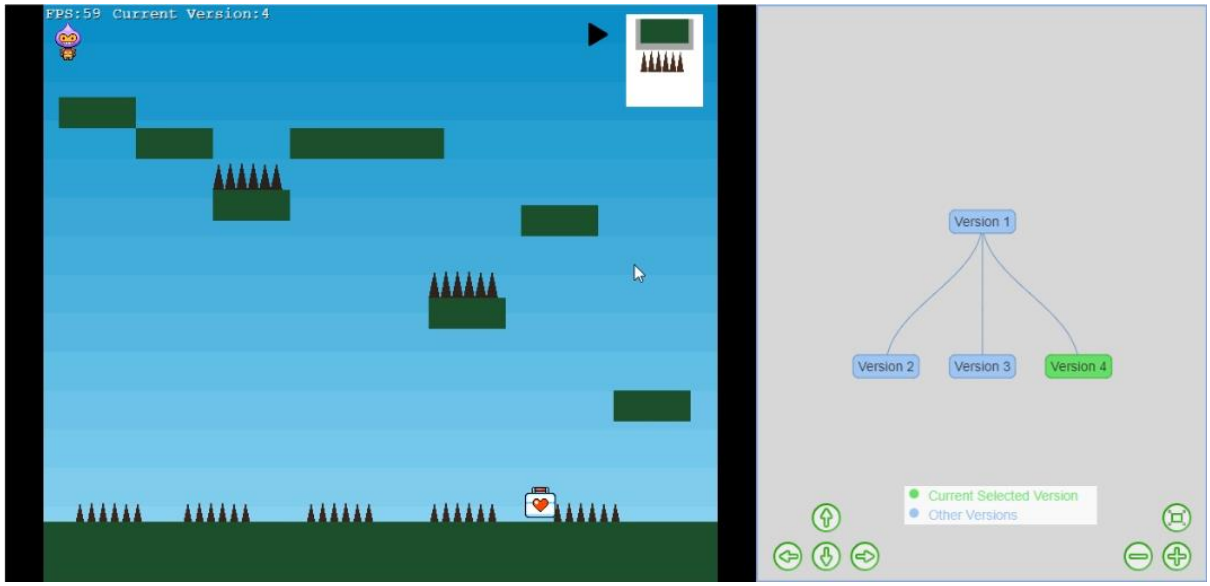


Figure 4.3.1 Version Tree panel with different versions represented as nodes of the tree.

4.3.3 Interaction with the tool

Users can switch between different versions by clicking on the nodes of the tree. This will set the document/model state and/or UI state of the application to the data stored in the currently selected version. For example, switching between different versions in a game level editor (**Section 4.4**) sets all sprites to the positions stored in that selected version whereas in the web analysis tool (**Section 4.5**) all values of the web form elements are set to the saved values. We will discuss saving and retrieving document/model state and UI state in the next sections.

The hover feature gives a visual preview of what is stored in that particular version in the tree. When users hover their cursor on a particular node, the color of that node changes to orange to reflect that the state has been changed to the preview state. Depending on the application in which the tool is used, the preview can be implemented in a variety of ways, such as previewing level changes (**Figure 4.3.2**) in the game scene (**Section 4.4**) or displaying stored commands as a tooltip.

(a) 

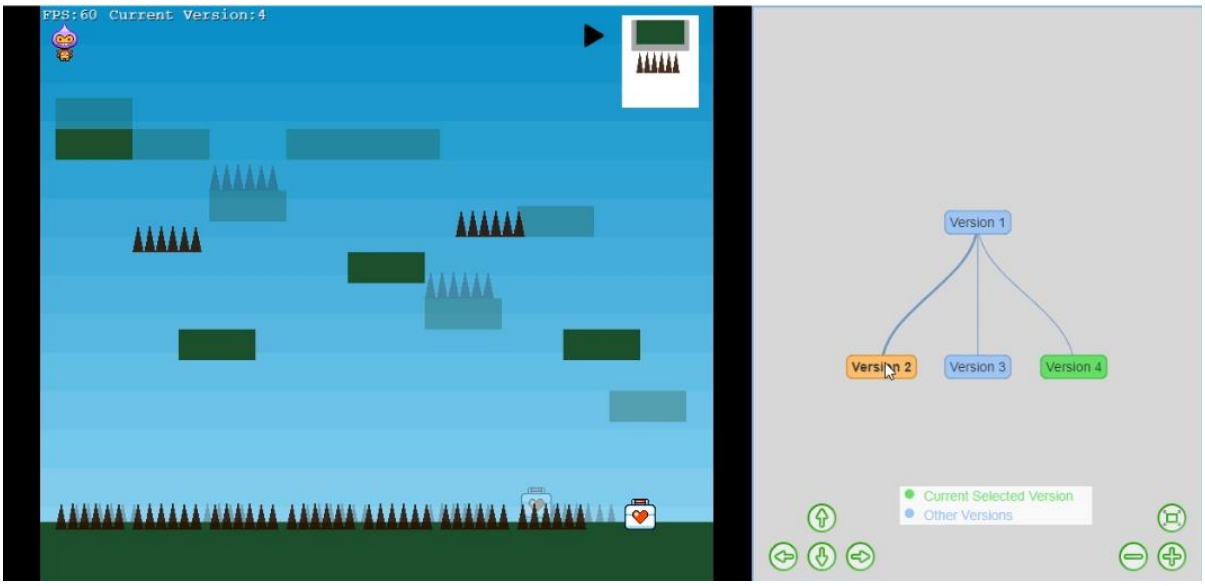
(b) 

Figure 4.3.2 (a) User working on a current version (Version 4) of a level (b) user previewing Version 2 of a level by hovering over a node where the changes from Version 4 are shown with lower opacity.

The tree view also provides zooming and panning using the features of the vis.js library. Users can zoom in and out of the tree view to change the size of all the nodes displayed in the panel. They can use either the mouse scroll wheel or the '+' and '-' buttons to zoom. The fit-to-view button above the zoom buttons fits the whole tree inside the view. Panning allows users to move around the tree view. Users can pan the view left, right, up, and down using the buttons (as seen in **Figure 4.3.3**) or by dragging the view with the cursor on the background.

Figure 4.3.3 Panning buttons (left) and Zoom buttons (right).

4.3.4 How versions are stored

A VCS system like Git can be integrated with our versioning tool in the backend to keep storing the application's data as versions in the git history. Since we were dealing with simple text data (no asset files or large files were included) to represent document/model state or UI state, we opted to store this data in JSON format for each version in our versioning tool (see **Figure 4.3.4**) instead of integrating any VCS with our tool.

(a)

(b)

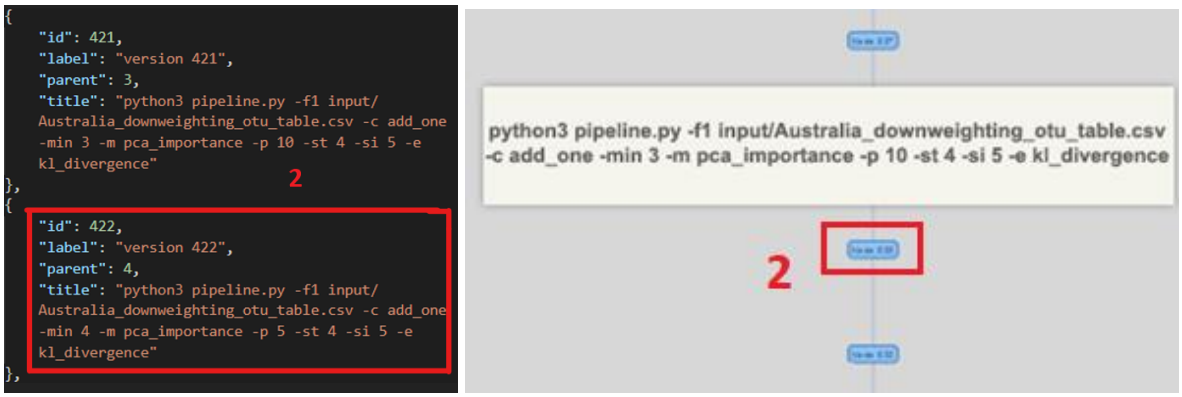


Figure 4.3.4 (a) JSON data of version 2 for the game level editor (left) and its corresponding version in a version tree (right), (b) JSON data of version 422 for the web analysis tool (left) and its corresponding version in a version tree (right).

Our versioning tool's preview feature displays the preview of the currently hovered version in a tree by lowering the opacity of the current version's content and overlaying the content of the hovered version, only if there are differences between the two versions. We compare the two versions to see if they store different data. We stored the data for each version in a JSON format which is then 'stringified' (converted into a JavaScript string). This resultant string is then compared with another version's saved data string using a strict equality operator (===) that uses the 'Strict Equality Comparison Algorithm' [116] to compare two operands. Since we made sure the order of data in the JSON data structure always remained the same, we decided to use a simple equality comparison instead of opting for a more advanced JavaScript comparison algorithm such as "fast-equals" [117]. It works well for our versioning tool prototype; however, a robust diffing algorithm [118] could be used for a production-ready versioning tool where only the result of diffing two versions can be previewed instead of displaying everything in the previewed version.

4.3.5 Integration within a web-based application

We integrated our versioning tool in the interactive systems interface (game level editor and web analysis tool, discussed in **Sections 4.4 and 4.5**) to store versions and display them in a version tree in the form of a panel that was attached to the right-hand side of the main interfaces. Our tool provided useful functions such as saving data to a version and displaying the version tree inside the panel. The application can call a function to save the document/model state from their model code, as well as the UI state information from its UI handlers, in JSON format for each version. All versions are stored in a JSON array which is used to display the version tree in a side panel. The tool also provided other functions that allow an application to access individual versions or implement their custom preview functionality.

In the game level editor, a version can be saved by pressing the keyboard shortcut 'Ctrl+S' (for saving changes in the current version) or 'Ctrl+Shift+S' (for saving changes to new version). In the web analysis tool, a new version is saved when a user presses the 'Run' button to run the analysis with currently selected parameters. Both systems allow users to explicitly create new versions, where the control is in the hand of the user. However, auto versioning is a very useful feature for backup and error recovery as we discussed in **Section 2.2.2**. So, we integrated an auto-

versioning feature inside our tool. It allows us to configure parameters for when to trigger a new version save. We currently support four trigger events to save a new version – on every play (saves a new version every time a user playtests the level), 3-changes (saves a new version on every 3 changes in the level), every-change (saves a new version on every change made to the level), and time-based (saves a new version at a set interval in seconds). We wanted to compare a version tree generated when a user explicitly saves a version and the trees generated during each of these trigger events (discussed in **Section 5.1.6.4)**.

4.4 GAME LEVEL EDITOR

We developed a 2D game level editor using Phaser [119] which is a Javascript game framework for Canvas and WebGL. It runs in the browser using the Express.js [120] web application framework running in a Node.js [121] environment.

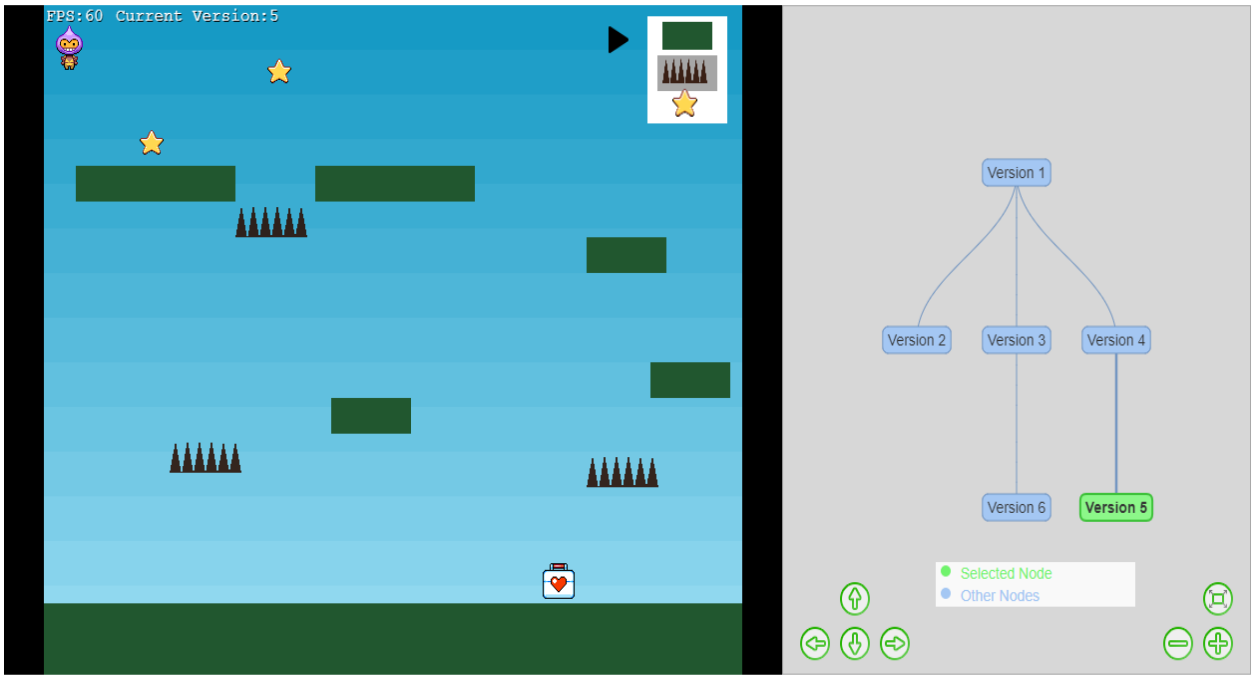


Figure 4.4.1 Game Level Editor (left) and Versioning Tool (right).

The game level editor allows a user to design game levels using three sprite types: ledges (green tiles), spikes (brown triangular tiles), and stars as shown in **Figure 4.4.2**. This application allows the users to create, delete and move the 2D sprites in a game view. Clicking the right mouse button allows them to add a new sprite of a selected type which can be changed from the sprite selector box located in the top right-hand corner of the editor as shown in **Figure 4.4.1**. The middle mouse button click deletes the current sprite under the cursor and the left mouse button click allows them to drag and move the sprite across the screen to change its position.

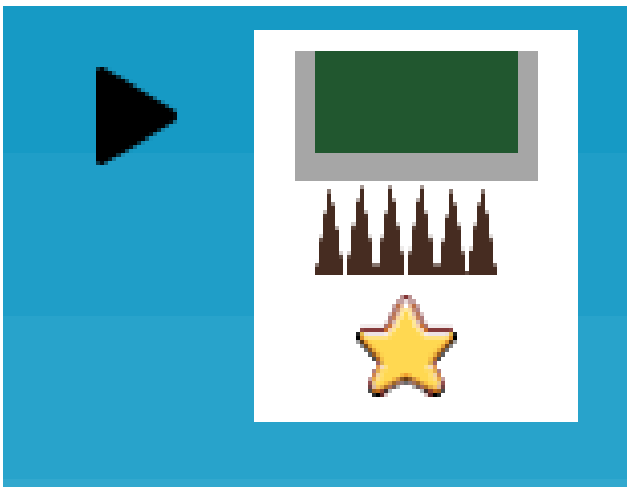


Figure 4.4.2 Sprite box selector and playtest button.

Users can playtest the level by entering into the Playtest mode, which can be enabled by clicking the play button shown to the left of the sprite selector box (**Figure 4.4.2**). Once the user is in playtest mode, they can control the character with the arrow keys to move left, right, or jump. To exit the playtest mode, the user can click the stop button that replaces the play button in the play mode. The objective of the game is for the character to reach the health icon represented as a white box with a heart shape on top. Users can step on the ledges to make their way toward the health box avoiding contact with the spikes that can kill the character, and can earn score points by collecting as many stars as possible before finishing the level. The user can restart the same level from a Restart button that pops up following the end of that level (**Figure 4.4.3**). The level also gets reset if the player collides with the spikes. The score increases by one with every star sprite collection and the time taken to complete the level are displayed when the level is finished.

A general undo-redo functionality exists in the tool which allows going back and forth between the previous actions taken in designing the level. This functionality can be utilized using the standard shortcut keys: ‘Ctrl+Z’ to undo and ‘Ctrl+Shift+Z’ or ‘Ctrl+Y’ to redo. The undo-redo actions were stored in a global stack (a data structure that follows the ‘last in, first out’ principle – where the first element is processed last and the last element is processed first) at an application level. Any action performed in the game level editor such as adding, deleting, and modifying the sprite’s position will add an undo-redo command in the undo-redo stack. The undo-redo functionality works at an application level and not at a version level, i.e., the undo-redo stack is cleared when a user switches to a different version or a new version is created.

We allowed the users to save their changes in the levels to the current version using explicit save triggers by using keyboard shortcuts: ‘Ctrl+S’ or to a new version with ‘Ctrl+Shift+S’. There are some caveats for saving the versions: a new version can only be created if there are changes in the level since the last save; the versions that already have one or more children cannot be overwritten – trying to save changes to such versions will create a new version that is a descendant of that version.

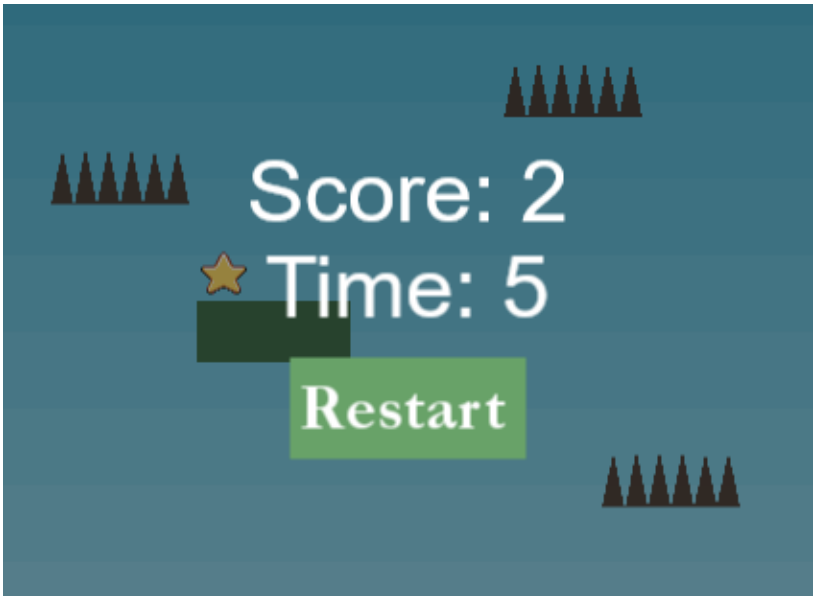


Figure 4.4.3 Game End screen with the restart button.

In **Figure 4.4.4 (left)**, the current version highlighted in green (Version 2) has a descendant version 'Version 4'. Making any changes in the current version will not overwrite the contents of this version but will create a new child 'Version 5' with the changes (**Figure 4.4.4 right**).

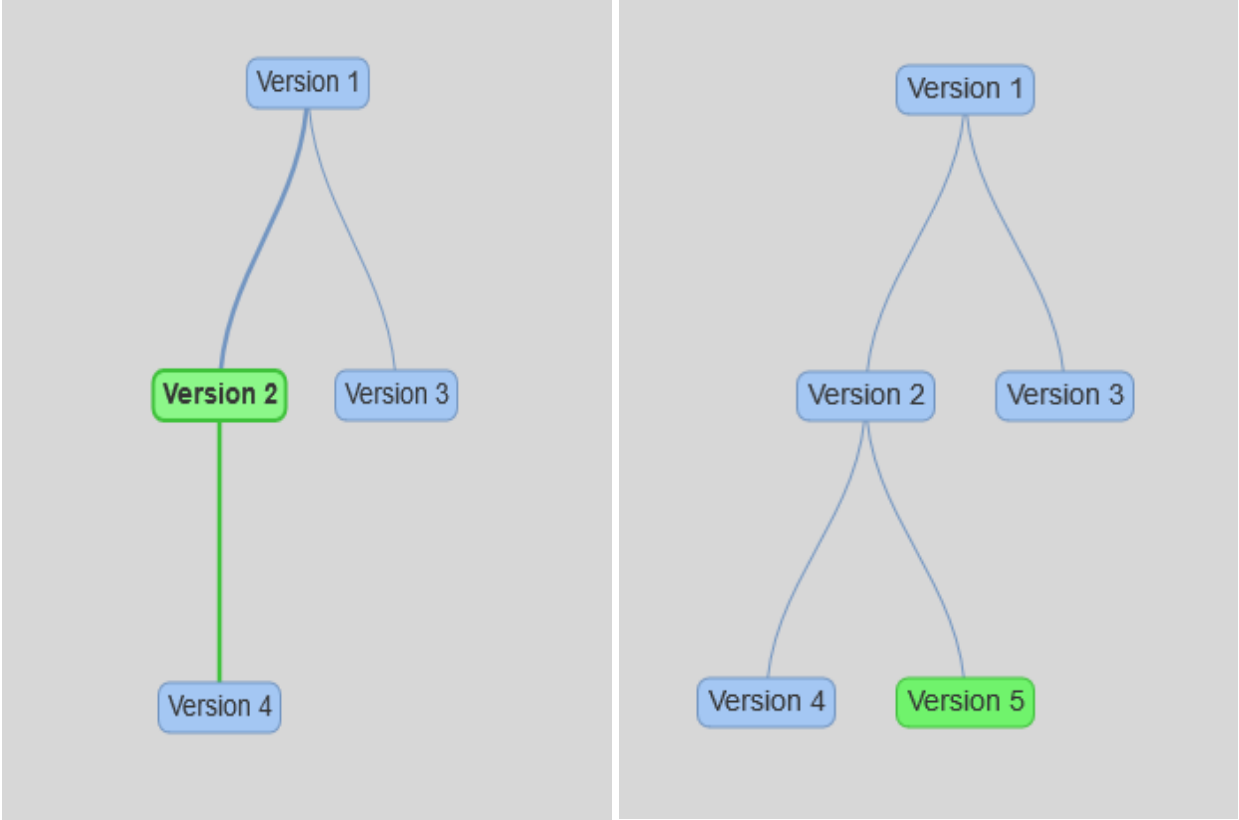


Figure 4.4.4 (Left) Current version is 'Version 2' highlighted in green. (Right) When the user saves changes while in 'Version 2', it does not save those changes to 'Version 2' but creates a new 'Version 5'.

4.4.1 What is stored in each version?

Each version in the game level editor stores data for positions of all the sprites (spikes, ledges, and stars). The undo-redo actions stored in the undo-redo stack of the application were not included in the data stored for each version. Our main goal was to store the application's document/model data that represents game levels to quickly iterate over their designs without worrying about creating

backups and inadvertently destroying their previous work/designs and even making use of templates.

4.4.2 Pilot Studies

We ran some initial pilot studies (with 5 participants recruited from the University of Saskatchewan) during the development of our versioning tool and integrated the tool inside a custom game level editor for evaluation purposes. It helped us finalize the design and interactions of our versioning tool that we tested in our main study (**Section 5.1**). The feedback from these pilot studies helped us identify and fix some key issues with our versioning tool as mentioned below:

1. The nodes should be different colors for different types of nodes.
2. The nodes should be easy to rename for better structuring of the tree.
3. The preview feature should change the opacity of all objects in the current and previous versions instead of changing the colors of sprites for better visualization.

We also found that some participants did not use the nodes in the tree when asked to make changes in the level. Explicitly telling them to change the version helped them get familiar with the concept of a different version of the same level. Once they understood how to interact with the nodes in the version tree, they were able to access the different versions of their game levels represented by those nodes.

4.5 WEB ANALYSIS TOOL

The Winnowing Pipeline is a web analysis system that runs a python script with various arguments to analyze a soil-microbiome dataset in a variety of ways [39]. We created a web interface for this analysis system (see **Figure 4.5.1**). Users can select a dataset or upload new dataset files and change parameters from the web form that consists of sliders and drop-down options. The changes in the various parameters alter the python command that is to be executed on a selected dataset file. The

user can then press the ‘Run’ button to execute the command that will start the analysis on the selected input file. The users can do multiple analyses by running the commands in a batch by listing the desired commands in the field titled ‘List’ which in turn runs all the commands in sequential order.

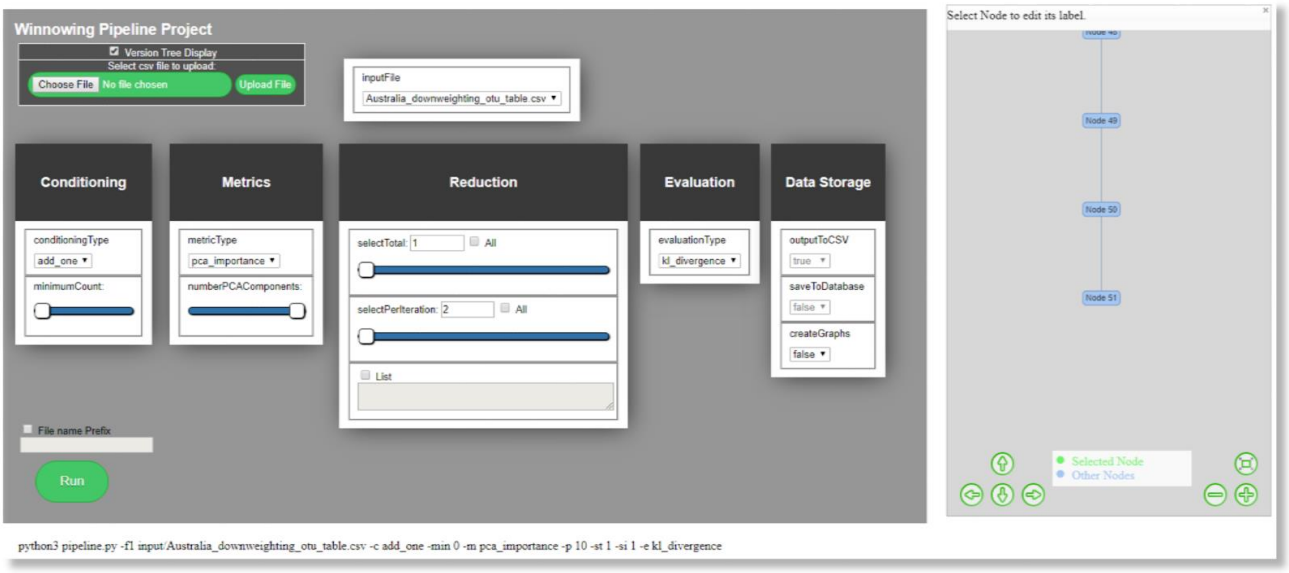


Figure 4.5.1 Web analysis tool’s interface (left) with versioning tool (right).

4.5.1 Augmenting the Winnowing System with Our Versioning Tool

We added our versioning tool to the web analysis tool. In this system, there was no document/model state, and so the versions stored only the UI state; and because the UI specified a parameter space for the analysis command, each version was a point in this parameter state. On each run, a new node was created in the visual tree on the right-hand side of the interface (except when there was no change in the command). If the user double-clicked on any previous node, the respective parameters in the form were changed, i.e., the form effectively went back to the previous command settings. The preview was enabled by hovering the mouse over the node, which showed the state values of the UI elements as parameters to the actual python command that was used to run the analysis (see **Figure 4.5.2**); this format for presenting the version was already familiar to the users.

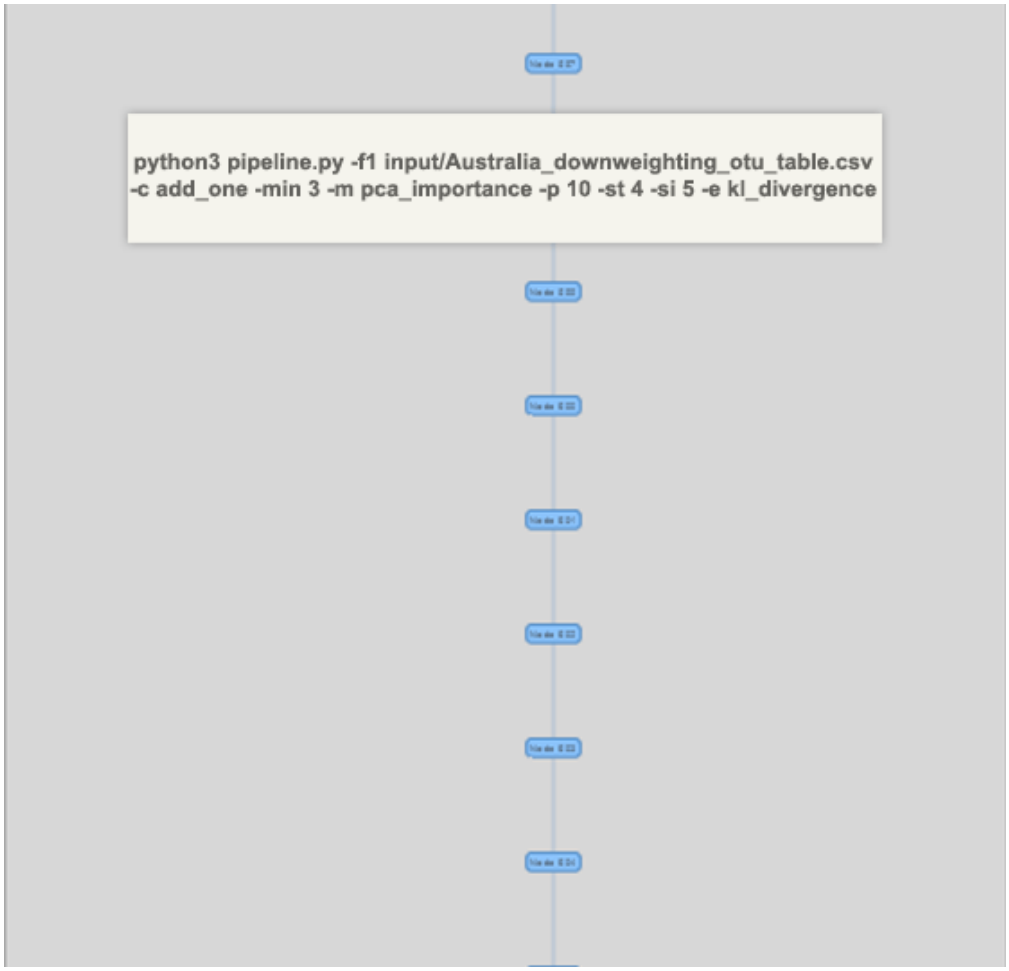


Figure 4.5.2 Tooltip displaying stored command in a version (node).

4.5.2 What is stored in each version?

Each version in this web analysis tool stored information for each field of the web form and the final python command that was generated after pressing the Run button. No resulting files and data were stored in any version. Our main goal was to store the UI state of the application, i.e., all settings associated with running an analysis, in each version. The idea was to understand how versions can help the users remember or save their history of analysis without having to explicitly save the parameters for the analysis in some external program or service.

4.6 SUMMARY

In this chapter, we introduced the tool that we built to implement versioning in interactive systems. We discussed the document/model state and UI state information saved in a version, how to interact with the tool, and how different versions were visualized within our tool. We also discussed the different representations that can be used for depicting version history and the most appropriate structure to represent versions and their relationships, i.e., tree – a non-linear hierarchical graph. We also introduced two different interactive applications (a game level editor and a web analysis tool) that were used to test the versioning tool.

CHAPTER 5

USER STUDIES

In this chapter, we discuss two user studies we performed to test our versioning tool in interactive environments. The first user study was designed with three goals: to evaluate our versioning tool in a game level editor, confirm if participants used our versioning tool for the different circumstances and scenarios identified in the survey (**Sections 3.2** and **3.3**), and identify key elements that can be used to improve our interactive versioning tool in future. We report how the participants performed study tasks of designing game levels and how they interacted with our versioning tool. We also discuss study design tasks along with the results and observations of the study in the following subsections. The second user study was designed with a goal to test our system in another interactive environment, i.e., a web analysis tool – a real-world application for carrying out analyses in microbiology. We discuss user feedback and what we observed when hundreds of analyses were run in the pipeline.

5.1 STUDY: GAME LEVEL EDITOR

5.1.1 Goals

We developed a custom web-based 2D game level editor to test our versioning tool in a working interactive environment (see **Section 4.4**). The game level editor provides a good mix of versioning and interactivity criteria. The process of level designing is an immersive experience that allows users to have rich interaction with the editor and provides them with ample opportunities to version their levels.

The game level editor includes two main components: a level editor and a versioning tool (described in Chapter Four). We conducted a study in which participants played the role of a game

level designer and designed levels for a 2D platformer game by adding, deleting, or modifying the positions of game sprites using the level editor. The goals of this study were:

1. To evaluate our versioning tool in an interactive system where versions involve document content and see if participants were able to understand and utilize all its features such as saving new versions, previewing other versions, switching to other versions, and creating multiple branches.

2. To confirm if participants used our versioning tool for the different capabilities identified in the survey – circumstances to create new versions and reasons to go back to previous versions (**Sections 3.2** and **3.3**).

3. To identify key elements that can be used to improve the later designs of our interactive versioning tool.

5.1.2 Apparatus

The study used a custom-built game level editor (explained in **Section 4.4**) that supported features such as general undo-redo, sprite manipulation (addition, deletion, and translation), and playtesting. The editor was augmented with our versioning tool that supports saving new versions, renaming versions, previewing saved versions with changes, zooming and panning tree, and switching to other versions. The web application was run on a Chrome Web Browser. We used a 24-inch display monitor with 1920 x 1080 resolution at a 60Hz refresh rate. A wired mouse with a movement resolution of 1000dpi was used by the participants to design game levels for a platformer game.

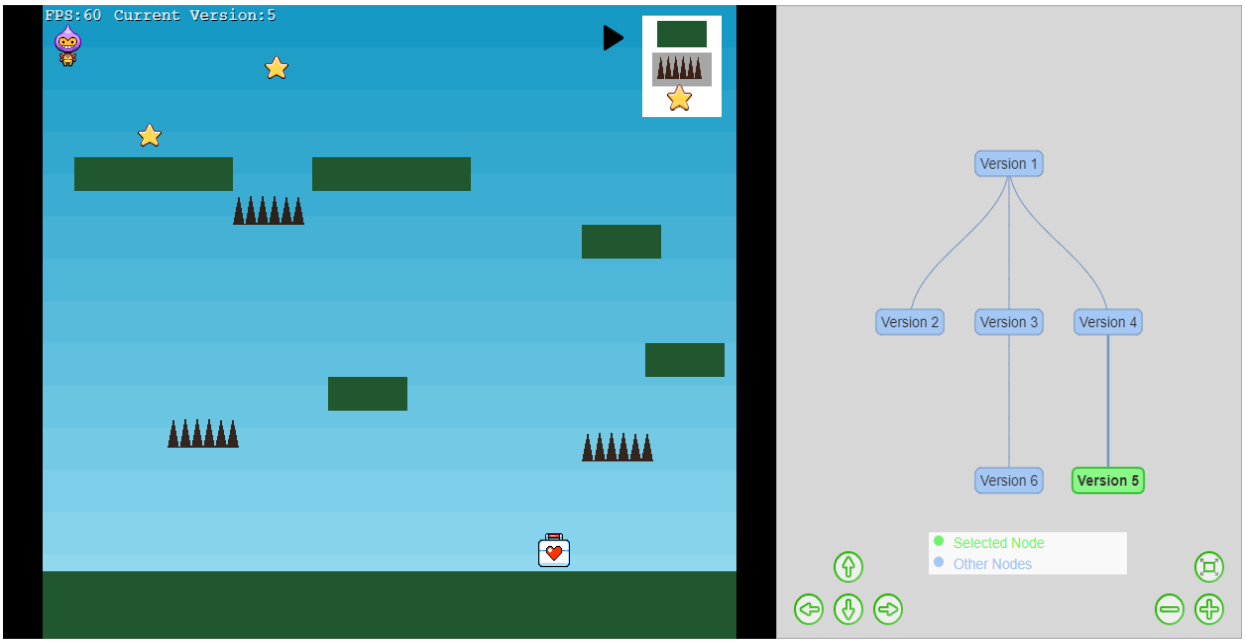


Figure 5.1.1 Game Level Editor (left) with Versioning Tree (right).

5.1.3 Participants

Ten participants (7 males and 3 females) were recruited from the Human-Computer Interaction Lab at the University of Saskatchewan to participate in this study and were given an honorarium of $10 for their valuable time and participation. We wanted to recruit participants with experience in game engine or level design to reduce the training required to work with our custom game level editor. All participants were students and ranged in age from 22 to 30 years. All the participants filled out demographic data as well as our web-based survey (see **Table 1** and **Appendix A3-A5**). We found that all of them were regular users of computers, averaging more than 10 hours per day of usage, and were also familiar with the concepts of versioning and file backup (see **Figure 5.1.2**). Six of the participants said that they played video games on an average of 2.5 hours daily. Nine of the participants had previously designed board games or video games such as first-person shooter, platformer, and puzzle. Seven of the nine participants who designed video games stated that they have used game engines such as Unity and Scratch to create their games.

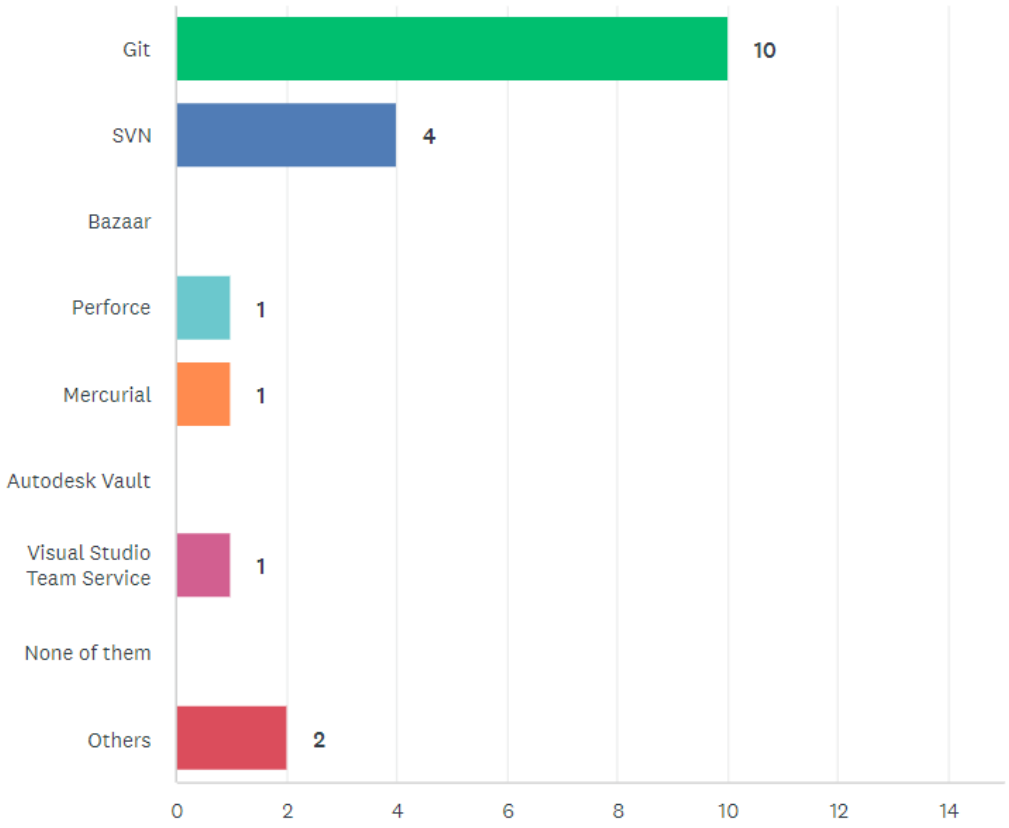


Figure 5.1.2 Participants' responses for versioning tools.

5.1.4 Procedure

The study took approximately 50 minutes to complete. After completing a consent form and a demographic questionnaire, participants went through a short briefing session where they were informed about the tasks (explained in **Section 5.1.5**). The participants were given instructions for each task prior to the start of the task and the same instructions were accessible inside the task itself using the instructions button available above the level editor window (see **Figure 5.1.3**). They were also encouraged to think aloud about their thought process for designing levels and how and why they were creating different versions of their game levels while doing the tasks.

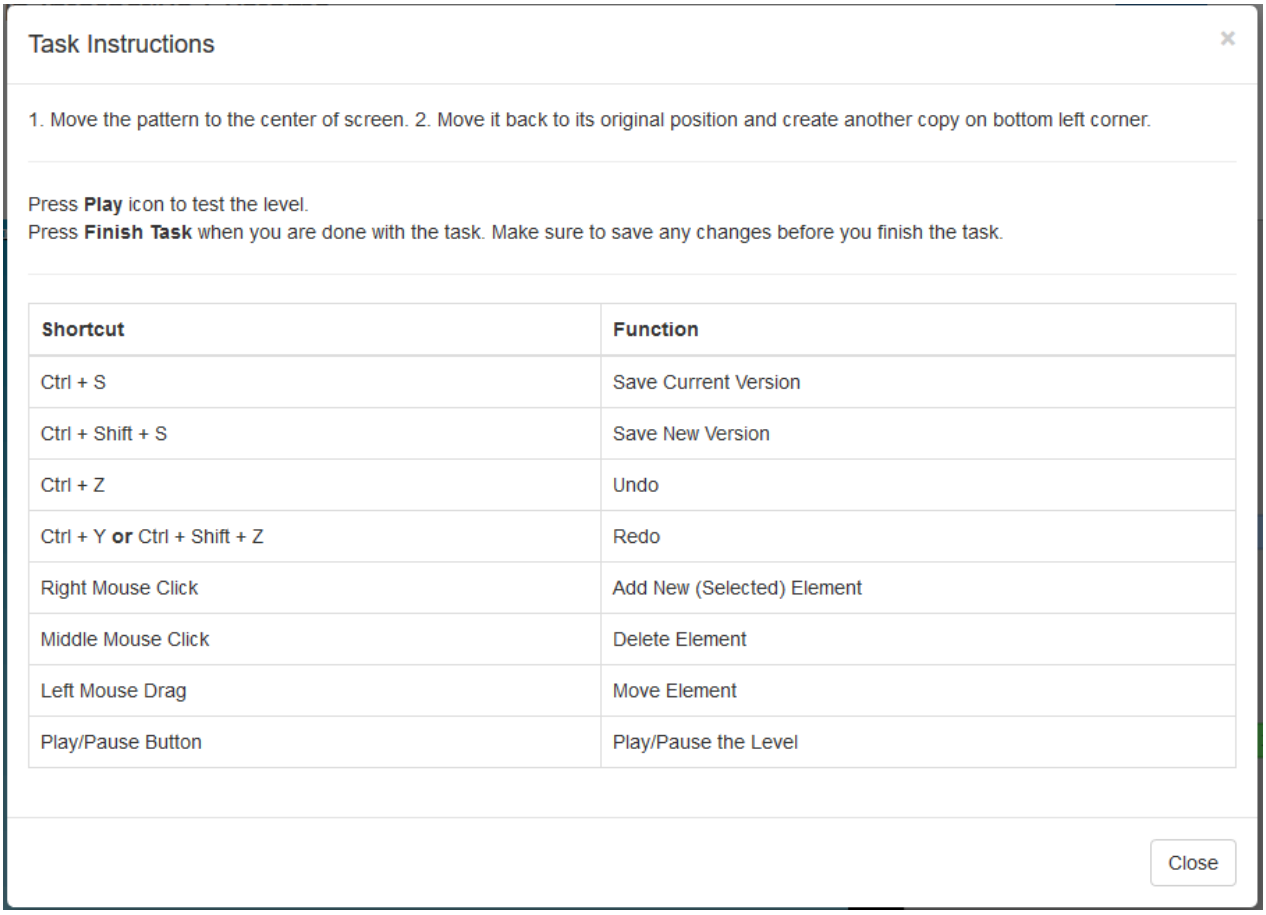


Figure 5.1.3 Pop-up window with task instructions.

We observed how our participants interacted with the system for each task. At the end of each task, we asked questions to our participants based on our observation of their performances. We asked them about their thought process behind creating a new version and why they switched to a previous version for every task. Participants were also encouraged to report any problem they had with the tool so that it could help us identify existing problems with our tool and what useful feature could help us improve it.

At the end of the study, the participants were asked a few open-ended questions about their decision-making for the tasks and their experience with the versioning tool (see **Table 4**).

S. No.	Question
1.	Did you have any issues understanding or working with the versioning tool?
2.	Did prior familiarity with versioning systems help you?
3.	Do you think the versioning tree was helpful? If yes, why and how did it help?
4.	Do you think users not familiar with any versioning tools will find working with this versioning tool easy?
5.	Any other feedback?

Table 4 Post Study Questions

5.1.5 Design Tasks

Participants were asked to perform a series of tasks to design game levels using the level editor. There was a total of seven design tasks (**Table 5**) in the study to see how the participants would make use of the versioning tool available in the level editor and to see when and how often they would create versions.

To familiarize participants with the interface, we asked them to perform easy tasks at the beginning (Tasks 1 and 2), such as adding a new sprite to the screen, deleting the sprite, and moving the sprite around the screen. Once they became familiar with how to work with the interface, we asked them to perform complex tasks that involved moving a particular pattern of sprites to a different location on the screen (Task 3) and altering the levels to add additional complexity for the players to play the levels (Task 4). We created these complex tasks (Tasks 5 and 6) that served as constrained design situations (with pre-existing levels needed to be modified or levels be designed with certain constraints) for a game level designer.

Tasks	Description and Instructions	Annotation
1 and 2	Practice tasks to familiarize users with the game level editor and version tree. *Task 1: Create or delete a few ledges in the level. Move them around and play the level.* *Task2: Hover on 'Version 1' and see the preview of that version. Click on 'Version 1' to change to this version. Now make changes to the level so that it is playable. Save a new version after making those changes and change its label to any name of your choice.*	Familiarization with basic features
3	Move pattern of sprites to different locations. *Task 3: Two steps – 1. Move the pattern to the center of the screen. 2. Move it back to its original position and create another copy in the bottom left corner.*	Previewing and referencing other versions
4	Multiple versions to adjust based on difficulty level. *Task 4: Adjust both of the versions to finish levels within 7 seconds. Then select the node and edit the label to 'easy' for the version that you find is easier to play.*	Templating test and branching
5 and 6	Create levels with certain constraints for a level. *Task 5: Suppose you are designing a level for a beginner player. Create new versions from all of the given versions without changing the spikes, so that players in this game can finish the levels within 8*	Constrained game level design challenges

	seconds. Rename the version that you find easiest to 'Easy' and the hardest to 'Hard'. *Task 6: Suppose you are designing a level for an intermediate player. Create as many levels(versions) as you want. Label them as necessary. Label an easy level that has exactly 5 ledges and 5 spikes to finish in exactly 10 seconds and a player is able to collect 5 stars. Create a hard level from the same template that has 5 ledges and 7 spikes to finish the level in no less than 12 seconds and be able to collect at least 5 stars.*	
7	Design levels without any restrictions. *Task 7: Design a few good levels that will be played by some users later on. No restrictions on any number of items.*	Open-ended design task

Table 5 All tasks in the game level editor study.

The first two tasks were practice tasks that helped participants understand how the basic interactions work in the system. This allowed them to get familiar with the editor to create, move, and delete sprites on the screen. They also learned how to save the current version, save their changes to a new version, preview the changes in other versions by hovering over the nodes, and finally switch between different versions. They also learned how to undo and redo their changes using shortcut keys ('Ctrl+Z' to undo and 'Ctrl+Shift+Z' or 'Ctrl+Y' to redo) to fix their mistakes in the editor along with being able to rename the versions to give more appropriate names to the new versions that they were generating.

In the third task, the participants were asked to move a pattern of sprites (see **Figure 5.1.4**) located in the top left corner of the screen to the middle of the screen, and then move it back to the original

position and create a copy of the same pattern to the bottom left of the screen. This task taught them how to use a version as a reference by utilizing the preview feature of the versioning tool to see the difference between their current version and a previous version. In **Figure 5.1.5**, the version tree displays the currently selected node (current version) “Version 3” highlighted in light pink whereas node “Version 1” is currently being previewed and highlighted with orange color. The changes specific to the current version are visible with lowered opacity while the changes for the preview version are displayed with full opacity.

(a)

(b)

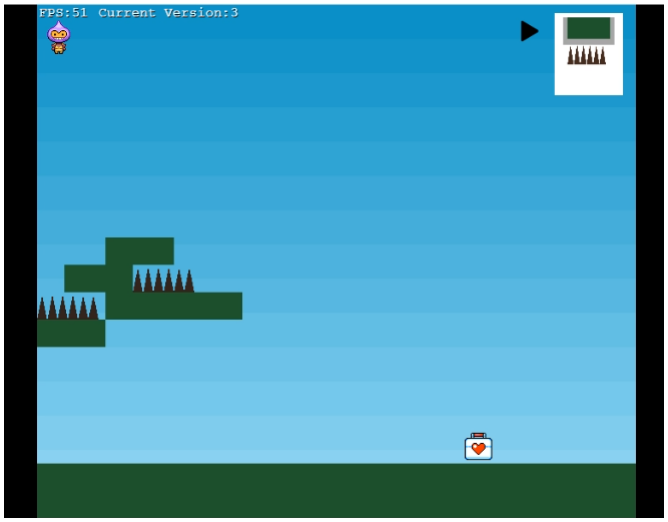


Figure 5.1.4 Task 3 with a pattern of sprites to move: (a) before moving the pattern (b) after moving the pattern.

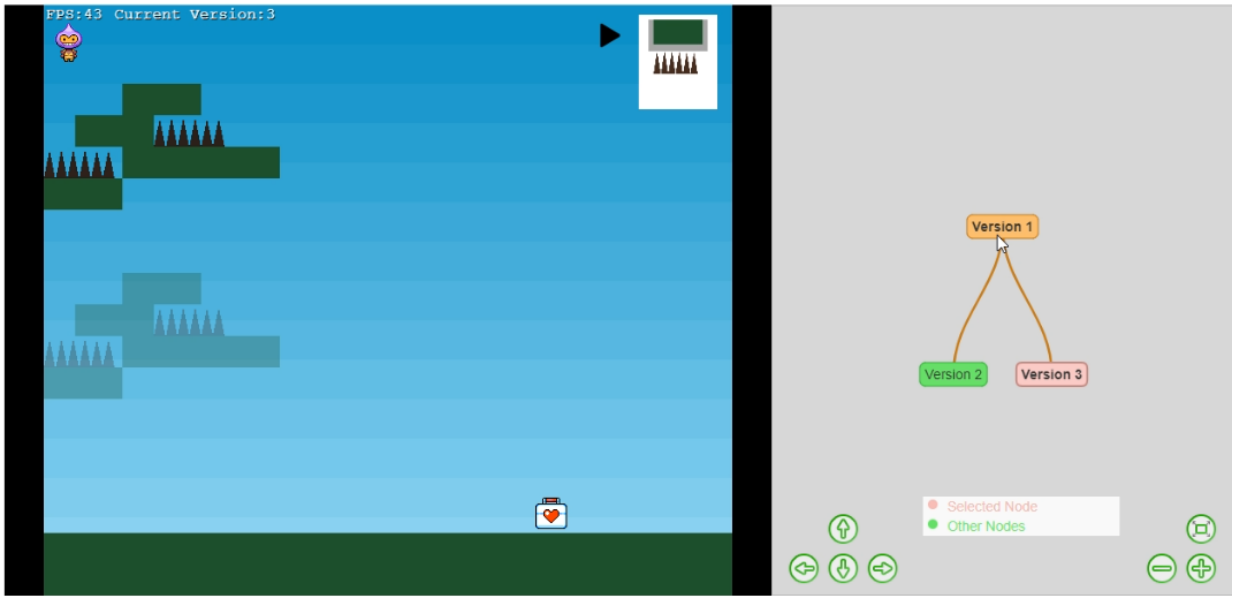


Figure 5.1.5 Task 3 when the previous version is being previewed to see the differences.

The fourth task asked the participants to adjust the two given versions of a level (see **Figure 5.1.6**) and label the easiest (for the players to play) of them as 'Easy', by renaming the node in the tree. This task was designed to make the participants use a version as a template and understand how a branch is created when they make any changes to the root version (Version 1) since any changes made to this version will result in creating a version node in a separate branch. Version 1 served as a template for the participants to create more levels.

In the fifth task, participants were asked to create new versions of a given level without changing the positions of the spikes already presented (see **Figure 5.1.7**), so that the levels can be played within 8 seconds. They were again asked to rename the levels: they were asked to name the easier of the two levels 'Easy' and harder of the two 'Hard'. This task was a design challenge for the participants to work with given game constraints while designing a level. We wanted to observe how participants will utilize our versioning tool's features such as templating and branching while doing a level design challenge.

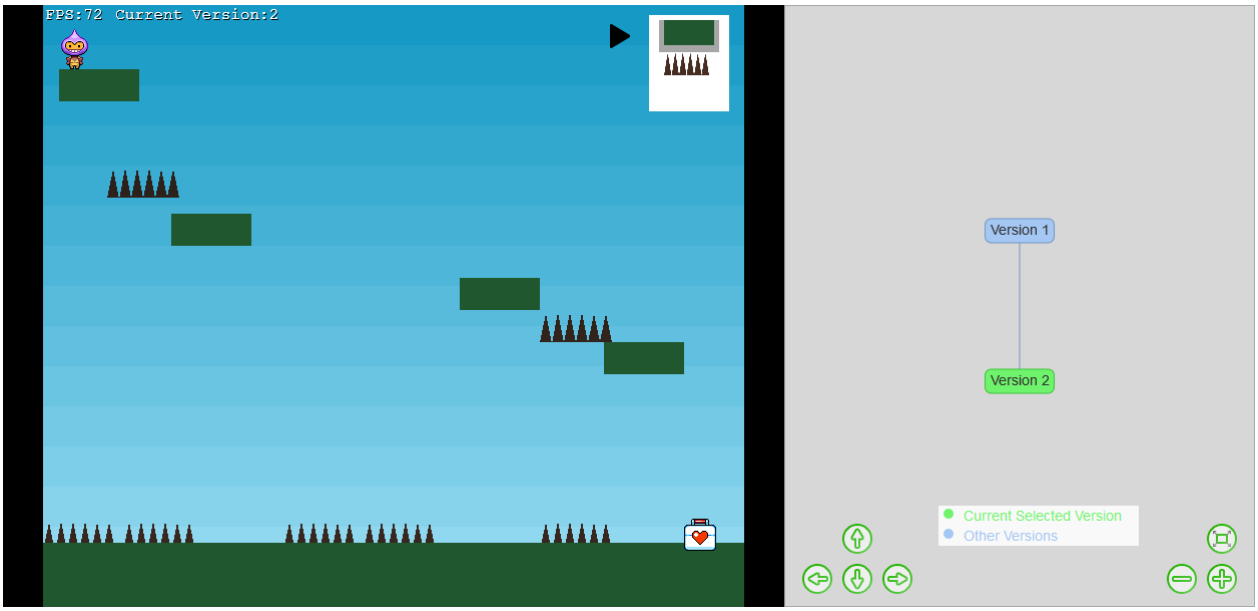


Figure 5.1.6 Task 4 with two given levels to modify.

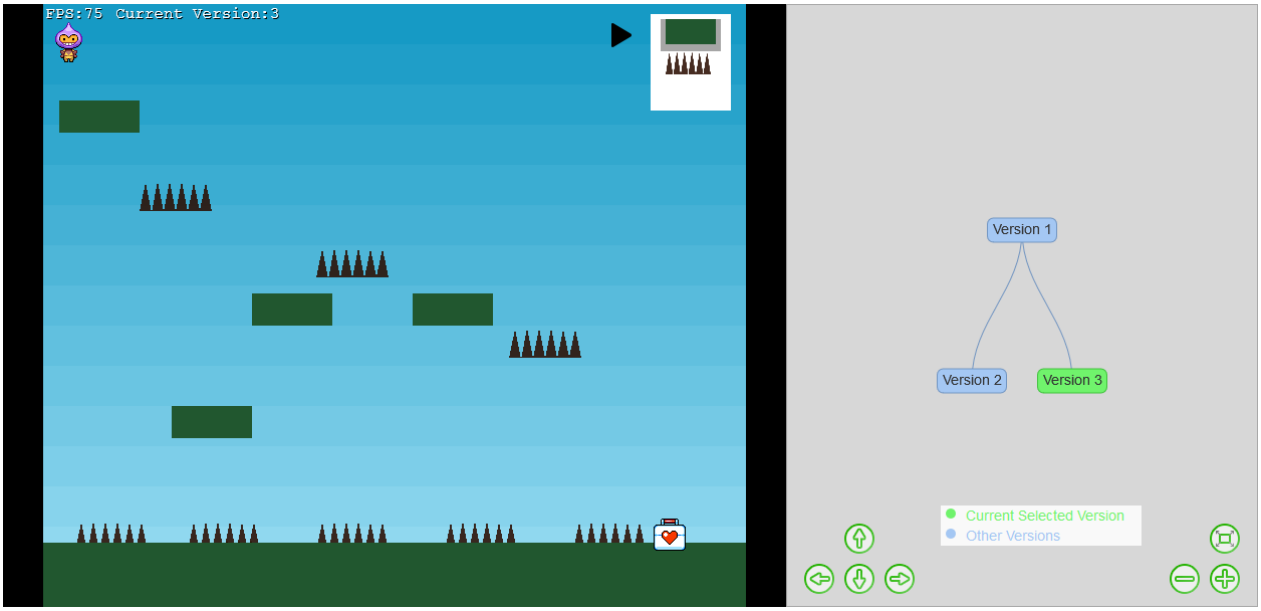


Figure 5.1.7 Task 5 to create separate difficulty levels without changing positions of the spikes.

The sixth task involved making use of the existing level (see **Figure 5.1.8**) to create two levels, an easy level that includes exactly 8 ledges and 8 spikes to make the level playable within 12 seconds, a hard level with exactly 8 ledges and 10 spikes, so that a player with an intermediate skill level may complete the level under 15 seconds. In addition to adding ledges and spikes, the participants were asked to place stars that a player must collect while playing these levels to give an experience of an actual platformer game. During playtesting, each star collected increased the player's score which was displayed at the end screen (see **Figure 4.4.3**). Similar to the previous task, this task was another constrained design challenge for the participants to design game levels with real constraints and for us to observe how our interactive versioning tool can help them when they are designing multiple levels where they could utilize templating and branching.

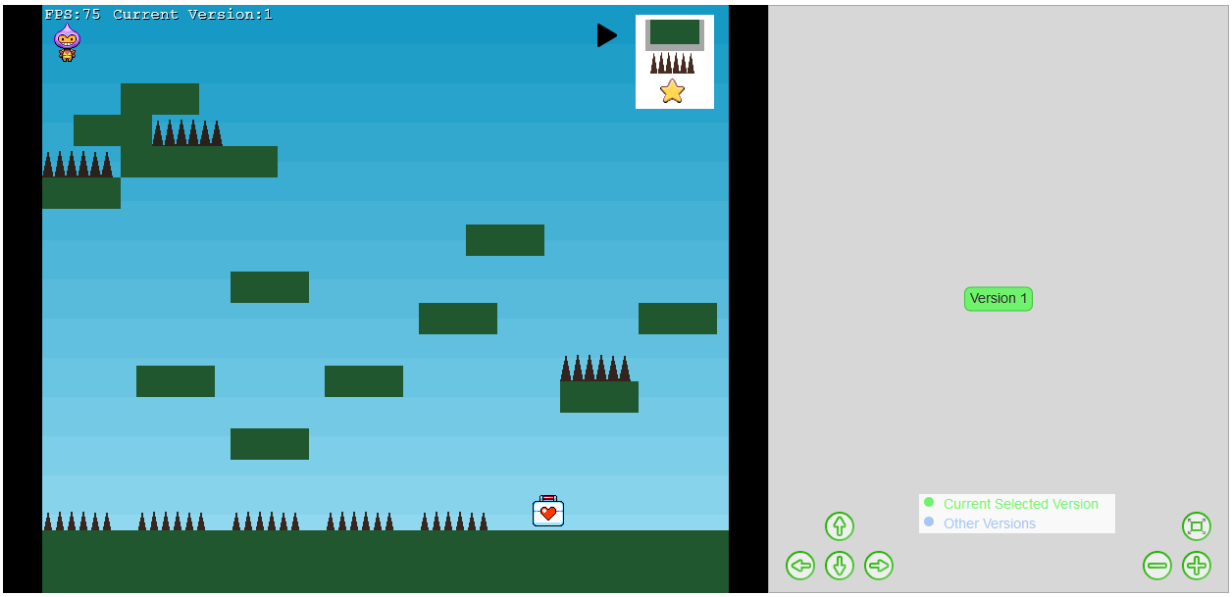


Figure 5.1.8 Task 6 to create separate difficulty levels with a specific number of ledges and spikes.

The final task asked the participants to create as many levels as they prefer without any restrictions or constraints on how many spikes or stars were to be used in any level. This task was left open-ended to allow participants to show their creativity while designing levels of various difficulties. It allowed us to see how the participants designed levels with all the experience of working with our

versioning tool in previous tasks. We encouraged participants to think aloud while performing this task to understand how their workflow is being aided by our versioning tool.

5.1.6 Results and Observations

In this game level editor study, all the participants were successful in performing all the tasks. They did not have any issues understanding the concept of creating versions of their levels. They were able to successfully interact with the version tree and grasped the concept of a parent-child relationship between the nodes. They were able to utilize all the features of our versioning tree that we were expecting them to use while designing levels such as preview changes in other versions while hovering on a node, creating new versions, and switching to new versions.

5.1.6.1 Use of versions and trees in different tasks

All participants previewed different versions by hovering over the nodes in the tree (**Table 6** shows the number of version previews done for each task). They mentioned in the post-study feedback that they liked the preview feature that overlayed the version currently being hovered on the screen at the same time. One of the participants commented on using preview feature, *"I liked you can see previous version in a task specific thing"*.

Previewing other versions made it easy for the participants to compare and make changes to their current version. One of the participants mentioned, "*hovering [on the nodes] over and over to see the difference helped me identify which was easier level that I designed*".

Table 6 shows that it was only for task 7 where 2 out of 10 participants did not use the preview feature. Since task 7 was an open-ended design task with no restrictions on the number of sprites, collectables (stars), or time limit, it was expected that some participants would be focused on just designing levels instead of comparing multiple versions that they were creating. Also, those two participants, who did not use the preview feature, created just a single version of their game level and did not switch to any other version during the whole task. Three more participants made zero version switches in task 7, i.e., they did not switch to another version of the level (see **Table 7**), however, they still utilized the preview feature to compare their version changes.

Participants	Task 3	Task 4	Task 5	Task 6	Task 7
1	5	4	7	2	3
2	8	3	5	1	0
3	7	3	3	2	6
4	3	5	3	0	7
5	5	3	7	6	0
6	8	2	4	3	1
7	6	5	6	1	3
8	4	2	4	3	1
9	2	3	8	1	2
10	6	4	6	4	2
Average	**5.4**	**3.4**	**5.3**	**2.3**	**2.5**

Table 6 Version hovers (preview) by participants for tasks.

Participants	Task 3	Task 4	Task 5	Task 6	Task 7
1	1	2	0	0	0
2	2	2	4	0	0
3	2	2	1	1	2
4	1	1	2	2	3
5	3	1	4	4	0
6	1	1	2	1	1
7	2	5	2	1	1
8	1	2	2	1	0
9	1	2	4	0	4
10	1	1	3	1	0
Average	**1.5**	**1.9**	**2.4**	**1.1**	**1.1**

Table 7 Version switches by participants for tasks.

Despite the fact that all participants used the undo-redo actions during their second task, only one participant used it to fix their mistakes (unintentionally adding, deleting, or moving sprites). The remaining participants either manually corrected their mistakes or just switched back to the previous version and started over. They reported that it was easier to go back to the old version to

quickly set things to their old positions if they have not introduced many changes to the level already. One participant remarked on the version switching feature that, *"[I] wasn't worried about breaking the system as it is easier to go back".*

We also noticed the participant who had the most experience with versioning systems (like Git), created more versions and previewed more versions than the other participants. Participant 10 has the most version hovers and switches, as shown in **Table 6** and **Table 7**. **Table 8** shows that while designing levels, the same participant also created the highest number of extra versions. This could be because the participant is used to the idea of working in a non-destructive manner and plan ahead of actually doing the work, even when they are just exploring alternative design ideas.

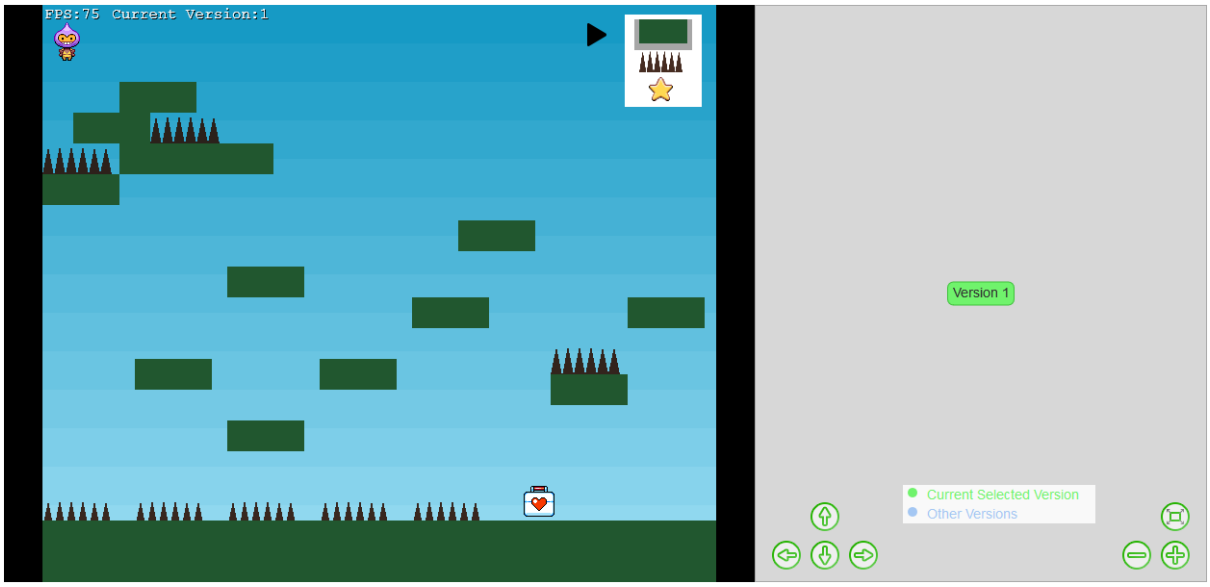
Participants	Task 3	Task 4	Task 5	Task 6	Task 7
1	0	0	0	0	0
2	0	1	0	0	0
3	0	0	0	0	1
4	0	0	0	0	0
5	0	1	0	1	0
6	1	0	0	0	0
7	0	0	0	0	0
8	0	1	0	0	1
9	0	0	1	0	1
10	0	1	4	0	4
Average	**0.1**	**0.4**	**0.5**	**0.1**	**0.7**

Table 8 Extra versions (besides the required ones) created by participants for tasks.

We observed two versioning approaches during the study – pre-versioning (version-then-change) and post-versioning (changes-then-version). Six out of ten participants made all their desired changes to the current version of a level before they saved their changes to a new version. However, when asked if making changes before saving to a new version is their preference, 2 out of those 6 participants said that they would prefer to save the versions first to avoid changing their base level accidentally.

To illustrate an example of pre-versioning, see **Figure 5.1.9** where we can see that one of the participants has a single version of a game level available (**Figure 5.1.9 (a)**). They made their changes by placing the stars around the level and moving around some spikes without making a new version (**Figure 5.1.9 (b)**). After they are done making their changes and playtesting the level, they saved their changes to a new level called “easy” (**Figure 5.1.9 (c)**).

(a)



(b)

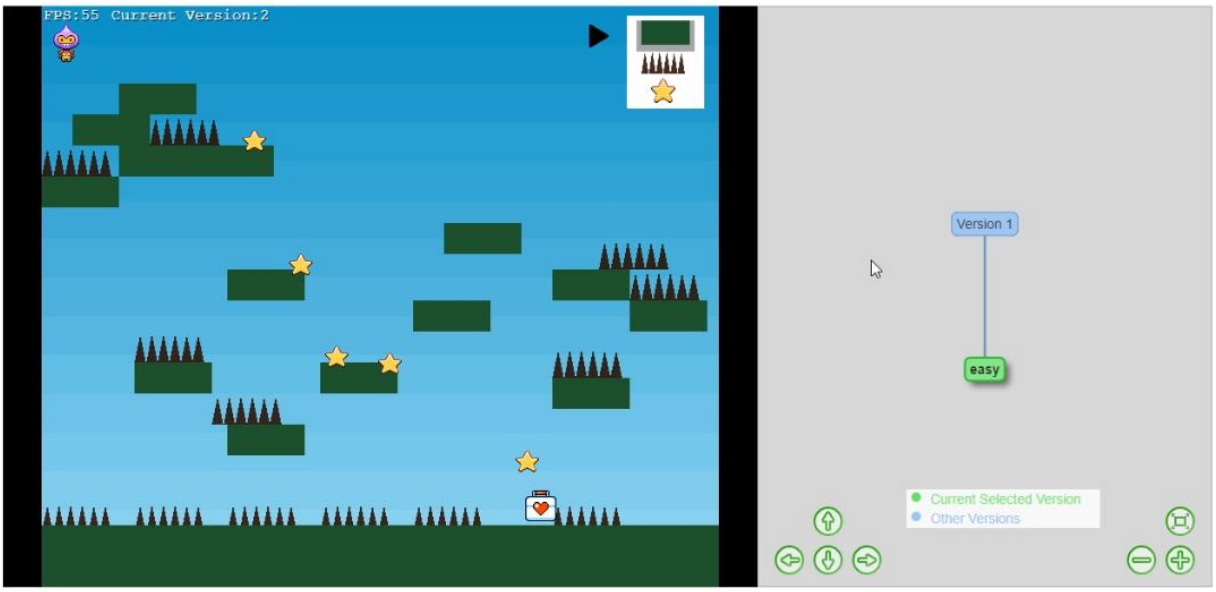
(c) 

Figure 5.1.9 Example of creating a new version where (b) changes are made by a participant first and then (c) changes are saved to a new version ‘easy’.

5.1.6.2 Observations for creating new versions and switching to previous ones

We found that some of the circumstances to create new versions (discussed in **Section 3.2**) were also observed in this study. All participants except one created new versions to avoid losing their work while making changes to their current game level. Three out of 10 participants created new versions to explore alternatives while designing levels of varying difficulties. One participant mentioned that they created a new version because they were testing their changes and were not sure if they were going to keep them or not. The rest of the circumstances such as collaborating with other users and time-based versioning were not observed (as expected) because the study did not include a collaboration process or allow time-based versioning since the users were only allowed to do explicit versioning.

We also observed that participants went back to their previous versions following some of the reasons mentioned in **Section 3.3**. All participants made a few mistakes while designing levels where they had to either use undo to fix the mistake or switch to a previous version to start from scratch. One participant went back to their previous version after testing their new changes which

they didn't like. All participants except one created alternate versions of their levels and therefore switched back to a previous version which they used as a template. We also noticed that 7 participants ended up with 2 or more branches when creating alternate versions of their levels. Other reasons such as broken or corrupt files or changes in requirements were not observed since the study was designed with pre-determined tasks where requirements didn't change and our application did not crash during the study that could have resulted in broken or corrupt files.

5.1.6.3 Responses and feedback

We asked the participants a few questions after the study to get more insight into their experience with the versioning tool.

5.1.6.3.1 Did you have any issues understanding or working with the versioning tool?

None of the participants reported having issues understanding the versioning tool. They were able to save their changes in the level to a new version, however, two participants did experience a moment of confusion when they lost their changes as they switched to another version without saving their changes first. One of the participants suggested that *"maybe there should be * on the name of the version you are changing to show that this version is being changed"*. When a user tries to switch to a different level, having a visual indication that some unsaved changes are present in the current level could potentially prevent users from losing their changes by accident. While working on tasks 5 and 6, two participants stated that they wanted to merge some of the versions because they assumed they could combine them to create a new version. They suggested that a merging option would be useful when a user wants to use some of the work (e.g., creating a pattern of sprites) that they have already done in another version.

5.1.6.3.2 Did prior familiarity with versioning systems help you?

The participants familiar with versioning agreed that their prior knowledge of versioning systems did help them understand the versioning tool quickly.

One participant with relatively less experience with such tools remarked that it was easy after the practice tasks and another participant with similar experience said that *"once [I] got familiar with the interface, it was easy to use"*.

5.1.6.3.3 Do you think the versioning tree was helpful? If yes, why and how did it help?

All participants responded affirmatively. One participant mentioned that it was easier to keep track of everything that they have done since the start. Another participant remarked similarly that *"one glance and you can read everything in a tree"*. Some participants mentioned that colors for the nodes were helpful and the tree *"helps you see what you are creating, and which version are you working with"*. Most of the participants remarked that seeing the preview of another version and seeing the difference was quite helpful.

One participant compared our versioning tool with the traditional versioning GUIs as, *"Git GUIs are confusing, and this [tree] is like an add-on"*, while another participant remarked that the visual differences which you can see are better than the textual or numbered differences found in most versioning tools.

5.1.6.3.4 Do you think users not familiar with any versioning tools will find working with this versioning tool easy?

All participants agreed that even if the users who are not familiar with any versioning tools will be able to understand and use it easily. One of the participants remarked that anyone can become familiar with the versioning tool quickly with proper guidance and a good tutorial. One participant responded that *"[It was] easy to convey the [basic] concept to non-programmers as you don't have to understand all the concepts of git to understand [the tool]"*. Another participant noted that with enough practice, the users not familiar with versioning tools will be able to use the tool efficiently. It was also mentioned by one of the participants that, *"it was beginner friendly and will make their life easier"*.

5.1.6.3.5 Any other feedback?

Two participants remarked that the tool would be more helpful for bigger projects rather than small projects, while another participant mentioned that they would like to have an option to duplicate the version first before making changes. The participant also expressed concern about doing so because it would allow for having many duplicates without any changes. Another participant mentioned that *"creating templates is not hard to remember but easy to forget because I was getting wrapped up in what I was doing, so I automatically ctrl+s to save and then realize I just saved over [previous version]*".

5.1.6.4 Post-hoc tree representations

Section 4.5 discusses how we integrated auto-versioning into our versioning tool as we wanted to explore how it will affect the version tree being generated. We generated different tree representations using auto-generation methods to save a new version - play, 3-changes, every-change, and time-based on the same tree that was created when the participants explicitly created a version using a keypress (Ctrl+Shift+S). The *play* method generated a tree with nodes created on every play button press. The *3-changes* method created a new node on every 3-changes made in the level by the participant. In the *every-change* method, a new node is created on every change made by the participant in the level which included adding, deleting, and moving the sprites along with changes made with undo-redo. The *time* method created a new node every 3 seconds (arbitrarily chosen value) working in the editor.

From **Figure 5.1.10** and **Figure 5.1.11**, it is evident that the trees generated in case of play and explicit save methods are almost equivalent and there are fewer nodes in their trees as compared to other trees (**Figure 5.1.11** and **Figure 5.1.12**). This could be because participants explicitly saved levels (that led to the creation of nodes) only after they made some significant changes. Similarly, they playtested the level once they were satisfied with their level design. However, the number of nodes in the tree, created by the play method, could grow depending on how many times the participant wants to test the level. This was observed for a few participants who playtested their levels frequently.

The every-change and 3-second methods created a few long parallel branches for all the participants because of a high frequency of changes in their designs in the case of the every-change method and fixed time interval version saves in the case of the 3-second method. These long-branched tree structures could be difficult to comprehend for the users because of the high number of nodes, but they can prove useful where the user wants to look at a fine granularity of changes being made in the design. The 3-change method, however, among other methods strikes a balance between storing enough useful change states (versions of level that contained changes that a user may consider important) and a manageable tree structure (relatively less number of versions making it easy to navigate the version tree).

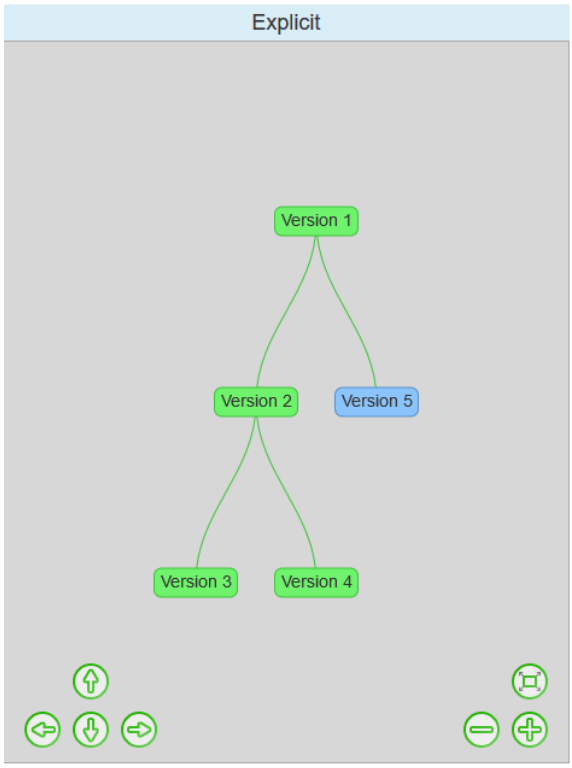


Figure 5.1.10 Tree generated by explicit saving (default method).

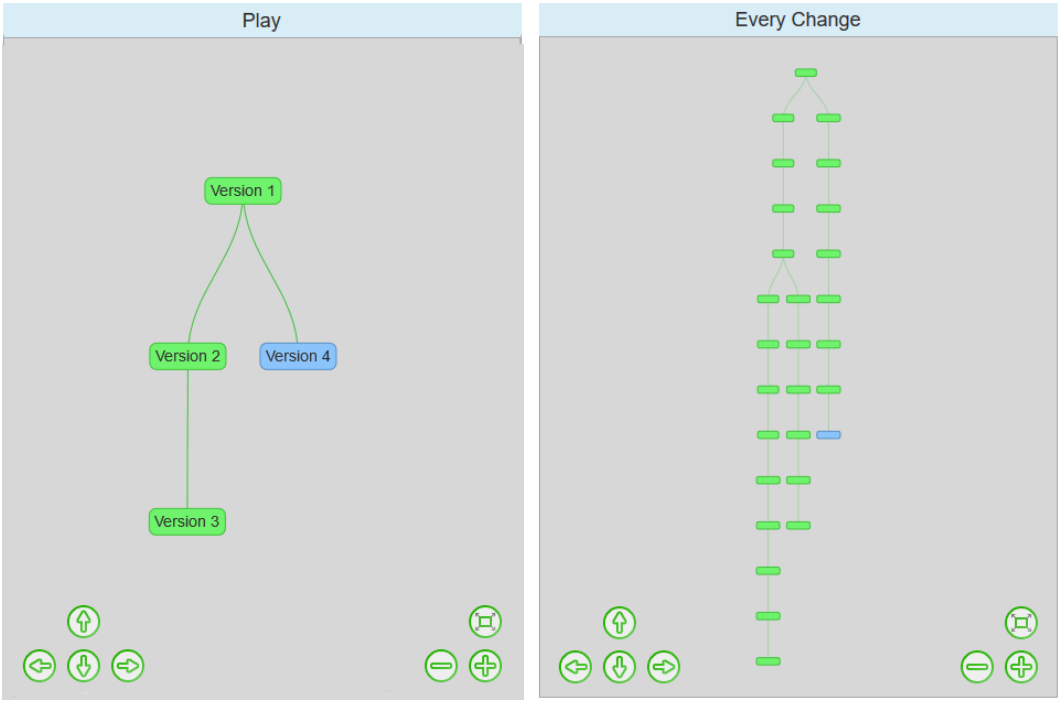


Figure 5.1.11 Trees generated by 'Play' and 'Every Change' methods.

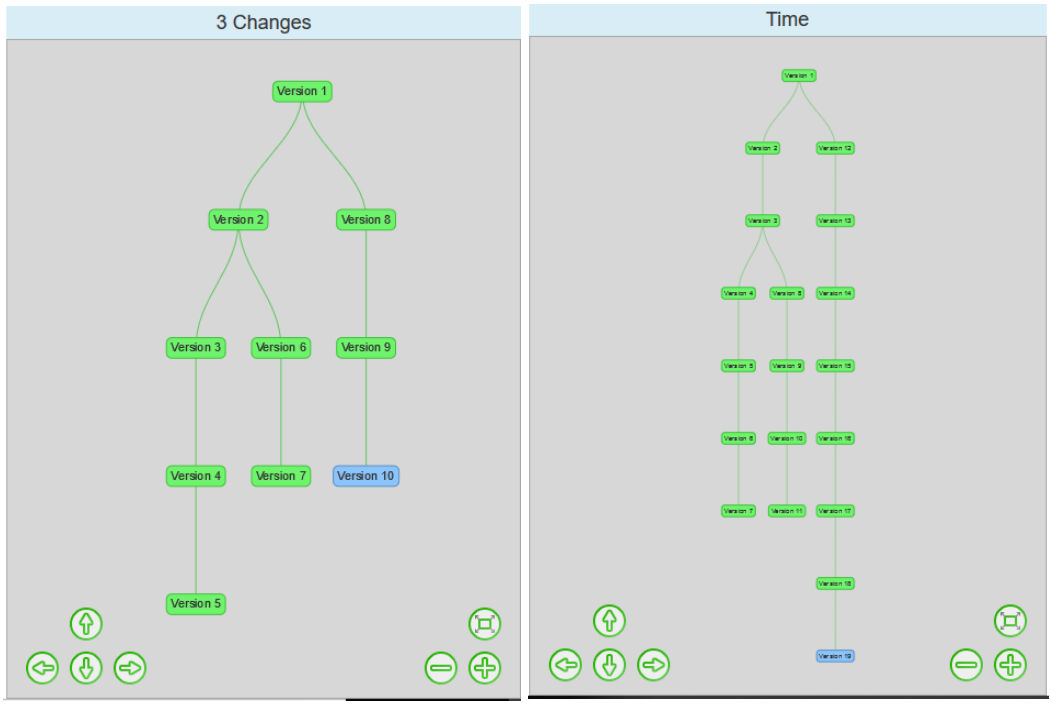


Figure 5.1.12 Trees generated by '3 changes' and '3 second Time' methods.

5.2 CASE STUDY: WEB ANALYSIS TOOL

In this subsection, we report a case study we did to test our versioning tool in another interactive environment, i.e., a web analysis tool. The goal of this case study was to observe how our versioning tool could be used in a setting where the UI state of the application (interactive elements of a web form) was stored instead of the result (output) in a version and to gather information for improving the future designs of the tool.

5.2.1 Usage

We wanted to test our tool with users in research who were working on projects that involved performing detailed analysis with multiple parameters over a long period. Therefore, our tool was used for a week by two users who were working on a real-world web application for carrying out analyses in microbiology (Winnowing Pipeline project). We demonstrated the working of our tool to them, and they were asked to use the web interface (explained in **Section 4.5**) to perform all of their data analyses. We interviewed the users and asked them about their experience and their opinions about our tool, discussed in the following subsection.

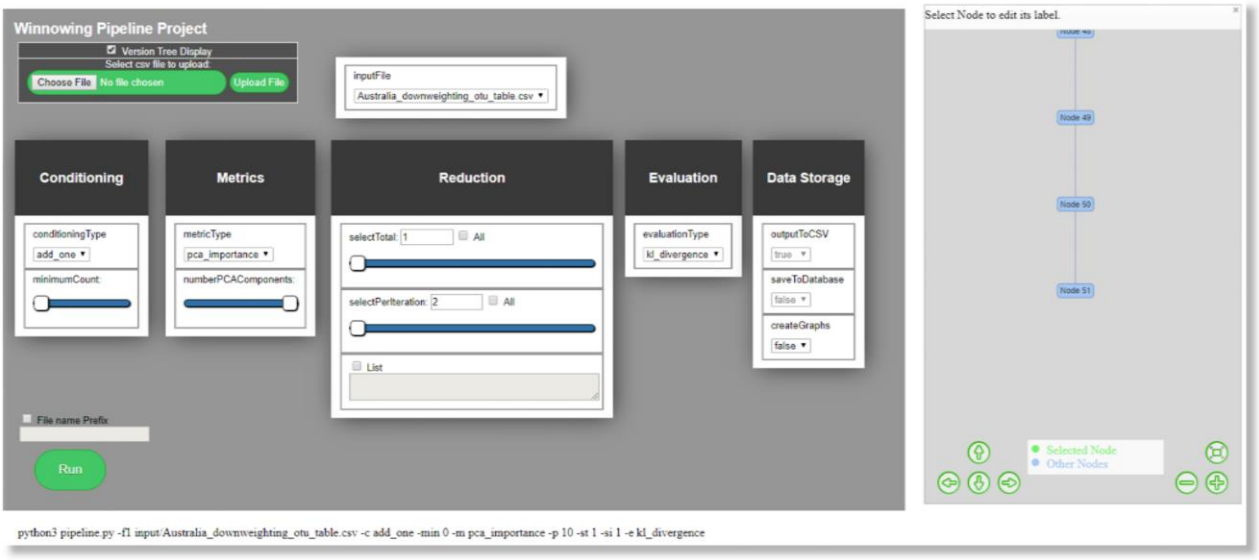


Figure 5.2.1 Web Analysis Tool interface with version tree.

5.2.2 User Feedback and Observations

In each version, the versioning tool saved the application's UI state, which contained a set of parameters in a python command format. All versions were presented in the version tree, allowing users to see their progress and quickly look over and review the commands they had previously run using the tool's hover capability. But with the increased number of commands executed, the number of nodes increased, which overwhelmed the users' ability to see how they have progressed. The users reported that it was useful to see the nodes to a certain extent, but it proved difficult to find the commands they were looking for - if they were looking for the commands in the tree. This was mainly because the nodes are not labeled as intuitively as they would have liked. The nodes were labeled as Version 1, Version 2, Version 3, and so on. They were not descriptive unless they hover over the nodes to reveal the command that was executed in that particular node. Moreover, when users ran the analysis in a batch (more than one command), it quickly created a lot of nodes and eventually became difficult to keep track of the executed commands.

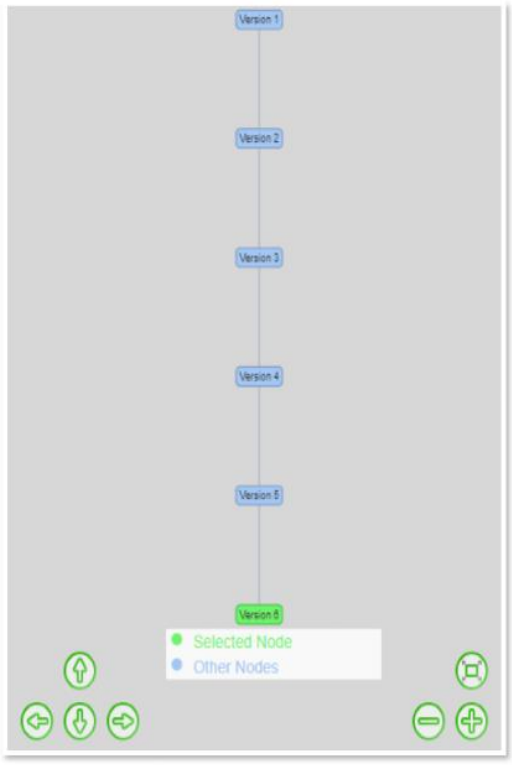


Figure 5.2.2 Single branched version tree after running analyses in a batch.

Another issue that surfaced with running analyses in a batch was the construction of a linear version tree. The tree generated, as seen in **Figure 5.2.2**, was a long list of nodes connected without a lot of branching. This result was not desired by the users who suggested that they would like to use a feature that can automatically branch the nodes in a tree based on the classification by some parameters. For example, the "metric type" parameter has two options - *pca_importance* and *graph_centrality* where the *pca_importance* deals with the number of PCA components, and *graph_centrality* include the correlation type, correlation property, centrality type, and other options. These parameters can be used to branch the tree structure so that the nodes that are created by a change in certain parameter values can be grouped as branches. But, classifying versions based on parameters is a challenging problem. When a single parameter is changed, a new version can be grouped with other versions that contain the same parameter changes; however, when two or more parameters are changed, a new version can belong to two or more groups of versions. This alternate representation of versions differs fundamentally from the way we currently organize versions in our version tree. The current method organizes versions using a parent-child relationship, in which a child version contains the data stored in a parent version as well as any new changes made to that data. In a parameter-based classification tree, on the other hand, some parent versions might not even hold any version data, but they can serve as branch nodes for a single parameter, allowing other versions containing changes for the same parameter to be grouped as children.

If the problem of classification of versions can be properly addressed, and a tree can be branched based on parameter requirements of the users, then the same single branched tree (**Figure 5.2.2**) could look like a tree in **Figure 5.2.3** and will be easier to navigate and comprehend. Moreover, the constrained parameter space allows the creation of a prospective version tree showing possible future designs similar to GEM-NI's design generation (**Figure 2.3.7**) based on parameter selection with Cartesian product [50].

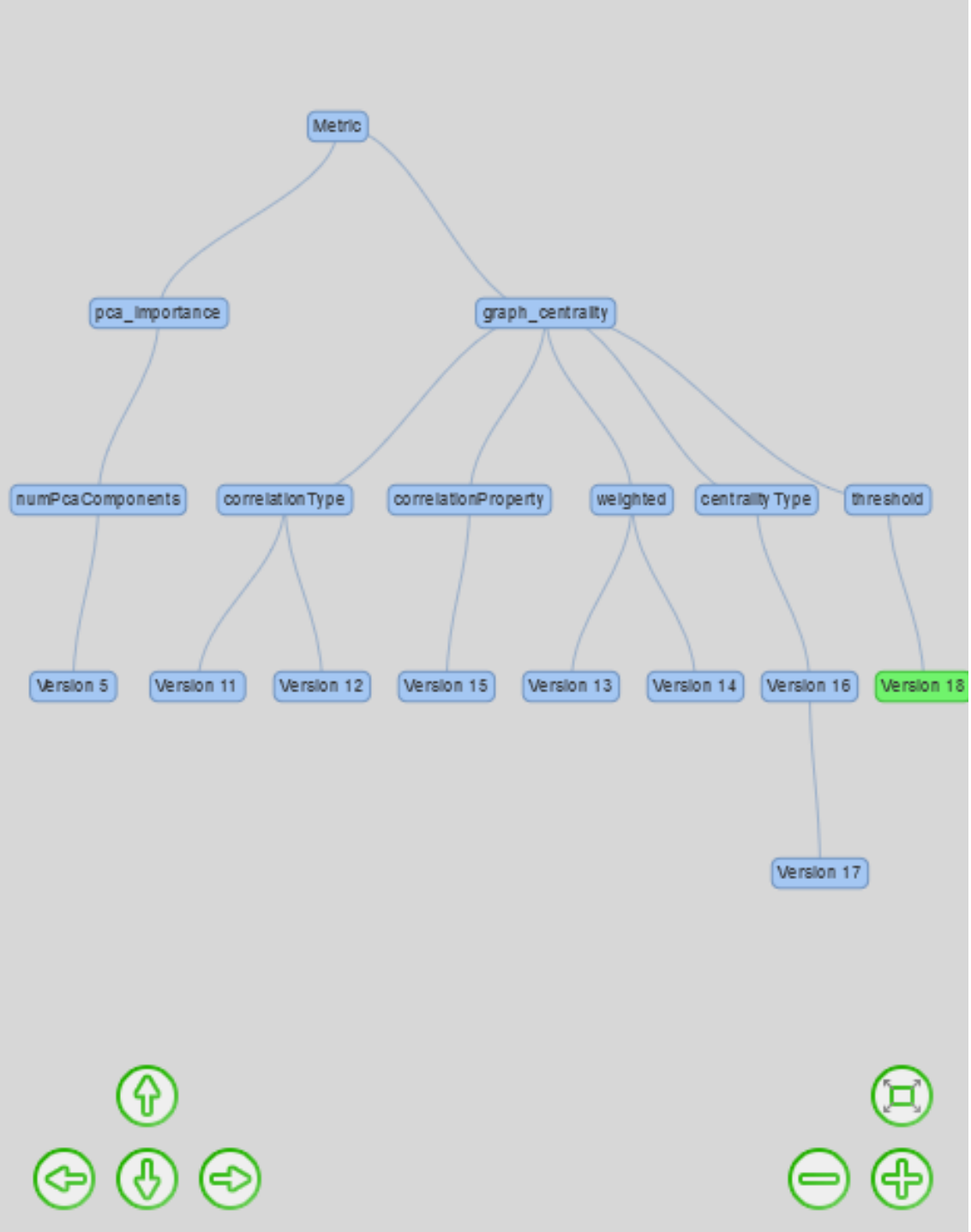


Figure 5.2.3 A mockup of a tree structure after classifying nodes based on parameters.

5.3 SUMMARY

In this chapter, we have discussed two user studies we performed to test our versioning tool in two interactive web-based applications – game level editor and web analysis tool. The first user study focused on three goals: evaluating the version tool where each version stored document/model data of an application, confirming circumstances and scenarios identified for creating new and revisiting previous versions, and identifying key elements that can be used to improve our tool. The second user study focused on testing the tool in a real-world application where UI state data of an application was stored in each version. We reported results on how the participants used our versioning tool in design and analysis tasks in both studies. We also discussed the observations we made and the feedback we received from the participants during both user studies. We also looked at the alternative version trees, created implicitly in four automatic version trigger mechanisms (play, 3-changes, every-change, and time), and compared them with the explicit version tree created by the participants during the tasks.

CHAPTER 6

DISCUSSION

In this chapter, we discuss findings and observations from our two studies, present explanation for the main findings, and consider how our results can be used to improve versioning tools in interactive systems. The discussion is organized around eight key elements in this chapter - versioning patterns, templating, branching, need for merging versions, using previous versions as a reference, versions as undo alternative, managing a large number of versions, and control over the granularity of saving versions.

6.1 VERSIONING PATTERNS

In our game level editor study, when participants were creating new versions while performing the design tasks, we observed two different types of versioning patterns – pre-versioning and post-versioning.

Pre-versioning involved the creation of a new version prior to making any changes to the system. The participants who used pre-versioning ensured that they had a structure in place for the number of levels they wanted to create; they could then modify these versions according to their needs. This pattern is also seen in file-based versioning where users create duplicates of their files and rename them before editing.

Post-versioning involved making changes to the system before saving those changes to a new version. It was used when participants worked on the current game level to introduce their desired variations to the level, tested those changes, and then saved those changes as a new version. This is similar to what is observed in VCS tools where changes are made to the files first and then new commits are made to save those changes in the version history.

In both cases, the resulting tree structure will be the same (**Figure 6.1.1**). Consider a ‘Base’ version containing a single star shape which should be filled with yellow color. In the case of pre-versioning, the user will create a new version called ‘New Version’ first, switch to this new version, fill the star shape with yellow color, and then save the changes. On the other hand, in the case of post-versioning, the user first makes changes in the ‘Base’ version itself, i.e., fills the shape with yellow color, and then saves these changes to a new version.

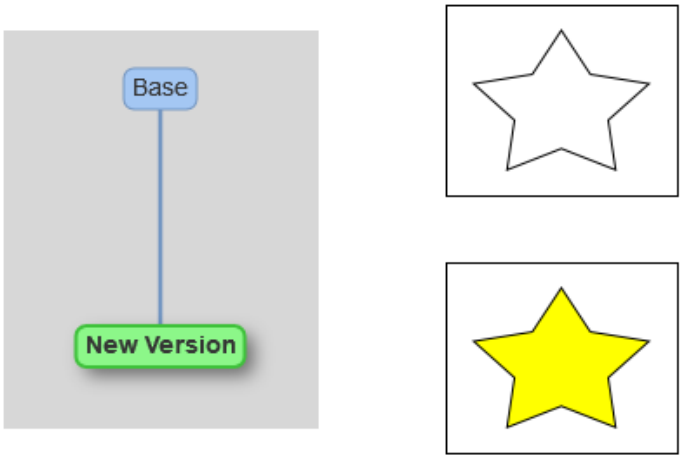


Figure 6.1.1 Left: Creation of a new version from a base version. Right: Artifact modification from base version (star shape) to saved changes in a new version (fill color).

One disadvantage of using the post-versioning pattern is that there is a chance of overwriting the current file with the changes that were supposed to go in a new file. For example, suppose a user is working on a text document named ‘file.txt’ and then decided to make changes to it later and save those changes to a separate file called ‘file_new.txt’. But instead of saving the current file changes as a separate file, the user accidentally saves the file using the ‘Save’ option instead of ‘Save As’. Now, the changes that were supposed to be saved as a separate version were saved to the same file, overwriting the contents of the original file. This was also observed happening with our versioning tool in the game level editor study when participants accidentally saved their changes in the same version rather than creating a new version. Therefore, avoiding chances of loss

of original data becomes a matter of concern and can be prevented by following the pre-versioning approach by the user. Moreover, the automatic pre-versioning feature can also be integrated into our versioning tool.

6.2 TEMPLATING

Graphics editing software applications such as Photoshop allow users to create multiple layers and groups [122]. With this feature, users can use their layers or groups as templates to extend or modify their designs by duplicating these layers or groups. For example, if a user creates a character design with one color that consists of multiple layers, then these layers can be grouped together in a layer folder which can then be duplicated to be used as a template for more characters with different variations such as color, size, and rotation. Another common templating mechanism is to save a base file that can then be used as a template to create new files. Photoshop also has Smart Objects [123] that can be used as templates for creating derivative objects. Therefore, designers may often create multiple instances of their designs using layers or smart objects to speed up their design process.

Similarly, in our game level editor study, we observed that participants made use of their older versions as templates for more complex levels that they wanted to create while performing design tasks. This provided them with a good starting point to design new levels instead of starting from scratch. Creating templates allowed our participants to quickly duplicate a version from the base work and then make changes or modifications to this new version, thus saving time.

Moreover, users sometimes need to explore different design paths such as designing game levels or designing character art where the project isn't limited to just a few layers. In many such settings, templates can be a huge help because they reduce time and effort to create a copy of the design allowing the designers to produce a variety of designs quickly from a single template.

6.3 PREVIOUS VERSIONS AS REFERENCES

Participants also used their previously created versions as references to compare the changes within the current version they were working on using the preview feature of our tool. To do so, they needed to hover over the version that they wanted to view and the objects in that version were visible with full opacity while the objects in the current version were shown with transparency. Participants made comparisons for the locations of objects and patterns across the two versions, at first to see if they correctly moved the pattern of sprites from one location to another (Task 4) and then to see how their current level differs from their previously created version (Task 5, 6 and 7).

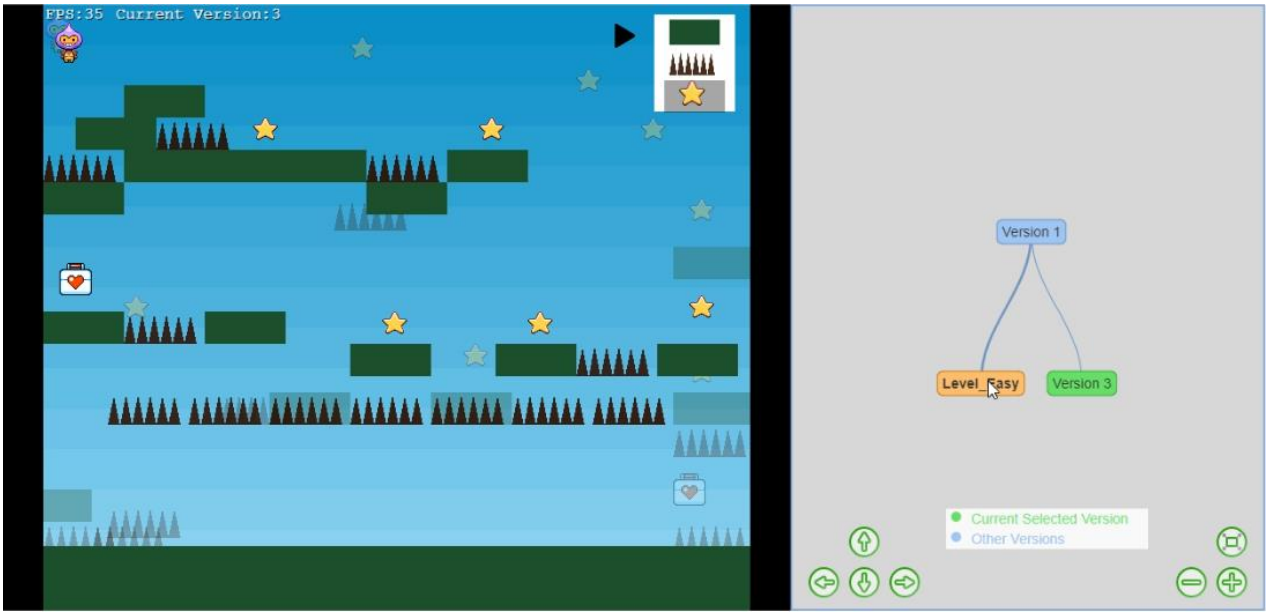


Figure 6.3.1 The objects of another version (hovered over by the user, 'Level_Easy') are overlaid on top of the transparent layer of objects from the current version, 'Version 3'.

Such quick visual comparisons inside the editor itself can be very useful in improving the workflow of a designer working on adjusting a level. This is not an obvious feature of versioning tools but when versions are easily accessible as they are in our versioning tool, it makes it easy to use them for other tasks like diffing (comparing two versions).

6.4 BRANCHING VERSIONS

In the last task (task 7) of the study, where the participants were asked to make any number of levels without any constraints on using a certain number of sprites, we observed participants working with their version tree ended up with multiple branches for creating game levels with varying difficulty (see **Section 5.1.6.2**). This happened because participants wanted to use previous versions as templates to create new versions of levels. The easy-to-play levels were branched out to create more complex (difficult to play) levels as it allowed the participants to quickly add more spikes and ledges without designing from scratch.

Figure 6.4.1 shows the work of a participant where branches at depth two of the version tree were created as testing versions to later branch them to create final versions 'easy' and 'difficult' at depth level three. This shows that they created a base level for themselves to branch out to new versions for more complex levels.

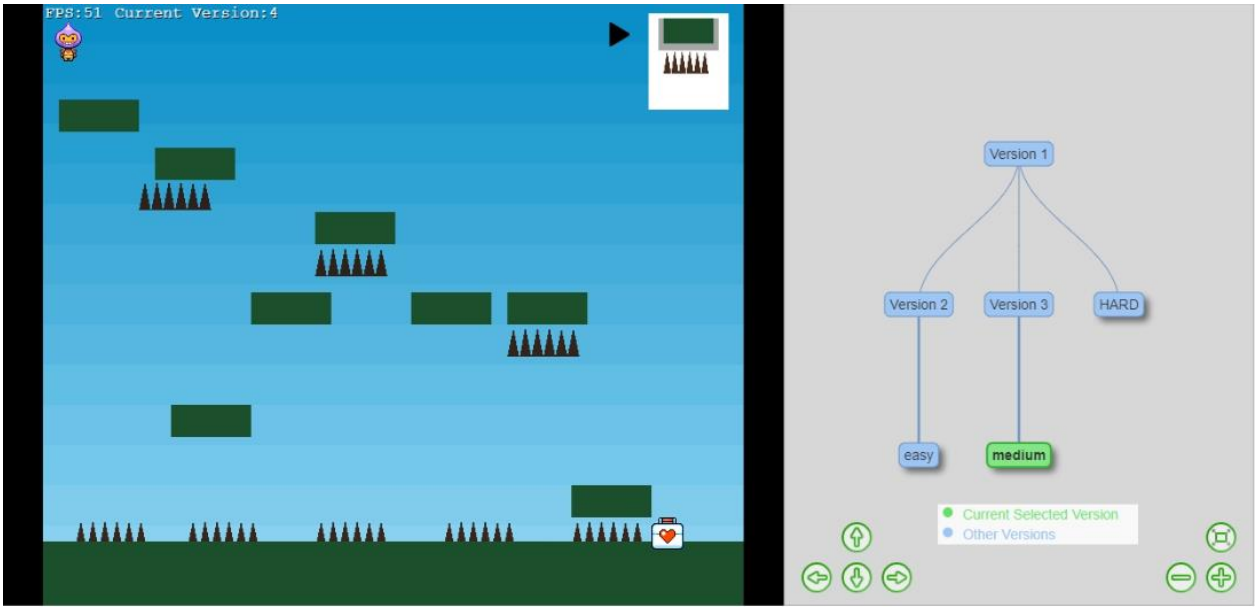


Figure 6.4.1 Branching in a version tree.

Branching can also allow the participants to identify versions based on some attribute (e.g., difficulty level) because versions with similar attributes are usually grouped in the same branch.

For example, a designer working on alternate designs of a hard level can easily identify all versions that are hard to play since they can be found grouped in the same branch.

6.5 NEED FOR MERGING

A couple of participants stated that they wanted to merge some of the versions to create a new version (see **Section 5.1.6.3.1**). Merging refers to a process of combining changes from multiple source branches into a single target branch that will initiate a conflict resolution process if changes are incompatible [36]. In other words, merging is a process to combine changes from two different versions from two separate branches to create a new version.

Although this feature of merging the nodes is not available in our game level editor, it could prove quite useful in generating a new design that blends two separate versions. However, it does require careful consideration on how to merge two versions as the number of objects present in both the versions would increase the number of collisions (conflicts between changes in the same file) that the user will need to resolve to decide as to which version of the object the user wants to keep in the newer version. One way to do this is to merge non-conflicting changes and then ask the user to manually select the conflicting changes from one of the versions they are merging.

6.6 VERSIONING AS AN UNDO ALTERNATIVE

We found that most of the participants did not use undo feature to fix their mistakes while designing a level (see **Section 5.1.6.1**); rather, they just switched to some previous version to start over. One of the major reasons for reverting by selecting a previous version could be the ease of going back to the previous versions (clicking on the nodes of a tree) than doing undo multiple times (pressing Ctrl+Z). But for situations where no previous versions are available in a tree, the need for undo-redo functionality is fairly obvious. When users are exploring the design space, quickly switching between versions by selecting them in the version tree can help save a lot of time.

Another valuable feature of using versions as an undo alternative is the ability to go back multiple versions with a single click (**Figure 6.6.1**) in the interaction history. This could be considered as a one-step multiple-undo which can be useful when the user does not want to keep any changes made after a particular version. For example, in Adobe Photoshop's history panel (see **Figure 4.2.2**), a user can undo multiple steps at a time. However, going back to an older version in a version tree and making new changes does not affect other versions since it will create a separate version (**Figure 6.6.2**), whereas the previous history is lost when a new change is made in a history list as explained below.

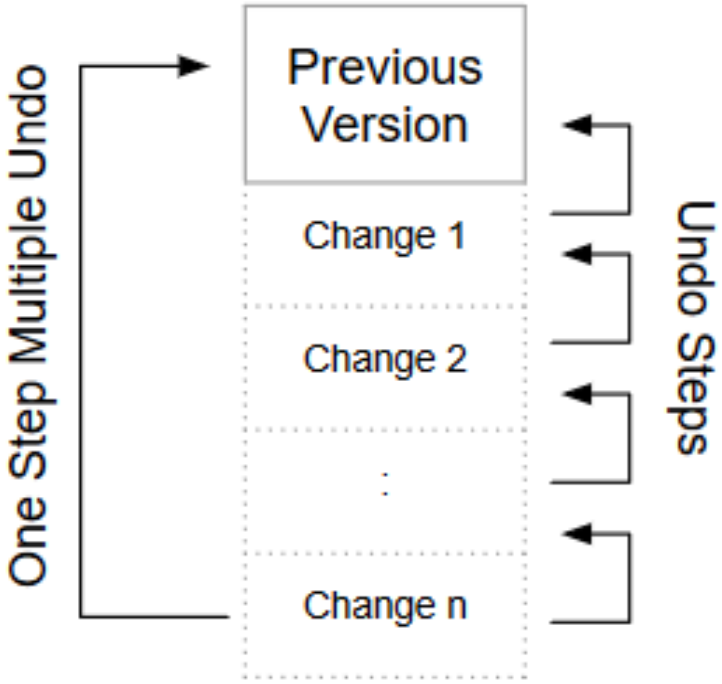


Figure 6.6.1 One-step-multiple-undo (left) when a version acts as an alternative of multiple-undos (right).

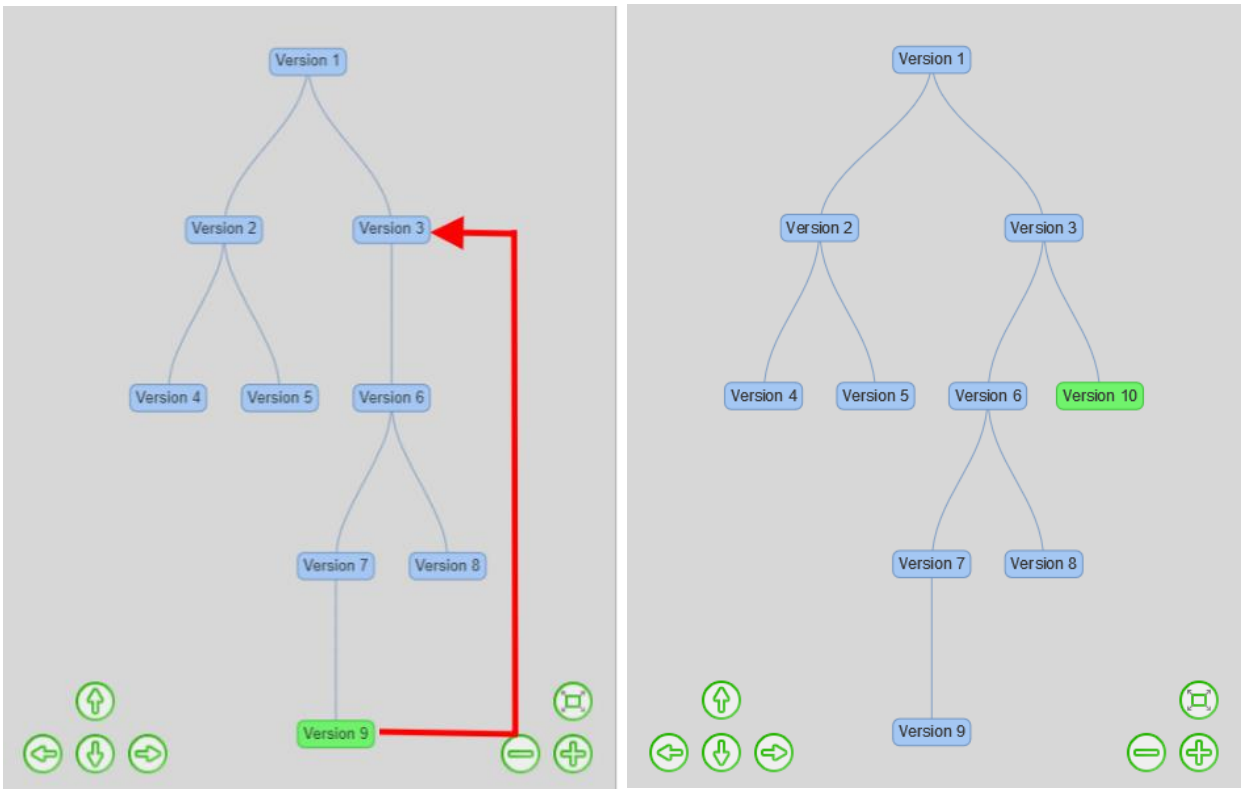
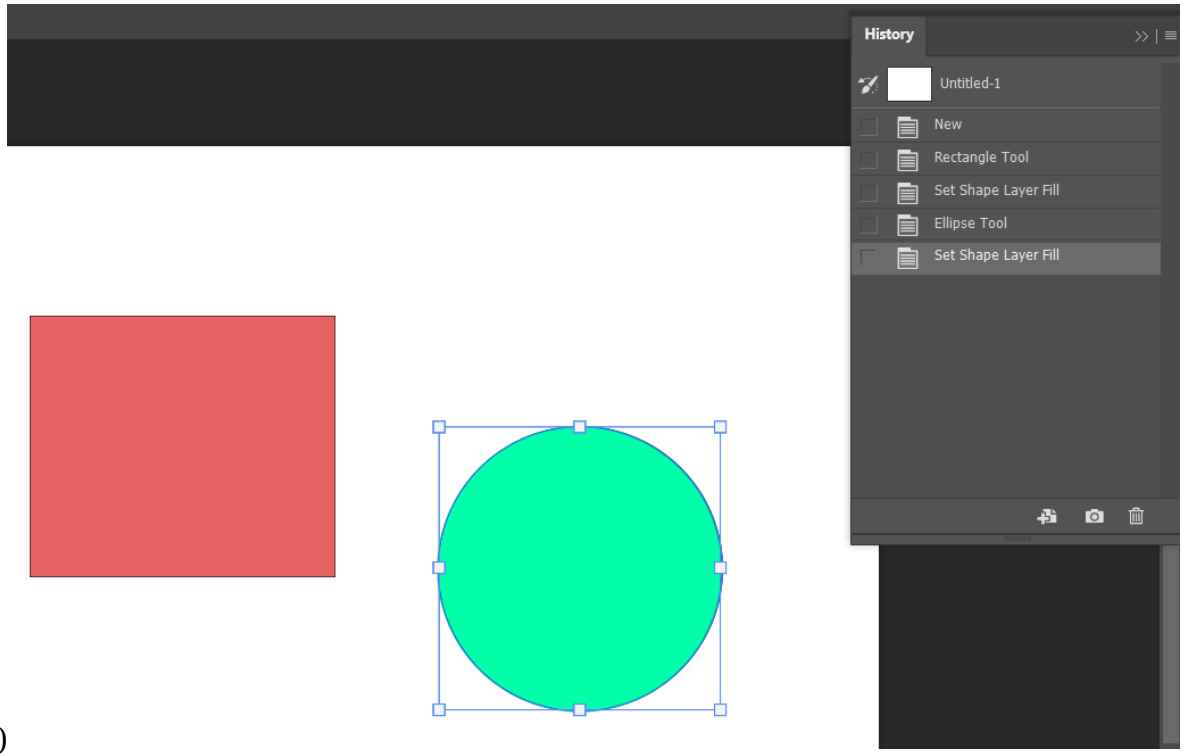
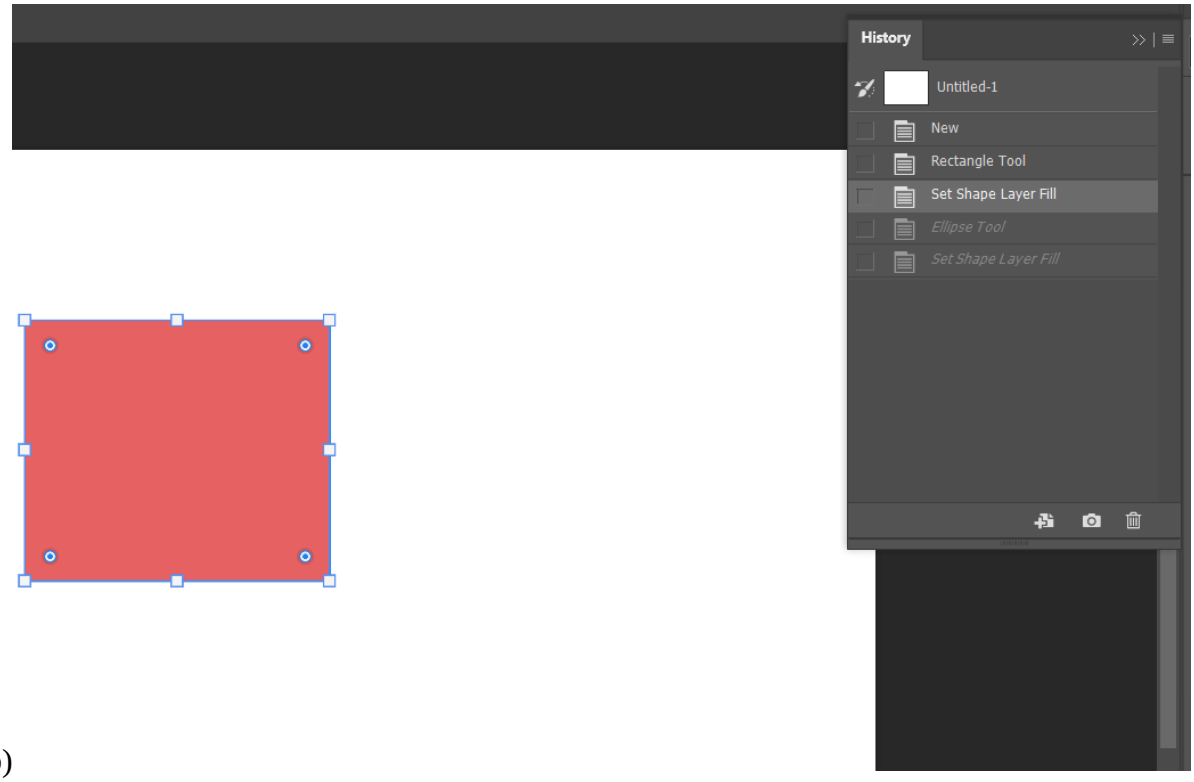


Figure 6.6.2 Version tree before a user switches from 'Version 9' to 'Version 3' (left). A new version 'Version 10' is created as a child to ‘Version 3’ while still preserving old branch (right).

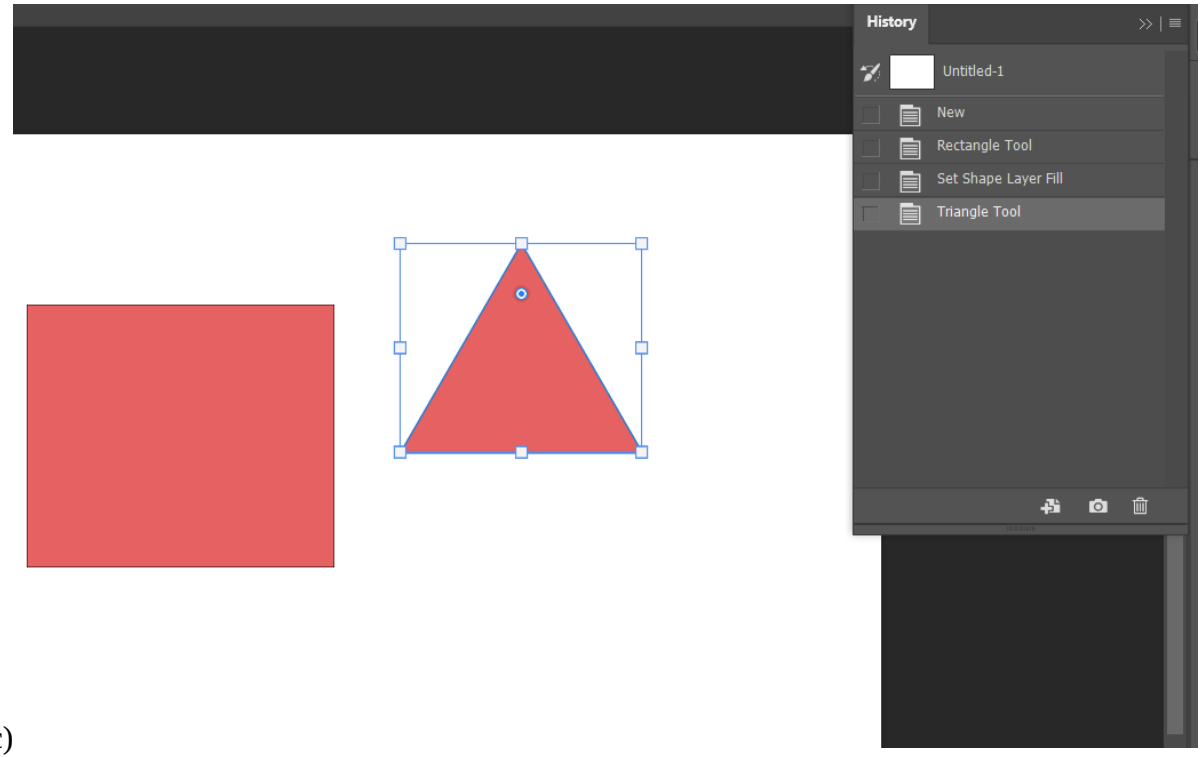
In **Figure 6.6.3 (a)**, we can see that a user created two shapes (rectangle and ellipse) and changed their colors. When that user reverted changes for the second shape (by selecting the step of “Set Shape Layer Fill” for rectangle in the history panel) to only have a rectangle in the drawing, we observe (**Figure 6.6.3 (b)**) that two of the steps in the history list have their font colors darken to indicate that these steps have been reverted. Lastly, when a user creates a new shape triangle, we can see (**Figure 6.6.3 (c)**) the previous two steps get deleted from the list and a new “Triangle Tool” step has been added. This indicates that the history is not preserved in Adobe’s History Panel when we make a new change after reverting to the previous step.


History
Untitled-1
New
Rectangle Tool
Set Shape Layer Fill
Ellipse Tool
Set Shape Layer Fill

(a)


History
Untitled-1
New
Rectangle Tool
Set Shape Layer Fill
Ellipse Tool
Set Shape Layer Fill

(b)



(c)

Figure 6.6.3 (a) History list with two shapes in the drawing (b) History list when reverted to previous step (c) History list losing history when a new change has been made.

6.7 MANAGING A LARGE NUMBER OF VERSIONS

In the web analysis tool, users were able to explore the multidimensional parameter space rather than spatial arrangement (as in the game level editor). When the participants ran multiple analysis commands in a batch, the version tree created auto-generated versions in a single branch (see **Figure 5.2.2**). The version tree became difficult to navigate and understand. This is because the auto-generated versions are not given meaningful and descriptive names to identify the commands run that make up a particular version, and a tree with a single branch does not help in identifying the nature of connections that could be established among various versions representing particular locations in parameter space.

One way to manage these versions is to branch them based on a classification of the parameters available in the system (**Figure 5.2.3**). This can help to ensure that the users can identify and focus on the subset of versions containing changes for a particular parameter instead of all the versions. When a command is run with at least one of its parameters changed, a new version is created and grouped in a branch with other versions based on the same parameter (see **Section 5.2.2**). Making associations with the parameters at branch and version levels can prove beneficial in the long run when the project evolves, and the branching structure also becomes complex. Associating versions with specific parameters that got modified during each command run can assist in finding versions of interest in the pipeline and avoid the overhead of revisiting and checking the contents of parameters of the command enclosed within each version separately. However, classifying based on the parameters will require careful consideration and prioritization of the parameters.

Another way to manage a large number of versions in a tree is to use semantic zoom [35] that allows the user to see the amount of detail in a view based on the zoom level. This will allow only a certain number of nodes in a tree to be visible at any given time. For example, at a lower zoom level (highly zoomed-out view of the tree), a tree with a single branch could show a certain number of nodes starting from the root node and a certain number of nodes from the leaf node, while the remaining nodes are hidden. In the case of a tree with multiple branches, branches that have more than a certain number of child nodes could be collapsed – the child nodes are hidden – and only the parent nodes are visible in the tree. Therefore, depending on the zoom level, the version tree will appear dense or sparse based on the number of nodes being visible.

6.8 CONTROL OVER GRANULARITY OF SAVING VERSIONS

Giving control to the users over the granularity of saving the versions based on their needs should prove beneficial since their workflows might differ for different kinds of work. Some users might prefer to save their versions periodically by using the auto-saving feature, while some might prefer to save a new version whenever there are some changes in their work, similar to what we had as

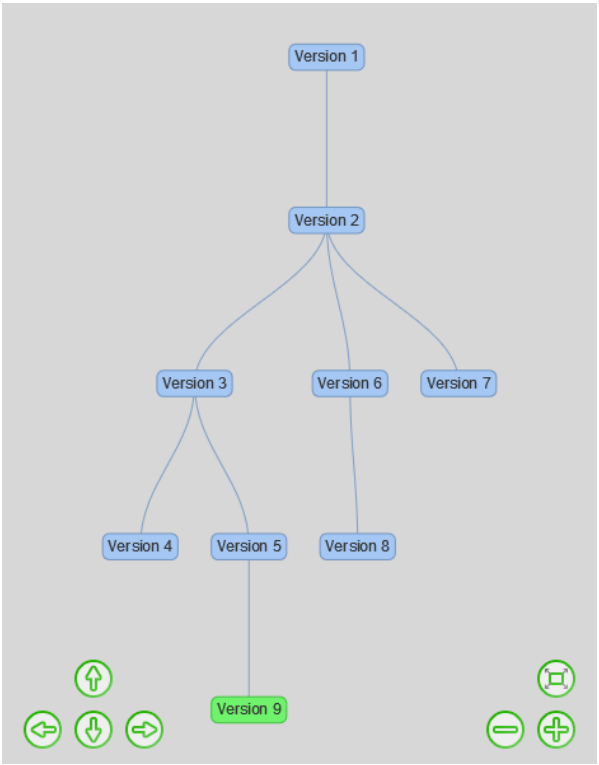
the 3-changes method (Chapter Five). This control over granularity could be a powerful addition to improving individual workflows of the users for interactive systems.

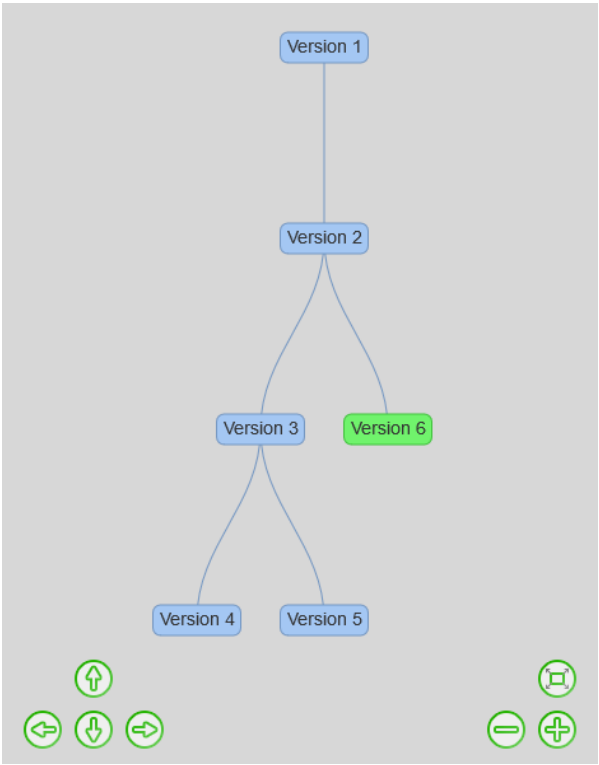
Consider an example of a user working on a design project in a 2D digital design application (such as Photoshop, Affinity Photo, Gimp, or Krita [63,124–126]). The user works on the design in two phases: base design and painting (**Figure 6.8.1**). Consider a scenario in which a user, during the base design phase, would perform a limited number of actions to obtain a base drawing whereas, in the painting phase, the number of actions would increase depending on the level of design complexity. If the system (drawing application) is configured to save the versions based on every change to the base layer, then the resulting tree would create a long single branch. And, if the user also switches between versions throughout the whole drawing process, the tree would have multiple branches, resulting in a dense tree. However, this does not mean that every version stored in a tree is useful for the user. The versions created should contain changes that are significant and useful for the user. Therefore, if the system is configured to save the versions only whenever a new layer is created, for example, the tree structure would be sparse since the number of versions created would be far less (**Figure 6.8.2**).

Figure 6.8.1 Example of 2D digital design project [17] with two phases: base design phase (left) and painting phase (right).

We see that during the design phase of a drawing if a new version is stored for every change, the number of versions created is more than if a new version is stored every time a new layer is created (see **Figure 6.8.2 (a)**). Similarly, during the painting phase, the number of versions stored for every change is way higher than versions stored only on every new layer created (see **Figure 6.8.2 (b)**). Storing a new version on every change made can lead to a tree with a dense structure especially during a complex phase such as the painting phase that requires a greater number of actions by users rather than during the design phase. This makes it difficult to navigate the version tree by users due to how large and dense it can get but provides far more flexibility to go to a certain point in history. Therefore, it is important to have control over the granularity of saving versions to get the desired number of versions in history that in turn creates sparse or dense version trees to navigate at a later time.

(a)





(b)

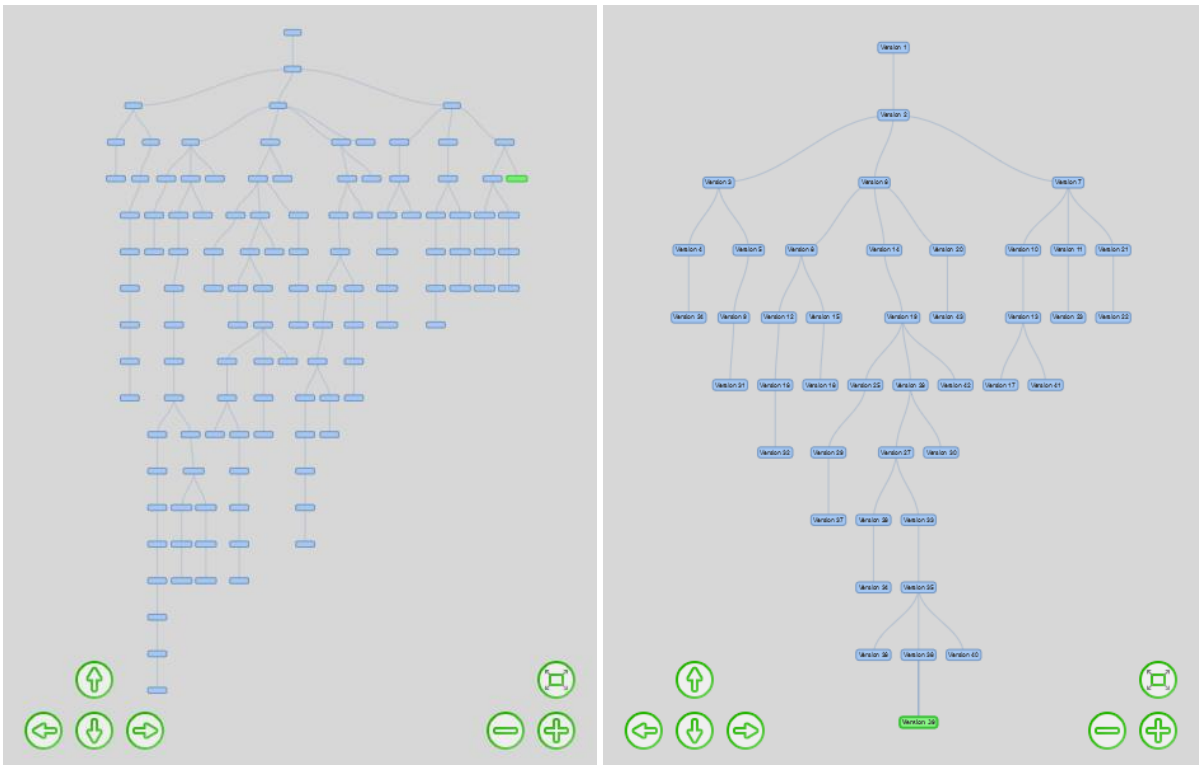


Figure 6.8.2 Trees generated during (a) base design phase: by every change (left) and by layer creation step only (right); (b) painting phase: by every change (left) and by layer creation step only (right).

6.9 SUMMARY

In this chapter, we have discussed various findings and observations from both of our studies (Chapter Five). We also present explanations for the main findings and consider how our results can be used to improve versioning for interactive systems. We discussed eight key elements - versioning patterns, templating, branching, need for merging versions, using previous versions as a reference, versions as undo alternative, managing a large number of versions, and control over the granularity of saving versions - that can be used to improve versioning tools for the interactive systems.

CHAPTER 7

CONCLUSION

Versioning in an interactive environment can help users deal with past states of a document and avoid using methods such as saving and naming multiple copies that make versioning a cumbersome process. However, we lack the understanding to develop effective versioning tools for interactive systems including design issues to deal with the past states of interactive work in a digital interactive workspace and the ability of such tools to save user's interaction history. We surveyed people to understand what constitutes a 'version' in a digital interactive system and why and how often they go back to previous versions. We performed two user studies with our custom versioning tool to see how users create and manipulate versions in an interactive environment. We discussed our findings and observations from both studies and presented explanations for the main findings. The results of the studies suggest that versioning can be a valuable component that can improve the usability of interactive systems and how our tool can help improve the workflow of the users in an interactive environment. The new understanding we gained about versioning in interactive environments by developing and evaluating a custom version tool can help us design more effective versioning tools for interactive systems.

7.1 CONTRIBUTIONS

7.1.1 Primary Contributions

There are three main contributions of this thesis.

7.1.1.1 Comprehension of 'version' creation in the context of interactive systems

Our survey informs us about the various reasons for creating a new version (**Section 3.2**). We also identify possible reasons to revisit or go back to the previous versions of the work (**Section 3.3**).

Our findings help us gain a new understanding of versioning in the context of digital interactive systems.

7.1.1.2 The interactive versioning tool for storing interaction history

The interactive versioning tool (**Section 4.3**) was developed for versioning in interactive systems that can be used for storing both the document/model state and UI state of an application. The tool also provides an interactive version tree that can be used to visualize and interact with versions stored in the tool. The tool can be extremely helpful in dealing with the past states without even leaving the working environment.

7.1.1.3 Identification of requirements for versioning

The evaluation of our custom versioning tool with two studies (game level editor study and web analysis tool study) helped us identify key elements such as versioning patterns, templating, branching, need for merging versions, using previous versions as a reference, versions as undo alternative, managing a large number of versions, and control over the granularity of saving versions that can be used to construct the powerful interactive versioning systems (see Chapter six).

7.1.2 Secondary Contributions

Our two secondary contributions in this thesis are the interactive systems we developed to evaluate our versioning tool - game level editor and web analysis tool (**Sections 4.4 and 4.5**). The game level editor integrates our versioning tool to save the document state of the editor, whereas the web analysis tool allows users to save the state of UI elements in each version. Both of these systems were developed with a focus to implement versioning in them and can serve as a good reference material for developers to design their interactive systems that support versioning.

7.2 FUTURE WORK

The research done in this thesis has laid a foundation for developing better interactive versioning tools for dealing with past states of interactive systems. There are two main concerns that we would like to address in our future work. The first concern is that our versioning tool lacks certain features that prevent users from effectively using the system such as making the interactions richer while working with the version tree or storing a state history of an object in each version.

The second concern we have is that the users' lack of expertise and familiarity with our custom game level editor could have influenced our observations. We want to test our tool in real-world applications that users are familiar with, which allows us to see how our tool could improve their existing workflows. We would be able to see how versioning with different granularities can change the created version trees and how users could utilize such versions trees in different scenarios.

To address our first concern, we propose the following features in our tool to enhance and extend the functionality of interactive versioning. We will explain the concepts using an example of a version tree from our game level editor.

7.2.1 Collapsing and archiving nodes

The web analysis tool case study showed that with the increase in the number of versions in the version management system, it becomes difficult to visually understand the version tree because the tree becomes extremely long which makes it tough to navigate. We believe that collapsing the nodes in a branch that are considered less important by the users can help hide the long chains of nodes in the tree. Additionally, archiving certain nodes or branches that are not currently in focus or required by the user can also decrease the number of versions on screen at once and help improve the understandability of the tree structure.

For instance, in a tree structure as depicted in **Figure 7.2.1 left**, the left sub-branch that stems out of node 'Base Test' at a depth of two has child nodes. This left sub-branch could be collapsed, by selecting the 'Collapse' option from the context menu, to hide the child nodes to declutter the tree

view when the child (leaf nodes) is not being considered in the bigger picture of the project structure. The node that is collapsed can be visually marked with a surrounding dark blue rectangle and a red plus '+' symbol attached to it (**Figure 7.2.1 right**). This ability or feature to collapse and expand the sub-branching nodes can help in improving a user's understanding of the project's version tree and let them focus on the nodes that they are interested in or find more useful. This feature can also be expanded to accommodate another useful feature of semantic zooming in the tree structure view that is explained in **Section 6.7**.

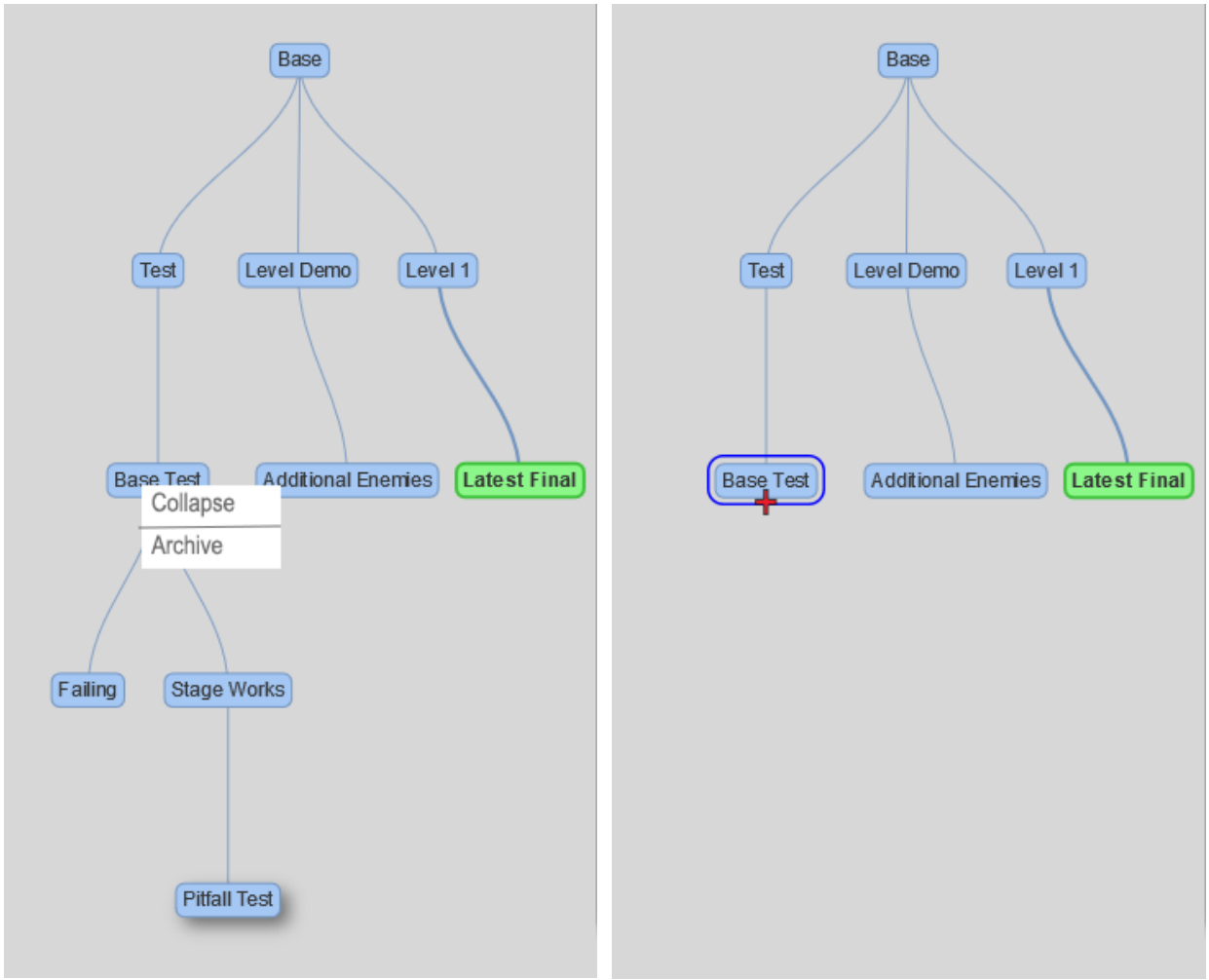


Figure 7.2.1 A mockup of the tree structure before collapsing with context menu (left) and after collapsing (right) branches.

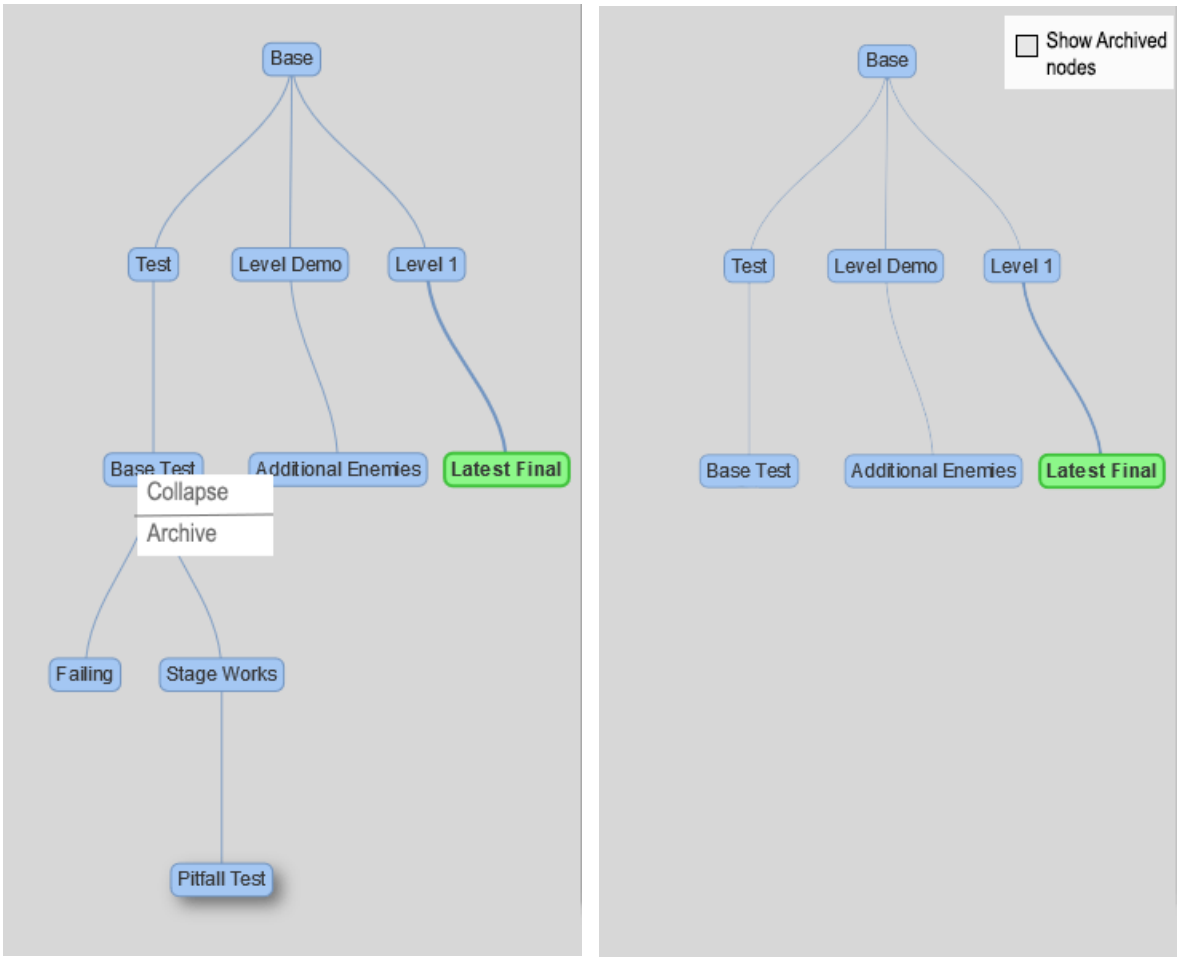


Figure 7.2.2 A mockup of the tree structure before archiving with context menu(left) and after archiving (right) branch.

Similarly, the same sub-branch from node ‘Base Test’ can be archived by selecting the ‘Archive’ option from the context menu (**Figure 7.2.2 left**). This results in hiding all of the child nodes of that sub-branch. These archived nodes are not permanently deleted - just hidden from the tree view. All of the archived nodes can be redisplayed by toggling the ‘Show Archived nodes’ option available in the top-right corner (**Figure 7.2.2 right**).

Both collapsing and archiving nodes can be done manually, automatically, or both. The tool can automatically collapse all nodes of other branches that the user isn’t currently working on.

Similarly, nodes can be archived automatically by the tool based on some date/time criteria such as when the nodes were created and/or when they were last visited.

Between collapsing and archiving nodes, there are two major distinctions. First, unlike collapsing, there is no apparent marker after archiving. Second, when the sub-branches are collapsed, they can be individually expanded by clicking on the '+' symbol, whereas when they are archived in the proposed implementation of archiving, expanding nodes might not be possible. They could only be re-displayed simultaneously.

7.2.2 Pruning branches

In both the studies presented in this thesis, we noticed that participants created some nodes that they did not use later. One participant of the game level editor study remarked that they "*created a few nodes accidentally*" while another mentioned that they created a few versions that "*were just [temporary] testing nodes and thought to delete [them] later*". So, they created those versions either unintentionally or their workflow changed when working through the design tasks. In the web analysis tool case study, users ran a large set of commands on the wrong dataset or accidentally selected incorrect parameters that led to the creation of some unwanted nodes in their tree representation. Such a scenario is a common mistake where users do not want to alter the tree structure but end up having nodes that they do not remember creating in the first place. Another situation is where users may have created some nodes at an earlier stage of a project and later realize that their project has evolved or changed significantly, and they no longer require those changes stored as nodes that have now become redundant. Therefore, with the ability to clean up a tree by pruning [127,128] (deleting) branches, users can keep the node clutter to a minimum and understand the node relationships better. **Figure 7.2.3** depicts a pruned branch where all the child nodes have been deleted, and the parent node 'Base Test' is marked red to show pruning was done on this node. Pruning a branch would result in the permanent deletion of nodes from the version tree and cannot be restored later. To temporarily hide a node or a branch, users can use the 'Collapse' or 'Archive' option as mentioned in **Section 7.2.1**.

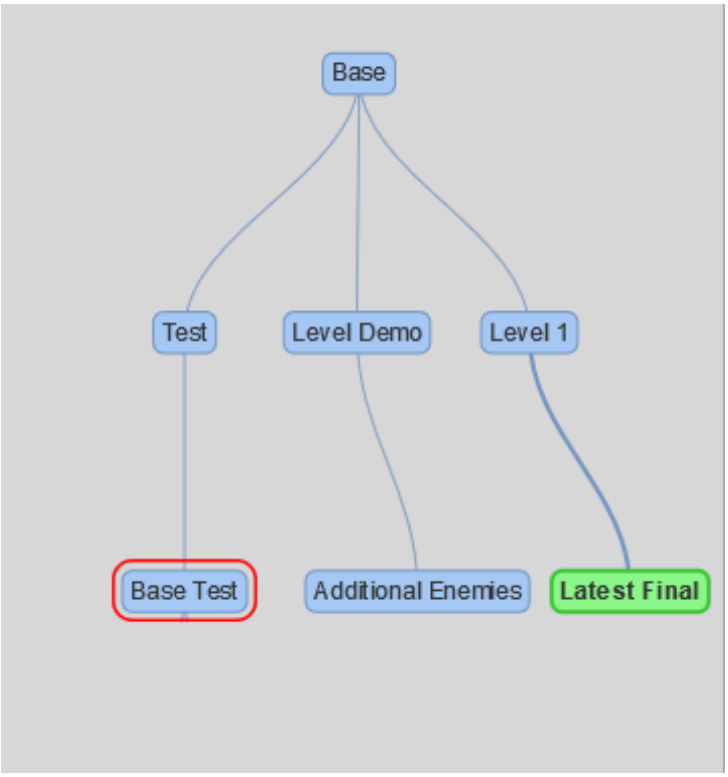


Figure 7.2.3 A mockup of the pruned node ‘Base Test’ marked with a red rectangle.

7.2.3 Color coding or tagging the nodes

When the number of nodes in a version tree increases throughout the evolution of a project, it becomes a challenge to identify specific nodes that correspond to a particular set of changes in the project. Although users can rename their nodes to give more descriptive names that indicate the types of changes stored in different versions, doing a visual search in a tree to find a node can still be cumbersome and could take a substantial amount of time and effort. Therefore, to resolve such an issue, color coding the nodes according to a category of work or a particular behavior can help speed up the navigate-and-search process for a user which in turn can save their time.

For example, the nodes marked with brown color (**Figure 7.2.4 left**) depict test levels while yellow-colored nodes are levels that contain some specific set of obstacles that are not found in any other versions of the same level.

Similarly, the nodes can be tagged with specific keywords such as ‘Enemy’ which can be easily searched for later. Only versions containing the correct tag are highlighted with blue color and the rest of the tree nodes are greyed out to emphasize the searched nodes only (**Figure 7.2.4 right**).

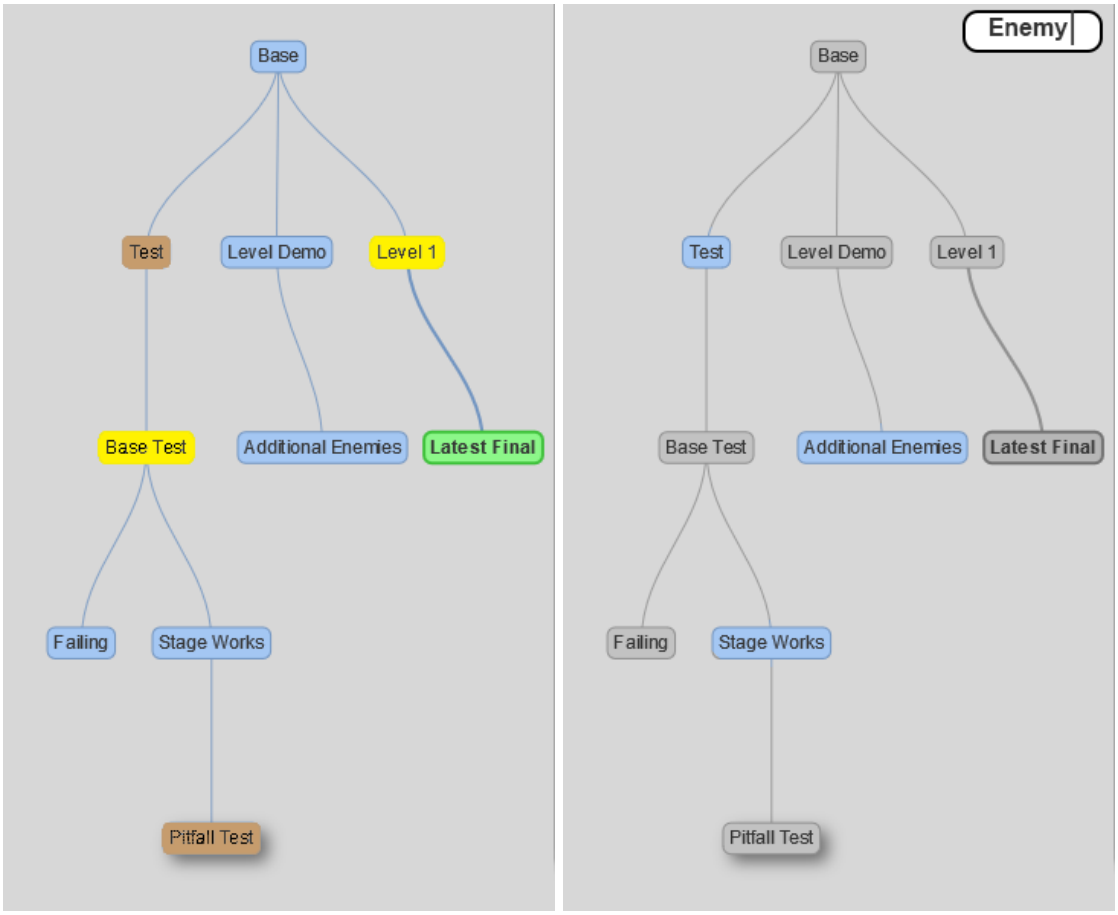


Figure 7.2.4 A mockup of the application screen with color-coded nodes (left) and nodes with tags along with search capability (right) that highlights only nodes with ‘Enemy’ tag.

Color coding could also be used to reflect the version tree's timeline or the number of changes between versions. The oldest node in the tree, for example, could be represented by a lighter shade of color, while the most recently generated node could be represented by a darker shade of color. Similarly, the color of the edge linking two nodes could indicate how many changes there are between the two versions. The darker hue of the edge connecting the two nodes will result from a large number of changes.

7.2.4 Annotating the nodes

Another interesting idea to augment the nodes with more useful information is to attach annotations to the nodes. A small but informative text message accompanying any node can help in providing additional information regarding the data stored in a version. This could allow users to leave notes and messages in the version tree for future reference or other users during collaborative projects.

Figure 7.2.5 shows several annotated nodes represented by the annotation symbol ‘i’ enclosed in a white circle. The node named ‘Stage Works’ shows the expanded text in the annotation box when a user hovers over the annotation symbol.

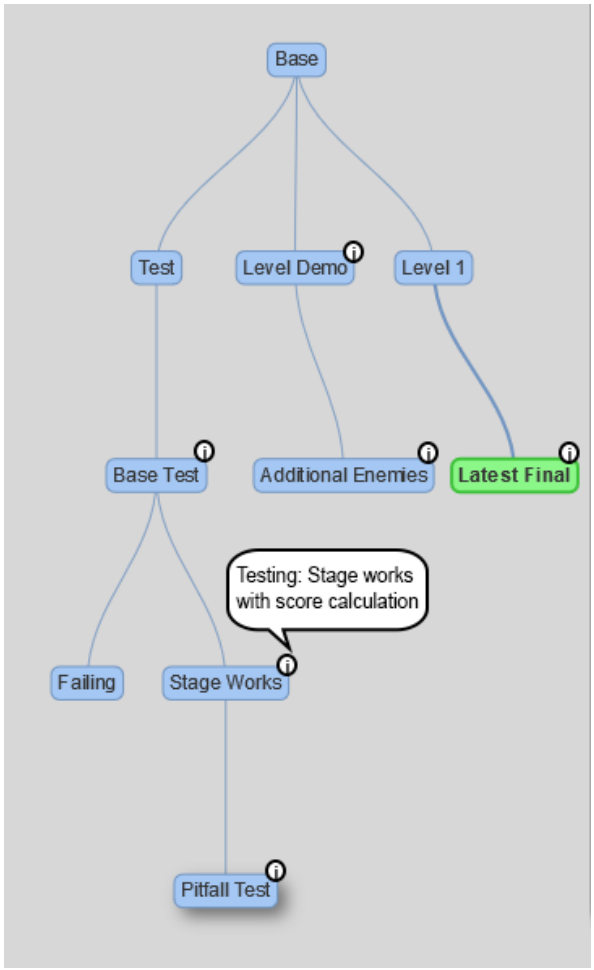


Figure 7.2.5 A mockup of the tree structure with annotations added to nodes.

The annotations can also be added to the preview screen similar to ordinal numbers displayed in a summary frame of Sketcholution [51] (see **Figure 2.3.3**) or storyboard with action depictions (graphical metaphors as icons, see **Figure 7.2.6**) [44] that allows users to undo their actions, or left as comments attached to the states (see **Figure 7.2.7**) in d.note – a revision tool for user interfaces expressed as control flow diagrams [21].

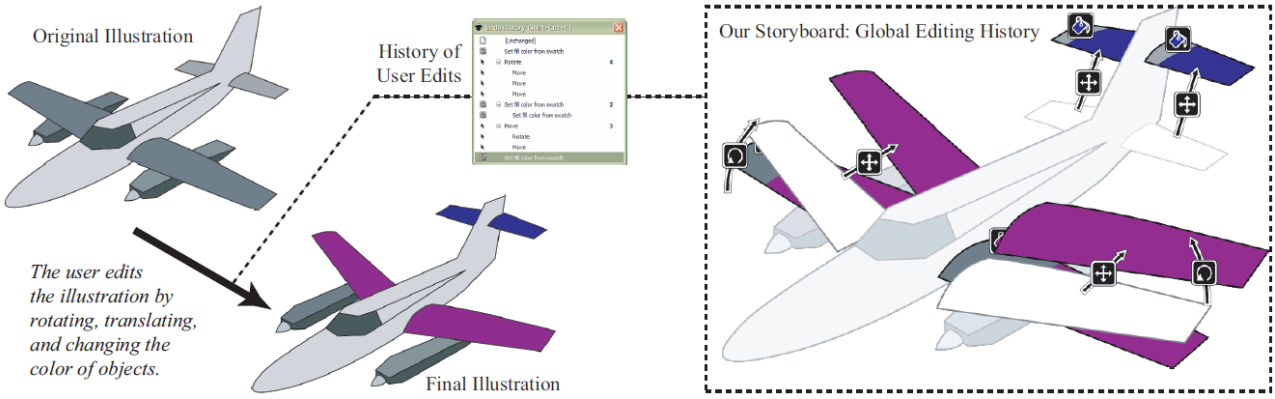


Figure 7.2.6 Editing history of an illustration where the user edits are depicted as arrows and icons that can be clicked to undo those edits [44].

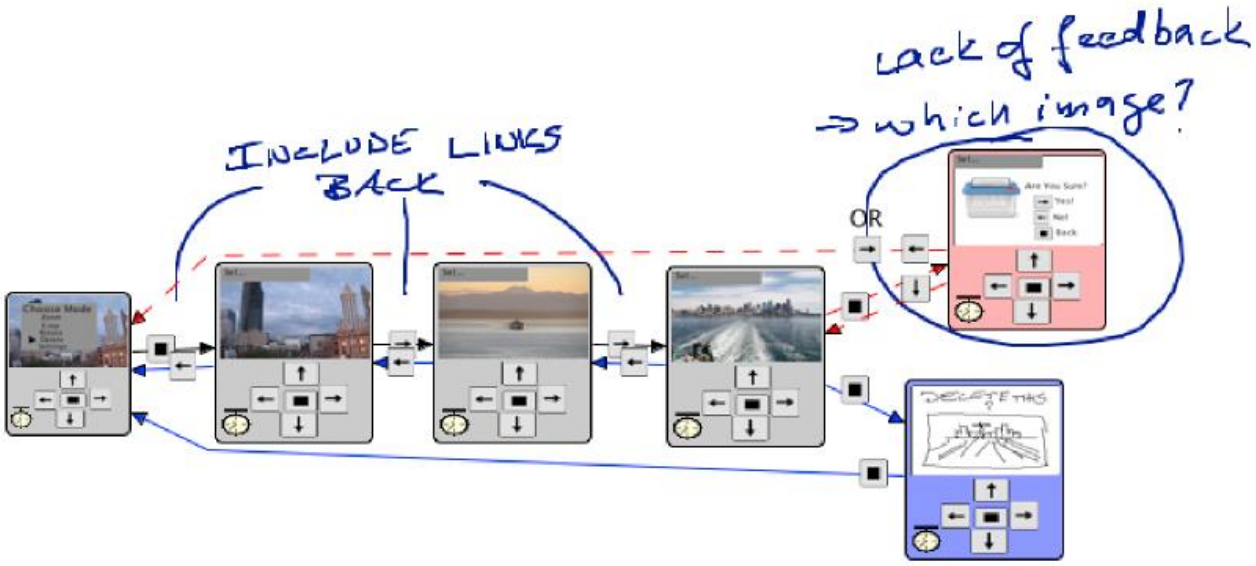


Figure 7.2.7 d.note enables interaction designers to leave comments attached to any state [21].

7.2.5 History of Specific Object States

The interaction history can be further enriched by saving the history of the states for individual objects in the environment. The user can select any object in the current version tree and see the related object states history for the selected object.

In the case of our game level editor, an individual sprite's state history can be saved as a single branch tree (**Figure 7.2.8 right**) that can be visualized as a strobe style path [1], a frame-by-frame illustration of an object's transition, as shown in **Figure 7.2.8 (left)**. The tool would start saving the state of the object when it is first created, and then all the subsequent changes to the object would result in a new object state. All states of the object would be stored as nodes of the Object States History tree where the intermediate states are depicted with blue color while the initial state is marked red and the final state is marked green. To provide additional information, the nodes would be labeled with descriptive texts for each state, e.g., the initial state labeled 'C (100, 180)' would refer that an object was created at position (100, 180) and the other nodes labeled with a text including 'M' would refer that the object was moved to a certain location in the level.

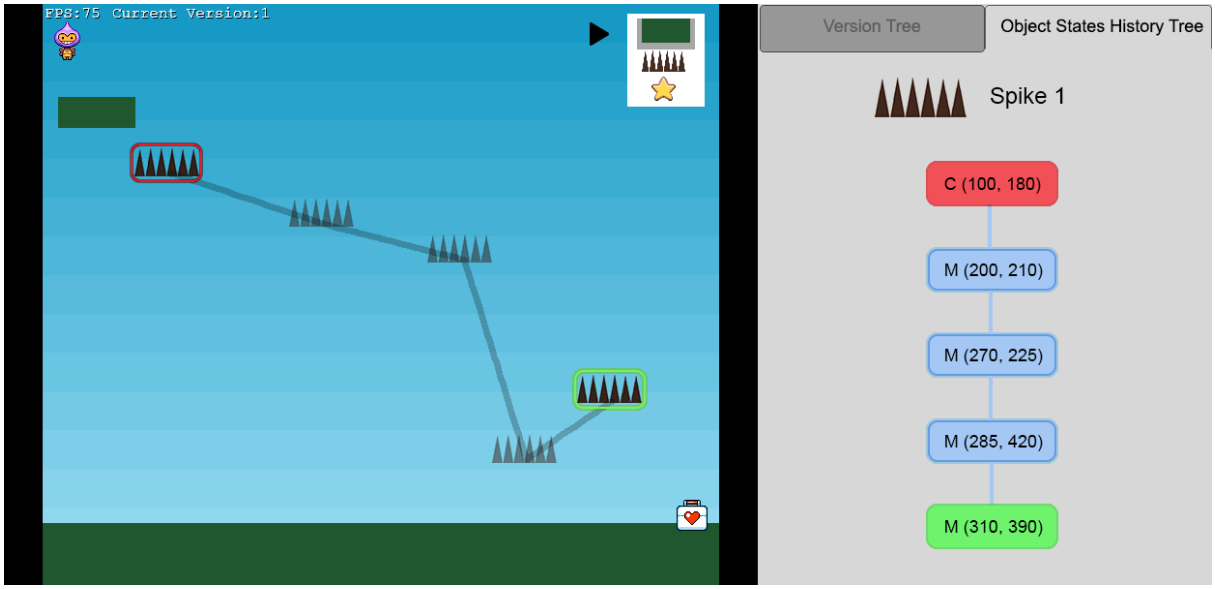


Figure 7.2.8 A mockup of the application screen showing the Object States History for 'Spike 1' sprite object with a strobe-styled path effect (left) and object interaction state history as a single branch tree (right).

We can even add visualization to the nodes to show interaction with them. For example, we can darken the outlines or color of nodes to represent how often the node has been visited by the user. This can help users see what node(s) has been worked with recently or more frequently.

Moreover, we can also use something like phosphor transitions using afterglow effects [1] to represent the interaction history for the UI elements like buttons, dropdowns, sliders, etc. in a project like a web analysis tool (Chapter Six) where the form elements can be highlighted with an afterglow with user-specified granular interaction (for example, interaction frequency) as shown in **Figure 7.2.9**.

Figure 7.2.9 A mockup of the web analysis tool's interface where the changed UI elements are highlighted in green after-glow effects.

7.2.6 Collaboration

In situations where designers are working in a collaborative environment, mining the local interaction histories of each user can help coordinate the team's activities [43]. The synchronous development changes can be reflected to each team member for real-time collaboration [22,23].

The real-time collaboration must be implemented inside the application itself using a real-time communication protocol, e.g., WebRTC [129]. The application would connect the users and would allow our custom version tool to sync the version history stored locally on each user's application. The version tool can then have the individual local interaction history for each user visualized by color-coding their interactions in the Object State History tree. On the other hand, each user interaction can be marked with separate colors to distinguish between the actions performed by respective users in the level editor window. Moreover, we can also annotate their actions in both the views (Level Editor window and Object States History panel).

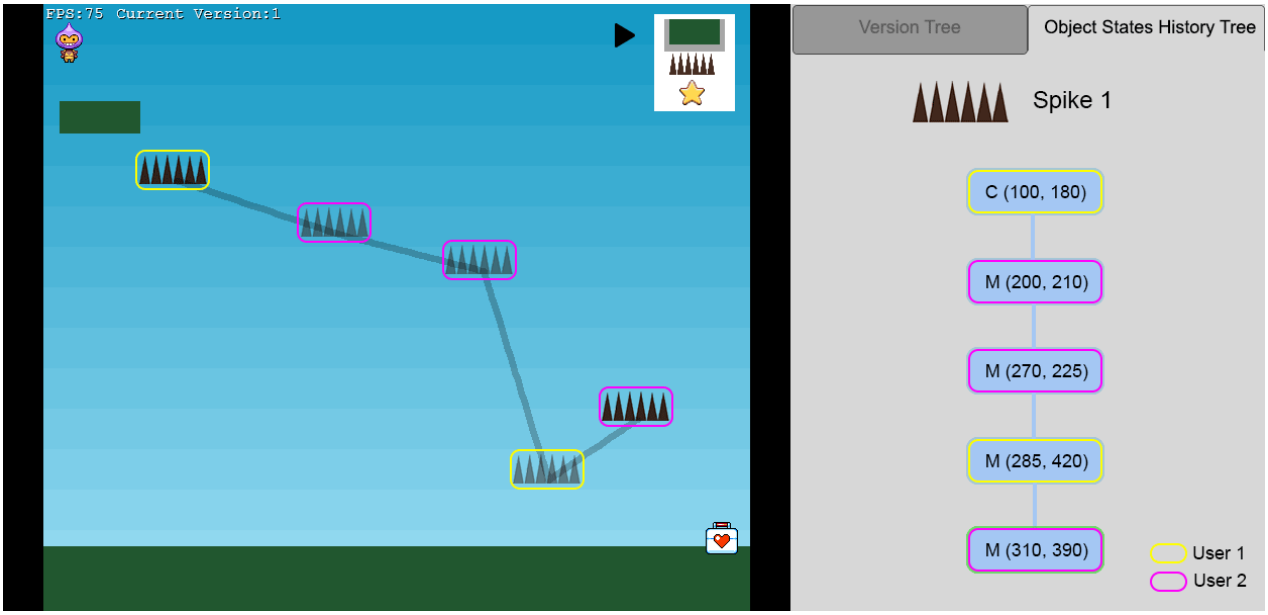


Figure 7.2.10 A mockup of the application screen showing the Object States History for 'Spike 1' sprite object for each user in collaboration with color-coding.

7.3 SUMMARY

In an interactive system, versioning can assist users in dealing with the past states of the system. But traditional versioning tools are often unsuitable for versioning in such systems. We lack the understanding to develop effective versioning tools for interactive systems, including design issues to deal with the past states of interactive work and the ability of such tools to save users' interaction history. We surveyed people to understand more about versioning in interactive systems including what is a 'version' in such systems, circumstances when users create new versions, and why and how often they go back to previous versions. We built a custom versioning tool that we tested in two web-based applications to see how users carry out design and analysis tasks. Our findings demonstrate how versioning may enhance the power and usability of interactive systems, as well as how our tool can improve user workflow in an interactive environment. The work presented in this thesis has given us a new understanding of versioning in interactive systems, which can aid us in developing more effective versioning tools.

REFERENCES

1. Patrick Baudisch, Desney Tan, Maxime Collomb, Dan Robbins, Ken Hinckley, Maneesh Agrawala, Shengdong Zhao, and Gonzalo Ramos. 2006. Phosphor: explaining transitions in the user interface using afterglow effects. In *Proceedings of the 19th annual ACM symposium on User interface software and technology - UIST '06*. https://doi.org/10.1145/1166253.1166280

2. Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2: 77–101. https://doi.org/10.1191/1478088706QP063OA

3. AB Brown and DA Patterson. 2002. Rewind , Repair , Replay : Three R ' s to Dependability. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*.

4. Tony Campbell, Justin Matejka, and George Fitzmaurice. 2010. Chronicle: Capture, Exploration, and Playback of Document Workflow Histories. In *Proceedings of the 23nd Annual ACM Symposium on User Interface Software and Technology*.

5. Aaron G. Cass and Chris S. T. Fernandes. 2007. Using Task Models for Cascading Selective Undo. In *Task Models and Diagrams for Users Interface Design*. https://doi.org/10.1007/978-3-540-70816-2_14

6. Hsiang-Ting Chen, Tovi Grossman, Li-Yi Wei, Ryan M. Schmidt, Björn Hartmann, George Fitzmaurice, and Maneesh Agrawala. 2014. History assisted view authoring for 3D models. https://doi.org/10.1145/2556288.2557009

7. Zach Cutler, Kiran Gadhave, and Alexander Lex. 2020. Trrack: A Library for Provenance-Tracking in Web-Based Visualizations. In *Proceedings - 2020 IEEE Visualization Conference, VIS 2020*. https://doi.org/10.1109/VIS47514.2020.00030

8. J D Denning, V Tibaldo, and F Pellacini. 2015. 3DFlow: Continuous Summarization of Mesh Editing Workflows. *Acm Transactions on Graphics*. https://doi.org/10.1145/2766936

9. Jonathan D. Denning, William B. Kerr, and Fabio Pellacini. 2011. MeshFlow: Interactive Visualization of Mesh Construction Sequences. *ACM SIGGRAPH 2011 papers on - SIGGRAPH '11*. https://doi.org/10.1145/1964921.1964961

10. Jonathan D Denning and Fabio Pellacini. 2013. MeshGit：Diffing and Merging Meshes for Polygonal Modeling. *ACM Trans. Graph.* https://doi.org/10.1145/2461912.2461942

11. J Doboš and a Steed. 2012. 3D Diff: an interactive approach to mesh differencing and conflict resolution. *SIGGRAPH Asia 2012 Technical Briefs*. https://doi.org/10.1145/2407746.2407766

12. Jozef Doboš, Niloy J. Mitra, and Anthony Steed. 2014. 3D Timeline: Reverse engineering of a part-based provenance from consecutive 3D models. *Computer Graphics Forum*. https://doi.org/10.1111/cgf.12311

13. Jozef Doboš and Anthony Steed. 2012. 3D revision control framework. https://doi.org/10.1145/2338714.2338736

14. Steven M. Drucker, Georg Petschnigg, and Maneesh Agrawala. 2006. Comparing and managing multiple versions of slide presentations. https://doi.org/10.1145/1166253.1166263

15. Eric Raymond. Understanding Version-Control Systems (DRAFT). Retrieved June 12, 2019 from http://www.catb.org/~esr/writings/version-control/version-control.html

16. Jennifer Fernquist, Tovi Grossman, and George Fitzmaurice. 2011. Sketch-Sketch Revolution: An Engaging Tutorial System for Guided Sketching and Application Learning. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*.

17. Floortje. 2017. Origami Crane. Retrieved from https://www.instagram.com/p/BXaZlhAAJ7j/?igshid=umtr5euupo6j

18. Dustin Freeman and Ravin Balakrishnan. 2011. Tangible actions. https://doi.org/10.1145/2076354.2076373

19. Erich Gamma. 1995. Design Patterns – Elements of Reusable Object-Oriented Software. *A New Perspective on Object-Oriented Design*. https://doi.org/10.1093/carcin/bgs084

20. François Guimbretiére, Morgan Dixon, and Ken Hinckley. 2007. ExperiScope: an analysis tool for interaction data. In *CHI '07*. https://doi.org/10.1145/1240624.1240826

21. Björn Hartmann, Sean Follmer, Antonio Ricciardi, Timothy Cardenas, and Scott R. Klemmer. 2010. d.note: revising user interfaces through change tracking, annotations, and alternatives. In *Proceedings of the 28th international conference on Human factors in computing systems - CHI '10*. https://doi.org/10.1145/1753326.1753400

22. Lile Hattori. 2010. Enhancing collaboration of multi-developer projects with synchronous changes. https://doi.org/10.1145/1810295.1810397

23. Lile Hattori and Michele Lanza. 2010. Syde : A Tool for Collaborative Software Development. In *ICSE 2010. Proceedings of the 32th International Conference on Software Engineering, 2010.*

24. James J. Hunt and Walter F. Tichy. 1998. Addendum to "Delta algorithms: an empirical analysis." *ACM Transactions on Software Engineering and Methodology*. https://doi.org/10.1145/292182.292200

25. Darris Hupp and Robert C Miller. 2007. Smart bookmarks: automatic retroactive macro recording on the web. *Proceedings of the ACM Symposium on User Interface Software and Technology*. https://doi.org/10.1145/1294211.1294226

26. Karel Jakubec, Marek Polák, Martin Nečaský, and Irena Holubová. 2014. Undo/redo operations in complex environments. In *Procedia Computer Science*. https://doi.org/10.1016/j.procs.2014.05.461

27. Nicholas Kong, Tovi Grossman, Björn Hartmann, Maneesh Agrawala, and George Fitzmaurice. 2012. Delta: A Tool For Representing and Comparing Workflows. *CHI*. https://doi.org/10.1145/2207676.2208549

28. J. Marks, W. Ruml, K. Ryall, J. Seims, S. Shieber, B. Andalman, P. A. Beardsley, W. Freeman, S. Gibson, J. Hodgins, T. Kang, B. Mirtich, and H. Pfister. 1997. Design galleries: a general approach to setting parameters for computer graphics and animation. *In Proc. of SIGGRAPH '97*. https://doi.org/10.1145/258734.258887

29. Robert Martin, John Demme, and Simha Sethumadhavan. 2012. TimeWarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings - International Symposium on Computer Architecture*. https://doi.org/10.1109/ISCA.2012.6237011

30. Paula Montoto, Alberto Pan, Juan Raposo, Fernando Bellas, and Javier López. 2009. Automating navigation sequences in AJAX websites. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. https://doi.org/10.1007/978-3-642-02818-2_12

31. Elizabeth D. Mynatt, Takeo Igarashi, W. Keith Edwards, Anthony LaMarca, Coyote Hill, and Keith Edwards. 1999. Flatland: New Dimensions in Office Whiteboards. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. https://doi.org/http://doi.acm.org/10.1145/302979.303108

32. Mathieu Nancel and Andy Cockburn. 2014. Causality - A Conceptual Model of Interaction History. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems - CHI '14*. https://doi.org/10.1145/2556288.2556990

33. Ulric Neisser and Walter R. Reitman. 2006. Cognition and Thought: An Information Processing Approach. *The American Journal of Psychology*. https://doi.org/10.2307/1421156

34. Peter O'Donovan, Aseem Agarwala, and Aaron Hertzmann. 2015. DesignScape: design with interactive layout suggestions. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*. https://doi.org/10.1145/2702123.2702149

35. Ken Perlin and David Fox. 1993. Pad: an alternative approach to the computer interface. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and*

interactive techniques. https://doi.org/10.1145/166117.166125

36. Shaun Phillips, Jonathan Sillito, and Rob Walker. 2011. Branching and merging: An investigation into current version control practices. In *Proceedings - International Conference on Software Engineering*. https://doi.org/10.1145/1984642.1984645

37. Catherine Plaisant, Brett Milash, Anne Rose, Seth Widoff, and Ben Shneiderman. 2003. LifeLines: visualizing personal histories. https://doi.org/10.1145/238386.238493

38. Atul Prakash and Michael J. Knister. 1994. A framework for undoing actions in collaborative systems. *ACM Transactions on Computer-Human Interaction*. https://doi.org/10.1145/198425.198427

39. Ellen Redlick and 0000-0003-1431-5516. 2020. A Modular Data Analytic Pipeline for Feature Selection in High Dimensional Microbial Data Sets. Retrieved November 30, 2021 from https://harvest.usask.ca/handle/10388/13284

40. Jun Rekimoto. 1999. Time-Machine Computing : A Time-centric Approach for the Information Environment. *CHI Letters*. https://doi.org/10.1145/320719.322582

41. Marc J. Rochkind. 1975. The Source Code Control System. *IEEE Transactions on Software Engineering*. https://doi.org/10.1109/TSE.1975.6312866

42. Alex Safonov. 1999. Web macros by example. In *CHI '99 extended abstracts on Human factors in computing systems - CHI '99*. https://doi.org/10.1145/632716.632761

43. K.A. Schneider. 2006. Mining a software developer's local interaction history. https://doi.org/10.1049/ic:20040486

44. Craig Scull, Steve Johnson, Frederick Aliaga, Sylvain Paris, Sara L. Su, and Fredo Durand. 2009. Interactive Visual Histories for Vector Graphics. Retrieved June 12, 2019 from https://dspace.mit.edu/handle/1721.1/45600

45. Frank M. Shipman and Haowei Hsieh. 2000. Navigable history: A reader's view of writer's time. *New Review of Hypermedia and Multimedia*. https://doi.org/10.1080/13614560008914721

46. Michael Terry and Elizabeth D. Mynatt. 2002. Side views: persistent, on-demand previews for open-ended tasks. In *Proceedings of the 15th annual ACM symposium on User interface software and technology - UIST '02*. https://doi.org/10.1145/571985.571996

47. Michael Terry, Elizabeth D Mynatt, Kumiyo Nakakoji, and Yasuhiro Yamamoto. 2004. Variation in element and action: supporting simultaneous development of alternative solutions. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, 711–718. https://doi.org/http://doi.acm.org.proxy.lib.sfu.ca/10.1145/985692.985782

48. Walter F. Tichy. 1985. Rcs — a system for version control. *Software: Practice and Experience*. https://doi.org/10.1002/spe.4380150703

49. Wesley Willett, Jeffrey Heer, and Maneesh Agrawala. 2007. Scented widgets: Improving navigation cues with embedded visualizations. *IEEE Transactions on Visualization and Computer Graphics*. https://doi.org/10.1109/TVCG.2007.70589

50. Loutfouz Zaman, Wolfgang Stuerzlinger, Christian Neugebauer, Robert Woodsbury, Maher Elkhaldi, Naghmi Shireen, and Michael Terry. 2015. GEM-NI : A System For Creating and Managing Alternatives In Generative Design. *Proceedings of the 2015 CHI Conference on Human Factors in Computing Systems*. https://doi.org/10.1145/2702123.2702398

51. Zhenpeng Zhao, William Benjamin, Niklas Elmqvist, and Karthik Ramani. 2015. Sketcholution: Interaction histories for sketching. *International Journal of Human Computer Studies*. https://doi.org/10.1016/j.ijhcs.2015.04.003

52. Interactive computing - Wikipedia. Retrieved January 3, 2022 from https://en.wikipedia.org/wiki/Interactive_computing

53. File User Preference — Blender Manual. Retrieved November 27, 2021 from https://docs.blender.org/manual/en/2.93/editors/preferences/save_load.html#auto-save

54. Backing Up and Archiving Scenes | 3ds Max 2020 | Autodesk Knowledge Network. Retrieved June 12, 2019 from https://knowledge.autodesk.com/support/3ds-max/learn-explore/caas/CloudHelp/cloudhelp/2020/ENU/3DSMax-Basics/files/GUID-FFCAA5A1-A5C7-4725-AC01-FC9EE8DA8982-htm.html

55. Dropbox. Retrieved November 28, 2021 from https://www.dropbox.com/

56. Free Cloud Storage - OneDrive Sign-In | Microsoft OneDrive. Retrieved November 28, 2021 from https://www.microsoft.com/en-ca/microsoft-365/onedrive/online-cloud-storage

57. Git. Retrieved November 28, 2021 from https://git-scm.com/

58. Apache Subversion. Retrieved November 28, 2021 from https://subversion.apache.org/

59. Mercurial SCM. Retrieved November 28, 2021 from https://www.mercurial-scm.org/

60. Visual Studio Code - Code Editing. Redefined. Retrieved January 5, 2022 from https://code.visualstudio.com/

61. Unity Real-Time Development Platform | 3D, 2D VR & AR Engine. Retrieved January 5, 2022 from https://unity.com/

62. The most powerful real-time 3D creation tool - Unreal Engine. Retrieved January 5, 2022 from https://www.unrealengine.com/en-US/?sessionInvalidated=true

63. Official Adobe Photoshop | Photo & Design Software. Retrieved November 29, 2021 from https://www.adobe.com/ca/products/photoshop.html

64. Software versioning - Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Software_versioning

65. What is AutoSave? - Office Support. Retrieved June 12, 2019 from https://support.office.com/en-us/article/what-is-autosave-6d6bd723-ebfd-4e40-b5f6-ae6e8088f7a5

66. Understanding backup and autosave files in AutoCAD | AutoCAD 2017 | Autodesk Knowledge Network. Retrieved June 12, 2019 from https://knowledge.autodesk.com/support/autocad/learn-explore/caas/sfdcarticles/sfdcarticles/Understanding-AutoCAD-backup-and-autosave-files.html

67. Basic Editing in Visual Studio Code. Retrieved June 12, 2019 from https://code.visualstudio.com/docs/editor/codebasics#_save-auto-save

68. File Menu — Krita Manual version 4.2.0. Retrieved June 12, 2019 from https://docs.krita.org/en/reference_manual/main_menu/file_menu.html

69. Quicksave | ZBrush Docs. Retrieved June 12, 2019 from http://docs.pixologic.com/reference-guide/preferences/quicksave/

70. Save versions of a document | Adobe. Retrieved June 12, 2019 from https://helpx.adobe.com/story/help/save-versions-document.html

71. Help protect your files in case of a crash - Office Support. Retrieved June 12, 2019 from https://support.office.com/en-us/article/help-protect-your-files-in-case-of-a-crash-551c29b1-6a4b-4415-a3ff-a80415b92f99?ui=en-US&rs=en-US&ad=US

72. How To Auto-Recover Files In Photoshop. Retrieved June 12, 2019 from https://www.addictivetips.com/windows-tips/auto-recover-files-in-photoshop/

73. Working with source code - Help | PyCharm. Retrieved June 12, 2019 from https://www.jetbrains.com/help/pycharm/working-with-source-code.html#auto_save

74. Autosaving on a build | Eclipse. Retrieved June 12, 2019 from https://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.cdt.doc.user%2Ftasks%2Fcdt_t_autosave.htm

75. Auto-save current buffer periodically | Vim Tips Wiki | FANDOM powered by Wikia. Retrieved June 12, 2019 from https://vim.fandom.com/wiki/Auto-save_current_buffer_periodically

76. EmacsWiki: Auto Save. Retrieved June 12, 2019 from https://www.emacswiki.org/emacs/AutoSave

77. Checkpoints (Concept) - Giant Bomb. Retrieved June 12, 2019 from https://www.giantbomb.com/checkpoints/3015-292/

78. Saved Game - Checkpoints | Wikipedia. Retrieved June 12, 2019 from https://en.wikipedia.org/wiki/Saved_game#Checkpoints

79. How to save and load the game in Assassin's Creed Odyssey? - Assassin's Creed Odyssey Guide | gamepressure.com. Retrieved June 12, 2019 from https://guides.gamepressure.com/assassins-creed-odyssey/guide.asp?ID=46549

80. Bazaar. Retrieved November 28, 2021 from https://bazaar.canonical.com/en/

81. GNU arch - GNU Project - Free Software Foundation (FSF). Retrieved June 12, 2019 from https://www.gnu.org/software/gnu-arch/

82. BitKeeper. Retrieved June 12, 2019 from https://www.bitkeeper.org/

83. RhodeCode › Version Control Systems Popularity in 2016. Retrieved June 12, 2019 from https://rhodecode.com/insights/version-control-systems-2016

84. Stack Overflow Developer Survey 2018. Retrieved June 12, 2019 from https://insights.stackoverflow.com/survey/2018#work-_-version-control

85. Git - A Short History of Git. Retrieved June 12, 2019 from https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git

86. What is Software as a Service (SaaS)? - Definition from WhatIs.com. Retrieved June 12, 2019 from https://searchcloudcomputing.techtarget.com/definition/Software-as-a-Service

87. The State of the Octoverse | The State of the Octoverse reflects on 2018 so far, teamwork across time zones, and 1.1 billion contributions. Retrieved June 12, 2019 from https://octoverse.github.com/

88. Create or run a macro - Word. Retrieved June 12, 2019 from https://support.office.com/en-ie/article/create-or-run-a-macro-c6b99036-905c-49a6-818a-dfb98b7c3c9c

89. Macros | Vim Tips Wiki | FANDOM powered by Wikia. Retrieved June 12, 2019 from https://vim.fandom.com/wiki/Macros

90. Automate tasks in Google Sheets - Computer - Docs Editors Help. Retrieved June 12, 2019 from https://support.google.com/docs/answer/7665004?co=GENIE.Platform%3DDesktop&hl=en

91. How to manage file versions in Google Docs, Sheets, and Slides - TechRepublic. Retrieved June 12, 2019 from https://www.techrepublic.com/article/version-history-essentials-for-google-docs-sheets-and-slides/

92. Screen Shots MacgHg. Retrieved June 12, 2019 from http://jasonfharris.com/machg/screenshots/

93. Abstract: The Design Tool We Deserve – Prototypr. Retrieved June 12, 2019 from https://blog.prototypr.io/abstract-the-design-tool-we-deserve-6157bb94469e

94. Introducing Time Travel Debugging for Visual Studio Enterprise 2019 | The Visual Studio Blog. Retrieved June 12, 2019 from https://devblogs.microsoft.com/visualstudio/introducing-time-travel-debugging-for-visual-studio-enterprise-2019/

95. Debugging .NET Apps with Time Travel Debugging (TTD) | .NET Blog. Retrieved June 12, 2019 from https://devblogs.microsoft.com/dotnet/debugging-net-apps-with-time-travel-debugging-ttd/

96. How to view and delete your browser history. Retrieved June 12, 2019 from https://www.telegraph.co.uk/technology/0/view-delete-browser-history/

97. Save and Restore Browser Sessions in Chrome, Firefox and Vivaldi. Retrieved June 12, 2019 from https://www.maketecheasier.com/save-restore-browser-sessions/

98. How to Use System Restore in Windows 7, 8, and 10. Retrieved June 12, 2019 from https://www.howtogeek.com/howto/windows-vista/using-windows-vista-system-restore/

99. Time Machine, the Backup Software You Should Be Using. Retrieved June 12, 2019 from https://www.lifewire.com/review-apples-time-machine-software-2260112

100. How To Backup And Restore Linux With Timeshift - It's FOSS. *https://itsfoss.com/*. Retrieved June 12, 2019 from https://itsfoss.com/backup-restore-linux-timeshift/

101. Windows 10 Timeline: How to use Microsoft's new organizational tool | PCWorld. Retrieved June 12, 2019 from https://www.pcworld.com/article/3263905/windows-10-how-to-use-timeline.html

102. Super Meat Boy. Retrieved June 12, 2019 from https://en.wikipedia.org/wiki/Super_Meat_Boy

103. Braid Video Game. Retrieved June 12, 2019 from https://en.wikipedia.org/wiki/Braid_(video_game)

104. Super Meat Boy on Steam. Retrieved January 7, 2022 from https://store.steampowered.com/app/40800/Super_Meat_Boy/

105. Braid on Steam. Retrieved January 7, 2022 from https://store.steampowered.com/app/26800/Braid/

106. Top 10 Best Time Manipulation Games of All Time - Gameranx. Retrieved June 12, 2019 from https://gameranx.com/features/id/28157/article/top-10-best-time-manipulation-games-of-all-time/

107. Prince of Persia: The Forgotten Sands™ on Steam. Retrieved January 7, 2022 from https://store.steampowered.com/app/33320/Prince_of_Persia_The_Forgotten_Sands/

108. Super Time Force Ultra on Steam. Retrieved January 7, 2022 from https://store.steampowered.com/app/250700/Super_Time_Force_Ultra/

109. Reddit - Dive into anything. Retrieved December 3, 2021 from https://www.reddit.com/

110. Timeline - Learn about this chart and tools to create it. Retrieved June 12, 2019 from https://datavizcatalogue.com/methods/timeline.html

111. Storyboard | Wikipedia. Retrieved June 12, 2019 from https://en.wikipedia.org/wiki/Storyboard

112. Undo/redo and history in Adobe Photoshop. Retrieved September 10, 2020 from https://helpx.adobe.com/ca/photoshop/using/undo-history.html#using_the_history_panel

113. Tree (data structure) - Wikipedia. Retrieved September 10, 2020 from https://en.wikipedia.org/wiki/Tree_(data_structure)

114. Mongoose ODM v5.5.14. Retrieved June 12, 2019 from https://mongoosejs.com/

115. vis.js - A dynamic, browser based visualization library. Retrieved June 12, 2019 from https://visjs.org/

116. ECMAScript Language Specification - ECMA-262 Edition 5.1. Retrieved November 22, 2021 from https://262.ecma-international.org/5.1/#sec-11.9.6

117. GitHub - planttheidea/fast-equals: A blazing fast equality comparison, either shallow or deep. Retrieved November 22, 2021 from https://github.com/planttheidea/fast-equals#readme

118. The definitive, practical guide to diff algorithms and patch formats | Ably Blog: Data in Motion. Retrieved November 22, 2021 from https://ably.com/blog/practical-guide-to-diff-algorithms

119. Phaser - A fast, fun and free open source HTML5 game framework. Retrieved June 12, 2019 from https://phaser.io/

120. Express - Node.js web application framework. Retrieved June 12, 2019 from https://expressjs.com/

121. Node.js. Retrieved June 12, 2019 from https://nodejs.org/en/

122. Learning layer basics in Photoshop. Retrieved June 12, 2019 from https://helpx.adobe.com/ca/photoshop/using/layer-basics.html

123. Work with Smart Objects in Photoshop. Retrieved June 12, 2019 from https://helpx.adobe.com/ca/photoshop/using/create-smart-objects.html

124. Affinity Photo – Professional Image Editing Software. Retrieved November 29, 2021 from https://affinity.serif.com/en-us/photo/

125. GIMP - GNU Image Manipulation Program. Retrieved November 29, 2021 from https://www.gimp.org/

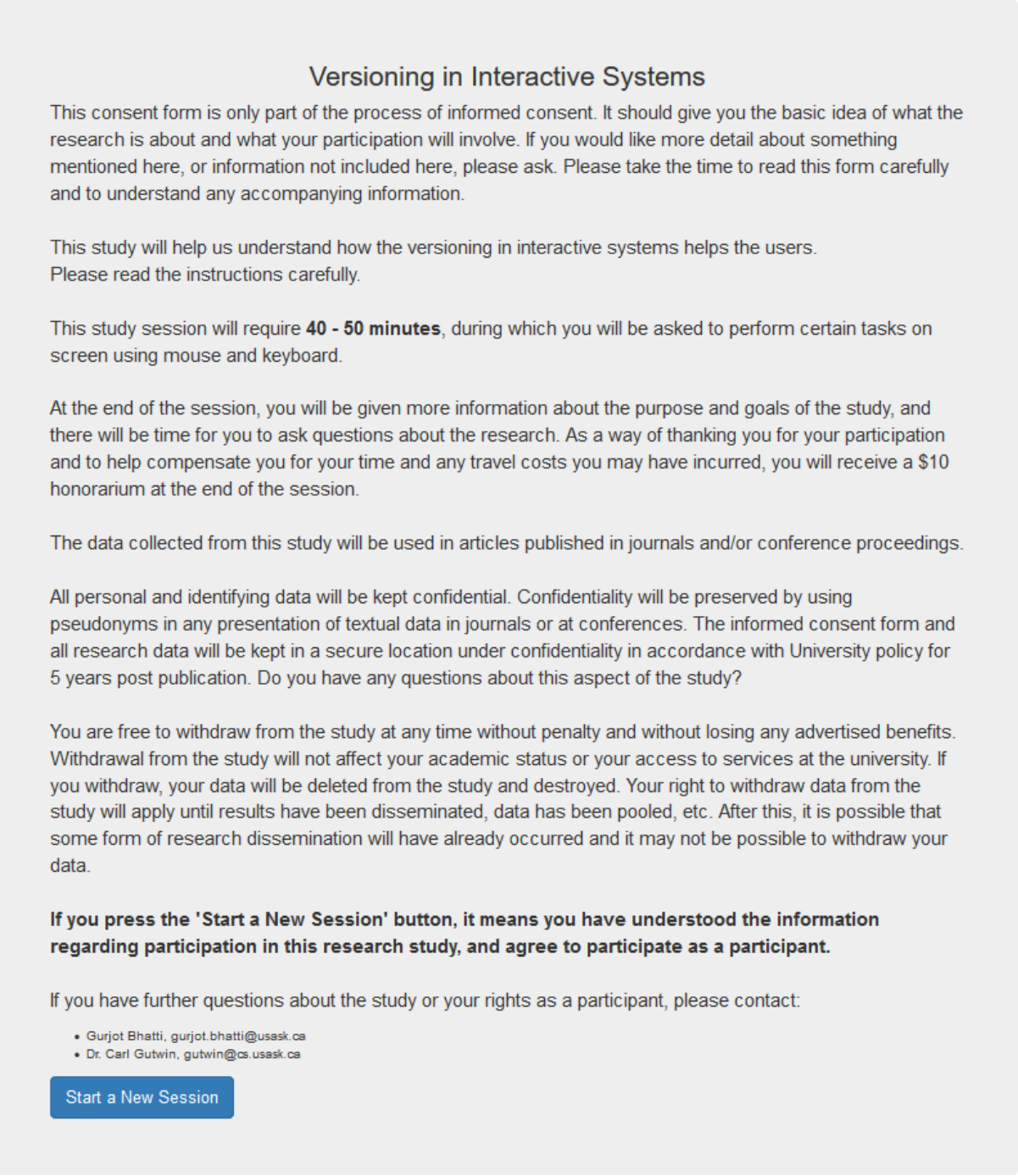
126. Krita | Digital Painting. Creative Freedom. Retrieved November 29, 2021 from https://krita.org/en/

127. Pruning decision trees. Retrieved June 13, 2019 from https://www.ibm.com/support/knowledgecenter/bg/SSEPGG_9.7.0/com.ibm.im.visual.doc/t_pruning.html

128. Git Prune | Atlassian Git Tutorial. Retrieved June 13, 2019 from https://www.atlassian.com/git/tutorials/git-prune

129. WebRTC. Retrieved December 2, 2021 from https://webrtc.org/

APPENDIX



Versioning in Interactive Systems

This consent form is only part of the process of informed consent. It should give you the basic idea of what the research is about and what your participation will involve. If you would like more detail about something mentioned here, or information not included here, please ask. Please take the time to read this form carefully and to understand any accompanying information.

This study will help us understand how the versioning in interactive systems helps the users.
Please read the instructions carefully.

This study session will require **40 - 50 minutes**, during which you will be asked to perform certain tasks on screen using mouse and keyboard.

At the end of the session, you will be given more information about the purpose and goals of the study, and there will be time for you to ask questions about the research. As a way of thanking you for your participation and to help compensate you for your time and any travel costs you may have incurred, you will receive a $10 honorarium at the end of the session.

The data collected from this study will be used in articles published in journals and/or conference proceedings.

All personal and identifying data will be kept confidential. Confidentiality will be preserved by using pseudonyms in any presentation of textual data in journals or at conferences. The informed consent form and all research data will be kept in a secure location under confidentiality in accordance with University policy for 5 years post publication. Do you have any questions about this aspect of the study?

You are free to withdraw from the study at any time without penalty and without losing any advertised benefits. Withdrawal from the study will not affect your academic status or your access to services at the university. If you withdraw, your data will be deleted from the study and destroyed. Your right to withdraw data from the study will apply until results have been disseminated, data has been pooled, etc. After this, it is possible that some form of research dissemination will have already occurred and it may not be possible to withdraw your data.

If you press the 'Start a New Session' button, it means you have understood the information regarding participation in this research study, and agree to participate as a participant.

If you have further questions about the study or your rights as a participant, please contact:

- Gurjot Bhatti, gurjot.bhatti@usask.ca
- Dr. Carl Gutwin, gutwin@cs.usask.ca

Start a New Session

Figure A.1 Consent form

Study Questionnaire

*** 1. Gender**

- ◯ Male
- ◯ Female
- ◯ Other
- ◯ Prefer not to say

*** 2. Age**

*** 3. On average, how many hours do you spend on desktop/laptop computers per day?**

*** 4. On average, how many hours do you play video games (on computer/console/mobile/tablet) per day?**

*** 5. Have you ever designed a game before? (Even a small level or a board game)**

- ◯ Yes
- ◯ No

6. What kind of game did you design?

*** 7. Have you ever used a game engine to design or create a video game?**

- ◯ Yes
- ◯ No

8. Name a game engine that you have used to design or create a video game?

Figure A.2 Study Questionnaire

Versioning and Naming Conventions

*** 1. Gender**

- ◯ Male
- ◯ Female
- ◯ Other
- ◯ Prefer not to say

*** 2. Age**

*** 3. On average, how many hours do you spend on desktop/laptop computers per day?**

*** 4. Specify top 3 software programs/packages that you use:**

For instance:

1. MS Word

2. Photoshop

3. Autodesk Maya

1.

2.

3.

Figure A.3 Online Survey Part 1

* **5. Are you familiar with any of the following versioning tools/systems/softwares?**

- [ ] Git
- [ ] SVN
- [ ] Bazaar
- [ ] Perforce
- [ ] Mercurial
- [ ] Autodesk Vault
- [ ] Visual Studio Team Service
- [ ] None of them
- [ ] Other (please specify)

* **6. Are you familiar with any of the following backup tools/systems/softwares?**

- [ ] Apple's Time Machine
- [ ] Windows Timeline
- [ ] Dropbox
- [ ] OneDrive
- [ ] Google Drive
- [ ] None of them
- [ ] Other (please specify)

* **7. What kind of naming conventions do you follow when you create multiple versions of files?**
Example: web_design.psd, web_design2.psd, web_design3.psd

* **8. In what circumstances do you feel the need to create a version of a file that you are working with?**

Figure A.4 Online Survey Part 2

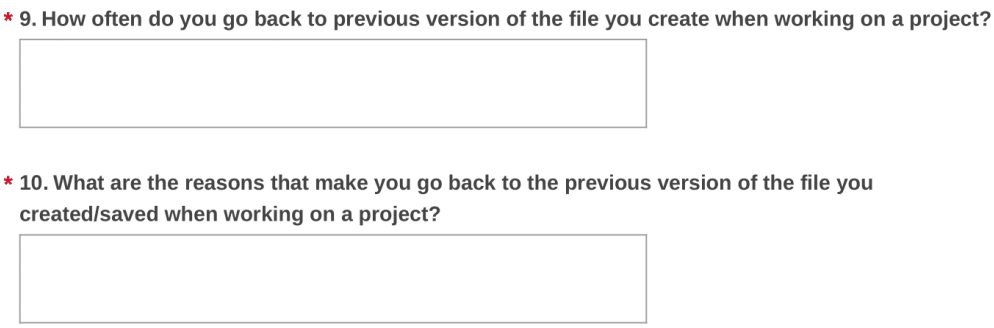


Figure A.5 Online Survey Part 3