

University of Nebraska - Lincoln

DigitalCommons@University of Nebraska - Lincoln

Computer Science and Engineering: Theses,
Dissertations, and Student Research

Computer Science and Engineering, Department of

Summer 5-2016

Sonifying Git History

Kevin J. North

University of Nebraska-Lincoln, knorth@huskers.unl.edu

Follow this and additional works at: <http://digitalcommons.unl.edu/computerscidiss>



Part of the [Graphics and Human Computer Interfaces Commons](#), and the [Software Engineering Commons](#)

North, Kevin J., "Sonifying Git History" (2016). *Computer Science and Engineering: Theses, Dissertations, and Student Research*. 103.
<http://digitalcommons.unl.edu/computerscidiss/103>

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in Computer Science and Engineering: Theses, Dissertations, and Student Research by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

SONIFYING GIT HISTORY

by

Kevin J. North

A THESIS

Presented to the Faculty of
The Graduate College at the University of Nebraska
In Partial Fulfillment of Requirements
For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professors Myra B. Cohen and Anita Sarma

Lincoln, Nebraska

May, 2016

SONIFYING GIT HISTORY

Kevin J. North, M.S.

University of Nebraska, 2016

Advisors: Myra B. Cohen and Anita Sarma

Version control is a technique that software developers use in industry to manage their source code artifacts. One benefit of using version control is that it produces a history of every change made to a codebase, which developers frequently analyze in order to aid the software development process. However, version control history contains highly multidimensional and temporal data. State of the art techniques can show several of these dimensions, but they cannot show a large number of dimensions simultaneously without becoming difficult to understand. An alternative technique to understand temporal data with high dimensionality is sonification. Sonification maps information to sound.

In this thesis we propose the use of earcons and parameter mapping sonification to show version control history. Using sonification, we can show more dimensions of version history simultaneously than other state of the art techniques. Our first technique, GitSonifier, uses only sonification to portray version history and historical conflict data. A user study shows that developers can easily understand the sonification, but we also find limitations where visualization may be preferred. Our second technique, GitVS, uses a combination of both visualization and sonification to overcome these limitations.

ACKNOWLEDGMENTS

I am grateful to my advisors, Dr. Myra B. Cohen and Dr. Anita Sarma. They believed in me and held me to expectations they knew I could achieve. As a result, their confidence in me led to me accomplishing much more than I thought possible.

I also am thankful to Dr. Brittany Duncan for being on my committee.

In addition, I want to thank several of my classmates. Shane Bolan worked with me on the GitSonifier and wrote some of the code for our GitVS implementation. Dr. Bakhtiar Kasi shared his research data on historical conflicts with me. Srikanth Maturu worked with me on a class paper, a portion of which appears in this thesis' background. Jonathan Saddler, Mikaela Cashman, and Justin Firestone, three of my labmates, helped me out frequently. Nina Pickrel, Jacob Lloyd, and Natasha Jahnke collaborated with me on Music++.

Finally, I have an incredible family and excellent friends. My parents, like my advisors, believed in me and encouraged me to fulfill my potential. My friends, especially Russ Canaday, Marsh Lemen, and Taylor Fiscus, are thoughtful, fun, and generous people to be with. My family and friends' encouragement and company have made my research much more enjoyable and successful.

This work was supported in part by NSF grants CCF-1253786, IIS-1110916 and CCF-1161767.

Contents

Contents	iv
List of Figures	ix
List of Tables	xiii
1 Introduction	1
1.1 Contributions of this Thesis	5
1.2 Overview of Thesis	5
2 Background	7
2.1 Version Control	7
2.2 Git	7
2.2.1 How Git is Implemented	8
2.2.1.1 A Git Repository as a Directed Graph	8
2.2.1.2 Decentralized Version Control	9
2.3 Merge Conflicts	11
2.3.1 How Merge Conflicts Affect Industry Teams	11
2.3.2 Tools for Detecting Conflicts	12
2.3.3 Viewing Historical Conflict Data	12

2.4	Sonification	13
2.4.1	The Advantages of Sonification	13
2.4.2	Earcon Sonification	14
2.4.3	Parameter Mapping Sonification	15
2.4.4	Other Forms of Sonification	16
2.5	Music++	16
2.6	Related Work	18
2.6.1	CocoViz	19
2.6.2	code_swarm	19
2.6.3	Orchestrating Change	20
2.6.4	Other Applications of Sonification to Software Engineering	22
3	GitSonifier: Sonifying Version History	23
3.1	Motivation	23
3.2	Design	25
3.2.1	Developer Earcons	25
3.2.2	Representing Time with Day Separators	25
3.2.3	Conflict Drums	26
3.3	Implementation	26
3.3.1	Architecture	27
3.3.2	Implementation Details of the Sonification	28
3.3.2.1	Developer Earcons	28
3.3.2.2	Conflict Drum Earcons	29
3.3.2.3	Tempo	29
3.4	User Evaluation	29
3.4.1	Participant Characteristics	30

3.4.2	Study Design	32
3.4.3	Threats to Validity	34
3.4.3.1	Number of Participants	34
3.4.3.2	Participant Characteristics	35
3.4.3.3	Task Design	35
3.4.3.4	Testing a Single System	35
3.5	Results	36
3.5.1	RQ1: How well do participants interpret the sounds representing a Git history?	36
3.5.2	RQ2: How efficient is the use of sonification for understanding a Git history?	38
3.5.3	RQ3: What was the participants' evaluation of sonification?	39
3.6	Discussion	40
3.6.1	Listening to Sonifications	40
3.6.2	Incorrect Responses	41
3.6.3	Participants' Backgrounds with Music and Version Control	41
3.6.4	Sonification Effects	42
3.6.5	Exit Questionnaire	43
3.7	Conclusions	43
4	GitVS: Combining Visualization and Sonification To Display Ver- sion History	45
4.1	Motivation	46
4.2	Use Case	47
4.3	Design	49
4.4	Implementation	52

4.4.1	Architecture	52
4.4.2	Input	54
4.4.3	The Data Collector	55
4.4.3.1	Walking the Git Repository	55
4.4.3.2	Obtaining Conflict Data	56
4.4.3.3	Getting the Directed Graph Representation of the Repository	56
4.4.3.4	Obtaining the Sorted List of Commits to Send to the Data Processor	57
4.4.4	The Data Processor	58
4.4.4.1	Processing Sonification Data	58
4.4.4.2	Processing Visualization Data	59
4.4.5	The Displayer	59
4.4.5.1	Design of the Displayer	59
4.4.5.2	Final Processing Steps	60
4.4.5.3	Representing Commits Visually	61
4.4.5.4	Showing Details about Individual Commits Visually	61
4.4.5.5	Developer and Conflict Earcons	61
4.5	Planned User Evaluation	62
4.5.1	Participant Demographics	63
4.5.2	Using the GitVS Tool	64
4.5.2.1	Training Session	65
4.5.2.2	Experiment Tasks	66
4.5.3	Post Study: Filling out a questionnaire about our GitVS tool	69
4.5.4	Post Study: Completing an oral exit interview	70
4.6	Conclusion	71

5	Conclusions and Future Work	72
5.1	Conclusion	72
5.2	Future Work	73
5.2.1	User Study for GitVS	73
5.2.2	Sonifying Additional Layers of Information	73
5.2.3	Sonification Design Ideas	73
5.2.4	Designing Tools for Additional Use Cases	74
	Bibliography	76
A	Music++	82
A.1	Design	83
A.1.1	Limitations	83
A.1.2	Sonification Design	86
A.2	Implementation	87
A.2.1	Cobertura	87
A.2.2	Parser	88
A.2.3	Music Generator	89
A.3	Related Work	89
A.3.1	Siren Songs and Swan Songs	90
A.3.2	Increasing Fault Detection Effectiveness Using Layered Program Auralization	90
A.4	Future Work	91
A.5	Conclusions	92
B	GitSonifier Experiment Materials	93
C	GitVS Experiment Materials	107

List of Figures

2.1	Example of the directed graph representation of a Git history.	9
2.2	Two copies of the repository with new commits, <i>c10</i> and <i>c10'</i>	10
2.3	The repository after being fast-forwarded.	10
2.4	The repository after being merged with a merge commit.	10
2.5	A still frame from a <code>code_swarm</code> video [1]	21
3.1	GitSonifier's Architecture	27
3.2	Correct Answers by Category (in order of task ID)	36
3.3	Correct Answers by Category (in the order participants encountered each task)	37
3.4	Correct Answers per Participant	37
3.5	Amount of Time to Answer Each Question	39
4.1	Screenshot of GitVS' View of Commits	50
4.2	Screenshot of GitVS' View of Selected Commits' Details	51
4.3	Screenshot of GitVS' Sonification Cursor. The sonification cursor is the cyan bar in the background.	51
4.4	GitVS' Architecture	53
4.5	Example of Achronological Commits in GitVS	55
4.6	Example Output of <code>git log --graph</code>	57

4.7	Screenshot of the View of Commits in GitVS Without Sound	64
4.8	Screenshot of the View of Selected Commits' Details in GitVS Without Sound	65
B.1	The first page shown to participants in the GitVS experiment.	93
B.2	Page 1 of training page in the GitSonifier experiment.	94
B.3	Page 2 of training page in the GitSonifier experiment.	95
B.4	Page 3 of training page in the GitSonifier experiment.	96
B.5	Page 4 of training page in the GitSonifier experiment.	97
B.6	The instructions in the GitSonifier experiment.	98
B.7	Page 1 of the questions page in the GitSonifier experiment.	99
B.8	Page 2 of the questions page in the GitSonifier experiment.	100
B.9	The page shown when the GitSonifier main task was completed.	100
B.10	Page 1 of the page that participants could view if they wanted to see which questions they answered correctly in GitVS.	101
B.11	Page 2 of the page that participants could view if they wanted to see which questions they answered correctly in GitVS.	102
B.12	Page 3 of the page that participants could view if they wanted to see which questions they answered correctly in GitVS.	103
B.13	Page 4 of the page that participants could view if they wanted to see which questions they answered correctly in GitVS.	104
B.14	Page 5 of the page that participants could view if they wanted to see which questions they answered correctly in GitVS.	105
B.15	The questionnaire given to participants at the end of the GitSonifier study.	106
C.1	The first page shown to participants in the GitVS experiment.	108
C.2	Page 1 of training page in the GitSonifier experiment.	109

C.3	Page 2 of training page in the GitSonifier experiment.	110
C.4	Page 3 of training page in the GitSonifier experiment.	111
C.5	Page 1 of the task instructions in the GitVS experiment.	112
C.6	Page 2 of the task instructions in the GitVS experiment.	113
C.7	Page 3 of the task instructions in the GitVS experiment.	114
C.8	Page 4 of the task instructions in the GitVS experiment.	115
C.9	Page 5 of the task instructions in the GitVS experiment.	116
C.10	Question 1 in the GitVS experiment.	116
C.11	Question 2 in the GitVS experiment.	116
C.12	Question 3 in the GitVS experiment.	117
C.13	Question 4 in the GitVS experiment.	117
C.14	Question 5 in the GitVS experiment.	118
C.15	Page 1 of the instructions shown when switching from one repository to another while answering the objective questions in the GitVS experiment.	119
C.16	Page 2 of the instructions shown when switching from one repository to another while answering the objective questions in the GitVS experiment.	120
C.17	Page 3 of the instructions shown when switching from one repository to another while answering the objective questions in the GitVS experiment.	121
C.18	The instructions shown before participants saw the subjective questions in the GitVS experiment.	121
C.19	Page 1 of the subjective questions in the GitVS experiment.	122
C.20	Page 2 of the GitVS experiment.	123
C.21	The page shown when the GitVS main task was completed.	123
C.22	Page 1 of the page that participants could view to review their answers to questions during the exit interview in GitVS.	124

C.23	Page 2 of the page that participants could view to review their answers to questions during the exit interview in GitVS.	125
C.24	Page 1 of the page that participants could view if they wanted to see which questions they answered correctly in GitVS.	126
C.25	Page 2 of the page that participants could view if they wanted to see which questions they answered correctly in GitVS.	127
C.26	Page 1 of the questionnaire given to participants at the end of the GitVS study.	128
C.27	Page 2 of the questionnaire given to participants at the end of the GitVS study.	129

List of Tables

3.1	Demographics information for the participants in the GitSonifier study.	31
3.2	The properties of each of the 10 clips used in the GitSonifier user study.	33
3.3	The multiple-choice questions asked during each of the 10 tasks.	34
3.4	Questionnaire Results	40
4.1	Objective questions that participants answer for each repository.	68
4.2	Subjective questions that ask participants to compare the repositories.	68
4.3	Post-study questionnaire questions.	70
A.1	List of the types of variables that Music++ can parse.	84
A.2	List of the arithmetic operators that Music++ can parse.	84
A.3	List of the multiline statements Music++ can parse and the pieces of code related to those statements that Music++ sonifies.	85

Chapter 1

Introduction

Version control is a software development technique that is widely used in collaborative development. It provides a way for developers to keep track of the changes made to a codebase, creating a history of the project over time. This history makes it easy to maintain multiple versions of the same project and to revert to old versions if changes are undesirable. In addition, version control allows multiple developers to work on the same files of a project simultaneously, combining them automatically when committed. Some tools that implement version control include CVS, Subversion, and Git [2]. Exploring version control history can help developers accomplish different tasks. For example, developers often use version control history to help them fix bugs [3, 4, 5], understand why a piece of code evolved the way it did, or keep up with changes to pieces of code on which they are actively working [4].

One problem that developers frequently encounter when using version control are merge conflicts. A conflict occurs when two developers make changes in the same location of the same file, but on different branches. When the branches are merged, version control cannot automatically combine the developers' changes, and one of the developers must combine the changes manually [6]. Merge conflicts greatly slow

down teams because they can take a large amount of time to resolve, and they may also reduce quality by introducing subtle bugs that may go unnoticed for a long time. The longer a conflict exists, the more disruptive it is likely to be. As a result, it is important for teams to quickly identify and resolve conflicts [6]. Teams sometimes find it useful to analyze historical conflict data in order to understand how they can produce fewer or completely avoid conflicts.

Over time, several techniques and tools have emerged to help developers manage and understand version control history. They use visualization techniques to represent the data contained in the version control history, making it possible for developers to comprehend and analyze the information. One of the most common techniques is a visualization that shows version control history as a directed graph where each commit appears adjacent to its predecessors in history. One example of a tool that uses this technique is the GitHub network view [7]. Another technique is called *organic visualization*. Instead of having strict rules about how pieces of data are represented, organic visualization allows elements of the visualization to organize themselves based on more fluid rules. This makes it easier to see relationships that are difficult to portray in traditional visualizations. Code_swarm is an example of a tool that uses this technique. It produces a video in which every commit is represented by a dot and each developer is represented by a string with their name. Each time a developer makes a commit, the files the developer committed move closer to their name. Over time, files and developers arrange themselves based on who works on which files the most frequently. [1]. There are several techniques that help developers discover and resolve merge conflicts. The primary technique used to discover conflicts is called *speculative merging*. It attempts to merge code across all copies of a codebase on different developers' machines, and then lets them know if any of those merges results

in conflicts with teammates. This technique informs developers of conflicts as early as possible, allowing them to resolve the conflicts quickly [6, 8, 9, 10].

However, each of these approaches has limitations. Version control history is ordered by time and contains multidimensional data with many pieces of information attached to each commit [11]. None of the state of the art visualization techniques attempt to show more than a handful of these dimensions at once. For example, the GitHub network graph, a version history visualization tool used in industry, only shows the date commits were made on, who made each commit, and the branching and merging structure of the history. However, it is important for version history techniques to display as many dimensions of data as possible because several tasks involve analyzing large amounts of information with respect to the version control history. For example, in order to evaluate a team's performance after making changes to the collaboration process used with a new version control system, a manager will likely be interested in seeing who made each commit [12], the branching and merging structure of the repository [13], the list of files that the commit changed [14, 15, 16], and when merge conflicts appear in history [6]. All of this data will need to be shown for a period of time that may span weeks or months and contain hundreds or thousands of commits on many branches.

In addition, while speculative merging techniques can show conflicts in real time, the field is missing techniques that show patterns of conflicts in a project's history. This is because, while it is possible to discover historical conflicts in version history, it is difficult to do so for many version control systems, including Git. Some researchers have analyzed Git repositories to find historical conflicts for use in their experiments [17, 18], but they have not shown the resulting conflicts to developers, and the methods these researchers have used to find historical conflicts are nontrivial. Since conflicts slow

teams down, if they had easy access to historical conflict data, this might allow them to avoid future conflicts or to make conflict resolution easier to accomplish.

In order to solve these problems, in this work I turn to sonification, the portrayal of data using sound. Sonification is especially well suited for portraying data that is multidimensional and time based. This is because it is possible to design multiple sound objects that play at the same time, but that listeners can nevertheless differentiate and interpret simultaneously. In addition, listeners hear sound over time, making it natural to map the passage of time in a sonification to the passage of time in the data being portrayed [19].

Accordingly, we have created GitSonifier, a technique for sonifying version history. It uses specific techniques called earcons and parameter mapping sonification in order to produce a song that represents who made each commit, when each commit was made, and when merge conflicts appeared in history. Conveniently, the GitSonifier technique allows us to treat historical merge conflicts as a dimension of data to portray, providing a natural way to incorporate that information into the sonification. We also conducted a user study to evaluate whether developers can understand GitSonifier. The results indicate that sonification of version history is a feasible technique [20].

However, our study shows that GitSonifier has a couple of limitations. First, while developers can understand each individual data element in the sonification, they have a harder time obtaining a holistic understanding of the data. In addition, like visualization, when too many dimensions are portrayed in a sonification, it can become difficult to interpret.

In order to overcome the limitations of GitSonifier, we next developed GitVS, a technique that portrays version history by combining visualization and GitSonifier's sonification. Using the visualization, developers can obtain a holistic view of the data. Furthermore, while visualization and sonification both are limited in the number of

dimensions they can portray on their own, combining them allows us to portray more dimensions of version history simultaneously without making the data difficult to understand.

1.1 Contributions of this Thesis

The contributions of this thesis are:

- A technique, GitSonifier, for displaying version control history and historical conflicts. This technique uses sound to portray who made each commit, when commits were made, and when conflicts were present in a project's history.
- A user study to evaluate GitSonifier. In this study, participants were trained to use GitSonifier, then asked to interpret 10 different sound clips generated by the GitSonifier technique. The study results show that the GitSonifier sonification technique makes it easy to understand individual pieces of data from sonification, but it is somewhat difficult to understand the data holistically.
- GitVS, a hybrid visualization that uses both novel visual elements and sound to enrich the understanding of history. This combination fixes GitSonifier's limitation of making it difficult to understand the overall picture of version history data. In addition, it allows more dimensions of version control history to be displayed at once than the state of the art.

1.2 Overview of Thesis

Chapter 2 of this thesis discusses the existing literature about Git, merge conflicts, and sonification, three topics that are important for understanding this thesis. Chapter

3 introduces GitSonifier, a technique that uses sonification to represent information about who made each commit, when commits were made, and when conflicts existed in version control history. In addition, the chapter discusses an exploratory user study that demonstrated sound can be an effective way of displaying version history. Chapter 4 introduces GitVS, a technique that combines the GitSonifier sonification with a novel visualization. Chapter 5 discuss avenues for future work and offers concluding thoughts on the thesis.

Chapter 2

Background

2.1 Version Control

A version control tool allows developers to maintain different major versions of the same software and record commits, or different changesets to their codebase, so that it is easy to revert to a stable version if necessary. It also allows multiple developers on the same team to concurrently modify the same source code file without interrupting each other.

Version control tools are an extremely common tool in industry. Some of the most well-known version control systems are CVS, Subversion, and Git [2].

2.2 Git

Git is one of the most prominent version control tools today. Unlike many other common version control systems, it is a distributed version control system. This means that each developer on a project hosted by Git has an entire copy of the repository hosted on their own computer. This allows a wider variety of version control workflows

than centralized version control systems, in which the only copy of the entire repository exists on a version control server. [2, 11]

2.2.1 How Git is Implemented

This section explains the data structures Git uses to store commits and branches. It also shows how new branches can be introduced to the history by developers sharing changes with each other, even if the developers do not explicitly create new branches themselves.

2.2.1.1 A Git Repository as a Directed Graph

Internally, Git represents the relationships between commits using a directed graph. Each vertex in the graph is a commit, and an edge goes from one commit to its parent commit.

Each commit is represented as a data structure with the following attributes:

- The commit's hash, calculated by computing the SHA1 hash of the rest of the commit's data [21].
- The commit's parents, or the commits made immediately before the given commit in history. This is stored as a list of hashes. Each hash in the list corresponds to one of the parents.
- A data structure representing which files were changed by the commit and how each file was modified.
- The time and date that the commit was created.
- The name and email of the developer who created the commit.

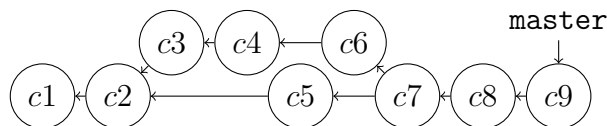


Figure 2.1: Example of the directed graph representation of a Git history.

For example, Figure 2.1 shows an example of what a typical Git repository might look like, represented as a directed graph. The repository has a branch that starts at commit $c2$ and is merged in commit $c7$. The arrows on the graph point from commits to their parents. This may be counterintuitive, but it is how Git represents commits internally: commits contain information about their parents, but not their children [11].

2.2.1.2 Decentralized Version Control

Git is a decentralized version control system. This means that in a team of developers, each individual developer has a full copy of the repository on their machine. This makes a wide variety of version control workflows possible. For example, while it is possible to do so, Git does not require developers to have a central server host the official copy of the repository.

Depending on how the developers use Git, the decentralized nature of Git sometimes causes branches to appear in history, even when developers do not explicitly create branches. Specifically, when a developer pulls changes from another copy of the repository into their own repository, Git will record the pulled changes as a new branch if the developer tells it to do so or if the pulled changes conflict with the changes the developer already has.

For example, suppose two developers are working on the project represented in Figure 2.1. Each developer has their own copy of the repository on their own machine. Then, both developers make changes. The first developer creates commit $c10$, and the

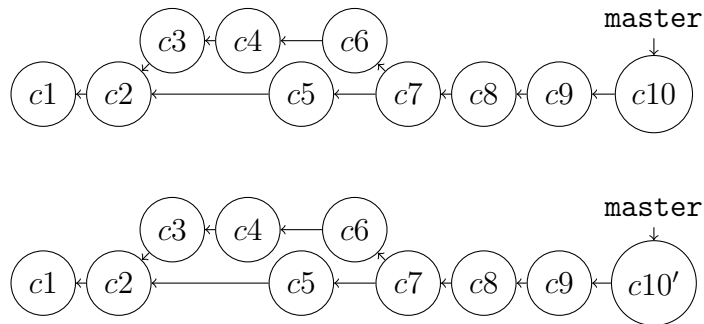


Figure 2.2: Two copies of the repository with new commits, c_{10} and c_{10}' .

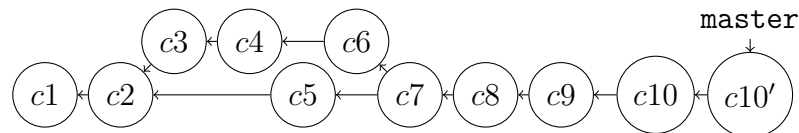


Figure 2.3: The repository after being fast-forwarded.

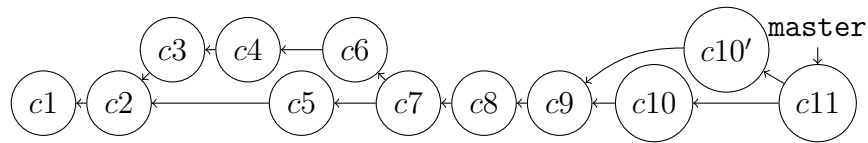


Figure 2.4: The repository after being merged with a merge commit.

second developer creates commit c_{10}' . Figure 2.2 show the developers' repositories after the changes.

If the first developer pulls the changes from the second developer, Git will produce either the history shown in Figure 2.3 or Figure 2.4. If the changes from c_{10} and c_{10}' don't have any conflicts, Git can *flatten* the history and produce the history shown in Figure 2.3 without any new branches. If there are conflicts between c_{10} and c_{10}' , or if the developer prefers it, Git will produce the history in Figure 2.4, which shows a new branch for the commit pulled from the second developer. The branch starts at commit c_9 and is merged in commit c_{11} [11].

2.3 Merge Conflicts

When using version control, developers will frequently edit the same files on different branches. In most typical cases, version control systems can automatically combine these changes when the branches are merged. Sometimes, however, the automatic merging process fails, requiring the developers to manually inspect their changes from each branch and decide which to keep. This situation is called a merge conflict.

Some merge conflicts arise when two or more developers change the same lines of code on the same files on different branches. In these conflicts, Git will not allow developers to merge those branches at all until they have manually edited the conflicting files to resolve the conflicting lines of code.

Other types of conflicts exist. For example, it is possible to have a conflict in which the code on two separate branches compile successfully, but upon being merged, the resulting commit has a build error. Conflicts other than the direct, textual conflicts in the previous paragraph are not automatically detected by version control tools [8].

2.3.1 How Merge Conflicts Affect Industry Teams

Merge conflicts make the development process more difficult. Conflicts that exist for a long period of time before being resolved can be especially costly to fix and may introduce subtle bugs even after being resolved. M. Guimarães and A. Silva express the common view of conflicts in one of their papers, writing, “[C]onflicts emerge due to concurrent work, and become more complex as changes grow without being integrated and as further developments are made. Consequently, the later conflicts are detected, the harder it is to resolve them because more code must be reworked. Besides, a conflict detected late is generally harder to resolve since the changes that caused it are no longer fresh in developers’ minds” [6].

One of the most difficult aspects of working with conflicts is that they can be difficult to identify. Without tools to help them, developers are frequently entirely unaware of conflicts until they attempt a merge with a conflicting branch. By that point, the conflict is likely far more difficult to resolve. In some cases, when the automatic merge appears to succeed but quietly introduces a runtime bug, the conflict might go unnoticed indefinitely [6, 10]!

2.3.2 Tools for Detecting Conflicts

To help developers deal with conflicts, researchers have developed tools that perform *speculative merging*. These tools continuously attempt to merge developers' uncommitted changes with commits that other developers have prepared but not yet pushed. When these merges fail, the tools are able to identify that there is a conflict and let the developers know, allowing them resolve it sooner when it is still relatively easy to fix. These tools will show conflicts as they are created in real time, but do not show historical conflict data [6, 8, 9, 10].

2.3.3 Viewing Historical Conflict Data

Git does not record when conflicts occurred in a project's history, so unfortunately, there is no easy way to explore historical conflict data. Historical conflicts can be discovered later by attempting to merge historical branches and seeing which branches generate conflicts, but to my knowledge, researchers have attempted to recover this information only for the purposes of performing experiments [17, 18]. There are no tools for industry or practitioners for exploring conflict patterns in a project's history.

2.4 Sonification

Sonification is the use of sound to represent data. This thesis primarily uses two sonification techniques. The first, *earcon* sonification, uses individual musical sounds to represent pieces of data. Changes to these musical ideas can provide details about the data they correspond to [19,22]. The second technique, *parameter mapping sonification*, represents continuous or near-continuous data by mapping each dimension of the data to an attribute of a single sound wave, then manipulating the sound wave's attributes over time to display the corresponding data [19].

2.4.1 The Advantages of Sonification

Sonification has several advantages over other techniques of portraying data, such as visualization. The two advantages most important to my work are that sonification can easily portray temporal data and multidimensional data.

Since listeners hear sound over time, it is natural to use sound to portray data with time as one of the axes. As a result, a sound can be produced such that the time that passes as the listener hears the sound corresponds to the passing of time in the data being sonified.

Sonification is also effective at portraying multidimensional data. This is because it is possible to design sounds so that multiple sounds play at once, but listeners can still distinguish them and follow each of the sounds at the same time. For example, if two sounds are played at the same time, one high-pitched and the other low-pitched, a listener can easily follow both sounds as separate auditory objects. Likewise, if one sound is played on the right speaker and the other on the left speaker, again, a listener can easily tell them apart. As a result, if there are multiple datasets that are related to each other, they can be sonified simultaneously. As long as the sonification

is carefully designed, a listener can understand all of the datasets at the same time. For example, if one sound is played out of the left speaker and another sound is played out of the right speaker on a stereo, a typical listener can easily follow both sounds simultaneously. [19]

2.4.2 Earcon Sonification

Earcons are analogous to visual icons. In a traditional visual GUI, an icon represents an event or object of interest. For example, the GUI for Apple’s OS X operating system uses an icon of a trash can to represent the concept of deleting a file. In addition, multiple icons can be combined or related to each other to suggest related ideas. For example, a circle with a diagonal line through it or a large “X” are both icons typically used to represent canceling or prohibiting an action. As a result, an icon with a large “X” over a trash can suggest that a file cannot be deleted.

Likewise, earcons represent ideas with musical sounds. In particular, earcons are typically designed as musical *motifs*, which are short musical phrases with distinct pitches and rhythms. Unlike icons, which typically have an obvious visual association with the ideas they represent, earcons tend to be abstractly mapped to the ideas they represent. As a result, earcons can be used to represent data that doesn’t have an obvious analogy to exploit for an icon. The disadvantage is that a listener will typically have to learn earcons ahead of time, then be able to mentally map an earcon when he or she hears it with the idea it represents.

The motifs used to create earcons have several characteristics. Pitch refers to which notes are played. Rhythm is the timing of the notes. Timbre is the sound of the instrument playing the motif. Related earcons can share musical characteristics so their relationship is easier to recall [19, 22, 23, 24].

A *measure* is a group of notes representing all or part of a motif, and is the basic unit of time in music. [23]

2.4.3 Parameter Mapping Sonification

In Parameter Mapping Sonification (PMSon), each dimension of a data set is mapped to a characteristic of a sound wave, which is then manipulated over time to represent the data. This allows multidimensional, continuous data to be represented with a single sound.

The Sonification Handbook [19] describes PMSon using the example of displaying the temperature of a kettle of water as it heated. For example, consider the temperature of a pot of water as it is boiled. The temperature increases with time until it reaches 100°C and boils. A parameter mapping sonification of the same data might map the temperature of the water with the pitch of a sound wave. When a user listens to such a sound, it would start at a low pitch, then gradually increase in pitch until it reached a pitch corresponding to the boiling point.

In general, a PMSon works well with data where multiple variables change in response to one continuous or near-continuous variable. In the example above, temperature changed in response to time. Other pieces of data could be sonified as well. For example, in order to measure how the water expanded as it got hotter, the temperature and volume of the water could have been sonified simultaneously over time. Once the data to be sonified has been selected, each dimension of the data is mapped to a different aspect of the sound wave used in the sonification. For example, in the boiling water example, two aspects of the sound wave, perhaps pitch and loudness, would be selected to represent the temperature and volume of the water respectively.

Loudness and pitch are two of the aspects of a sound wave that PMSon can map data to. Other aspects include which vowel sound the sound wave most resembles [19,24].

2.4.4 Other Forms of Sonification

There are several other techniques for designing sonifications. Audification is a technique in which data that is already in the form of a physical wave is directly played as sound. Frequently, mild transformations are applied to the data to make it easier or possible for human listeners to hear the sound. For instance, seismologists audify earthquake records, but since the waves have a frequency lower than humans can detect, the records are sped up significantly first [19].

Auditory icons are sounds representing pieces of data. They are selected so that their meanings are intuitive. For example, an early computer program called SonicFinder added sound effects to a file explorer, including the sound of an object scraping on the ground while a user dragged a file. The sound effect is intuitive because dragging an object in the real world produces a scraping sound [19,25].

2.5 Music++

Music++ was an early proof-of-concept sonification for my research that demonstrated sonification is a viable technique in software engineering. It established the idea of using earcons to represent pieces of data in order to produce a song as a common theme in my research.

Music++ was a joint project with Dr. Myra Cohen, Dr. Anita Sarma, and my classmates Nina Pickrel, Jacob Lloyd, and Natasha Jahnke. This section includes material from a class paper that my classmates and I worked on together [26].

The Music++ sonification was implemented in a tool that sonifies the execution paths of simple Java programs. The tool has multiple earcons that correspond to different elements in a Java program, and when each line of code in a sonified Java program's execution path is reached, the earcons for that line of code's elements play simultaneously. Music++ can be used to help teach programming by demonstrating to students how their computers "see" their programs when the programs are run.

Music++ uses an earcon-based sonification. It assigns different earcons to Java data types, arithmetic operations, and control statements. When sonifying the execution path, as each line of code in the program is encountered, all of the earcons corresponding to the elements in the line of code play simultaneously. For example, a line of code that declares a `String` variable would play the `String` earcon, but no other sounds. A line of code that added two integers, then assigned the result to an `int` variable would play the `int` earcon and the addition earcon simultaneously.

When the execution path goes inside the body of a control statement, the sonification continuously plays an earcon to indicate that the execution path is inside of the body. For example, to sonify a line of code that adds two integers, assigns the result to a variable, and occurs inside of an if statement's body, the sonification would play the earcons for addition, the `int` type, and the if statement body simultaneously.

In addition to sonifying each line of code, Music++ also sonifies the conditions in conditional and loop statements. For example, when sonifying an if statement, Music++ will first sonify the statement's condition. Then, if the statement evaluated to true, the sonification will sonify each of the elements in the if statement body.

Each earcon is a measure long because untrained listeners can intuitively tell when a measure begins and ends [23]. All of the earcons were written in the same key so that when they are combined with each other, they still sound musically pleasing. The earcons are written using a wide variety of instruments, pitches, and rhythms

in order to make them easier to tell apart. However, earcons related to the same conditional or loop statement are written using the same instrument and similar pitches. For example, both the while condition and while body earcons are written using a trombone. This makes it easy to identify which earcons are related to the same while loop when multiple earcons play at the same time inside of the while loop's body. Reusing instruments for related earcons is an established earcon sonification technique [19].

In addition, to make them more distinct, the arithmetic operator earcons are all written using drum sounds instead of pitched instruments. Each of the four arithmetic earcons are written with a different drum that plays one loud, easily noticeable note on the same rhythm. As a result, it is easy to identify an arithmetic operator earcon.

Three of the ideas that I initially explored in Music++ became common features of my other sonifications. First, I routinely use earcons that are a measure long. Second, the earcons I use in other sonifications use a wide variety of rhythms, pitches, and instruments for one type of data, and a consistent drum pattern for another type of data. Third, I continue to play multiple earcons simultaneously in order to provide multiple details about a single data points.

For more information about Music++, refer to Appendix A.

2.6 Related Work

Several other researchers have explored making version control data easier to understand and using sound in software engineering.

2.6.1 **CocoViz**

CocoViz is a tool that uses a combination of visualization and sonification to display code metrics for a software project. Each file in the repository is represented by a 3D shape that has several properties, including color, X and Y position on a grid, and volume. Each of these attributes is mapped to a different code metric, such as number of lines of code, number of functions, or Cyclomatic complexity. Which specific metrics are used is configurable by the end user, and CocoViz connects with IDE plug-ins in order to obtain metric data, giving users a large number of options for which metrics to select.

If the user maps all of the available visual parameters to metrics and wishes to view even more metrics, CocoViz uses sound to show the additional metrics. CocoViz's creators explored two different techniques with sound. In one, a marker is added to the visualization and can be moved around by the user. All of the 3D shapes continuously emit sounds, and the marker picks up the sounds of the 3D shapes closest to it as though it had a microphone attached to it. By moving the marker around, the user can get a sense of what kinds of values are typical in different portions of the codebase.

In the second technique, whenever the user hovers their mouse over a 3D shape, it plays its sound.

Unlike CocoViz, my work sonifies version control history. CocoViz focuses specifically on visualizing and sonifying code metrics without regard for version history [27,28].

2.6.2 **code_swarm**

Code_swarm is a tool that shows version control history in a way that makes it easy to see patterns in development. It uses a technique called organic visualization

which aims to make the qualitative aspects of data easier to understand instead of the quantitative aspects.

Code_swarm generates a video showing the history of a project. Each developer is represented by a piece of text with their name on it, and each file is represented by a dot. Whenever a developer makes a commit, all of the files that the developer modified and the developer's name all move closer to each other, as though they were briefly attached by invisible springs. As a result, over time, developers become surrounded by "bubbles" of dots showing which files they work on the most frequently. Other patterns, such as files gradually moving from one developer to another, can be easily observed.

Figure 2.5 shows an example of what code_swarm looks like.

Unlike my work, code_swarm does not use sound and does not show historical conflicts [1].

2.6.3 Orchestrating Change

Orchestrating change: An artistic representation of software evolution introduces a method to systematically generate music based on software projects' version control histories. Each component in the project is represented by a different musical motif, and each developer is represented by a different instrument. Each measure represents a period of time, such as a day or an hour. Then, for each measure, all of the commits created during that time period are converted into music by finding which software components were modified by which developers in each of the commits, then playing the corresponding motifs with the corresponding instruments [29].

Unlike my work, this paper's technique is solely intended to be enjoyable to listen to, not to make it easier to understand version control history. The Orchestrating Changes

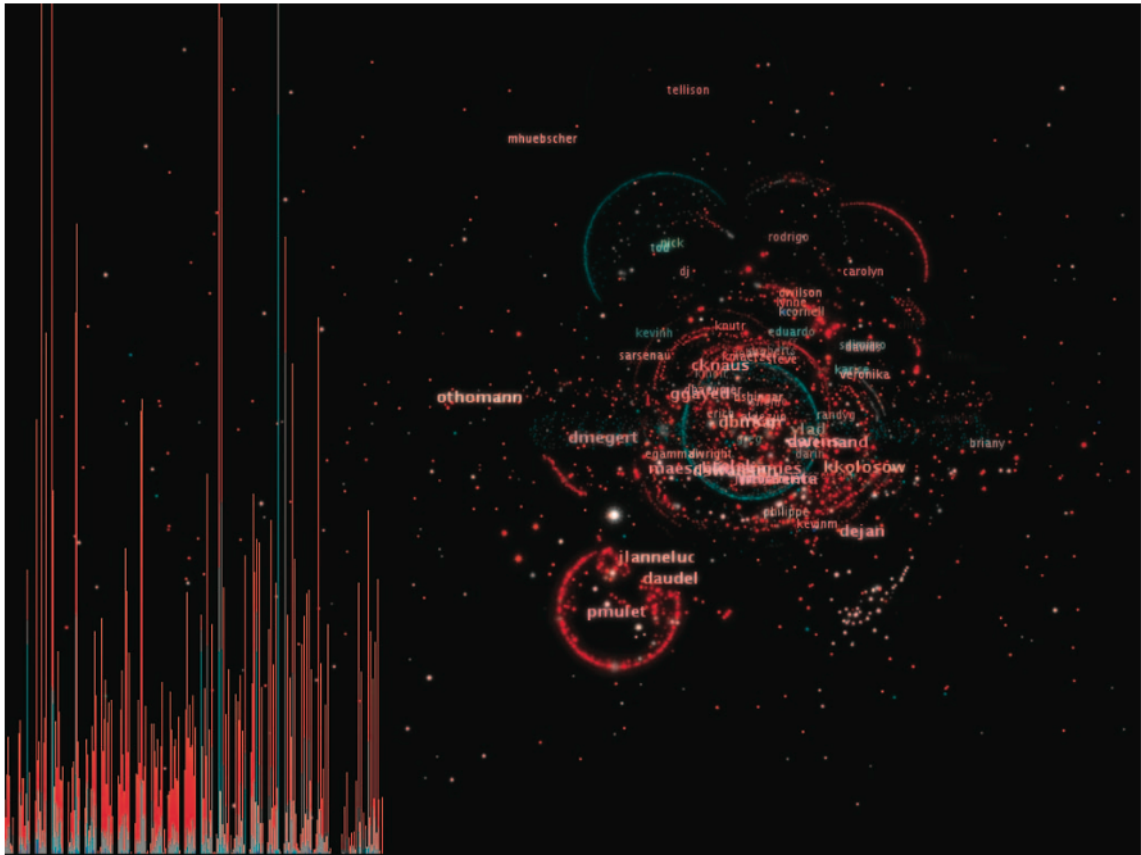


Figure 2.5: A still frame from a code_swarm video [1]

paper expressly states that its primary goals are to generate music that is aesthetically beautiful and that accurately reflects the history of the sonified repository, but it does not mention using the generated music to make the history easier to understand. Furthermore, this paper does not run an experiment to determine whether it's possible to understand the data represented by the music. [29] In contrast, my work's primary concern is to make version control history easy to understand, and making the sonification enjoyable is a secondary goal.

2.6.4 Other Applications of Sonification to Software

Engineering

Sonification has been successfully applied in many software engineering disciplines besides version control history.

One of the first tools to use sonification, SonicFinder, used sound to provide additional information to users of Apple's operating system in the 1980s. For example, dragging a file across the desktop generated a scraping sound. The larger the file, the lower-pitched the dragging sound was. As another example, selecting a file or folder generated a sound of hitting a wooden or metallic material. Again, the size of the file or folder being selected affected the pitch of the sound [25].

Hussein et al. show that sonification is a good alternative to visualization for program comprehension. They created a tool that sonifies the number of lines of code, the number of function calls, and the number of times a specific API is called by a Java program. They then ran an experiment where some participants used the sonification and other participants used a visualization portraying the same data. The experiment found that participants were able to answer questions as accurately and nearly as quickly when using the sonification compared to using the visualization. The visualization was still more usable, but the sonification was close enough to suggest that the technique could be improved until it was better than the visualization. [30].

Sonification has also been used for program comprehension [31], performance tuning [32], and debugging [33, 34].

While these projects provide an argument for sonification, my work focuses on version control history specifically, and version history is a separate subfield altogether than the examples listed above. There is no work that satisfies my goal of using sound to understand historical version control data.

Chapter 3

GitSonifier: Sonifying Version History

This section discusses GitSonifier, a technique that combines earcons to represent the history of Git repositories. The sonification represents who made each commit and when each commit was made using earcons. It also shows when conflicts were introduced and resolved in a repository's history using drums playing in the background.

In a user study on an open source project's data, I found that users can easily understand GitSonifier, indicating that sonification has the potential to help developers understand version control history more effectively. In addition, users can comprehend GitSonifier easily regardless of their prior experience with music.

Some of the material in this chapter was published in [20].

3.1 Motivation

Developers use Git routinely. Oftentimes, exploring Git history will help them accomplish different tasks. For example, developers often use Git history to help them fix bugs [3,4,5], understand why a piece of code evolved the way it did, or keep up with

changes to pieces of code they are actively working with [4]. Furthermore, as discussed in Section 2.3, merge conflicts can significantly slow down software development teams. Teams could look at when conflicts were introduced and resolved in their projects' histories in order to decide how to better handle and avoid conflicts in the future. To our knowledge, no one has considered showing developers historical conflict information before us.

There are several tools that already exist for displaying version control history, but they have limitations. Many tools, such as GitHub's network graph view, display a project's history as a directed graph [7]. These directed graph views have limited dimensionality, which means that there are already so many pieces of information mapped to different visual elements that trying to include additional types of information will quickly cause the visualization to become difficult to understand. Another tool, `code_swarm`, uses visuals to show historical data, but doesn't show information about conflicts [1]. Tools exist to let software developers know when they are conflicting with other developers, but these tools only show current conflicts, not historical conflicts [6, 8, 9].

As noted earlier in Section 2.4, sonification is an excellent tool to use for multidimensional and time-based data. In a version control history, each commit is associated with many different pieces of information, such as who made the commit and whether the commit introduces or resolves conflicts. In addition, commits record when they were created, so they occur over time. As a result, version control history is both multidimensional and time-based. Thus, by using sonification, we can potentially display more dimensions of version control history than we can with images. One of those dimensions can be when conflicts were introduced and resolved, providing useful information about conflicts to developers as well.

3.2 Design

GitSonifier uses earcons to create songs that represent version control history.

3.2.1 Developer Earcons

Each developer is assigned a musical motif as an earcon that is created using unique timbre, pitches, and rhythm. According to previous research, this makes them easier to differentiate [19]. To ensure that the earcons sound musically pleasant with each other, they are all written in the same key.

3.2.2 Representing Time with Day Separators

Each commit is represented by a measure of music, a decision we made because untrained listeners can intuitively tell when a measure begins and ends [23]. As a result, the amount of time the sonification plays to represent a day's worth of commits is longer or shorter depending on how many commits were made during the day. We used a variable length of time for a day in order to clearly represent activity sequentially. Since our goal was to hear each of the commits in order, we wanted to make sure that each commit could be clearly differentiated more than we wanted to closely simulate the passage of time.

Accordingly, each day is represented by an earcon called a day separator. The day separator plays for one measure when one day ends and another begins so that each day's commits are bookended by day separators. For instance, if developer A and developer B commit in that order one day, and then developer A makes another commit the next day, then developer A's earcon, developer B's earcon, a day separator, and developer A's earcon would play in that order for one measure each. On days

when no activity occurs, the day separator is played multiple consecutive times to show the number of days that pass.

To make it easier to identify, the day separator icon is written in a different key than the developer earcons. In addition, the day separator is played by a distinctive instrument, further differentiating it from the other earcons.

3.2.3 Conflict Drums

Conflict earcons are represented by a drum motif. The more unresolved conflicts there are during a commit or day separator, the louder the earcons will be during the corresponding measure. When a conflict is introduced, the drums start playing or get louder, and when a conflict is resolved, the drums get quieter. When there are no active conflicts, the drums do not play at all.

Conflict drums are played as an overlay onto the commit timeline. For example, if a developer makes a commit while there is one conflict active, the developer's earcon and the quiet drums signaling one commit will play simultaneously. We do this because conflicts can exist in a Git project concurrently with commits being added. In future work, we can parameterize and overlay additional data, such as the size of commits.

3.3 Implementation

Figure 3.1 shows how GitSonifier works¹. GitSonifier parses Git data (#2), flattens the data into a timeline (#3), adds a layer indicating the number of conflicts (#4), and renders the music clip (#5).

Our implementation of GitSonifier is compatible specifically with Git repositories.

¹GitSonifier sounds and experimental data can be found at: <http://cse.unl.edu/~myra/artifacts/NIER15/>

3.3.1 Architecture

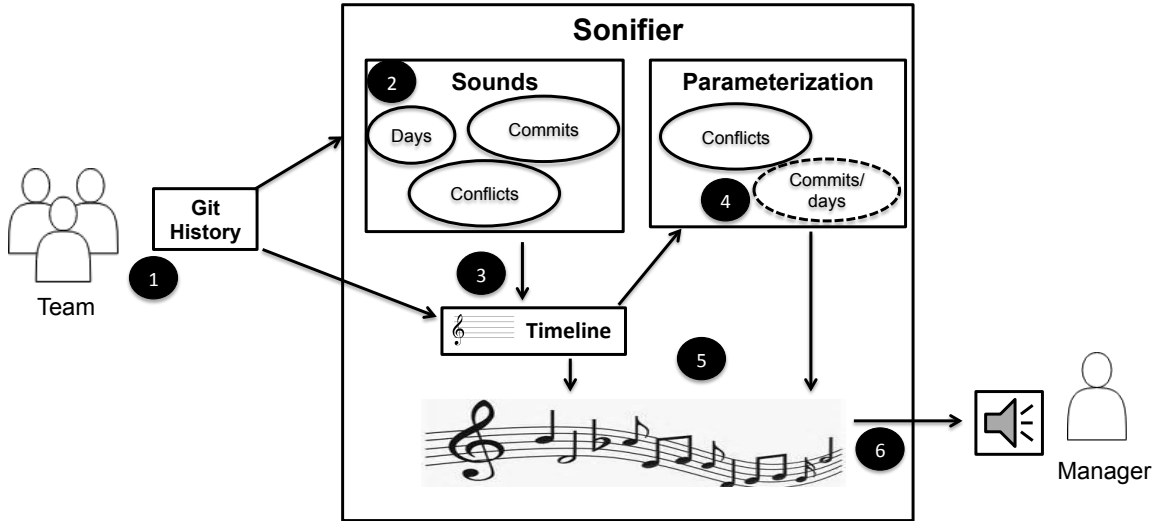


Figure 3.1: GitSonifier's Architecture

To start (#1), data is collected from a team's Git repository. GitSonifier walks through the Git repository, collecting each commit's SHA1 hash, author, timestamp, and list of parents. This information is combined with a list of when each conflict in the project's history was introduced and resolved. GitSonifier does not attempt to find conflicts, and the end user is expected to provide the conflict information themselves.²

Next, the Git data is passed to the sonifier module (step #2), which collects three key data elements for each commit: the commit's author, the day on which the commit was made, and whether the commit introduced or resolved conflicts. In step #3, the data for commit authors and days is arranged into a timeline. The timeline is represented by a list of **Measures**, where each **Measure** represents all of the information needed to sonify a single commit or day separator. The list is in the order the **Measures** will be played in the sonification.

²I want to thank Dr. Bakhtiar Kasi, one of my classmates while I was working on GitSonifier, for providing us with lists of conflicts he had found during his research.

In step #4, we use the information about which commits introduce and resolve conflicts in order to count how many conflicts are active during each commit and day separator. This is used to determine during which measures the conflict drums will play and how loudly they play when active.

Although steps #3 and #4 are conceptually separate, our software implementation performs these two steps simultaneously. A loop iterates over each commit in order and determines which conflict drums and developer earcon or day separator to combine for each measure of music. As a result, after a single pass, the information for both developer earcons and conflict drums has been processed.

Finally, in step #5 we combine the timeline data and the conflicts into a music clip, which is exported and played (step #6) for a manager or software development professional.

To implement GitSonifier, we created `.wav` files for our earcons. Each file is one measure long, making it easy to combine them. We used JGit, a library for working with Git repositories [35], to parse the repository's history. We then used Beads, a Java library, to combine the sounds on the timeline [36]. Beads can combine sounds both sequentially, playing commits and day separators in order, and concurrently, playing commits and conflicts at the same time.

3.3.2 Implementation Details of the Sonification

3.3.2.1 Developer Earcons

In the implementation we created, there are 14 developer earcons altogether, although in principal, this number can be easily changed. The 13 developers who make the most commits are each assigned their own earcon, while the remaining, less active developers share the 14th earcon.

We made this decision for three reasons. First, in the initial version control repository that we studied, we saw that the 13 most active developers contributed approximately 75% of the repository’s commits [37]. Second, assigning less active developers the same earcon makes it easy to tell when a commit was made by an outside developer. The 14th earcon uses a distinctive synthesizer for its instrument, making it easy to recognize. Finally, if we created one earcon for every developer in a repository, many of the earcons would start to sound very similar to each other, and it would be difficult for an end user to differentiate between them.

3.3.2.2 Conflict Drum Earcons

Our implementation of GitSonifier can portray up to four conflicts at once. If there are five or more simultaneous conflicts, the drums will continue playing, but they won’t get any louder. In principal, the maximum number of conflicts that GitSonifier can show can set to whatever value desired.

3.3.2.3 Tempo

The music generated by GitSonifier is played at a tempo of 100 beats per minute. In our informal experience, this is fast enough to be enjoyable, but slow enough to be understood.

3.4 User Evaluation

To evaluate whether sonification is an effective means for portraying development history, there are several questions that need to be answered. As an initial step, we performed an exploratory study to evaluate the basic assumption that developers can distinguish a set of information encoded in a sonification. Before we evaluate whether

users can listen and comprehend the music as part of a more complex tool or while performing other tasks, we needed to first determine whether sonification is feasible.

We asked the following three research questions:

- **RQ1:** How well do participants interpret the sounds representing a Git history?
- **RQ2:** How efficient is the use of sonification for understanding a Git history?
- **RQ3:** What was the participants' evaluation of sonification?

3.4.1 Participant Characteristics

We recruited six participants who have experience working in teams and using version control systems. They had experience in Git, SVN, and CVS, ranging from 1.5 months to 10 years. Most had experience working in small teams. Three had no music background, whereas the others had 10 to 25 years of musical experience. Four were male and two were female. Table 3.1 shows the demographics information for each participant.

Table 3.1: Demographics information for the participants in the GitSonifer study.

Participant	Which version control systems is the participant familiar with?	How long have they been using them?	How many people were on the largest software team they have worked in?	Do they have any musical experience?	If yes, how many years?	Gender	Year in school
P1	GitHub, CVS, SVN, a proprietary system	9-10 years	5-6 people	No		Male	Out of school
P2	SVN, Git, CVS	3 years	20-30 people	Yes	25 years	Male	Graduate student, 12 years
P3	Git	1 year	2 people	Yes	10 years	Female	Graduate student, 3 years
P4	GitHub	1 year	2 people	No		Female	Graduate student 1.5 years
P5	Git, SVN	10+ years	40-50 people	No		Male	Out of school
P6	GitHub	1.5 months	6 people	Yes	16 years	Male	Out of school

3.4.2 Study Design

We used GitSonifier to create a soundclip of the history of a GitHub project, Voldemort [37]. We sonified data from November 4 and 5, 2009, which includes 3 developers and 1 conflict. The conflict in this data was a test conflict, which means that Git was able to automatically merge the files together, but tests started failing in Voldemort after the merge. This resulted in a 26-second long clip. We then created ten variants of this clip by adding or removing users, days, and conflicts from the history. Some variants changed only one parameter, while others change two or all three parameters. Each variant includes 1 to 3 days, 2 to 4 developers, and 0 to 3 conflicts. Across all of the clips, we used 5 different developer earcons. Each sonification variant is 24-29 seconds long so that there is a significant amount of data encoded, but at the time same it is feasible to complete a set of 10 clips in the study time period. Of the 10 clips, one is the same as the original, and two, clips #4 and #8, are duplicates. Table 3.2 shows the properties of each clip.

Participants were trained by a website on what earcons meant in the sonification of the original data (from here on, I call it the training clip). They then had to complete a brief quiz about the data in the training clip. They could review the training clip and the earcons multiple times. Participants had to answer all the questions correctly before proceeding to the next part of the experiment, regardless of how long or how many tries it took them to do so.

In the next part of the experiment, the participants completed a set of ten tasks (one per variant). Each task had 3 multiple-choice questions that asked if the training clip and the variant clip had the same, more, or fewer developers, days, and conflicts. The training clip was included in the task description for review. We measured the amount of time it took each participant to complete each task and which questions

Table 3.2: The properties of each of the 10 clips used in the GitSonifier user study.

Clip	Number of Developers	Number of Days	Number of Conflicts	Length (in seconds)
Original / Training	3	2	1	26
1 [†]	3 [†]	3	1	29
2 [†]	3 [†]	2	2	26
3 [†]	3 [†]	2	1	26
4	2	2	1	26
5 [†]	3 [†]	1	2	24
6 [†]	3 [†]	2	0	26
7	4	1	3	29
8	2	2	1	26
9 [*]	3 [*]	1	0	24
10 [*]	3 [*]	2	2	26

[†] These clips had the exact same 3 developers as the training clip.

^{*} These clips had the same number of developers as the training clip, but they were a different set of three developers.

each participant answered correctly. Tasks were presented in a random order to minimize learning effects. Table 3.3 shows the questions that participants were asked.

If participants wished, they could see which questions they answered correctly after the experiment ended.

At the end of the experiment, participants filled out a questionnaire on the usability of the sonification. Each of the questions in the questionnaire was on a Likert scale from 1 to 5, 5 being the most positive. Table 3.4 in Section 3.5 below shows the questions. Finally, there was an exit interview in which we verbally asked participants about any unusual patterns we noticed.

Appendix B shows screenshots of the website we used to run the experiment. It shows the precise wording and format of the training and questions we showed to participants.

Table 3.3: The multiple-choice questions asked during each of the 10 tasks.

Question	Possible Answers
How many developers were there in this song compared to the original?	<ul style="list-style-type: none"> • There were more developers. • There were fewer developers. • There were the same number of developers [sic (from study)], but they were different developers. • There were the exact same developers.
How many days passed in this song compared to the original?	<ul style="list-style-type: none"> • More days passed. • Fewer days passed. • The same number of days passed.
How many conflicts were there in this song compared to the original?	<ul style="list-style-type: none"> • There were more conflicts. • There were fewer conflicts. • There were the same number of conflicts.

3.4.3 Threats to Validity

There are several threats to validity in this study. Several of these stem from the fact that this was an exploratory study. We wanted to quickly evaluate whether sonifying version control history was a worthwhile effort for us to invest more time and energy into, so we accepted some limitations on our study in order to obtain results more quickly.

3.4.3.1 Number of Participants

We had six participants in our study. The small number of participants prevents us from saying whether our results are statistically significant. Nonetheless, as will be discussed below in Section 3.5, we had very positive results.

In addition, our small number of participants means that our sample group may not be sufficiently heterogeneous in ways that made our results more positive or negative than we would have found with a more thoroughly randomized group.

3.4.3.2 Participant Characteristics

Our study recruited students and professors as participants. In fact, only one of our participants was a full-time developer. Thus, our results may not be indicative of how developers in industry would perform when using GitSonifier.

3.4.3.3 Task Design

Our tasks are not wholly representative of a real-world problem. While we used real-world data to design our training sonification clip, we used artificial version control data for the ten task clips. This is because we wanted to design our questions around comparing two different sonification clips in order to see whether users can understand GitSonifier well enough to see how the data between two different sonifications compare. As a result, we made sure the sonification clip participants compared against used real data. Then, we made ten sonification clips that modified that data to make clips that could be compared with the original.

Furthermore, the questions we asked participants tested whether the participants could comprehend the clips, but not how they would use the clips in a real-world scenario. As a result, we cannot be sure that the specific data we selected or the questions we asked about it are representative of the kinds of data and the questions developers would ask in real-world use.

3.4.3.4 Testing a Single System

We did not compare GitSonifier's performance against any other tools. Therefore, it is possible that, while GitSonifier performed well on its own, it may be more difficult to understand than other techniques and tools or only represent a marginal improvement on other tools.

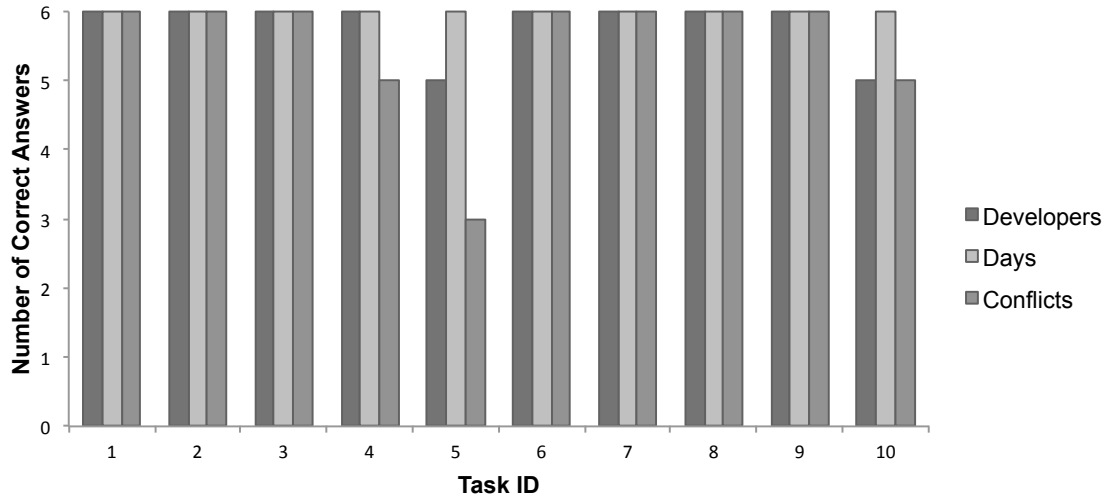


Figure 3.2: Correct Answers by Category (in order of task ID)

3.5 Results

In this section we discuss data related to each of our research questions in order.

3.5.1 RQ1: How well do participants interpret the sounds representing a Git history?

Figure 3.2 shows how many correct answers were given per sonification clip. The task IDs in the table correspond to the clip IDs in Table 3.2. Figure 3.3 shows how many correct answers were given on each question in the order the participants encountered each question. For example, Question 1 is the first task clip that each participant encountered, but not necessarily clip 1 in Table 3.2. Figure 3.4 shows how many questions in each category each participant answered correctly.

Each participant answered 27 to 30 questions out of 30 correctly, with an average of 29 correct answers per participant. Four of the six participants got all answers correct. Every participant answered every question about the number of days correctly. On

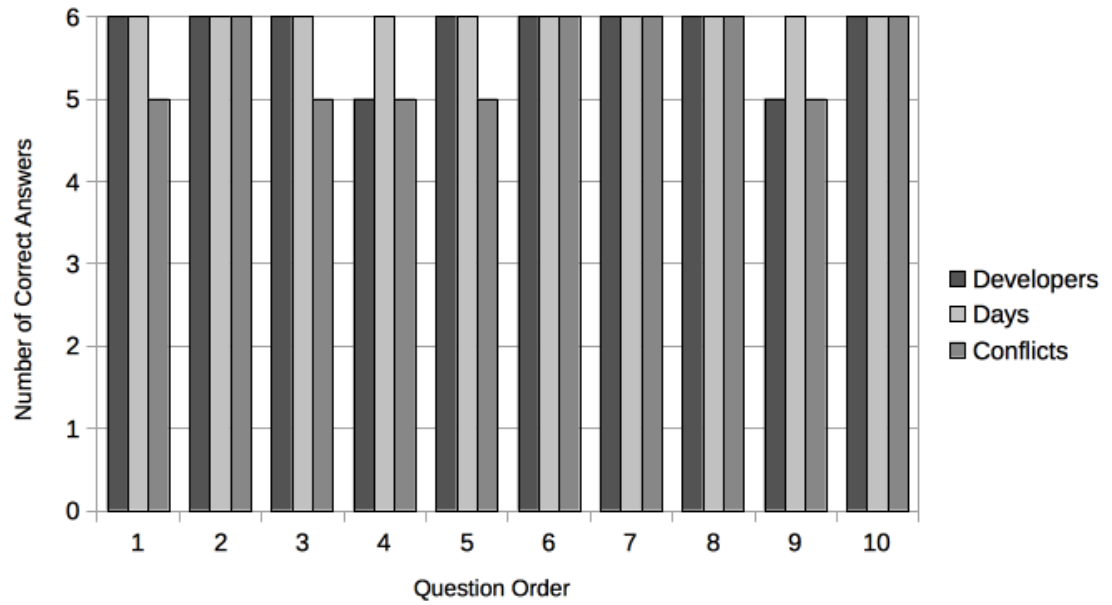


Figure 3.3: Correct Answers by Category (in the order participants encountered each task)

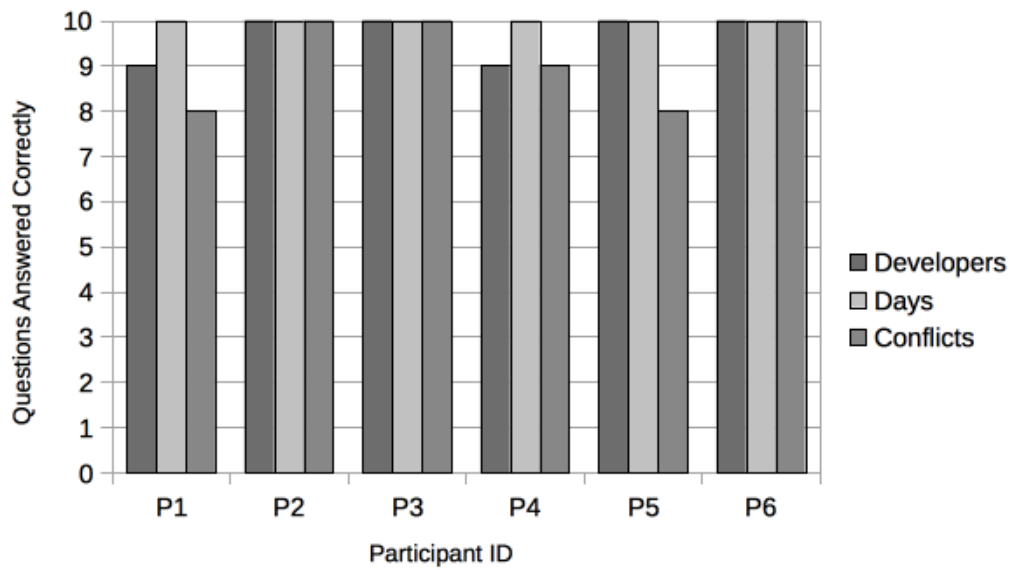


Figure 3.4: Correct Answers per Participant

five separate occasions, the participants answered questions about conflicts incorrectly. The participants answered three questions about developers incorrectly.

In terms of how many questions participants answered correctly, there were no discernible learning effects. Participants were as likely to miss some of the first questions they were shown as they were to answer later questions incorrectly.

Altogether, our results indicate that participants correctly understand GitSonifier.

3.5.2 RQ2: How efficient is the use of sonification for understanding a Git history?

Figure 3.5 shows the time to completion of each task, sorted in the order in which tasks were attempted. (Remember that the question order was randomized for each participant, so, for instance, the first question one participant answered might have been the last question for a different participant.) The overall average time to complete each task across all participants was 1 minute and 11 seconds. The longest time for a task was 6 minutes and 21 seconds, which was participant's P3 first task, but P3 rapidly improved while progressing through the tasks. Since each sonification clip was 24-29 seconds long, this means that participants typically listened to the clip 1 to 3 times while completing each task.

Four of the participants took roughly the same amount of time to answer each question. Participants P3 and P4 took several minutes to answer their first question, then quickly became faster at answering questions after that. After finishing 3 questions, both participants were as quick at answering questions as the other participants. This suggests that there is a learning effect in terms of how long it takes users to understand the tool: some users must spend a significant amount of time to understand the tool

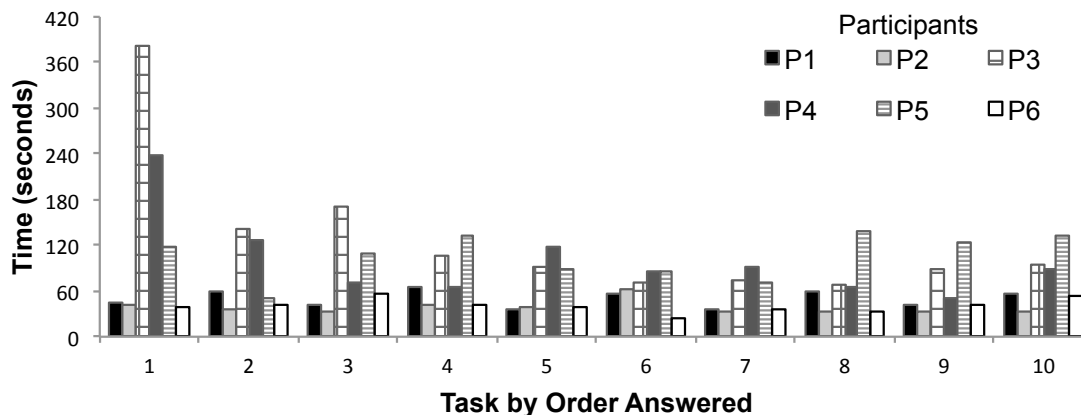


Figure 3.5: Amount of Time to Answer Each Question

at first, but they quickly move past this inexperience and can use the tool efficiently afterwards.

A key factor in how long it took to perform a task depends on how many times a participant listened to the sonifications. Three of the participants, P1, P2, and P6, answered most of the questions (per task) after only listening to the task sonification once or twice and without listening to the training clip. They were able to answer questions consistently in about 50 seconds.

Overall, participants were able complete the tasks reasonably quickly and often became faster as they progressed through the tasks.

3.5.3 RQ3: What was the participants' evaluation of sonification?

The exit survey included questions on the usability of the sonification on a Likert scale from 1 to 5, with 5 being the highest score. See Table 3.4 for the mean and median scores.

For most questions, the median score is 5 and the average score is more than 4.5.

Table 3.4: Questionnaire Results

Question	Mean	Median
It was easy to tell the different sounds apart	4.8	5.0
It was easy to hear who each developer was	4.8	5.0
It was easy to hear the number of conflicts	4.7	5.0
It was easy to hear when days passed	4.7	5.0
The sound helped me understand the data	4.2	4.0
I would be interested in hearing the development data of my own teams' projects	4.5	4.5

Even the lowest-scoring question has a median of 4 and an average of 4.2. The lowest score any participant gave for any question was 3, which only happened once — all of the remaining responses we received gave scores of 4 or 5. This shows that participants found the sonifications easy to understand.

3.6 Discussion

3.6.1 Listening to Sonifications

Four out of the six participants listened to the training clip at least once during the study. Two re-listened on their first and third tasks, respectively, to ensure that they remembered the original sonification. Another re-listened when they realized the current clip was the same as the training clip and wanted to confirm it. A fourth re-listened on questions 1 and 6 to double-check their work. When we asked participants why they did not listen to the training clip more often, they said that they had memorized the important information from the training clip. For example, one participant said, “*I kind of memorized the number and type of developers, the conflicts, and the number of days, so I didn't really need to [listen to the training clip again].*”

One participant with ten years of version control experience but no musical

background, P5, always listened to the current task both before and after answering each sub-question in order to be confident of their answers. This significantly added to their completion time.

3.6.2 Incorrect Responses

Only one question, Task 5 in Figure 3.2, received a notable number of incorrect responses. Three participants got the sub-question about conflicts wrong. The task's sonification had two conflicts, and the conflict that was introduced first was resolved one commit before the second conflict was introduced. As a result, drums appeared, then stopped for one measure, then restarted. This was likely the most challenging task for the participants because they failed to notice the brief silence between the two conflicts' drums. This suggests that a single measure of silence may not be enough.

3.6.3 Participants' Backgrounds with Music and Version

Control

All participants who had experience playing an instrument or composing music got a perfect score on each task. This suggests that people with music backgrounds had more success in interpreting sonifications. However, the effect size of this is not high: the lowest overall score that any participant got was 27 out of 30 questions correct. This suggests that while having musical experience lets one use sonification easily, sonification is nonetheless nearly as easy to use for users without musical experience.

There was no connection between experience and using sonification and there was no correlation between the participants' music experience and the time to completion for tasks. The two slowest participants (P3 and P4 in Table 3.5) were non-native English speakers. They quickly improved their times as they progressed. It's possible

that the learning effect they experienced was due to them reading the instructions in English instead of them having trouble using GitSonifier.

Experience in version control had no effect on the participants' results with GitSonifier.

3.6.4 Sonification Effects

When considering developer sounds, two participants missed one question each. One of them misread the question and realized this mistake on their own later. The other answered the question incorrectly and did not realize their mistake, although it was not clear why not.

Questions regarding conflicts had the most incorrect answers. As discussed above, task 5 in Figure 3.2 had a short gap between two sets of conflict drums, and 3 participants missed the “new” conflict. Two other questions regarding a conflict were missed as well, but the circumstances in which these questions were missed didn't have anything else in common.

All participants correctly answered questions about the number of days. The day separator sound may have stood out among the rest of the sounds because it was written in a different key and had a distinct motif. This might indicate that earcons that are very distinct from the rest of the sounds may work better. However, one needs to consider how the different notes sound when put together, and a series of earcons written in different keys may not be aesthetically pleasing. We may experiment overlaying different kinds and lengths of earcons in the future, especially for conflict earcons.

3.6.5 Exit Questionnaire

In the exit questionnaire results, one question, “The sound helped me understand the data.” received a considerably lower score than the other questions (although it still had a high score). The other questions ask whether participants can understand individual pieces of data from the sonification and whether the tool is enjoyable to use. The question with the lowest score, however, was about whether the tool made it easy to understand the data holistically. As a result, one of the weaknesses with GitSonifier is making it easy to understand the big picture of what is happening in a Git repository. As I will discuss later, my second technique, GitVS, addresses this problem.

3.7 Conclusions

GitSonifier introduces a novel technique for sonifying version control and conflict history data so that project managers or development team members can easily comprehend it. We have applied this to conflict data from a Git repository and performed a formative user study to evaluate whether or not users can understand the content by listening to the resulting sound clips. Our study shows that the majority of users were able to differentiate the data between clips and thought that differences were easy to hear.

GitSonifier does have some limitations. While participants were able to understand each individual piece of data in the sonification well enough, they reported on the end-of-study questionnaire that they didn’t feel completely comfortable understanding the data holistically. In addition, while the participants only needed to listen to each sonification clip one to three times to answer its questions, the sound clips themselves were 24 to 29 seconds long each and only sonified two days’ worth of history. Listening

to a longer period of history could become a time-consuming task, and jumping around in an audio file while keeping track of where one is in the data could be difficult.

The next chapter introduces another tool, GitVS, that addresses these limitations. It does so by combining GitSonifier's sonification technique with an interactive visualization, thereby gaining the benefits of both sonification and visualization.

Chapter 4

GitVS: Combining Visualization and Sonification To Display Version

History

In the previous chapter, I introduced GitSonifier, a sonification for presenting version control and conflict history to developers, and discussed a user study that shows GitSonifier is a viable approach. However, we discovered limitations of that technique and address some of those here. I have developed a hybrid technique, GitVS, that combines a static visualization with sonification to represent a richer set of data.

GitVS combines the sonification developed in GitSonifier with visualizations in order to be truly useful in real-world development situations. Combining sonification and visualization is helpful because, on their own, each technique is limited in the number of different dimensions of data it can show at once. By combining them, I show more pieces of data while still keeping the resulting hybrid visualization easy to understand. In addition, as shown in my GitSonifier experiment, a weakness of sonification is it can be difficult for a user to understand the data holistically. By

using a visualization, I make it easy to get a holistic view of the data being presented, then let users listen to the sonification for more details.

GitVS uses the same sonification design that GitSonifier introduced. However, GitVS is implemented using a completely different architecture.

4.1 Motivation

As discussed in Section 3.1, developers use Git history data for a wide variety of uses, and showing developers the history of conflicts can also be helpful. GitSonifier demonstrated that sonifying this data has potential. GitVS is an attempt to realize some of that potential.

While sonification has advantages for portraying multidimensional and time-based data [19], visualization has several advantages of its own. Unlike a sonification, it is easy to navigate to different points of a visualization merely by looking at a different location. Likewise, one does not have to look over the entire visualization before getting a general idea of the distribution or quality of the data portrayed, but a sonification makes it difficult to understand data holistically. For example, in the results for the GitSonifier study, participants reported that they could understand each individual part of the sonification easily enough, but they had a more difficult time understanding the data in general (see Section 3.6.5).

In addition, visualization and sonification both share a weakness: the more types of data one attempts to portray in a single visualization or sonification, the more difficult and unwieldy it becomes for the reader or listener to understand that data. CocoViz (see Section 2.6.1) demonstrated that by combining visualization and sonification, a visualization can be extended to include more types of data without becoming too cluttered to be easily understood [27, 28].

As a result, like CocoViz, GitVS combines sonification with visualization in order to gain the advantages of both techniques while losing the disadvantage of having limited dimensionality. GitVS uses visualization to show the branching and merging pattern of version history, when commits were made, and which files were modified in each commit. Then, sonification augments the visualization by indicating who made each commit and where conflicts are located. While it is possible to show all of this information in the visualization alone, the visualization would be complicated and visually busy, so it would be difficult to interpret as a result.

4.2 Use Case

As noted in Section 3.1, developers often explore Git history to accomplish different tasks, such as fixing bugs [3, 4, 5], understand how code evolves, or stay abreast of important changes [4]. All of these use cases can benefit from better techniques for exploring version history.

In this chapter, we focus on another use case in particular. There are many different workflows and methodologies for working with version control, and teams must decide on which methodologies to use. Examples of these methodologies include committing frequently [14, 15], using continuous integration to automatically test each commit [12], creating new branches for each new feature [13], and removing extraneous commits before pushing to a central repository [16]. Each of these workflows is intended to make teams more efficient, in part by changing the way they use version control. As a result, after implementing a new workflow, a team can evaluate whether the new workflow is having the intended helpful effect by analyzing their version control history.

In order to determine whether a new workflow is beneficial, there are many pieces of information that developers and managers are likely to want to know:

- Who made each commit. For example, Vasilescu et. al. have found that using continuous integration typically allows teams to accept changes from outside contributors to an open-source project more quickly [12]. Seeing who made each commit will allow teams to identify commits from outside developers and see whether their changes are being accepted.
- The branching and merging structure of the version history. This is important for seeing the effects of creating a unique branch for each figure [13].
- The number of files or changes in each commit, as well as when each commit was made. The idea behind committing frequently is that it is easier to share changes with teammates more quickly [14,15], so a team adopting such a workflow would probably expect to see fewer changes being made per commit and commits being made more frequently. Conversely, a team that removes extraneous commits before sharing them [16] would probably expect to see larger commits made less frequently in the history after small commits are removed.
- When merge conflicts appear in the history. As discussed in Section 2.3, merge conflicts can slow down a team significantly, so a workflow that reduces the number of conflicts may be preferable.

As a result, showing all of these pieces of data at once can help a team decide whether a new workflow is helping them.

4.3 Design

GitVS shows the following pieces of information about a version control repository:

1. When each commit in the repository was made.
2. The branching and merging patterns in the repository's history.
3. How many files were modified in each commit.
4. Each commit's ID.
5. Which specific files were modified by each commit.
6. Who made each commit.
7. Which commits are made by developers who don't commit frequently.
8. When conflicts were introduced and resolved in the project's history.

Our software implementation shows this information for Git repositories in particular, but the technique can be applied to other version control systems.

To show all of this information, GitVS relies on a combination of images, interactivity, and sound. In order to show all of the data listed above, GitVS:

- Represents each commit with a circle. The date and time each commit was made on is shown to the side of the commit. Figure 4.1 shows several commit circles.
- Visually shows the directed graph representation of the repository. Each branch is drawn with a different color in order to make it easier to differentiate between individual branches when there are many on screen at once. Figure 4.1 shows how the branches are represented.

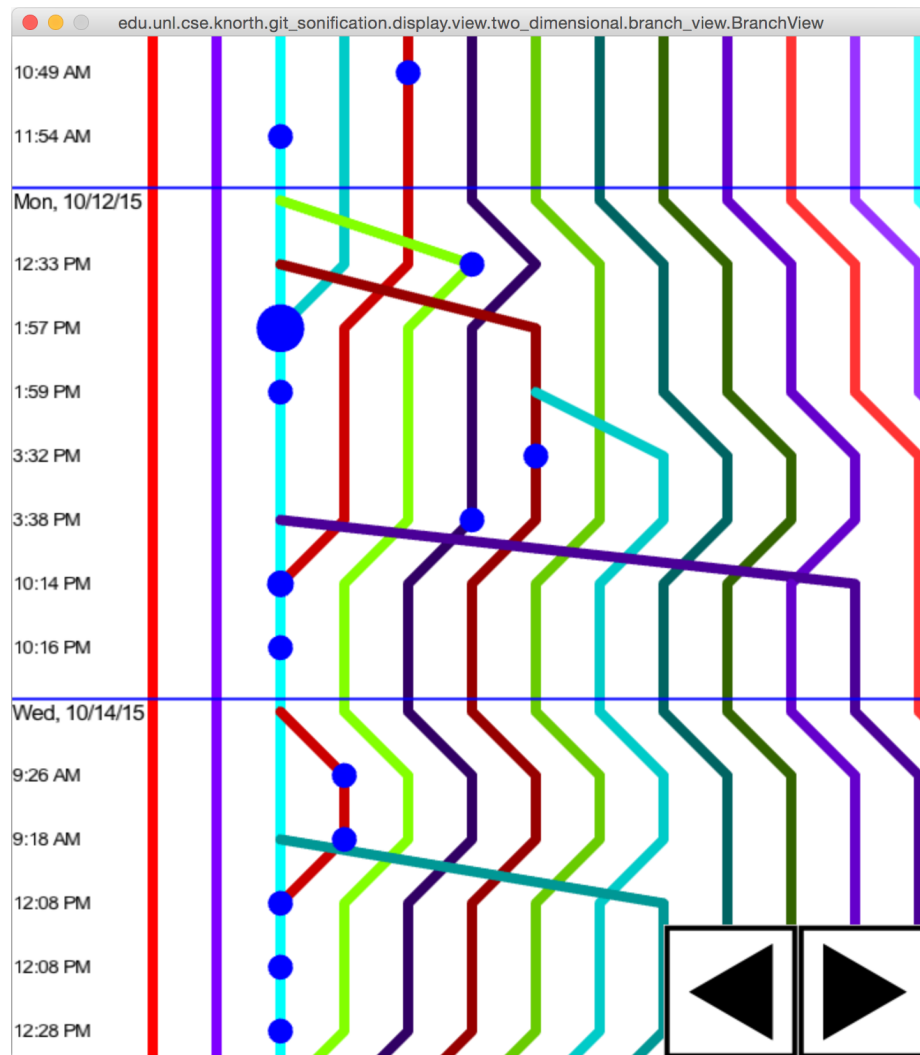


Figure 4.1: Screenshot of GitVS' View of Commits

- Shows the date and time that each commit was made along the left side of the visualization. Horizontal bars separate commits made on different days, making it easier to see how many commits were made each day. Figure 4.1 shows how the timestamps are displayed.
- Changes the size of the circles representing commits. The more files a commit modifies, the larger the commit's circle's radius is. Figure 4.1 shows several commits of different sizes.

Commit	Author	Timestamp	# of Files	.gitignore	CHANGELOG.md	DEVELOPER.md	README.markdown	SECURITY.md	bin/istorm	bin/istorm-config.cmd	bin/istorm.cmd
0bca173	Sean Zhong	Tue, 6/3/14 1:53 AM	1								
baee2f4	Robert (Bobby) Evans	Tue, 6/3/14 7:27 PM	85	●	●	●	●	●	●	●	●
6dae731	Robert (Bobby) Evans	Tue, 6/3/14 7:28 PM	1		●						
f6915d4	Sean Zhong	Wed, 6/4/14 5:14 AM	2								
a918059	Kyle Nusbaum	Wed, 6/4/14 1:35 PM	2	●	●		●				

Figure 4.2: Screenshot of GitVS' View of Selected Commits' Details

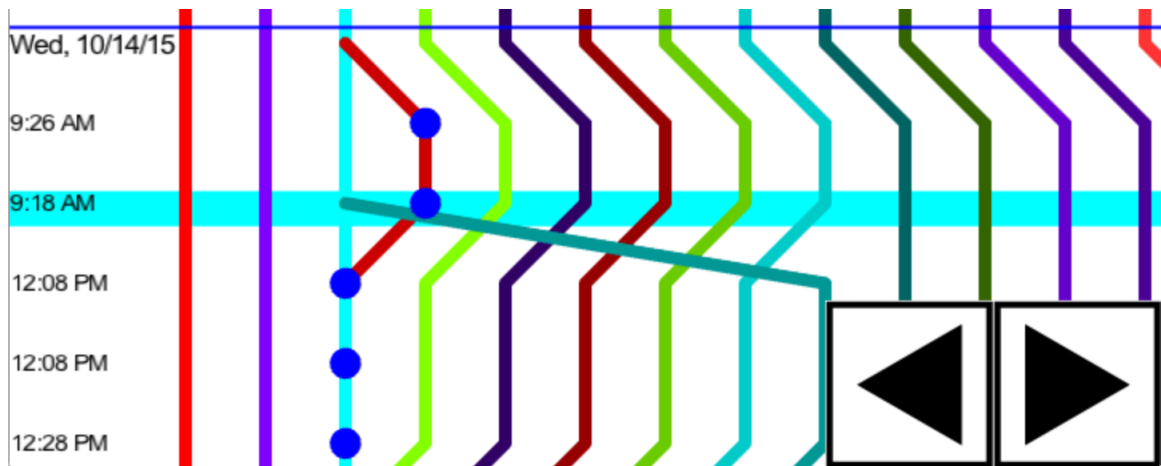


Figure 4.3: Screenshot of GitVS' Sonification Cursor. The sonification cursor is the cyan bar in the background.

- Allows users to click and drag over one or more commits to open an additional window that gives more details about the selected commits. This window shows the selected commits' IDs, timestamps, contributors, and numbers of files modified. It also includes a table showing which files each selected commit modified. Figure 4.2 shows the additional window.
- Plays sound interactively. A “sonification cursor,” a horizontal blue bar, is drawn in the background of the visualization. By clicking on buttons in the corner of the screen, the user can move the sonification cursor up and down.

When the sonification cursor touches a commit's circle, a developer earcon plays corresponding to the developer who made the commit. Figure 4.3 shows a screenshot of the sonification cursor.

- Uses the same sonification design as GitSonifier for the developer, conflict, and day separator sounds. The 13 most prolific developers each get their own individual developer earcon sound. All of the remaining developers share a 14th sound. Likewise, the more conflicts are active at a commit, the louder the drums in the background are.

In addition, when the sonification cursor passes a date, the day separator sound plays once for every calendar day in-between the current date and the previous one. This allows listeners to understand when multiple days have passed.

- Allows users to sonify specific commits at any time. By right-clicking, the user can jump the sonification cursor to the mouse. If the sonification cursor jumps on top of a commit, it will immediately play the commit's earcons.

4.4 Implementation

4.4.1 Architecture

Figure 4.4 shows the architecture of GitVS. GitVS is implemented in three main components:

1. The Data Collector component queries the Git repository being analyzed for its raw data.
2. The Data Processor transforms this data into a form that can be used to systematically create the visualization and sonification.

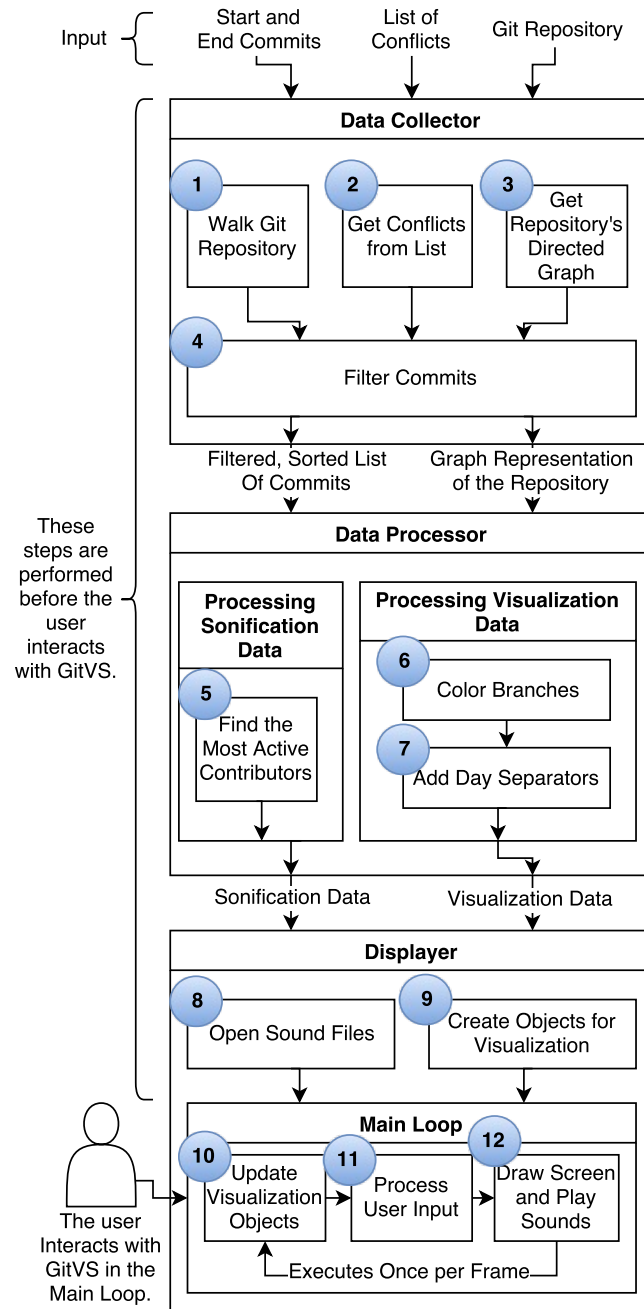


Figure 4.4: GitVS' Architecture

3. The Displayer shows the combination visualization and sonification.

Overall, these three components are fit together using a pipeline architecture. The

Data Collector feeds the data it collects to the Data Processor, and when the Data Processor finishes transforming it, the data is send to the Displayer.

While the sonifications in GitVS share the same earcon design as GitSonifier, our implementation of GitVS has a completely different architecture.

4.4.2 Input

GitVS takes the following information as input:

- The path to the repository to be displayed in the visualization.
- A list of conflicts present in the repository’s history. Like GitSonifier, GitVS does not find conflicts itself and expects the end user to provide it with historical conflicts. As a result, whether GitVS displays only conflicts that version control tools can detect automatically or also includes other types of conflicts depends on the conflicts that the end user gives as input. While working on GitVS, we developed a small script that finds historical conflicts that Git can automatically detect.
- The first and last commits to display in the visualization.

At first, it might seem more user-friendly to take in the first and last date to display, allowing participants to specify an input date range, instead of the first and last commits. However, that approach has a problem. It is common to find commits with timestamps out of order in real-world repositories. For example, Figure 4.5 shows commits timestamped October 7, 2009 appearing after commits timestamped October 9, 2009, even though they were added to the repository first! We found multiple examples of these achronological commits in both Voldemort and Storm, two repositories we used to test GitVS [37, 38].

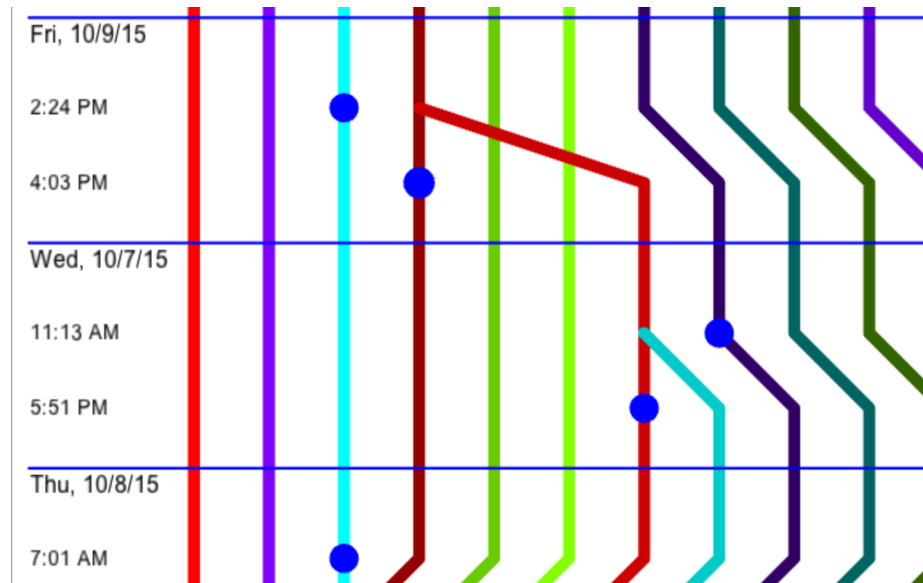


Figure 4.5: Example of Achronological Commits in GitVS

As a result, using dates to specify the time frame is ambiguous. For example, consider the example from Figure 4.5 again. If the user inputs October 9, 2009 for the starting date, the user probably wants to see the commits at the top of the figure. Since they are followed by commits timestamped October 7, 2009, should the commits from October 7 be displayed as well, even though they have dates before the input date? Letting the user enter the first and last commit to display avoids this ambiguity.

4.4.3 The Data Collector

4.4.3.1 Walking the Git Repository

The Data Collector performs several steps. First, using a library called JGit [35], the Data Collector walks the target Git repository and collects information on all of its commits. This is step #1 in Figure 4.4. The following pieces of data are collected for each commit:

- The commit's SHA1 hash.

- The commit's author.
- The timestamp indicating when the commit was made.
- A list of the commit's parents' SHA1 hashes.
- A list of the names of the files that the commit modified.

In addition, this step counts the number of commits each author made.

4.4.3.2 Obtaining Conflict Data

Next, the Data Collector reads the input list of when conflicts were introduced and resolved in the project's history. This is step #2 in Figure 4.4.

4.4.3.3 Getting the Directed Graph Representation of the Repository

In the next step, the Data Collector obtains a directed graph representation of the repository as described in Section 2.2.1.1. This is step #3 in Figure 4.4.

To obtain the directed graph, the Data Collector runs the following Git command:
`git log --format=%H --graph --author-date-order --no-color`

This command produces a visual representation of the repository's history as a directed graph. Figure 4.6 shows an example of the output generated by the command.

The Data Collector then parses the Git command's output and produces a data structure that can be used to recreate the visual representation of the graph later. The data structure has two main components:

- The first is information that describes how the visual representation of the repository's history can be recreated. It includes the location of each commit and each branch in the directed graph.
- The second component in the data structure is a topological sorting of the commits in the directed graph representation of the Git repository. A topological

```

* 7511c47dbbfa283a75e6984c48d7d2ca214c924f
*   cdb10a34c2302dc5998a7a651e8dc99daf04eab9
|\
| *   936eb0bda4388244a4fc9c78529368abd40eac8c
|  |\
* | | df7582f707c336b5dc75632a0568eedc814365ff
* | | 0223062ccdf2d51e1ce2b9c97b37ae2a67949fa6
| | *  cdcacd4e2391fa8a413436347466386e758fc7f0
| * | 55a693f7269ef830478a914383dfb10dd2e48d2f
|  | /
| *  fb25b61e6620e6b2ff6dac823eddc47000f743f6
* |   5410639d757426de5ad56d8fec33ab664bc761d
|\  \
| * | 9e57a7205c9caab78c28e5c8808ab4b0d3cb3456
| | * 09de305b49ea7e1c2759be23b8b752faa934047f
* | | 3e4ad808cae13234d45167e129b205767f1b2ec1
| * | 7111485a5fa2e627e04789bbf6e26aaf0220c5bd
|  | /
* | 9c222e6c981fc5a267f689092e891a733273168d
| /
* 5760e50293fca30ccfb99ca16b859a94da6b3cc0

```

Figure 4.6: Example Output of `git log --graph`

sorting is a sorting of elements in a directed graph such that no element in the sorted list appears before any of its parents [39].

This information is used in later steps to both decide which commits will be in the final visualization and to decide how to visually represent the directed graph.

4.4.3.4 Obtaining the Sorted List of Commits to Send to the Data Processor

Finally, the Data Collector produces a final list of commits to send to the Data Processor. This is step #4 in Figure 4.4. First, it filters the data by selecting all commits that appear in the topological sorting in-between the input starting commit

and ending commit. This gets a sublist of all commits that appear in the directed graph representation of the repository that are sandwiched between the two input commits.

As discussed in Section 4.4.2, we take specific commits as input instead of dates because commits frequently appear in the repository out of date order. For example, a commit timestamped October 9, 2009 might appear before a commit timestamped October 7, 2009, even though the former commit was added to the repository first. As a result, taking in two specific commits as input is much less ambiguous than taking a date range as input.

Next, the selected commits are sorted so that they are in the same order they appear in the topological sorting. This will make it easier to produce the visual representation of the directed graph later.

Finally, this sublist, the information about the visual representation of the repository, and the topological sorting are then sent to the Data Processor.

4.4.4 The Data Processor

The Data Processor is split into two main steps. The first prepares data for sonification, and the second prepares data for visualization.

4.4.4.1 Processing Sonification Data

Ultimately, each developer will be assigned an earcon based on how many commits they made relative to the other developers in the same way they were in the GitSonifier sonification. The most prolific developer will be assigned one earcon, the second developer will be assigned the next earcon, and so forth.

In order to prepare for assigning earcons to the correct developers, the Data

Processor produces a list of commit authors that is sorted in the order of how many commits each commit author made in descending order. This is step #5 in Figure 4.4.

4.4.4.2 Processing Visualization Data

Processing the data for visualization is split into two main steps: coloring branches and adding day separators.

Coloring Branches Each branch is assigned a different color so that when they are ultimately drawn in the visualization, they are easy to differentiate. This is performed in step #6 in Figure 4.4.

The colors are selected by iterating over each of the branches present at each commit. In each iteration, if the branch existed in the previous commit, it keeps its existing color. If the branch did not exist, then it is assigned a new color.

Preparing the Data Structure for Visualization. In the next step, step # 7 in Figure 4.4, the Data Processor adds day separators to the directed graph representation of the Git repository. Day separators indicate when a commit's timestamp is on a different day than its parent.

While GitVS represents day separators with sound, GitVS shows day separators both visually and sonically.

4.4.5 The Displayer

4.4.5.1 Design of the Displayer

The Displayer is implemented using the game loop pattern, a design pattern most typically found in the programming for video games and computer art projects. This pattern is useful when a GUI must update quickly in response to user input.

In the pattern, there is a main loop that is called once every frame when the computer updates what is displayed to the screen. First, classes that represent the entities that can be drawn to the screen are updated, allowing them to change state each frame automatically (step #10 in Figure 4.4). Then, the loop reads any input the user may be providing, such as pressing keyboard buttons or dragging the mouse, and performs more changes to the entities that are drawn to the screen accordingly (step #11). Finally, a draw function renders everything that will be shown on the computer screen based on the states of the entities in the previous steps (step #12) [40, 41].

For the Displayer component in GitVS, we implemented the game loop pattern using Processing [42], a Java library for working with sound and visuals programmatically. The main loop reads user input in order to allow the user to explore the information is displayed by the visualization and sonification. Then, all of the commits, their timestamps, and the branching and merging patterns are represented by a list of rendering objects. Depending on the user's input, the main loop will select the rendering objects that should be visible on the user's screen, then draw those.

4.4.5.2 Final Processing Steps

Once the Displayer component receives the data from the Data Processor, it performs some final processing before the first frame of the visualization is drawn.

First, it opens all of the sound files representing the developer and conflict earcons so that the earcons can be easily played later. This is step #8 in Figure 4.4.

After that, it creates all of the rendering objects that will be used to represent the data visually. Using the data produced by the Data Processor, the Displayer determines where each commit and branch line should be located in the visualization as well as what properties, such as their size or color, each commit and branch line should have. The Displayer also creates rendering objects representing the commits'

timestamps, the sonification cursor, and buttons used to control the sonification. This is all performed in step #9 in Figure 4.4.

4.4.5.3 Representing Commits Visually

The commit, branch line, and timestamp rendering objects produced in the previous step are used to show the visualization shown in Figure 4.1.

The user can use the arrow keys on the keyboard to move the screen up and down and side to side, thereby showing different portions of the graph visualization. They can also use the mouse wheel to zoom in and out.

4.4.5.4 Showing Details about Individual Commits Visually

When the user clicks and drags over one or more commits, a second window appears displaying the screen shown in Figure 4.2. This window works similarly to the main window: the table and all of the information contained within it is represented by a set of rendering objects that are created when the window first appears.

4.4.5.5 Developer and Conflict Earcons

The sonification cursor shown in Figure 4.3 moves up and down when the user clicks on the left and right triangle buttons, respectively. Whenever the sonification cursor touches a commit circle, the Displayer plays the commit's author's developer earcon.

The Displayer keeps track of a mapping between authors and the developer earcons that correspond to them. When a commit is touched by the sonification cursor, the Displayer looks up this map to determine which developer earcon to play. Similarly, the Displayer keeps track of a map of the number of simultaneous conflicts to the different conflict earcons and selects which conflict drums to play accordingly.

4.5 Planned User Evaluation

In order to determine whether the GitVS tool is truly helpful, we are conducting a user study similar to the study performed for GitSonifier. This study compares how effectively GitVS helps participants compared to a variant of GitVS without sound and to the GitHub network view [7], a state of the practice Git history visualizer.

The study was underway as of when this thesis was submitted.

We are asking the following research questions:

- **RQ1:** Can participants understand GitVS more correctly than existing tools?
- **RQ2:** Can participants use GitVS more quickly than existing tools?
- **RQ3:** Does sound help participants understand the data in GitVS?
- **RQ4:** How does the use of GitVS affect participants' perceptions of the Git repositories they analyze? How do these perceptions compare to when participants use other tools and GitVS without sound?
- **RQ5:** What are the participants' evaluation of GitVS compared to other tools and GitVS without sound?

The experiment uses a between-groups design. There are three groups. One group completes the experiment using GitVS, the second group completes the experiment using a version of GitVS without sound, and the third group completes the experiment using a tool on GitHub, the network view, a state of the practice website for working with open-source software projects [7].

For the group that uses GitVS without sound, we created a modified implementation of GitVS with the following changes:

- None of the commits make any sounds, and the sonification cursor has been removed.
- In the main view, commits that were made while at least one conflict was present are highlighted with a red outline. Figure 4.7 shows an example.
- When commits are selected, the table that appears has an additional column that shows the exact number of conflicts that were present during each commit. Figure 4.8 shows an example.

In order to see who made each commit, participants who use GitVS without sound can still look up committers' names in the commit details view.

The user study has four parts:

1. Completing training using the assigned tool.
2. Answering questions using the assigned tool.
3. Filling out a questionnaire about the assigned tool (post study).
4. An oral exit interview about the assigned tool (post study).

4.5.1 Participant Demographics

We are recruiting undergraduate and graduate computer science students for our study. Only students with experience with version control are able to participate in the study. We ask each participant how many years of experience they have with version control and music. Afterwards, we will see whether there are any correlations between participants' experience levels and the results we obtain in the user study. We hope to recruit at least 15 participants.

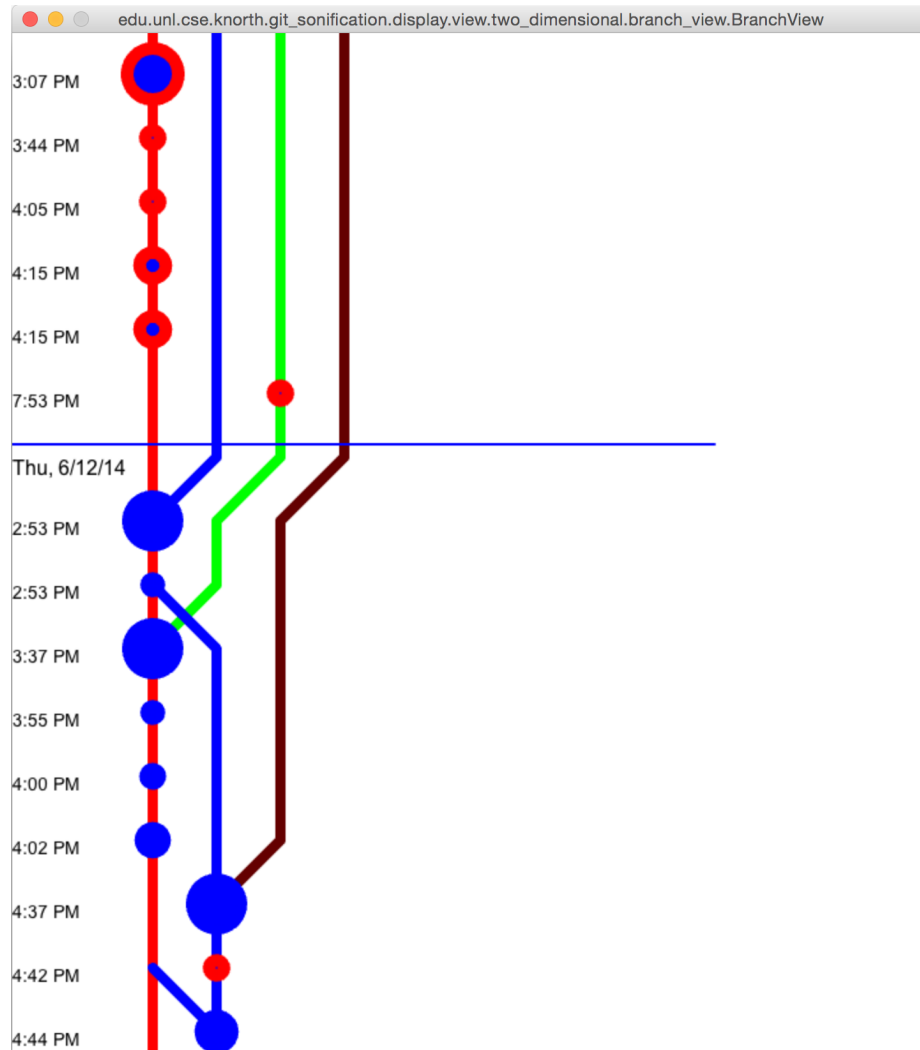


Figure 4.7: Screenshot of the View of Commits in GitVS Without Sound

4.5.2 Using the GitVS Tool

The main user study has two parts:

1. Training on the GitVS tool
2. A set of questions to test the effectiveness of GitVS

Immediately before beginning the study, each participant is assigned a tool at random. The first 10 participants are split randomly between using GitVS with sound

Commit	Author	Timestamp	# of Conflicts	# of Files	CHANGELOG	DEVELOP	LICENSE
ab99b91	P. Taylor Goetz	Wed, 6/11/14 3:07 PM	1	27			●
8cc69ec	P. Taylor Goetz	Wed, 6/11/14 3:44 PM	1	1			
bced187	P. Taylor Goetz	Wed, 6/11/14 4:05 PM	1	1			
2bb6e4e	P. Taylor Goetz	Wed, 6/11/14 4:15 PM	1	7			
2a60e99	P. Taylor Goetz	Wed, 6/11/14 4:15 PM	1	7			
8b57097	Sean Zhong	Wed, 6/11/14 7:53 PM	1	1			
862f673	Robert (Bobby) Evans	Thu, 6/12/14 2:53 PM	0	57	●	●	●
0826b93	Robert (Bobby) Evans	Thu, 6/12/14 2:53 PM	0	1	●		

Figure 4.8: Screenshot of the View of Selected Commits' Details in GitVS Without Sound

and the GitHub network view. The next 5 participants use GitVS without sound. Any additional participants are assigned treatments randomly. We are using this distribution because we are not sure how many participants we will be able to recruit before the end of the study, and we believe the research questions about comparing GitVS to existing tools are our top priorities.

4.5.2.1 Training Session

During the training session, participants are shown a video that demonstrates the use of the assigned tool. This example uses version control history from a dummy software project created for the demo.

Once the participants have been trained to use the tool, they are asked several questions about the data portrayed by the tool. The participant is allowed to move to the experiment tasks only after they have correctly answered the training questions. When answering the questions, they can look over the training video and use the assigned tool however they wish. No data is collected during this phase.

4.5.2.2 Experiment Tasks

Based on the use case we are using to design GitVS, participants take on the role of the manager of a software development team. In the scenario, their team currently uses Git with the following formal processes for coordinating version control:

1. Whenever a new feature is started, the developers can choose to create a new branch for that feature, but do not have to.
2. Before merging a completed feature, branches will be code-reviewed.

They are considering switching to all of the following processes:

1. Whenever a new feature is started, the developers must create a new branch for that feature.
2. Before merging a completed feature, branches will be code-reviewed.
3. All commits will be sent to a continuous integration sever, which will automatically build and test each commit.

Notice that in the new list of processes, process #1 is stricter, and process #3 has been added.

For process #3, many software development teams use continuous integration servers. A continuous integration server automatically pulls, builds, and tests each commit made to a repository. If there are any compile errors or failing tests, the server will notify the team members that there was a problem. Vasilescu et. al. found that open-source development teams which use continuous integration merge more pull requests per month from core developers and reject fewer pull requests per month from outside contributors. In addition, core developers find more bugs on teams using continuous integration, meaning that continuous integration makes it easier to discover and proactively resolve faults [12]. Continuous integration doesn't have a direct effect

on the patterns a team uses to create commits, but as Vasilescu et. al. have shown, it has measurable indirect effects.

The manager wants to understand how these three new processes will affect their team, so they have found another project that has a similar amount of activity to theirs, but follows all of the processes they want to use.

In the experiment, the participant is shown, one at a time, the version control history for both the manager's current project and the project that already follows the new processes. Two weeks' worth of development history are shown to the participants for each project. The order the two repositories are shown to participants is random but counterbalanced.

In addition, participants are given lists of core developers and outside developers for each repository. Participants who use GitVS are able to hear the earcons that correspond to outside developers.

Although the identities of the projects are not shown to the participants, both of the projects' data is from real-world open-source repositories. Voldemort is a distributed storage system [37], and Storm is a distributed computation system similar to Hadoop [38]. Voldemort has over 4,100 commits and received its first commit on April 17, 2011 [37], and Storm has over 6,500 commits and received its first commit on September 15, 2011 [38], so the two projects have received a similar amount of activity and are about the same age. Voldemort uses the formal methodologies that the participant's hypothetical repository uses [43], while Storm uses the formal methodologies that the repository the participant is comparing against uses [44, 45].

Each participant is asked a series of questions with objective answers, listed in Table 4.1, for each of the two projects, one project at a time. Then, they are asked a series of subjective questions, listed in Table 4.2, asking them to compare the two

Table 4.1: Objective questions that participants answer for each repository.

Question
How many commits were made in-between [three of the days portrayed by the tool]?
On average, what is the number of files modified per commit in-between [three of the days portrayed by the tool]?
How many conflicts were present in-between [the first and last date portrayed by the tool]?
Look at the branches which were merged directly into the master branch. In days, what is the average number of days these branches existed?
What is the average number of days each branch with at least one commit from an outsider developer existed before being merged?

Table 4.2: Subjective questions that ask participants to compare the repositories.

Question
Based on the previous questions, which process would you recommend if your priority is to have a large number of small commits ?
Based on the previous questions, which process would you recommend if your priority is to minimize conflicts ?
Based on the previous questions, which process would you recommend if your priority is to accept pull requests quickly ?
Based on the previous questions, which process would you recommend if your priority is to quickly merge commits from outside developers ?

projects. For the first questions with objective answers, we also calculate how long it takes participants to answer each question.

When necessary, we answer questions the participants ask about how to correctly interpret the questions. Furthermore, if a participant takes longer than 5 minutes to answer one of the questions with objectively correct answers, we let them skip the question, but they do not have to do so.

Although participants are not told this information, each of the comparison questions is based off of what the literature says about how methodologies will affect teams' efficiencies. The question about having a large number of small commits is based off best practices in industry which recommend developers make commits

frequently. This makes it easier for developers to quickly share changes with their teammates [14,15]. The question about minimizing conflicts comes from the literature on how merge conflicts adversely affect teams, which is covered in Section 2.3. The remaining two questions about accepting pull requests and accepting contributions from outside developers comes from research by Vasilescu et. al. that found that open-source projects that use continuous integration take less time to accept GitHub pull requests and accept contributions from outside developers more quickly [12].

Each of the objective questions corresponds to one of the comparison questions. When participants answer the comparison questions, they are able to review their answers to the objective questions. This lets them use their answers as evidence to support which process they select for each question.

Appendix C shows screenshots of the website we are using to run the experiment. It shows the precise wording and format of the training and questions we showed to participants.

4.5.3 Post Study: Filling out a questionnaire about our GitVS tool

After the experiment study, participants fill out a survey about how the participant felt using their assigned tool. 10 of the questions, shown in Table 4.3, are Likert scale questions on a scale of 1 to 5. For each of the three questions that end, “(Why or why not?)” the participant is also asked to write a short answer explaining why they selected the Likert rating they did for that question.

2 of the questions are strictly open-ended. The open ended questions are:

- Would you recommend this tool to others?
- In your opinion, how can we improve the tool further?

Table 4.3: Post-study questionnaire questions.

Question
I could tell where commits were located.
I could identify who made each commit.
I could tell where conflicts were located.
I could tell when each commit was made.
I could see which files were changed by each commit.
I could tell where branches were located.
I could understand the details of the development data. (Why or why not?)
I could understand overall patterns in the development data. (Why or why not?)
I could see relationships between different pieces of development data. (Why or why not?)
I would be interested in using this tool for my own team.

The first open-ended question is similar to the Likert-scale question asking, “I would be interested in using this tool for my own team.” There are two main differences between the two questions. First, “Would you recommend this tool to others?” asks for a short written response, while, “I would be interested in using this tool for my own team,” asks only for a Likert rating. Second, it’s possible that a participant enjoyed using the tool they were assigned themselves but doubt other people would enjoy it. The phrasing of the question, “Would you recommend this tool to others?” allows us to learn whether that’s the case.

4.5.4 Post Study: Completing an oral exit interview

Finally, we conduct and audio record unstructured oral exit interviews. There is no script for these interviews. We ask each participant about any specific activity where the participant apparently had difficulty or otherwise performed in a way that was notable or unusual.

4.6 Conclusion

GitVS combines the sonification from GitSonifier with an interactive visualization. This visualization allows GitVS to overcome the disadvantages of sonification, including the fact that a lone sonification must be listened to completely to get a holistic view of the data and that navigating a sonification by itself is difficult. Nonetheless, GitVS keeps sonification's advantages of being a good fit for multidimensional and time-based data. In addition, the combination of visualization and sonification allows GitVS to display more dimensions of data than other tools without becoming overwhelming.

We have designed a user study that will be conducted to evaluate GitVS. This study will demonstrate whether GitVS is more useful than existing tools.

Chapter 5

Conclusions and Future Work

5.1 Conclusion

In this thesis, I introduced GitSonifier, a technique that sonifies version control history and historical conflict data. Then, I performed a user study which shows the GitSonifier sonification is easy to understand. Finally, I created GitVS, which combines the sonification from GitSonifier with a visualization in order to produce a technique that will be helpful to real developers.

Throughout all three tools, I used earcon sonification, a sonification technique that represents data using sound. In GitSonifier and GitVS, each developer is assigned a different earcon. In addition, the number of conflicts active in a project is represented using drum earcons. By combining the developer earcons and conflict drums, I was able to create a sonification that showed multiple pieces of information about each commit in a repository while remaining easy to understand.

In addition, by sonifying historical conflict data, GitSonifier and GitVS are the first techniques to help developers understand when conflicts occurred in their projects' histories.

5.2 Future Work

5.2.1 User Study for GitVS

As discussed in the chapter on GitVS, I have designed a user study for GitVS. I will carry out the experiment as described in Section 4.5.

5.2.2 Sonifying Additional Layers of Information

In GitSonifier and GitVS, I used sonification to represent when historical conflicts were introduced and resolved. In theory, we can easily design earcons for other layers of information. For example, we could use sonification to indicate when bugs were discovered and resolved in history. We could also use sonification to indicate which commits correspond to major releases. In the future, we can experiment with designing such layers.

5.2.3 Sonification Design Ideas

We had several ideas for how to design our sonification, and we will implement these designs, then run user studies to compare them to the sonification design we used in GitSonifier and GitVS. One idea is to create a conflict resolution earcon in order to make it easier to interpret when conflicts end. Since conflicts are represented by drums, the conflict resolution earcon can be a long cymbal crash, a sound that is typically used in music to signal when drums are finished playing.

Furthermore, we can write each developer earcon in a different key in order to make the developer earcons easier to distinguish. We can experiment both with changing the tonal center, such as writing one earcon in the key of C major and another in the key of G major, and with writing earcons in major and minor keys. This might

make the earcons easier to understand by making them even more different from each other. However, it also might make them more difficult to understand by destroying the musicality of the generated music. We will conduct experiments to determine which is the case.

Another idea is to use hierarchical earcons. This is a sonification technique in which different musical aspects of an earcon are mapped to different pieces of data. For example, perhaps which instrument plays an earcon can be mapped to the developer who made the commit, then the notes the earcon plays can indicate how many files were modified in the commit. [19] In this way, we can use a single layer of music in a sonification to portray more data at once.

5.2.4 Designing Tools for Additional Use Cases

In the chapters on GitSonifier and GitVS, I discussed two specific use cases for sonification of version control. In the future, I will evaluate alternative use cases for these techniques.

One use case is helping managers keep track of their teams' performances. At the beginning of each day or week, a manager could listen to a sonification portraying performance metrics obtained from version control history. While listening to the sonification, the manager could do other tasks, such as reading email. If any of the data is unusual or potentially indicates problems, it could be designed in a way to grab the manager's attention, allowing them to temporarily stop what they're working on, listen to the sonification, and get more details about the situation. To my knowledge, there is not much literature suggesting that this is an important use case, so this use case was not our first priority even though I find it interesting.

Another use case is helping developers debug programs. Developers frequently use

version control history to help them diagnose bugs [3,4,5]. A sonification tool could potentially help developers discover the commits that are relevant to their debugging tasks more quickly. Unlike GitVS, which shows a large amount of data at once and allows users to make their own interpretations of that data, a debugging tool would focus on helping developers quickly decide which pieces of information are relevant to their search and allowing them to process just that information quickly.

Bibliography

- [1] M. Ogawa and K.-L. Ma, “code_swarm: A design study in organic software visualization,” *Visualization and Computer Graphics, IEEE Transactions on*, vol. 15, no. 6, pp. 1097–1104, Nov 2009.
- [2] “Git - about version control,” <https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>, accessed: 2016-03-28.
- [3] A. J. Ko, R. DeLine, and G. Venolia, “Information needs in collocated software development teams,” in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, May 2007, pp. 344–353.
- [4] M. Codoban, S. S. Ragavan, D. Dig, and B. Bailey, “Software history under the lens: a study on why and how developers examine it,” in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE, 2015, pp. 1–10.
- [5] D. L. Atkins, *System Configuration Management: ECOOP’98 SCM-8 Symposium Brussels, Belgium, July 20–21, 1998 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, ch. Version sensitive editing: Change history as a programming tool, pp. 146–157. [Online]. Available: <http://dx.doi.org/10.1007/BFb0053886>

- [6] M. L. Guimarães and A. R. Silva, “Improving early detection of software merge conflicts,” in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 342–352.
- [7] “Say hello to the network graph visualizer,” <https://github.com/blog/39-say-hello-to-the-network-graph-visualizer>, accessed: 2016-03-16.
- [8] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, “Proactive detection of collaboration conflicts,” in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 168–178.
- [9] A. Sarma, D. F. Redmiles, and A. Van Der Hoek, “Palantír: Early detection of development conflicts arising from parallel code changes,” *Software Engineering, IEEE Transactions on*, vol. 38, no. 4, pp. 889–908, 2012.
- [10] K. North, “How version control conflicts affect the software development process,” University of Nebraska–Lincoln, Tech. Rep., May 2015.
- [11] “Git - documentation,” <https://git-scm.com/doc>, accessed: 2016-03-15.
- [12] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, “Quality and productivity outcomes relating to continuous integration in github,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 805–816. [Online]. Available: <http://doi.acm.org/10.1145/2786805.2786850>
- [13] “Git workflows and tutorials | atlassian git tutorial,” <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>, accessed: 2016-03-28.

- [14] “Check in early, check in often,” <https://blog.codinghorror.com/check-in-early-check-in-often/>, accessed: 2016-03-27.
- [15] “Version control best practices,” <https://www.git-tower.com/learn/git/ebook/command-line/appendix/best-practices>, accessed: 2016-03-27.
- [16] “Understanding the git workflow,” <https://sandofsky.com/blog/git-workflow.html>, accessed: 2016-03-28.
- [17] B. K. Kasi and A. Sarma, “Cassandra: Proactive conflict minimization through optimized task scheduling,” in *Software Engineering (ICSE), 2013 35th International Conference on*, May 2013, pp. 732–741.
- [18] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, “Proactive detection of collaboration conflicts,” in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 168–178.
- [19] T. Hermann, A. Hunt, and J. G. Neuhoff, *The sonification handbook*. Logos Verlag Berlin, 2011.
- [20] K. J. North, S. Bolan, A. Sarma, and M. B. Cohen, “Gitsonifier: Using sound to portray developer conflict history,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 886–889. [Online]. Available: <http://doi.acm.org/10.1145/2786805.2803199>
- [21] “Git - git-show documentation,” <https://git-scm.com/docs/git-show>, accessed: 2016-03-25.

- [22] M. M. Blattner, D. A. Sumikawa, and R. M. Greenberg, "Earcons and icons: Their structure and common design principles," *Human-Computer Interaction*, vol. 4, no. 1, pp. 11–44, 1989. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1207/s15327051hci0401_1
- [23] M. Schonbrun, *Reading Music: A Step-By-Step Introduction To Understanding Music Notation And Theory*. Fall River Press, December 2012.
- [24] S. Maturu and K. North, "Sonification of the etoc genetic algorithm process," University of Nebraska–Lincoln, Tech. Rep., December 2014.
- [25] W. W. Gaver, "The SonicFinder: An interface that uses auditory icons," *HCI*, vol. 4, no. 1, pp. 67–94, Mar. 1989.
- [26] N. Pickrel, N. Jahnke, J. Lloyd, and K. North, "Final report: Music++," University of Nebraska–Lincoln, Tech. Rep., April 2014.
- [27] S. Boccuzzo and H. C. Gall, "Cocoviz with ambient audio software exploration," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 571–574. [Online]. Available: <http://dx.doi.org/10.1109/ICSE.2009.5070558>
- [28] S. Boccuzzo and H. Gall, "Software visualization with audio supported cognitive glyphs," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, Sept 2008, pp. 366–375.
- [29] S. McIntosh, K. Legere, and A. Hassan, "Orchestrating change: An artistic representation of software evolution," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, Feb 2014, pp. 348–352.

- [30] K. Hussein, E. Tilevich, I. Bukvic, and S. Kim, “Sonification design guidelines to enhance program comprehension,” in *ICPC*, May 2009, pp. 120–129.
- [31] A. Stefik, K. Fitz, and R. Alexander, “Layered program auralization: Using music to increase runtime program comprehension and debugging effectiveness,” pp. 89–93, 2006.
- [32] C. Henthorne and E. Tilevich, “Sonifying performance data to facilitate tuning of complex systems: Performance tuning: Music to my ears,” in *OOPSLA*, 2010, pp. 35–42.
- [33] A. Stefik, K. Fitz, and R. T. Alexander, “Increasing fault detection effectiveness using layered program auralization.” in *Software engineering research and practice*, 2006, pp. 959–965.
- [34] P. Vickers and J. L. Alty, “Siren songs and swan songs: Debugging with music,” *Com. ACM*, vol. 46, no. 7, Jul. 2003.
- [35] “Jgit,” <https://eclipse.org/jgit>, accessed: 2014-12-14.
- [36] “Beads,” <http://www.beadsproject.net/>, accessed: 2015-06-04.
- [37] “voldemort/voldemort,” <https://github.com/voldemort/voldemort>, accessed: 2016-03-20.
- [38] “apache/storm,” <https://github.com/apache/storm>, accessed: 2016-03-20.
- [39] M. C. Golumbic, *Algorithmic graph theory and perfect graphs*. Elsevier, 2004, vol. 57.
- [40] “Game loop,” <http://gameprogrammingpatterns.com/game-loop.html>, accessed: 2016-03-02.

- [41] “Anatomy of a video game,” <https://developer.mozilla.org/en-US/docs/Games/Anatomy>, accessed: 2016-03-02.
- [42] “Processing.org,” <https://processing.org/>, accessed: 2015-07-17.
- [43] “Developer info - voldemort,” <http://www.project-voldemort.com/voldemort/developer.html>, accessed: 2016-03-20.
- [44] “Developer documentation,” <https://github.com/apache/storm/blob/master/DEVELOPER.md>, accessed: 2016-03-20.
- [45] “apache/storm - travis ci,” <https://travis-ci.org/apache/storm>, accessed: 2016-03-20.
- [46] “cobertura/cobertura,” <https://github.com/cobertura/cobertura>, accessed: 2016-03-20.
- [47] “Overview of the midi package,” <https://docs.oracle.com/javase/tutorial/sound/overview-MIDI.html>, accessed: 2016-03-20.

Appendix A

Music++

Music++ is a technique that sonifies the execution paths of simple Java programs. The tool has multiple earcons that correspond to different elements in a Java program, and when each line of code in a sonified Java program's execution path is reached, the earcons for that line of code's elements play simultaneously. Music++ can be used to help teach programming by demonstrating to students how their computers "see" their programs when the programs are run.

Music++ was an early proof-of-concept program for my research that demonstrated sonification is a viable technique in software engineering. The idea of using earcons to represent pieces of data in order to produce a song is a common theme in my research.

Music++ was a joint project with Dr. Myra Cohen, Dr. Anita Sarma, and my classmates Nina Pickrel, Jacob Lloyd, and Natasha Jahnke. I worked on Music++ as an undergraduate student.

This appendix chapter includes material from a class paper that Nina Pickrel, Jacob Lloyd, Natasha Jahnke, and I worked on together [26].

A.1 Design

The purpose of Music++ was to explore whether sonification is a viable approach to software engineering. In addition, the use case for Music++, teaching students how a computer “understands” their Java programs, doesn’t require Music++ to implement complex functionality that would be more appropriate for experienced users. As a result, Music++ can only sonify relatively simple Java programs, but the sonification design is robust.

A.1.1 Limitations

Music++ has several limitations on the input files it can process. This section explains those limitations.

An input program can only contain one class with a main function and no other functions. The contents of the main function will be sonified. The class cannot contain any class or instance variables.

Music++ can only parse the following types of statements:

- Variable declarations
- Assignments to variables with simple mathematical expressions on the right-hand side
- If, else, and switch statements
- For, while, and do/while loops

Music++ will only work with variables that have one of seven built-in types. Table A.1 lists the built-in types that Music++ can parse.

When parsing the right-hand side of an assignment statement, the right-hand side is limited to performing one operation on two operands. For example, adding two

Table A.1: List of the types of variables that Music++ can parse.

Type
<code>int</code>
<code>double</code>
<code>String</code>
<code>byte</code>
<code>float</code>
<code>char</code>
<code>boolean</code>

Table A.2: List of the arithmetic operators that Music++ can parse.

Arithmetic operation	Token in Java
Addition and String Concatenation	<code>+</code>
Subtraction	<code>-</code>
Multiplication	<code>*</code>
Division	<code>/</code>

variables is fine, but adding three variables would not be parsed correctly. In addition, Music++ can only parse four different operations, which are listed in Table A.2.

Music++ can parse `if`, `else`, `switch`, `for`, `while`, and `do/while` statements and loops. A detailed list of which of these statements Music++ can parse and which subexpressions Music++ extracts from these statements is shown in Table A.3.

Music++ will only parse conditional and loop statements nested one level deep. For example, Music++ can parse an `if` statement, but it will not correctly parse an `if` statement nested inside of a `for` statement.

Even though these limitations are quite restrictive, they are still generous enough to be helpful in an introductory programming class. In addition, these limitations made it possible to create a well-designed sonification early in my research.

Table A.3: List of the multiline statements Music++ can parse and the pieces of code related to those statements that Music++ sonifies.

Statement	Related Pieces of Code	Description
switch	Switch variable	i.e., <code>x</code> in the statement <code>switch(x)</code>
	Switch body	The switch statement's case statements and the case statements' bodies.
case	Case condition	The value to which the switch variable is compared.
	Case body	The lines of code that will be executed if the case condition is equal to the switch variable.
default	Default body	The lines of code that will be executed if all of the corresponding case conditions are not equal to the switch variable.
do/while	Do/while condition	The boolean expression that is evaluated at the end of each loop iteration to determine whether to stay in the loop.
	Do/while body	The body of the loop.
if	If condition	The boolean expression that determines whether to enter the if body.
	If body	The lines of code that are executed if the if condition is true.
else if	Else if condition	The boolean expression that determines whether to enter the else if body.
	Else if body	The lines of code that will be executed if the else if condition is true.
else	Else body	The lines of code that are executed if all of the related if and else if conditions are false.
for	For variable expression	The portion of the for loop that defines the counter variables.
	For condition	The boolean expression that is evaluated at the start of each loop iteration to determine whether to stay in the loop.
	For incrementation	The portion of the for loop that is executed at the end of each iteration to increment the loop variables.
	For body	The body of the loop.
while	While condition	The boolean expression that is evaluated at the start of each loop iteration to determine whether to stay in the loop.
	While body	The body of the loop.

A.1.2 Sonification Design

Music++ uses an earcon-based sonification. Each program element listed in Table A.1, Table A.2, and Table A.3 is assigned a different earcon. When sonifying the execution path, as each line of code in the program is encountered, all of the earcons corresponding to the elements in the line of code play simultaneously. For example, a line of code that declares a `String` variable would play the `String` earcon, but no other sounds. A line of code that added two integers, then assigned the result to an `int` variable would play the `int` earcon and the addition earcon simultaneously.

When the execution path goes inside the body of a conditional or loop statement, the sonification continuously plays an earcon to indicate that the execution path is inside of the body. For example, to sonify a line of code that adds two integers, assigns the result to a variable, and occurs inside of an if statement, the sonification would play the earcons for addition, the `int` type, and the if statement body simultaneously.

In addition to sonifying each line of code, Music++ also sonifies the conditions in conditional and loop statements. For example, when sonifying an if statement, Music++ will first sonify the statement's condition. Then, if the statement evaluated to true, the sonification will sonify each of the elements in the if statement body. Table A.3 shows precisely which substatements will be sonified for each of the conditional and loop statements.

Each earcon is a measure long because untrained listeners can intuitively tell when a measure begins and ends [23]. All of the earcons were written in the same key so that when they are combined with each other, they still sound musically pleasing. The earcons are written using a wide variety of instruments, pitches, and rhythms in order to make them easier to tell apart. However, earcons related to the same conditional or loop statement are written using the same instrument and similar

itches. For example, both the while condition and while body earcons are written using a trombone. This makes it easy to identify which earcons are related to the same while loop when multiple earcons play at the same time inside of the while loop's body.

In addition, to make them more distinct, the arithmetic operator earcons are all written using drum sounds instead of pitched instruments. Each of the four arithmetic earcons are written with a different drum that plays one loud, easily noticeable note on the same rhythm. As a result, it is easy to identify an arithmetic operator earcon.

A.2 Implementation

Music++ receives the end user's Java code as input. This code is sent to Cobertura [46], a code coverage tool for Java, which finds the execution path of the program. A hand-written Java parser then uses both the original source code and the coverage report produced by Cobertura to discover the program execution path. The execution path is then given to a music generator, which produces the final sonification as a MIDI file. Finally, the sonification MIDI file is played to the user or saved.

A.2.1 Cobertura

Music++ has a `CoberCaller` class that interacts with Cobertura. First, the `CoberCaller` class compiles the input program using version 1.6 of Java. (This is because the version of Cobertura that was used to create Music++ isn't compatible with Java 1.7 or 1.8.)

Next, the compiled class file is instrumented for Cobertura. Then, the program calls Cobertura on the instrumented file. Cobertura runs the program once and

collects code coverage information. The `CoverCaller` class then calls Cobertura's report system to produce a coverage report.

A.2.2 Parser

Music++ has a hand-written Java parser that can parse the subset of Java specified in Section A.1.1 above. The parser uses the original input source file and the coverage report produced by Cobertura as inputs and uses these to create a data structure representing the input program's execution path.

The data structure is a tree of "node" objects, where each node represents a line of code in the program. Performing a pre-order traversal of the tree will recover the original execution path. Leaves are usually "data structure" nodes, which represent lines of code that declare or assign values to variables. Some leaf nodes and all non-leaf nodes are "scope" nodes, which represent lines of code like if statements and for loops that always have a body of one or more statements that are conditionally executed or looped. If a scope node corresponds to a loop, the scope node is also marked with the number of times that the loop was executed.

Data structure nodes correspond to the data types in Table A.1, and scope nodes always correspond to the statements in Table A.3. If an arithmetic operation is performed on any particular node, the node also records that the arithmetic operation was performed on the corresponding line of code. The arithmetic operations that can be recorded are listed in Table A.2.

The source file is parsed line by line and a corresponding node is created for each line. Lines of code that assign variables are parsed as data structure nodes, and the type assigned to each node is the type of the variable on the left-hand side of the

assignment. Lines of code corresponding to loops and conditional statements are parsed as scope nodes.

A.2.3 Music Generator

The music generator was implemented using two different sub-components. The `MidiUtilities` class uses the built-in Java MIDI library [47] to combine MIDI files together. The `MusicGenerator` class takes the abstract representation that Cobertura and the parser created and combines the correct earcons to create the final MIDI project.

The `MidiUtilities` class includes functionality for opening MIDI files, copying MIDI data from one object to another, and saving MIDI files.

The `MusicGenerator` walks the execution path tree in pre-order traversal. At each node, including non-leaf nodes, it generates a measure of music for the line of code corresponding to the node. In addition, by walking the execution path as a tree, the `MusicGenerator` keeps track of each node's ancestors in the tree, allowing it to create layered music for code inside of loops and conditional statements by playing the loops and conditionals' body earcons at the same time as playing the earcons for the lines of code inside the bodies.

After the `MusicGenerator` creates a MIDI song to represent the program, it can save it as a file or play it immediately.

A.3 Related Work

Several other researchers have created tools that sonify the execution paths of programs. Unfortunately, this means that Music++ is not a completely novel tool.

A.3.1 Siren Songs and Swan Songs

In their paper *Siren Songs and Swan Songs: Debugging with Music*, Vickers and Alty created a tool that sonifies Pascal programs. Their technique, much like that of Music++, sonifies when IF and WHILE statements have their conditions evaluated and their bodies executed. They play earcons written in major keys, which sound generally happy, when a condition evaluates to true and earcons in minor keys, which generally sound sad, when a condition evaluates to false. In addition, they play low-pitched, drawn-out droning notes to indicate when the sonified programs are inside of loop and conditional bodies. The drones start when entering a loop or conditional body, become layered with additional droning notes when entering nested loops and conditionals, and stop playing when the bodies are exited [34].

A.3.2 Increasing Fault Detection Effectiveness Using Layered Program Auralization

Stefik, Fitz, and Alexander created a layered sonification of program execution in their papers *Increasing Fault Detection Effectiveness Using Layered Program Auralization* and *Layered Program Auralization: Using Music to Increase Runtime Program Comprehension and Debugging Effectiveness*. Each layer portrays a different aspect of the executed program and controls a different aspect of the generated sonification. For example, to indicate when loop and conditional statements are entered and exited, a cadential or control flow layer changes the chord progression that the sonification uses. The chord progressions are based on cadences, or well-known chord progressions that have a strong sense of finality when they are finished. The cadences start when entering a loop or conditional statement body and resolve, reaching their sense of

finality, when the loop or conditional bodies exit. For nested loop and conditional statements, the sonification changes key.

Another layer, the orchestration or program state layer, changes the number of instruments in the sonification based on how much memory the program is using. In their paper, Stefik, Fitz, and Alexander gave the example of keeping track of the number of elements in an important linked list. As elements are added, more instruments join the sonification, and as elements are removed from the list, instruments drop out. A third layer, the lyrics or semantic data layer, produces lyrics for singing pieces of data or control flow statements of interest [31, 33].

In *Layered Program Auralization: Using Music to Increase Runtime Program Comprehension and Debugging Effectiveness*, Stefik, Fitz, and Alexander conducted a user study that showed the control flow and state layers were helpful, but the semantic layer's lyrics were confusing [31].

A.4 Future Work

Although I did not work on this addition, my classmate Shane Bolan has modified Music++ so that when the sonification plays, the source code being sonified is shown on the computer screen. As each line of code is sonified, it is highlighted on the screen, making it even easier to understand the sonification and the program's execution path.

In the future, the handwritten parser that Music++ uses will be replaced with a robust third-party parser. This will allow the limitations on the programs that Music++ can parse to be lifted. At the same time, the sonification will be extended so that more Java programs can be sonified. Ultimately, the goal is to sonify all possible Java programs.

A.5 Conclusions

With Music++, we created a technique that uses earcon sonification to produce a song that represents the execution path of a simple Java program. Music++ introduced several ideas that I use in the rest of my research projects. I consistently use earcon sonification to represent pieces of data as well the idea of using a measure of music for each earcon to separate earcons. In addition, I continue use the idea of layering earcons to provide multiple pieces of data simultaneously.

While Music++ isn't a novel tool, creating this tool set the course for the remainder of my research.

Appendix B

GitSonifier Experiment Materials

Figures B.1 through B.15 show the materials used in the GitSonifier experiment. The GitSonifier experiment is discussed in Chapter 3.

Thank you for volunteering to participate in our research experiment.

As part of the experiment, you will complete three tasks:

1. You will learn to use a new tool we've developed, the Git Sonifier. The tool portrays data from an open-source's version control history using sound.
2. You will use the Git Sonifier to answer some questions.
3. Finally, we will have you fill out a questionnaire asking what you thought of the Git Sonifier, then verbally ask you some questions about it.

When you are ready, you can [start learning about the Git Sonifier](#).

Figure B.1: The first page shown to participants in the GitVS experiment.

Examples

Developer Sounds

In our tool, each developer in a software project is represented by a different sound.

Sarah is represented by a **harp**:



Devin is represented by a **choir**:



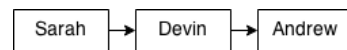
Andrew is represented by a **trombone**:



Representing Commits

These sounds are combined into songs that represent programming projects' version control histories. When the sounds are combined, each measure of music represents a single version control commit.

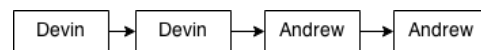
For example, listen to this song:



As you listen to it, you will hear three measures of music:

1. The first measure of music is **Sarah's harp**, so Sarah made the **first commit** in the project.
2. **Devin's choir** represents the **second commit**.
3. **Andrew's trombone** represents the **third commit**.

Here is another example:



1. **Devin's choir** plays twice, indicating that she made the **first two commits**.
2. Then, **Andrew's trombone** plays twice, so he made the **next two commits**.

Figure B.2: Page 1 of training page in the GitSonifier experiment.

Day Separators

Multiple commits can be made in a single day. **Day separators** indicate when one day ends and another begins.

A day separator sounds like this:



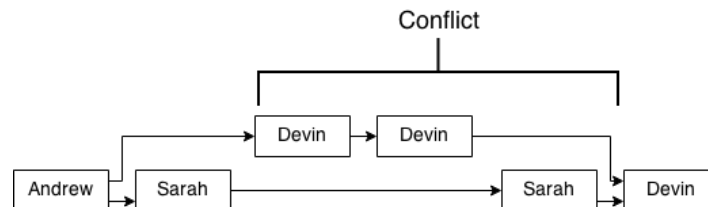
For example:



- **Andrew** makes **two commits**.
- The **day separator** indicates that the day is over.
- Next, **Andrew** makes a third commit on the **next day**.

Conflicts

Conflict drums indicate when **version control conflicts** are introduced and resolved. For example:

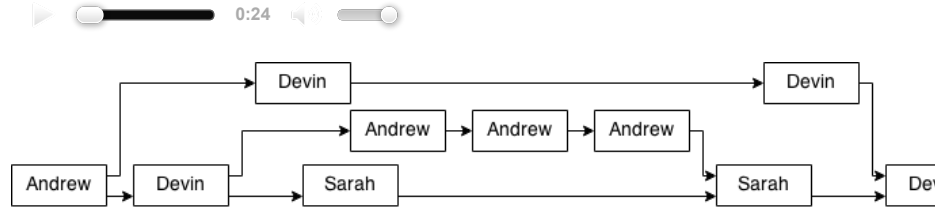


- **Andrew and Sarah** make commits.
- **Devin** makes a third commit that **introduces a conflict**, as indicated by the drums.
- **Devin and Sarah** make the next two commits. Neither resolves the conflict.
- Devin makes one more commit. The conflict drums stop on this commit, indicating that **the conflict was resolved**.

Note that when the drums start playing, the commit that resolved the conflict is the one **after** the drums stop.

In some cases, a new conflict will be introduced before an old one is resolved. In this case, the conflict drums will become **louder**. For example:

Figure B.3: Page 2 of training page in the GitSonifier experiment.



- **Andrew and Devin** make commits.
- **Devin** makes a second commit that **introduces a conflict**.
- Sarah makes a commit.
- **Andrew** makes another commit. This one **introduces a second conflict**, so the conflict drums become **louder**.
- Andrew makes two more commits.
- **Sarah** makes a commit that **resolves a conflict**. There is one unresolved conflict left, so **the conflict drums become quieter**.
- **Devin** makes two more commits. The second one **resolves** the remaining conflict.

Real-World Example

The following sonification represents some real-world data obtained from sonifying part of an open-source project's version control history:



To make sure you understand the training, please answer the following questions:

Which developer made the first commit in this sonification?

- Sarah
- Devin
- Andrew

How many conflicts are there in this sonification?

- 1
- 2
- 3 or more
- There are no conflicts in the sonification.

If there is at least one conflict in the sonification, who introduced the first conflict?

- Sarah
- Devin
- Andrew
- There are no conflicts in the sonification.

If there is at least one conflict in the sonification, who resolved the final conflict?

- Sarah
- Devin

Figure B.4: Page 3 of training page in the GitSonifier experiment.

- Andrew
- There are no conflicts in the sonification.

How many days altogether are portrayed in this sonification, counting the first and last days?

- 1
- 2
- 3 or more

Finish

Figure B.5: Page 4 of training page in the GitSonifier experiment.

Instructions

We will now show you ten songs. They are all similar to the real-world song you were asked questions about during the training. Some of them have been changed slightly, changing the meaning of the data they portray. Some (or perhaps none) of them are identical to the original sonification.

We will ask you the following questions about each song:

How many developers were there in this song compared to the original?

1. There were **more** developers.
2. There were **fewer** developers.
3. There were **the same number** of developers, but they were **different developers**.
4. There were the **exact same** developers.

How many days passed in this song compared to the original?

1. **More** days passed.
2. **Fewer** days passed.
3. **The same number** of days passed.

How many conflicts were there in this song compared to the original?

1. There were **more** conflicts.
2. There were **fewer** conflicts.
3. There were **the same number** of conflicts.

In addition to showing you a new song and questions to answer, each page will also include the original song and all of the sounds covered in the training for your reference.

When you are ready, [click here to begin the first question](#).

Figure B.6: The instructions in the GitSonifier experiment.

Question 1

Listen to the following song:



Please answer these questions:

How many developers were there in this song compared to the original?

1. There were **more** developers.
2. There were **fewer** developers.
3. There were **the same number** of developers, but they were **different developers**.
4. There were the **exact same** developers.

How many days passed in this song compared to the original?

1. **More** days passed.
2. **Fewer** days passed.
3. **The same number** of days passed.

How many conflicts were there in this song compared to the original?

1. There were **more** conflicts.
2. There were **fewer** conflicts.
3. There were **the same number** of conflicts.

Reference

For your reference, here are the original song and important sounds.

Original Song



Developers' Sounds



- Sarah
 
- Devin
 

Figure B.7: Page 1 of the questions page in the GitSonifier experiment.

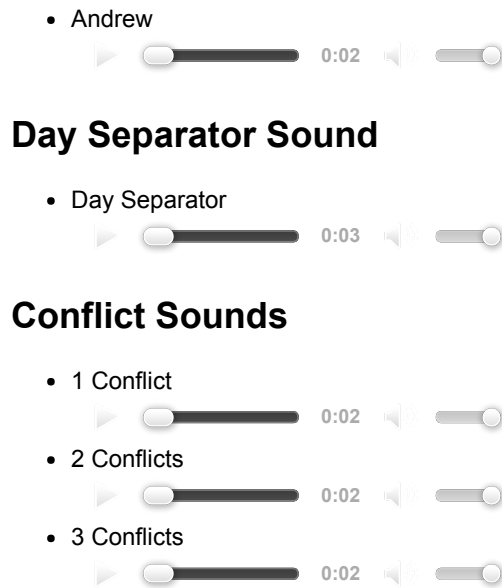


Figure B.8: Page 2 of the questions page in the GitSonifier experiment.

Finished

You are now finished learning about and working with the Git Sonifier tool. We will now ask you to complete a questionnaire on paper, then verbally ask you some questions. When those are finished, the experiment will be complete.

Figure B.9: The page shown when the GitSonifier main task was completed.

Results for Questions

Overall Score: 6 out of 30

• Question 1



- How many developers were there in this song compared to the original?
 - **Incorrect**
 - Your answer: There were **more** developers.
 - Correct answer: There were **fewer** developers.
- How many days passed in this song compared to the original?
 - **Incorrect**
 - Your answer: **More** days passed.
 - Correct answer: **The same number** of days passed.
- How many conflicts were there in this song compared to the original?
 - **Incorrect**
 - Your answer: There were **more** conflicts.
 - Correct answer: There were **the same number** of conflicts.

• Question 2



- How many developers were there in this song compared to the original?
 - **Incorrect**
 - Your answer: There were **more** developers.
 - Correct answer: There were **fewer** developers.
- How many days passed in this song compared to the original?
 - **Incorrect**
 - Your answer: **More** days passed.
 - Correct answer: **The same number** of days passed.
- How many conflicts were there in this song compared to the original?
 - **Incorrect**
 - Your answer: There were **more** conflicts.
 - Correct answer: There were **the same number** of conflicts.

• Question 3



- How many developers were there in this song compared to the original?
 - **Incorrect**
 - Your answer: There were **more** developers.
 - Correct answer: There were the **exact same** developers.
- How many days passed in this song compared to the original?
 - **Correct**

Figure B.10: Page 1 of the page that participants could view if they wanted to see which questions they answered correctly in GitVS.

- Your answer: **More** days passed.
 - Correct answer: **More** days passed.
 - How many conflicts were there in this song compared to the original?
 - **Incorrect**
 - Your answer: There were **more** conflicts.
 - Correct answer: There were **the same number** of conflicts.

• Question 4



- How many developers were there in this song compared to the original?
 - **Incorrect**
 - Your answer: There were **more** developers.
 - Correct answer: There were the **exact same** developers.
- How many days passed in this song compared to the original?
 - **Incorrect**
 - Your answer: **More** days passed.
 - Correct answer: **The same number** of days passed.
- How many conflicts were there in this song compared to the original?
 - **Correct**
 - Your answer: There were **more** conflicts.
 - Correct answer: There were **more** conflicts.

• Question 5



- How many developers were there in this song compared to the original?
 - **Incorrect**
 - Your answer: There were **more** developers.
 - Correct answer: There were the **exact same** developers.
- How many days passed in this song compared to the original?
 - **Incorrect**
 - Your answer: **More** days passed.
 - Correct answer: **The same number** of days passed.
- How many conflicts were there in this song compared to the original?
 - **Incorrect**
 - Your answer: There were **more** conflicts.
 - Correct answer: There were **fewer** conflicts.

• Question 6



- How many developers were there in this song compared to the original?
 - **Incorrect**
 - Your answer: There were **more** developers.
 - Correct answer: There were **the same number** of developers, but they were **different developers**.

Figure B.11: Page 2 of the page that participants could view if they wanted to see which questions they answered correctly in GitVS.

- How many days passed in this song compared to the original?
 - **Incorrect**
 - Your answer: **More** days passed.
 - Correct answer: **The same number** of days passed.
- How many conflicts were there in this song compared to the original?
 - **Correct**
 - Your answer: There were **more** conflicts.
 - Correct answer: There were **more** conflicts.

• Question 7



- How many developers were there in this song compared to the original?
 - **Incorrect**
 - Your answer: There were **more** developers.
 - Correct answer: There were the **exact same** developers.
- How many days passed in this song compared to the original?
 - **Incorrect**
 - Your answer: **More** days passed.
 - Correct answer: **The same number** of days passed.
- How many conflicts were there in this song compared to the original?
 - **Incorrect**
 - Your answer: There were **more** conflicts.
 - Correct answer: There were **the same number** of conflicts.

• Question 8



- How many developers were there in this song compared to the original?
 - **Incorrect**
 - Your answer: There were **more** developers.
 - Correct answer: There were **the same number** of developers, but they were **different developers**.
- How many days passed in this song compared to the original?
 - **Incorrect**
 - Your answer: **More** days passed.
 - Correct answer: **Fewer** days passed.
- How many conflicts were there in this song compared to the original?
 - **Incorrect**
 - Your answer: There were **more** conflicts.
 - Correct answer: There were **fewer** conflicts.

• Question 9



- How many developers were there in this song compared to the original?
 - **Incorrect**

Figure B.12: Page 3 of the page that participants could view if they wanted to see which questions they answered correctly in GitVS.

- Your answer: There were **more** developers.
 - Correct answer: There were the **exact same** developers.
 - How many days passed in this song compared to the original?
 - **Incorrect**
 - Your answer: **More** days passed.
 - Correct answer: **Fewer** days passed.
 - How many conflicts were there in this song compared to the original?
 - **Correct**
 - Your answer: There were **more** conflicts.
 - Correct answer: There were **more** conflicts.

• Question 10



- How many developers were there in this song compared to the original?
 - **Correct**
 - Your answer: There were **more** developers.
 - Correct answer: There were **more** developers.
- How many days passed in this song compared to the original?
 - **Incorrect**
 - Your answer: **More** days passed.
 - Correct answer: **Fewer** days passed.
- How many conflicts were there in this song compared to the original?
 - **Correct**
 - Your answer: There were **more** conflicts.
 - Correct answer: There were **more** conflicts.

Reference

For your reference, here are the original song and important sounds.

Original Song



Developers' Sounds




- Sarah
 - 
- Devin
 - 
- Andrew
 - 

Figure B.13: Page 4 of the page that participants could view if they wanted to see which questions they answered correctly in GitVS.

- Additional Developer (Flute)
 0:02
- Additional Developer (Oboe)
 0:02

Day Separator Sound

- Day Separator
 0:03

Conflict Sounds

- 1 Conflict
 0:02
- 2 Conflicts
 0:02
- 3 Conflicts
 0:02

Figure B.14: Page 5 of the page that participants could view if they wanted to see which questions they answered correctly in GitVS.

Questionnaire

Please answer the following questions regarding the Git Sonifier:

	Never	Rarely	Sometimes	Frequently	Always
It was easy to tell the different sounds apart.					
It was easy to hear who each developer was.					
It was easy to hear how many conflicts there were.					
It was easy to hear when days passed.					
The sound helped me understand the development data.					

	Strongly Disagree	Disagree	Neither Agree nor Disagree	Agree	Strongly Agree
I would be interested in hearing the development data of my own teams' projects.					

Figure B.15: The questionnaire given to participants at the end of the GitSonifier study.

Appendix C

GitVS Experiment Materials

Figures C.1 through C.27 show the materials used in the GitVS experiment. The GitVS experiment is discussed in Chapter 4.

The materials shown are specific to the participants who used GitVS. Participants who used GitVS without sound and the GitHub network view saw very similar pages with some small differences. For example, only participants who used GitVS with sound were shown the developer and conflict earcons on the task instructions and training pages.

Thank you for volunteering to participate in our research experiment.

In this experiment, you will take on the role of the manager of a software development team. Your team currently uses Git with the following formal processes for coordinating version control:

- Whenever a new feature is started, the developers can choose to create a new branch for that feature, but do not have to.
- Before merging a completed feature, branches will be code-reviewed.

You are considering switching to the following process:

- Whenever a new feature is started, the developers must create a new branch for that feature.
- Before merging a completed feature, branches will be code-reviewed.
- All commits will be sent to a continuous integration sever, which will automatically build and test each commit.

You want to understand how these changes will affect your team, so you have found another project that has a similar amount of activity to yours, but follows all of the processes you want to use.

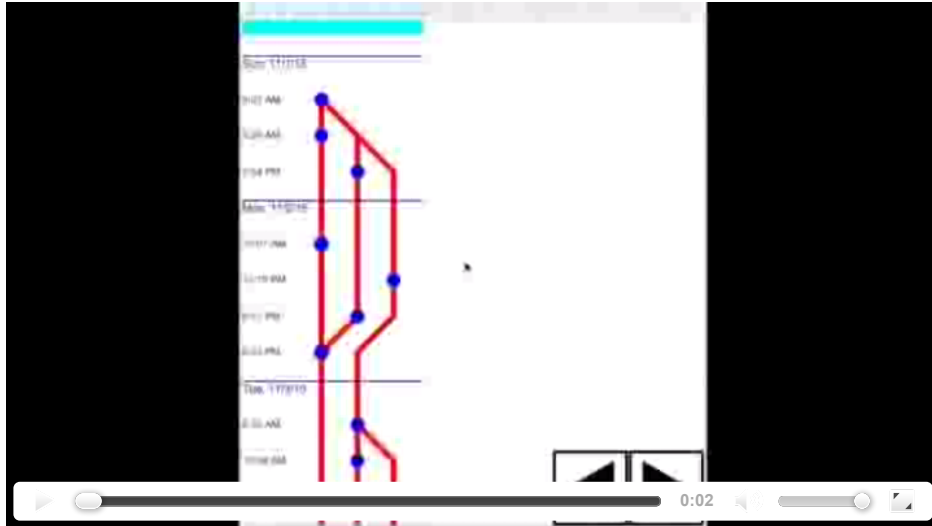
In the experiment, you will be shown the version control history for both your current project and the project that already follows the new processes. You will analyze these projects using GitVS, a tool for exploring Git history.

When you are ready to begin, you will [learn how to use GitVS](#).

Figure C.1: The first page shown to participants in the GitVS experiment.

Watch the following video and practice using GitVS, and then complete the training quiz below.

Feel free to use GitVS and ask questions.



After watching the video, please complete these questions.

How many developers contributed to this project?

- 2
- 3
- 4
- 5

How many conflicts are present in the project?

- 2
- 3
- 4
- 5

Who introduced the second conflict?

- Andrew
- Alex
- Devin
- Sarah

Figure C.2: Page 1 of training page in the GitSonifier experiment.

How long, in days, is the longest period of time during which there's always at least one conflict?

- 1 day
- 2 days
- 3 days
- 4 days

How many commits modify exactly two files?

- 5
- 6
- 7
- 8

How many branches are there in this project?

- 5
- 6
- 7
- 8

How many commits are made by outside developers?

- 0
- 1
- 2
- 3

Finish

Reference

Conflict Sounds

- 1 Conflict
 0:02
- 2 Conflicts
 0:02
- 3 Conflicts
 0:02
- 4+ Conflicts

Figure C.3: Page 2 of training page in the GitSonifier experiment.



Core Developers

The following developers are active developers on the training project:

- Sarah



- Andrew



- Devin



The following developer is an outside developer:

- Alex



Figure C.4: Page 3 of training page in the GitSonifier experiment.

Instructions

You will now be asked questions about the repository that uses the processes you are considering adopting.

The team that maintains this repository follows several methodologies that you are considering adopting in your own team. You want to see how it is different from your team so you can understand what changes you can expect.

The methodologies this team follows are:

- Whenever a new feature is started, the developers must create a new branch for that feature.
- Before merging a completed feature, branches will be code-reviewed.
- All commits will be sent to a continuous integration sever, which will automatically build and test each commit.

Some of the questions may take a long time to complete. Please focus on each question for at least 5 minutes. If you take longer than 5 minutes, you can skip to the next question (or continue working on the question if you prefer).

[Begin Questions](#)

As you work on the questions, the following information will be displayed on each question's page:

Reference

Show/Hide Conflict Sounds

Conflict Sounds









- 1 Conflict
▶  0:02 
- 2 Conflicts
▶  0:02 
- 3 Conflicts
▶  0:02 
- 4+ Conflicts
▶  0:02 

Figure C.5: Page 1 of the task instructions in the GitVS experiment.











Show/Hide List of Core and Outside Developers

Core Developers

The following developers are active developers on the repository that uses the processes you are considering adopting:

- Nathan Marz

- P. Taylor Goetz

- Jason Jackson

- Robert (Bobby) Evans


The following developers are outside developers:

- James Xu

- Derek Dagit

- wurstmeister

- Michael G. Noll

- afeng

- anfang

- Sean Zhong

- John Gilmore

- Ben Hughes

- Everyone else


Show All Outside Developers

Figure C.6: Page 2 of the task instructions in the GitVS experiment.

- ankitoshniwal
- Sriharsha Chintalapani
- Sergey Lukjanov
- Kang Xiao
- Thomas Jack
- darthbear
- Kyle Nusbaum
- Sam Ritchie
- Homer Strong
- Martin Kleppmann
- Sjoerd Mulder
- Danijel Schiavuzzi
- Argyris Zymnis
- Evan Chan
- haitao.yao
- Prabeesh K
- ChitturiPadma
- supercargo
- David Lao
- Muneyuki Noguchi
- David James
- Philip (flip) Kromer
- Jake Donham
- Michael Cetrulo
- nathanmarz
- Niels Basjes
- Mike Blume
- Gabriel Silk
- Brenden Matthews
- roadkill001
- Suresh Srinivas
- troyding
- Soren Macbeth
- Stuart Anderson
- smelody@wgen.net
- Ross Feinstein
- Dan Harvey
- yerenkow
- Vinod Chandru
- Yu L Li
- Bryan Baugher
- Dan Dillinger
- Lorcan Coyle
- Andrew Otto
- Trevor Wennblom
- Guanpeng Xu

Figure C.7: Page 3 of the task instructions in the GitVS experiment.

- Andrew Olson
- Ryan
- Michael Allman
- thomas
- Kyle Bolton
- Bryan
- Bryan Peterson
- Tudor Scurtu
- okapies
- minghan
- danehammer
- thinker0
- Steven Phung
- Debo~ Dutta
- Brian O'Neill
- tombrown52
- Srinivas Prasad Gumdelli
- Bart Olsthoorn
- Boris
- Jean Vancoppenolle
- davidlgr
- Edison Xu
- ashley
- Jeroen van Dijk
- xumingming
- Fabian Neumann
- dmmata
- David Losada
- engineerdev
- Paul O'Fallon
- Kasper Madsen
- millerjam
- Travis Wellman
- Gabriel Grant
- Drew
- Haitao Yao
- Srinivas Gumdelli
- Alexey S. Kachayev
- Adrian Petrescu
- aprooks
- dennis zhuang
- Trevor Summers Smith
- Patrick Houk
- ptgoetz
- Taylor Goetz
- Bertrand Dechoux

Figure C.8: Page 4 of the task instructions in the GitVS experiment.

- o Tom Payne
- o Sam Stokes

Figure C.9: Page 5 of the task instructions in the GitVS experiment.

Please answer this question about **the repository that uses the processes you are considering adopting**:

1. How many **commits** were made in-between June 5 and June 7?

Reference

Show/Hide Conflict Sounds

Show/Hide List of Core and Outside Developers

Figure C.10: Question 1 in the GitVS experiment.

Please answer this question about **the repository that uses the processes you are considering adopting**:

1. On average, what is the number of **files** modified per **commit** in-between June 5 and June 7?

Commit	Files	
1	<input type="text"/>	Delete Row

Add Row

Calculate Average

Final average:

Reference

Show/Hide Conflict Sounds

Show/Hide List of Core and Outside Developers

Figure C.11: Question 2 in the GitVS experiment.

Please answer this question about **the repository that uses the processes you are considering adopting**:

1. How many **conflicts** were present in-between May 29 and June 12?

Reference

Show/Hide Conflict Sounds

Show/Hide List of Core and Outside Developers

Figure C.12: Question 3 in the GitVS experiment.

Please answer this question about **the repository that uses the processes you are considering adopting**:

1. Look at the **branches** which were merged directly into the master branch. In **days**, what is the average number of days these branches existed?

Branch	Days	
1	<input type="text"/>	Delete Row

Add Row

Calculate Average

Final average:

Reference

Show/Hide Conflict Sounds

Show/Hide List of Core and Outside Developers

Figure C.13: Question 4 in the GitVS experiment.

Please answer this question about **the repository that uses the processes you are considering adopting**:

1. What is the average number of **days** each **branch** with at least one **commit from an outsider developer** existed before being merged?

Branch	Days	
1	<input type="text"/>	Delete Row

Final average:

Reference

Figure C.14: Question 5 in the GitVS experiment.

Instructions

Next, you will be asked questions about your team's repository.

You want to determine how adopting new, stricter methodologies for version control will change your team's performance. Currently, your team uses these methodologies:

- Whenever a new feature is started, the developers can choose to create a new branch for that feature, but do not have to.
- Before merging a completed feature, branches will be code-reviewed.

[Begin Questions](#)









As a reminder, please focus on each question for at least 5 minutes. If you take longer than that, you can choose whether skip to the next question or continue working.

As you work on the questions, the following information will be displayed on each question's page:

Reference

Show/Hide Conflict Sounds

Conflict Sounds

- 1 Conflict
▶  0:02 🔊 
- 2 Conflicts
▶  0:02 🔊 
- 3 Conflicts
▶  0:02 🔊 
- 4+ Conflicts
▶  0:02 🔊 

Show/Hide List of Core and Outside Developers

Core Developers

The following developers are active developers on your team's repository:

- Alex Feinberg

Figure C.15: Page 1 of the instructions shown when switching from one repository to another while answering the objective questions in the GitVS experiment.

- ▶ 0:02
- Roshan Sumbaly
- ▶ 0:02
- Jay Kreps
- ▶ 0:02
- Bhupesh Bansal
- ▶ 0:02

The following developers are outside developers:

- Ismael Juma
- ▶ 0:02
- kirktrue
- ▶ 0:02
- Kirk True
- ▶ 0:02
- Rob Adams
- ▶ 0:02
- bbansal
- ▶ 0:02
- Lei Gao
- ▶ 0:02
- Elias Torres
- ▶ 0:02
- Siddharth Singh
- ▶ 0:02
- Geir Magnusson
- ▶ 0:02
- Everyone else
- ▶ 0:02

Show All Outside Developers

- Geir Magnusson Jr
- Jonathan Traupman
- Jakob Homan
- Janne Hietamäki
- Alejandro Crosa
- Aleksandr Feinberg
- Paul Lindner

Figure C.16: Page 2 of the instructions shown when switching from one repository to another while answering the objective questions in the GitVS experiment.

- Michael R. Head
- unknown
- claudio
- Chris Riccomini
- Padraig
- Mike Frost
- Anthony Lauzon
- Scott Wheeler
- jtuberville
- Sergey Shepelev
- Antoine Toulme
- Ben Hardy
- Barak A. Pearlmutter
- Shannon Zhang

Figure C.17: Page 3 of the instructions shown when switching from one repository to another while answering the objective questions in the GitVS experiment.

Comparison Questions

We will now ask you some questions that ask you to compare your team's repository to the one that uses the processes you're considering. These questions do not have right or wrong answers. Rather, we want to see what your opinion is.

Each question has a drop-down list and an open-ended text box. Please both select an item from each drop-down list and write a short explanation for your choice in each text box.

Your answers to the previous 10 questions will be displayed at the bottom of the screen for your reference.

[Begin questions](#)

Figure C.18: The instructions shown before participants saw the subjective questions in the GitVS experiment.

Please answer these questions:

Based on the previous questions, which process would you recommend if your priority is to **have a large number of small commits**?

Select a repository 

Why did you select that repository?

Based on the previous questions, which process would you recommend if your priority is to **minimize conflicts**?

Select a repository 


Why did you select that repository?

Based on the previous questions, which process would you recommend if your priority is to **accept pull requests quickly**?

Select a repository 

Why did you select that repository?

Based on the previous questions, which process would you recommend if your priority is to **quickly merge commits from outside developers**?

Select a repository 

Why did you select that repository?

Here are your answers to the previous questions.

Data Set	Question	Your Answer
Your Repository	How many commits were made in-between June 10 and June 12?	(participant's answer)
Your Repository	On average, what is the number of files modified per commit in-between June 10 and June 12?	(participant's answer)
	How many conflicts were present in-between June 10	(participant's

Figure C.19: Page 1 of the subjective questions in the GitVS experiment.

Your Repository	and June 24?	(answer)
Your Repository	Look at the branches which were merged directly into the master branch. In days , what is the average number of days these branches existed?	(participant's answer)
Your Repository	What is the average number of days each branch with at least one commit from an outsider developer existed before being merged?	(participant's answer)
Comparison Repository	How many commits were made in-between June 5 and June 7?	(participant's answer)
Comparison Repository	On average, what is the number of files modified per commit in-between June 5 and June 7?	(participant's answer)
Comparison Repository	How many conflicts were present in-between May 29 and June 12?	(participant's answer)
Comparison Repository	Look at the branches which were merged directly into the master branch. In days , what is the average number of days these branches existed?	(participant's answer)
Comparison Repository	What is the average number of days each branch with at least one commit from an outsider developer existed before being merged?	(participant's answer)

Figure C.20: Page 2 of the GitVS experiment.

Thank you for answering these questions.

We are finished using GitVS. The final steps in the experiment are:

1. A questionnaire about what you thought of using GitVS
2. A verbal exit interview about GitVS
3. If you wish, you can see your results on the questions asked during the experiment

Figure C.21: The page shown when the GitVS main task was completed.

Your Answers

Questions about **your team's repository**:

1. How many **commits** were made in-between June 10 and June 12?
 - Your answer: **(participant's answer)**
2. On average, what is the number of **files** modified per **commit** in-between June 10 and June 12?
 - Your answer: **(participant's answer)**
3. How many **conflicts** were present in-between June 10 and June 24?
 - Your answer: **(participant's answer)**
4. Look at the **branches** which were merged directly into the master branch. In **days**, what is the average number of days these branches existed?
 - Your answer: **(participant's answer)**
5. What is the average number of **days** each **branch** with at least one **commit from an outsider developer** existed before being merged?
 - Your answer: **(participant's answer)**

Questions about **the repository that uses the processes you are considering adopting**:

1. How many **commits** were made in-between June 5 and June 7?
 - Your answer: **(participant's answer)**
2. On average, what is the number of **files** modified per **commit** in-between June 5 and June 7?
 - Your answer: **(participant's answer)**
3. How many **conflicts** were present in-between May 29 and June 12?
 - Your answer: **(participant's answer)**
4. Look at the **branches** which were merged directly into the master branch. In **days**, what is the average number of days these branches existed?
 - Your answer: **(participant's answer)**
5. What is the average number of **days** each **branch** with at least one **commit from an outsider developer** existed before being merged?
 - Your answer: **(participant's answer)**

Comparison questions:

1. Based on the previous questions, which process would you recommend if your priority is to **have a large number of small commits**?
 - Your answer:

Comparison Repository

(participant's answer)

Figure C.22: Page 1 of the page that participants could view to review their answers to questions during the exit interview in GitVS.

2. Based on the previous questions, which process would you recommend if your priority is to **minimize conflicts**?
 - Your answer:

Your Repository

(participant's answer)
3. Based on the previous questions, which process would you recommend if your priority is to **accept pull requests quickly**?
 - Your answer:

Comparison Repository

(participant's answer)
4. Based on the previous questions, which process would you recommend if your priority is to **quickly merge commits from outside developers**?
 - Your answer:

Your Repository

(participant's answer)

Figure C.23: Page 2 of the page that participants could view to review their answers to questions during the exit interview in GitVS.

Results for Questions

Overall Score: 0 out of 10

Questions about **your team's repository**:

1. How many **commits** were made in-between June 10 and June 12?
 - **Incorrect**
 - Your answer: **(participant's answer)**
 - Correct answer: **21**
2. On average, what is the number of **files** modified per **commit** in-between June 10 and June 12?
 - **Incorrect**
 - Your answer: **(participant's answer)**
 - Correct answer: **8**
3. How many **conflicts** were present in-between June 10 and June 24?
 - **Incorrect**
 - Your answer: **(participant's answer)**
 - Correct answer: **3**
4. Look at the **branches** which were merged directly into the master branch. In **days**, what is the average number of days these branches existed?
 - **Incorrect**
 - Your answer: **(participant's answer)**
 - Correct answer: **10**
5. What is the average number of **days** each **branch** with at least one **commit from an outsider developer** existed before being merged?
 - **Incorrect**
 - Your answer: **(participant's answer)**
 - Correct answer: **8**

Questions about **the repository that uses the processes you are considering adopting**:

1. How many **commits** were made in-between June 5 and June 7?
 - **Incorrect**
 - Your answer: **(participant's answer)**
 - Correct answer: **15**
2. On average, what is the number of **files** modified per **commit** in-between June 5 and June 7?
 - **Incorrect**
 - Your answer: **(participant's answer)**
 - Correct answer: **26**
3. How many **conflicts** were present in-between May 29 and June 12?
 - **Incorrect**

Figure C.24: Page 1 of the page that participants could view if they wanted to see which questions they answered correctly in GitVS.

- Your answer: **(participant's answer)**
 - Correct answer: **5**
4. Look at the **branches** which were merged directly into the master branch. In **days**, what is the average number of days these branches existed?
- **Incorrect**
 - Your answer: **(participant's answer)**
 - Correct answer: **6**
5. What is the average number of **days** each **branch** with at least one **commit from an outsider developer** existed before being merged?
- **Incorrect**
 - Your answer: **(participant's answer)**
 - Correct answer: **7**

Figure C.25: Page 2 of the page that participants could view if they wanted to see which questions they answered correctly in GitVS.

Questionnaire

Please answer the following questions regarding the tool that you used in the experiment:

	Never	Rarely	Sometimes	Frequently	Always
It was easy to tell when each commits was made.					
It was easy to identify who made each commit.					
It was easy to tell where conflicts were located.					
It was easy to tell when each commit was made.					
It was easy to see which files were changed by each commit.					
It was easy to tell where branches were located.					
It was easy to understand the details of the development data.					
Why did you select your answer?					
It easy to understand overall patterns in the development data.					
Why did you select your answer?					
It easy to see relationships between different pieces of development data.					
Why did you select your answer?					

Figure C.26: Page 1 of the questionnaire given to participants at the end of the GitVS study.

	Strongly Disagree	Disagree	Neither Agree nor Disagree	Agree	Strongly Agree
I would be interested in using this tool for my own team.					

Would you recommend this tool to others?

In your opinion, how can this tool be improved?

Figure C.27: Page 2 of the questionnaire given to participants at the end of the GitVS study.